Doug Bierer , Altaf Hussain
Branko Ajzele

PHP 7: Real World Application Development

Learning Path

Use new features of PHP 7 to solve practical, real-world problems faced by PHP developers like yourself every day





Packt

Table of Contents

PHP 7: Real World Application Development
PHP 7: Real World Application Development
Credits
Preface
- What this learning path covers
- What you need for this learning path
- Who this learning path is for
- Reader feedback
- Customer support
  - Downloading the example code
  - Errata
  - Piracy
  - Questions

1. Module 1
- 1. Building a Foundation
  - Introduction
  - PHP 7 installation considerations
    - How to do it...
      - Installing directly from source
      - Installing PHP 7 from pre-compiled binaries
      - Installing a *AMP package
    - There's more...
    - See also
  - Using the built-in PHP web server
    - How to do it...
  - Defining a test MySQL database
    - How to do it...
  - Installing PHPUnit
  - Implementing class autoloading
    - Getting ready
    - How to do it...
    - How it works...
  - Hoovering a website

How to do it...
How it works...
See also
Building a deep web scanner
How to do it...
How it works...
See also
Creating a PHP 5 to PHP 7 code converter
Getting ready
How to do it...
How it works...
See also
2. Using PHP 7 High Performance Features
Introduction
Understanding the abstract syntax tree
How to do it...
How it works...
See also
Understanding differences in parsing
How to do it...
How it works...
See also
Understanding differences in foreach() handling
How to do it...
How it works...
See also
Improving performance using PHP 7 enhancements
How to do it...
How it works...
There's more...
Iterating through a massive file
How to do it...
How it works...
Uploading a spreadsheet into a database
Getting ready...
How to do it...
How it works...

- Recursive directory iterator
  - How to do it...
  - How it works...
- 3. Working with PHP Functions
  - Introduction
  - Developing functions
    - How to do it...
    - How it works...
  - Hinting at data types
    - How to do it...
    - How it works...
    - See also
  - Using return value data typing
    - How to do it...
    - How it works...
    - There's more...
    - See also
  - Using iterators
    - How to do it...
    - How it works...
  - Writing your own iterator using generators
    - How to do it...
    - How it works...
- 4. Working with PHP Object-Oriented Programming
  - Introduction
  - Developing classes
    - How to do it...
    - How it works...
    - See also...
  - Extending classes
    - How to do it...
    - How it works...
  - Using static properties and methods
    - How to do it...
    - How it works...
    - See also
  - Using namespaces

- How to do it...
- How it works...
- Defining visibility
  - How to do it...
  - How it works...
  - See also
- Using interfaces
  - How to do it...
  - How it works...
- Using traits
  - How to do it...
  - How it works...
- Implementing anonymous classes
  - How to do it...
  - How it works...

5. Interacting with a Database
- Introduction
- Using PDO to connect to a database
  - How to do it...
  - How it works...
  - See also
- Building an OOP SQL query builder
  - How to do it...
  - How it works...
  - See also
- Handling pagination
  - How to do it...
  - How it works...
  - See also
- Defining entities to match database tables
  - How to do it...
  - How it works...
  - See also
- Tying entity classes to RDBMS queries
  - How to do it...
  - How it works...
  - There's more...

- Embedding secondary lookups into query results
  - How to do it...
  - How it works...
- Implementing jQuery DataTables PHP lookups
  - How to do it...
  - How it works...
  - There's more...

6. Building Scalable Websites
- Introduction
- Creating a generic form element generator
  - How to do it...
  - How it works...
- Creating an HTML radio element generator
  - How to do it...
  - How it works...
  - There's more...
- Creating an HTML select element generator
  - How to do it...
  - How it works...
- Implementing a form factory
  - How to do it...
  - How it works...
- Chaining $_POST filters
  - How to do it...
  - How it works...
  - There's more...
  - See also
- Chaining $_POST validators
  - How to do it...
  - How it works...
- Tying validation to a form
  - How to do it...
  - How it works...

7. Accessing Web Services
- Introduction
- Converting between PHP and XML
  - How to do it...

- How it works...
- Creating a simple REST client
  - How to do it...
    - Creating a streams-based REST client
    - Defining a cURL-based REST client
  - How it works...
  - There's more...
  - See also
- Creating a simple REST server
  - How to do it...
  - How it works...
  - There's more...
- Creating a simple SOAP client
  - How to do it...
  - How it works...
  - See also
- Creating a simple SOAP server
  - How to do it...
  - How it works...
  - See also

8. Working with Date/Time and International Aspects
- Introduction
- Using emoticons or emoji in a view script
  - How to do it...
  - How it works...
  - See also
- Converting complex characters
  - How to do it...
  - How it works...
- Getting the locale from browser data
  - How to do it...
  - How it works...
  - See also
- Formatting numbers by locale
  - How to do it...
  - How it works...
  - See also

- Handling currency by locale
  - How to do it...
  - How it works...
  - See also
- Formatting date/time by locale
  - How to do it...
  - How it works...
  - See also
- Creating an HTML international calendar generator
  - How to do it...
    - Refining internationalized output
  - How it works...
  - See also
- Building a recurring events generator
  - How to do it...
  - How it works...
  - See also
- Handling translation without gettext
  - How to do it...
  - How it works...
  - See also

9. Developing Middleware
- Introduction
- Authenticating with middleware
  - How to do it...
  - How it works...
  - See also
- Using middleware to implement access control
  - How to do it...
  - How it works...
  - See also
- Improving performance using the cache
  - How to do it...
  - How it works...
  - There's more...
  - See also
- Implementing routing

How to do it...
How it works...
See also
Making inter-framework system calls
How to do it...
How it works...
Using middleware to cross languages
How to do it...
10. Looking at Advanced Algorithms
Introduction
Using getters and setters
How to do it...
How it works...
Implementing a linked list
How to do it...
How it works...
There's more...
Building a bubble sort
How to do it...
How it works...
Implementing a stack
How to do it...
How it works...
Building a binary search class
How to do it...
How it works...
See also
Implementing a search engine
How to do it...
How it works...
Displaying a multi-dimensional array and accumulating totals
How to do it...
How it works...
11. Implementing Software Design Patterns
Introduction
Creating an array to object hydrator
How to do it...

- How it works...
- Building an object to array hydrator
  - How to do it...
  - How it works...
- Implementing a strategy pattern
  - How to do it...
  - How it works...
- Defining a mapper
  - How to do it...
  - How it works...
- Implementing object-relational mapping
  - How to do it...
    - Technique #1 - pre-loading all child information
    - Technique #2 - embedding secondary lookups
  - How it works...
  - See also
- Implementing the Pub/Sub design pattern
  - How to do it...
  - How it works...
  - There's more...
  - See also

12. Improving Web Security
- Introduction
- Filtering $_POST data
  - How to do it...
  - How it works...
  - See also
- Validating $_POST data
  - How to do it...
  - How it works...
  - See also
- Safeguarding the PHP session
  - How to do it...
  - How it works...
  - See also
- Securing forms with a token
  - How to do it...

How it works...
See also
Building a secure password generator
How to do it...
How it works...
See also
Safeguarding forms with a CAPTCHA
How to do it...
Generating a text CAPTCHA
Generating an image CAPTCHA
How it works...
There's more...
See also
Encrypting/decrypting without mcrypt
How to do it...
How it works...
There's more...
See also
13. Best Practices, Testing, and Debugging
Introduction
Using Traits and Interfaces
How to do it...
How it works...
Universal exception handler
How to do it...
How it works...
See also
Universal error handler
How to do it...
How it works...
See also
Writing a simple test
How to do it...
Running simple tests
Testing database Model classes
Using mock classes
Using anonymous classes as mock objects

Using Mock Builder
How it works...
Running simple tests
Testing database model classes
Using mock classes
There's more...
See also...
Writing a test suite
How to do it...
How it works...
See also...
Generating fake test data
How to do it...
How it works...
There's more...
Customizing sessions using session_start parameters
How to do it...
How it works...
See also...
A. Defining PSR-7 Classes
Introduction
Implementing PSR-7 value object classes
Getting ready
How to do it...
How it works...
See also
Developing a PSR-7 Request class
How to do it...
How it works...
See also
Defining a PSR-7 Response class
How to do it...
How it works...
See also
2. Module 2
1. Setting Up the Environment
Setting up Windows

- Setting up Debian or Ubuntu
  - Debian
  - Ubuntu
- Setting up CentOS
  - Installing NGINX
  - Installing PHP 7
  - Installing Percona Server
- Setting up Vagrant
- Summary

2. New Features in PHP 7
- OOP features
  - Type hints
    - Scalar type hints
    - Return type hints
  - Namespaces and group use declaration
    - Non mixed group use declarations
    - Mixed group use declarations
    - The compound namespace declaration
  - The anonymous classes
  - Old-style constructor deprecation
  - The throwable interface
    - Error
- New operators
  - The Spaceship operator (<=>)
  - The null coalesce operator(??)
- Uniform variable syntax
- Miscellaneous features and changes
  - Constant arrays
  - Multiple default cases in the switch statement
  - The options array for session_start function
  - Filtered unserialize function
- Summary

3. Improving PHP 7 Application Performance
- NGINX and Apache
  - Apache
  - NGINX
- HTTP server optimization

- Caching static files
  - Apache
  - NGINX
- HTTP persistent connection
  - Apache
  - NGINX
  - GZIP compression
    - Apache
    - NGINX
  - Using PHP as a separate service
  - Disabling unused modules
    - Apache
    - NGINX
  - Web server resources
    - NGINX
- Content Delivery Network (CDN)
  - Using CDN
- CSS and JavaScript optimization
  - Merging
  - Minifying
    - Minify
    - Grunt
- Full page caching
- Varnish
- The infrastructure
  - Web servers
  - The database server
  - Load balancer (LB)
  - HAProxy load balancing
    - HAProxy installation
    - HAProxy load balancing
- Summary

4. Improving Database Performance
- The MySQL database
  - Query caching
- Storage engines
  - The MyISAM storage engine

- The InnoDB storage engine
  - innodb_buffer_pool_size
  - innodb_buffer_pool_instances
  - innodb_log_file_size
- The Percona Server - a fork of MySQL
  - Installing the Percona Server
- MySQL performance monitoring tools
  - phpMyAdmin
  - The MySQL workbench
  - Percona Toolkit
    - pt-query-digest
    - pt-duplicate-key-checker
- Percona XtraDB Cluster (PXC)
- Redis – the key-value cache store
  - Connecting with the Redis server
  - Storing and fetching data from the Redis server
  - Redis management tools
- Memcached key-value cache store
- Summary

5. Debugging and Profiling
- Xdebug
  - Debugging with Sublime Text
  - Debugging with Eclipse
- Profiling with Xdebug
- PHP DebugBar
- Summary

6. Stress/Load Testing PHP Applications
- Apache JMeter
- ApacheBench (ab)
- Siege
- Load testing real-world applications
  - Magento 2
  - WordPress 4
  - Drupal 8
- Summary

7. Best Practices in PHP Programming
- Coding styles

- Test-driven development (TDD)
- Design patterns
- Service-oriented architecture (SOA)
- Being object-oriented and reusable always
- PHP frameworks
- Version control system (VCS) and Git
- Deployment and Continuous Integration (CI)
- Summary

A. Tools to Make Life Easy
- Composer – A dependency manager for PHP
  - Composer installation
  - Using Composer
- Git – A version control system
  - Git installation
  - Using Git
  - Creating new branches and merging
  - Cloning a repository
  - Webhooks
  - Desktop tools to manage repositories
- Grunt watch
- Summary

B. MVC and Frameworks
- The MVC design pattern
  - Model
  - Views
  - Controllers
- Laravel
  - Installation
  - Features
    - Routing
  - Eloquent ORM
  - Artisan CLI
  - Migrations
  - Blade templates
  - Other features
- Lumen
- Apigility

- Summary
- 3. Module 3
  - 1. Legacy Applications
    - The typical PHP application
      - File Structure
      - Page Scripts
      - Rewrite or Refactor?
      - The Pros and Cons of Rewriting
      - Why Don't Rewrites Work?
        - The Context-switching problem
        - The Knowledge problem
        - The Schedule Problem
        - Iterative Refactoring
    - Legacy Frameworks
      - Framework-based Legacy Applications
      - Refactoring to a Framework
    - Review and next steps
  - 2. Prerequisites
    - Revision control
      - PHP version
      - Editor/IDE
      - Style Guide
    - Test suite
    - Review and next steps
  - 3. Implement an Autoloader
    - PSR-0
    - A Single Location for Classes
      - Add Autoloader Code
      - As a Global Function
        - As a Closure
        - As a Static or Instance method
    - Using The __autoload() Function
      - Autoloader Priority
    - Common Questions
      - What If I Already Have An Autoloader?
      - What are the Performance Implications Of Autoloading?
      - How Do Class Names Map To File Names?

Review and next steps
4. Consolidate Classes and Functions
Consolidate Class Files
Find a candidate include
Move the class file
Remove the related include calls
Spot check the codebase
Commit, Push, Notify QA
Do ... While
Consolidate functions into class files
Find a candidate include
Convert the function file to a class file
Change function calls to static method calls
Spot check the static method calls
Move the class file
Do ... While
Common Questions
Should we remove the autoloader include call?
How should we pick files for candidate include calls?
What if an include defines more than one class?
What if the one-class-per-file rule is disagreeable?
What if a Class or Function is defined inline?
What if a definition file also executes logic?
What if two classes have the same name?
What about third-party libraries?
What about system-wide libraries?
For functions, can we use instance methods instead of static methods?
Can we automate this process?
Review and next steps
5. Replace global With Dependency Injection
Global Dependencies
The replacement process
Find a global variable
Convert global variables to properties
Spot check the class
Convert global properties to constructor parameters

- Convert instantiations to use parameters
- Spot check, Commit, Push, Notify QA
- Do ... While
- Common Questions
  - What if we find a global in a static method?
  - Is there an alternative conversion process?
  - What about class names in variables?
  - What about superglobals?
  - What about $GLOBALS?
- Review and next steps

6. Replace new with Dependency Injection
- Embedded instantiation
- The replacement process
  - Find a new keyword
  - Extract One-Time creation to dependency injection
  - Extract repeated creation to factory
- Change instantiation calls
  - Spot Check, Commit, Push, Notify QA
  - Do ... While
  - Common Questions
    - What About Exceptions and SPL Classes?
    - What about Intermediary Dependencies?
    - Isn't this a lot of code?
    - Should a factory create collections?
  - Can we automate all these Injections?
- Review and next steps

7. Write Tests
- Fighting test resistance
- The way of Testivus
  - Setting up a test suite
    - Install PHPUnit
    - Create a tests/ directory
    - Pick a class to test
    - Write a test case
    - Do ... While
  - Common Questions
    - Can we skip this step and do it later?

Come On, Really, Can We Do This Later?
What about hard-to-test classes?
What about our earlier characterization tests?
Should we test private and protected methods?
Can we change a test after we write it?
Do we need to test Third-party libraries?
What about code coverage?
Review and next steps
8. Extract SQL statements to Gateways
Embedded SQL Statements
The extraction process
Search for SQL statements
Move SQL to a Gateway class
Namespace and Class names
Method names
An initial Gateway class method
Defeating SQL Injection
Write a test
Replace the original code
Test, Commit, Push, Notify QA
Do ... While
Common Questions
What about INSERT, UPDATE, and DELETE Statements?
What about Repetitive SQL strings?
What about complex query strings?
What about queries inside non-Gateway classes?
Can we extend from a base Gateway class?
What about multiple queries and complex result structures?
What if there is no Database Class?
Review and next steps
9. Extract Domain Logic to Transactions
Embedded Domain Logic
Domain logic patterns
The Extraction Process
Search for uses of Gateway
Discover and Extract Relevant Domain Logic

Example Extraction
Spot check the remaining original code
Write tests for the extracted transactions
Spot check again, Commit, Push, Notify QA
Do ... While
Common Questions
Are we talking about SQL transactions?
What about repeated Domain Logic?
Are printing and echoing part of Domain Logic?
Can a transaction be a class instead of a Method?
What about Domain Logic in Gateway classes?
What about Domain logic embedded in Non-Domain classes?
Review and next steps
10. Extract Presentation Logic to View Files
Embedded presentation logic
The Extraction process
Search for Embedded presentation logic
Rearrange the Page script and Spot Check
Extract Presentation to View file and Spot Check
Create a views/ Directory
Pick a View File name
Move Presentation Block to View file
Add Proper Escaping
Write View File Tests
The tests/views/ directory
Writing a View File Test
Asserting Correctness Of Content
Commit, Push, Notify QA
Do ... While
Common Questions
What about Headers and Cookies?
What if we already have a Template system?
What about Streaming Content?
What if we have lots of Presentation variables?
What about class methods that generate output?
What about Business Logic Mixed into the presentation?

        What if a page contains only presentation logic?
    Review and next steps
11. Extract Action Logic to Controllers
    Embedded action logic
        The Extraction Process
        Search for Embedded Action Logic
        Rearrange the Page Script and Spot Check
            Identify Code Blocks
            Move Code to Its Related Block
            Spot Check the Rearranged Code
            Extract a Controller Class
            Pick a Class Name
            Create a Skeleton Class File
        Move the Action Logic and Spot Check
        Convert Controller to Dependency Injection and Spot Check
            Write a Controller Test
            Commit, Push, Notify QA
            Do ... While
        Common Questions
            Can we pass parameters to the Controller method?
        Can a Controller have Multiple actions?
        What If the Controller contains include Calls?
    Review and next steps
12. Replace Includes in Classes
    Embedded include Calls
    The Replacement process
        Search for include Calls
        Replacing a Single include Call
        Replacing Multiple include Calls
        Copy include file to Class Method
        Replace the original include Call
        Discover coupled variables through testing
        Replace other include Calls and Test
        Delete the include file and test
            Write a test and refactor
            Convert to Dependency Injection and test
            Commit, Push, Notify QA

Do ... While
Common QuestionsCan one class receive logic from many include files?
What about include calls originating in non-class files?
Review and next steps
13. Separate Public and Non-Public Resources
Intermingled resources
The separation process
Coordinate with operations personnel
Create a document root directory
Reconfigure the server
Move public resources
Commit, push, coordinate
Common Questions
Is This Really Necessary?
Review and next steps
14. Decouple URL Paths from File Paths
Coupled Paths
The Decoupling Process
Coordinate with Operations
Add a Front Controller
Create a pages/ Directory
Reconfigure the Server
Spot check
Move Page scripts
Commit, Push, Coordinate
Common Questions
Did we really Decouple the Paths?
Review and next steps
15. Remove Repeated Logic in Page Scripts
Repeated logic
The Removal Process
Modify the Front controller
Remove Logic from Page Scripts
Spot Check, Commit, Push, Notify QA
Common Questions
What if the Setup Work Is Inconsistent?

What if we used inconsistent naming?
Review and next steps
16. Add a Dependency Injection Container
What is a Dependency Injection Container?
Adding a DI Container
Add a DI Container Include File
Add a Router Service
Modify the Front Controller
Extract Page Scripts to Services
Create a Container Service
Route the URL Path to the Container Service
Spot Check and Commit
Do ... While
Remove pages/, Commit, Push, Notify QA
Common Questions
How can we refine our service definitions?
What if there are includes In the Page Script?
Can we reduce the size of the services.php file?
Can we reduce the size of the router service?
What if we cannot update to PHP 5.3?
Review and next steps
17. Conclusion
Opportunities for improvement
Conversion to Framework
Review and next steps
A. Typical Legacy Page Script
B. Code before Gateways
C. Code after Gateways
D. Code after Transaction Scripts
E. Code before Collecting Presentation Logic
F. Code after Collecting Presentation Logic
G. Code after Response View File
H. Code after Controller Rearrangement
I. Code after Controller Extraction
J. Code after Controller Dependency Injection
Bibliography
Index

PHP 7: Real World Application Development

PHP 7: Real World Application Development

Use new features of PHP 7 to solve practical, real-world problems faced by PHP developers like yourself every day.

A course in three modules



BIRMINGHAM - MUMBAI

PHP 7: Real World Application Development

Copyright © 2016 Packt Publishing

All rights reserved. No part of this course may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this course to ensure the accuracy of the information presented. However, the information contained in this course is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this course.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this course by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Published on: Month 2011

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78712-900-9

www.packtpub.com

Credits

Authors

Doug Bierer

Altaf Hussain

Paul Jones

Reviewers

Salvatore Pappalardo

Vincenzo Provenza

Raul Mesa Ros

Adam Culp

Content Development Editor

Onkar Wani

Graphics

Abhinash Sahu

Production Coordinator

Shraddha Falebhai

Preface

PHP 7 has taken the open source community by storm, breaking records for speed, which is, metaphorically, causing heads to turn. In its most fundamental sense, the core engineering team has effected a major rewrite of the language but has still managed to maintain backward compatibility to a high degree. PHP is a great language for developing web applications. It is essentially a server-side scripting language that is also used for general-purpose programming. PHP 7 is the latest version, providing major backward-compatibility breaks and focusing on improved performance and speed. This means you can maintain high traffic on your websites with low-cost hardware and servers through a multithreading web server.

What this learning path covers

Module 1*, PHP 7 Programming Cookbook,* This module demonstrates intermediate to advanced PHP techniques with a focus on PHP 7. Each recipe is designed to solve practical, real-world problems faced by PHP developers like yourself every day. It also cover new ways of writing PHP code made possible only in version 7. In addition, we discuss backward-compatibility breaks and give you plenty of guidance on when and where PHP 5 code needs to be changed to produce the correct results when running under PHP 7. This module also incorporates the latest PHP 7.x features.By the end of the module, you will be equipped with the tools and skills required to deliver efficient applications for your websites and enterprises

Module 2*, Learning PHP 7 High Performance*, This module is fast-paced introduction to PHP 7 will improve your productivity and coding skills. The concepts covered will allow you, as a PHP programmer, to improve the performance standards of your applications. We will introduce you to the new features in PHP 7 and then will run through the concepts of object-oriented programming

(OOP) in PHP 7. Next, we will shed some light on how to improve your PHP 7 applications' performance and database performance. Through this module, you will be able to improve the performance of your programs using the various benchmarking tools discussed in the module. At the end,module discusses some best practices in PHP programming to help you improve the quality of your code

Module 3*, Modernizing Legacy Applications in PHP*, This module will show you how to modernize your application in terms of practice and technique, rather than in terms of using tools such as frameworks and libraries, by extracting and replacing its legacy artifacts. We will use a step-by-step approach, moving slowly and methodically, to improve your application from the ground up. We'll show you how dependency injection can replace both the new and global dependencies. We'll also show you how to change the presentation logic to view files and the action logic to a controller. Moreover, we'll keep your application running the whole time. Each completed step in the process will keep your codebase fully operational with higher quality. When we are done, you will be able to breeze through your code like the wind. Your code will be autoloaded, dependency-injected, unit-tested, layer-separated, and front-controlled. Most of the very limited code we will add to your application is specific to this module. We will be improving ourselves as programmers, as well as improving the quality of our legacy application.

What you need for this learning path

Module 1:

All you need, to successfully implement the recipes presented in this module will be a computer, 100MB of extra disk space, and a text or code editor (not a word processor!). The first chapter will cover how to set up a PHP 7 development environment. Having a web server is optional as PHP 7 includes a development web server. An Internet connection is not required, but it might be useful to download code (such as the set of PSR-7 interfaces), and review PHP 7.x documentation.

Module 2:

Any hardware specification that is compliant to run the latest versions of the following software should be enough to get through this module:

- Operating systems: Debian or Ubuntu
- Software: NGINX, PHP 7, MySQL, PerconaDB, Redis, Memcached, Xdebug,

Apache JMeter, ApacheBench, Siege, and Git

Module 3:

You will have refer ‘Chapter 2, Prerequisites’ of this module to understand the basic hardware and software requirements needed on this module. This chapter gives a detailed description of the requirements.

Who this learning path is for

If you are an aspiring web developer, mobile developer, or back-end programmer, who has basic experience in PHP programming and wants to develop performance-critical applications, then this course is for you. It will take your PHP programming skills to next level

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this course—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail <feedback@packtpub.com>, and mention the course's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a course, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt course, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this course from your account at http://www.packtpub.com. If you purchased this course elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the course in the **Search** box.
5. Select the course for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this course from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the course's webpage at the Packt Publishing website. This page can be accessed by entering the course's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac

- 7-Zip / PeaZip for Linux

The code bundle for the course is also hosted on GitHub at https://github.com/PacktPublishing/PHP-7-Be-Pro-at-Applications-Development. We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our courses—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this course. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your course, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to https://www.packtpub.com/books/content/support and enter the name of the course in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at <copyright@packtpub.com> with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this course, you can contact us at <questions@packtpub.com>, and we will do our best to address the problem.

Part 1. Module 1

PHP 7 Programming Cookbook

Over 80 recipes that will take your PHP 7 web development skills to the next level!

Chapter 1. Building a Foundation

In this chapter, we will cover the following topics:

- PHP 7 installation considerations
- Using the built-in PHP web server
- Defining a test MySQL database
- Installing PHPUnit
- Implementing class autoloading
- Hoovering a website
- Building a deep web scanner
- Creating a PHP 5 to PHP 7 code converter

Introduction

This chapter is designed as a *quick start* that will get you up and running on PHP 7 so that you can start implementing the recipes right away. The underlying assumption for this book is that you already have a good knowledge of PHP and programming. Although this book will not go into detail about the actual installation of PHP, given that PHP 7 is relatively new, we will do our best to point out the quirks and *gotchas* you might encounter during a PHP 7 installation.

PHP 7 installation considerations

There are three primary means of acquiring PHP 7:

- Downloading and installing directly from the source code
- Installing *pre-compiled* binaries
- Installing a *AMP package (that is, XAMPP, WAMP, LAMP, MAMP, and so on)

How to do it...

The three methods are listed in order of difficulty. However, the first approach, although tedious, will give you the most finite control over extensions and options.

Installing directly from source

In order to utilize this approach, you will need to have a C compiler available. If you are running Windows, **MinGW** is a free compiler that has proven popular. It is based on the **GNU Compiler Collection** (**GCC)** compiler provided by the **GNU** project. Non-free compilers include the classic **Turbo C** compiler from Borland, and, of course, the compiler that is preferred by Windows developers is **Visual Studio**. The latter, however, is designed mainly for C++ development, so when you compile PHP, you will need to specify C mode.

When working on an Apple Mac, the best solution is to install the **Apple Developer Tools**. You can use the **Xcode IDE** to compile PHP 7, or run `gcc` from a terminal window. In a Linux environment, from a terminal window, run `gcc`.

When compiling from a terminal window or command line, the normal procedure is as follows:

- `configure`
- `make`
- `make test`
- `make install`

For information on configuration options (that is, when running `configure`), use the `help` option:

```
configure --help
```

Errors you might encounter during the configuration stage are mentioned in the following table:

Error	Fix
`configure: error: xml2-config not found. Please check your libxml2 installation`	You just need to install `libxml2`. For this error, please refer to the following link: http://superuser.com/questions/740399/how-to-fix-php-installation-when-xml2-config-is-missing
`configure: error: Please reinstall readline - I cannot find readline.h`	Install `libreadline-dev`
`configure: WARNING: unrecognized options: --enable-spl, --enable-reflection, --with-libxml`	Not a big deal. These options are defaults and don't need to be included. For more details, please refer to the following link: http://jcutrer.com/howto/linux/how-to-compile-php7-on-ubuntu-14-04

Installing PHP 7 from pre-compiled binaries

As the title implies, **pre-compiled** binaries are a set of binary files that somebody else has kindly compiled from PHP 7 source code and has made available.

In the case of Windows, go to http://windows.php.net/. You will find a good set of tips in the left column that pertain to which version to choose, **thread safe** versus **non-read safe**, and so forth. You can then click on **Downloads** and look for the ZIP file that applies to your environment. Once the ZIP file has

been downloaded, extract the files into the folder of your choice, add `php.exe` to your path, and configure PHP 7 using the `php.ini` file.

To install the pre-compiled binaries on a Mac OS X system, it is best to involve a package management system. The ones recommended for PHP include the following:

- MacPorts
- Liip
- Fink
- Homebrew

In the case of Linux, the packaging system used depends on which Linux distribution you are using. The following table, organized by Linux distribution, summarizes where to look for the PHP 7 package.

Distribution	Where to find PHP 7	Notes

Debian	`packages.debian.org/stable/php`
	
	`repos-source.zend.com/zend-server/early-access/php7/php-7*DEB*`
		Use this command:
		
		`sudo apt-get install php7`
		
		Alternatively, you can use a graphical package management tool such as **Synaptic**.
		
		Make sure you select **php7** (and not php5).

Distribution	Where to find PHP 7	Notes
Ubuntu	packages.ubuntu.com repos-source.zend.com/zend-server/early-access/php7/php-7*DEB*	Use this command: sudo apt-get install php7 Be sure to choose the right version of Ubuntu. Alternatively, you can use a graphical package management tool such as **Synaptic**.
Fedora / Red Hat	admin.fedoraproject.org/pkgdb/packages repos-source.zend.com/zend-server/early-access/php7/php-7*RHEL*	Make sure you are the root user: su Use this command: **dnf install php7** Alternatively, you can use a graphical package management tool such as the GNOME Package Manager.

Distribution	Where to find PHP 7	Notes
OpenSUSE	`software.opensuse.org/package/php7`	Use this command: `yast -i php7` Alternatively, you can run `zypper`, or use **YaST** as a graphical tool.

Installing a *AMP package

AMP refers to **Apache**, **MySQL**, and **PHP** (also **Perl** and **Python**). The * refers to Linux, Windows, Mac, and so on (that is, LAMP, WAMP, and MAMP). This approach is often the easiest, but gives you less control over the initial PHP installation. On the other hand, you can always modify the `php.ini` file and install additional extensions to customize your installation as needed. The following table summarizes a number of popular *AMP packages:

Package	Where is it found	Free?	Supports*
`XAMPP`	www.apachefriends.org/download.html	Y	WML
`AMPPS`	www.ampps.com/downloads	Y	WML
`MAMP`	www.mamp.info/en	Y	WM
`WampServer`	sourceforge.net/projects/wampserver	Y	W

Package	Where is it found	Free?	Supports*
`EasyPHP`	www.easyphp.org	Y	W
`Zend Server`	www.zend.com/en/products/zend_server	N	WML

In the preceding table, we've enlisted the *AMP packages where ***** is replaced by **W** for Windows, **M** for Mac OS X, and **L** for Linux.

There's more...

When you install a pre-compiled binary from a package, only `core` extensions are installed. Non-core PHP extensions must be installed separately.

It's worth noting that PHP 7 installation on cloud computing platforms will often follow the installation procedure outlined for pre-compiled binaries. Find out if your cloud environment uses Linux, Mac, or Windows virtual machines, and then follow the appropriate procedure as mentioned in this recipe.

It's possible that PHP 7 hasn't yet reached your favorite repository for pre-compiled binaries. You can always install from source, or consider installing one of the *AMP packages (see the next section). An alternative for Linux-based systems is to use the **Personal Package Archive** (**PPA**) approach. Because PPAs have not undergone a rigorous screening process, however, security could be a concern. A good discussion on security considerations for PPAs is found at http://askubuntu.com/questions/35629/are-ppas-safe-to-add-to-my-system-and-what-are-some-red-flags-to-watch-out-fo.

See also

General installation considerations, as well as instructions for each of the three major OS platforms (Windows, Mac OS X, and Linux), can be found at http://php.net/manual/en/install.general.php.

The website for MinGW is http://www.mingw.org/.

Instructions on how to compile a C program using Visual Studio can be found at https://msdn.microsoft.com/en-us/library/bb384838.

Another possible way to test PHP 7 is by using a virtual machine. Here are a couple of tools with their links, which might prove useful:

- **Vagrant**: https://github.com/rlerdorf/php7dev (php7dev is a Debian 8 Vagrant image that is preconfigured for testing PHP apps and developing extensions across many versions of PHP)
- **Docker**: https://hub.docker.com/r/coderstephen/php7/ (it contains a PHP7 Docker container)

Using the built-in PHP web server

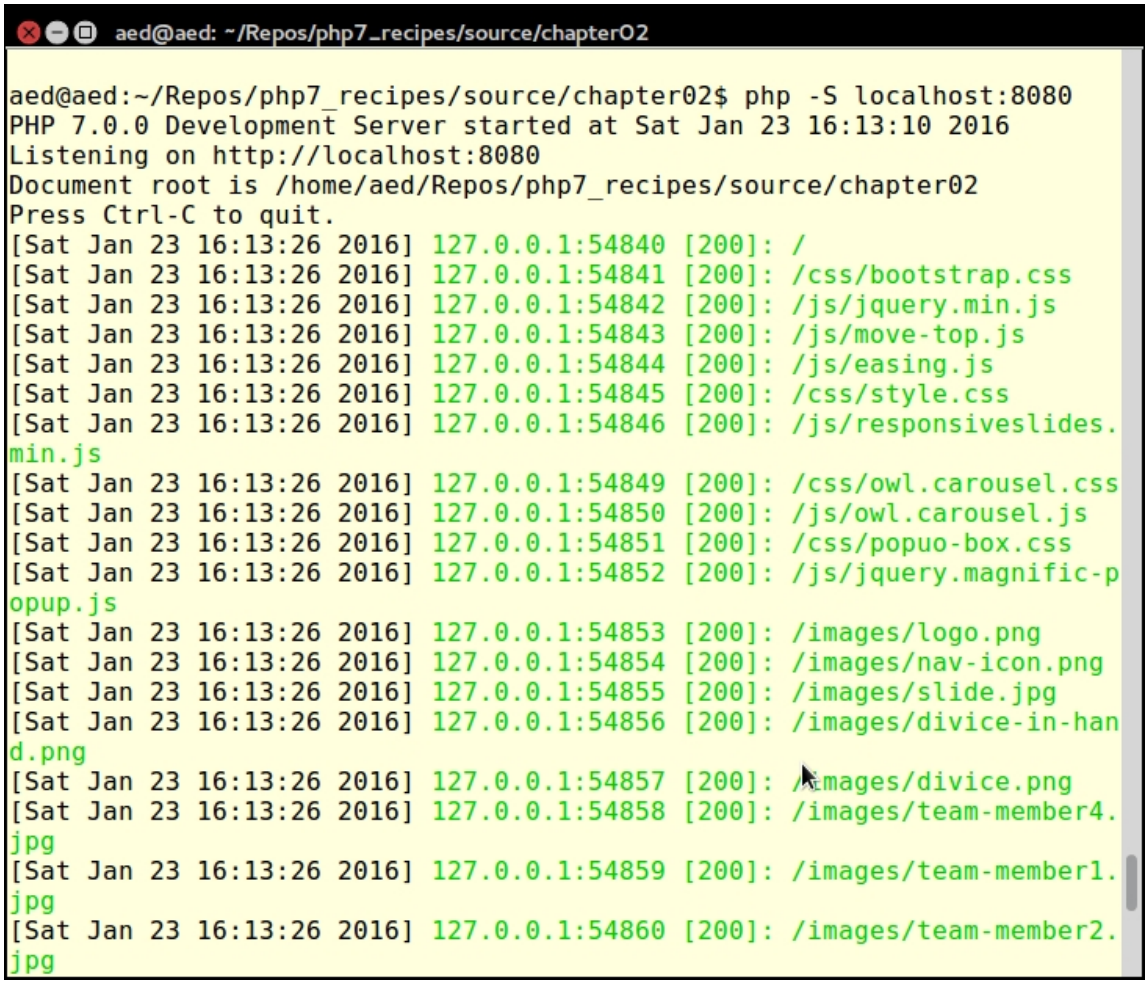
Aside from unit testing and running PHP directly from the command line, the obvious way to test your applications is to use a web server. For long-term projects, it would be beneficial to develop a virtual host definition for a web server that most closely mirrors the one used by your customer. Creating such definitions for the various web servers (that is, Apache, NGINX, and so on) is beyond the scope of this book. Another quick and easy-to-use alternative (which we have room to discuss here) is to use the built-in PHP 7 web server.

How to do it...

1. To activate the PHP web server, first change to the directory that will serve as the base for your code.
2. You then need to supply the hostname or IP address and, optionally, a port. Here is an example you can use to run the recipes supplied with this book:

```
cd /path/to/recipes
php -S localhost:8080
```

   You will see output on your screen that looks something like this:

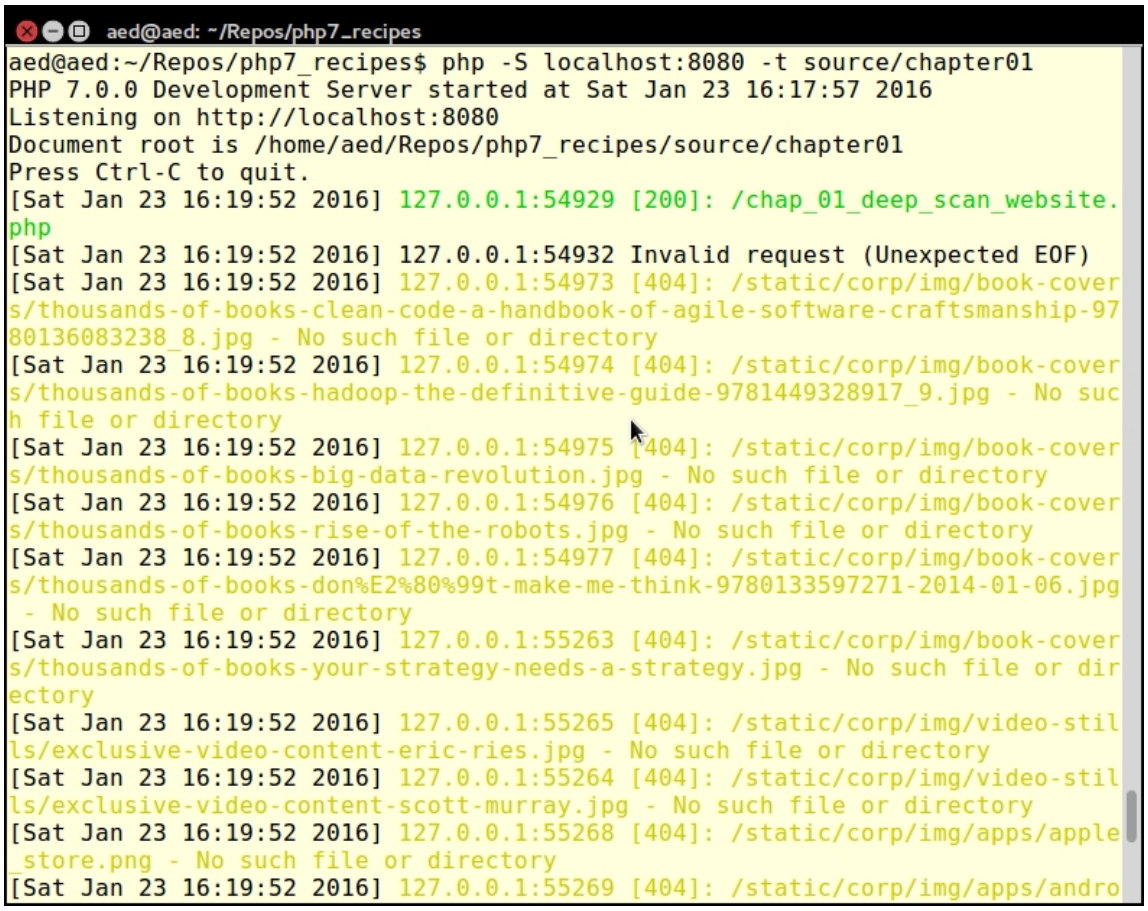


3. As the built-in web server continues to service requests, you will also see access information, HTTP status codes, and request information.
4. If you need to set the web server document root to a directory other than the current one, you can use the `-t` flag. The flag must then be followed by a valid directory path. The built-in web server will treat this directory as if it were the web document root, which is useful for security reasons. For security reasons, some frameworks, such as Zend Framework, require that the web document root is different from where your actual source code resides.

   Here is an example using the `-t` flag:

```
php -S localhost:8080 -t source/chapter01
```

Here is an example of the output:



Defining a test MySQL database

For test purposes, along with the source code for the book, we've provided an SQL file with sample data at https://github.com/dbierer/php7cookbook. The name of the database used in the recipes for this book is `php7cookbook`.

How to do it...

1. Define a MySQL database, `php7cookbook`. Also assign rights to the new database to a user called `cook` with the password `book`. The following table summarizes these settings:

Item	**Notes**
Database name	`php7cookbook`
Database user	`cook`
Database user password	`book`

2. Here is an example of SQL needed to create the database:

```
CREATE DATABASE IF NOT EXISTS dbname DEFAULT
CHARACTER SET utf8 COLLATE utf8_general_ci;
CREATE USER 'user'@'%' IDENTIFIED WITH
mysql_native_password;
SET PASSWORD FOR 'user'@'%' =
PASSWORD('userPassword');
GRANT ALL PRIVILEGES ON dbname.* to
'user'@'%';
```

```
GRANT ALL PRIVILEGES ON dbname.* to
'user'@'localhost';
FLUSH PRIVILEGES;
```

3. Import the sample values into the new database. The import file, `php7cookbook.sql`, is located at https://github.com/dbierer/php7cookbook/blob/master/php7cookbook.sql.

Installing PHPUnit

Unit testing is arguably the most popular means of testing PHP code. Most developers will agree that a solid suite of tests is a requirement for any properly developed project. Few developers actually write these tests. A lucky few have an independent testing group that writes the tests for them! After months of skirmishing with the testing group, however, the remains of the lucky few tend to grumble and complain. In any event, any book on PHP would not be complete without at least a nod and a wink towards testing.

The place to find the latest version of **PHPUnit** is https://phpunit.de/. PHPUnit5.1 and above support PHP 7. Click on the link for the desired version, and you will download a `phpunit.phar` file. You can then execute commands using the archive, as follows:

```
php phpunit.phar <command>
```

Tip

The `phar` command stands for **PHP Archive**. The technology is based on `tar`, which itself was used in UNIX. A `phar` file is a collection of PHP files that are packed together into a single file for convenience.

Implementing class autoloading

When developing PHP using an **object-oriented programming** (**OOP**) approach, the recommendation is to place each class in its own file. The advantage of following this recommendation is the ease of long-term maintenance and improved readability. The disadvantage is that each class definition file must be included (that is, using `include` or its variants). To address this issue, there is a mechanism built into the PHP language that will *autoload* any class that has not already been specifically included.

Getting ready

The minimum requirement for PHP autoloading is to define a global `__autoload()` function. This is a *magic* function called automatically by the PHP engine when a class is requested but where said class has not been included. The name of the requested class will appear as a parameter when `__autoload()` is invoked (assuming that you have defined it!). If you are using PHP namespaces, the full namespaced name of the class will be passed. Because `__autoload()` is a *function*, it must be in the global namespace; however, there are limitations on its use. Accordingly, in this recipe, we will make use of the `spl_autoload_register()` function, which gives us more flexibility.

How to do it...

1. The class we will cover in this recipe is `Application\Autoload\Loader`. In order to take advantage of the relationship between PHP namespaces and autoloading, we name the file `Loader.php` and place it in the `/path/to/cookbook/files/Application/Autoload` folder.
2. The first method we will present simply loads a file. We use `file_exists()` to check before running `require_once()`. The reason for this is that if the file is not found, `require_once()` will

generate a fatal error that cannot be caught using PHP 7's new error handling capabilities:

```
protected static function loadFile($file)
{
    if (file_exists($file)) {
        require_once $file;
        return TRUE;
    }
    return FALSE;
}
```

3. We can then test the return value of `loadFile()` in the calling program and loop through a list of alternate directories before throwing an `Exception` if it's ultimately unable to load the file.

> ## Tip
>
> You will notice that the methods and properties in this class are static. This gives us greater flexibility when registering the autoloading method, and also lets us treat the `Loader` class like a **Singleton**.

4. Next, we define the method that calls `loadFile()` and actually performs the logic to locate the file based on the namespaced classname. This method derives a filename by converting the PHP namespace separator `\` into the directory separator appropriate for this server and appending `.php`:

```
public static function autoLoad($class)
{
    $success = FALSE;
    $fn = str_replace('\\',
DIRECTORY_SEPARATOR, $class)
          . '.php';
    foreach (self::$dirs as $start) {
        $file = $start . DIRECTORY_SEPARATOR .
$fn;
        if (self::loadFile($file)) {
            $success = TRUE;
```

```
                break;
            }
        }
        if (!$success) {
            if (!self::loadFile(__DIR__
                . DIRECTORY_SEPARATOR . $fn)) {
                throw new \Exception(
                    self::UNABLE_TO_LOAD . ' ' .
$class);
            }
        }
        return $success;
    }
```

5. Next, the method loops through an array of directories we call `self::$dirs`, using each directory as a starting point for the derived filename. If not successful, as a last resort, the method attempts to load the file from the current directory. If even that is not successful, an `Exception` is thrown.
6. Next, we need a method that can add more directories to our list of directories to test. Notice that if the value provided is an array, `array_merge()` is used. Otherwise, we simply add the directory string to the `self::$dirs` array:

```
    public static function addDirs($dirs)
    {
        if (is_array($dirs)) {
            self::$dirs = array_merge(self::$dirs,
$dirs);
        } else {
            self::$dirs[] = $dirs;
        }
    }
```

7. Then, we come to the most important part; we need to register our `autoload()` method as a **Standard PHP Library** (**SPL**) autoloader. This is accomplished using `spl_autoload_register()` with the `init()` method:

```
    public static function init($dirs = array())
    {
        if ($dirs) {
            self::addDirs($dirs);
        }
```

```
            if (self::$registered == 0) {
                spl_autoload_register(__CLASS__ .
'::autoload');
                self::$registered++;
            }
        }
```

8. At this point, we can define `__construct()`, which calls `self::init($dirs)`. This allows us to also create an instance of `Loader` if desired:

```
        public function __construct($dirs = array())
        {
            self::init($dirs);
        }
```

How it works...

In order to use the autoloader class that we just defined, you will need to `require Loader.php`. If your namespace files are located in a directory other than the current one, you should also run `Loader::init()` and supply additional directory paths.

In order to make sure the autoloader works, we'll also need a test class. Here is a definition of `/path/to/cookbook/files/Application/Test/TestClass.php`:

```
<?php
namespace Application\Test;
class TestClass
{
    public function getTest()
    {
        return __METHOD__;
    }
}
```

Now create a sample `chap_01_autoload_test.php` code file to test the autoloader:

```
<?php
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
```

Next, get an instance of a class that has not already been loaded:

```
$test = new Application\Test\TestClass();
echo $test->getTest();
```

Finally, try to get a `fake` class that does not exist. Note that this will throw an error:

```
$fake = new Application\Test\FakeClass();
echo $fake->getTest();
```

Hoovering a website

Very frequently, it is of interest to scan a website and extract information from specific tags. This basic mechanism can be used to trawl the web in search of useful bits of information. At other times you need to get a list of `<IMG>` tags and the `SRC` attribute, or `<A>` tags and the corresponding `HREF` attribute. The possibilities are endless.

How to do it...

1. First of all, we need to grab the contents of the target website. At first glance it seems that we should make a cURL request, or simply use `file_get_contents()`. The problem with these approaches is that we will end up having to do a massive amount of string manipulation, most likely having to make inordinate use of the dreaded regular expression. In order to avoid all of this, we'll simply take advantage of an already existing PHP 7 class `DOMDocument`. So we create a `DOMDocument` instance, setting it to **UTF-8**. We don't care about whitespace, and use the handy `loadHTMLFile()` method to load the contents of the website into the object:

```
public function getContent($url)
{
    if (!$this->content) {
        if (stripos($url, 'http') !== 0) {
            $url = 'http://' . $url;
        }
        $this->content = new
DOMDocument('1.0', 'utf-8');
        $this->content->preserveWhiteSpace =
FALSE;
        // @ used to suppress warnings
generated from // improperly configured web
pages
        @$this->content->loadHTMLFile($url);
    }
    return $this->content;
}
```

Tip

Note that we precede the call to the `loadHTMLFile()` method with an `@`. This is not done to obscure bad coding (`!`) as was often the case in PHP 5! Rather, the `@` suppresses notices generated when the parser encounters poorly written HTML. Presumably, we could capture the notices and log them, possibly giving our `Hoover` class a diagnostic capability as well.

2. Next, we need to extract the tags which are of interest. We use the `getElementsByTagName()` method for this purpose. If we wish to extract *all* tags, we can supply `*` as an argument:

```
public function getTags($url, $tag)
{
    $count    = 0;
    $result   = array();
    $elements = $this->getContent($url)
                      -
>getElementsByTagName($tag);
    foreach ($elements as $node) {
        $result[$count]['value'] =
trim(preg_replace('/\s+/', ' ', $node-
>nodeValue));
        if ($node->hasAttributes()) {
            foreach ($node->attributes as
$name => $attr)
            {
                $result[$count]['attributes']
[$name] =
                    $attr->value;
            }
        }
        $count++;
    }
    return $result;
}
```

3. It might also be of interest to extract certain attributes rather than tags. Accordingly, we define another method for this

purpose. In this case, we need to parse through all tags and use `getAttribute()`. You'll notice that there is a parameter for the DNS domain. We've added this in order to keep the scan within the same domain (if you're building a web tree, for example):

```
public function getAttribute($url, $attr,
$domain = NULL)
{
    $result   = array();
    $elements = $this->getContent($url)
                      -
>getElementsByTagName('*');
    foreach ($elements as $node) {
        if ($node->hasAttribute($attr)) {
            $value = $node-
>getAttribute($attr);
            if ($domain) {
                if (stripos($value, $domain)
!== FALSE) {
                    $result[] = trim($value);
                }
            } else {
                $result[] = trim($value);
            }
        }
    }
    return $result;
}
```

How it works...

In order to use the new `Hoover` class, initialize the autoloader (described previously) and create an instance of the `Hoover` class. You can then run the `Hoover::getTags()` method to produce an array of tags from the URL you specify as an argument.

Here is a block of code from `chap_01_vacuuming_website.php` that uses the `Hoover` class to scan the O'Reilly website for `<A>` tags:

```
<?php
// modify as needed
define('DEFAULT_URL', 'http://oreilly.com/');
```

```
define('DEFAULT_TAG', 'a');

require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');

// get "vacuum" class
$vac = new Application\Web\Hoover();

// NOTE: the PHP 7 null coalesce operator is used
$url = strip_tags($_GET['url'] ?? DEFAULT_URL);
$tag = strip_tags($_GET['tag'] ?? DEFAULT_TAG);

echo 'Dump of Tags: ' . PHP_EOL;
var_dump($vac->getTags($url, $tag));
```

The output will look something like this:



See also

For more information on DOM, see the PHP reference page at http://php.net/manual/en/class.domdocument.php.

Building a deep web scanner

Sometimes you need to scan a website, but go one level deeper. For example, you want to build a web tree diagram of a website. This can be accomplished by looking for all `<A>` tags and following the `HREF` attributes to the next web page. Once you have acquired the child pages, you can then continue scanning in order to complete the tree.

How to do it...

1. A core component of a deep web scanner is a basic `Hoover` class, as described previously. The basic procedure presented in this recipe is to scan the target website and hoover up all the `HREF` attributes. For this purpose, we define a `Application\Web\Deep` class. We add a property that represents the DNS domain:

```
namespace Application\Web;
class Deep
{
    protected $domain;
```

2. Next, we define a method that will hoover the tags for each website represented in the scan list. In order to prevent the scanner from trawling the entire **World Wide Web** (**WWW**), we've limited the scan to the target domain. The reason why `yield from` has been added is because we need to yield the entire array produced by `Hoover::getTags()`. The `yield from` syntax allows us to treat the array as a sub-generator:

```
public function scan($url, $tag)
{
    $vac    = new Hoover();
    $scan   = $vac->getAttribute($url, 'href',
       $this->getDomain($url));
    $result = array();
    foreach ($scan as $subSite) {
        yield from $vac->getTags($subSite,
```

```
$tag);
        }
        return count($scan);
    }
```

> ## Note
>
> The use of `yield from` turns the `scan()` method into a PHP 7 delegating generator. Normally, you would be inclined to store the results of the scan into an array. The problem, in this case, is that the amount of information retrieved could potentially be massive. Thus, it's better to immediately yield the results in order to conserve memory and to produce immediate results. Otherwise, there would be a lengthy wait, which would probably be followed by an out of memory error.

3. In order to keep within the same domain, we need a method that will return the domain from the URL. We use the convenient `parse_url()` function for this purpose:

```
public function getDomain($url)
{
    if (!$this->domain) {
        $this->domain = parse_url($url,
PHP_URL_HOST);
    }
    return $this->domain;
}
```

How it works...

First of all, go ahead and define the `Application\Web\Deep` class defined previously, as well as the `Application\Web\Hoover` class defined in the previous recipe.

Next, define a block of code from `chap_01_deep_scan_website.php` that sets up autoloading (as described earlier in this chapter):

```
<?php
// modify as needed
define('DEFAULT_URL', 'unlikelysource.com');
define('DEFAULT_TAG', 'img');

require __DIR__ .
'/../../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/../..');
```

Next, get an instance of our new class:

```
$deep = new Application\Web\Deep();
```

At this point, you can retrieve URL and tag information from URL parameters. The PHP 7 `null coalesce` operator is useful for establishing fallback values:

```
$url = strip_tags($_GET['url'] ?? DEFAULT_URL);
$tag = strip_tags($_GET['tag'] ?? DEFAULT_TAG);
```

Some simple HTML will display results:

```
foreach ($deep->scan($url, $tag) as $item) {
    $src = $item['attributes']['src'] ?? NULL;
    if ($src && (stripos($src, 'png') ||
stripos($src, 'jpg'))) {
        printf('<br><img src="%s"/>', $src);
    }
}
```

See also

For more information on generators and `yield from`, please see the article at http://php.net/manual/en/language.generators.syntax.php.

Creating a PHP 5 to PHP 7 code converter

For the most part, PHP 5.x code can run unchanged on PHP 7. There are a few changes, however, that are classified as *backwards incompatible*. What this means is that if your PHP 5 code is written in a certain way, or uses functions that have been removed, your code will break, and you'll have a nasty error on your hands.

Getting ready

The *PHP 5 to PHP 7 Code Converter* does two things:

- Scans your code file and converts PHP 5 functionality that has been removed to its equivalent in PHP 7
- Adds comments with `// WARNING` where changes in language usage have occurred, but where a re-write is not possible

> ### Note
>
> Please note that after running the converter, your code is *not* guaranteed to work in PHP 7. You will still have to review the `// WARNING` tags added. At the least, this recipe will give you a good head start converting your PHP 5 code to work in PHP 7.

The core of this recipe is the new PHP 7 `preg_replace_callback_array()` function. What this amazing function allows you to do is to present an array of regular expressions as keys, with the value representing an independent callback. You can then pass the string through a series of transformations. Not only that, the subject of the array of callbacks can *itself* be an array.

How to do it...

1. In a new class `Application\Parse\Convert`, we begin with a `scan()` method, which accepts a filename as an argument. It checks to see if the file exists. If so, it calls the PHP `file()` function, which loads the file into an array, with each array element representing one line:

```
public function scan($filename)
{
    if (!file_exists($filename)) {
        throw new Exception(
            self::EXCEPTION_FILE_NOT_EXISTS);
    }
    $contents = file($filename);
    echo 'Processing: ' . $filename . PHP_EOL;

    $result = preg_replace_callback_array( [
```

2. Next, we start passing a series of key/value pairs. The key is a regular expression, which is processed against the string. Any matches are passed to the callback, which is represented as the value part of the key/value pair. We check for opening and closing tags that have been removed from PHP 7:

```
        // replace no-longer-supported opening
tags
        '!^\<\%(\n| )!' =>
            function ($match) {
                return '<?php' . $match[1];
            },

        // replace no-longer-supported opening
tags
        '!^\<\%=(\n| )!' =>
            function ($match) {
                return '<?php echo ' . $match[1];
            },

        // replace no-longer-supported closing tag
        '!\%\>!' =>
            function ($match) {
                return '?>';
            },
```

3. Next is a series of warnings when certain operations are detected and there is a potential code-break between how they're handled in PHP 5 versus PHP 7. In all these cases, the code is not re-written. Instead, an inline comment with the word `WARNING` is added:

```
        // changes in how $$xxx interpretation is
    handled
        '!(.*?)\$\$!' =>
            function ($match) {
                return '// WARNING: variable
    interpolation
                        . ' now occurs left-to-
    right' . PHP_EOL
                        . '// see:
    http://php.net/manual/en/'
                        . '//
    migration70.incompatible.php'
                        . $match[0];
            },

        // changes in how the list() operator is
    handled
        '!(.*?)list(\s*?)?\(!' =>
            function ($match) {
                return '// WARNING: changes have
    been made '
                        . 'in list() operator
    handling.'
                        . 'See:
    http://php.net/manual/en/'
                        .
    'migration70.incompatible.php'
                        . $match[0];
            },

        // instances of \u{
        '!(.*?)\\\u\{!' =>
            function ($match) {
            return '// WARNING: \\u{xxx} is now
    considered '
                    . 'unicode escape syntax' .
    PHP_EOL
                    . '// see:
    http://php.net/manual/en/'
                    . 'migration70.new-
```

```
features.php'
                . '#migration70.new-
features.unicode-'
                . 'codepoint-escape-syntax' .
PHP_EOL
                . $match[0];
    },

    // relying upon set_error_handler()
    '!(.*?)set_error_handler(\s*?)?.*\(!' =>
        function ($match) {
            return '// WARNING: might not '
                   . 'catch all errors'
                   . '// see:
http://php.net/manual/en/'
                   . '//
language.errors.php7.php'
                   . $match[0];
        },

    // session_set_save_handler(xxx)
    '!(.*?)session_set_save_handler(\s*?)?\
((.*?)\)!' =>
        function ($match) {
            if (isset($match[3])) {
                return '// WARNING: a bug
introduced in'
                       . 'PHP 5.4 which '
                       . 'affects the handler
assigned by '
                       .
'session_set_save_handler() and '
                       . 'where
ignore_user_abort() is TRUE
                       . 'has been fixed in
PHP 7.'
                       . 'This could
potentially break '
                       . 'your code under '
                       . 'certain
circumstances.' . PHP_EOL
                       . 'See:
http://php.net/manual/en/'
                       .
'migration70.incompatible.php'
                       . $match[0];
            } else {
                return $match[0];
```

```
            }
        },
```

4. Any attempts to use `<<` or `>>` with a negative operator, or beyond 64, is wrapped in a `try { xxx } catch() { xxx }` block, looking for an `ArithmeticError` to be thrown:

```
    // wraps bit shift operations in try /
catch
    '!^(.*?)(\d+\s*(\<\<|\>\>)\s*-?\d+)
(.*?)$!' =>
        function ($match) {
            return '// WARNING: negative and '
                   . 'out-of-range bitwise '
                   . 'shift operations will
now
                   . 'throw an
ArithmeticError' . PHP_EOL
                   . 'See:
http://php.net/manual/en/'
                   .
'migration70.incompatible.php'
                   . 'try {' . PHP_EOL
                   . "\t" . $match[0] .
PHP_EOL
                   . '} catch
(\\ArithmeticError $e) {'
                   . "\t" . 'error_log("File:"
                   . $e->getFile()
                   . " Message:" . $e-
>getMessage());'
                   . '}' . PHP_EOL;
        },
```

Note

PHP 7 has changed how errors are handled. In some cases, errors are moved into a similar classification as exceptions, and can be caught! Both the `Error` and the `Exception` class implement the `Throwable` interface. If you want to catch either an `Error` or an `Exception`, catch `Throwable`.

5. Next, the converter rewrites any usage of `call_user_method*()`, which has been removed in PHP 7. These are replaced with the equivalent using `call_user_func*()`:

```
    // replaces "call_user_method()" with
    // "call_user_func()"
    '!call_user_method\((.*?),(.*?)(,.*?)\)
(\b|;)!' =>
        function ($match) {
            $params = $match[3] ?? '';
            return '// WARNING:
call_user_method() has '
                        . 'been removed from PHP
7' . PHP_EOL
                        . 'call_user_func(['.
trim($match[2]) . ','
                        . trim($match[1]) . ']'
. $params . ');';
        },

    // replaces "call_user_method_array()"
    // with "call_user_func_array()"
    '!call_user_method_array\((.*?),(.*?),
(.*?)\)(\b|;)!' =>
        function ($match) {
            return '// WARNING:
call_user_method_array()'
                    . 'has been removed from
PHP 7'
                    . PHP_EOL
                    . 'call_user_func_array(['
                    . trim($match[2]) . ','
                    . trim($match[1]) . '], '
                    . $match[3] . ');';
        },
```

6. Finally, any attempt to use `preg_replace()` with the `/e` modifier is rewritten using a `preg_replace_callback()`:

```
     '!^(.*?)preg_replace.*?/e(.*?)$!' =>
    function ($match) {
        $last = strrchr($match[2], ',');
        $arg2 = substr($match[2], 2, -1 *
(strlen($last)));
        $arg1 = substr($match[0],
```

```
                        strlen($match[1]) + 12,
                        -1 * (strlen($arg2) +
strlen($last)));
            $arg1 = trim($arg1, '(');
            $arg1 = str_replace('/e', '/',
$arg1);
            $arg3 = '// WARNING: preg_replace()
"/e" modifier
                    . 'has been removed from
PHP 7'
                    . PHP_EOL
                    . $match[1]
                    . 'preg_replace_callback('
                    . $arg1
                    . 'function ($m) { return '
                    .    str_replace('$1','$m',
$match[1])
                    .      trim($arg2, '"\'') .
'; }, '
                    .      trim($last, ',');
            return str_replace('$1', '$m',
$arg3);
        },

            // end array
            ],

            // this is the target of the
transformations
            $contents
        );
        // return the result as a string
        return implode('', $result);
}
```

How it works...

To use the converter, run the following code from the command line. You'll need to supply the filename of the PHP 5 code to be scanned as an argument.

This block of code, `chap_01_php5_to_php7_code_converter.php`, run from the command line, calls the converter:

```
<?php
// get filename to scan from command line
$filename = $argv[1] ?? '';

if (!$filename) {
    echo 'No filename provided' . PHP_EOL;
    echo 'Usage: ' . PHP_EOL;
    echo __FILE__ . ' <filename>' . PHP_EOL;
    exit;
}

// setup class autoloading
require __DIR__ .
'/../Application/Autoload/Loader.php';

// add current directory to the path
Application\Autoload\Loader::init(__DIR__ .
'/..');

// get "deep scan" class
$convert = new Application\Parse\Convert();
echo $convert->scan($filename);
echo PHP_EOL;
```

See also

For more information on backwards incompatible changes, please refer to http://php.net/manual/en/migration70.incompatible.php.

Chapter 2. Using PHP 7 High Performance Features

In this chapter we will discuss and understand the syntax differences between PHP 5 and PHP 7, featuring the following recipes:

- Understanding the abstract syntax tree
- Understanding differences in parsing
- Understanding differences in `foreach()` handling
- Improving performance using PHP 7 enhancements
- Iterating through a massive file
- Uploading a spreadsheet into a database
- Recursive directory iterator

Introduction

In this chapter we will move directly into PHP 7, presenting recipes that take advantage of new high performance features. First, however, we will present a series of smaller recipes that serve to illustrate the differences in how PHP 7 handles parameter parsing, syntax, a `foreach()` loop, and other enhancements. Before we go into depth in this chapter, let's discuss some basic differences between PHP 5 and PHP 7.

PHP 7 introduced a new layer referred to as the **Abstract Syntax Tree** (**AST**), which effectively decouples the parsing process from the pseudo-compile process. Although the new layer has little or no impact on performance, it gives the language a new uniformity of syntax, which was not possible previously.

Another benefit of AST is the process of *dereferencing*. Dereferencing, simply put, refers to the ability to immediately acquire a property from, or run a method of, an object, immediately access an array element, and immediately execute a callback. In PHP 5

such support was inconsistent and incomplete. To execute a callback, for example, often you would first need to assign the callback or anonymous function to a variable, and then execute it. In PHP 7 you can execute it immediately.

Understanding the abstract syntax tree

As a developer, it might be of interest for you to be free from certain syntax restrictions imposed in PHP 5 and earlier. Aside from the uniformity of the syntax mentioned previously, where you'll see the most improvement in syntax is the ability to call any return value, which is **callable** by simply appending an extra set of parentheses. Also, you'll be able to directly access any array element when the return value is an array.

How to do it...

1. Any function or method that returns a callback can be immediately executed by simply appending parentheses `()` (with or without parameters). An element can be immediately dereferenced from any function or method that returns an array by simply indicating the element using square brackets `[];`. In the short (but trivial) example shown next, the function `test()` returns an array. The array contains six anonymous functions. `$a` has a value of `$t`. `$$a` is interpreted as `$test`:

```
function test()
{
    return [
        1 => function () { return [
            1 => function ($a) { return 'Level
1/1:' . ++$a; },
            2 => function ($a) { return 'Level
1/2:' . ++$a; },
        ];},
        2 => function () { return [
            1 => function ($a) { return 'Level
2/1:' . ++$a; },
            2 => function ($a) { return 'Level
2/2:' . ++$a; },
        ];}
    ];
```

```
}

$a = 't';
$t = 'test';
echo $$a()[1]()[2](100);
```

2. AST allows us to issue the `echo $$a()[1]()[2](100)` command. This is parsed left-to-right, which executes as follows:

    - `$$a()` interprets as `test()`, which returns an array
    - `[1]` dereferences array element `1`, which returns a callback
    - `()` executes this callback, which returns an array of two elements
    - `[2]` dereferences array element `2`, which returns a callback
    - `(100)` executes this callback, supplying a value of `100`, which returns `Level 1/2:101`

    > **Tip**
    >
    > Such a statement is not possible in PHP 5: a parse error would be returned.

3. The following is a more substantive example that takes advantage of AST syntax to define a data filtering and validating class. First of all, we define the `Application\Web\Securityclass`. In the constructor, we build and define two arrays. The first array consists of filter callbacks. The second array has validation callbacks:

```
public function __construct()
  {
    $this->filter = [
      'striptags' => function ($a) { return
strip_tags($a); },
      'digits'    => function ($a) { return
preg_replace(
      '/[^0-9]/', '', $a); },
      'alpha'     => function ($a) { return
preg_replace(
```

```
        '/[^A-Z]/i', '', $a); }
      ];
      $this->validate = [
        'alnum'  => function ($a) { return
ctype_alnum($a); },
        'digits' => function ($a) { return
ctype_digit($a); },
        'alpha'  => function ($a) { return
ctype_alpha($a); }
      ];
    }
```

4. We want to be able to call this functionality in a *developer-friendly* manner. Thus, if we want to filter digits, then it would be ideal to run a command such as this:

```
$security->filterDigits($item));
```

5. To accomplish this we define the magic method `__call()`, which gives us access to non-existent methods:

```
public function __call($method, $params)
{

  preg_match('/^(filter|validate)(.*?)$/i',
$method, $matches);
  $prefix   = $matches[1] ?? '';
  $function = strtolower($matches[2] ?? '');
  if ($prefix && $function) {
    return $this->$prefix[$function]
($params[0]);
  }
  return $value;
}
```

We use `preg_match()` to match the `$method` param against `filter` or `validate`. The second sub-match will then be converted into an array key in either `$this->filter` or `$this->validate`. If both sub-patterns produce a sub-match, we assign the first sub-match to `$prefix`, and the second sub-match `$function`. These end up as variable parameters when executing the appropriate callback.

Tip

Don't go too crazy with this stuff!

As you revel in your new found freedom of expression, made possible by AST, be sure to keep in mind that the code you end up writing could, in the long run, be extremely cryptic. This will ultimately cause long-term maintenance problems.

How it works...

First of all, we create a sample file, `chap_02_web_filtering_ast_example.php`, to take advantage of the autoloading class defined in Chapter 1, *Building the Foundation*, to obtain an instance of `Application\Web\Security`:

```
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
$security = new Application\Web\Security();
```

Next, we define a block of test data:

```
$data = [
    '<ul><li>Lots</li><li>of</li><li>Tags</li>
</ul>',
    12345,
    'This is a string',
    'String with number 12345',
];
```

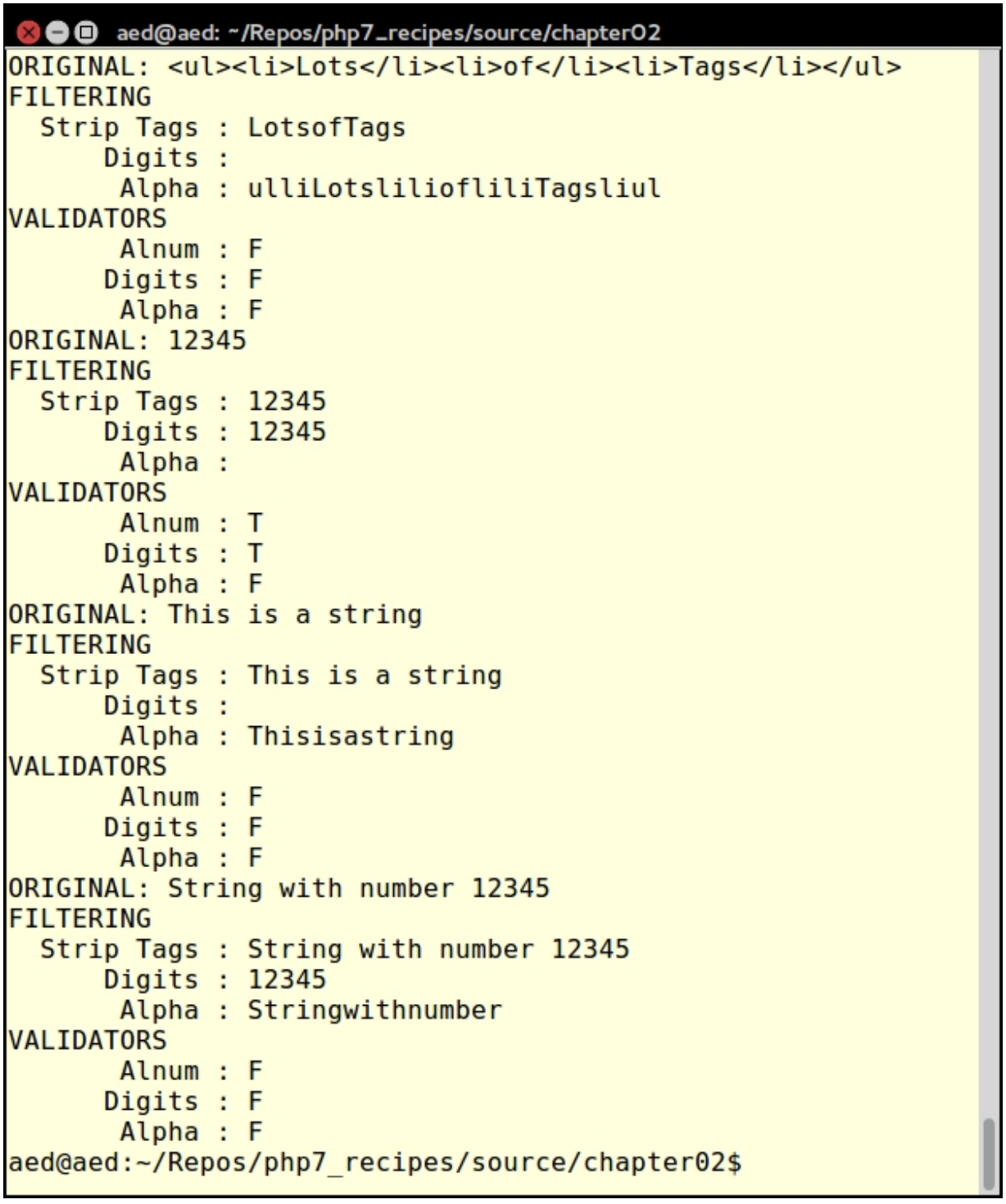
Finally, we call each filter and validator for each item of test data:

```
foreach ($data as $item) {
  echo 'ORIGINAL: ' . $item . PHP_EOL;
  echo 'FILTERING' . PHP_EOL;
  printf('%12s : %s' . PHP_EOL,'Strip Tags',
$security->filterStripTags($item));
  printf('%12s : %s' . PHP_EOL, 'Digits',
$security->filterDigits($item));
```

```
  printf('%12s : %s' . PHP_EOL, 'Alpha',
$security->filterAlpha($item));

  echo 'VALIDATORS' . PHP_EOL;
  printf('%12s : %s' . PHP_EOL, 'Alnum',
  ($security->validateAlnum($item))  ? 'T' : 'F');
  printf('%12s : %s' . PHP_EOL, 'Digits',
  ($security->validateDigits($item)) ? 'T' : 'F');
  printf('%12s : %s' . PHP_EOL, 'Alpha',
  ($security->validateAlpha($item))  ? 'T' : 'F');
}
```

Here is the output of some input strings:



See also

For more information on AST, please consult the RFC that addresses the **Abstract Syntax Tree**, which can be viewed at

https://wiki.php.net/rfc/abstract_syntax_tree.

Understanding differences in parsing

In PHP 5, expressions on the right side of an assignment operation were parsed *right-to-left*. In PHP 7, parsing is consistently *left-to-right*.

How to do it...

1. A variable-variable is a way of indirectly referencing a value. In the following example, first `$$foo` is interpreted as `${$bar}`. The final return value is thus the value of `$bar` instead of the direct value of `$foo` (which would be `bar`):

```
$foo = 'bar';
$bar = 'baz';
echo $$foo; // returns  'baz';
```

2. In the next example we have a variable-variable `$$foo`, which references a multi-dimensional array with a `bar key` and a `baz sub-key`:

```
$foo = 'bar';
$bar = ['bar' => ['baz' => 'bat']];
// returns 'bat'
echo $$foo['bar']['baz'];
```

3. In PHP 5, parsing occurs right-to-left, which means the PHP engine would be looking for an `$foo array`, with a `bar key` and a `baz`. sub-key The return value of the element would then be interpreted to obtain the final value `${$foo['bar']['baz']}`.
4. In PHP 7, however, parsing is consistently left-to-right, which means that `$foo` is interpreted first `($$foo)['bar']['baz']`.
5. In the next example you can see that `$foo->$bar['bada']` is interpreted quite differently in PHP 5, compared with PHP 7. In the following example, PHP 5 would first interpret `$bar['bada']`, and reference this return value against a `$foo object instance`.

In PHP 7, on the other hand, parsing is consistently left-to-right, which means that `$foo->$bar` is interpreted first, and expects an array with a `bada element`. You will also note, incidentally, that this example uses the PHP 7 *anonymous class* feature:

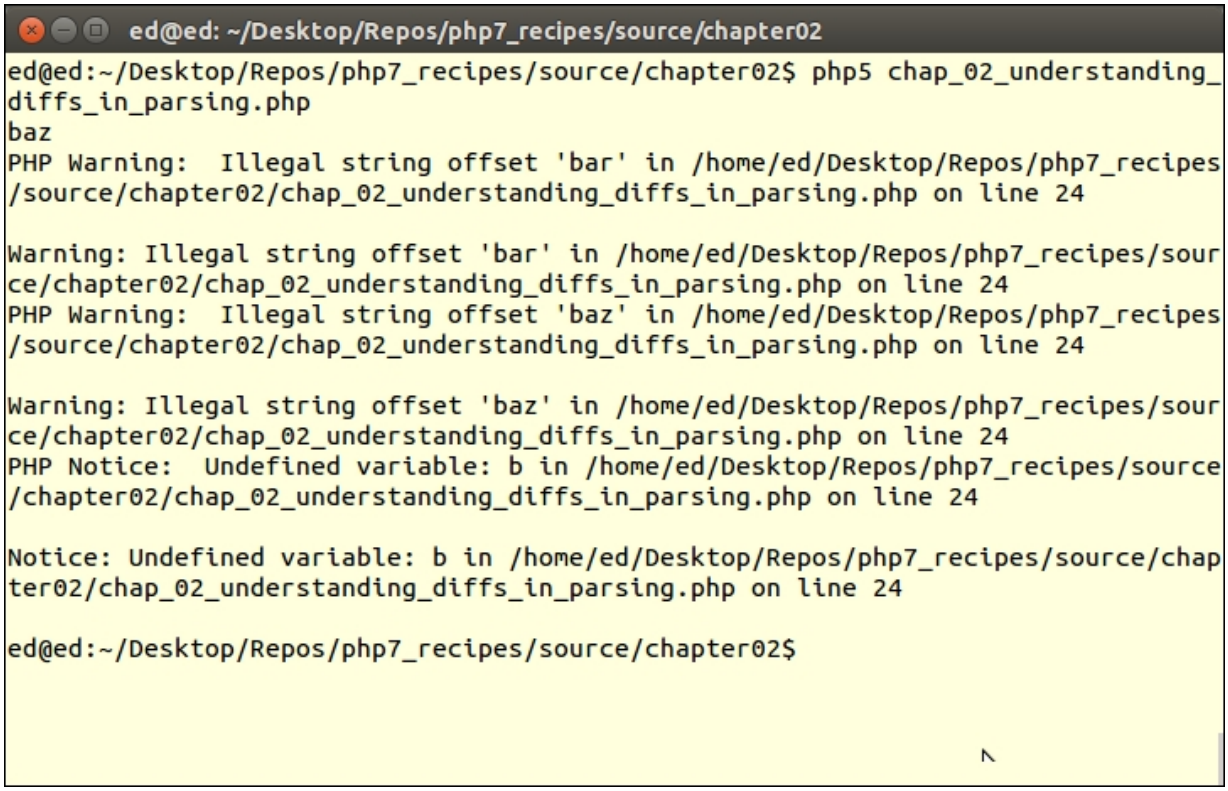
```
// PHP 5: $foo->{$bar['bada']}
// PHP 7: ($foo->$bar)['bada']
$bar = 'baz';
// $foo = new class
{
    public $baz = ['bada' => 'boom'];
};
// returns 'boom'
echo $foo->$bar['bada'];
```

6. The last example is the same as the one immediately above, except that the return value is expected to be a callback, which is then immediately executed as follows:

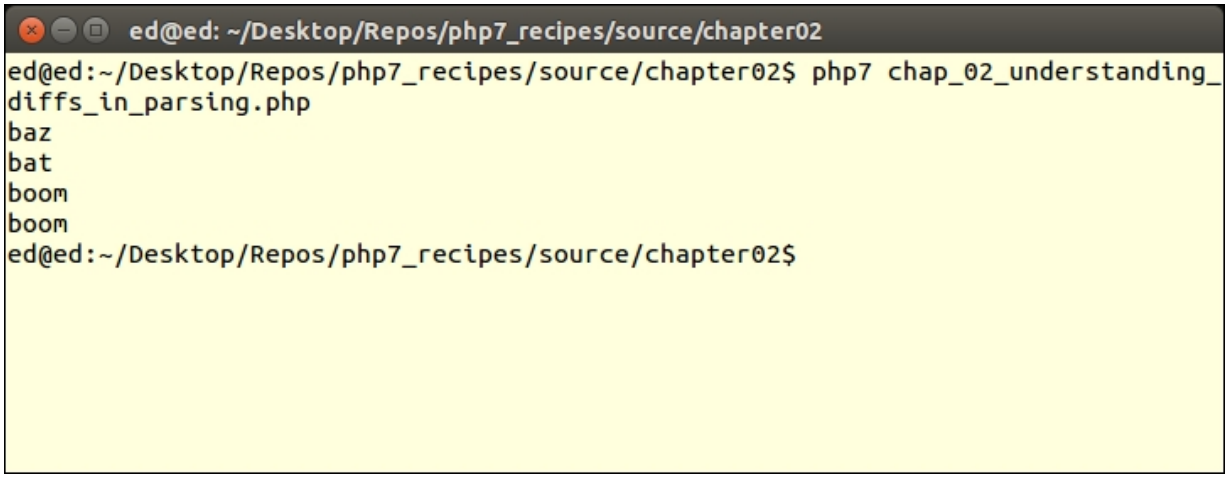
```
// PHP 5: $foo->{$bar['bada']}()
// PHP 7: ($foo->$bar)['bada']()
$bar = 'baz';
// NOTE: this example uses the new PHP 7
anonymous class feature
$foo = new class
{
     public function __construct()
    {
        $this->baz = ['bada' => function () {
return 'boom'; }];
    }
};
// returns 'boom'
echo $foo->$bar['bada']();
```

How it works...

Place the code examples illustrated in 1 and 2 into a single PHP file that you can call `chap_02_understanding_diffs_in_parsing.php`. Execute the script first using PHP 5, and you will notice that a series of errors will result, as follows:



The reason for the errors is that PHP 5 parses inconsistently, and arrives at the wrong conclusion regarding the state of the variable variables requested (as previously mentioned). Now you can go ahead and add the remaining examples, as shown in steps 5 and 6. If you then run this script in PHP 7, the results described will appear, as shown here:



See also

For more information on parsing, please consult the RFC, which addresses **Uniform Variable Syntax**, and can be viewed at https://wiki.php.net/rfc/uniform_variable_syntax.

Understanding differences in foreach() handling

In certain relatively obscure circumstances, the behavior of code inside a `foreach()` loop will vary between PHP 5 and PHP 7. First of all, there have been massive internal improvements, which means that in terms of sheer speed, processing inside the `foreach()` loop will be much faster running under PHP 7, compared with PHP 5. Problems that are noticed in PHP 5 include the use of `current()`, and `unset()` on the array inside the `foreach()` loop. Other problems have to do with passing values by reference while manipulating the array itself.

How to do it...

1. Consider the following block of code:

```
$a = [1, 2, 3];
foreach ($a as $v) {
  printf("%2d\n", $v);
  unset($a[1]);
}
```

2. In both PHP 5 and 7, the output would appear as follows:

```
 1
 2
 3
```

3. If you add an assignment before the loop, however, the behavior changes:

```
$a = [1, 2, 3];
$b = &$a;
foreach ($a as $v) {
  printf("%2d\n", $v);
  unset($a[1]);
}
```

4. Compare the output of PHP 5 and 7:

PHP 5	**PHP 7**
1 **3**	**1** **2** **3**

5. Working with functions that reference the internal array pointer also caused inconsistent behavior in PHP 5. Take the following code example:

```
$a = [1,2,3];
foreach($a as &$v) {
    printf("%2d - %2d\n", $v, current($a));
}
```

> ## Tip
>
> Every array has an internal pointer to its `current` element starting from `1`, `current()` returns the current element in an array.

6. Notice that the output running in PHP 7 is normalized and consistent:

PHP 5	**PHP 7**

PHP 5	PHP 7
1 - 2 **2 - 3** **3 - 0**	**1 - 1** **2 - 1** **3 - 1**

7. Adding a new element inside the `foreach()` loop, once the array iteration by reference is complete, is also problematic in PHP 5. This behavior has been made consistent in PHP 7. The following code example demonstrates this:

```
$a = [1];
foreach($a as &$v) {
    printf("%2d -\n", $v);
    $a[1]=2;
}
```

8. We will observe the following output:

PHP 5	PHP 7
1 -	**1 -** **2-**

9. Another example of bad PHP 5 behavior addressed in PHP 7, during array iteration by reference, is the use of functions that modify the array, such as `array_push()`, `array_pop()`, `array_shift()`, and `array_unshift()`.

Have a look at this example:

```
$a=[1,2,3,4];
foreach($a as &$v) {
    echo "$v\n";
    array_pop($a);
}
```

10. You will observe the following output:

PHP 5	**PHP 7**
1 **2** **1** **1**	**1** **2**

11. Finally, we have a case where you are iterating through an array by reference, with a nested `foreach()` loop, which itself iterates on the same array by reference. In PHP 5 this construct simply did not work. In PHP 7 this has been fixed. The following block of code demonstrates this behavior:

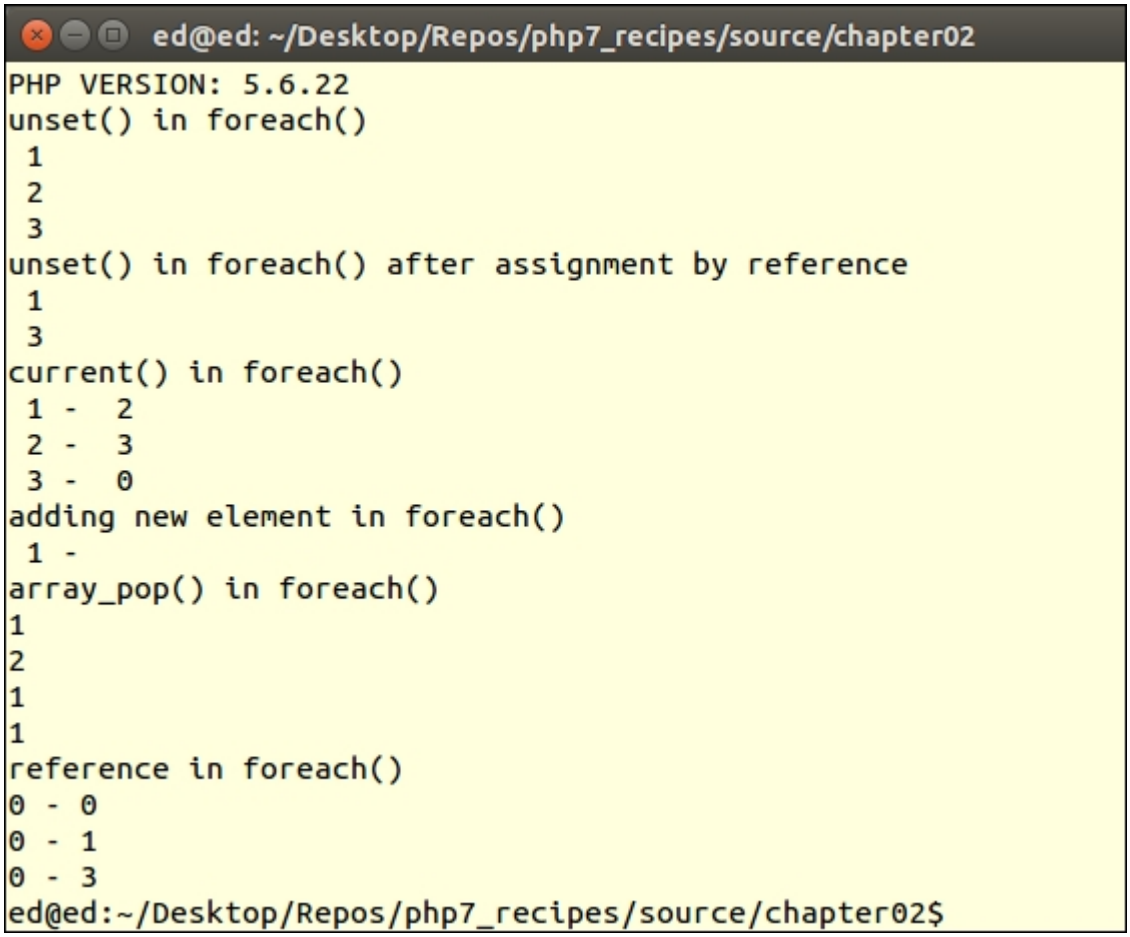
```
$a = [0, 1, 2, 3];
foreach ($a as &$x) {
       foreach ($a as &$y) {
         echo "$x - $y\n";
         if ($x == 0 && $y == 1) {
           unset($a[1]);
           unset($a[2]);
         }
       }
}
```

12. And here is the output:

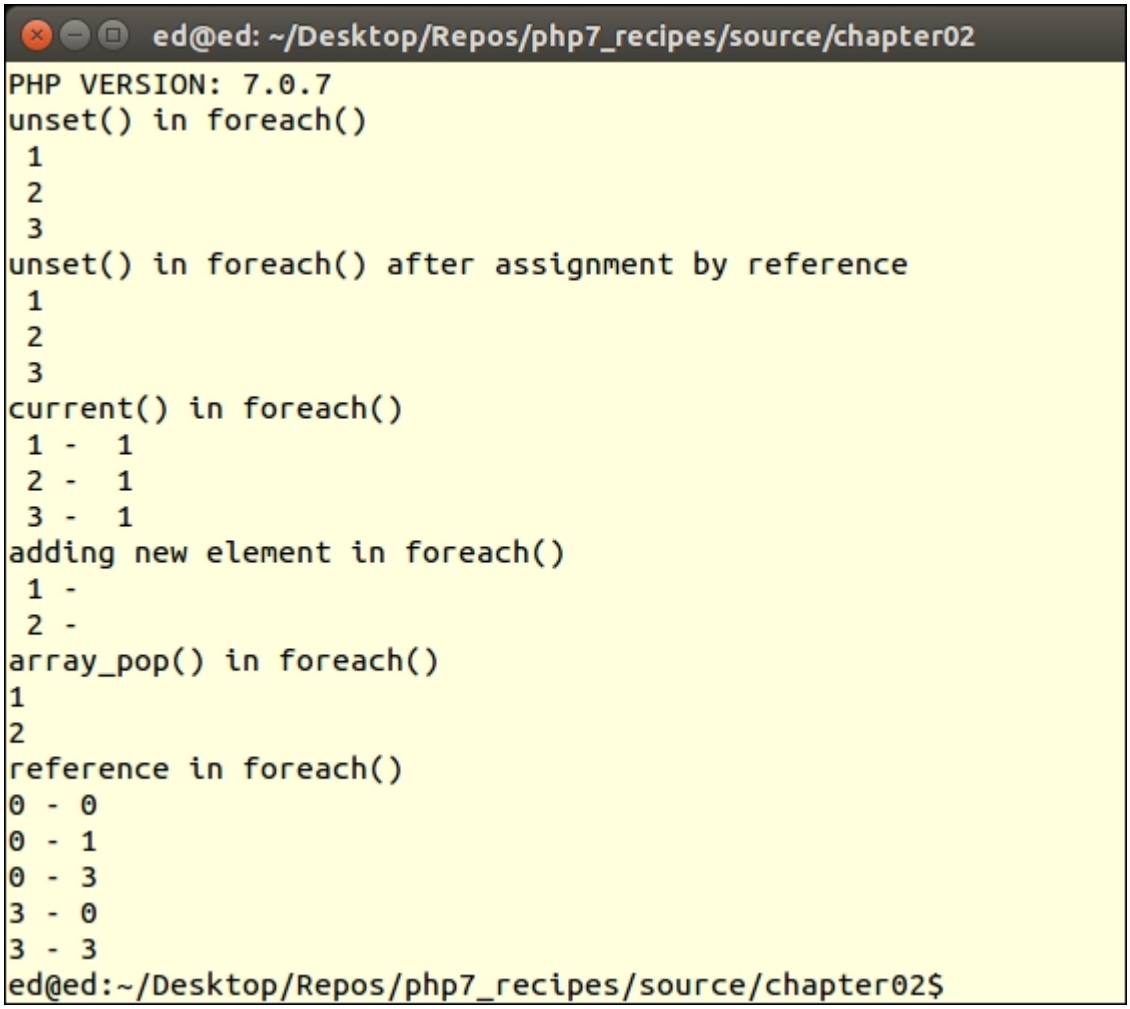
PHP 5	PHP 7
0 - 0 **0 - 1** **0 - 3**	**0 - 0** **0 - 1** **0 - 3** **3 - 0** **3 -3**

How it works...

Add these code examples to a single PHP file, `chap_02_foreach.php`. Run the script under PHP 5 from the command line. The expected output is as follows:



Run the same script under PHP 7 and notice the difference:



See also

For more information, consult the RFC addressing this issue, which was accepted. A write-up on this RFC can be found at: https://wiki.php.net/rfc/php7_foreach.

Improving performance using PHP 7 enhancements

One trend that developers are taking advantage of is the use of **anonymous functions**. One classic problem, when dealing with anonymous functions, is to write them in such a way that any object can be bound to `$this` and the function will still work. The approach used in PHP 5 code is to use `bindTo()`. In PHP 7, a new method, `call()`, was added, which offers similar functionality, but vastly improved performance.

How to do it...

To take advantage of `call()`, execute an anonymous function in a lengthy loop. In this example, we will demonstrate an anonymous function, that scans through a log file, identifying IP addresses sorted by how frequently they appear:

1. First, we define a `Application\Web\Access class`. In the constructor, we accept a filename as an argument. The log file is opened as an `SplFileObject` and assigned to `$this->log`:

```
Namespace Application\Web;

use Exception;
use SplFileObject;
class Access
{
  const ERROR_UNABLE = 'ERROR: unable to open file';
  protected $log;
  public $frequency = array();
  public function __construct($filename)
  {
    if (!file_exists($filename)) {
      $message = __METHOD__ . ' : ' . self::ERROR_UNABLE .
PHP_EOL;
      $message .= strip_tags($filename) . PHP_EOL;
      throw new Exception($message);
    }
    $this->log = new SplFileObject($filename, 'r');
  }
```

2. Next, we define a generator that iterates through the file, line by line:

```
public function fileIteratorByLine()
{
  $count = 0;
  while (!$this->log->eof()) {
    yield $this->log->fgets();
    $count++;
  }
```

```
    return $count;
  }
```

3. Finally, we define a method that looks for, and extracts as a sub-match, an IP address:

```
public function getIp($line)
{
  preg_match('/(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})/',
$line, $match);
  return $match[1] ?? '';
  }
}
```

How it works...

First of all, we define a calling program, `chap_02_performance_using_php7_enchancement_call.php`, that takes advantage of the autoloading class defined in Chapter 1 , *Building a Foundation,* to obtain an instance of `Application\Web\Access`:

```
define('LOG_FILES', '/var/log/apache2/*access*.log');
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
```

Next we define the anonymous function, which processes one line in the log file. If an IP address is detected, it becomes a key in the `$frequency array`, and the current value for this key is incremented:

```
// define functions
$freq = function ($line) {
  $ip = $this->getIp($line);
  if ($ip) {
    echo '.';
    $this->frequency[$ip] =
    (isset($this->frequency[$ip])) ? $this->frequency[$ip] + 1 :
1;
  }
};
```

We then loop through the iteration of lines in each log file found, processing IP addresses:

```
foreach (glob(LOG_FILES) as $filename) {
  echo PHP_EOL . $filename . PHP_EOL;
  // access class
  $access = new Application\Web\Access($filename);
  foreach ($access->fileIteratorByLine() as $line) {
    $freq->call($access, $line);
```

```
    }
}
```

Tip

You can actually do the same thing in PHP 5. Two lines of code are required, however:

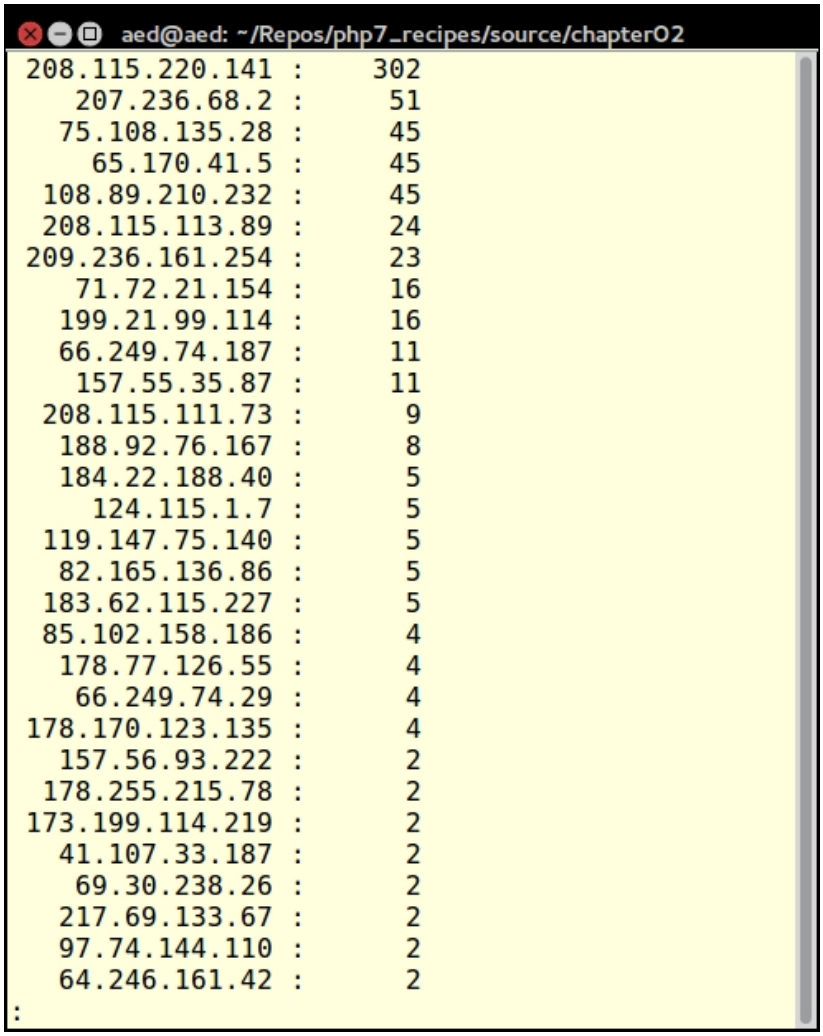
```
$func = $freq->bindTo($access);
$func($line);
```

Performance is 20% to 50% slower than using `call()` in PHP 7.

Finally, we reverse-sort the array, but maintain the keys. The output is produced in a simple `foreach()` loop:

```
arsort($access->frequency);
foreach ($access->frequency as $key => $value) {
  printf('%16s : %6d' . PHP_EOL, $key, $value);
}
```

The output will vary depending on which `access.log` you process. Here is a sample:



There's more...

Many of the PHP 7 performance improvements have nothing to do with new features and functions. Rather, they take the form of internal improvements, which are *invisible* until you start running your programs. Here is a short list of improvements that fall into this category:

Feature	**More info:**	**Notes**

Feature	More info:	Notes
Fast parameter parsing	https://wiki.php.net/rfc/fast_zpp	In PHP 5, parameters provided to functions have to be parsed for every single function call. The parameters were passed in as a string, and parsed in a manner similar to the `scanf()` function. In PHP 7 this process has been optimized and made much more efficient, resulting in a significant performance improvement. The improvement is difficult to measure, but seems to be in the region of 6%.
PHP NG	https://wiki.php.net/rfc/phpng	The PHP **NG** (**Next Generation**) initiative represents a rewrite of

Feature	More info:	Notes
		most of the PHP language. It retains existing functionality, but involves any and all time-savings and efficiency measures imaginable. Data structures have been compacted, and memory is used more efficiently. Just one change, which affects array handling, for example, has resulted in a significant performance increase, while at the same time greatly reducing memory usage.

Feature	More info:	Notes
Removing dead weight	https://wiki.php.net/rfc/removal_of_dead_sapis_and_exts	There were approximately two dozen extensions that fell into one of these categories: deprecated, no longer maintained, unmaintained dependencies, or not ported to PHP 7. A vote by the group of core developers determined to remove about 2/3 or the extensions on the "short list". This results in reduced overhead and faster overall future development of the PHP language.

Iterating through a massive file

Functions such as `file_get_contents()` and `file()` are quick and easy to use however, owing to memory limitations, they quickly cause problems when dealing with massive files. The default setting for the `php.ini memory_limit` setting is 128 megabytes. Accordingly, any file larger than this will not be loaded.

Another consideration when parsing through massive files is how quickly does your function or class method produce output? When producing user output, for example, although it might at first glance seem better to accumulate output in an array. You would then output it all at once for improved efficiency. Unfortunately, this might have an adverse impact on the user experience. It might be better to create a **generator**, and use the `yield keyword` to produce immediate results.

How to do it...

As mentioned before, the `file*` functions (that is, `file_get_contents()`), are not suitable for large files. The simple reason is that these functions, at one point, have the entire contents of the file represented in memory. Accordingly, the focus of this recipe will be on the `f*` functions (that is, `fopen()`).

In a slight twist, however, instead of using the `f*` functions directly, instead we will use the `SplFileObject` class, which is included in the **SPL** (**Standard PHP Library**):

1. First, we define a `Application\Iterator\LargeFile` class with the appropriate properties and constants:

```
namespace Application\Iterator;

use Exception;
use InvalidArgumentException;
use SplFileObject;
```

```
use NoRewindIterator;

class LargeFile
{
  const ERROR_UNABLE = 'ERROR: Unable to open
file';
  const ERROR_TYPE   = 'ERROR: Type must be
"ByLength", "ByLine" or "Csv"';
  protected $file;
  protected $allowedTypes = ['ByLine',
'ByLength', 'Csv'];
```

2. We then define a `__construct()` method that accepts a filename as an argument and populates the `$file` property with an `SplFileObject` instance. This is also a good place to throw an exception if the file does not exist:

```
public function __construct($filename, $mode =
'r')
{
  if (!file_exists($filename)) {
    $message = __METHOD__ . ' : ' .
self::ERROR_UNABLE . PHP_EOL;
    $message .= strip_tags($filename) .
PHP_EOL;
    throw new Exception($message);
  }
  $this->file = new SplFileObject($filename,
$mode);
}
```

3. Next we define a method `fileIteratorByLine()method` which uses `fgets()` to read one line of the file at a time. It's not a bad idea to create a complimentary `fileIteratorByLength()` method that does the same thing but uses `fread()` instead. The method that uses `fgets()` would be suitable for text files that include linefeeds. The other method could be used if parsing a large binary file:

```
protected function fileIteratorByLine()
{
  $count = 0;
  while (!$this->file->eof()) {
    yield $this->file->fgets();
    $count++;
```

```
    }
    return $count;
  }

  protected function
  fileIteratorByLength($numBytes = 1024)
  {
    $count = 0;
    while (!$this->file->eof()) {
      yield $this->file->fread($numBytes);
      $count++;
    }
    return $count;
  }
```

4. Finally, we define a `getIterator()` method that returns a `NoRewindIterator()` instance. This method accepts as arguments either `ByLine` or `ByLength`, which refer to the two methods defined in the previous step. This method also needs to accept `$numBytes` in case `ByLength` is called. The reason we need a `NoRewindIterator()` instance is to enforce the fact that we're reading through the file only in one direction in this example:

```
public function getIterator($type = 'ByLine',
$numBytes = NULL)
{
  if(!in_array($type, $this->allowedTypes)) {
    $message = __METHOD__ . ' : ' .
self::ERROR_TYPE . PHP_EOL;
    throw new
InvalidArgumentException($message);
  }
  $iterator = 'fileIterator' . $type;
  return new NoRewindIterator($this-
>$iterator($numBytes));
}
```

How it works...

First of all, we take advantage of the autoloading class defined in Chapter 1, *Building a Foundation*, to obtain an instance of

`Application\Iterator\LargeFile` in a calling program, `chap_02_iterating_through_a_massive_file.php`:

```
define('MASSIVE_FILE',
'/../data/files/war_and_peace.txt');
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
```

Next, inside a `try {...} catch () {...}` block, we get an instance of a `ByLine` iterator:

```
try {
  $largeFile = new
Application\Iterator\LargeFile(__DIR__ .
MASSIVE_FILE);
  $iterator = $largeFile->getIterator('ByLine');
```

We then provide an example of something useful to do, in this case, defining an average of words per line:

```
$words = 0;
foreach ($iterator as $line) {
  echo $line;
  $words += str_word_count($line);
}
echo str_repeat('-', 52) . PHP_EOL;
printf("%-40s : %8d\n", 'Total Words', $words);
printf("%-40s : %8d\n", 'Average Words Per Line',
($words / $iterator->getReturn()));
echo str_repeat('-', 52) . PHP_EOL;
```

We then end the `catch` block:

```
} catch (Throwable $e) {
  echo $e->getMessage();
}
```

The expected output (too large to show here!) shows us that there are 566,095 words in the project Gutenberg version of *War and*

Peace. Also, we find the average number of words per line is eight.

Uploading a spreadsheet into a database

Although PHP does not have any direct capability to read a specific spreadsheet format (that is, XLSX, ODS, and so on), it does have the ability to read **(CSV Comma Separated Values**) files. Accordingly, in order to process customer spreadsheets, you will need to either ask them to furnish their files in CSV format, or you will need to perform the conversion yourself.

Getting ready...

When uploading a spreadsheet (that is, a CSV file) into a database, there are three major considerations:

- Iterating through a (potentially) massive file
- Extracting each spreadsheet row into a PHP array
- Inserting the PHP array into the database

Massive file iteration will be handled using the preceding recipe. We will use the `fgetcsv()` function to convert a CSV row into a PHP array. Finally, we will use the **(PDO PHP Data Objects**) class to make a database connection and perform the insert.

How to do it...

1. First, we define a `Application\Database\Connection` class that creates a PDO instance based on a set of parameters supplied to the constructor:

```
<?php
  namespace Application\Database;

  use Exception;
  use PDO;
```

```
  class Connection
  {
    const ERROR_UNABLE = 'ERROR: Unable to
create database connection';
    public $pdo;

    public function __construct(array $config)
    {
      if (!isset($config['driver'])) {
        $message = __METHOD__ . ' : ' .
self::ERROR_UNABLE . PHP_EOL;
        throw new Exception($message);
    }
    $dsn = $config['driver']
    . ':host=' . $config['host']
    . ';dbname=' . $config['dbname'];
    try {
      $this->pdo = new PDO($dsn,
      $config['user'],
      $config['password'],
      [PDO::ATTR_ERRMODE =>
$config['errmode']]);
    } catch (PDOException $e) {
      error_log($e->getMessage());
    }
  }

}
```

2. We then incorporate an instance of `Application\Iterator\LargeFile`. We add a new method to this class that is designed to iterate through CSV files:

```
protected function fileIteratorCsv()
{
  $count = 0;
  while (!$this->file->eof()) {
    yield $this->file->fgetcsv();
    $count++;
  }
  return $count;
}
```

3. We also need to add `Csv` to the list of allowed iterator methods:

```
  const ERROR_UNABLE = 'ERROR: Unable to open
file';
```

```
    const ERROR_TYPE   = 'ERROR: Type must be
"ByLength", "ByLine" or "Csv"';

    protected $file;
    protected $allowedTypes = ['ByLine',
'ByLength', 'Csv'];
```

How it works...

First we define a config file, `/path/to/source/config/db.config.php`, that contains database connection parameters:

```
<?php
return [
  'driver'   => 'mysql',
  'host'     => 'localhost',
  'dbname'   => 'php7cookbook',
  'user'     => 'cook',
  'password' => 'book',
  'errmode'  => PDO::ERRMODE_EXCEPTION,
];
```

Next, we take advantage of the autoloading class defined in Chapter 1, *Building a Foundation*, to obtain an instance of `Application\Database\Connection` and `Application\Iterator\LargeFile`, defining a calling program, `chap_02_uploading_csv_to_database.php`:

```
define('DB_CONFIG_FILE',
'/../data/config/db.config.php');
define('CSV_FILE',
'/../data/files/prospects.csv');
require __DIR__ .
'/../../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
```

After that, we set up a `try {...} catch () {...}` block, which catches `Throwable`. This allows us to `catch` both exceptions and errors:

```
try {
  // code goes here
} catch (Throwable $e) {
  echo $e->getMessage();
}
```

Inside the `try {...} catch () {...}` block we get an instance of the connection and large file iterator classes:

```
$connection = new Application\Database\Connection(
include __DIR__ . DB_CONFIG_FILE);
$iterator  = (new
Application\Iterator\LargeFile(__DIR__ .
CSV_FILE))
->getIterator('Csv');
```

We then take advantage of the PDO prepare/execute functionality. The SQL for the prepared statement uses `?` to represent values that are supplied in a loop:

```
$sql = 'INSERT INTO `prospects` '
  .
'(`id`,`first_name`,`last_name`,`address`,`city`,`
state_province`,'
  .
'`postal_code`,`phone`,`country`,`email`,`status`,
`budget`,`last_updated`) '
  . ' VALUES (?,?,?,?,?,?,?,?,?,?,?,?,?)';
$statement = $connection->pdo->prepare($sql);
```

We then use `foreach()` to loop through the file iterator. Each `yield` statement produces an array of values that represents a row in the database. We can then use these values with `PDOStatement::execute()` to execute the prepared statement, inserting the row of values into the database:

```
foreach ($iterator as $row) {
  echo implode(',', $row) . PHP_EOL;
  $statement->execute($row);
}
```

You can then examine the database to verify that the data was successfully inserted.

Recursive directory iterator

Getting a list of files in a directory is extremely easy. Traditionally, developers have used the `glob()` function for this purpose. To recursively get a list of all files and directories from a specific point in a directory tree is more problematic. This recipe takes advantage of an **(SPL Standard PHP Library**) class `RecursiveDirectoryIterator`, which will serve this purpose admirably.

What this class does is to parse the directory tree, finding the first child, then it follows the branches, until there are no more children, and then it stops! Unfortunately this is not what we want. Somehow we need to get the `RecursiveDirectoryIterator` to continue parsing every tree and branch, from a given starting point, until there are no more files or directories. It so happens there is a marvelous class, `RecursiveIteratorIterator`, that does exactly that. By wrapping `RecursiveDirectoryIterator` inside `RecursiveIteratorIterator,` we accomplish a complete traversal of any directory tree.

> ## Tip
>
> **Warning!**
>
> Be very careful where you start the filesystem traversal. If you start at the root directory, you could end up crashing your server as the recursion process will not stop until all files and directories have been located!

How to do it...

1. First, we define a `Application\Iterator\Directory` class that defines the appropriate properties and constants and uses external classes:

```
namespace Application\Iterator;

use Exception;
use RecursiveDirectoryIterator;
use RecursiveIteratorIterator;
use RecursiveRegexIterator;
use RegexIterator;

class Directory
{

  const ERROR_UNABLE = 'ERROR: Unable to read
directory';

  protected $path;
  protected $rdi;
  // recursive directory iterator
```

2. The constructor creates a `RecursiveDirectoryIterator` instance inside `RecursiveIteratorIterator` based on a directory path:

```
public function __construct($path)
{
  try {
    $this->rdi = new
RecursiveIteratorIterator(
      new RecursiveDirectoryIterator($path),
      RecursiveIteratorIterator::SELF_FIRST);
  } catch (\Throwable $e) {
    $message = __METHOD__ . ' : ' .
self::ERROR_UNABLE . PHP_EOL;
    $message .= strip_tags($path) . PHP_EOL;
    echo $message;
    exit;
  }
}
```

3. Next, we decide what to do with the iteration. One possibility is to mimic the output of the Linux `ls -l -R` command. Notice that we use the `yield` keyword, effectively making this method into a **Generator**, which can then be called from the outside. Each object produced by the directory iteration is an SPL `FileInfo` object, which can give us useful information on the file. Here is how this method might look:

```
public function ls($pattern = NULL)
{
  $outerIterator = ($pattern)
  ? $this->regex($this->rdi, $pattern)
  : $this->rdi;
  foreach($outerIterator as $obj){
    if ($obj->isDir()) {
      if ($obj->getFileName() == '..') {
        continue;
      }
      $line = $obj->getPath() . PHP_EOL;
    } else {
      $line = sprintf('%4s %1d %4s %4s %10d
%12s %-40s' . PHP_EOL,
      substr(sprintf('%o', $obj->getPerms()),
-4),
      ($obj->getType() == 'file') ? 1 : 2,
      $obj->getOwner(),
      $obj->getGroup(),
      $obj->getSize(),
      date('M d Y H:i', $obj->getATime()),
      $obj->getFileName());
    }
    yield $line;
  }
}
```

4. You may have noticed that the method call includes a file pattern. We need a way of filtering the recursion to only include files that match. There is another iterator available from the SPL that perfectly suits this need: the `RegexIterator` class:

```
protected function regex($iterator, $pattern)
{
  $pattern = '!^.' . str_replace('.', '\\.',
$pattern) . '$!';
  return new RegexIterator($iterator,
$pattern);
}
```

5. Finally, here is another method, but this time we will mimic the `dir /s` command:

```
public function dir($pattern = NULL)
{
  $outerIterator = ($pattern)
```

```
            ? $this->regex($this->rdi, $pattern)
            : $this->rdi;
            foreach($outerIterator as $name => $obj){
                yield $name . PHP_EOL;
              }
            }
        }
```

How it works...

First of all, we take advantage of the autoloading class defined in Chapter 1, *Building a Foundation*, to obtain an instance of `Application\Iterator\Directory`, defining a calling program, `chap_02_recursive_directory_iterator.php`:

```
define('EXAMPLE_PATH', realpath(__DIR__ .
'/../'));
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
$directory = new
Application\Iterator\Directory(EXAMPLE_PATH);
```

Then, in a `try {...} catch () {...}` block, we make a call to our two methods, using an example directory path:

```
try {
  echo 'Mimics "ls -l -R" ' . PHP_EOL;
  foreach ($directory->ls('*.php') as $info) {
    echo $info;
  }

  echo 'Mimics "dir /s" ' . PHP_EOL;
  foreach ($directory->dir('*.php') as $info) {
    echo $info;
  }

} catch (Throwable $e) {
  echo $e->getMessage();
}
```

The output for `ls()` will look something like this:

```
aed@aed: ~/Repos/php7_recipes/source/chapter02
Mimics "ls -l -R"
0664 1 1000 1000         20 Dec 31 2015 22:59 info.php
0664 1 1000 1000     438308 Jan 17 2016 16:40 js.php
0664 1 1000 1000     438308 Jan 17 2016 16:39 js.php
0664 1 1000 1000       1334 Jan 16 2016 22:30 City.php
0664 1 1000 1000        671 Jan 16 2016 22:30 Dispatch.php
0664 1 1000 1000        264 Jan 18 2016 23:24 Tree.php
0664 1 1000 1000        154 Jan 18 2016 23:27 Node.php
0664 1 1000 1000        577 Jan 17 2016 16:40 Manager.php
0664 1 1000 1000        117 Jan 17 2016 15:20 Listener.php
0664 1 1000 1000        156 Jan 11 2016 06:01 TestClass.php
0664 1 1000 1000        189 Jan 11 2016 06:01 Class.php
0664 1 1000 1000       2687 Jan 23 2016 20:20 Directory.php
0664 1 1000 1000       1799 Jan 23 2016 03:31 LargeFile.php
0664 1 1000 1000        215 Jan 10 2016 15:19 XmlPath.php
0664 1 1000 1000       4124 Jan 11 2016 06:01 Tree.php
0664 1 1000 1000       8837 Jan 22 2016 20:11 Convert.php
0664 1 1000 1000       6003 Jan 10 2016 15:56 WebTree.php
:
```

The output for `dir()` will appear as follows:

```
aed@aed: ~/Repos/php7_recipes/source/chapter02
Mimics "dir /s"
/home/aed/Repos/php7_recipes/info.php
/home/aed/Repos/php7_recipes/reference/PHP_rfc_uniform_variable_syntax_files/js.php
/home/aed/Repos/php7_recipes/reference/PHP_rfc_abstract_syntax_tree_files/js.php
/home/aed/Repos/php7_recipes/Application/Mvc/City.php
/home/aed/Repos/php7_recipes/Application/Mvc/Dispatch.php
/home/aed/Repos/php7_recipes/Application/Generic/Tree.php
/home/aed/Repos/php7_recipes/Application/Generic/Node.php
/home/aed/Repos/php7_recipes/Application/Event/Manager.php
/home/aed/Repos/php7_recipes/Application/Event/Listener.php
/home/aed/Repos/php7_recipes/Application/Test/TestClass.php
/home/aed/Repos/php7_recipes/Application/Test/Class.php
/home/aed/Repos/php7_recipes/Application/Iterator/Directory.php
/home/aed/Repos/php7_recipes/Application/Iterator/LargeFile.php
/home/aed/Repos/php7_recipes/Application/Parse/XmlPath.php
/home/aed/Repos/php7_recipes/Application/Parse/Tree.php
/home/aed/Repos/php7_recipes/Application/Parse/Convert.php
/home/aed/Repos/php7_recipes/Application/Parse/WebTree.php
:
```

Chapter 3. Working with PHP Functions

In this chapter we will cover the following topics:

- Developing functions
- Hinting at data types
- Using return value data typing
- Using iterators
- Writing your own iterator using generators

Introduction

In this chapter we will consider recipes that take advantage of PHP's **functional programming** capabilities. Functional, or **procedural**, programming is the traditional way PHP code was written prior to the introduction of the first implementation of **object-oriented programming** (**OOP**) in PHP version 4. Functional programming is a programming paradigm—a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. This file can then be included in any future scripts, allowing the functions that are defined to be called at will.

Developing functions

The most difficult aspect is deciding how to break up programming logic into functions. The mechanics of developing a function in PHP, on the other hand, are quite easy. Just use the `function` keyword, give it a name, and follow it with parentheses.

How to do it...

1. The code itself goes inside curly braces as follows:

```
function someName ($parameter)
{
  $result = 'INIT';
  // one or more statements which do something
  // to affect $result
  $result .= ' and also ' . $parameter;
  return $result;
}
```

2. You can define one or more **parameters**. To make one of them optional, simply assign a default value. If you are not sure what default value to assign, use `NULL`:

```
function someOtherName ($requiredParam,
$optionalParam = NULL)
  {
    $result = 0;
    $result += $requiredParam;
    $result += $optionalParam ?? 0;
    return $result;
  }
```

Note

You cannot redefine functions. The only exception is when duplicate functions are defined in separate namespaces. This definition would generate an error:

```
function someTest()
{
  return 'TEST';
}
function someTest($a)
{
  return 'TEST:' . $a;
}
```

3. If you don't know how many parameters will be supplied to your function, or if you want to allow for an infinite number of parameters, use `...` followed by a variable name. All parameters supplied will appear as an array in the variable:

```
function someInfinite(...$params)
{
  // any params passed go into an array
$params
  return var_export($params, TRUE);
}
```

4. A function can call itself. This is referred to as **recursion**. The following function performs a recursive directory scan:

```
function someDirScan($dir)
{
  // uses "static" to retain value of $list
  static $list = array();
  // get a list of files and directories for
this path
  $list = glob($dir . DIRECTORY_SEPARATOR .
'*');
  // loop through
  foreach ($list as $item) {
    if (is_dir($item)) {
      $list = array_merge($list,
someDirScan($item));
    }
  }
  return $list;
}
```

Note

Usage of the `static` keyword inside functions has been in the language for more than 12 years. What `static` does is to initialize the variable once (that is, at the time `static` is declared), and then retain the value between function calls within the same request.

If you need to retain the value of a variable between HTTP requests, make sure the PHP session has been started and store the value in `$_SESSION`.

5. Functions are constrained when defined within a PHP **namespace**. This characteristic can be used to your advantage to provide additional logical separation between libraries of functions. In order to *anchor* the namespace, you need to add the `use` keyword. The following examples are placed in separate namespaces. Notice that even though the function name is the same, there is no conflict as they are not visible to each other.
6. We define `someFunction()` in namespace `Alpha`. We save this to a separate PHP file, `chap_03_developing_functions_namespace_alpha.php`:

```
<?php
namespace Alpha;

function someFunction()
{
  echo __NAMESPACE__ . ':' . __FUNCTION__ .
PHP_EOL;
}
```

7. We then define `someFunction()` in namespace `Beta`. We save this to a separate PHP file, `chap_03_developing_functions_namespace_beta.php`:

```
<?php
namespace Beta;
```

```
function someFunction()
{
  echo __NAMESPACE__ . ':' . __FUNCTION__ .
PHP_EOL;
}
```

8. We can then call `someFunction()` by prefixing the function name with the namespace name:

```
include (__DIR__ . DIRECTORY_SEPARATOR
         .
'chap_03_developing_functions_namespace_alpha.
php');
include (__DIR__ . DIRECTORY_SEPARATOR
         .
'chap_03_developing_functions_namespace_beta.p
hp');
      echo Alpha\someFunction();
      echo Beta\someFunction();
```

> ## Tip
>
> **Best practice**
>
> It is considered best practice to place function libraries (and classes too!) into separate files: one file per namespace, and one class or function library per file.
>
> It is possible to define many classes or function libraries in a single namespace. The only reason you would develop into a separate namespace is if you want to foster logical separation of functionality.

How it works...

It is considered best practice to place all logically related functions into a separate PHP file. Create a file called

`chap_03_developing_functions_library.php` and place these functions (described previously) inside:

- `someName()`
- `someOtherName()`
- `someInfinite()`
- `someDirScan()`
- `someTypeHint()`

This file is then included in the code that uses these functions.

```
include (__DIR__ . DIRECTORY_SEPARATOR .
'chap_03_developing_functions_library.php');
```

To call the `someName()` function, use the name and supply the parameter.

```
echo someName('TEST');    // returns "INIT and also
TEST"
```

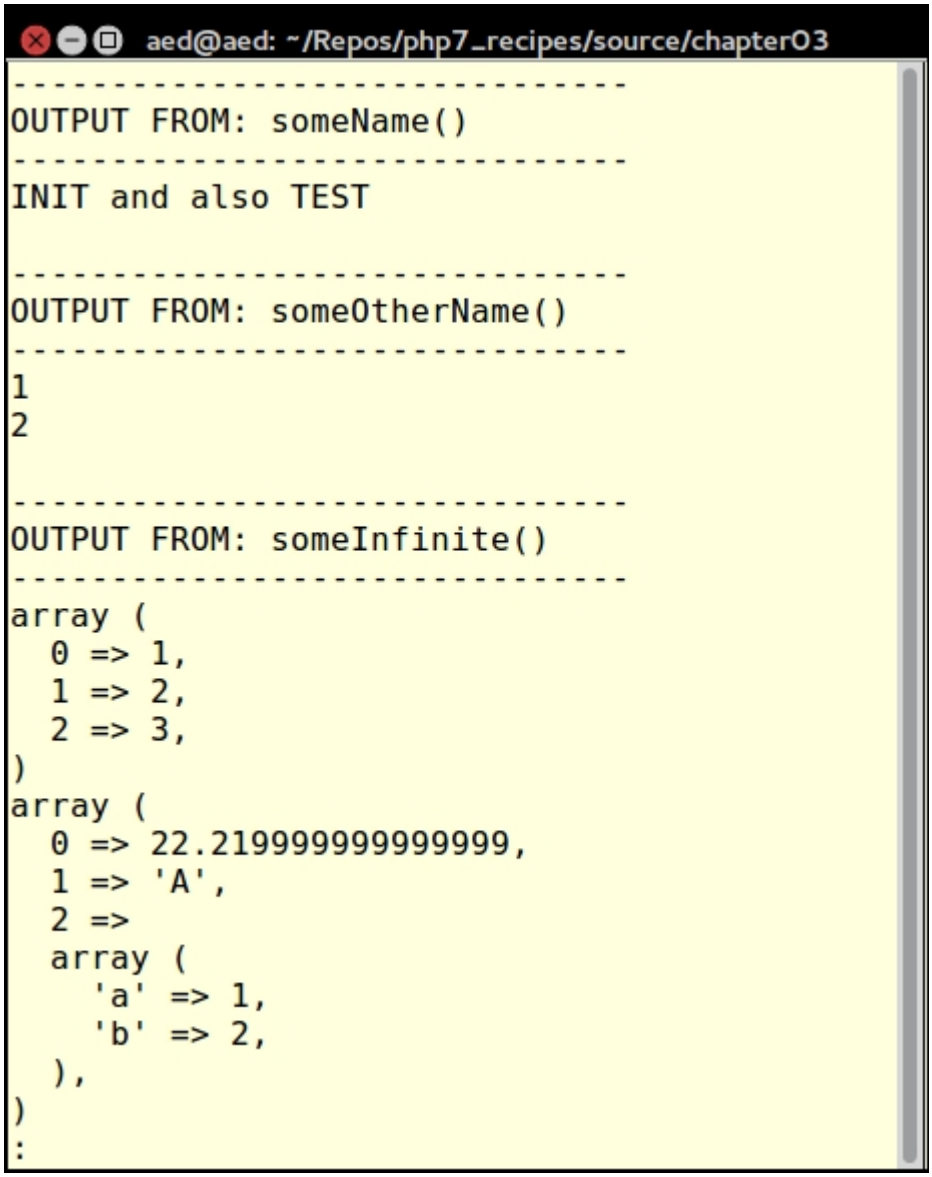
You can call the `someOtherName()` function using one or two parameters, as shown here:

```
echo someOtherName(1);     // returns  1
echo someOtherName(1, 1);    //  returns 2
```

The `someInfinite()` function accepts an infinite (or variable) number of parameters. Here are a couple of examples calling this function:

```
echo someInfinite(1, 2, 3);
echo PHP_EOL;
echo someInfinite(22.22, 'A', ['a' => 1, 'b' =>
2]);
```

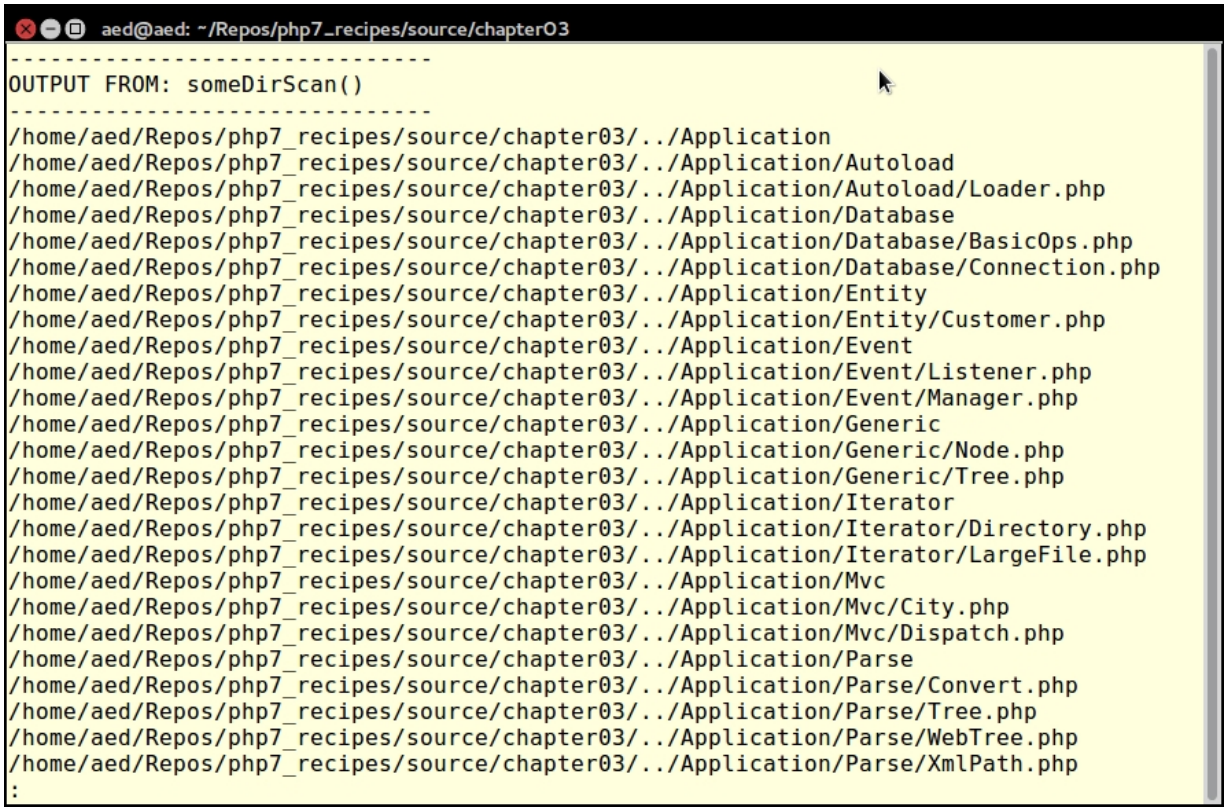
The output looks like this:



We can call `someInfinite()` as follows:

```
foreach (someDirScan(__DIR__ . DIRECTORY_SEPARATOR
. '..') as $item) {
    echo $item . PHP_EOL;
}
```

The output looks like this:


aed@aed: ~/Repos/php7_recipes/source/chapter03

OUTPUT FROM: someDirScan()

/home/aed/Repos/php7_recipes/source/chapter03/../Application
/home/aed/Repos/php7_recipes/source/chapter03/../Application/Autoload
/home/aed/Repos/php7_recipes/source/chapter03/../Application/Autoload/Loader.php
/home/aed/Repos/php7_recipes/source/chapter03/../Application/Database
/home/aed/Repos/php7_recipes/source/chapter03/../Application/Database/BasicOps.php
/home/aed/Repos/php7_recipes/source/chapter03/../Application/Database/Connection.php
/home/aed/Repos/php7_recipes/source/chapter03/../Application/Entity
/home/aed/Repos/php7_recipes/source/chapter03/../Application/Entity/Customer.php
/home/aed/Repos/php7_recipes/source/chapter03/../Application/Event
/home/aed/Repos/php7_recipes/source/chapter03/../Application/Event/Listener.php
/home/aed/Repos/php7_recipes/source/chapter03/../Application/Event/Manager.php
/home/aed/Repos/php7_recipes/source/chapter03/../Application/Generic
/home/aed/Repos/php7_recipes/source/chapter03/../Application/Generic/Node.php
/home/aed/Repos/php7_recipes/source/chapter03/../Application/Generic/Tree.php
/home/aed/Repos/php7_recipes/source/chapter03/../Application/Iterator
/home/aed/Repos/php7_recipes/source/chapter03/../Application/Iterator/Directory.php
/home/aed/Repos/php7_recipes/source/chapter03/../Application/Iterator/LargeFile.php
/home/aed/Repos/php7_recipes/source/chapter03/../Application/Mvc
/home/aed/Repos/php7_recipes/source/chapter03/../Application/Mvc/City.php
/home/aed/Repos/php7_recipes/source/chapter03/../Application/Mvc/Dispatch.php
/home/aed/Repos/php7_recipes/source/chapter03/../Application/Parse
/home/aed/Repos/php7_recipes/source/chapter03/../Application/Parse/Convert.php
/home/aed/Repos/php7_recipes/source/chapter03/../Application/Parse/Tree.php
/home/aed/Repos/php7_recipes/source/chapter03/../Application/Parse/WebTree.php
/home/aed/Repos/php7_recipes/source/chapter03/../Application/Parse/XmlPath.php
:

Hinting at data types

In many cases when developing functions, you might reuse the same library of functions in other projects. Also, if you work with a team, your code might be used by other developers. In order to control the use of your code, it might be appropriate to make use of a **type hint**. This involves specifying the data type your function expects for that particular parameter.

How to do it...

1. Parameters in functions can be prefixed by a type hint. The following type hints are available in both PHP 5 and PHP 7:

    - Array
    - Class
    - Callable

2. If a call to the function is made, and the wrong parameter type is passed, a `TypeError` is thrown. The following example requires an array, an instance of `DateTime`, and an anonymous function:

```
function someTypeHint(Array $a, DateTime $t,
Callable $c)
{
  $message = '';
  $message .= 'Array Count: ' . count($a) .
PHP_EOL;
  $message .= 'Date: ' . $t->format('Y-m-d') .
PHP_EOL;
  $message .= 'Callable Return: ' . $c() .
PHP_EOL;
  return $message;
}
```

> **Tip**

You don't have to provide a type hint for every single parameter. Use this technique only where supplying a different data type would have a negative effect on the processing of your function. As an example, if your function uses a `foreach()` loop, if you do not supply an array, or something which implements `Traversable`, an error will be generated.

3. In PHP 7, presuming the appropriate `declare()` directive is made, **scalar** (that is, integer, float, boolean, and string) type hints are allowed. Another function demonstrates how this is accomplished. At the top of the code library file which contains the function in which you wish to use scalar type hinting, add this `declare()` directive just after the opening PHP tag:

```
declare(strict_types=1);
```

4. Now you can define a function that includes scalar type hints:

```
function someScalarHint(bool $b, int $i, float
$f, string $s)
{
  return sprintf("\n%20s : %5s\n%20s :
%5d\n%20s " .
                 ": %5.2f\n%20s : %20s\n\n",
                 'Boolean', ($b ? 'TRUE' :
'FALSE'),
                 'Integer', $i,
                 'Float',   $f,
                 'String',  $s);
}
```

5. In PHP 7, assuming strict type hinting has been declared, boolean type hinting works a bit differently from the other three scalar types (that is, integer, float, and string). You can supply any scalar as an argument and no `TypeError` will be thrown! However, the incoming value will automatically be converted to the boolean data type once passed into the function. If you pass any data type other than scalar (that is, array or object) a `TypeError` will be thrown. Here is an example of a function that

defines a `boolean` data type. Note that the return value will be automatically converted to a `boolean`:

```
function someBoolHint(bool $b)
{
  return $b;
}
```

How it works...

First of all, you can place the three functions, `someTypeHint()`, `someScalarHint()`, and `someBoolHint()`, into a separate file to be included. For this example, we will name the file `chap_03_developing_functions_type_hints_library.php`. Don't forget to add `declare(strict_types=1)` at the top!

In our calling code, you would then include the file:

```
include (__DIR__ . DIRECTORY_SEPARATOR .
'chap_03_developing_functions_type_hints_library.p
hp');
```

To test `someTypeHint()`, call the function twice, once with the correct data types, and the second time with incorrect types. This will throw a `TypeError`, however, so you will need to wrap the function calls in a `try { ... } catch () { ...}` block:

```
try {
    $callable = function () { return 'Callback
Return'; };
    echo someTypeHint([1,2,3], new DateTime(),
$callable);
    echo someTypeHint('A', 'B', 'C');
} catch (TypeError $e) {
    echo $e->getMessage();
    echo PHP_EOL;
}
```

As you can see from the output shown at the end of this sub-section, when passing the correct data types there is no problem. When passing the incorrect types, a `TypeError` is thrown.

Note

In PHP 7, certain errors have been converted into an `Error` class, which is processed in a somewhat similar manner to an `Exception`. This means you can catch an `Error`. `TypeError` is a specific descendant of `Error` that is thrown when incorrect data types are passed to functions.

All PHP 7 `Error` classes implement the `Throwable` interface, as does the `Exception` class. If you are not sure if you need to catch an `Error` or an `Exception`, you can add a block which catches `Throwable`.

Next you can test `someScalarHint()`, calling it twice with correct and incorrect values, wrapping the calls in a `try { ... } catch () { ...}` block:

```
try {
    echo someScalarHint(TRUE, 11, 22.22, 'This is
a string');
    echo someScalarHint('A', 'B', 'C', 'D');
} catch (TypeError $e) {
    echo $e->getMessage();
}
```

As expected, the first call to the function works, and the second throws a `TypeError`.

When type hinting for boolean values, any scalar value passed will *not* cause a `TypeError` to be thrown! Instead, the value will be interpreted into its boolean equivalent. If you subsequently return this value, the data type will be changed to boolean.

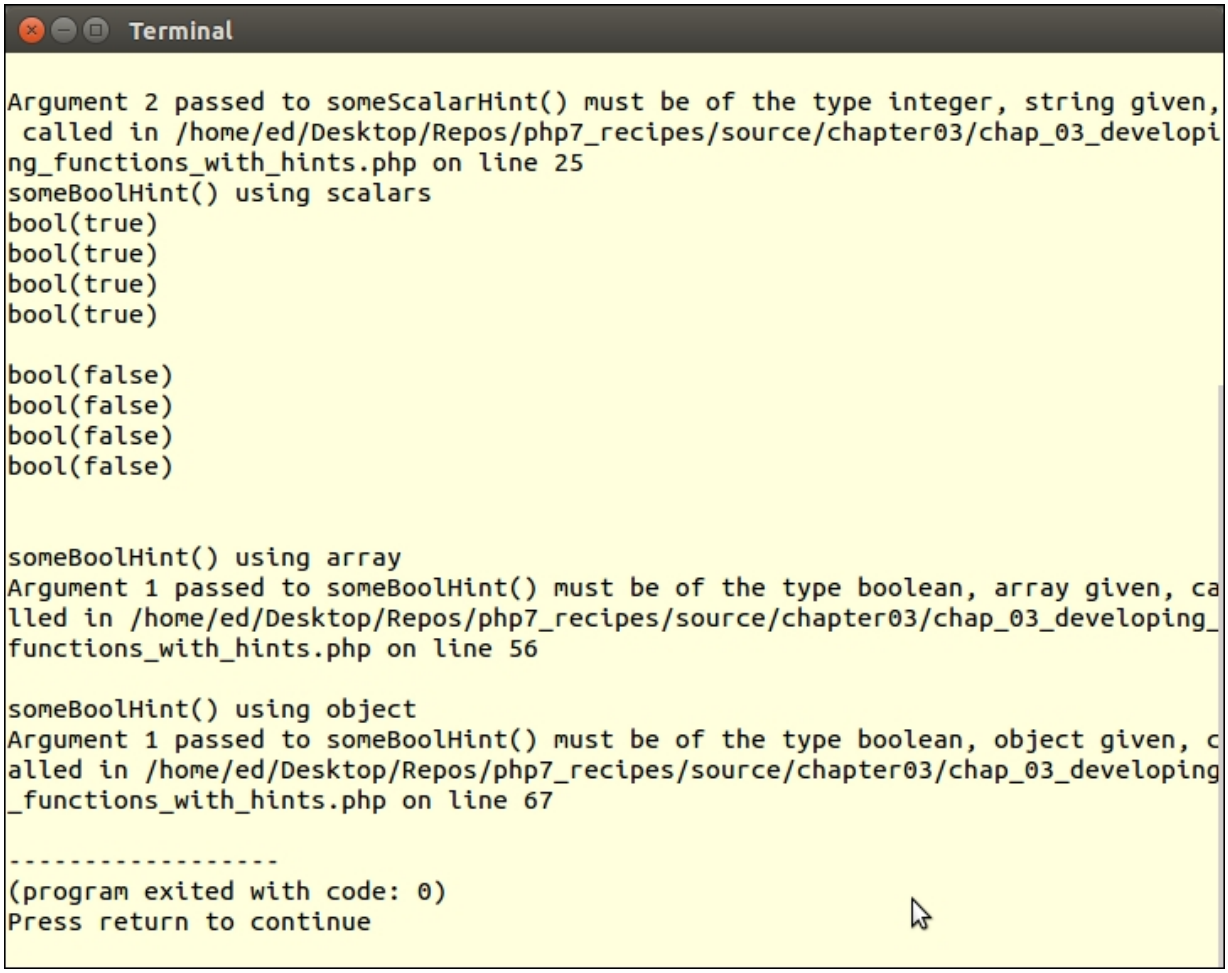
To test this, call the `someBoolHint()` function defined previously, and pass any scalar value in as an argument. The `var_dump()` method reveals that the data type is always boolean:

```
try {
    // positive results
    $b = someBooleanHint(TRUE);
    $i = someBooleanHint(11);
    $f = someBooleanHint(22.22);
    $s = someBooleanHint('X');
    var_dump($b, $i, $f, $s);
    // negative results
    $b = someBooleanHint(FALSE);
    $i = someBooleanHint(0);
    $f = someBooleanHint(0.0);
    $s = someBooleanHint('');
    var_dump($b, $i, $f, $s);
} catch (TypeError $e) {
    echo $e->getMessage();
}
```

If you now try the same function call, but pass in a non-scalar data type, a `TypeError` is thrown:

```
try {
    $a = someBoolHint([1,2,3]);
    var_dump($a);
} catch (TypeError $e) {
    echo $e->getMessage();
}
try {
    $o = someBoolHint(new stdClass());
    var_dump($o);
} catch (TypeError $e) {
    echo $e->getMessage();
}
```

Here is the overall output:



See also

PHP 7.1 introduced a new type hint `iterable` which allows arrays, `Iterators` or `Generators` as arguments. See this for more information:

- https://wiki.php.net/rfc/iterable

For a background discussion on the rationale behind the implementation of scalar type hinting, have a look at this article:

- https://wiki.php.net/rfc/scalar_type_hints_v5

Using return value data typing

PHP 7 allows you to specify a data type for the return value of a function. Unlike scalar type hinting, however, you don't need to add any special declarations.

How to do it...

1. This example shows you how to assign a data type to a function return value. To assign a return data type, first define the function as you would normally. After the closing parenthesis, add a space, followed by the data type and a colon:

```
function returnsString(DateTime $date,
$format) : string
{
  return $date->format($format);
}
```

> **Note**
>
> PHP 7.1 introduced a variation on return data typing called **nullable types**. All you need to do is to change `string` to `?string`. This allows the function to return either `string` or `NULL`.

2. Anything returned by the function, regardless of its data type inside the function, will be converted to the declared data type as a return value. Notice, in this example, the values of `$a`, `$b`, and `$c` are added together to produce a single sum, which is returned. Normally you would expect the return value to be a numeric data type. In this case, however, the return data type is declared as `string`, which overrides PHP's type-juggling process:

```
function convertsToString($a, $b, $c) : string

  return $a + $b + $c;
}
```

3. You can also assign classes as a return data type. In this example, we assign a return type of `DateTime`, part of the PHP `DateTime` extension:

```
function makesDateTime($year, $month, $day) :
DateTime
{
  $date = new DateTime();
  $date->setDate($year, $month, $day);
  return $date;
}
```

> ## Note
>
> The `makesDateTime()` function would be a potential candidate for scalar type hinting. If `$year`, `$month`, or `$day` are not integers, a `Warning` is generated when `setDate()` is called. If you use scalar type hinting, and the wrong data types are passed, a `TypeError` is thrown. Although it really doesn't matter whether a warning is generated or a `TypeError` is thrown, at least the `TypeError` will cause the errant developer who is misusing your code to sit up and take notice!

4. If a function has a return data type, and you return the wrong data type in your function code, a `TypeError` will be thrown at runtime. This function assigns a return type of `DateTime`, but returns a string instead. A `TypeError` will be thrown, but not until runtime, when the PHP engine detects the discrepancy:

```
function wrongDateTime($year, $month, $day) :
DateTime
{
  return date($year . '-' . $month . '-' .
```

```
$day);
}
```

> **Note**
>
> If the return data type class is not one of the built-in PHP classes (that is, a class that is part of the SPL), you will need to make sure the class has been auto-loaded, or included.

How it works...

First, place the functions mentioned previously into a library file called `chap_03_developing_functions_return_types_library.php`. This file needs to be included in the `chap_03_developing_functions_return_types.php` script that calls these functions:

```
include (__DIR__ .
'/chap_03_developing_functions_return_types_librar
y.php');
```

Now you can call `returnsString()`, supplying a `DateTime` instance and a format string:

```
$date   = new DateTime();
$format = 'l, d M Y';
$now    = returnsString($date, $format);
echo $now . PHP_EOL;
var_dump($now);
```

As expected, the output is a string:

```
aed@aed: ~/Repos/php7_recipes/source/chapter03
aed@aed:~/Repos/php7_recipes/source/chapter03$ php chap_03_developing_functions_with_hints.php

returnsString()
Sunday, 31 Jan 2016
string(19) "Sunday, 31 Jan 2016"
aed@aed:~/Repos/php7_recipes/source/chapter03$
```

Now you can call `convertsToString()` and supply three integers as arguments. Notice that the return type is string:

```
echo "\nconvertsToString()\n";
var_dump(convertsToString(2, 3, 4));
```

```
aed@aed: ~/Repos/php7_recipes/source/chapter03
aed@aed:~/Repos/php7_recipes/source/chapter03$ php chap_03_developing_functions_with_hints.php

convertsToString()
string(1) "9"
aed@aed:~/Repos/php7_recipes/source/chapter03$
```

To demonstrate that, you can assign a class as a return value, call `makesDateTime()` with three integer parameters:

```
echo "\nmakesDateTime()\n";
$d = makesDateTime(2015, 11, 21);
var_dump($d);
```

```
aed@aed: ~/Repos/php7_recipes/source/chapter03
aed@aed:~/Repos/php7_recipes/source/chapter03$ php chap_03_developing_functions_with_hints.php

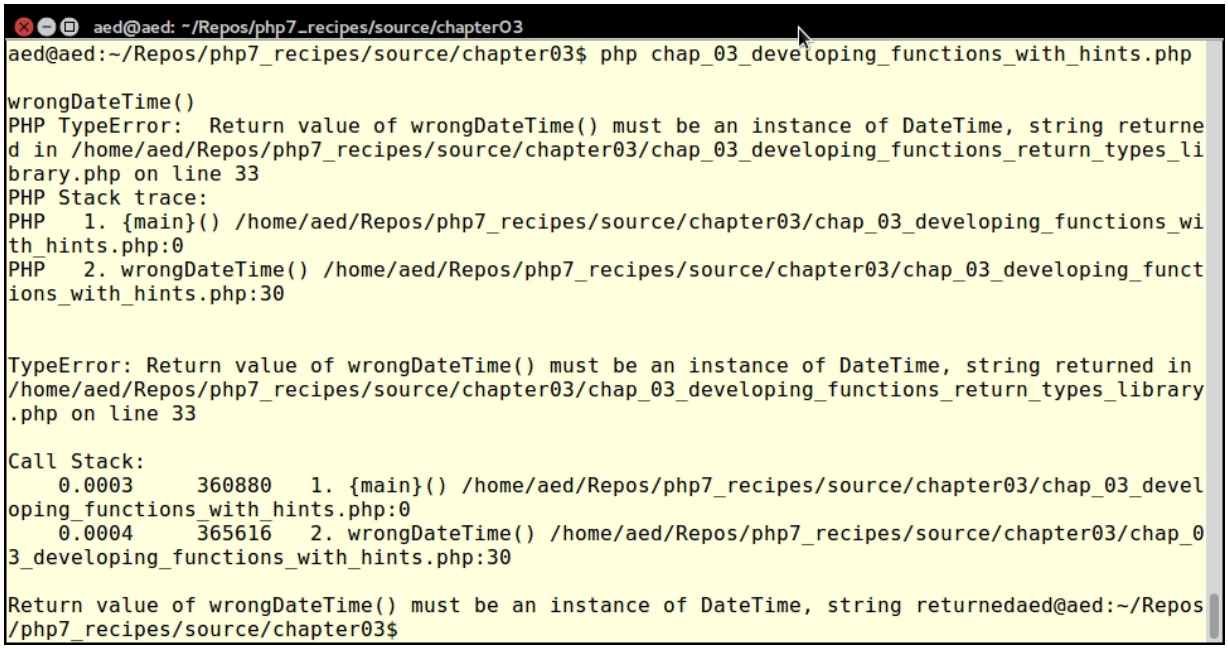
makesDateTime()
class DateTime#1 (3) {
  public $date =>
  string(26) "2015-11-21 18:32:25.000000"
  public $timezone_type =>
  int(3)
  public $timezone =>
  string(13) "Europe/London"
}
aed@aed:~/Repos/php7_recipes/source/chapter03$
```

Finally, call `wrongDateTime()` with three integer parameters:

```
try {
    $e = wrongDateTime(2015, 11, 21);
    var_dump($e);
```

```
} catch (TypeError $e) {
    echo $e->getMessage();
}
```

Notice that a `TypeError` is thrown at runtime:



There's more...

PHP 7.1 adds a new return value type, `void`. This is used when you do not wish to return any value from the function. For more information, please refer to https://wiki.php.net/rfc/void_return_type.

See also

For more information on return type declarations, see the following articles:

- http://php.net/manual/en/functions.arguments.php#functions.arguments.type-declaration.strict
- https://wiki.php.net/rfc/return_types

For information on nullable types, please refer to this article:

- https://wiki.php.net/rfc/nullable_types

Using iterators

An **iterator** is a special type of class that allows you to **traverse** a *container* or list. The keyword here is *traverse*. What this means is that the iterator provides the means to go through a list, but it does not perform the traversal itself.

The SPL provides a rich assortment of generic and specialized iterators designed for different contexts. The `ArrayIterator`, for example, is designed to allow object-oriented traversal of arrays. The `DirectoryIterator` is designed for filesystem scanning.

Certain SPL iterators are designed to work with others, and add value. Examples include `FilterIterator` and `LimitIterator`. The former gives you the ability to remove unwanted values from the parent iterator. The latter provides a pagination capability whereby you can designate how many items to traverse along with an offset that determines where to start.

Finally, there are a series of *recursive* iterators, which allow you to repeatedly call the parent iterator. An example would be `RecursiveDirectoryIterator` which scans a directory tree all the way from a starting point to the last possible subdirectory.

How to do it...

1. We first examine the `ArrayIterator` class. It's extremely easy to use. All you need to do is to supply an array as an argument to the constructor. After that you can use any of the methods that are standard to all SPL-based iterators, such as `current()`, `next()`, and so on.

```
$iterator = new ArrayIterator($array);
```

> **Note**
>
> Using `ArrayIterator` converts a standard PHP array into an iterator. In a certain sense, this provides a bridge between procedural programming and OOP.

2. As an example of a practical use for the iterator, have a look at this example. It takes an iterator and produces a series of HTML `<ul>` and `<li>` tags:

```
function htmlList($iterator)
{
  $output = '<ul>';
  while ($value = $iterator->current()) {
    $output .= '<li>' . $value . '</li>';
    $iterator->next();
  }
  $output .= '</ul>';
  return $output;
}
```

3. Alternatively, you can simply wrap the `ArrayIterator` instance into a simple `foreach()` loop:

```
function htmlList($iterator)
{
  $output = '<ul>';
  foreach($iterator as $value) {
    $output .= '<li>' . $value . '</li>';
  }
  $output .= '</ul>';
  return $output;
}
```

4. `CallbackFilterIterator` is a great way to add value to any existing iterator you might be using. It allows you to wrap any existing iterator and screen the output. In this example we'll define `fetchCountryName()`, which iterates through a database query which produces a list of country names. First, we define an `ArrayIterator` instance from a query that uses the

`Application\Database\Connection` class defined in Chapter 1, *Building a Foundation*:

```
function fetchCountryName($sql, $connection)
{
  $iterator = new ArrayIterator();
  $stmt = $connection->pdo->query($sql);
  while($row = $stmt->fetch(PDO::FETCH_ASSOC))
{
    $iterator->append($row['name']);
  }
  return $iterator;
}
```

5. Next, we define a filter method, `nameFilterIterator()`, which accepts a partial country name as an argument along with the `ArrayIterator` instance:

```
function nameFilterIterator($innerIterator,
$name)
{
  if (!$name) return $innerIterator;
  $name = trim($name);
  $iterator = new
CallbackFilterIterator($innerIterator,
    function($current, $key, $iterator) use
($name) {
      $pattern = '/' . $name . '/i';
      return (bool) preg_match($pattern,
$current);
    }
  );
  return $iterator;
}
```

6. `LimitIterator` adds a basic pagination aspect to your applications. To use this iterator, you only need to supply the parent iterator, an offset, and a limit. `LimitIterator` will then only produce a subset of the entire data set starting at the offset. Taking the same example mentioned in step 2, we'll paginate the results coming from our database query. We can do this quite simply by wrapping the iterator produced by the `fetchCountryName()` method inside a `LimitIterator` instance:

```
$pagination = new
LimitIterator(fetchCountryName(
$sql, $connection), $offset, $limit);
```

> **Note**
>
> Be careful when using `LimitIterator`. It needs to have the *entire* data set in memory in order to effect a limit. Accordingly, this would not be a good tool to use when iterating through large data sets.

7. Iterators can be *stacked*. In this simple example, an `ArrayIterator` is processed by a `FilterIterator`, which in turn is limited by a `LimitIterator`. First we set up an instance of `ArrayIterator`:

```
$i = new ArrayIterator($a);
```

8. Next, we plug the `ArrayIterator` into a `FilterIterator` instance. Note that we are using the new PHP 7 anonymous class feature. In this case the anonymous class extends `FilterIterator` and overrides the `accept()` method, allowing only letters with even-numbered ASCII codes:

```
$f = new class ($i) extends FilterIterator {
  public function accept()
  {
    $current = $this->current();
    return !(ord($current) & 1);
  }
};
```

9. Finally, we supply the `FilterIterator` instance as an argument to `LimitIterator`, and provide an offset (`2` in this example) and a limit (`6` in this example):

```
$l = new LimitIterator($f, 2, 6);
```

10. We could then define a simple function to display output, and call each iterator in turn to see the results on a simple array produced by `range('A', 'Z')`:

```
function showElements($iterator)
{
  foreach($iterator as $item)  echo $item . '
';
  echo PHP_EOL;
}

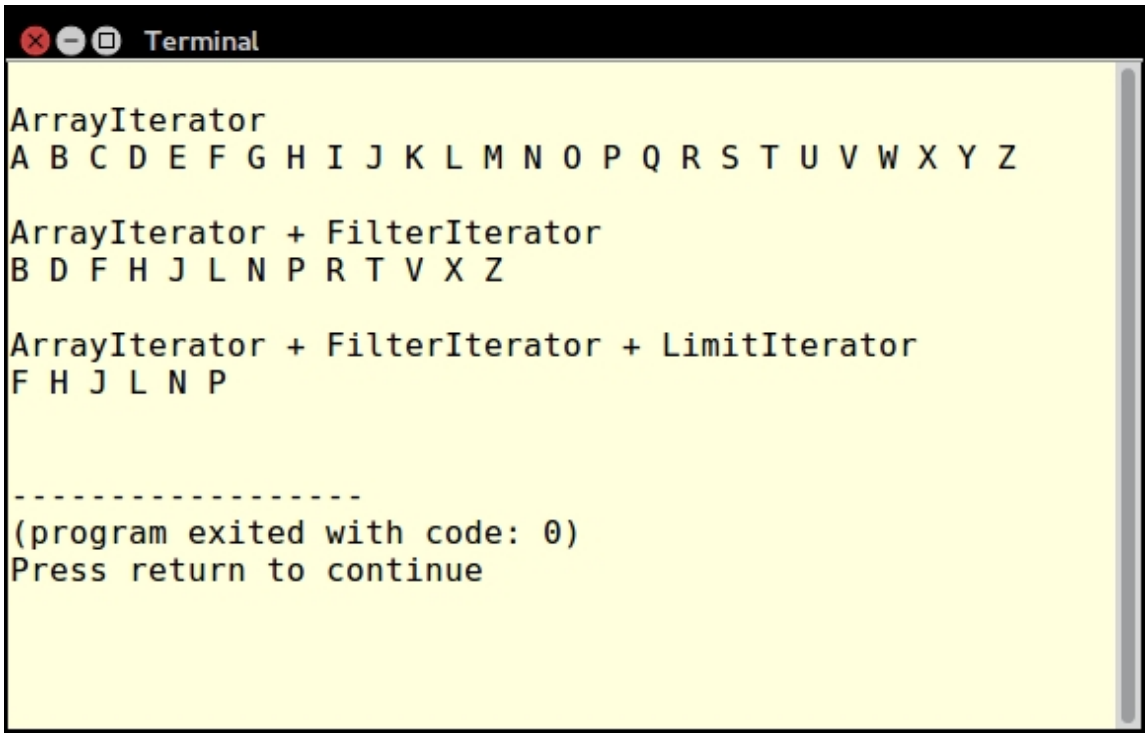
$a = range('A', 'Z');
$i = new ArrayIterator($a);
showElements($i);
```

11. Here is a variation that produces every other letter by stacking a `FilterIterator` on top of an `ArrayIterator`:

```
$f = new class ($i) extends FilterIterator {
public function accept()
  {
    $current = $this->current();
    return !(ord($current) & 1);
  }
};
showElements($f);
```

12. And here's yet another variation that only produces `F H J L N P`, which demonstrates a `LimitIterator` that consumes a `FilterIterator`, which in turn consumes an `ArrayIterator`. The output of these three examples is as follows:

```
$l = new LimitIterator($f, 2, 6);
showElements($l);
```



13. Returning to our example that produces a list of country names, suppose, instead of only the country name, we wished to iterate through a multi-dimensional array consisting of country names and ISO codes. The simple iterators mentioned so far would not be sufficient. Instead, we will use what are known as **recursive** iterators.
14. First of all, we need to define a method that uses the database connection class mentioned previously to pull all columns from the database. As before, we return an `ArrayIterator` instance populated with data from the query:

```
function fetchAllAssoc($sql, $connection)
{
  $iterator = new ArrayIterator();
  $stmt = $connection->pdo->query($sql);
  while($row = $stmt->fetch(PDO::FETCH_ASSOC))
{
    $iterator->append($row);
  }
  return $iterator;
}
```

15. At first glance one would be tempted to simply wrap a standard `ArrayIterator` instance inside `RecursiveArrayIterator`. Unfortunately, this approach only performs a **shallow** iteration, and doesn't give us what we want: an iteration through all elements of the multi-dimensional array that is returned from a database query:

```
$iterator = fetchAllAssoc($sql, $connection);
$shallow  = new
RecursiveArrayIterator($iterator);
```

16. Although this returns an iteration where each item represents a row from the database query, in this case we wish to provide an iteration that will iterate through all columns of all rows returned by the query. In order to accomplish this, we'll need to roll out the big brass by way of a `RecursiveIteratorIterator`.
17. Monty Python fans will revel in the rich irony of this class name as it brings back fond memories of the *The Department of Redundancy Department*. Fittingly, this class causes our old friend the `RecursiveArrayIterator` class to work overtime and perform a **deep** iteration through all levels of the array:

```
$deep     = new
RecursiveIteratorIterator($shallow);
```

How it works...

As a practical example, you can develop a test script which implements filtering and pagination using iterators. For this illustration, you could call the `chap_03_developing_functions_filtered_and_paginated.php` test code file.

First of all, following best practices, place the functions described above into an include file called `chap_03_developing_functions_iterators_library.php`. In the test script, be sure to include this file.

The data source is a table called `iso_country_codes`, which contains ISO2, ISO3, and country names. The database connection could be in a `config/db.config.php` file. You could also include the `Application\Database\Connection` class discussed in the previous chapter:

```
define('DB_CONFIG_FILE',
'/../config/db.config.php');
define('ITEMS_PER_PAGE', [5, 10, 15, 20]);
include (__DIR__ .
'/chap_03_developing_functions_iterators_library.p
hp');
include (__DIR__ .
'/../Application/Database/Connection.php');
```

Note

In PHP 7 you can define constants as arrays. In this example, `ITEMS_PER_PAGE` was defined as an array, and used to generate an HTML `SELECT` element.

Next, you can process input parameters for the country name and the number of items per page. The current page number will start at `0` and can be incremented (next page) or decremented (previous page):

```
$name = strip_tags($_GET['name'] ?? '');
$limit  = (int) ($_GET['limit'] ?? 10);
$page   = (int) ($_GET['page']  ?? 0);
$offset = $page * $limit;
$prev   = ($page > 0) ? $page - 1 : 0;
$next   = $page + 1;
```

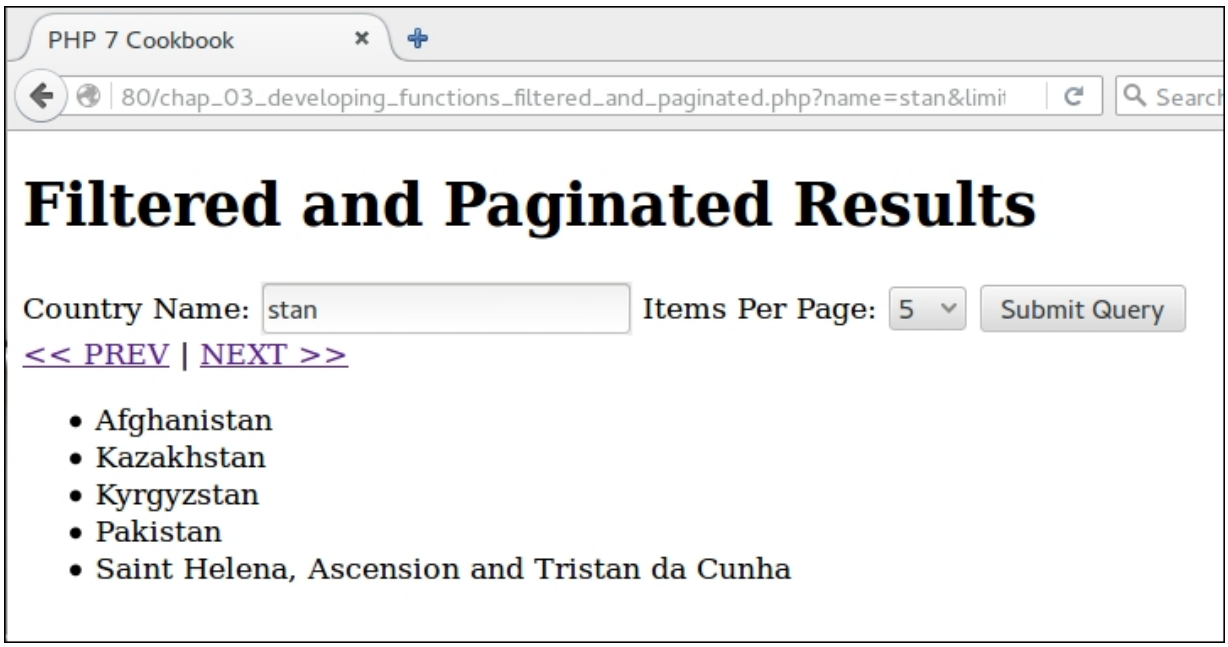
Now you're ready to fire up the database connection and run a simple `SELECT` query. This should be placed in a `try {} catch {}` block. You can then place the iterators to be stacked inside the `try {}` block:

```
try {
    $connection = new
Application\Database\Connection(
      include __DIR__ . DB_CONFIG_FILE);
    $sql    = 'SELECT * FROM iso_country_codes';
    $arrayIterator    = fetchCountryName($sql,
$connection);
    $filteredIterator =
nameFilterIterator($arrayIterator, $name);
    $limitIterator    = pagination(
    $filteredIterator, $offset, $limit);
} catch (Throwable $e) {
    echo $e->getMessage();
}
```

Now we're ready for the HTML. In this simple example we present a form that lets the user select the number of items per page and the country name:

```
<form>
  Country Name:
  <input type="text" name="name"
         value="<?= htmlspecialchars($name) ?>">
  Items Per Page:
  <select name="limit">
    <?php foreach (ITEMS_PER_PAGE as $item) : ?>
      <option<?= ($item == $limit) ? ' selected' :
'' ?>>
      <?= $item ?></option>
    <?php endforeach; ?>
  </select>
  <input type="submit" />
</form>
  <a href="?name=<?= $name ?>&limit=<?= $limit ?>
    &page=<?= $prev ?>">
  << PREV</a>
  <a href="?name=<?= $name ?>&limit=<?= $limit ?>
    &page=<?= $next ?>">
  NEXT >></a>
<?= htmlList($limitIterator); ?>
```

The output will look something like this:



Finally, in order to test the recursive iteration of the country database lookup, you will need to include the iterator's library file, as well as the `Application\Database\Connection` class:

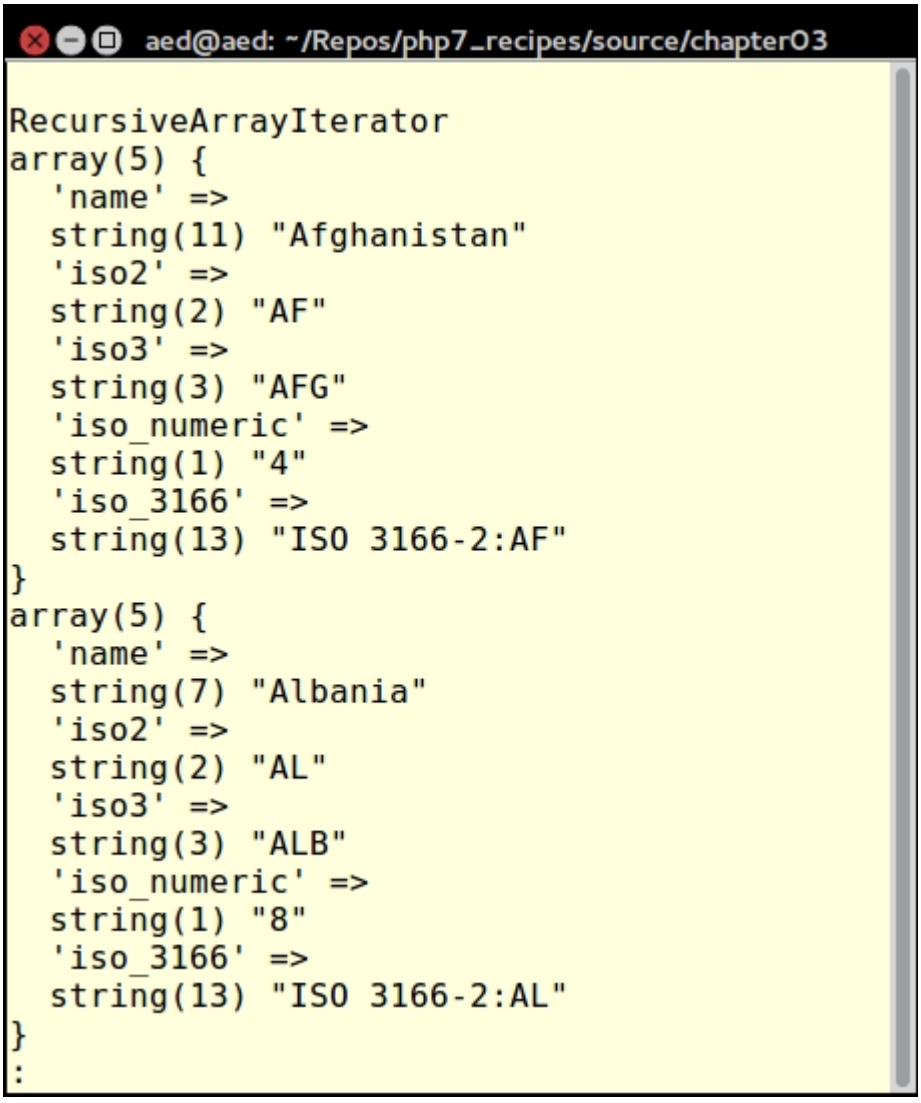
```
define('DB_CONFIG_FILE',
'/../config/db.config.php');
include (__DIR__ .
'/chap_03_developing_functions_iterators_library.p
hp');
include (__DIR__ .
'/../Application/Database/Connection.php');
```

As before, you should wrap your database query in a `try {} catch {}` block. You can then place the code to test the recursive iteration inside the `try {}` block:

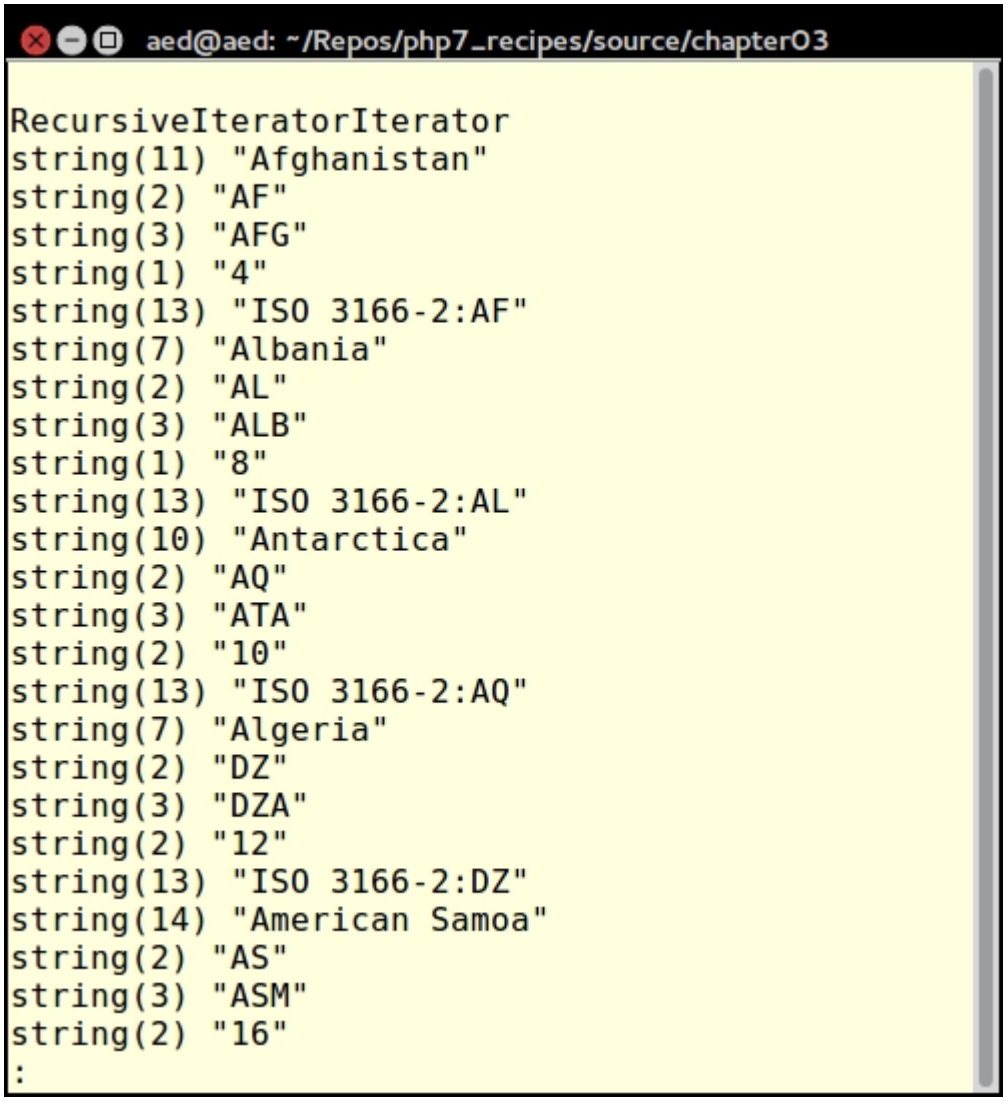
```
try {
    $connection = new
Application\Database\Connection(
    include __DIR__ . DB_CONFIG_FILE);
    $sql    = 'SELECT * FROM iso_country_codes';
    $iterator = fetchAllAssoc($sql, $connection);
    $shallow  = new
RecursiveArrayIterator($iterator);
    foreach ($shallow as $item) var_dump($item);
```

```
    $deep     = new
RecursiveIteratorIterator($shallow);
    foreach ($deep as $item) var_dump($item);
} catch (Throwable $e) {
    echo $e->getMessage();
}
```

Here is what you can expect to see in terms of output from `RecursiveArrayIterator`:



Here is the output after using `RecursiveIteratorIterator`:


aed@aed: ~/Repos/php7_recipes/source/chapter03
RecursiveIteratorIterator
string(11) "Afghanistan"
string(2) "AF"
string(3) "AFG"
string(1) "4"
string(13) "ISO 3166-2:AF"
string(7) "Albania"
string(2) "AL"
string(3) "ALB"
string(1) "8"
string(13) "ISO 3166-2:AL"
string(10) "Antarctica"
string(2) "AQ"
string(3) "ATA"
string(2) "10"
string(13) "ISO 3166-2:AQ"
string(7) "Algeria"
string(2) "DZ"
string(3) "DZA"
string(2) "12"
string(13) "ISO 3166-2:DZ"
string(14) "American Samoa"
string(2) "AS"
string(3) "ASM"
string(2) "16"
:

Writing your own iterator using generators

In the preceding set of recipes we demonstrated the use of iterators provided in the PHP 7 SPL. But what if this set doesn't provide you with what is needed for a given project? One solution would be to develop a function that, instead of building an array that is then returned, uses the `yield` keyword to return values progressively by way of iteration. Such a function is referred to as a **generator**. In fact, in the background, the PHP engine will automatically convert your function into a special built-in class called `Generator`.

There are several advantages to this approach. The main benefit is seen when you have a large container to traverse (that is, parsing a massive file). The traditional approach has been to build up an array, and then return that array. The problem with this is that you are effectively doubling the amount of memory required! Also, performance is affected in that results are only achieved once the final array has been returned.

How to do it...

1. In this example we build on the library of iterator-based functions, adding a generator of our own design. In this case we will duplicate the functionality described in the section above on iterators where we stacked an `ArrayIterator`, `FilterIterator`, and `LimitIterator`.
2. Because we need access to the source array, the desired filter, page number, and number of items per page, we include the appropriate parameters into a single `filteredResultsGenerator()` function. We then calculate the offset based on the page number and limit (that is, number of items per page). Next, we loop through the array, apply the filter,

and continue the loop if the offset has not yet been reached, or break if the limit has been reached:

```
function filteredResultsGenerator(array
$array, $filter, $limit = 10, $page = 0)
  {
    $max    = count($array);
    $offset = $page * $limit;
    foreach ($array as $key => $value) {
      if (!stripos($value, $filter) !== FALSE)
continue;
      if (--$offset >= 0) continue;
      if (--$limit <= 0) break;
      yield $value;
    }
  }
```

3. You'll notice the primary difference between this function and others is the `yield` keyword. The effect of this keyword is to signal the PHP engine to produce a `Generator` instance and encapsulate the code.

How it works...

To demonstrate the use of the `filteredResultsGenerator()` function we'll have you implement a web application that scans a web page and produces a filtered and paginated list of URLs hoovered from `HREF` attributes.

First you need to add the code for the `filteredResultsGenerator()` function to the library file used in the previous recipe, then place the functions described previously into an include file, `chap_03_developing_functions_iterators_library.php`.

Next, define a test script, `chap_03_developing_functions_using_generator.php`, that includes both the function library as well as the file that defines `Application\Web\Hoover`, described in Chapter 1, *Building a Foundation*:

```
include (__DIR__ . DIRECTORY_SEPARATOR .
'chap_03_developing_functions_iterators_library.ph
p');
include (__DIR__ .
'/../Application/Web/Hoover.php');
```

You will then need to gather input from the user regarding which URL to scan, what string to use as a filter, how many items per page, and the current page number.

Note

The **null coalesce** operator (`??`) is ideal for getting input from the Web. It does not generate any notices if not defined. If the parameter is not received from user input, you can supply a default.

```
$url    = trim(strip_tags($_GET['url'] ?? ''));
$filter = trim(strip_tags($_GET['filter'] ?? ''));
$limit  = (int) ($_GET['limit'] ?? 10);
$page   = (int) ($_GET['page']  ?? 0);
```

Tip

Best practice

Web security should always be a priority consideration. In this example you can use `strip_tags()` and also force the data type to integer `(int)` as measures to sanitize user input.

You are then in a position to define variables used in links for previous and next pages in the paginated list. Note that you could also apply a *sanity check* to make sure the next page doesn't go off the end of the result set. For the sake of brevity, such a sanity check was not applied in this example:

```
$next   = $page + 1;
$prev   = $page - 1;
$base   = '?url=' . htmlspecialchars($url)
        . '&filter=' . htmlspecialchars($filter)
        . '&limit=' . $limit
        . '&page=';
```

We then need to create an `Application\Web\Hoover` instance and grab `HREF` attributes from the target URL:

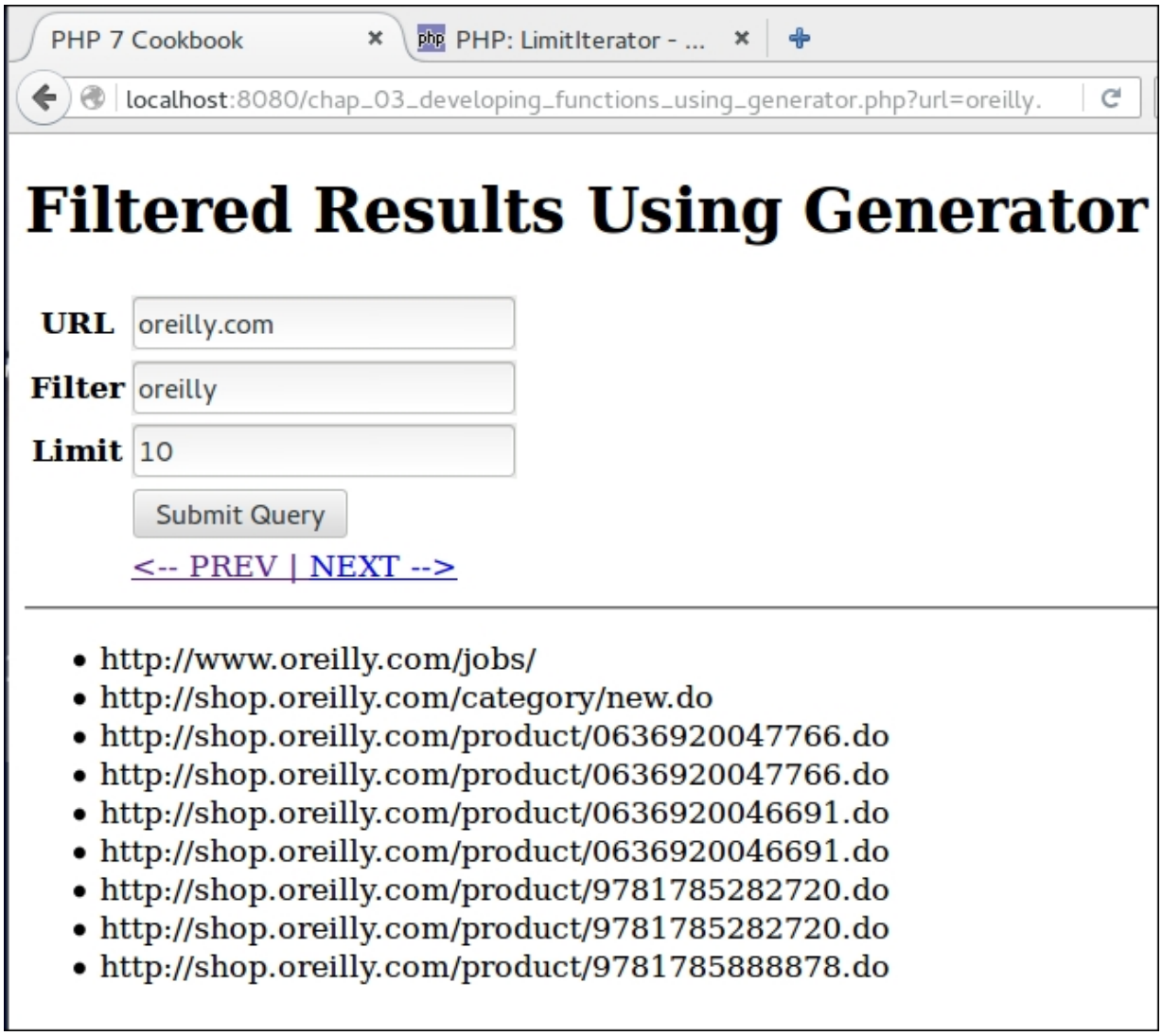
```
$vac    = new Application\Web\Hoover();
$list   = $vac->getAttribute($url, 'href');
```

Finally, we define HTML output that renders an input form and runs our generator through the `htmlList()` function described previously:

```
<form>
<table>
<tr>
<th>URL</th>
<td>
<input type="text" name="url"
  value="<?= htmlspecialchars($url) ?>"/>
</td>
</tr>
<tr>
<th>Filter</th>
<td>
<input type="text" name="filter"
  value="<?= htmlspecialchars($filter) ?>"/></td>
</tr>
<tr>
<th>Limit</th>
<td><input type="text" name="limit" value="<?=
$limit ?>"/></td>
</tr>
<tr>
<th> </th><td><input type="submit" /></td>
</tr>
<tr>
<td> </td>
<td>
<a href="<?= $base . $prev ?>"><-- PREV |
<a href="<?= $base . $next ?>">NEXT --></td>
```

```
</tr>
</table>
</form>
<hr>
<?= htmlList(filteredResultsGenerator(
$list, $filter, $limit, $page)); ?>
```

Here is an example of the output:



Chapter 4. Working with PHP Object-Oriented Programming

In this chapter we will cover:

- Developing classes
- Extending classes
- Using static properties and methods
- Using namespaces
- Defining visibility
- Using interfaces
- Using traits
- Implementing anonymous classes

Introduction

In this chapter, we will consider recipes that take advantage of the **object-oriented programming** (**OOP**) capabilities available in PHP 7.0, 7.1, and above. Most of the OOP functionality available in PHP 7.x is also available in PHP 5.6. A new feature introduced in PHP 7 is support for **anonymous classes**. In PHP 7.1, you can modify the visibility of class constants.

> ## Note
>
> Another radically new feature is the ability to **catch** certain types of error. This is discussed in greater detail in Chapter 13, *Best Practices, Testing, and Debugging*.

Developing classes

The traditional development approach is to place the class into its own file. Typically, classes contain logic that implements a single purpose. Classes are further broken down into self-contained functions which are referred to as **methods**. Variables defined inside classes are referred to as **properties**. It is recommended to develop a test class at the same time, a topic discussed in more detail in Chapter 13, *Best Practices, Testing, and Debugging*.

How to do it...

1. Create a file to contain the class definition. For the purposes of autoloading it is recommended that the filename match the classname. At the top of the file, before the keyword `class`, add a **DocBlock**. You can then define properties and methods. In this example, we define a class `Test`. It has a property `$test`, and a method `getTest()`:

```
<?php
declare(strict_types=1);
/**
 * This is a demonstration class.
 *
 * The purpose of this class is to get and set
 * a protected property $test
 *
 */
class Test
{

  protected $test = 'TEST';

  /**
   * This method returns the current value of
$test
   *
   * @return string $test
   */
  public function getTest() : string
```

```
    {
      return $this->test;
    }

    /**
     * This method sets the value of $test
     *
     * @param string $test
     * @return Test $this
     */
    public function setTest(string $test)
    {
      $this->test = $test;
      return $this;
    }
}
```

Tip

Best practice

It is considered best practice to name the file after the class. Although class names in PHP are not case sensitive, it is further considered best practice to use an uppercase letter for the first name of a class. You should not put executable code in a class definition file.

Each class should contain a **DocBlock** before the keyword `class`. In the DocBlock you should include a short description of the purpose of the class. Skip a line, and then include a more detailed description. You can also include `@` tags such as `@author`, `@license` and so on. Each method should likewise be preceded by a DocBlock that identifies the purpose of the method, as well as its incoming parameters and return value.

2. It's possible to define more than one class per file, but is not considered best practice. In this example we create a file, `NameAddress.php`, which defines two classes, `Name` and `Address`:

```
<?php
declare(strict_types=1);
class Name
{

  protected $name = '';

  public function getName() : string
  {
    return $this->name;
  }

  public function setName(string $name)
  {
    $this->name = $name;

    return $this;
  }
}

class Address
{

  protected $address = '';

  public function getAddress() : string
  {
    return $this->address;
  }

  public function setAddress(string $address)
  {
    $this->address = $address;
    return $this;
  }
}
```

Tip

> Although you can define more than one class in a single file, as shown in the preceding code snippet, it is not considered best practice. Not only does this negate the logical purity of the file, but it makes autoloading more difficult.

3. Class names are case-insensitive. Duplications will be flagged as errors. In this example, in a file `TwoClass.php`, we define two classes, `TwoClass` and `twoclass`:

```
<?php
class TwoClass
{
  public function showOne()
  {
    return 'ONE';
  }
}

// a fatal error will occur when the second
class definition is parsed
class twoclass
{
  public function showTwo()
  {
    return 'TWO';
  }
}
```

4. PHP 7.1 has addressed inconsistent behavior in the use of the keyword `$this`. Although permitted in PHP 7.0 and PHP 5.x, any of the following uses of `$this` will now generate an error as of PHP 7.1, if `$this` is used as:

    - A parameter
    - A `static` variable
    - A `global` variable
    - A variable used in `try...catch` blocks
    - A variable used in `foreach()`
    - As an argument to `unset()`
    - As a variable (that is, `$a = 'this'; echo $$a`)
    - Indirectly via reference

5. If you need to create an object instance but don't care to define a discreet class, you can use the generic `stdClass` which is built into PHP. `stdClass` allows you to define properties *on the fly* without having to define a discreet class that extends `stdClass`:

```
$obj = new stdClass();
```

6. This facility is used in a number of different places in PHP. As an example, when you use **PHP Data Objects** (**PDO**) to do a database query, one of the fetch modes is `PDO::FETCH_OBJ`. This mode returns instances of `stdClass` where the properties represent database table columns:

```
$stmt = $connection->pdo->query($sql);
$row  = $stmt->fetch(PDO::FETCH_OBJ);
```

How it works...

Take the example for the `Test` class shown in the preceding code snippet, and place the code in a file named `Test.php`. Create another file called `chap_04_oop_defining_class_test.php`. Add the following code:

```
require __DIR__ . '/Test.php';

$test = new Test();
echo $test->getTest();
echo PHP_EOL;

$test->setTest('ABC');
echo $test->getTest();
echo PHP_EOL;
```

The output will show the initial value of the `$test` property, followed by the new value modified by calling `setTest()`:



The next example has you define two classes, `Name` and `Address` in a single file `NameAddress.php`. You can call and use these two classes with the following code:

```
require __DIR__ . '/NameAddress.php';

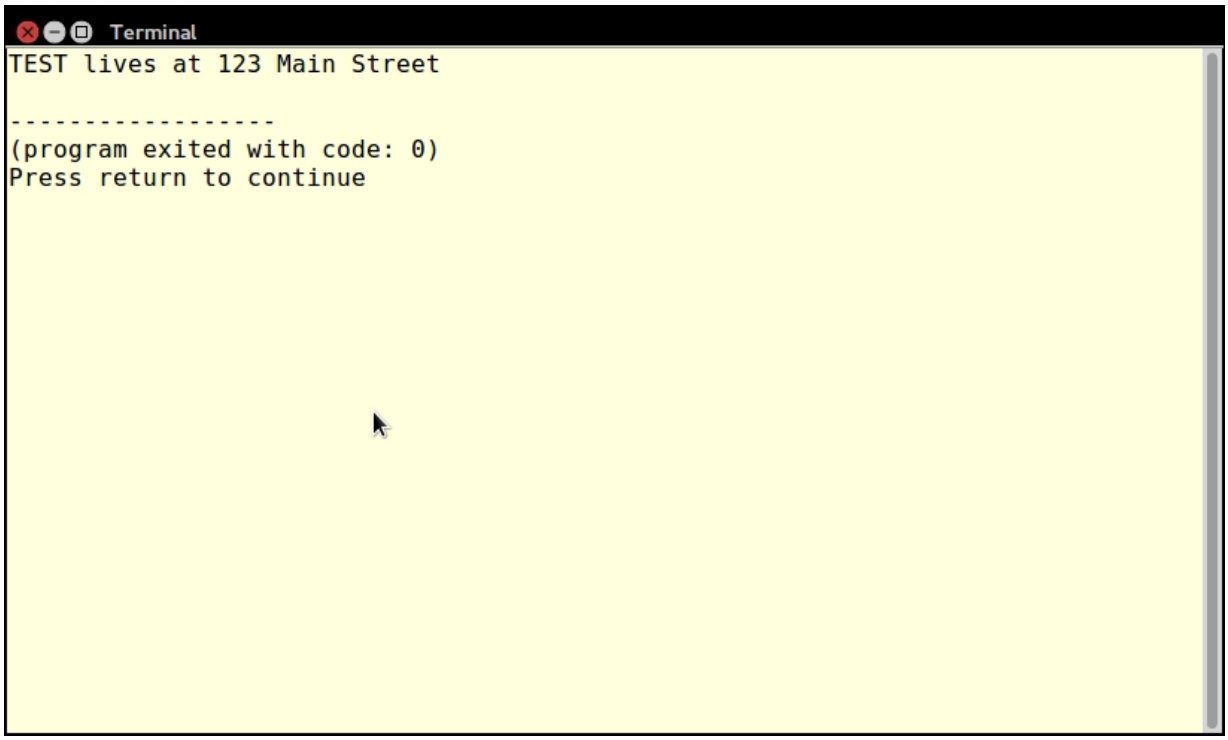
$name = new Name();
$name->setName('TEST');
$addr = new Address();
$addr->setAddress('123 Main Street');

echo $name->getName() . ' lives at ' . $addr-
>getAddress();
```

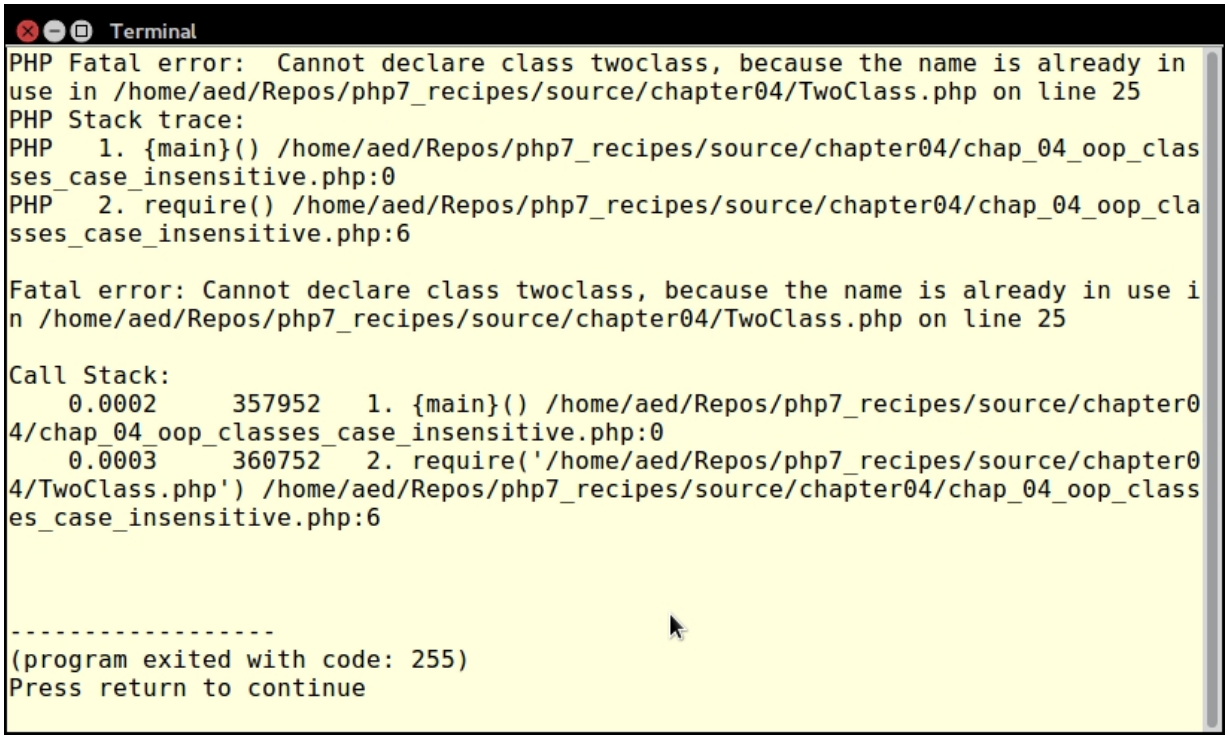
Note

Although no errors are generated by the PHP interpreter, by defining multiple classes, the logical purity of the file is compromised. Also, the filename doesn't match the classname, which could impact the ability to autoload.

The output from this example is shown next:



Step 3 also shows two class definitions in one file. In this case, however, the objective is to demonstrate that classnames in PHP are case-insensitive. Place the code into a file, `TwoClass.php`. When you try to include the file, an error is generated:



To demonstrate the direct use of `stdClass`, create an instance, assign a value to a property, and use `var_dump()` to display the results. To see how `stdClass` is used internally, use `var_dump()` to display the results of a `PDO` query where the fetch mode is set to `FETCH_OBJ`.

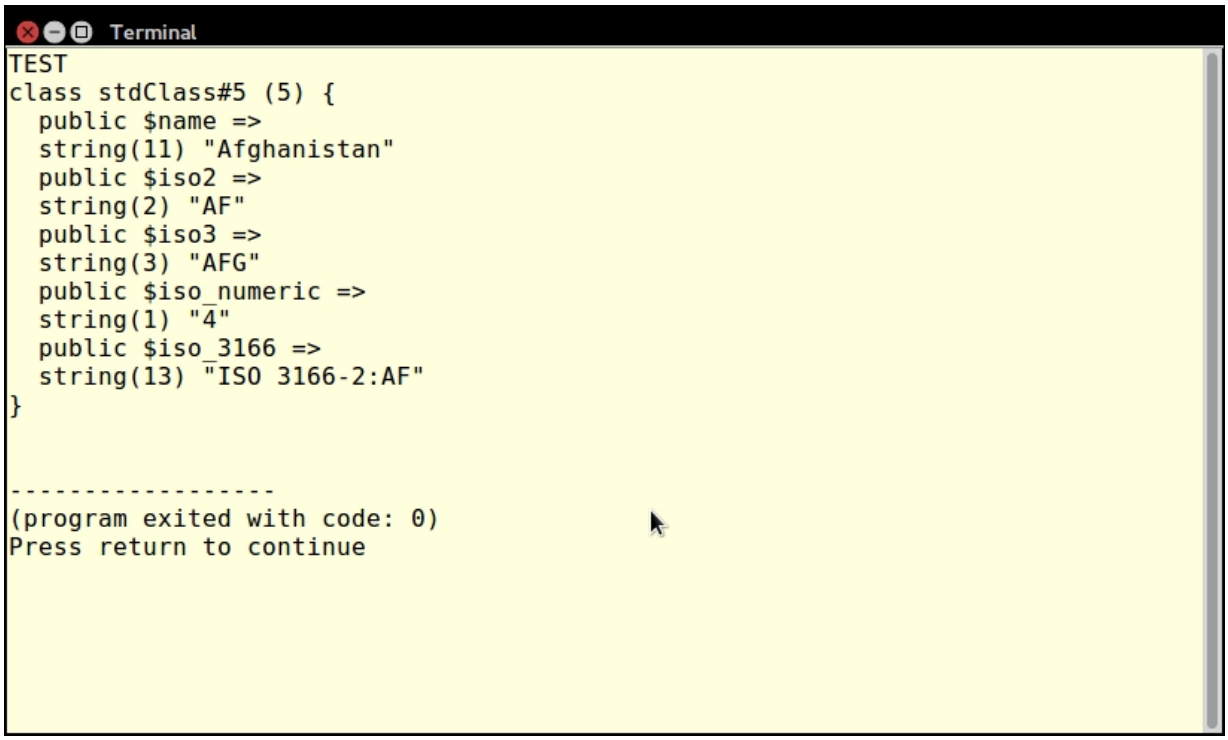
Enter the following code:

```
$obj = new stdClass();
$obj->test = 'TEST';
echo $obj->test;
echo PHP_EOL;

include (__DIR__ .
'/../Application/Database/Connection.php');
$connection = new Application\Database\Connection(
  include __DIR__ . DB_CONFIG_FILE);

$sql  = 'SELECT * FROM iso_country_codes';
$stmt = $connection->pdo->query($sql);
$row  = $stmt->fetch(PDO::FETCH_OBJ);
var_dump($row);
```

Here is the output:



See also...

For more information on refinements in PHP 7.1 on the keyword `$this`, please see https://wiki.php.net/rfc/this_var.

Extending classes

One of the primary reasons developers use OOP is because of its ability to re-use existing code, yet, at the same time, add or override functionality. In PHP, the keyword `extends` is used to establish a parent/child relationship between classes.

How to do it...

1. In the `child` class, use the keyword `extends` to set up inheritance. In the example that follows, the `Customer` class extends the `Base` class. Any instance of `Customer` will inherit visible methods and properties, in this case, `$id`, `getId()` and `setId()`:

```
class Base
{
  protected $id;
  public function getId()
  {
    return $this->id;
  }
  public function setId($id)
  {
    $this->id = $id;
  }
}

class Customer extends Base
{
  protected $name;
  public function getName()
  {
    return $this->name;
  }
  public function setName($name)
  {
    $this->name = $name;
  }
}
```

2. You can force any developer using your class to define a method by marking it `abstract`. In this example, the `Base` class defines as `abstract` the `validate()` method. The reason why it must be abstract is because it would be impossible to determine exactly how a child class would be validated from the perspective of the parent `Base` class:

```
abstract class Base
{
  protected $id;
  public function getId()
  {
    return $this->id;
  }
  public function setId($id)
  {
    $this->id = $id;
  }
  public function validate();
}
```

> ## Note
>
> If a class contains an **abstract method**, the class itself must be declared as `abstract`.

3. PHP only supports a single line of inheritance. The next example shows a class, `Member`, which inherits from `Customer`. `Customer`, in turn, inherits from `Base`:

```
class Base
{
  protected $id;
  public function getId()
  {
    return $this->id;
  }
  public function setId($id)
  {
    $this->id = $id;
```

```
  }
}

class Customer extends Base
{
  protected $name;
  public function getName()
  {
    return $this->name;
  }
  public function setName($name)
  {
    $this->name = $name;
  }
}

class Member extends Customer
{
  protected $membership;
  public function getMembership()
  {
    return $this->membership;
  }
  public function setMembership($memberId)
  {
    $this->membership = $memberId;
  }
}
```

4. To satisfy a type-hint, any child of the target class can be used. The `test()` function, shown in the following code snippet, requires an instance of the `Base` class as an argument. Any class within the line of inheritance can be accepted as an argument. Anything else passed to `test()` throws a `TypeError`:

```
function test(Base $object)
{
  return $object->getId();
}
```

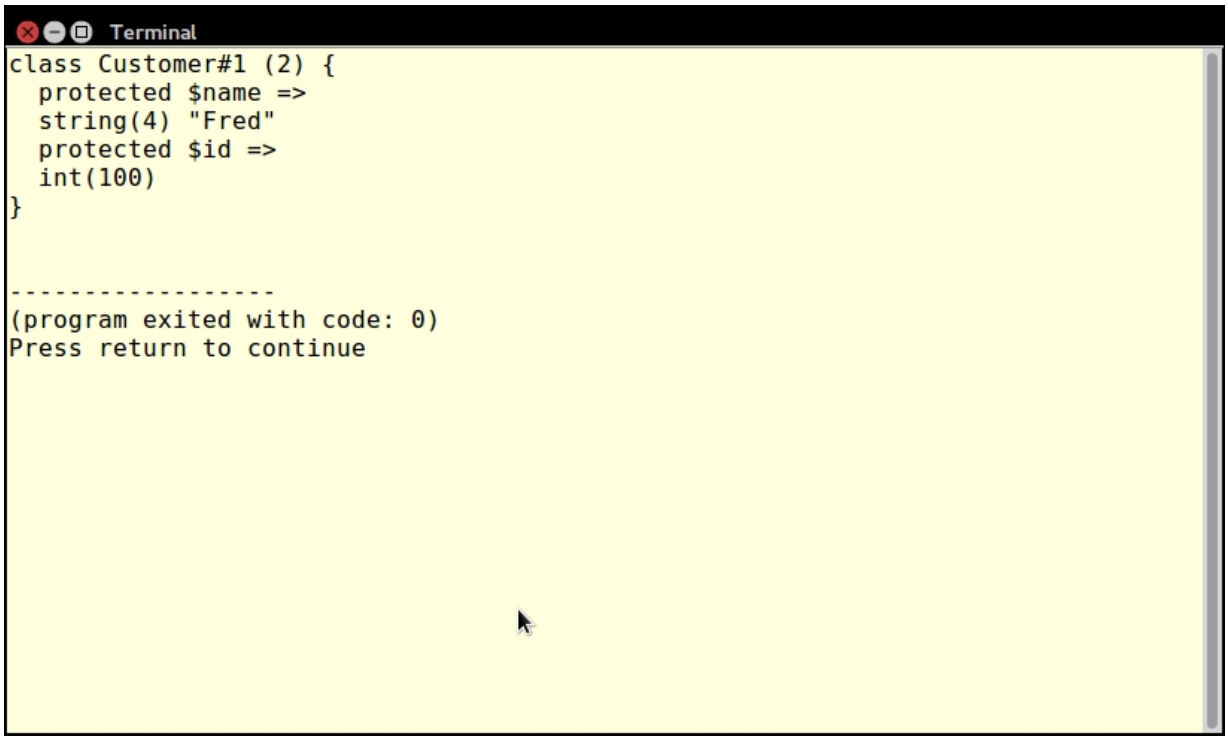
How it works...

In the first bullet point, a `Base` class and a `Customer` class were defined. For the sake of demonstration, place these two class

definitions in a single file, `chap_04_oop_extends.php`, and add the following code:

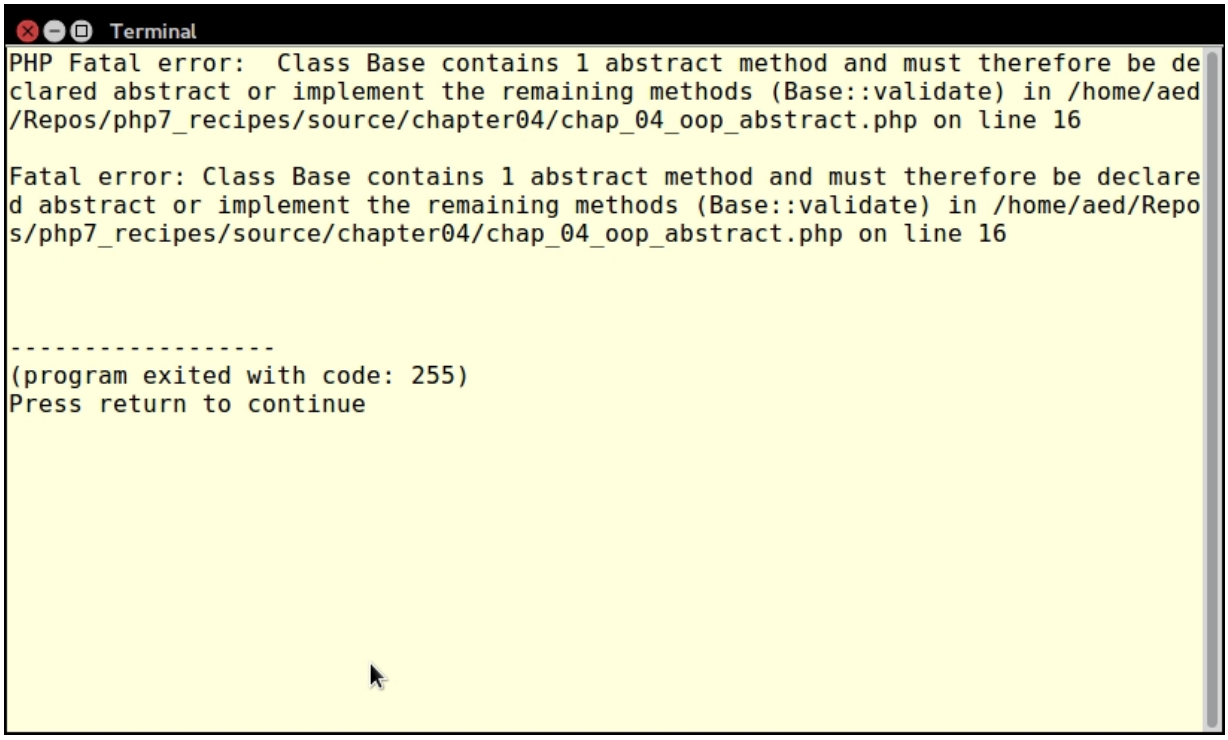
```
$customer = new Customer();
$customer->setId(100);
$customer->setName('Fred');
var_dump($customer);
```

Note that the `$id` property and the `getId()` and `setId()` methods are inherited from the parent `Base` class into the child `Customer` class:



To illustrate the use of an `abstract` method, imagine that you wish to add some sort of validation capability to any class that extends `Base`. The problem is that there is no way to know what might be validated in the inherited classes. The only thing that is certain is that you must have a validation capability.

Take the same `Base` class mentioned in the preceding explanation and add a new method, `validate()`. Label the method as `abstract`, and do not define any code. Notice what happens when the child `Customer` class extends `Base`.



If you then label the `Base` class as `abstract`, but fail to define a `validate()` method in the child class, the *same error* will be generated. Finally, go ahead and implement the `validate()` method in a child `Customer` class:

```
class Customer extends Base
{
  protected $name;
  public function getName()
  {
    return $this->name;
  }
  public function setName($name)
  {
    $this->name = $name;
  }
  public function validate()
  {
    $valid = 0;
    $count = count(get_object_vars($this));
    if (!empty($this->id) &&is_int($this->id))
$valid++;
    if (!empty($this->name)
    &&preg_match('/[a-z0-9 ]/i', $this->name))
```

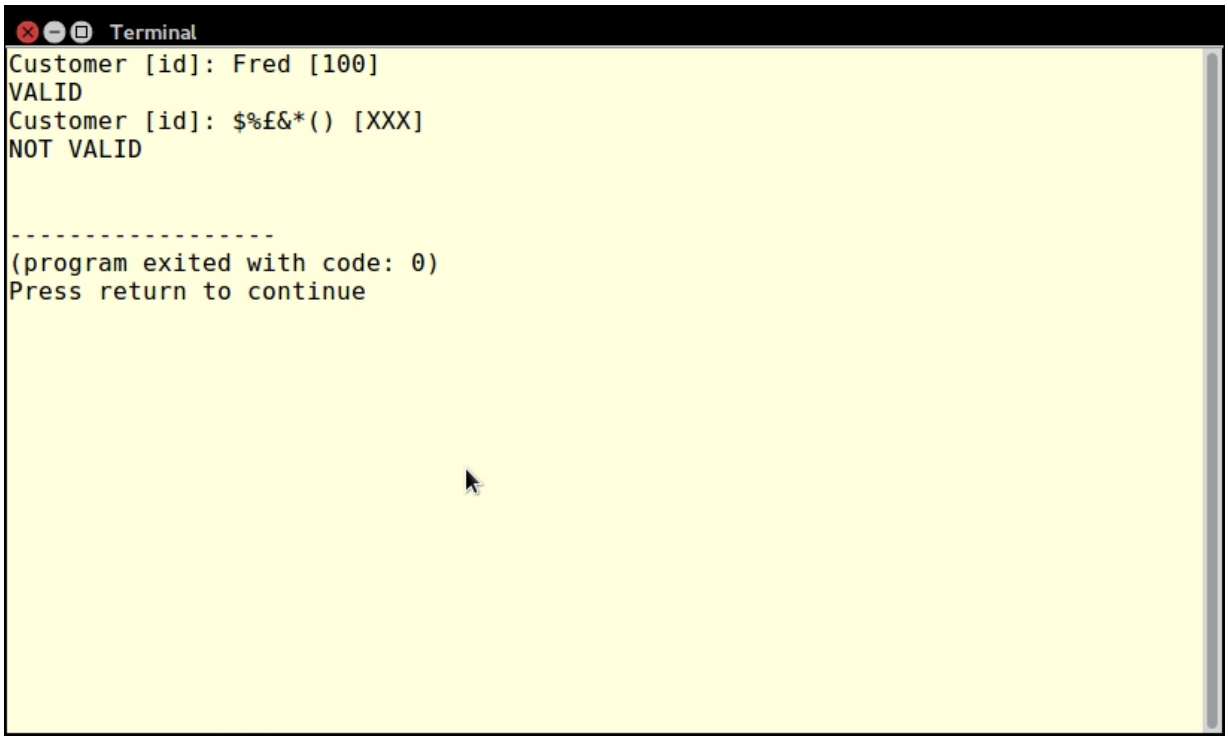
```
        $valid++;
            return ($valid == $count);
          }
        }
```

You can then add the following procedural code to test the results:

```
        $customer = new Customer();

        $customer->setId(100);
        $customer->setName('Fred');
        echo "Customer [id]: {$customer->getName()}" .
             . "[{$customer->getId()}]\n";
        echo ($customer->validate()) ? 'VALID' : 'NOT
        VALID';
        $customer->setId('XXX');
        $customer->setName('$%£&*()');
        echo "Customer [id]: {$customer->getName()}"
          . "[{$customer->getId()}]\n";
        echo ($customer->validate()) ? 'VALID' : 'NOT
        VALID';
```

Here is the output:



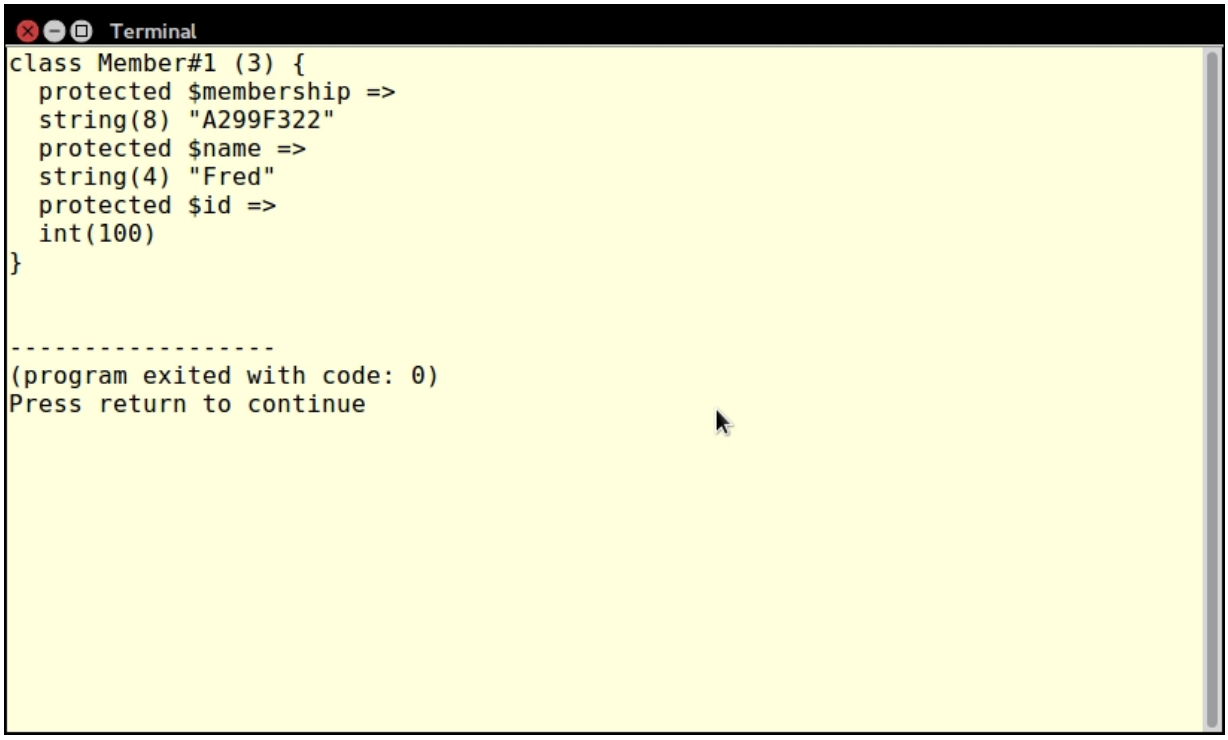
To show a single line of inheritance, add a new `Member` class to the first example of `Base` and `Customer` shown in the preceding step 1:

```
class Member extends Customer
{
  protected $membership;
  public function getMembership()
  {
    return $this->membership;
  }
  public function setMembership($memberId)
  {
    $this->membership = $memberId;
  }
}
```

Create an instance of `Member`, and notice, in the following code, that all properties and methods are available from every inherited class, even if not directly inherited:

```
$member = new Member();
$member->setId(100);
$member->setName('Fred');
$member->setMembership('A299F322');
var_dump($member);
```

Here is the output:



Now define a function, `test()`, which takes an instance of `Base` as an argument:

```
function test(Base $object)
{
  return $object->getId();
}
```

Notice that instances of `Base`, `Customer`, and `Member` are all acceptable as arguments:

```
$base = new Base();
$base->setId(100);

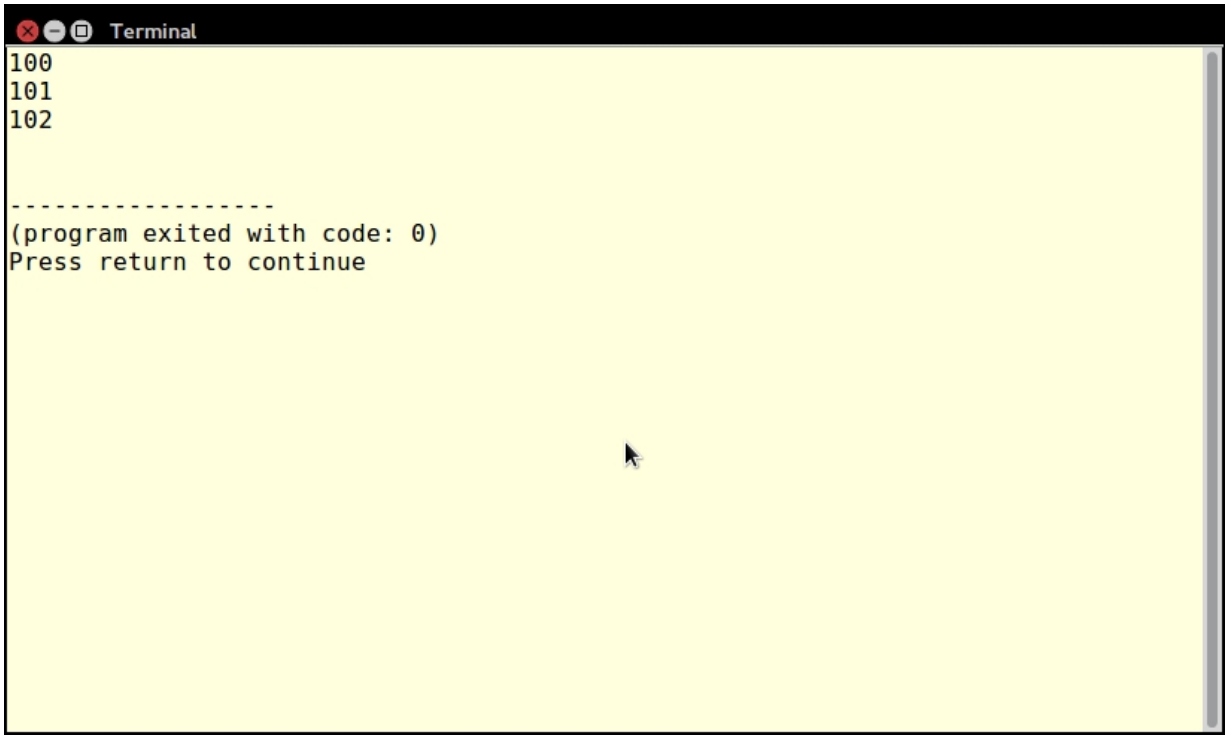
$customer = new Customer();
$customer->setId(101);

$member = new Member();
$member->setId(102);

// all 3 classes work in test()
echo test($base)     . PHP_EOL;
```

```
echo test($customer) . PHP_EOL;
echo test($member)   . PHP_EOL;
```

Here is the output:



However, if you try to run `test()` with an object instance that is not in the line of inheritance, a `TypeError` is thrown:

```
class Orphan
{
  protected $id;
  public function getId()
  {
    return $this->id;
  }
  public function setId($id)
  {
    $this->id = $id;
  }
}
try {
    $orphan = new Orphan();
    $orphan->setId(103);
    echo test($orphan) . PHP_EOL;
```

```
    } catch (TypeError $e) {
        echo 'Does not work!' . PHP_EOL;
        echo $e->getMessage();
    }
```

We can observe this in the following image:



Using static properties and methods

PHP lets you access properties or methods without having to create an instance of the class. The keyword used for this purpose is **static**.

How to do it...

1. At its simplest, simply add the `static` keyword after stating the visibility level when declaring an ordinary property or method. Use the `self` keyword to reference the property internally:

```
class Test
{
  public static $test = 'TEST';
  public static function getTest()
  {
    return self::$test;
  }
}
```

2. The `self` keyword will bind early, which will cause problems when accessing static information in child classes. If you absolutely need to access information from the child class, use the `static` keyword in place of `self`. This process is referred to as **Late Static Binding**.
3. In the following example, if you echo `Child::getEarlyTest()`, the output will be **TEST**. If, on the other hand, you run `Child::getLateTest()`, the output will be **CHILD**. The reason is that PHP will bind to the *earliest* definition when using `self`, whereas the *latest* binding is used for the `static` keyword:

```
class Test2
{
  public static $test = 'TEST2';
  public static function getEarlyTest()
  {
    return self::$test;
```

```
  }
  public static function getLateTest()
  {
    return static::$test;
  }
}

class Child extends Test2
{
  public static $test = 'CHILD';
}
```

4. In many cases, the **Factory** design pattern is used in conjunction with static methods to produce instances of objects given different parameters. In this example, a static method `factory()` is defined which returns a PDO connection:

```
public static function factory(
  $driver,$dbname,$host,$user,$pwd,array
$options = [])
  {
    $dsn = sprintf('%s:dbname=%s;host=%s',
    $driver, $dbname, $host);
    try {
        return new PDO($dsn, $user, $pwd,
$options);
    } catch (PDOException $e) {
        error_log($e->getMessage);
    }
  }
```

How it works...

You can reference static properties and methods using the **class resolution operator** "`::`". Given the `Test` class shown previously, if you run this code:

```
echo Test::$test;
echo PHP_EOL;
echo Test::getTest();
echo PHP_EOL;
```

You will see this output:



To illustrate Late Static Binding, based on the classes `Test2` and `Child` shown previously, try this code:

```
echo Test2::$test;
echo Child::$test;
echo Child::getEarlyTest();
echo Child::getLateTest();
```

The output illustrates the difference between `self` and `static`:



Finally, to test the `factory()` method shown previously, save the code into the `Application\Database\Connection` class in a `Connection.php` file in the `Application\Database` folder. You can then try this:

```
include __DIR__ .
'/../Application/Database/Connection.php';
use Application\Database\Connection;
$connection = Connection::factory(
'mysql', 'php7cookbook', 'localhost', 'test',
'password');
$stmt = $connection->query('SELECT name FROM
iso_country_codes');
while ($country = $stmt->fetch(PDO::FETCH_COLUMN))
echo $country . '';
```

You will see a list of countries pulled from the sample database:



See also

For more information on Late Static Binding, see this explanation in the PHP documentation:

http://php.net/manual/en/language.oop5.late-static-bindings.php

Using namespaces

An aspect that is critical to advanced PHP development is the use of namespaces. The arbitrarily defined namespace becomes a prefix to the class name, thereby avoiding the problem of accidental class duplication, and allowing you extraordinary freedom of development. Another benefit to the use of a namespace, assuming it matches the directory structure, is that it facilitates autoloading, as discussed in Chapter 1, *Building a Foundation*.

How to do it...

1. To define a class within a namespace, simply add the keyword `namespace` at the top of the code file:

```
namespace Application\Entity;
```

> **Note**
>
> **Best practice**
>
> As with the recommendation to have only one class per file, likewise you should have only one namespace per file.

2. The only PHP code that should precede the keyword `namespace` would be a comment and/or the keyword `declare`:

```
<?php
declare(strict_types=1);
namespace Application\Entity;
/**
 * Address
 *
 */
class Address
{
```

```
  // some code
}
```

3. In PHP 5, if you needed to access a class in an external namespace you could prepend a `use` statement containing only the namespace. You would need to then prefix any class reference within this namespace with the last component of the namespace:

```
use Application\Entity;
$name = new Entity\Name();
$addr = new Entity\Address();
$prof = new Entity\Profile();
```

4. Alternatively, you could distinctly specify all three classes:

```
use Application\Entity\Name;
use Application\Entity\Address;
use Application\Entity\Profile;
$name = new Name();
$addr = new Address();
$prof = new Profile();
```

5. PHP 7 has introduced a syntactical improvement referred to as **group use** which greatly improves code readability:

```
use Application\Entity\ {
  Name,
  Address,
  Profile
};
$name = new Name();
$addr = new Address();
$prof = new Profile();
```

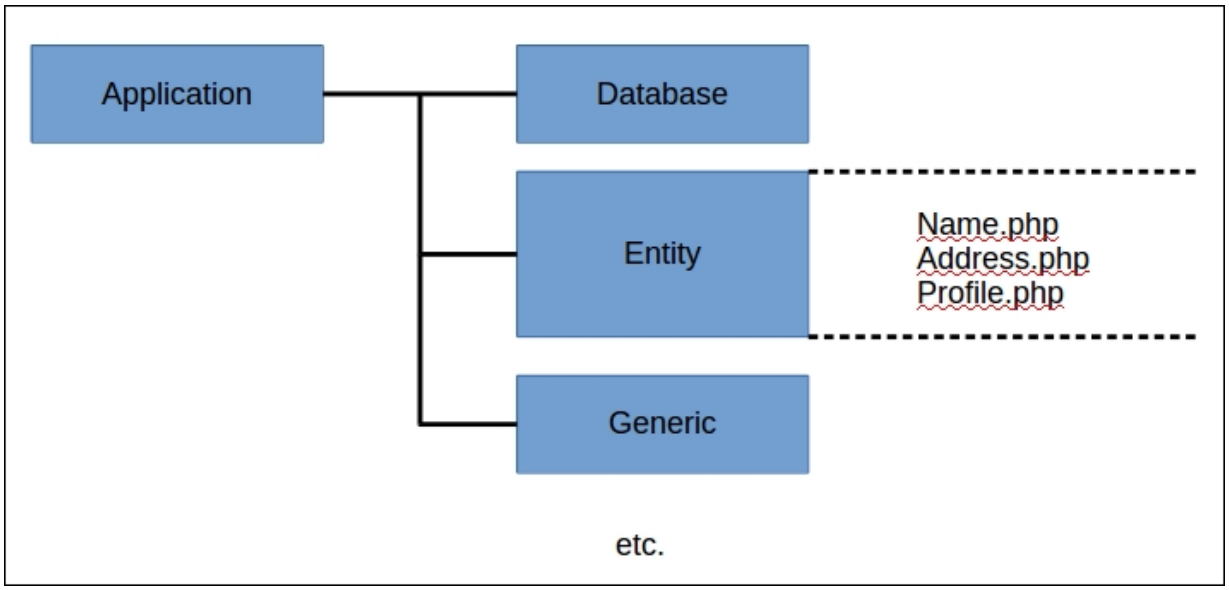
6. As mentioned in Chapter 1, *Building a Foundation*, namespaces form an integral part of the **autoloading** process. This example shows a demonstration autoloader which echoes the argument passed, and then attempts to include a file based on the namespace and class name. This assumes that the directory structure matches the namespace:

```
function __autoload($class)
{
  echo "Argument Passed to Autoloader =
```

```
$class\n";
  include __DIR__ . '/../' . str_replace('\\',
DIRECTORY_SEPARATOR, $class) . '.php';
}
```

How it works...

For illustration purposes, define a directory structure that matches the `Application\*` namespace. Create a base folder `Application`, and a sub-folder `Entity`. You can also include any sub-folders as desired, such as `Database` and `Generic`, used in other chapters:



Next, create three `entity` classes, each in their own file, under the `Application/Entity` folder: `Name.php`, `Address.php`, and `Profile.php`. We only show `Application\Entity\Name` here. `Application\Entity\Address` and `Application\Entity\Profile` will be the same, except that `Address` has an `$address` property, and `Profile` has a `$profile` property, each with an appropriate `get` and `set` method:

```
<?php
declare(strict_types=1);
namespace Application\Entity;
/**
 * Name
```

```
 *
 */
class Name
{

  protected $name = '';

  /**
   * This method returns the current value of
$name
   *
   * @return string $name
   */
  public function getName() : string
  {
    return $this->name;
  }

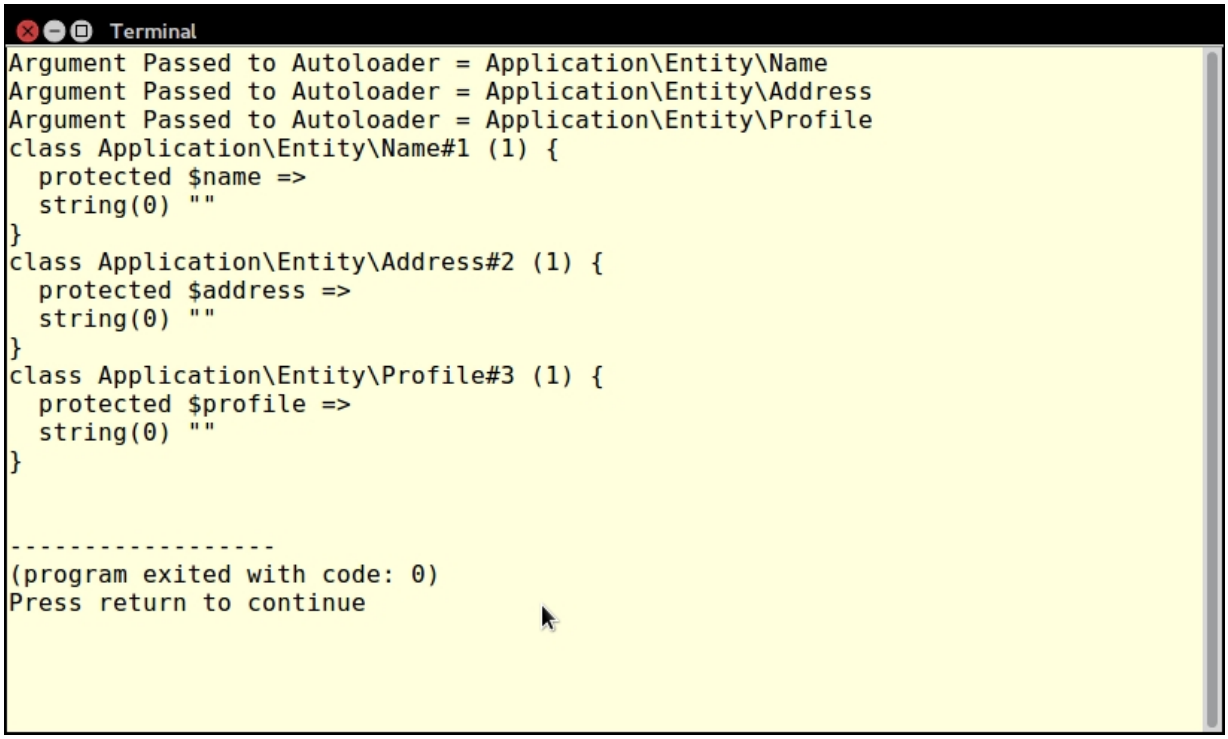
  /**
   * This method sets the value of $name
   *
   * @param string $name
   * @return name $this
   */
  public function setName(string $name)
  {
    $this->name = $name;
    return $this;
  }
}
```

You can then either use the autoloader defined in Chapter 1, *Building a Foundation*, or use the simple autoloader mentioned previously. Place the commands to set up autoloading in a file, `chap_04_oop_namespace_example_1.php`. In this file, you can then specify a use statement which only references the namespace, not the class names. Create instances of the three entity classes `Name`, `Address` and `Profile`, by prefixing the class name with the last part of the namespace, `Entity`:

```
use Application\Entity;
$name = new Entity\Name();
$addr = new Entity\Address();
$prof = new Entity\Profile();
```

```
var_dump($name);
var_dump($addr);
var_dump($prof);
```

Here is the output:



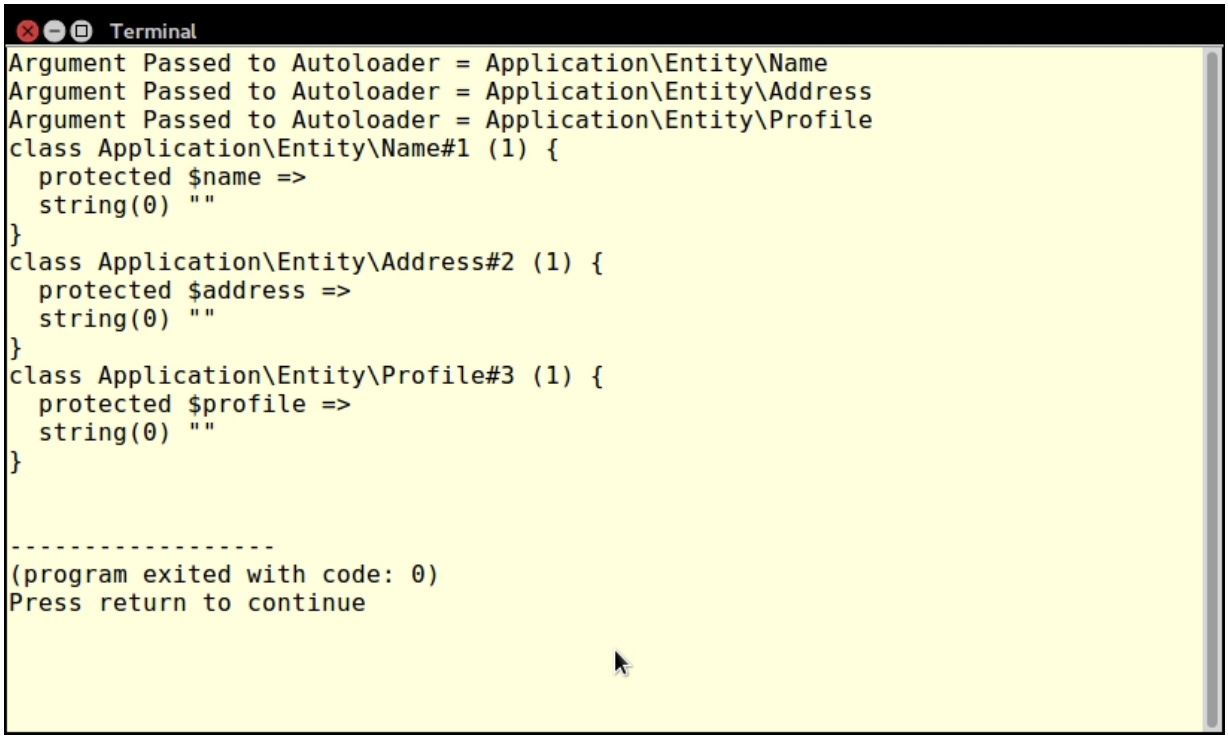
Next, use **Save as** to copy the file to a new one named `chap_04_oop_namespace_example_2.php`. Change the `use` statement to the following:

```
use Application\Entity\Name;
use Application\Entity\Address;
use Application\Entity\Profile;
```

You can now create class instances using only the class name:

```
$name = new Name();
$addr = new Address();
$prof = new Profile();
```

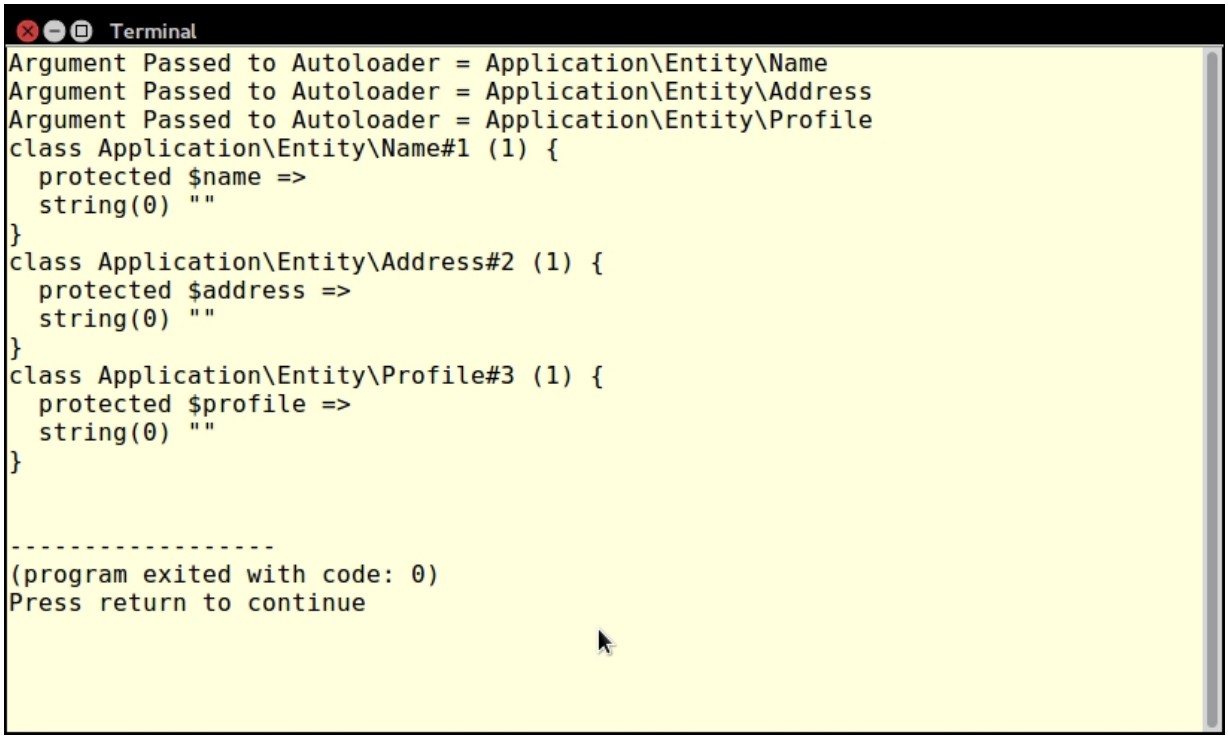
When you run this script, here is the output:



Finally, again run **Save as** and create a new file, `chap_04_oop_namespace_example_3.php`. You can now test the **group use** feature introduced in PHP 7:

```
use Application\Entity\ {
  Name,
  Address,
  Profile
};
$name = new Name();
$addr = new Address();
$prof = new Profile();
```

Again, when you run this block of code, the output will be the same as the preceding output:


Terminal
Argument Passed to Autoloader = Application\Entity\Name
Argument Passed to Autoloader = Application\Entity\Address
Argument Passed to Autoloader = Application\Entity\Profile
class Application\Entity\Name#1 (1) {
protected $name =>
string(0) ""
}
class Application\Entity\Address#2 (1) {
protected $address =>
string(0) ""
}
class Application\Entity\Profile#3 (1) {
protected $profile =>
string(0) ""
}

(program exited with code: 0)
Press return to continue

Defining visibility

Deceptively, the word *visibility* has nothing to do with application security! Instead it is simply a mechanism to control the use of your code. It can be used to steer an inexperienced developer away from the *public* use of methods that should only be called inside the class definition.

How to do it...

1. Indicate the visibility level by prepending the `public`, `protected`, or `private` keyword in front of any property or method definition. You can label properties as `protected` or `private` to enforce access only through public `getters` and `setters`.
2. In this example, a `Base` class is defined with a protected property `$id`. In order to access this property, the `getId()` and `setId()` public methods are defined. The protected method `generateRandId()` can be used internally, and is inherited in the `Customer` child class. This method cannot be called directly outside of class definitions. Note the use of the new PHP 7 `random_bytes()` function to create a random ID.

```
class Base
{
  protected $id;
  private $key = 12345;
  public function getId()
  {
    return $this->id;
  }
  public function setId()
  {
    $this->id = $this->generateRandId();
  }
  protected function generateRandId()
  {
    return unpack('H*', random_bytes(8))[1];
  }
}
```

```
class Customer extends Base
{
  protected $name;
  public function getName()
  {
    return $this->name;
  }
  public function setName($name)
  {
    $this->name = $name;
  }
}
```

Note

Best practice

Mark properties as protected, and define the public getNameOfProperty() and setNameOfProperty() methods to control access to the property. Such methods are referred to as `getters` and `setters`.

3. Mark a property or method as `private` to prevent it from being inherited or visible from *outside* the class definition. This is a good way to create a class as a **singleton**.
4. The next code example shows a class `Registry`, of which there can only be one instance. Because the constructor is marked as `private`, the only way an instance can be created is through the static method `getInstance()`:

```
class Registry
{
  protected static $instance = NULL;
  protected $registry = array();
  private function __construct()
  {
    // nobody can create an instance of this
class
  }
  public static function getInstance()
```

```
    {
      if (!self::$instance) {
        self::$instance = new self();
      }
      return self::$instance;
    }
    public function __get($key)
    {
      return $this->registry[$key] ?? NULL;
    }
    public function __set($key, $value)
    {
      $this->registry[$key] = $value;
    }
}
```

> ## Note
>
> You can mark a method as `final` to prevent it from being overridden. Mark a class as `final` to prevent it from being extended.

5. Normally, class constants are considered to have a visibility level of `public`. As of PHP 7.1, you can declare class constants to be `protected` or `private`. In the following example, the `TEST_WHOLE_WORLD` class constant behaves exactly as in PHP 5. The next two constants, `TEST_INHERITED` and `TEST_LOCAL`, follow the same rules as any `protected` or `private` property or method:

```
class Test
{

  public const TEST_WHOLE_WORLD  =
'visible.everywhere';

  // NOTE: only works in PHP 7.1 and above
  protected const TEST_INHERITED =
'visible.in.child.classes';

  // NOTE: only works in PHP 7.1 and above
  private const TEST_LOCAL=
```

```
'local.to.class.Test.only';

  public static function getTestInherited()
  {
    return static::TEST_INHERITED;
  }

  public static function getTestLocal()
  {
    return static::TEST_LOCAL;
  }

}
```

How it works...

Create a file `chap_04_basic_visibility.php` and define two classes: `Base` and `Customer`. Next, write code to create instances of each:

```
$base     = new Base();
$customer = new Customer();
```

Notice that the following code works OK, and is in fact considered the best practice:

```
$customer->setId();
$customer->setName('Test');
echo 'Welcome ' . $customer->getName() . PHP_EOL;
echo 'Your new ID number is: ' . $customer-
>getId() . PHP_EOL;
```

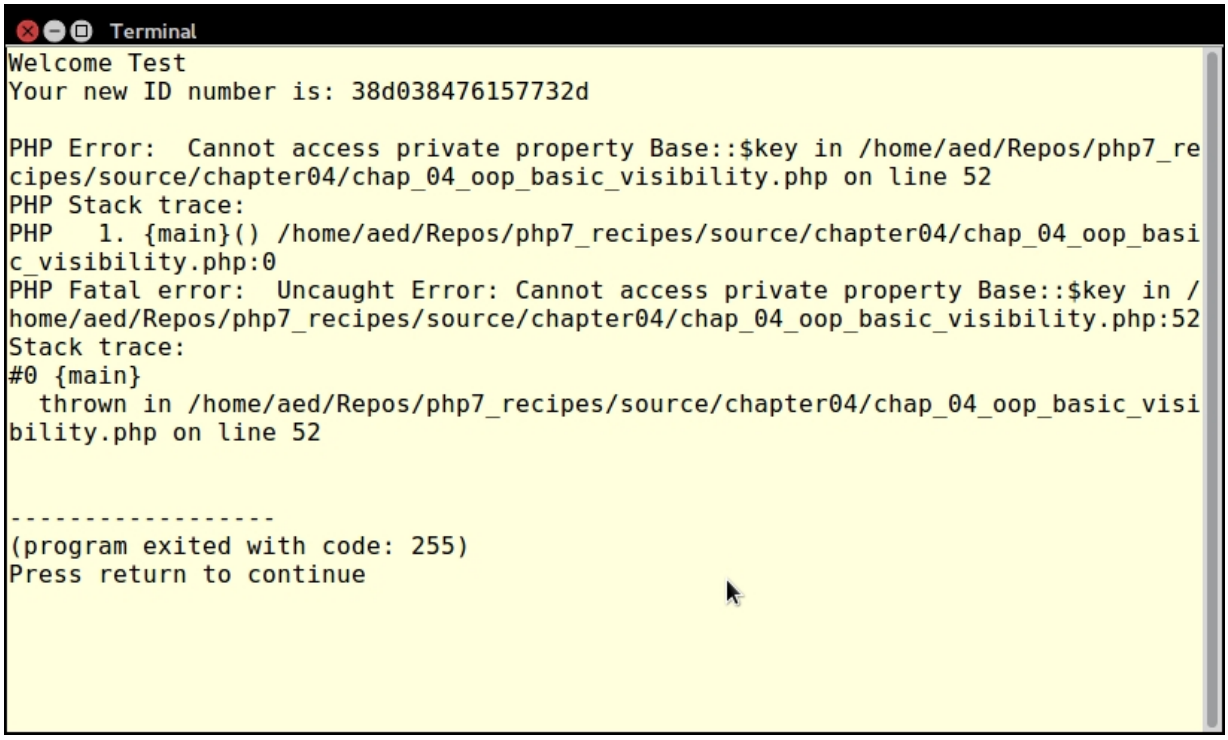
Even though `$id` is `protected`, the corresponding methods, `getId()` and `setId()`, are both `public`, and therefore accessible from outside the class definition. Here is the output:



The following lines of code will not work, however, as `private` and `protected` properties are not accessible from outside the class definition:

```
echo 'Key (does not work): ' . $base->key;
echo 'Key (does not work): ' . $customer->key;
echo 'Name (does not work): ' . $customer->name;
echo 'Random ID (does not work): ' . $customer-
>generateRandId();
```

The following output shows the expected errors:



See also

For more information on `getters` and `setters`, see the recipe in this chapter entitled *Using getters and setters*. For more information on PHP 7.1 class constant visibility settings, please see https://wiki.php.net/rfc/class_const_visibility.

Using interfaces

Interfaces are useful tools for systems architects and are often used to prototype an **Application Programming Interface** (**API**). Interfaces don't contain actual code, but can contain names of methods as well as method signatures.

> ## Note
>
> All methods identified in the `Interface` have a visibility level of `public`.

How to do it...

1. Methods identified by the interface cannot contain actual code implementations. You can, however, specify the data types of method arguments.
2. In this example, `ConnectionAwareInterface` identifies a method, `setConnection()`, which requires an instance of `Connection` as an argument:

```
interface ConnectionAwareInterface
{
  public function setConnection(Connection
$connection);
}
```

3. To use the interface, add the keyword `implements` after the open line that defines the class. We have defined two classes, `CountryList` and `CustomerList`, both of which require access to the `Connection` class via a method, `setConnection()`. In order to identify this dependency, both classes implement `ConnectionAwareInterface`:

```
class CountryList implements
ConnectionAwareInterface
```

```
{
  protected $connection;
  public function setConnection(Connection
$connection)
  {
    $this->connection = $connection;
  }
  public function list()
  {
    $list = [];
    $stmt = $this->connection->pdo->query(
      'SELECT iso3, name FROM
iso_country_codes');
    while ($country = $stmt-
>fetch(PDO::FETCH_ASSOC)) {
      $list[$country['iso3']] =
$country['name'];
    }
    return $list;
  }

}
class CustomerList implements
ConnectionAwareInterface
{
  protected $connection;
  public function setConnection(Connection
$connection)
  {
    $this->connection = $connection;
  }
  public function list()
  {
    $list = [];
    $stmt = $this->connection->pdo->query(
      'SELECT id, name FROM customer');
    while ($customer = $stmt-
>fetch(PDO::FETCH_ASSOC)) {
      $list[$customer['id']] =
$customer['name'];
    }
    return $list;
  }

}
```

4. Interfaces can be used to satisfy a type hint. The following class, `ListFactory`, contains a `factory()` method, which initializes any class that implements `ConnectionAwareInterface`. The interface is a guarantee that the `setConnection()` method is defined. Setting the type hint to the interface instead of a specific class instance makes the `factory` method more generically useful:

```
namespace Application\Generic;

use PDO;
use Exception;
use Application\Database\Connection;
use
Application\Database\ConnectionAwareInterface;

class ListFactory
{
  const ERROR_AWARE = 'Class must be
Connection Aware';
  public static function factory(
    ConnectionAwareInterface $class,
$dbParams)
  {
    if ($class instanceof
ConnectionAwareInterface) {
        $class->setConnection(new
Connection($dbParams));
        return $class;
    } else {
        throw new
Exception(self::ERROR_AWARE);
    }
    return FALSE;
  }
}
```

5. If a class implements multiple interfaces, a **naming collision** occurs if method signatures do not match. In this example, there are two interfaces, `DateAware` and `TimeAware`. In addition to defining the `setDate()` and `setTime()` methods, they both define `setBoth()`. Having duplicate method names is not an issue, although it is not considered best practice. The problem lies in the fact that the method signatures differ:

```
interface DateAware
{
  public function setDate($date);
  public function setBoth(DateTime $dateTime);
}

interface TimeAware
{
  public function setTime($time);
  public function setBoth($date, $time);
}

class DateTimeHandler implements DateAware,
TimeAware
{
  protected $date;
  protected $time;
  public function setDate($date)
  {
    $this->date = $date;
  }
  public function setTime($time)
  {
    $this->time = $time;
  }
  public function setBoth(DateTime $dateTime)
  {
    $this->date = $date;
  }
}
```

6. As the code block stands, a fatal error will be generated (which cannot be caught!). To resolve the problem, the preferred approach would be to remove the definition of `setBoth()` from one or the other interface. Alternatively, you could adjust the method signatures to match.

Note

Best practice

Do not define interfaces with duplicate or overlapping method definitions.

How it works...

In the `Application/Database` folder, create a file, `ConnectionAwareInterface.php`. Insert the code discussed in the preceding step 2.

Next, in the `Application/Generic` folder, create two files, `CountryList.php` and `CustomerList.php`. Insert the code discussed in step 3.

Next, in a directory parallel to the `Application` directory, create a source code file, `chap_04_oop_simple_interfaces_example.php`, which initializes the autoloader and includes the database parameters:

```
<?php
define('DB_CONFIG_FILE',
'/../config/db.config.php');
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
$params = include __DIR__ . DB_CONFIG_FILE;
```

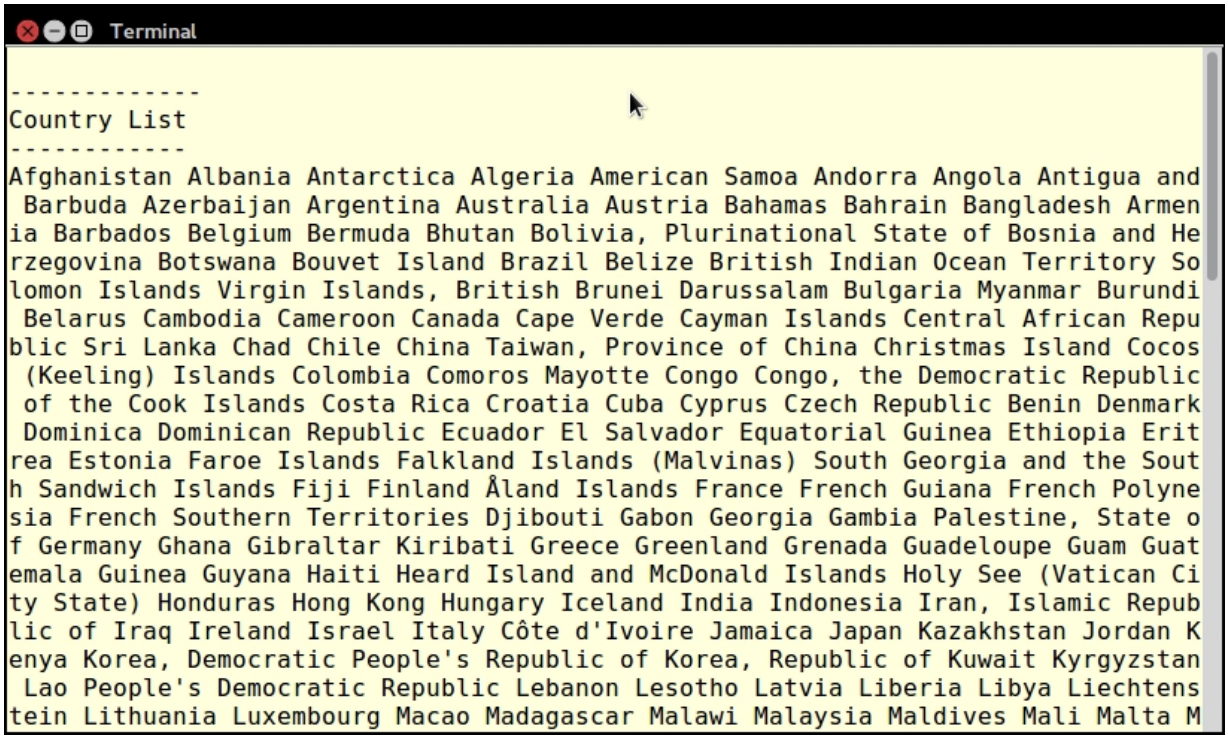
The database parameters in this example are assumed to be in a database configuration file indicated by the `DB_CONFIG_FILE` constant.

You are now in a position to use `ListFactory::factory()` to generate `CountryList` and `CustomerList` objects. Note that if these classes did not implement `ConnectionAwareInterface`, an error would be thrown:

```
  $list =
Application\Generic\ListFactory::factory(
    new Application\Generic\CountryList(),
$params);
```

```
  foreach ($list->list() as $item) echo $item .
'';
```

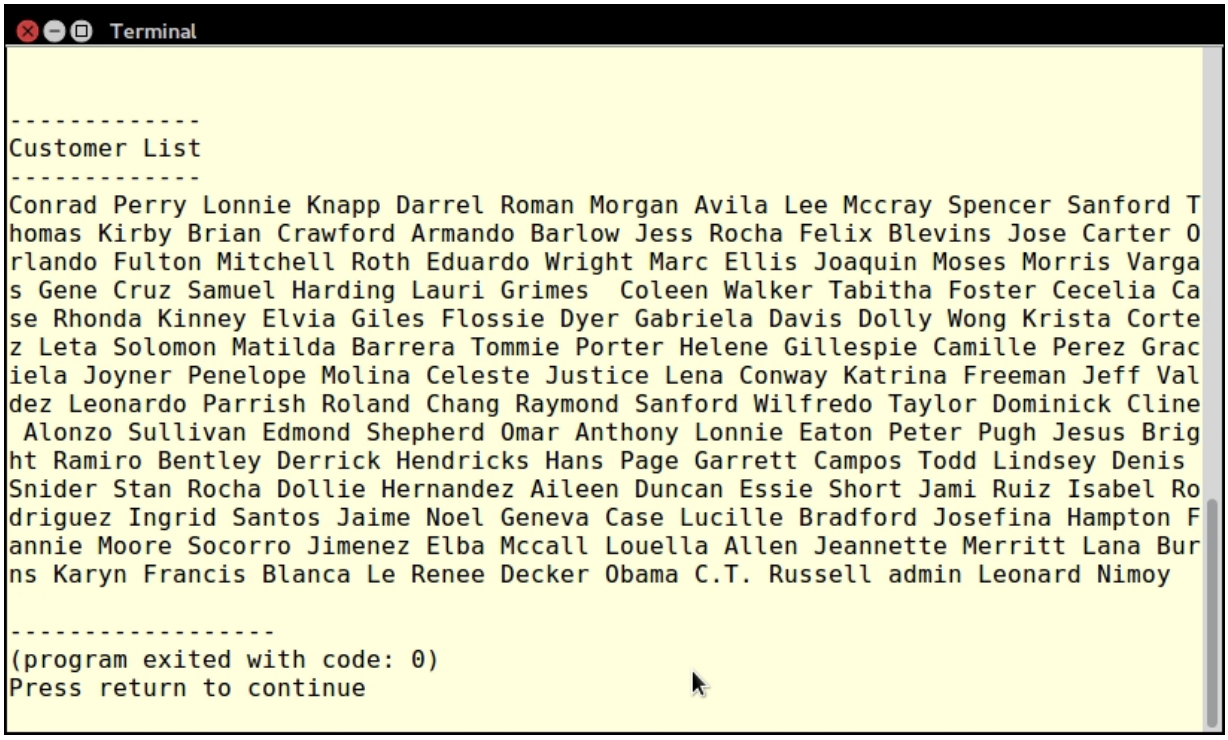
Here is the output for country list:



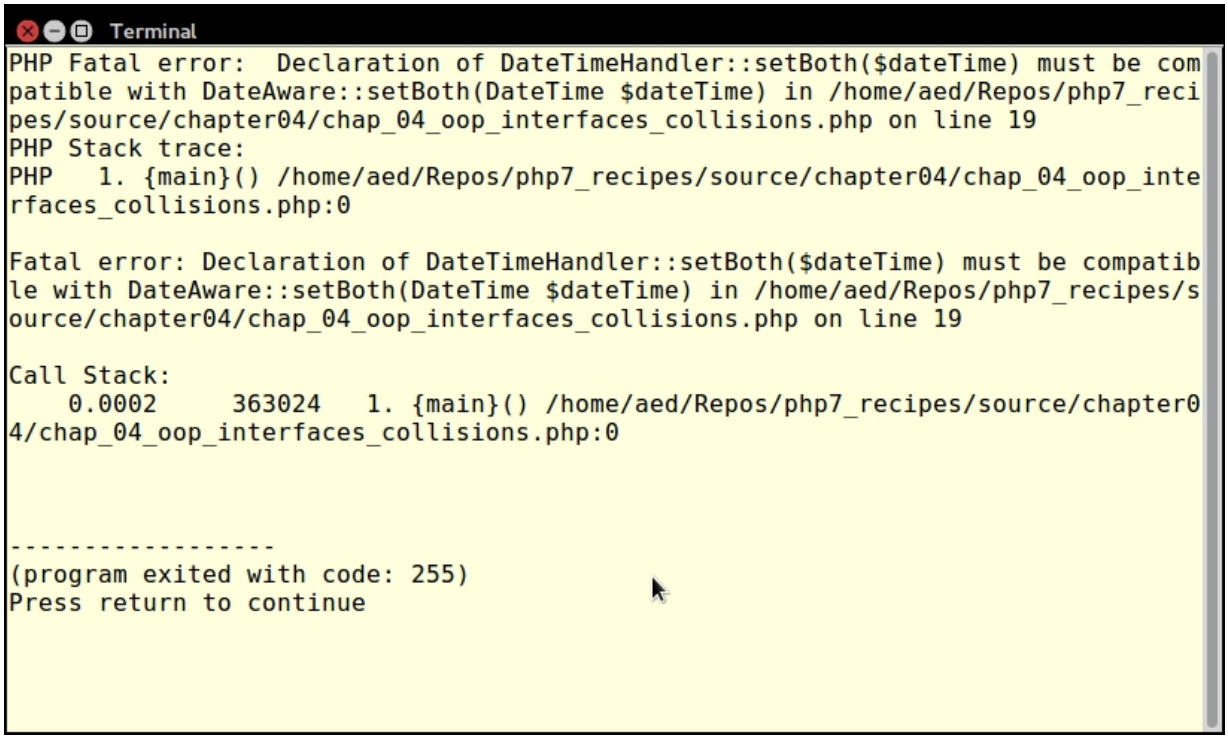
You can also use the `factory` method to generate a `CustomerList` object and use it:

```
  $list =
Application\Generic\ListFactory::factory(
    new Application\Generic\CustomerList(),
$params);
  foreach ($list->list() as $item) echo $item .
'';
```

Here is the output for `CustomerList`:



If you want to examine what happens when multiple interfaces are implemented, but where the method signature differs, enter the code shown in the preceding step 4 into a file, `chap_04_oop_interfaces_collisions.php`. When you try to run the file, an error is generated, as shown here:



If you make the following adjustment in the `TimeAware` interface, no errors will result:

```
interface TimeAware
{
  public function setTime($time);
  // this will cause a problem
  public function setBoth(DateTime $dateTime);
}
```

Using traits

If you have ever done any C programming, you are perhaps familiar with macros. A macro is a predefined block of code that *expands* at the line indicated. In a similar manner, traits can contain blocks of code that are copied and pasted into a class at the line indicated by the PHP interpreter.

How to do it...

1. Traits are identified with the keyword `trait`, and can contain properties and/or methods. You may have noticed duplication of code when examining the previous recipe featuring the `CountryList` and `CustomerList` classes. In this example, we will re-factor the two classes, and move the functionality of the `list()` method into a `Trait`. Notice that the `list()` method is the same in both classes.
2. Traits are used in situations where there is duplication of code between classes. Please note, however, that the conventional approach to creating an abstract class and extending it might have certain advantages over using traits. Traits cannot be used to identify a line of inheritance, whereas abstract parent classes can be used for this purpose.
3. We will now copy `list()` into a trait called `ListTrait`:

```
trait ListTrait
{
  public function list()
  {
    $list = [];
    $sql  = sprintf('SELECT %s, %s FROM %s',
      $this->key, $this->value, $this->table);
    $stmt = $this->connection->pdo-
>query($sql);
    while ($item = $stmt-
>fetch(PDO::FETCH_ASSOC)) {
      $list[$item[$this->key]] = $item[$this-
>value];
    }
```

```
        return $list;
      }
    }
```

4. We can then insert the code from `ListTrait` into a new class, `CountryListUsingTrait`, as shown in the following code snippet. The entire `list()` method can now be removed from this class:

```
class CountryListUsingTrait implements
ConnectionAwareInterface
{

  use ListTrait;

  protected $connection;
  protected $key   = 'iso3';
  protected $value = 'name';
  protected $table = 'iso_country_codes';

  public function setConnection(Connection
$connection)
  {
    $this->connection = $connection;
  }

}
```

> ## Note
>
> Any time you have duplication of code, a potential problem arises when you need to make a change. You might find yourself having to do too many global search and replace operations, or cutting and pasting of code, often with disastrous results. Traits are a great way to avoid this maintenance nightmare.

5. Traits are affected by namespaces. In the example shown in step 1, if our new `CountryListUsingTrait` class is placed into a namespace, `Application\Generic`, we will also need to move `ListTrait` into that namespace as well:

```
namespace Application\Generic;

use PDO;

trait ListTrait
{
  public function list()
  {
    // code as shown above
  }
}
```

6. Methods in traits override inherited methods.
7. In the following example, you will notice that the return value for the `setId()` method differs between the `Base` parent class and the `Test` trait. The `Customer` class inherits from `Base`, but also uses `Test`. In this case, the method defined in the trait will override the method defined in the `Base` parent class:

```
trait Test
{
  public function setId($id)
  {
    $obj = new stdClass();
    $obj->id = $id;
    $this->id = $obj;
  }
}

class Base
{
  protected $id;
  public function getId()
  {
    return $this->id;
  }
  public function setId($id)
  {
    $this->id = $id;
  }
}

class Customer extends Base
{
  use Test;
  protected $name;
```

```
  public function getName()
  {
    return $this->name;
  }
  public function setName($name)
  {
    $this->name = $name;
  }
}
```

Note

In PHP 5, traits could also override properties. In PHP 7, if the property in a trait is initialized to a different value than in the parent class, a fatal error is generated.

8. Methods directly defined in the class that use the trait override duplicate methods defined in the trait.
9. In this example, the `Test` trait defines a property `$id` along with the `getId()` methods and `setId()`. The trait also defines `setName()`, which conflicts with the same method defined in the `Customer` class. In this case, the directly defined `setName()` method from `Customer` will override the `setName()` defined in the trait:

```
trait Test
{
  protected $id;
  public function getId()
  {
    return $this->id;
  }
  public function setId($id)
  {
    $this->id = $id;
  }
  public function setName($name)
  {
    $obj = new stdClass();
    $obj->name = $name;
```

```
    $this->name = $obj;
  }
}

class Customer
{
  use Test;
  protected $name;
  public function getName()
  {
    return $this->name;
  }
  public function setName($name)
  {
    $this->name = $name;
  }
}
```

10. Use the `insteadof` keywords to resolve method name conflicts when using multiple traits. In conjunction, use the `as` keyword to alias method names.
11. In this example, there are two traits, `IdTrait` and `NameTrait`. Both traits define a `setKey()` method, but express the key in different ways. The `Test` class uses both traits. Note the `insteadof` keyword, which allows us to distinguish between the conflicting methods. Thus, when `setKey()` is called from the `Test` class, the source will be drawn from `NameTrait`. In addition, `setKey()` from `IdTrait` will still be available, but under an alias, `setKeyDate()`:

```
trait IdTrait
{
  protected $id;
  public $key;
  public function setId($id)
  {
    $this->id = $id;
  }
  public function setKey()
  {
    $this->key = date('YmdHis')
    . sprintf('%04d', rand(0,9999));
  }
}
```

```
trait NameTrait
{
  protected $name;
  public $key;
  public function setName($name)
  {
    $this->name = $name;
  }
  public function setKey()
  {
    $this->key = unpack('H*',
random_bytes(18))[1];
  }
}

class Test
{
  use IdTrait, NameTrait {
    NameTrait::setKeyinsteadofIdTrait;
    IdTrait::setKey as setKeyDate;
  }
}
```

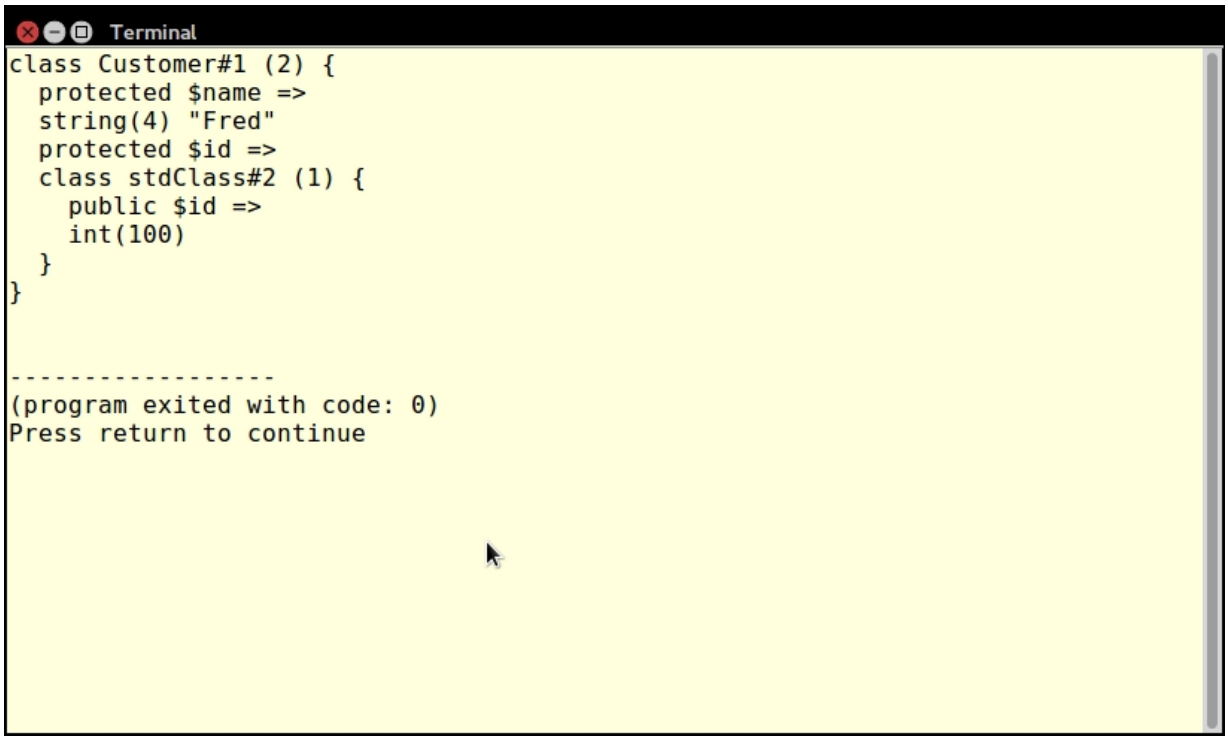
How it works...

From step 1, you learned that traits are used in situations where there is duplication of code. You need to gauge whether or not you could simply define a base class and extend it, or whether using a trait better serves your purposes. Traits are especially useful where the duplication of code is seen in logically unrelated classes.

To illustrate how trait methods override inherited methods, copy the block of code mentioned in step 7 into a separate file, `chap_04_oop_traits_override_inherited.php`. Add these lines of code:

```
$customer = new Customer();
$customer->setId(100);
$customer->setName('Fred');
var_dump($customer);
```

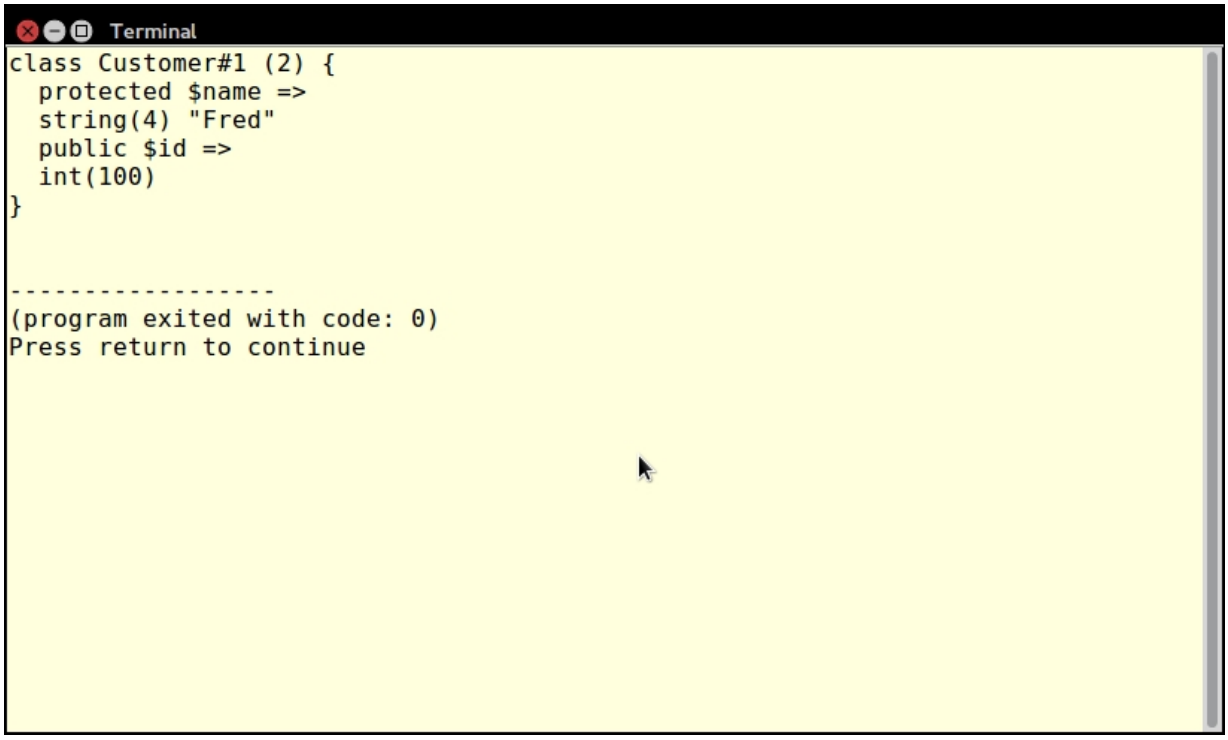
As you can see from the output (shown next), the property `$id` is stored as an instance of `stdClass()`, which is the behavior defined in the trait:



To illustrate how directly defined class methods override trait methods, copy the block of code mentioned in step 9 into a separate file, `chap_04_oop_trait_methods_do_not_override_class_methods.php`. Add these lines of code:

```
$customer = new Customer();
$customer->setId(100);
$customer->setName('Fred');
var_dump($customer);
```

As you can see from the following output, the `$id` property is stored as an integer, as defined in the `Customer` class, whereas the trait defines `$id` as an instance of `stdClass`:



In step 10, you learned how to resolve duplicate method name conflicts when using multiple traits. Copy the block of code shown in step 11 into a separate file, `chap_04_oop_trait_multiple.php`. Add the following code:

```
$a = new Test();
$a->setId(100);
$a->setName('Fred');
$a->setKey();
var_dump($a);

$a->setKeyDate();
var_dump($a);
```

Notice in the following output that `setKey()` yields the output produced from the new PHP 7 function, `random_bytes()` (defined in `NameTrait`), whereas `setKeyDate()` produces a key using the `date()` and `rand()` functions (defined in `IdTrait`):


Terminal
class Test#1 (3) {
protected $id =>
int(100)
public $key =>
string(36) "823b46fb10071c64baa373a4cdb8181c6d9c"
protected $name =>
string(4) "Fred"
}
class Test#1 (3) {
protected $id =>
int(100)
public $key =>
string(18) "201602180643172034"
protected $name =>
string(4) "Fred"
}

(program exited with code: 0)
Press return to continue

Implementing anonymous classes

PHP 7 introduced a new feature, **anonymous classes**. Much like anonymous functions, anonymous classes can be defined as part of an expression, creating a class that has no name. Anonymous classes are used in situations where you need to create an object *on the fly*, which is used and then discarded.

How to do it...

1. An alternative to `stdClass` is to define an anonymous class.

   In the definition, you can define any properties and methods (including magic methods). In this example, we define an anonymous class with two properties and a magic method, `__construct()`:

```
$a = new class (123.45, 'TEST') {
  public $total = 0;
  public $test  = '';
  public function __construct($total, $test)
  {
    $this->total = $total;
    $this->test  = $test;
  }
};
```

2. An anonymous class can extend any class.

   In this example, an anonymous class extends `FilterIterator`, and overrides both the `__construct()` and `accept()` methods. As an argument, it accepts `ArrayIterator $b`, which represents an array of 10 to 100 in increments of 10. The second argument serves as a limit on the output:

```
$b = new ArrayIterator(range(10,100,10));
$f = new class ($b, 50) extends FilterIterator
{
  public $limit = 0;
  public function __construct($iterator,
$limit)
  {
    $this->limit = $limit;
    parent::__construct($iterator);
  }
  public function accept()
  {
    return ($this->current() <= $this->limit);
  }
};
```

3. An anonymous class can implement an interface.

   In this example, an anonymous class is used to generate an HTML color code chart. The class implements the built-in PHP `Countable` interface. A `count()` method is defined, which is called when this class is used with a method or function that requires `Countable`:

```
define('MAX_COLORS', 256 ** 3);

$d = new class () implements Countable {
  public $current = 0;
  public $maxRows = 16;
  public $maxCols = 64;
  public function cycle()
  {
    $row = '';
    $max = $this->maxRows * $this->maxCols;
    for ($x = 0; $x < $this->maxRows; $x++) {
      $row .= '<tr>';
      for ($y = 0; $y < $this->maxCols; $y++)
{
        $row .= sprintf(
          '<td style="background-color:
#%06X;"',
          $this->current);
        $row .= sprintf(
          'title="#%06X"> </td>',
          $this->current);
```

```
            $this->current++;
            $this->current = ($this->current
>MAX_COLORS) ? 0
                : $this->current;
        }
        $row .= '</tr>';
      }
      return $row;
    }
    public function count()
    {
      return MAX_COLORS;
    }
};
```

4. Anonymous classes can use traits.
5. This last example is a modification from the preceding one defined immediately. Instead of defining a class `Test`, we define an anonymous class instead:

```
$a = new class() {
  use IdTrait, NameTrait {
    NameTrait::setKeyinsteadofIdTrait;
    IdTrait::setKey as setKeyDate;
  }
};
```

How it works...

In an anonymous class you can define any properties or methods. Using the preceding example, you could define an anonymous class that accepts constructor arguments, and where you can access properties. Place the code described in step 2 into a test script `chap_04_oop_anonymous_class.php`. Add these `echo` statements:

```
echo "\nAnonymous Class\n";
echo $a->total .PHP_EOL;
echo $a->test . PHP_EOL;
```

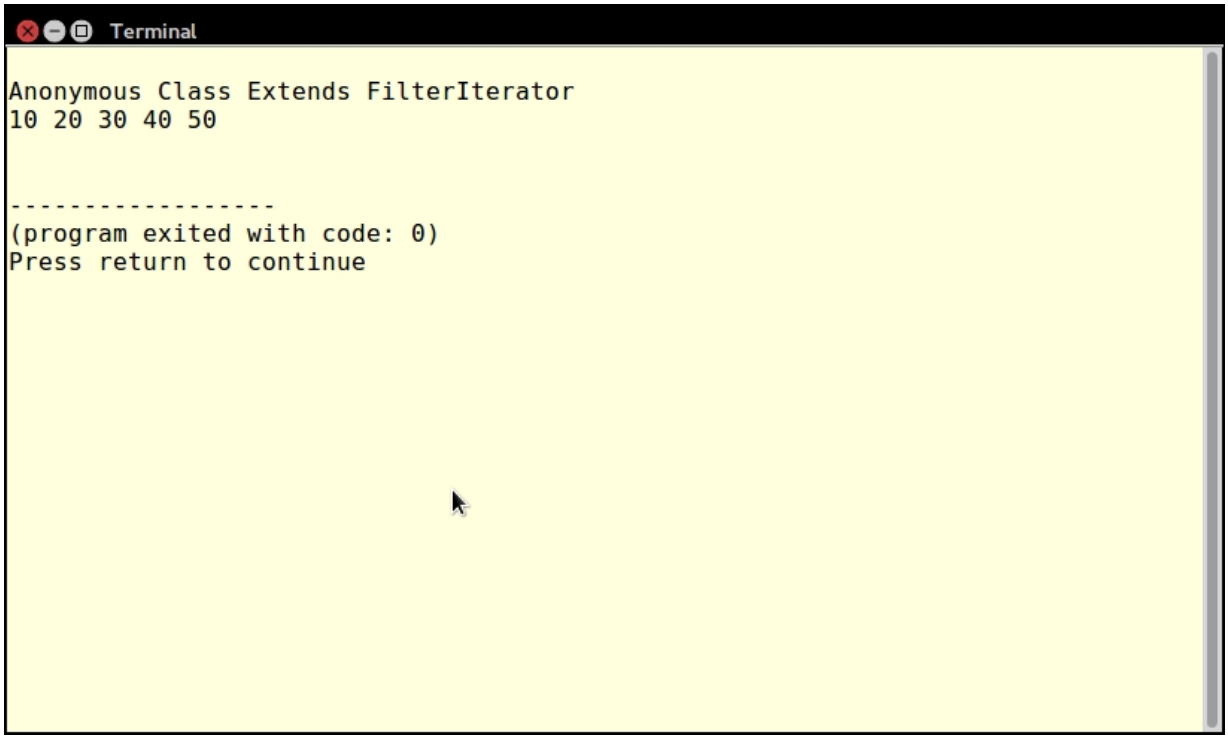
Here is the output from the anonymous class:



In order to use `FilterIterator` you *must* override the `accept()` method. In this method, you define criteria for which elements of the iteration are to be included as output. Go ahead now and add the code shown in step 4 to the test script. You can then add these `echo` statements to test the anonymous class:

```
echo "\nAnonymous Class Extends FilterIterator\n";
foreach ($f as $item) echo $item . '';
echo PHP_EOL;
```

In this example, a limit of 50 is established. The original `ArrayIterator` contains an array of values, 10 to 100, in increments of 10, as seen in the following output:



To have a look at an anonymous class that implements an interface, consider the example shown in steps 5 and 6. Place this code in a file, `chap_04_oop_anonymous_class_interfaces.php`.

Next, add code that lets you paginate through the HTML color chart:

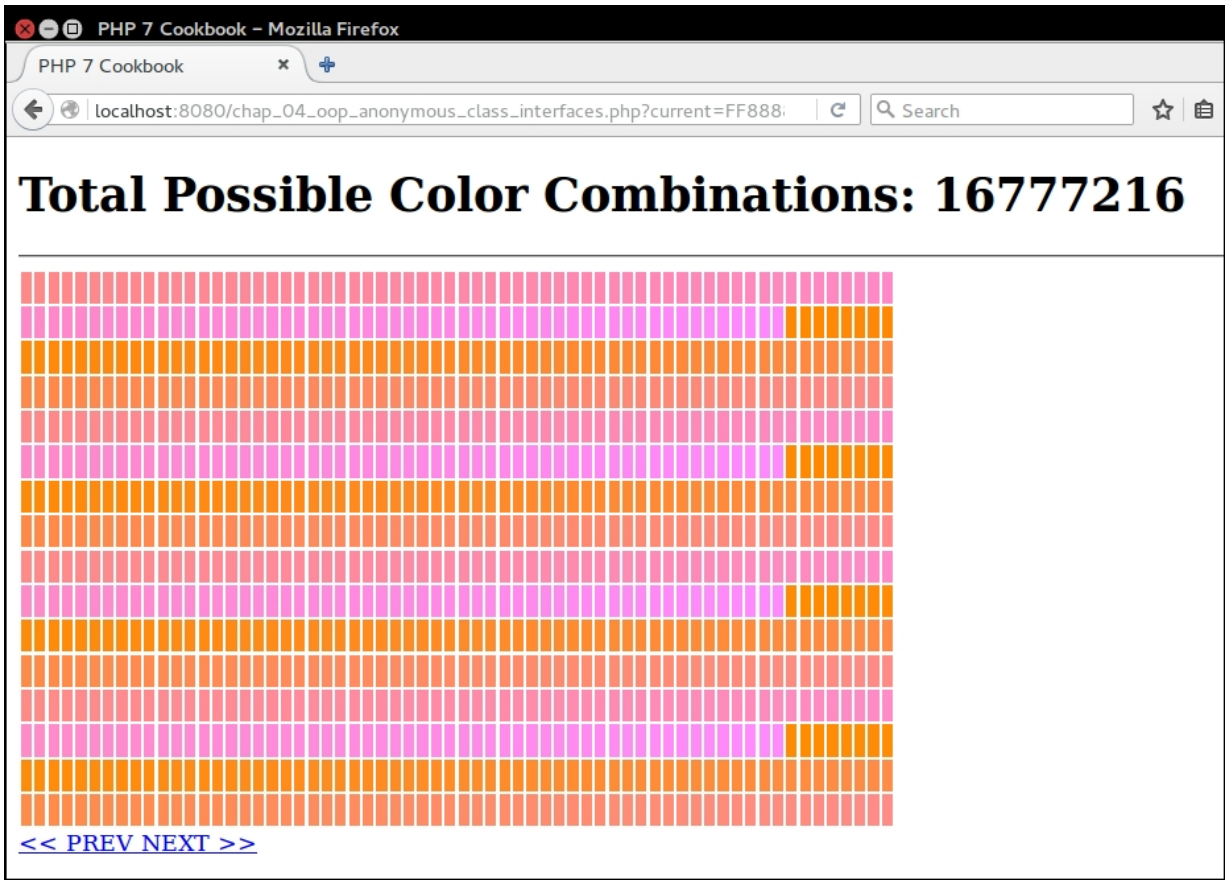
```
$d->current = $_GET['current'] ?? 0;
$d->current = hexdec($d->current);
$factor = ($d->maxRows * $d->maxCols);
$next = $d->current + $factor;
$prev = $d->current - $factor;
$next = ($next <MAX_COLORS) ? $next : MAX_COLORS -
$factor;
$prev = ($prev>= 0) ? $prev : 0;
$next = sprintf('%06X', $next);
$prev = sprintf('%06X', $prev);
?>
```

Finally, go ahead and present the HTML color chart as a web page:

```
<h1>Total Possible Color Combinations: <?=
count($d); ?></h1>
<hr>
```

```
<table>
<?= $d->cycle(); ?>
</table>
<a href="?current=<?= $prev ?>"><<PREV</a>
<a href="?current=<?= $next ?>">NEXT >></a>
```

Notice that you can take advantage of the `Countable` interface by passing the instance of the anonymous class into the `count()` function (shown between `<H1>` tags). Here is the output shown in a browser window:



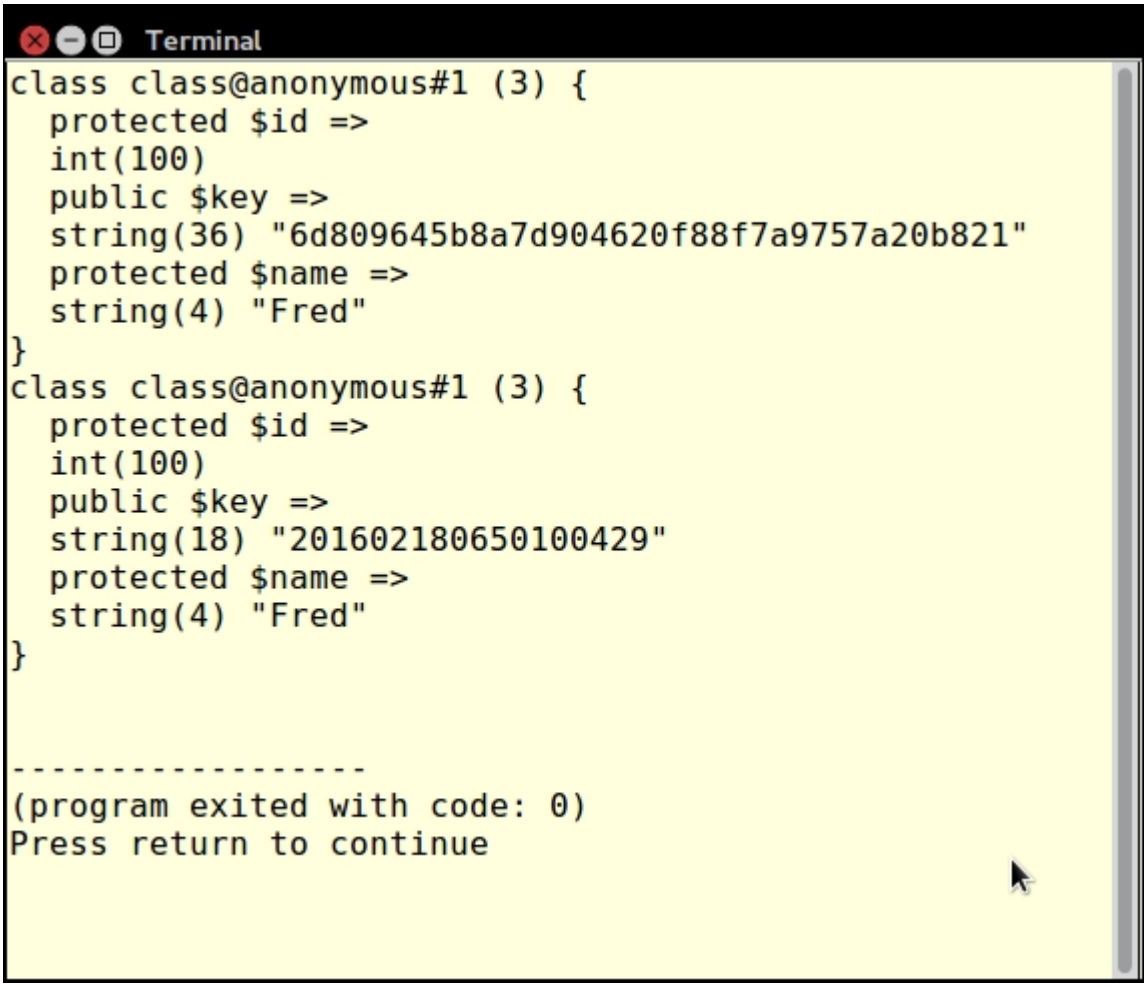
Lastly, to illustrate the use of traits in anonymous classes, copy the `chap_04_oop_trait_multiple.php` file mentioned in the previous recipe to a new file, `chap_04_oop_trait_anonymous_class.php`. Remove the definition of the `Test` class, and replace it with an anonymous class:

```
$a = new class() {
  use IdTrait, NameTrait {
    NameTrait::setKeyinsteadofIdTrait;
    IdTrait::setKey as setKeyDate;
  }
};
```

Remove this line:

```
$a = new Test();
```

When you run the code, you will see exactly the same output as shown in the preceding screenshot, except that the class reference will be anonymous:



Chapter 5. Interacting with a Database

In this chapter, we will cover the following topics:

- Using PDO to connect to a database
- Building an OOP SQL query builder
- Handling pagination
- Defining entities to match database tables
- Tying entity classes to RDBMS queries
- Embedding secondary lookups into query results
- Implementing jQuery DataTables PHP lookups

Introduction

In this chapter, we will cover a series of database connectivity recipes that take advantage of the **PHP Data Objects** (**PDO**) extension. Common programming problems such as **Structured Query Language** (**SQL**) generation, pagination, and tying objects to database tables, will be addressed. Finally, at the end, we will present code that processes secondary lookups in the form of embedded anonymous functions, and using jQuery DataTables to make AJAX requests.

Using PDO to connect to a database

PDO is a highly performant and actively maintained database extension that has a unique advantage over vendor-specific extensions. It has a common **Application Programming Interface** (**API**) that is compatible with almost a dozen different **Relational Database Management Systems** (**RDBMS**). Learning how to use this extension will save you hours of time trying to master the command subsets of the equivalent individual vendor-specific database extensions.

PDO is subdivided into four main classes, as summarized in the following table:

Class	**Functionality**
`PDO`	Maintains the actual connection to the database, and also handles low-level functionality such as transaction support
`PDOStatement`	Processes results
`PDOException`	Database-specific exceptions
`PDODriver`	Communicates with the actual vendor-specific database

How to do it...

1. Set up the database connection by creating a `PDO` instance.
2. You need to construct a **Data Source Name** (**DSN**). The information contained in the DSN varies according to the database driver used. As an example, here is a DSN used to connect to a **MySQL** database:

```
$params = [
  'host' => 'localhost',
  'user' => 'test',
  'pwd'  => 'password',
  'db'   => 'php7cookbook'
];

try {
  $dsn  = sprintf('mysql:host=%s;dbname=%s',
  $params['host'], $params['db']);
  $pdo  = new PDO($dsn, $params['user'],
$params['pwd']);
} catch (PDOException $e) {
  echo $e->getMessage();
} catch (Throwable $e) {
  echo $e->getMessage();
}
```

3. On the other hand, **SQlite**, a simpler extension, only requires the following command:

```
$params = [
  'db'   => __DIR__ .
'/../data/db/php7cookbook.db.sqlite'
];
$dsn  = sprintf('sqlite:' . $params['db']);
```

4. **PostgreSQL**, on the other hand, includes the username and password directly in the DSN:

```
$params = [
  'host' => 'localhost',
  'user' => 'test',
  'pwd'  => 'password',
  'db'   => 'php7cookbook'
];
```

```
$dsn  =
sprintf('pgsql:host=%s;dbname=%s;user=%s;passw
ord=%s',
                $params['host'],
                $params['db'],
                $params['user'],
                $params['pwd']);
```

5. The DSN could also include server-specific directives, such as `unix_socket`, as shown in the following example:

```
$params = [
  'host' => 'localhost',
  'user' => 'test',
  'pwd'  => 'password',
  'db'   => 'php7cookbook',
  'sock' => '/var/run/mysqld/mysqld.sock'
];

try {
  $dsn  =
sprintf('mysql:host=%s;dbname=%s;unix_socket=%
s',
                  $params['host'],
$params['db'], $params['sock']);
  $opts = [PDO::ATTR_ERRMODE =>
PDO::ERRMODE_EXCEPTION];
  $pdo  = new PDO($dsn, $params['user'],
$params['pwd'], $opts);
} catch (PDOException $e) {
  echo $e->getMessage();
} catch (Throwable $e) {
  echo $e->getMessage();
}
```

Note

Best practice

Wrap the statement that creates the PDO instance in a `try {} catch {}` block. Catch a `PDOException` for database-specific information in case of failure. Catch `Throwable` for errors or any other exceptions. Set the PDO error mode to

`PDO::ERRMODE_EXCEPTION` for best results. See step 8 for more details about error modes.

In PHP 5, if the PDO object cannot be constructed (for example, when invalid parameters are used), the instance is assigned a value of `NULL`. In PHP 7, an `Exception` is thrown. If you wrap the construction of the PDO object in a `try {} catch {}` block, and the `PDO::ATTR_ERRMODE` is set to `PDO::ERRMODE_EXCEPTION`, you can catch and log such errors without having to test for `NULL`.

6. Send an SQL command using `PDO::query()`. A `PDOStatement` instance is returned, against which you can fetch results. In this example, we are looking for the first 20 customers sorted by ID:

```
$stmt = $pdo->query(
'SELECT * FROM customer ORDER BY id LIMIT
20');
```

Note

PDO also provides a convenience method, `PDO::exec()`, which does not return a result iteration, just the number of rows affected. This method is best used for administrative operations such as `ALTER TABLE`, `DROP TABLE`, and so on.

7. Iterate through the `PDOStatement` instance to process results. Set the **fetch mode** to either `PDO::FETCH_NUM` or `PDO::FETCH_ASSOC` to return results in the form of a numeric or associative array. In this example we use a `while()` loop to process results. When the last result has been fetched, the result is a boolean `FALSE`, ending the loop:

```
while ($row = $stmt->fetch(PDO::FETCH_ASSOC))
{
  printf('%4d | %20s | %5s' . PHP_EOL,
```

```
$row['id'],
  $row['name'], $row['level']);
}
```

Note

PDO fetch operations involve a **cursor** that defines the direction (that is, forward or reverse) of the iteration. The second argument to `PDOStatement::fetch()` can be any of the `PDO::FETCH_ORI_*` constants. Cursor orientations include prior, first, last, absolute, and relative. The default cursor orientation is `PDO::FETCH_ORI_NEXT`.

8. Set the fetch mode to `PDO::FETCH_OBJ` to return results as a `stdClass` instance. Here you will note that the `while()` loop takes advantage of the fetch mode, `PDO::FETCH_OBJ`. Notice that the `printf()` statement refers to object properties, in contrast with the preceding example, which references array elements:

```
while ($row = $stmt->fetch(PDO::FETCH_OBJ)) {
  printf('%4d | %20s | %5s' . PHP_EOL,
  $row->id, $row->name, $row->level);
}
```

9. If you want to create an instance of a specific class while processing a query, set the fetch mode to `PDO::FETCH_CLASS`. You must also have the class definition available, and `PDO::query()` should set the class name. As you can see in the following code snippet, we have defined a class called `Customer`, with public properties `$id`, `$name`, and `$level`. Properties need to be `public` for the fetch injection to work properly:

```
class Customer
{
  public $id;
  public $name;
  public $level;
}
```

```
$stmt = $pdo->query($sql, PDO::FETCH_CLASS,
'Customer');
```

10. When fetching objects, a simpler alternative to the technique shown in step 5 is to use `PDOStatement::fetchObject()`:

```
while ($row = $stmt->fetchObject('Customer'))
{
  printf('%4d | %20s | %5s' . PHP_EOL,
  $row->id, $row->name, $row->level);
}
```

11. You could also use `PDO::FETCH_INTO`, which is essentially the same as `PDO::FETCH_CLASS`, but you need an active object instance instead of a class reference. Each iteration through the loop re-populates the same object instance with the current information set. This example assumes the same class `Customer` as in step 5, with the same database parameters and PDO connections as defined in step 1:

```
$cust = new Customer();
while ($stmt->fetch(PDO::FETCH_INTO)) {
  printf('%4d | %20s | %5s' . PHP_EOL,
  $cust->id, $cust->name, $cust->level);
}
```

12. If you do not specify an error mode, the default PDO error mode is `PDO::ERRMODE_SILENT`. You can set the error mode using the `PDO::ATTR_ERRMODE` key, and either the `PDO::ERRMODE_WARNING` or the `PDO::ERRMODE_EXCEPTION` value. The error mode can be specified as the fourth argument to the PDO constructor in the form of an associative array. Alternatively, you can use `PDO::setAttribute()` on an existing instance.
13. Let us assume you have the following DSN and SQL (before you start thinking that this is a new form of SQL, please be assured: this SQL statement will not work!):

```
$params = [
  'host' => 'localhost',
  'user' => 'test',
  'pwd'  => 'password',
  'db'   => 'php7cookbook'
```

```
];
$dsn  = sprintf('mysql:host=%s;dbname=%s',
$params['host'], $params['db']);
$sql  = 'THIS SQL STATEMENT WILL NOT WORK';
```

14. If you then formulate your PDO connection using the default error mode, the only clue that something is wrong is that instead of producing a `PDOStatement` instance, the `PDO::query()` will return a boolean `FALSE`:

```
$pdo1  = new PDO($dsn, $params['user'],
$params['pwd']);
$stmt = $pdo1->query($sql);
$row = ($stmt) ? $stmt-
>fetch(PDO::FETCH_ASSOC) : 'No Good';
```

15. The next example shows setting the error mode to `WARNING` using the constructor approach:

```
$pdo2 = new PDO(
  $dsn,
  $params['user'],
  $params['pwd'],
  [PDO::ATTR_ERRMODE =>
PDO::ERRMODE_WARNING]);
```

16. If you need full separation of the prepare and execute phases, use `PDO::prepare()` and `PDOStatement::execute()` instead. The statement is then sent to the database server to be pre-compiled. You can then execute the statement as many times as is warranted, most likely in a loop.
17. The first argument to `PDO::prepare()` can be an SQL statement with placeholders in place of actual values. An array of values can then be supplied to `PDOStatement::execute()`. PDO automatically provides database quoting, which helps safeguard against **SQL Injection**.

Note

Best practice

Any application in which external input (that is, from a form posting) is combined with an SQL statement is subject to an SQL injection attack. All external input must first be properly filtered, validated, and otherwise sanitized. Do not put external input directly into the SQL statement. Instead, use placeholders, and provide the actual (sanitized) values during the execution phase.

18. To iterate through the results in reverse, you can change the orientation of the **scrollable cursor**. Alternatively, and probably more easily, just reverse the `ORDER BY` from `ASC` to `DESC`. This line of code sets up a `PDOStatement` object requesting a scrollable cursor:

```
$dsn  =
sprintf('pgsql:charset=UTF8;host=%s;dbname=%s'
, $params['host'], $params['db']);
$opts = [PDO::ATTR_ERRMODE =>
PDO::ERRMODE_EXCEPTION];
$pdo  = new PDO($dsn, $params['user'],
$params['pwd'], $opts);
$sql  = 'SELECT * FROM customer '
    . 'WHERE balance > :min AND balance < :max
'
    . 'ORDER BY id LIMIT 20';
$stmt = $pdo->prepare($sql, [PDO::ATTR_CURSOR
=> PDO::CURSOR_SCROLL]);
```

19. You also need to specify cursor instructions during the fetch operation. This example gets the last row in the result set, and then scrolls backwards:

```
$stmt->execute(['min' => $min, 'max' =>
$max]);
$row = $stmt->fetch(PDO::FETCH_ASSOC,
PDO::FETCH_ORI_LAST);
do {
  printf('%4d | %20s | %5s | %8.2f' . PHP_EOL,
       $row['id'],
       $row['name'],
       $row['level'],
       $row['balance']);
```

```
} while ($row = $stmt->fetch(PDO::FETCH_ASSOC,
PDO::FETCH_ORI_PRIOR));
```

20. Neither MySQL nor SQLite support scrollable cursors! To achieve the same results, try the following modifications to the preceding code:

```
$dsn  =
sprintf('mysql:charset=UTF8;host=%s;dbname=%s'
, $params['host'], $params['db']);
$opts = [PDO::ATTR_ERRMODE =>
PDO::ERRMODE_EXCEPTION];
$pdo  = new PDO($dsn, $params['user'],
$params['pwd'], $opts);
$sql  = 'SELECT * FROM customer '
    . 'WHERE balance > :min AND balance < :max
'
    . 'ORDER BY id DESC
       . 'LIMIT 20';
$stmt = $pdo->prepare($sql);
while ($row = $stmt->fetch(PDO::FETCH_ASSOC));
printf('%4d | %20s | %5s | %8.2f' . PHP_EOL,
       $row['id'],
       $row['name'],
       $row['level'],
       $row['balance']);
}
```

21. PDO provides support for transactions. Borrowing the code from step 9, we can wrap the `INSERT` series of commands into a transactional block:

```
try {
    $pdo->beginTransaction();
    $sql  = "INSERT INTO customer ('"
    . implode("','", $fields) . "') VALUES
(?,?,?,?,?,?)";
    $stmt = $pdo->prepare($sql);
    foreach ($data as $row) $stmt-
>execute($row);
    $pdo->commit();
} catch (PDOException $e) {
    error_log($e->getMessage());
    $pdo->rollBack();
}
```

22. Finally, to keep everything modular and re-usable, we can wrap the PDO connection into a separate class `Application\Database\Connection`. Here, we build a connection through the constructor. Alternatively, there is a static `factory()` method that lets us generate a series of PDO instances:

```
namespace Application\Database;
use Exception;
use PDO;
class Connection
{
    const ERROR_UNABLE = 'ERROR: no database
connection';
    public $pdo;
    public function __construct(array $config)
    {
        if (!isset($config['driver'])) {
            $message = __METHOD__ . ' : '
            . self::ERROR_UNABLE . PHP_EOL;
            throw new Exception($message);
        }
        $dsn = $this->makeDsn($config);
        try {
            $this->pdo = new PDO(
                $dsn,
                $config['user'],
                $config['password'],
                [PDO::ATTR_ERRMODE =>
$config['errmode']]);
            return TRUE;
        } catch (PDOException $e) {
            error_log($e->getMessage());
            return FALSE;
        }
    }

    public static function factory(
      $driver, $dbname, $host, $user,
      $pwd, array $options = array())
    {
        $dsn = $this->makeDsn($config);

        try {
            return new PDO($dsn, $user, $pwd,
$options);
        } catch (PDOException $e) {
```

```
                error_log($e->getMessage);
            }
        }
```

23. An important component of this `Connection` class is a generic method that can be used to construct a DSN. All we need for this to work is to establish the `PDODriver` as a prefix, followed by "`:`". After that, we simply append key/value pairs from our configuration array. Each key/value pair is separated by a semi-colon. We also need to strip off the trailing semi-colon, using `substr()` with a negative limit for that purpose:

```
    public function makeDsn($config)
    {
      $dsn = $config['driver'] . ':';
      unset($config['driver']);
      foreach ($config as $key => $value) {
        $dsn .= $key . '=' . $value . ';';
      }
      return substr($dsn, 0, -1);
    }
  }
```

How it works...

First of all, you can copy the initial connection code from step 1 into a `chap_05_pdo_connect_mysql.php` file. For the purposes of this illustration, we will assume you have created a MySQL database called `php7cookbook`, with a username of cook and a password of book. Next, we send a simple SQL statement to the database using the `PDO::query()` method. Finally, we use the resulting statement object to fetch results in the form of an associative array. Don't forget to wrap your code in a `try {} catch {}` block:

```
<?php
$params = [
  'host' => 'localhost',
  'user' => 'test',
  'pwd'  => 'password',
  'db'   => 'php7cookbook'
```

```
];
try {
  $dsn  =
sprintf('mysql:charset=UTF8;host=%s;dbname=%s',
    $params['host'], $params['db']);
  $pdo  = new PDO($dsn, $params['user'],
$params['pwd']);
  $stmt = $pdo->query(
    'SELECT * FROM customer ORDER BY id LIMIT
20');
  printf('%4s | %20s | %5s | %7s' . PHP_EOL,
    'ID', 'NAME', 'LEVEL', 'BALANCE');
  printf('%4s | %20s | %5s | %7s' . PHP_EOL,
    '----', str_repeat('-', 20), '-----', '-------
');
  while ($row = $stmt->fetch(PDO::FETCH_ASSOC)) {
    printf('%4d | %20s | %5s | %7.2f' . PHP_EOL,
    $row['id'], $row['name'], $row['level'],
$row['balance']);
  }
} catch (PDOException $e) {
  error_log($e->getMessage());
} catch (Throwable $e) {
  error_log($e->getMessage());
}
```

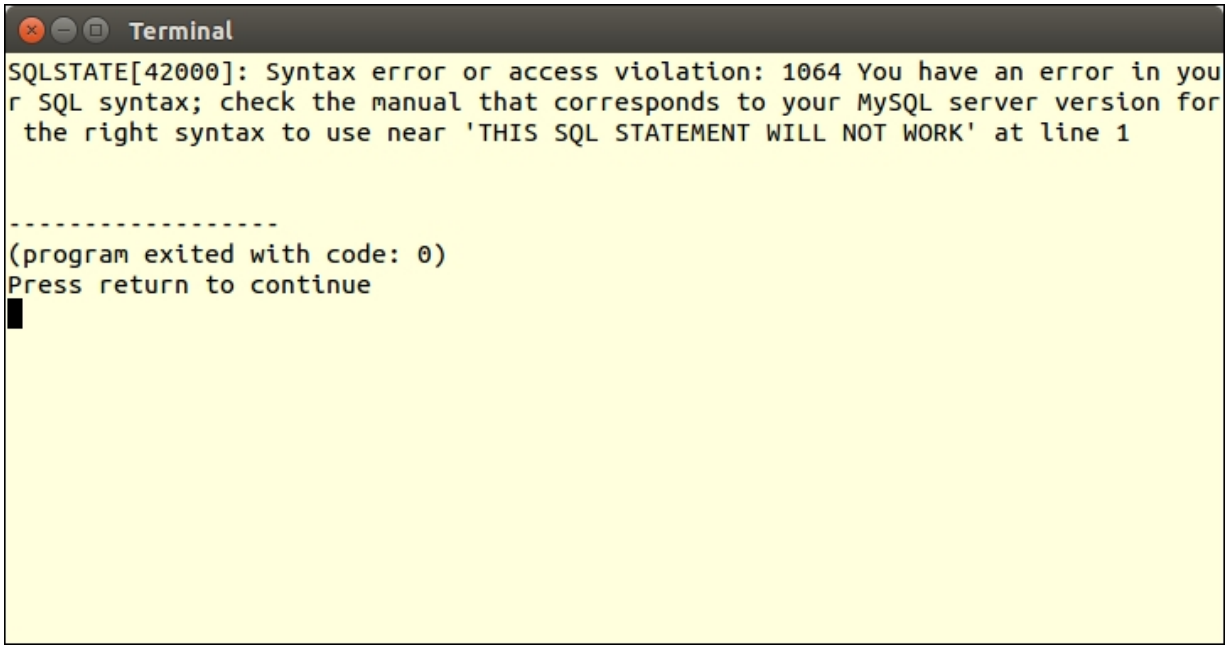
Here is the resulting output:



Add the option to the PDO constructor, which sets the error mode to `EXCEPTION`. Now alter the SQL statement and observe the resulting error message:

```
$opts = [PDO::ATTR_ERRMODE =>
PDO::ERRMODE_EXCEPTION];
$pdo  = new PDO($dsn, $params['user'],
$params['pwd'], $opts);
$stmt = $pdo->query('THIS SQL STATEMENT WILL NOT
WORK');
```

You will observe something like this:



Placeholders can be named or positional. **Named placeholders** are preceded by a colon (`:`) in the prepared SQL statement, and are references as keys in an associative array provided to `execute()`. **Positional placeholders** are represented as question marks (`?`) in the prepared SQL statement.

In the following example, named placeholders are used to represent values in a `WHERE` clause:

```
try {
  $dsn  = sprintf('mysql:host=%s;dbname=%s',
                  $params['host'], $params['db']);
  $pdo  = new PDO($dsn,
                  $params['user'],
                  $params['pwd'],
                  [PDO::ATTR_ERRMODE =>
PDO::ERRMODE_EXCEPTION]);
  $sql  = 'SELECT * FROM customer '
      . 'WHERE balance < :val AND level = :level '
      . 'ORDER BY id LIMIT 20'; echo $sql .
PHP_EOL;
  $stmt = $pdo->prepare($sql);
  $stmt->execute(['val' => 100, 'level' =>
'BEG']);
  while ($row = $stmt->fetch(PDO::FETCH_ASSOC)) {
    printf('%4d | %20s | %5s | %5.2f' . PHP_EOL,
```

```
        $row['id'], $row['name'], $row['level'],
$row['balance']);
  }
} catch (PDOException $e) {
  echo $e->getMessage();
} catch (Throwable $e) {
  echo $e->getMessage();
}
```

This example shows using positional placeholders in an `INSERT` operation. Notice that the data to be inserted as the fourth customer includes a potential SQL injection attack. You will also notice that some awareness of the SQL syntax for the database being used is required. In this case, MySQL column names are quoted using back-ticks (`):

```
$fields = ['name', 'balance', 'email',
           'password', 'status', 'level'];
$data = [
  ['Saleen',0,'saleen@test.com',
'password',0,'BEG'],
  ['Lada',55.55,'lada@test.com',
'password',0,'INT'],

['Tonsoi',999.99,'tongsoi@test.com','password',1,'
ADV'],
  ['SQL Injection',0.00,'bad','bad',1,
   'BEG\';DELETE FROM customer;--'],
];

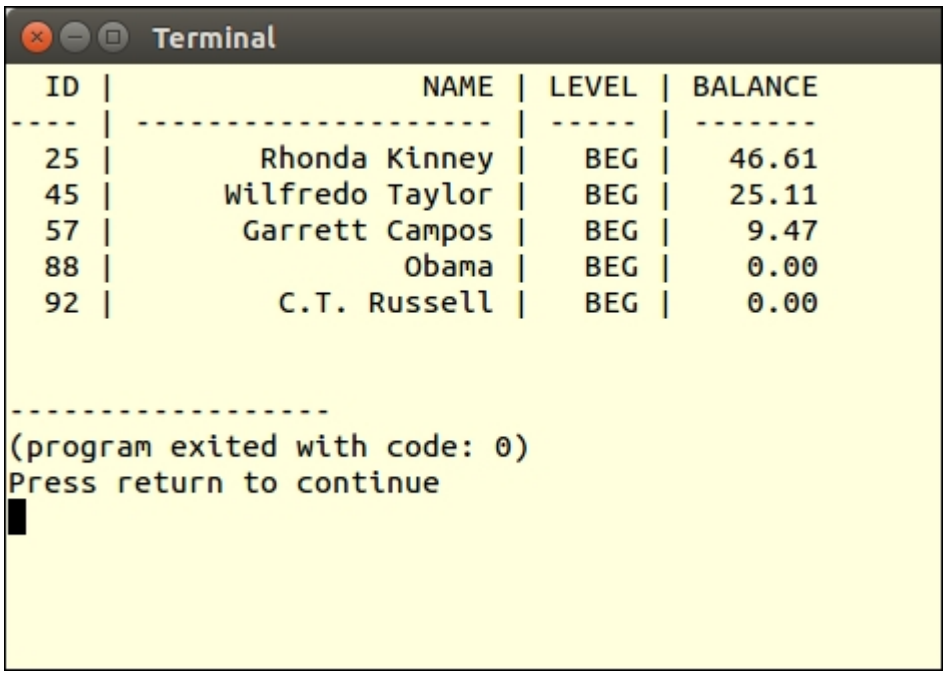
try {
  $dsn  = sprintf('mysql:host=%s;dbname=%s',
    $params['host'], $params['db']);
  $pdo  = new PDO($dsn,
                  $params['user'],
                  $params['pwd'],
                  [PDO::ATTR_ERRMODE =>
PDO::ERRMODE_EXCEPTION]);
  $sql  = "INSERT INTO customer ('"
   . implode("','", $fields)
   . "') VALUES (?,?,?,?,?,?)";
  $stmt = $pdo->prepare($sql);
  foreach ($data as $row) $stmt->execute($row);
} catch (PDOException $e) {
  echo $e->getMessage();
```

```
} catch (Throwable $e) {
  echo $e->getMessage();
}
```

To test the use of a prepared statement with named parameters, modify the SQL statement to add a `WHERE` clause that checks for customers with a balance less than a certain amount, and a level equal to either `BEG`, `INT`, or `ADV` (that is, beginning, intermediate, or advanced). Instead of using `PDO::query()`, use `PDO::prepare()`. Before fetching results, you must then perform `PDOStatement::execute()`, supplying the values for balance and level:

```
$sql  = 'SELECT * FROM customer '
      . 'WHERE balance < :val AND level = :level '
      . 'ORDER BY id LIMIT 20';
$stmt = $pdo->prepare($sql);
$stmt->execute(['val' => 100, 'level' => 'BEG']);
```

Here is the resulting output:



Instead of providing parameters when calling `PDOStatement::execute()`, you could alternatively bind parameters.

This allows you to assign variables to placeholders. At the time of execution, the current value of the variable is used.

In this example, we bind the variables `$min`, `$max`, and `$level` to the prepared statement:

```
$min   = 0;
$max   = 0;
$level = '';

try {
  $dsn  = sprintf('mysql:host=%s;dbname=%s',
$params['host'], $params['db']);
  $opts = [PDO::ATTR_ERRMODE =>
PDO::ERRMODE_EXCEPTION];
  $pdo  = new PDO($dsn, $params['user'],
$params['pwd'], $opts);
  $sql  = 'SELECT * FROM customer '
      . 'WHERE balance > :min '
      . 'AND balance < :max AND level = :level '
      . 'ORDER BY id LIMIT 20';
  $stmt = $pdo->prepare($sql);
  $stmt->bindParam('min',   $min);
  $stmt->bindParam('max',   $max);
  $stmt->bindParam('level', $level);

  $min   =  5000;
  $max   = 10000;
  $level = 'ADV';
  $stmt->execute();
  showResults($stmt, $min, $max, $level);

  $min   = 0;
  $max   = 100;
  $level = 'BEG';
  $stmt->execute();
  showResults($stmt, $min, $max, $level);

} catch (PDOException $e) {
  echo $e->getMessage();
} catch (Throwable $e) {
  echo $e->getMessage();
}
```

When the values of these variables change, the next execution will reflect the modified criteria.

> ## Tip
>
> **Best practice**
>
> Use `PDO::query()` for one-time database commands. Use `PDO::prepare()` and `PDOStatement::execute()` when you need to process the same statement multiple times but using different values.

See also

For information on the syntax and unique behavior associated with different vendor-specific PDO drivers, have a look this article:

- http://php.net/manual/en/pdo.drivers.php

For a summary of PDO predefined constants, including fetch modes, cursor orientation, and attributes, see the following article:

- http://php.net/manual/en/pdo.constants.php

Building an OOP SQL query builder

PHP 7 implements something called a **context sensitive lexer**. What this means is that words that are normally reserved can be used if the context allows. Thus, when building an object-oriented SQL builder, we can get away with using methods named `and`, `or`, `not`, and so on.

How to do it...

1. We define a `Application\Database\Finder` class. In the class, we define methods that match our favorite SQL operations:

```
namespace Application\Database;
class Finder
{
  public static $sql      = '';
  public static $instance = NULL;
  public static $prefix   = '';
  public static $where    = array();
  public static $control  = ['', ''];

    // $a == name of table
    // $cols = column names
    public static function select($a, $cols =
NULL)
    {
      self::$instance  = new Finder();
      if ($cols) {
           self::$prefix = 'SELECT ' . $cols .
' FROM ' . $a;
      } else {
        self::$prefix = 'SELECT * FROM ' . $a;
      }
      return self::$instance;
    }

    public static function where($a = NULL)
    {
```

```
        self::$where[0] = ' WHERE ' . $a;
        return self::$instance;
    }

    public static function like($a, $b)
    {
        self::$where[] = trim($a . ' LIKE ' .
$b);
        return self::$instance;
    }

    public static function and($a = NULL)
    {
        self::$where[] = trim('AND ' . $a);
        return self::$instance;
    }

    public static function or($a = NULL)
    {
        self::$where[] = trim('OR ' . $a);
        return self::$instance;
    }

    public static function in(array $a)
    {
        self::$where[] = 'IN ( ' .
implode(',', $a) . ' )';
        return self::$instance;
    }

    public static function not($a = NULL)
    {
        self::$where[] = trim('NOT ' . $a);
        return self::$instance;
    }

    public static function limit($limit)
    {
        self::$control[0] = 'LIMIT ' . $limit;
        return self::$instance;
    }

    public static function offset($offset)
    {
        self::$control[1] = 'OFFSET ' .
$offset;
        return self::$instance;
    }
```

```
    public static function getSql()
    {
      self::$sql = self::$prefix
         . implode(' ', self::$where)
                . ' '
                . self::$control[0]
                . ' '
                . self::$control[1];
      preg_replace('/  /', ' ', self::$sql);
      return trim(self::$sql);
    }
  }
```

2. Each function used to generate an SQL fragment returns the same property, `$instance`. This allows us to represent the code using a fluent interface, such as this:

```
$sql = Finder::select('project')-
>where('priority > 9') ... etc.
```

How it works...

Copy the code defined precedingly into a `Finder.php` file in the `Application\Database` folder. You can then create a `chap_05_oop_query_builder.php` calling program, which initializes the autoloader defined in Chapter 1, *Building a Foundation*. You can then run `Finder::select()` to generate an object from which the SQL string can be rendered:

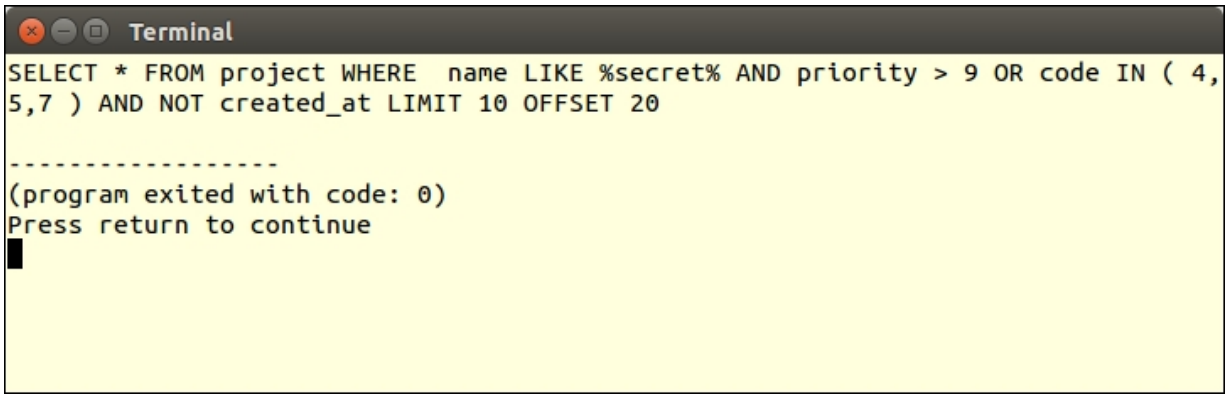
```
<?php
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
use Application\Database\Finder;

$sql = Finder::select('project')
  ->where()
  ->like('name', '%secret%')
  ->and('priority > 9')
  ->or('code')->in(['4', '5', '7'])
  ->and()->not('created_at')
```

```
        ->limit(10)
        ->offset(20);

    echo Finder::getSql();
```

Here is the result of the precding code:



See also

For more information on the context-sensitive lexer, have a look at this article:

https://wiki.php.net/rfc/context_sensitive_lexer

Handling pagination

Pagination involves providing a limited subset of the results of a database query. This is usually done for display purposes, but could easily apply to other situations. At first glance, it would seem the `LimitIterator` class is ideally suited for the purposes of pagination. In cases where the potential result set could be massive; however, `LimitIterator` is not such an ideal candidate, as you would need to supply the entire result set as an inner iterator, which would most likely exceed memory limitations. The second and third arguments to the `LimitIterator` class constructor are offset and count. This suggests the pagination solution we will adopt, which is native to SQL: adding `LIMIT` and `OFFSET` clauses to a given SQL statement.

How to do it...

1. First, we create a class called `Application\Database\Paginate` to hold the pagination logic. We add properties to represent values associated with pagination, `$sql`, `$page`, and `$linesPerPage`:

```
namespace Application\Database;

class Paginate
{

  const DEFAULT_LIMIT  = 20;
  const DEFAULT_OFFSET = 0;

  protected $sql;
  protected $page;
  protected $linesPerPage;

}
```

2. Next, we define a `__construct()` method that accepts a base SQL statement, the current page number, and the number of lines per page as arguments. We then need to refactor the SQL string modifying or adding the `LIMIT` and `OFFSET` clauses.

3. In the constructor, we need to calculate the offset using the current page number and the number of lines per page. We also need to check to see if `LIMIT` and `OFFSET` are already present in the SQL statement. Finally, we need to revise the statement using lines per page as our `LIMIT` with the recalculated `OFFSET`:

```
public function __construct($sql, $page,
$linesPerPage)
{
  $offset = $page * $linesPerPage;
  switch (TRUE) {
    case (stripos($sql, 'LIMIT') &&
strpos($sql, 'OFFSET')) :
      // no action needed
      break;
    case (stripos($sql, 'LIMIT')) :
      $sql .= ' LIMIT ' . self::DEFAULT_LIMIT;
      break;
    case (stripos($sql, 'OFFSET')) :
      $sql .= ' OFFSET ' .
self::DEFAULT_OFFSET;
      break;
    default :
      $sql .= ' LIMIT ' . self::DEFAULT_LIMIT;
      $sql .= ' OFFSET ' .
self::DEFAULT_OFFSET;
      break;
  }
  $this->sql = preg_replace('/LIMIT
\d+.*OFFSET \d+/Ui',
     'LIMIT ' . $linesPerPage . ' OFFSET ' .
$offset,
     $sql);
}
```

4. We are now ready to execute the query using the `Application\Database\Connection` class discussed in the first recipe.
5. In our new pagination class, we add a `paginate()` method, which takes a `Connection` instance as an argument. We also need the PDO fetch mode, and optional prepared statement parameters:

```
use PDOException;
public function paginate(
```

```
    Connection $connection,
    $fetchMode,
    $params = array())
    {
    try {
      $stmt = $connection->pdo->prepare($this-
>sql);
      if (!$stmt) return FALSE;
      if ($params) {
        $stmt->execute($params);
      } else {
        $stmt->execute();
      }
      while ($result = $stmt->fetch($fetchMode))
yield $result;
    } catch (PDOException $e) {
      error_log($e->getMessage());
      return FALSE;
    } catch (Throwable $e) {
      error_log($e->getMessage());
      return FALSE;
    }
}
```

6. It might not be a bad idea to provide support for the query builder class mentioned in the previous recipe. This will make updating `LIMIT` and `OFFSET` much easier. All we need to do to provide support for `Application\Database\Finder` is to use the class and modify the `__construct()` method to check to see if the incoming SQL is an instance of this class:

```
    if ($sql instanceof Finder) {
      $sql->limit($linesPerPage);
      $sql->offset($offset);
      $this->sql = $sql::getSql();
    } elseif (is_string($sql)) {
      switch (TRUE) {
        case (stripos($sql, 'LIMIT')
        && strpos($sql, 'OFFSET')) :
            // remaining code as shown in bullet
#3 above
        }
    }
```

7. Now all that remains to be done is to add a `getSql()` method in case we need to confirm that the SQL statement was correctly

formed:

```
public function getSql()
{
  return $this->sql;
}
```

How it works...

Copy the preceding code into a `Paginate.php` file in the `Application/Database` folder. You can then create a `chap_05_pagination.php` calling program, which initializes the autoloader defined in Chapter 1, *Building a Foundation*:

```
<?php
define('DB_CONFIG_FILE',
'/../config/db.config.php');
define('LINES_PER_PAGE', 10);
define('DEFAULT_BALANCE', 1000);
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
```

Next, use the `Application\Database\Finder`, `Connection`, and `Paginate` classes, create an instance of `Application\Database\Connection`, and use `Finder` to generate SQL:

```
use Application\Database\ { Finder, Connection,
Paginate};
$conn = new Connection(include __DIR__ .
DB_CONFIG_FILE);
$sql = Finder::select('customer')->where('balance
< :bal');
```

We can now get the page number and balance from `$_GET` parameters, and create the `Paginate` object, ending the PHP block:

```
$page = (int) ($_GET['page'] ?? 0);
$bal  = (float) ($_GET['balance'] ??
```

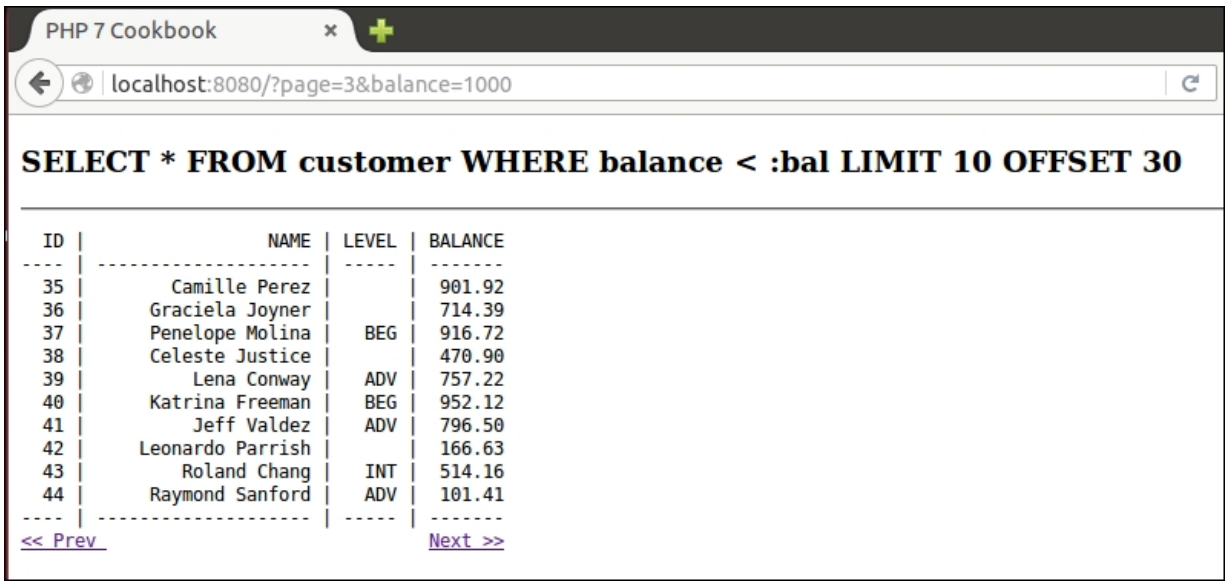
```
DEFAULT_BALANCE);
$paginate = new Paginate($sql::getSql(), $page,
LINES_PER_PAGE);
?>
```

In the output portion of the script, we simply iterate through the pagination using a simple `foreach()` loop:

```
<h3><?= $paginate->getSql(); ?></h3>
<hr>
<pre>
<?php
printf('%4s | %20s | %5s | %7s' . PHP_EOL,
  'ID', 'NAME', 'LEVEL', 'BALANCE');
printf('%4s | %20s | %5s | %7s' . PHP_EOL,
  '----', str_repeat('-', 20), '-----', '-------
');
foreach ($paginate->paginate($conn,
PDO::FETCH_ASSOC,
  ['bal' => $bal]) as $row) {
  printf('%4d | %20s | %5s | %7.2f' . PHP_EOL,

$row['id'],$row['name'],$row['level'],$row['balanc
e']);
}
printf('%4s | %20s | %5s | %7s' . PHP_EOL,
  '----', str_repeat('-', 20), '-----', '-------
');
?>
<a href="?page=<?= $page - 1; ?>&balance=<?= $bal
?>">
<< Prev </a>  
<a href="?page=<?= $page + 1; ?>&balance=<?= $bal
?>">
Next >></a>
</pre>
```

Here is page 3 of the output, where the balance is less than 1,000:



See also

For more information on the `LimitIterator` class, refer to this article:

- http://php.net/manual/en/class.limititerator.php

Defining entities to match database tables

A very common practice among PHP developers is to create classes that represent database tables. Such classes are often referred to as **entity** classes, and form the core of the **domain model** software design pattern.

How to do it...

1. First of all, we will establish some common features of a series of entity classes. These might include common properties and common methods. We will put these into a `Application\Entity\Base` class. All future entity classes will then extend `Base`.
2. For the purposes of this illustration, let's assume all entities will have two properties in common: `$mapping` (discussed later), and `$id` (with its corresponding getter and setter):

```
namespace Application\Entity;

class Base
{

  protected $id = 0;
  protected $mapping = ['id' => 'id'];

  public function getId() : int
  {
    return $this->id;
  }

  public function setId($id)
  {
    $this->id = (int) $id;
  }
}
```

3. It's not a bad idea to define a `arrayToEntity()` method, which converts an array to an instance of the entity class, and vice versa (`entityToArray()`). These methods implement a process often referred to as **hydration**. As these methods should be generic, they are best placed in the `Base` class.
4. In the following methods, the `$mapping` property is used to translate between database column names and object property names. `arrayToEntity()` populates values of this object instance from an array. We can define this method as static in case we need to call it outside of an active instance:

```
public static function arrayToEntity($data,
Base $instance)
{
  if ($data && is_array($data)) {
    foreach ($instance->mapping as $dbColumn
=> $propertyName) {
      $method = 'set' .
ucfirst($propertyName);
      $instance->$method($data[$dbColumn]);
    }
    return $instance;
  }
  return FALSE;
}
```

5. The `entityToArray()` produces an array from current instance property values:

```
public function entityToArray()
{
  $data = array();
  foreach ($this->mapping as $dbColumn =>
$propertyName) {
    $method = 'get' . ucfirst($propertyName);
    $data[$dbColumn] = $this->$method() ??
NULL;
  }
  return $data;
}
```

6. To build the specific entity, you need to have the structure of the database table you plan to model at hand. Create properties

that map to the database columns. The initial values assigned should reflect the ultimate data-type of the database column.

7. In this example we'll use the `customer` table. Here is the `CREATE` statement from a MySQL data dump, which illustrates its data structure:

```
CREATE TABLE 'customer' (
  'id' int(11) NOT NULL AUTO_INCREMENT,
  'name' varchar(256) CHARACTER SET latin1
COLLATE latin1_general_cs NOT NULL,
  'balance' decimal(10,2) NOT NULL,
  'email' varchar(250) NOT NULL,
  'password' char(16) NOT NULL,
  'status' int(10) unsigned NOT NULL DEFAULT
'0',
  'security_question' varchar(250) DEFAULT
NULL,
  'confirm_code' varchar(32) DEFAULT NULL,
  'profile_id' int(11) DEFAULT NULL,
  'level' char(3) NOT NULL,
  PRIMARY KEY ('id'),
  UNIQUE KEY 'UNIQ_81398E09E7927C74' ('email')
);
```

8. We are now in a position to flesh out the class properties. This is also a good place to identify the corresponding table. In this case, we will use a `TABLE_NAME` class constant:

```
namespace Application\Entity;

class Customer extends Base
{
  const TABLE_NAME = 'customer';
  protected $name = '';
  protected $balance = 0.0;
  protected $email = '';
  protected $password = '';
  protected $status = '';
  protected $securityQuestion = '';
  protected $confirmCode = '';
  protected $profileId = 0;
  protected $level = '';
}
```

9. It is considered a best practice to define the properties as `protected`. In order to access these properties, you will need to design `public` methods that `get` and `set` the properties. Here is a good place to put to use the PHP 7 ability to data-type to the return value.
10. In the following block of code, we have defined getters and setters for `$name` and `$balance`. You can imagine how the remainder of these methods will be defined:

```
    public function getName() : string
    {
      return $this->name;
    }
    public function setName($name)
    {
      $this->name = $name;
    }
    public function getBalance() : float
    {
      return $this->balance;
    }
    public function setBalance($balance)
    {
      $this->balance = (float) $balance;
    }
}
```

> **Tip**
>
> It is not a good idea to data type check the incoming values on the setters. The reason is that the return values from a RDBMS database query will all be a `string` data type.

11. If the property names do not exactly match the corresponding database column, you should consider creating a `mapping` property, an array of key/value pairs where the key represents the database column name and the value the property name.

12. You will note that three properties, `$securityQuestion`, `$confirmCode`, and `$profileId`, do not correspond to their equivalent column names, `security_question`, `confirm_code`, and `profile_id`. The `$mapping` property will ensure that the appropriate translation takes place:

```
protected $mapping = [
  'id'                => 'id',
  'name'              => 'name',
  'balance'           => 'balance',
  'email'             => 'email',
  'password'          => 'password',
  'status'            => 'status',
  'security_question' => 'securityQuestion',
  'confirm_code'      => 'confirmCode',
  'profile_id'        => 'profileId',
  'level'             => 'level'
];
```

How it works...

Copy the code from steps 2, 4, and 5 into a `Base.php` file in the `Application/Entity` folder. Copy the code from steps 8 through 12 into a `Customer.php` file, also in the `Application/Entity` folder. You will then need to create getters and setters for the remaining properties not shown in step 10: `email`, `password`, `status`, `securityQuestion`, `confirmCode`, `profileId`, and `level`.

You can then create a `chap_05_matching_entity_to_table.php` calling program, which initializes the autoloader defined in Chapter 1, *Building a Foundation*, uses the `Application\Database\Connection`, and the newly created `Application\Entity\Customer` classes:

```
<?php
define('DB_CONFIG_FILE',
'/../config/db.config.php');
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
```

```
use Application\Database\Connection;
use Application\Entity\Customer;
```

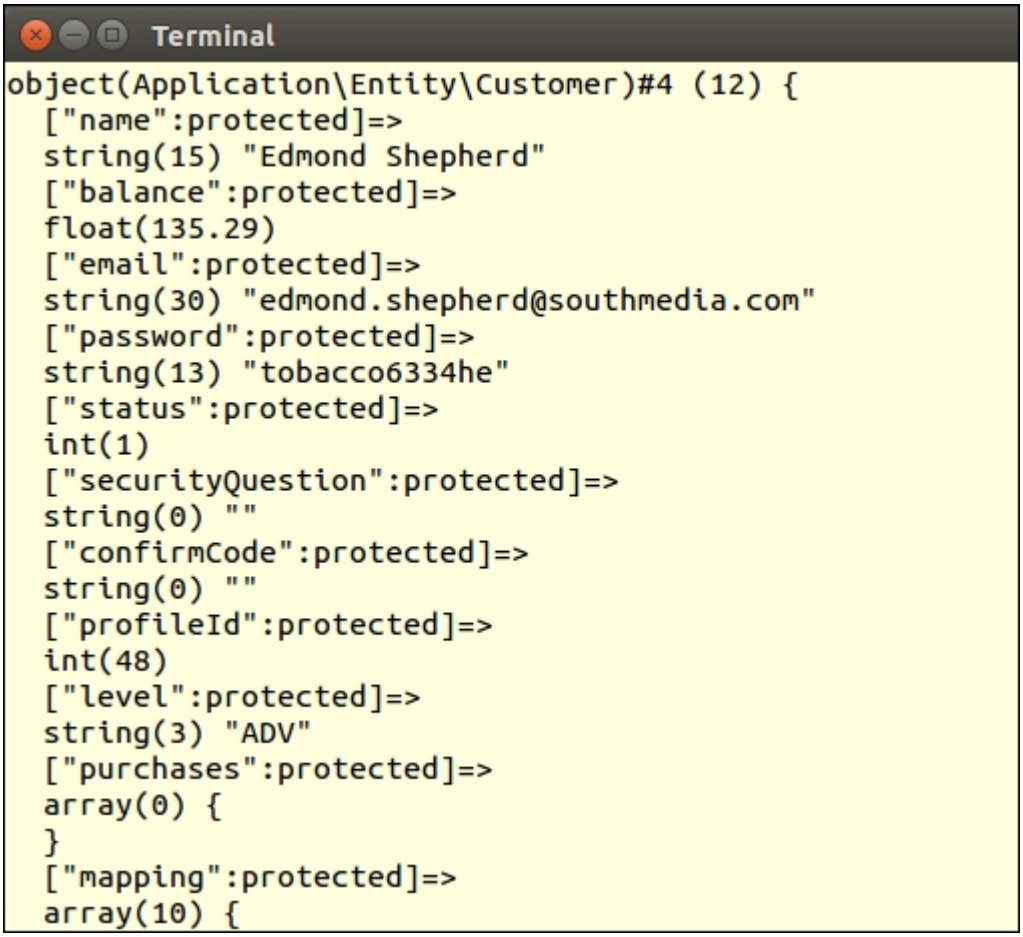
Next, get a database connection, and use the connection to acquire an associative array of data for one customer at random:

```
$conn = new Connection(include __DIR__ .
DB_CONFIG_FILE);
$id = rand(1,79);
$stmt = $conn->pdo->prepare(
  'SELECT * FROM customer WHERE id = :id');
$stmt->execute(['id' => $id]);
$result = $stmt->fetch(PDO::FETCH_ASSOC);
```

Finally, you can create a new `Customer` entity instance from the array and use `var_dump()` to view the result:

```
$cust = Customer::arrayToEntity($result, new
Customer());
var_dump($cust);
```

Here is the output of the preceding code:



See also

There are many good works that describe the domain model. Probably the most influential is *Patterns of Enterprise Application Architecture* by Martin Fowler (see http://martinfowler.com/books/eaa.html). There is also a nice study, also available as a free download, entitled *Domain Driven Design Quickly* by InfoQ (see http://www.infoq.com/minibooks/domain-driven-design-quickly).

Tying entity classes to RDBMS queries

Most commercially viable RDBMS systems evolved at a time when procedural programming was at the fore. Imagine the RDBMS world as two dimensional, square, and procedurally oriented. In contrast, entities could be thought of as round, three dimensional, and object oriented. This gives you a picture of what we want to accomplish by tying the results of an RDBMS query into an iteration of entity instances.

> ## Note
>
> The **relational model**, upon which modern RDBMS systems are based, was first described by the mathematician Edgar F. Codd in 1969. The first commercially viable systems evolved in the mid-to-late 1970s. So, in other words, RDBMS technology is over 40 years old!

How to do it...

1. First of all, we need to design a class which will house our query logic. If you are following the Domain Model, this class might be called a **repository**. Alternatively, to keep things simple and generic, we could simply call the new class `Application\Database\CustomerService`. The class will accept an `Application\Database\Connection` instance as an argument:

```
namespace Application\Database;

use Application\Entity\Customer;

class CustomerService
{
```

```
    protected $connection;

    public function __construct(Connection
$connection)
    {
      $this->connection = $connection;
    }

}
```

2. Now we will define a `fetchById()` method, which takes a customer ID as an argument, and returns a single `Application\Entity\Customer` instance or boolean `FALSE` on failure. At first glance, it would seem a no-brainer to simply use `PDOStatement::fetchObject()` and specify the entity class as an argument:

```
public function fetchById($id)
{
  $stmt = $this->connection->pdo
                -
>prepare(Finder::select('customer')
               ->where('id = :id')::getSql());
  $stmt->execute(['id' => (int) $id]);
  return $stmt-
>fetchObject('Application\Entity\Customer');
}
```

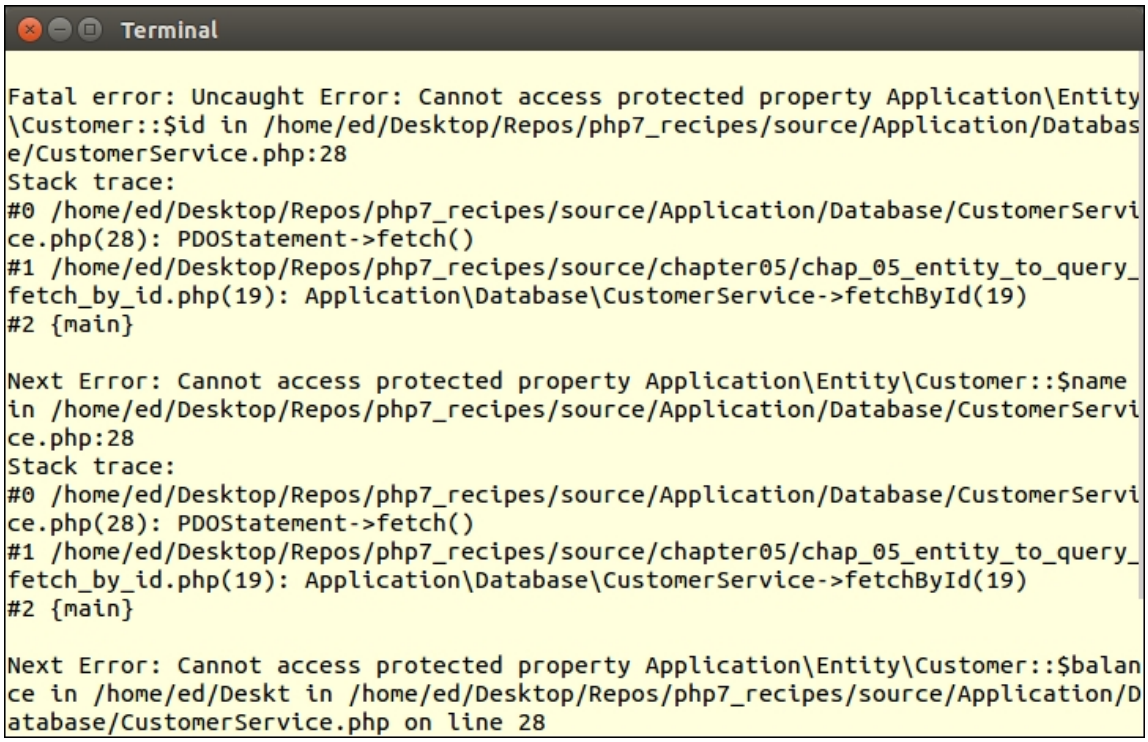
Note

The danger here, however, is that `fetchObject()` actually populates the properties (even if they are protected) before the constructor is called! Accordingly, there is a danger that the constructor could accidentally overwrite values. If you don't define a constructor, or if you can live with this danger, we're done. Otherwise, it starts to get tougher to properly implement the tie between RDBMS query and OOP results.

3. Another approach for the `fetchById()` method is to create the object instance first, thereby running its constructor, and setting the fetch mode to `PDO::FETCH_INTO`, as shown in the following example:

```
public function fetchById($id)
{
  $stmt = $this->connection->pdo
                  -
>prepare(Finder::select('customer')
                ->where('id = :id')::getSql());
  $stmt->execute(['id' => (int) $id]);
  $stmt->setFetchMode(PDO::FETCH_INTO, new
Customer());
  return $stmt->fetch();
}
```

4. Here again, however, we encounter a problem: `fetch()`, unlike `fetchObject()`, is not able to overwrite protected properties; the following error message is generated if it tries. This means we will either have to define all properties as `public`, or consider another approach.



5. The last approach we will consider will be to fetch the results in the form of an array, and manually *hydrate* the entity. Even though this approach is slightly more costly in terms of performance, it allows any potential entity constructor to run properly, and keeps properties safely defined as `private` or `protected`:

```
public function fetchById($id)
{
  $stmt = $this->connection->pdo
                -
>prepare(Finder::select('customer')
               ->where('id = :id')::getSql());
  $stmt->execute(['id' => (int) $id]);
  return Customer::arrayToEntity(
    $stmt->fetch(PDO::FETCH_ASSOC));
}
```

6. To process a query that produces multiple results, all we need to do is to produce an iteration of populated entity objects. In this example, we implement a `fetchByLevel()` method that returns all customers for a given level, in the form of `Application\Entity\Customer` instances:

```
public function fetchByLevel($level)
{
  $stmt = $this->connection->pdo->prepare(
            Finder::select('customer')
            ->where('level =
:level')::getSql());
  $stmt->execute(['level' => $level]);
  while ($row = $stmt-
>fetch(PDO::FETCH_ASSOC)) {
    yield Customer::arrayToEntity($row, new
Customer());
  }
}
```

7. The next method we wish to implement is `save()`. Before we can proceed, however, some thought must be given to what value will be returned if an `INSERT` takes place.
8. Normally, we would return the newly completed entity class after an `INSERT`. There is a convenient `PDO::lastInsertId()` method

which, at first glance, would seem to do the trick. Further reading of the documentation reveals, however, that not all database extensions support this feature, and the ones that do are not consistent in their implementation. Accordingly, it would be a good idea to have a unique column other than `$id` that can be used to uniquely identify the new customer.

9. In this example we have chosen the `email` column, and thus need to implement a `fetchByEmail()` service method:

```
public function fetchByEmail($email)
{
  $stmt = $this->connection->pdo->prepare(
    Finder::select('customer')
    ->where('email = :email')::getSql());
  $stmt->execute(['email' => $email]);
  return Customer::arrayToEntity(
    $stmt->fetch(PDO::FETCH_ASSOC), new
Customer());
}
```

10. Now we are ready to define the `save()` method. Rather than distinguish between `INSERT` and `UPDATE`, we will architect this method to update if the ID already exists, and otherwise do an insert.
11. First, we define a basic `save()` method, which accepts a `Customer` entity as an argument, and uses `fetchById()` to determine if this entry already exists. If it exists, we call an `doUpdate()` update method; otherwise, we call a `doInsert()` insert method:

```
public function save(Customer $cust)
{
  // check to see if customer ID > 0 and
exists
  if ($cust->getId() && $this-
>fetchById($cust->getId())) {
    return $this->doUpdate($cust);
  } else {
    return $this->doInsert($cust);
  }
}
```

12. Next, we define `doUpdate()`, which pulls `Customer` entity object properties into an array, builds an initial SQL statement, and calls a `flush()` method, which pushes data to the database. We do not want the ID field updated, as it's the primary key. Also we need to specify which row to update, which means appending a `WHERE` clause:

```
protected function doUpdate($cust)
{
  // get properties in the form of an array
  $values = $cust->entityToArray();
  // build the SQL statement
  $update = 'UPDATE ' . $cust::TABLE_NAME;
  $where = ' WHERE id = ' . $cust->getId();
  // unset ID as we want do not want this to
be updated
  unset($values['id']);
  return $this->flush($update, $values,
$where);
}
```

13. The `doInsert()` method is similar, except that the initial SQL needs to start with `INSERT INTO ...` and the `id` array element needs to be unset. The reason for the latter is that we want this property to be auto-generated by the database. If this is successful, we use our newly defined `fetchByEmail()` method to look up the new customer and return a completed instance:

```
protected function doInsert($cust)
{
  $values = $cust->entityToArray();
  $email  = $cust->getEmail();
  unset($values['id']);
  $insert = 'INSERT INTO ' . $cust::TABLE_NAME
. ' ';
  if ($this->flush($insert, $values)) {
    return $this->fetchByEmail($email);
  } else {
    return FALSE;
  }
}
```

14. Finally, we are in a position to define `flush()`, which does the actual preparation and execution:

```
protected function flush($sql, $values, $where
= '')
{
  $sql .=  ' SET ';
  foreach ($values as $column => $value) {
    $sql .= $column . ' = :' . $column . ',';
  }
  // get rid of trailing ','
  $sql     = substr($sql, 0, -1) . $where;
  $success = FALSE;
  try {
    $stmt = $this->connection->pdo-
>prepare($sql);
    $stmt->execute($values);
    $success = TRUE;
  } catch (PDOException $e) {
    error_log(__METHOD__ . ':' . __LINE__ .
':'
    . $e->getMessage());
    $success = FALSE;
  } catch (Throwable $e) {
    error_log(__METHOD__ . ':' . __LINE__ .
':'
    . $e->getMessage());
    $success = FALSE;
  }
  return $success;
}
```

15. To round off the discussion, we need to define a `remove()` method, which deletes a customer from the database. Again, as with the `save()` method defined previously, we use `fetchById()` to ensure the operation was successful:

```
public function remove(Customer $cust)
{
  $sql = 'DELETE FROM ' . $cust::TABLE_NAME .
' WHERE id = :id';
  $stmt = $this->connection->pdo-
>prepare($sql);
  $stmt->execute(['id' => $cust->getId()]);
  return ($this->fetchById($cust->getId())) ?
FALSE : TRUE;
}
```

How it works...

Copy the code described in steps 1 to 5 into a `CustomerService.php` file in the `Application/Database` folder. Define a `chap_05_entity_to_query.php` calling program. Have the calling program initialize the autoloader, using the appropriate classes:

```
<?php
define('DB_CONFIG_FILE',
'/../config/db.config.php');
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
use Application\Database\Connection;
use Application\Database\CustomerService;
```

You can now create an instance of the service, and fetch a single customer at random. The service will then return a customer entity as a result:

```
// get service instance
$service = new CustomerService(new
Connection(include __DIR__ . DB_CONFIG_FILE));

echo "\nSingle Result\n";
var_dump($service->fetchById(rand(1,79)));
```

Here is the output:



Now copy the code shown in steps 6 to 15 into the service class. Add the data to insert to the `chap_05_entity_to_query.php` calling program. We then generate a `Customer` entity instance using this data:

```
// sample data
$data = [
  'name'              => 'Doug Bierer',
  'balance'           => 326.33,
  'email'             => 'doug' . rand(0,999) .
'@test.com',
  'password'          => 'password',
  'status'            => 1,
  'security_question' => 'Who\'s on first?',
  'confirm_code'      => 12345,
  'level'             => 'ADV'
];

// create new Customer
```

```
$cust = Customer::arrayToEntity($data, new
Customer());
```

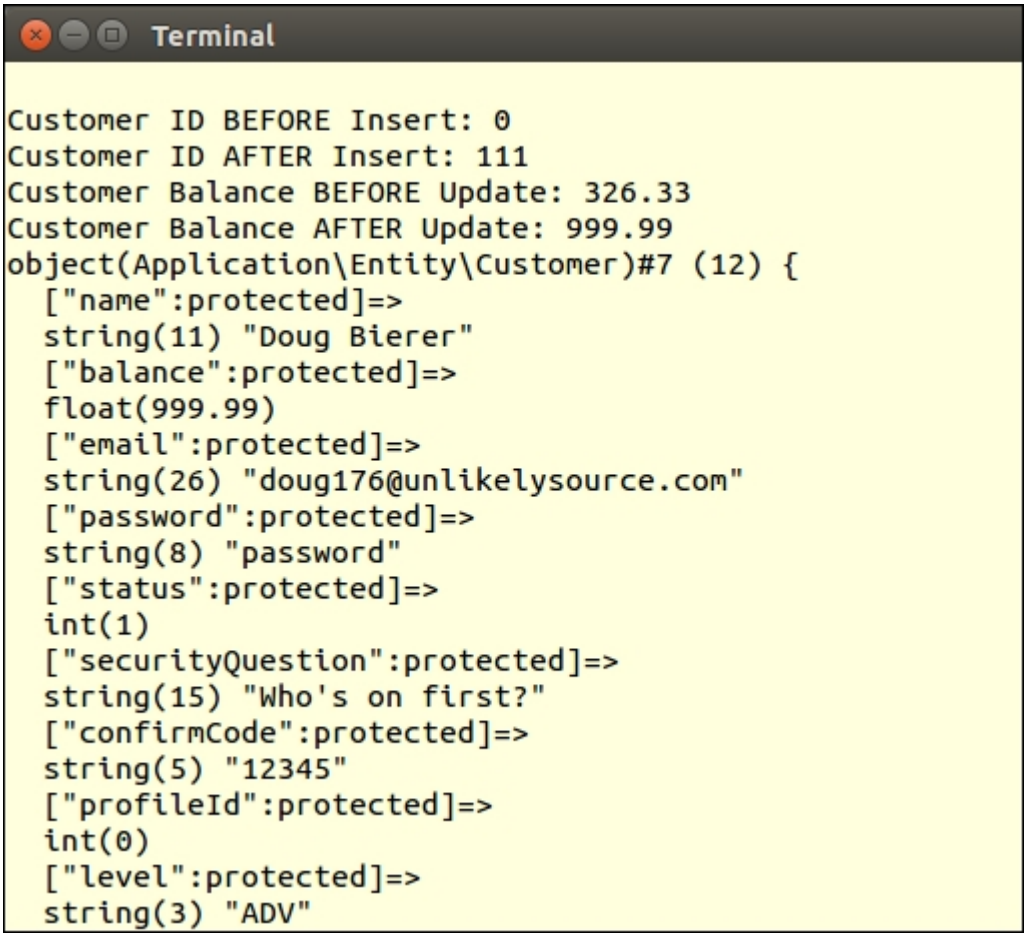
We can then examine the ID before and after the call to `save()`:

```
echo "\nCustomer ID BEFORE Insert: {$cust-
>getId()}\n";
$cust = $service->save($cust);
echo "Customer ID AFTER Insert: {$cust-
>getId()}\n";
```

Finally, we modify the balance, and again call `save()`, viewing the results:

```
echo "Customer Balance BEFORE Update: {$cust-
>getBalance()}\n";
$cust->setBalance(999.99);
$service->save($cust);
echo "Customer Balance AFTER Update: {$cust-
>getBalance()}\n";
var_dump($cust);
```

Here is the output from the calling program:



There's more...

For more information on the relational model, please refer to https://en.wikipedia.org/wiki/Relational_model. For more information on RDBMS, please refer to https://en.wikipedia.org/wiki/Relational_database_management_system. For information on how `PDOStatement::fetchObject()` inserts property values even before the constructor, have a look at the comment by "rasmus at mindplay dot dk" in the php.net documentation reference on `fetchObject()` (http://php.net/manual/en/pdostatement.fetchobject.php).

Embedding secondary lookups into query results

On the road towards implementing relationships between entity classes, let us first take a look at how we can embed the code needed to perform a secondary lookup. An example of such a lookup is when displaying information on a customer, have the view logic perform a second lookup that gets a list of purchases for that customer.

> ## Note
>
> The advantage of this approach is that processing is deferred until the actual view logic is executed. This will ultimately smooth the performance curve, with the workload distributed more evenly between the initial query for customer information, and the later query for purchase information. Another benefit is that a massive `JOIN` is avoided with its inherent redundant data.

How to do it...

1. First of all, define a function that finds a customer based on their ID. For the purposes of this illustration, we will simply fetch an array using the fetch mode `PDO::FETCH_ASSOC`. We will also continue to use the `Application\Database\Connection` class discussed in Chapter 1, *Building a Foundation*:

```
function findCustomerById($id, Connection
$conn)
{
  $stmt = $conn->pdo->query(
    'SELECT * FROM customer WHERE id = ' .
(int) $id);
  $results = $stmt->fetch(PDO::FETCH_ASSOC);
```

```
  return $results;
}
```

2. Next, we analyze the purchases table to see how the `customer` and `product` tables are linked. As you can see from the `CREATE` statement for this table, the `customer_id` and `product_id` foreign keys form the relationships:

```
CREATE TABLE 'purchases' (
  'id' int(11) NOT NULL AUTO_INCREMENT,
  'transaction' varchar(8) NOT NULL,
  'date' datetime NOT NULL,
  'quantity' int(10) unsigned NOT NULL,
  'sale_price' decimal(8,2) NOT NULL,
  'customer_id' int(11) DEFAULT NULL,
  'product_id' int(11) DEFAULT NULL,
  PRIMARY KEY ('id'),
  KEY 'IDX_C3F3' ('customer_id'),
  KEY 'IDX_665A' ('product_id'),
  CONSTRAINT 'FK_665A' FOREIGN KEY
('product_id')
  REFERENCES 'products' ('id'),
  CONSTRAINT 'FK_C3F3' FOREIGN KEY
('customer_id')
  REFERENCES 'customer' ('id')
);
```

3. We now expand the original `findCustomerById()` function, defining the secondary lookup in the form of an anonymous function, which can then be executed in a view script. The anonymous function is assigned to the `$results['purchases']` element:

```
function findCustomerById($id, Connection
$conn)
{
  $stmt = $conn->pdo->query(
       'SELECT * FROM customer WHERE id = ' .
(int) $id);
  $results = $stmt->fetch(PDO::FETCH_ASSOC);
  if ($results) {
    $results['purchases'] =
      // define secondary lookup
      function ($id, $conn) {
        $sql = 'SELECT * FROM purchases AS u '
```

```
            . 'JOIN products AS r '
            . 'ON u.product_id = r.id '
            . 'WHERE u.customer_id = :id '
            . 'ORDER BY u.date';
          $stmt = $conn->pdo->prepare($sql);
          $stmt->execute(['id' => $id]);
          while ($row = $stmt-
>fetch(PDO::FETCH_ASSOC)) {
            yield $row;
          }
      };
  }
  return $results;
}
```

4. Assuming we have successfully retrieved customer information into a `$results` array, in the view logic, all we need to do is to loop through the return value of the anonymous function. In this example, we retrieve customer information at random:

```
$result = findCustomerById(rand(1,79), $conn);
```

5. In the view logic, we loop through the results returned by the secondary lookup. The call to the embedded anonymous function is highlighted in the following code:

```
<table>
  <tr>
<th>Transaction</th><th>Date</th><th>Qty</th>
<th>Price</th><th>Product</th>
  </tr>
<?php
foreach ($result['purchases']($result['id'],
$conn) as $purchase) : ?>
  <tr>
    <td><?= $purchase['transaction'] ?></td>
    <td><?= $purchase['date'] ?></td>
    <td><?= $purchase['quantity'] ?></td>
    <td><?= $purchase['sale_price'] ?></td>
    <td><?= $purchase['title'] ?></td>
  </tr>
<?php endforeach; ?>
</table>
```

How it works...

Create a `chap_05_secondary_lookups.php` calling program and insert the code needed to create an instance of `Application\Database\Connection`:

```
<?php
define('DB_CONFIG_FILE',
'/../config/db.config.php');
include __DIR__ .
'/../Application/Database/Connection.php';
use Application\Database\Connection;
$conn = new Connection(include __DIR__ .
DB_CONFIG_FILE);
```

Next, add the `findCustomerById()` function shown in step 3. You can then pull information for a random customer, ending the PHP part of the calling program:

```
function findCustomerById($id, Connection $conn)
{
  // code shown in bullet #3 above
}
$result = findCustomerById(rand(1,79), $conn);
?>
```

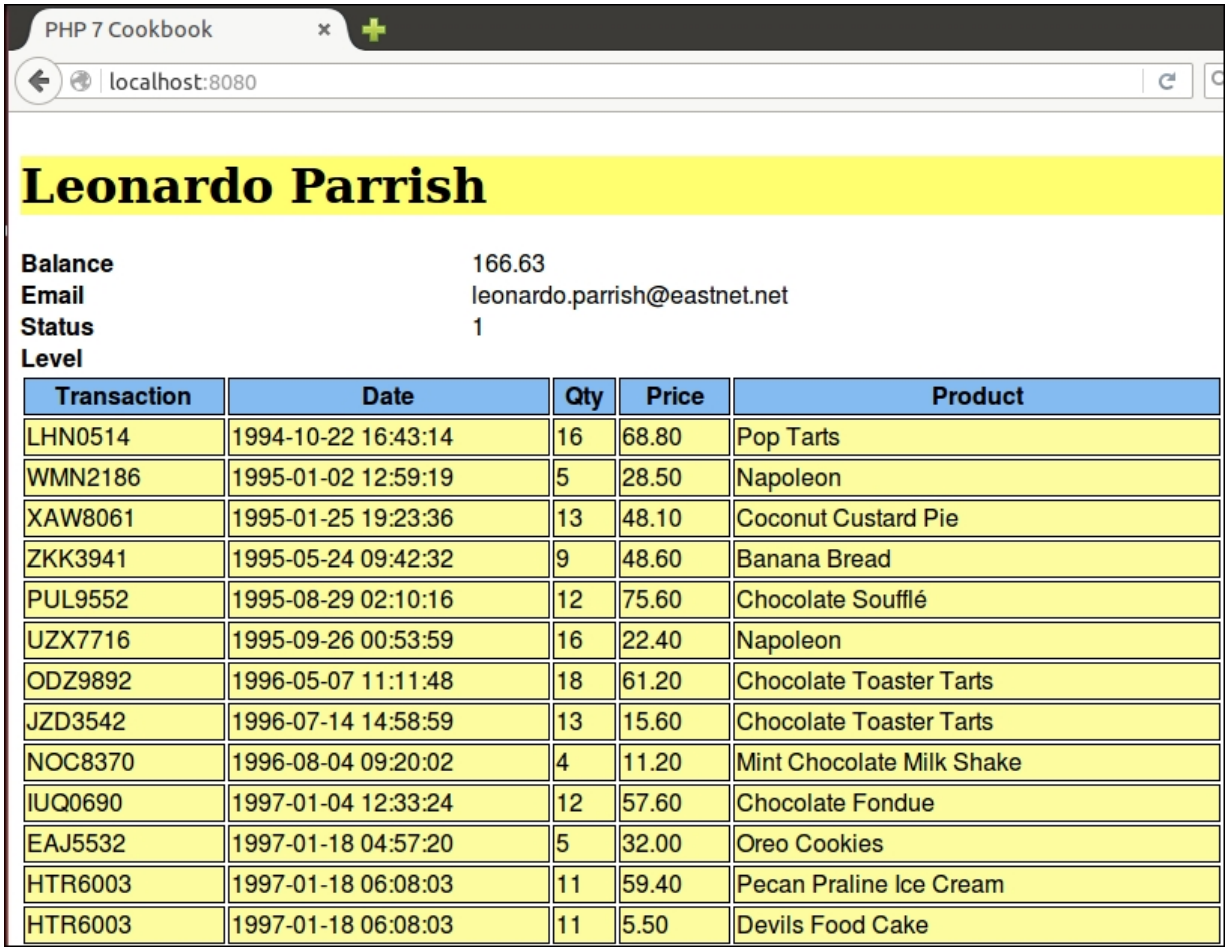
For the view logic, you can display core customer information as shown in several of the preceding recipes:

```
<h1><?= $result['name'] ?></h1>
<div class="row">
<div class="left">Balance</div>
<div class="right"><?= $result['balance']; ?>
</div>
</div>
<!-- etc.l -->
```

You can display information on purchases like so:

```
<table>
<tr><th>Transaction</th><th>Date</th><th>Qty</th>
<th>Price</th><th>Product</th></tr>
  <?php
  foreach ($result['purchases']($result['id'],
$conn) as $purchase) : ?>
  <tr>
    <td><?= $purchase['transaction'] ?></td>
    <td><?= $purchase['date'] ?></td>
    <td><?= $purchase['quantity'] ?></td>
    <td><?= $purchase['sale_price'] ?></td>
    <td><?= $purchase['title'] ?></td>
  </tr>
<?php endforeach; ?>
</table>
```

The critical piece is that the secondary lookup is performed as part of the view logic by calling the embedded anonymous function, `$result['purchases']($result['id'], $conn)`. Here is the output:



Leonardo Parrish

Balance 166.63
Email leonardo.parrish@eastnet.net
Status 1
Level

Transaction	Date	Qty	Price	Product
LHN0514	1994-10-22 16:43:14	16	68.80	Pop Tarts
WMN2186	1995-01-02 12:59:19	5	28.50	Napoleon
XAW8061	1995-01-25 19:23:36	13	48.10	Coconut Custard Pie
ZKK3941	1995-05-24 09:42:32	9	48.60	Banana Bread
PUL9552	1995-08-29 02:10:16	12	75.60	Chocolate Soufflé
UZX7716	1995-09-26 00:53:59	16	22.40	Napoleon
ODZ9892	1996-05-07 11:11:48	18	61.20	Chocolate Toaster Tarts
JZD3542	1996-07-14 14:58:59	13	15.60	Chocolate Toaster Tarts
NOC8370	1996-08-04 09:20:02	4	11.20	Mint Chocolate Milk Shake
IUQ0690	1997-01-04 12:33:24	12	57.60	Chocolate Fondue
EAJ5532	1997-01-18 04:57:20	5	32.00	Oreo Cookies
HTR6003	1997-01-18 06:08:03	11	59.40	Pecan Praline Ice Cream
HTR6003	1997-01-18 06:08:03	11	5.50	Devils Food Cake

Implementing jQuery DataTables PHP lookups

Another approach to secondary lookups is to have the frontend generate the request. In this recipe, we will make a slight modification to the secondary lookup code presented in the preceding recipe, Embedding secondary lookups into QueryResults. In the previous recipe, even though the view logic is performing the lookup, all processing is still done on the server. When using **jQuery DataTables**, however, the secondary lookup is actually performed directly by the client, in the form of an **Asynchronous JavaScript and XML** (**AJAX**) request issued by the browser.

How to do it...

1. First we need to spin-off the secondary lookup logic (discussed in the recipe above) into a separate PHP file. The purpose of this new script is to perform the secondary lookup and return a JSON array.
2. The new script we will call `chap_05_jquery_datatables_php_lookups_ajax.php`. It looks for a `$_GET` parameter, `id`. Notice that the `SELECT` statement is very specific as to which columns are delivered. You will also note that the fetch mode has been changed to `PDO::FETCH_NUM`. You might also notice that the last line takes the results and assigns it to a `data` key in a JSON-encoded array.

> ### Tip
>
> It is *extremely* important when dealing with zero configuration jQuery DataTables to only return the exact number of columns matching the header.

```
$id  = $_GET['id'] ?? 0;
sql = 'SELECT u.transaction,u.date,
u.quantity,u.sale_price,r.title '
   . 'FROM purchases AS u '
   . 'JOIN products AS r '
   . 'ON u.product_id = r.id '
   . 'WHERE u.customer_id = :id';
$stmt = $conn->pdo->prepare($sql);
$stmt->execute(['id' => (int) $id]);
$results = array();
while ($row = $stmt->fetch(PDO::FETCH_NUM)) {
  $results[] = $row;
}
echo json_encode(['data' => $results]);
```

3. Next, we need to modify the function that retrieves customer information by ID, removing the secondary lookup embedded in the previous recipe:

```
function findCustomerById($id, Connection
$conn)
{
  $stmt = $conn->pdo->query(
    'SELECT * FROM customer WHERE id = ' .
(int) $id);
  $results = $stmt->fetch(PDO::FETCH_ASSOC);
  return $results;
}
```

4. After that, in the view logic, we import the minimum jQuery, DataTables, and stylesheets for a zero configuration implementation. At a minimum, you will need jQuery itself (in this example `jquery-1.12.0.min.js`) and DataTables (`jquery.dataTables.js`). We've also added a convenient stylesheet associated with DataTables, `jquery.dataTables.css`:

```
<!DOCTYPE html>
<head>
  <script src="https://code.jquery.com/jquery-
1.12.0.min.js">
  </script>
    <script type="text/javascript"
      charset="utf8"

src="//cdn.datatables.net/1.10.11/js/jquery.da
```

```
taTables.js">
    </script>
  <link rel="stylesheet"
    type="text/css"

href="//cdn.datatables.net/1.10.11/css/jquery.
dataTables.css">
</head>
```

5. We then define a jQuery document `ready` function, which associates a table with DataTables. In this case, we assign an id attribute of `customerTable` to the table element that will be assigned to DataTables. You'll also notice that we specify the AJAX data source as the script defined in step 1, `chap_05_jquery_datatables_php_lookups_ajax.php`. As we have the `$id` available, this is appended to the data source URL:

```
<script>
$(document).ready(function() {
  $('#customerTable').DataTable(
    { "ajax":
'/chap_05_jquery_datatables_php_lookups_ajax.p
hp?id=<?= $id ?>'
  });
} );
</script>
```

6. In the body of the view logic, we define the table, making sure the `id` attribute matches the one specified in the preceding code. We also need to define headers that will match the data presented in response to the AJAX request:

```
<table id="customerTable" class="display"
cellspacing="0" width="100%">
  <thead>
    <tr>
      <th>Transaction</th>
      <th>Date</th>
      <th>Qty</th>
      <th>Price</th>
      <th>Product</th>
    </tr>
  </thead>
</table>
```

7. Now, all that remains to do is to load the page, choose the customer ID (in this case, at random), and let jQuery make the request for the secondary lookup.

How it works...

Create a `chap_05_jquery_datatables_php_lookups_ajax.php` script, which will respond to an AJAX request. Inside, place the code to initialize auto-loading and create a `Connection` instance. You can then append the code shown in step 2 of the preceding recipe:

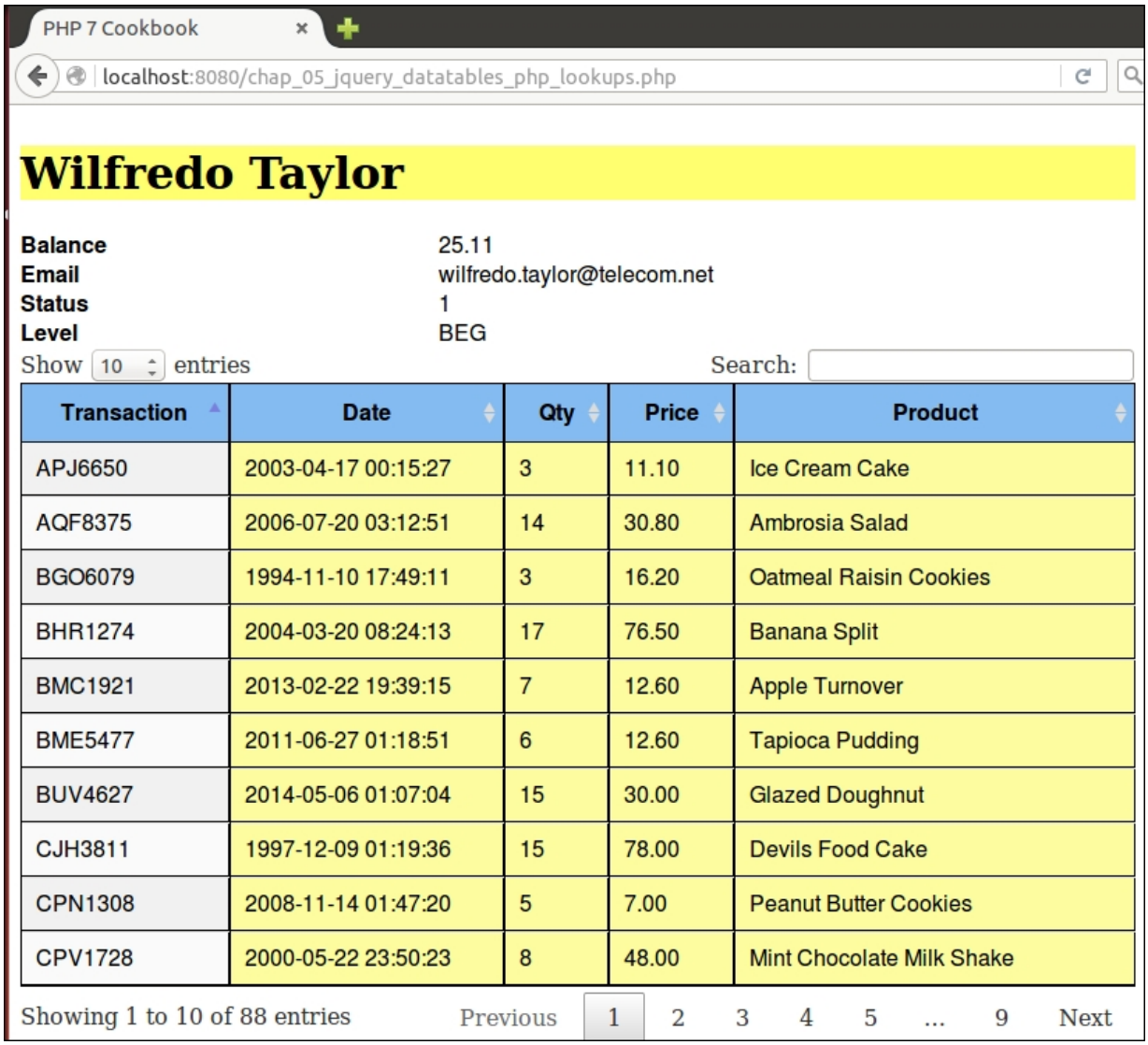
```
<?php
define('DB_CONFIG_FILE',
'/../config/db.config.php');
include __DIR__ .
'/../Application/Database/Connection.php';
use Application\Database\Connection;
$conn = new Connection(include __DIR__ .
DB_CONFIG_FILE);
```

Next, create a `chap_05_jquery_datatables_php_lookups.php` calling program that will pull information on a random customer. Add the function described in step 3 of the preceding code:

```
<?php
define('DB_CONFIG_FILE',
'/../config/db.config.php');
include __DIR__ .
'/../Application/Database/Connection.php';
use Application\Database\Connection;
$conn = new Connection(include __DIR__ .
DB_CONFIG_FILE);
// add function findCustomerById() here
$id     = random_int(1,79);
$result = findCustomerById($id, $conn);
?>
```

The calling program will also contain the view logic that imports the minimum JavaScript to implement jQuery DataTables. You can add the code shown in step 3 of the preceding code. Then, add the

document `ready` function and the display logic shown in steps 5 and 6. Here is the output:



There's more...

For more information on jQuery, please visit their website at https://jquery.com/. To read about the DataTables plugin to jQuery, refer to this article at https://www.datatables.net/. Zero configuration data tables are discussed at https://datatables.net/examples/basic_init/zero_configuration.html. For more information on AJAX sourced data, have a look at https://datatables.net/examples/data_sources/ajax.html.

Chapter 6. Building Scalable Websites

In this chapter, we will cover the following topics:

- Creating a generic form element generator
- Creating an HTML radio element generator
- Creating an HTML select element generator
- Implementing a form factory
- Chaining `$_POST` filters
- Chaining `$_POST` validators
- Tying validation to a form

Introduction

In this chapter, we will show you how to build classes that generate HTML form elements. The generic element generator can be used for text, text areas, passwords, and similar HTML input types. After that, we will show variations that allow you to pre-configure the element with an array of values. The form factory recipe will bring all these generators together, allowing you to render an entire form using a single configuration array. Finally, we introduce recipes that allow filtering and the validation of incoming `$_POST` data.

Creating a generic form element generator

It's pretty easy to create a function that simply outputs a form input tag such as `<input type="text" name="whatever" >`. In order to make a form generator generically useful, however, we need to think about the bigger picture. Here are some other considerations over and above the basic input tag:

- The form `input` tag and its associated HTML attributes
- A label that tells the user what information they are entering
- The ability to display entry errors following validation (more on that later!)
- Some sort of wrapper, such as a `<div>` tag, or an HTML table `<td>` tag

How to do it...

1. First, we define a `Application\Form\Generic` class. This will also later serve as a base class for specialized form elements:

```
namespace Application\Form;

class Generic
{
  // some code ...
}
```

2. Next, we define some class constants, which will be generally useful in form element generation.
3. The first three will become keys associated with the major components of a single form element. We then define supported input types and defaults:

```
const ROW = 'row';
const FORM = 'form';
const INPUT = 'input';
```

```
const LABEL = 'label';
const ERRORS = 'errors';
const TYPE_FORM = 'form';
const TYPE_TEXT = 'text';
const TYPE_EMAIL = 'email';
const TYPE_RADIO = 'radio';
const TYPE_SUBMIT = 'submit';
const TYPE_SELECT = 'select';
const TYPE_PASSWORD = 'password';
const TYPE_CHECKBOX = 'checkbox';
const DEFAULT_TYPE = self::TYPE_TEXT;
const DEFAULT_WRAPPER = 'div';
```

4. Next, we can define properties and a constructor that sets them.
5. In this example, we require two properties, `$name` and `$type`, as we cannot effectively use the element without these attributes. The other constructor arguments are optional. Furthermore, in order to base one form element on another, we include a provision whereby the second argument, `$type`, can alternatively be an instance of `Application\Form\Generic`, in which case we simply run the *getters* (discussed later) to populate properties:

```
protected $name;
protected $type    = self::DEFAULT_TYPE;
protected $label   = '';
protected $errors  = array();
protected $wrappers;
protected $attributes;    // HTML form
attributes
protected $pattern =  '<input type="%s"
name="%s" %s>';

public function __construct($name,
                $type,
                $label = '',
                array $wrappers = array(),
                array $attributes = array(),
                array $errors = array())
{
  $this->name = $name;
  if ($type instanceof Generic) {
      $this->type       = $type->getType();
      $this->label      = $type-
>getLabelValue();
      $this->errors     = $type-
>getErrorsArray();
```

```
            $this->wrappers   = $type-
>getWrappers();
            $this->attributes = $type-
>getAttributes();
        } else {
            $this->type       = $type ??
self::DEFAULT_TYPE;
            $this->label      = $label;
            $this->errors     = $errors;
            $this->attributes = $attributes;
            if ($wrappers) {
                $this->wrappers = $wrappers;
            } else {
                $this->wrappers[self::INPUT]['type']
=
                    self::DEFAULT_WRAPPER;
                $this->wrappers[self::LABEL]['type']
=
                    self::DEFAULT_WRAPPER;
                $this->wrappers[self::ERRORS]
['type'] =
                    self::DEFAULT_WRAPPER;
            }
        }
        $this->attributes['id'] = $name;
    }
```

Note

Note that `$wrappers` has three primary subkeys: `INPUT`, `LABEL`, and `ERRORS`. This allows us to define separate wrappers for labels, the input tag, and errors.

6. Before defining the core methods that will produce HTML for the label, input tag, and errors, we should define a `getWrapperPattern()` method, which will produce the appropriate *wrapping* tags for the label, input, and error display.
7. If, for example, the wrapper is defined as `<div>`, and its attributes include `['class' => 'label']`, this method will return a `sprintf()` format pattern that looks like this: `<div`

`class="label">%s</div>`. The final HTML produced for the label, for example, would then replace `%s`.

8. Here is how the `getWrapperPattern()` method might look:

```
public function getWrapperPattern($type)
{
  $pattern = '<' . $this->wrappers[$type]
['type'];
  foreach ($this->wrappers[$type] as $key =>
$value) {
    if ($key != 'type') {
      $pattern .= ' ' . $key . '="' . $value .
'"';
    }
  }
  $pattern .= '>%s</' . $this->wrappers[$type]
['type'] . '>';
  return $pattern;
}
```

9. We are now ready to define the `getLabel()` method. All this method needs to do is to plug the label into the wrapper using `sprintf()`:

```
public function getLabel()
{
  return sprintf($this-
>getWrapperPattern(self::LABEL),
                 $this->label);
}
```

10. In order to produce the core `input` tag, we need a way to assemble the attributes. Fortunately, this is easily accomplished as long as they are supplied to the constructor in the form of an associative array. All we need to do, in this case, is to define a `getAttribs()` method that produces a string of key-value pairs separated by a space. We return the final value using `trim()` to remove excess spaces.
11. If the element includes either the `value` or `href` attribute, for security reasons we should escape the values on the assumption that they are, or could be, user-supplied (and therefore suspect). Accordingly, we need to add an `if` statement that checks and then uses `htmlspecialchars()` or `urlencode()`:

```
public function getAttribs()
{
  foreach ($this->attributes as $key =>
$value) {
    $key = strtolower($key);
    if ($value) {
      if ($key == 'value') {
        if (is_array($value)) {
            foreach ($value as $k => $i)
              $value[$k] =
htmlspecialchars($i);
        } else {
            $value = htmlspecialchars($value);
        }
      } elseif ($key == 'href') {
          $value = urlencode($value);
      }
      $attribs .= $key . '="' . $value . '" ';
    } else {
        $attribs .= $key . ' ';
    }
  }
  return trim($attribs);
}
```

12. For the core input tag, we split the logic into two separate methods. The primary method, `getInputOnly()`, produces *only* the HTML input tag. The second method, `getInputWithWrapper()`, produces the input embedded in a wrapper. The reason for the split is that when creating spin-off classes, such as a class to generate radio buttons, we will not need the wrapper:

```
public function getInputOnly()
{
  return sprintf($this->pattern, $this->type,
$this->name,
                 $this->getAttribs());
}

public function getInputWithWrapper()
{
  return sprintf($this-
>getWrapperPattern(self::INPUT),
```

```
                    $this->getInputOnly());
    }
```

13. We now define a method that displays element validation errors. We will assume that the errors will be supplied in the form of an array. If there are no errors, we return an empty string. Otherwise, errors are rendered as `<ul><li>error 1</li> <li>error 2</li></ul>` and so on:

```
    public function getErrors()
    {
      if (!$this->errors || count($this->errors ==
    0)) return '';
      $html = '';
      $pattern = '<li>%s</li>';
      $html .= '<ul>';
      foreach ($this->errors as $error)
      $html .= sprintf($pattern, $error);
      $html .= '</ul>';
      return sprintf($this-
    >getWrapperPattern(self::ERRORS), $html);
    }
```

14. For certain attributes, we might need more finite control over various aspects of the property. As an example, we might need to add a single error to the already existing array of errors. Also, it might be useful to set a single attribute:

```
    public function setSingleAttribute($key,
    $value)
    {
      $this->attributes[$key] = $value;
    }
    public function addSingleError($error)
    {
      $this->errors[] = $error;
    }
```

15. Finally, we define getters and setters that allow us to retrieve or set the values of properties. For example, you might have noticed that the default value for `$pattern` is `<input type="%s" name="%s" %s>`. For certain tags (for example, `select` and `form` tags), we will need to set this property to a different value:

```
public function setPattern($pattern)
{
  $this->pattern = $pattern;
}
public function setType($type)
{
  $this->type = $type;
}
public function getType()
{
  return $this->type;
}
public function addSingleError($error)
{
  $this->errors[] = $error;
}
// define similar get and set methods
// for name, label, wrappers, errors and
attributes
```

16. We also need to add methods that will give the label value (not the HTML), as well as the errors array:

```
public function getLabelValue()
{
  return $this->label;
}
public function getErrorsArray()
{
  return $this->errors;
}
```

How it works...

Be sure to copy all the preceding code into a single `Application\Form\Generic` class. You can then define a `chap_06_form_element_generator.php` calling script that sets up autoloading and anchors the new class:

```
<?php
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
```

```
'/..');
use Application\Form\Generic;
```

Next, define the wrappers. For illustration, we'll use HTML table data and header tags. Note that the label uses `TH`, whereas input and errors use `TD`:

```
$wrappers = [
  Generic::INPUT => ['type' => 'td', 'class' =>
'content'],
  Generic::LABEL => ['type' => 'th', 'class' =>
'label'],
  Generic::ERRORS => ['type' => 'td', 'class' =>
'error']
];
```

You can now define an email element by passing parameters to the constructor:

```
$email = new Generic('email', Generic::TYPE_EMAIL,
'Email', $wrappers,
                    ['id' => 'email',
                     'maxLength' => 128,
                     'title' => 'Enter address',
                     'required' => '']);
```

Alternatively, define the password element using setters:

```
$password = new Generic('password', $email);
$password->setType(Generic::TYPE_PASSWORD);
$password->setLabel('Password');
$password->setAttributes(['id' => 'password',
                          'title' => 'Enter your
password',
                          'required' => '']);
```

Lastly, be sure to define a submit button:

```
$submit = new Generic('submit',
  Generic::TYPE_SUBMIT,
  'Login',
```

```
    $wrappers,
    ['id' => 'submit','title' => 'Click to
login','value' => 'Click Here']);
```

The actual display logic might look like this:

```
<div class="container">
  <!-- Login Form -->
  <h1>Login</h1>
  <form name="login" method="post">
  <table id="login" class="display"
    cellspacing="0" width="100%">
    <tr><?= $email->render(); ?></tr>
    <tr><?= $password->render(); ?></tr>
    <tr><?= $submit->render(); ?></tr>
    <tr>
      <td colspan=2>
        <br>
        <?php var_dump($_POST); ?>
      </td>
    </tr>
  </table>
  </form>
</div>
```

Here is the actual output:



Creating an HTML radio element generator

A radio button element generator will share similarities with the generic HTML form element generator. As with any generic element, a set of radio buttons needs the ability to display an overall label and errors. There are two major differences, however:

- Typically, you will want two or more radio buttons
- Each button needs to have its own label

How to do it...

1. First of all, create a new `Application\Form\Element\Radio` class that extends `Application\Form\Generic`:

```
namespace Application\Form\Element;
use Application\Form\Generic;
class Radio extends Generic
{
  // code
}
```

2. Next, we define class constants and properties that pertain to the special needs of a set of radio buttons.
3. In this illustration, we will need a `spacer`, which will be placed between the radio button and its label. We also need to decide whether to place the radio button label before or after the actual button, thus, we use the `$after` flag. If we need a default, or if we are re-displaying existing form data, we need a way of designating the selected key. Finally, we need an array of options from which we will populate the list of buttons:

```
const DEFAULT_AFTER = TRUE;
const DEFAULT_SPACER = '&nbps;';
const DEFAULT_OPTION_KEY = 0;
const DEFAULT_OPTION_VALUE = 'Choose';
```

```
protected $after = self::DEFAULT_AFTER;
protected $spacer = self::DEFAULT_SPACER;
protected $options = array();
protected $selectedKey = DEFAULT_OPTION_KEY;
```

4. Given that we are extending `Application\Form\Generic`, we have the option of expanding the `__construct()` method, or, alternatively, simply defining a method that can be used to set specific options. For this illustration, we have chosen the latter course.
5. To ensure the property `$this->options` is populated, the first parameter (`$options`) is defined as mandatory (without a default). All other parameters are optional.

```
public function setOptions(array $options,
  $selectedKey = self::DEFAULT_OPTION_KEY,
  $spacer = self::DEFAULT_SPACER,
  $after  = TRUE)
{
  $this->after = $after;
  $this->spacer = $spacer;
  $this->options = $options;
  $this->selectedKey = $selectedKey;
}
```

6. Finally, we are ready to override the core `getInputOnly()` method.
7. We save the `id` attribute into an independent variable, `$baseId`, and later combine it with `$count` so that each `id` attribute is unique. If the option associated with the selected key is defined, it is assigned as the value; otherwise, we use the default:

```
public function getInputOnly()
{
  $count  = 1;
  $baseId = $this->attributes['id'];
```

8. Inside the `foreach()` loop we check to see if the key is the one selected. If so, the `checked` attribute is added for that radio button. We then call the parent class `getInputOnly()` method to return the HTML for each button. Note that the `value` attribute of the input element is the options array key. The button label is the options array element value:

```
        foreach ($this->options as $key => $value) {
          $this->attributes['id'] = $baseId .
        $count++;
          $this->attributes['value'] = $key;
          if ($key == $this->selectedKey) {
              $this->attributes['checked'] = '';
          } elseif (isset($this-
        >attributes['checked'])) {
                    unset($this-
        >attributes['checked']);
          }
          if ($this->after) {
              $html = parent::getInputOnly() . $value;
          } else {
              $html = $value . parent::getInputOnly();
          }
          $output .= $this->spacer . $html;
          }
          return $output;
        }
```

How it works...

Copy the preceding code into a new `Radio.php` file in the `Application/Form/Element` folder. You can then define a `chap_06_form_element_radio.php` calling script that sets up autoloading and anchors the new class:

```
<?php
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
use Application\Form\Generic;
use Application\Form\Element\Radio;
```

Next, define the wrappers using the `$wrappers` array defined in the previous recipe.

Then you can define a `$status` array and create an element instance by passing parameters to the constructor:

```
$statusList = [
  'U' => 'Unconfirmed',
  'P' => 'Pending',
  'T' => 'Temporary Approval',
  'A' => 'Approved'
];

$status = new Radio('status',
          Generic::TYPE_RADIO,
          'Status',
          $wrappers,
          ['id' => 'status']);
```

Now you can see if there is any status input from `$_GET` and set the options. Any input will become the selected key. Otherwise, the selected key is the default:

```
$checked = $_GET['status'] ?? 'U';
$status->setOptions($statusList, $checked, '<br>',
TRUE);
```

Lastly, don't forget to define a submit button:

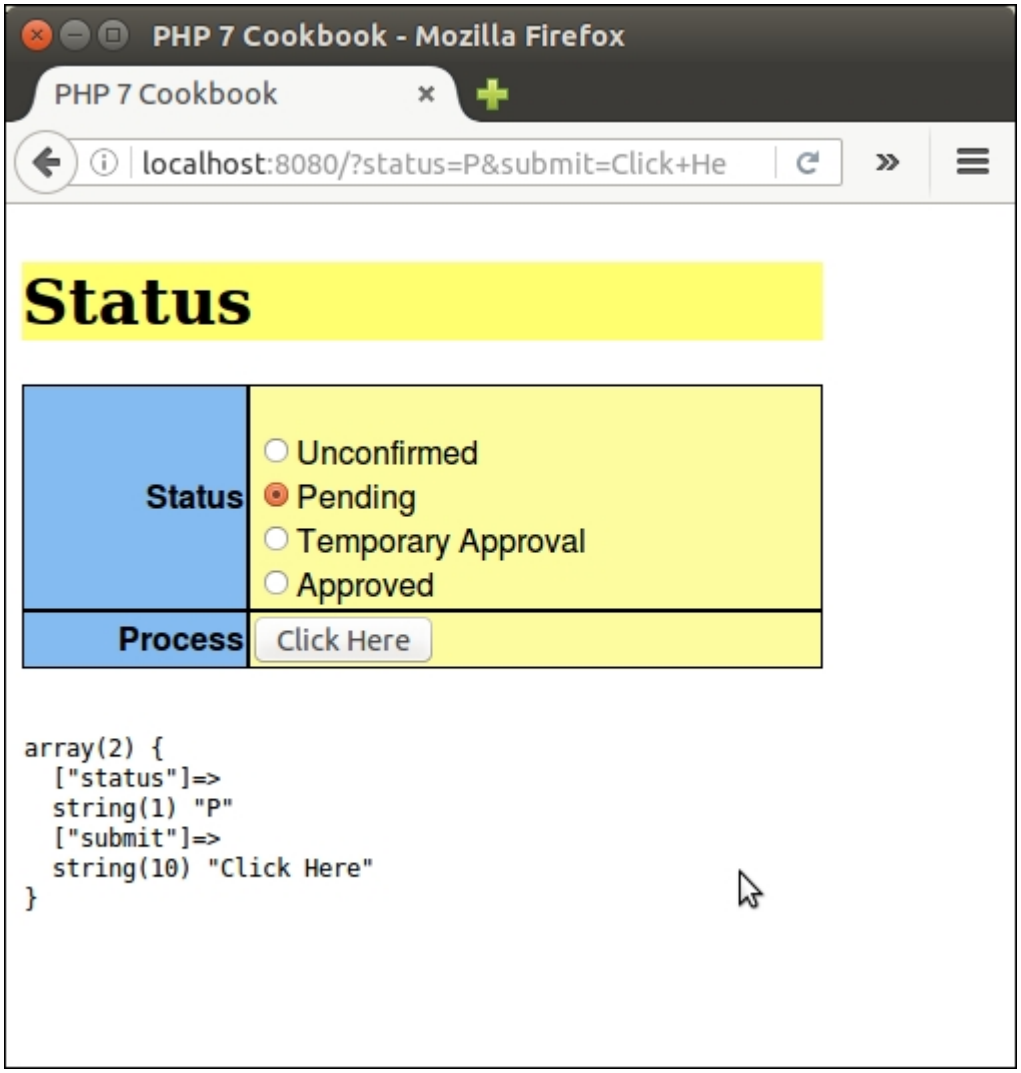
```
$submit = new Generic('submit',
          Generic::TYPE_SUBMIT,
          'Process',
          $wrappers,
          ['id' => 'submit','title' =>
          'Click to process','value' => 'Click
Here']);
```

The display logic might look like this:

```
<form name="status" method="get">
<table id="status" class="display" cellspacing="0"
width="100%">
  <tr><?= $status->render(); ?></tr>
  <tr><?= $submit->render(); ?></tr>
  <tr>
    <td colspan=2>
      <br>
      <pre><?php var_dump($_GET); ?></pre>
```

```
    </td>
  </tr>
</table>
</form>
```

Here is the actual output:



There's more...

A checkbox element generator would be almost identical to the HTML radio button generator. The main difference is that a set of checkboxes can have more than one value checked. Accordingly,

you would use PHP array notation for the element names. The element type should be `Generic::TYPE_CHECKBOX`.

Creating an HTML select element generator

Generating an HTML single select element is similar to the process of generating radio buttons. The tags are structured differently, however, in that both a `SELECT` tag and a series of `OPTION` tags need to be generated.

How to do it...

1. First of all, create a new `Application\Form\Element\Select` class that extends `Application\Form\Generic`.
2. The reason why we extend `Generic` rather than `Radio` is because the structuring of the element is entirely different:

```
namespace Application\Form\Element;

use Application\Form\Generic;

class Select extends Generic
{
  // code
}
```

3. The class constants and properties will only need to add slightly to `Application\Form\Generic`. Unlike radio buttons or checkboxes, there is no need to account for *spacers* or the placement of the selected text:

```
const DEFAULT_OPTION_KEY = 0;
const DEFAULT_OPTION_VALUE = 'Choose';

protected $options;
protected $selectedKey = DEFAULT_OPTION_KEY;
```

4. Now we turn our attention to setting options. As an HTML select element can select single or multiple values, the `$selectedKey` property could be either a string or an array. Accordingly, we do

not add a **type hint** for this property. It is important, however, that we identify whether or not the `multiple` attribute has been set. This can be obtained from a `$this->attributes` property via inheritance from the parent class.

5. If the `multiple` attribute has been set, it's important to formulate the `name` attribute as an array. Accordingly, we would append `[]` to the name if this were the case:

```
public function setOptions(array $options,
$selectedKey = self::DEFAULT_OPTION_KEY)
{
  $this->options = $options;
  $this->selectedKey = $selectedKey;
  if (isset($this->attributes['multiple'])) {
    $this->name .= '[]';
  }
}
```

> ## Note
>
> In PHP, if the HTML select `multiple` attribute has been set, and the `name` attribute is not specified as an array, only a single value will be returned!

6. Before we can define the core `getInputOnly()` method, we need to define a method to generate the `select` tag. We then return the final HTML using `sprintf()`, using `$pattern`, `$name`, and the return value of `getAttribs()` as arguments.
7. We replace the default value for `$pattern` with `<select name="%s" %s>`. We then loop through the attributes, adding them as key-value pairs with spaces in between:

```
protected function getSelect()
{
  $this->pattern = '<select name="%s" %s> ' .
PHP_EOL;
  return sprintf($this->pattern, $this->name,
  $this->getAttribs());
}
```

8. Next, we define a method to obtain the `option` tags that will be associated with the `select` tag.
9. As you will recall, the *key* from the `$this->options` array represents the return value, whereas the *value* part of the array represents the text that will appear on screen. If `$this->selectedKey` is in array form, we check to see if the value is in the array. Otherwise, we assume `$this-> selectedKey` is a string and we simply determine if it is equal to the key. If the selected key matches, we add the `selected` attribute:

```
protected function getOptions()
{
  $output = '';
  foreach ($this->options as $key => $value) {
    if (is_array($this->selectedKey)) {
        $selected = (in_array($key, $this-
>selectedKey))
        ? ' selected' : '';
    } else {
        $selected = ($key == $this-
>selectedKey)
        ? ' selected' : '';
    }
        $output .= '<option value="' . $key .
'"'
        . $selected  . '>'
        . $value
        . '</option>';
  }
  return $output;
}
```

10. Finally we are ready to override the core `getInputOnly()` method.
11. You will note that the logic for this method only needs to capture the return values from the `getSelect()` and `getOptions()` methods described in the preceding code. We also need to add the closing `</select>` tag:

```
public function getInputOnly()
{
  $output = $this->getSelect();
  $output .= $this->getOptions();
```

```
        $output .= '</' . $this->getType() . '>';
        return $output;
    }
```

How it works...

Copy the code described above into a new `Select.php` file in the `Application/Form/Element` folder. Then define a `chap_06_form_element_select.php` calling script that sets up autoloading and anchors the new class:

```
<?php
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
use Application\Form\Generic;
use Application\Form\Element\Select;
```

Next, define the wrappers using the array `$wrappers` defined in the first recipe. You can also use the `$statusList` array defined in the *Creating an HTML radio element generator* recipe. You can then create instances of `SELECT` elements. The first instance is single select, and the second is multiple:

```
$status1 = new Select('status1',
           Generic::TYPE_SELECT,
           'Status 1',
           $wrappers,
           ['id' => 'status1']);
$status2 = new Select('status2',
           Generic::TYPE_SELECT,
           'Status 2',
           $wrappers,
           ['id' => 'status2',
            'multiple' => '',
            'size' => '4']);
```

See if there is any status input from `$_GET` and set the options. Any input will become the selected key. Otherwise, the selected key is

the default. As you will recall, the second instance is multiple select, so the value obtained from `$_GET` and the default setting should both be in the form of an array:

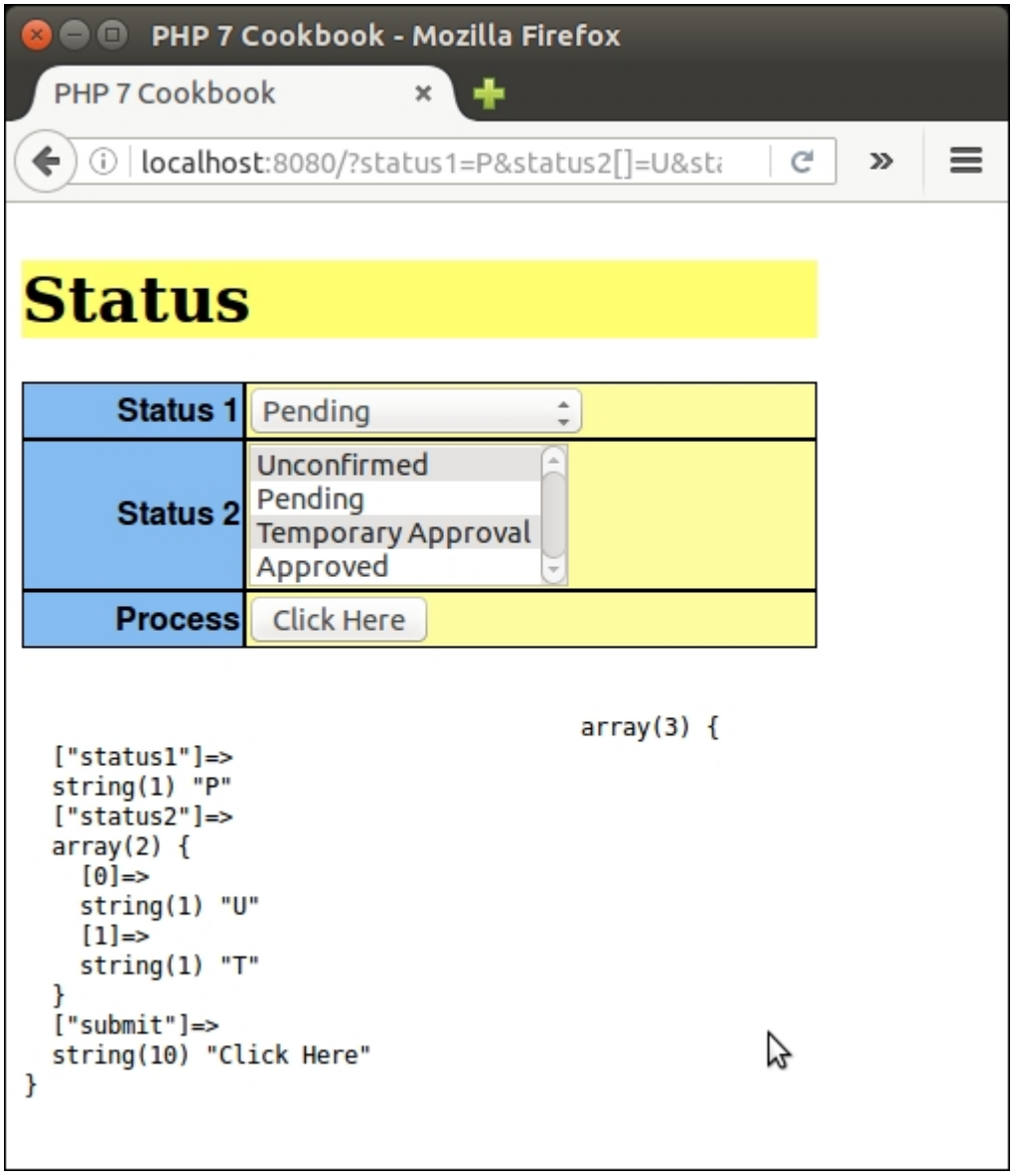
```
$checked1 = $_GET['status1'] ?? 'U';
$checked2 = $_GET['status2'] ?? ['U'];
$status1->setOptions($statusList, $checked1);
$status2->setOptions($statusList, $checked2);
```

Lastly, be sure to define a submit button (as shown in the *Creating a generic form element generator* recipe of this chapter).

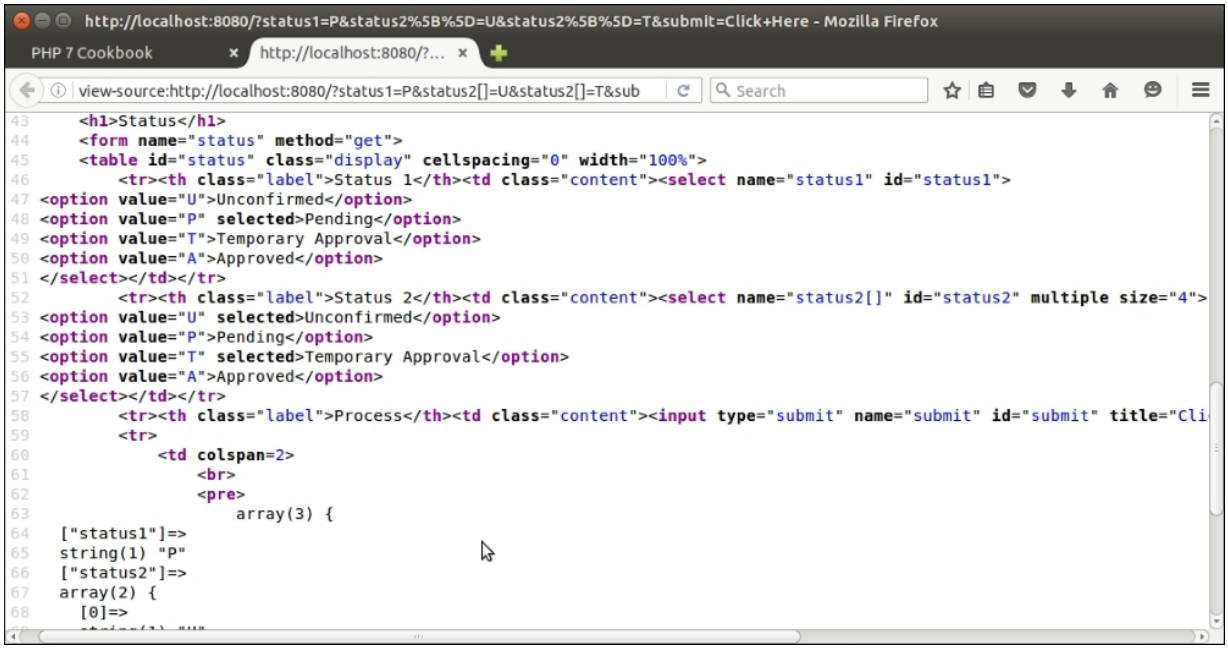
The actual display logic is identical to the radio button recipe, except that we need to render two separate HTML select instances:

```
<form name="status" method="get">
<table id="status" class="display" cellspacing="0"
width="100%">
  <tr><?= $status1->render(); ?></tr>
  <tr><?= $status2->render(); ?></tr>
  <tr><?= $submit->render(); ?></tr>
  <tr>
    <td colspan=2>
      <br>
      <pre>
        <?php var_dump($_GET); ?>
      </pre>
    </td>
  </tr>
</table>
</form>
```

Here is the actual output:



Also, you can see how the elements appear in the *view source* page:


http://localhost:8080/?status1=P&status2%5B%5D=U&status2%5B%5D=T&submit=Click+Here - Mozilla Firefox
PHP 7 Cookbook
http://localhost:8080/?...
view-source:http://localhost:8080/?status1=P&status2[]=U&status2[]=T&sub
Search
43 <h1>Status</h1>
44 <form name="status" method="get">
45 <table id="status" class="display" cellspacing="0" width="100%">
46 <tr><th class="label">Status 1</th><td class="content"><select name="status1" id="status1">
47 <option value="U">Unconfirmed</option>
48 <option value="P" selected>Pending</option>
49 <option value="T">Temporary Approval</option>
50 <option value="A">Approved</option>
51 </select></td></tr>
52 <tr><th class="label">Status 2</th><td class="content"><select name="status2[]" id="status2" multiple size="4">
53 <option value="U" selected>Unconfirmed</option>
54 <option value="P">Pending</option>
55 <option value="T" selected>Temporary Approval</option>
56 <option value="A">Approved</option>
57 </select></td></tr>
58 <tr><th class="label">Process</th><td class="content"><input type="submit" name="submit" id="submit" title="Cli
59 <tr>
60 <td colspan=2>
61
62 <pre>
63 array(3) {
64 ["status1"]=>
65 string(1) "P"
66 ["status2"]=>
67 array(2) {
68 [0]=>

Implementing a form factory

The purpose of a form factory is to generate a usable form object from a single configuration array. The form object should have the ability to retrieve the individual elements it contains so that output can be generated.

How to do it...

1. First, let's create a class called `Application\Form\Factory` to contain the factory code. It will have only one property, `$elements`, with a getter:

```
namespace Application\Form;

class Factory
{
  protected $elements;
  public function getElements()
  {
    return $this->elements;
  }
  // remaining code
}
```

2. Before we define the primary form generation method, it's important to consider what configuration format we plan to receive, and what exactly the form generation will produce. For this illustration, we will assume that the generation will produce a `Factory` instance, with an `$elements` property. This property would be an array of `Application\Form\Generic` or `Application\Form\Element` classes.
3. We are now ready to tackle the `generate()` method. This will cycle through the configuration array, creating the appropriate `Application\Form\Generic` or `Application\Form\Element\*` objects, which in turn will be stored in the `$elements` array. The new method will accept the configuration array as an argument. It is convenient to define this method as static so that we can

generate as many instances as are needed using different blocks of configuration.

4. We create an instance of `Application\Form\Factory`, and then we start looping through the configuration array:

```
public static function generate(array $config)
{
  $form = new self();
  foreach ($config as $key => $p) {
```

5. Next, we check for parameters that are optional in the constructor for the `Application\Form\Generic` class:

```
    $p['errors'] = $p['errors'] ?? array();
    $p['wrappers'] = $p['wrappers'] ?? array();
    $p['attributes'] = $p['attributes'] ??
array();
```

6. Now that all the constructor parameters are in place, we can create the form element instance, which is then stored in `$elements`:

```
    $form->elements[$key] = new $p['class']
    (
      $key,
      $p['type'],
      $p['label'],
      $p['wrappers'],
      $p['attributes'],
      $p['errors']
    );
```

7. Next, we turn our attention to options. If the `options` parameter is set, we extract the array values into variables using `list()`. We then test the element type using `switch()` and run `setOptions()` with the appropriate number of parameters:

```
    if (isset($p['options'])) {
      list($a,$b,$c,$d) = $p['options'];
      switch ($p['type']) {
        case Generic::TYPE_RADIO    :
        case Generic::TYPE_CHECKBOX :
          $form->elements[$key]-
>setOptions($a,$b,$c,$d);
          break;
```

```
            case Generic::TYPE_SELECT   :
              $form->elements[$key]-
>setOptions($a,$b);
              break;
            default                     :
              $form->elements[$key]-
>setOptions($a,$b);
              break;
          }
        }
      }
```

8. Finally, we return the form object and close out the method:

```
      return $form;
}
```

9. Theoretically, at this point, we could easily render the form in our view logic by simply iterating through the array of elements and running the `render()` method. The view logic might look like this:

```
<form name="status" method="get">
  <table id="status" class="display"
cellspacing="0" width="100%">
    <?php foreach ($form->getElements() as
$element) : ?>
      <?php echo $element->render(); ?>
    <?php endforeach; ?>
  </table>
</form>
```

10. Finally, we return the form object and close out the method.
11. Next, we need to define a discrete `Form` class under `Application\Form\Element`:

```
namespace Application\Form\Element;
class Form extends Generic
{
  public function getInputOnly()
  {
    $this->pattern = '<form name="%s" %s> ' .
PHP_EOL;
    return sprintf($this->pattern, $this-
>name,
                   $this->getAttribs());
  }
```

```
  public function closeTag()
  {
    return '</' . $this->type . '>';
  }
}
```

12. Returning to the `Application\Form\Factory` class, we now need to define a simple method that returns a `sprintf()` wrapper pattern that will serve as an envelope for input. As an example, if the wrapper is `div` with an attribute `class="test"` we would produce this pattern: `<div class="test">%s</div>`. Our content would then be substituted in place of `%s` by the `sprintf()` function:

```
protected function getWrapperPattern($wrapper)
{
  $type = $wrapper['type'];
  unset($wrapper['type']);
  $pattern = '<' . $type;
  foreach ($wrapper as $key => $value) {
    $pattern .= ' ' . $key . '="' . $value .
'"';
  }
  $pattern .= '>%s</' . $type . '>';
  return $pattern;
}
```

13. Finally, we are ready to define a method that does overall form rendering. We obtain wrapper `sprintf()` patterns for each form row. We then loop through the elements, render each one, and wrap the output in the row pattern. Next, we generate an `Application\Form\Element\Form` instance. We then retrieve the form wrapper `sprintf()` pattern and check the `form_tag_inside_wrapper` flag, which tells us whether we need to place the form tag inside or outside the form wrapper:

```
public static function render($form,
$formConfig)
{
  $rowPattern = $form->getWrapperPattern(
  $formConfig['row_wrapper']);
  $contents   = '';
  foreach ($form->getElements() as $element) {
    $contents .= sprintf($rowPattern,
```

```
$element->render());
  }
  $formTag = new Form($formConfig['name'],
                 Generic::TYPE_FORM,
                 '',
                 array(),
                 $formConfig['attributes']);

  $formPattern = $form->getWrapperPattern(
  $formConfig['form_wrapper']);
  if
(isset($formConfig['form_tag_inside_wrapper'])
      &&
!$formConfig['form_tag_inside_wrapper']) {
        $formPattern = '%s' . $formPattern .
'%s';
        return sprintf($formPattern, $formTag-
>getInputOnly(),
        $contents, $formTag->closeTag());
  } else {
        return sprintf($formPattern, $formTag-
>getInputOnly()
        . $contents . $formTag->closeTag());
  }
}
```

How it works...

Referring to the preceding code, create the `Application\Form\Factory` and `Application\Form\Element\Form` classes.

Next, you can define a `chap_06_form_factor.php` calling script that sets up autoloading and anchors the new class:

```
<?php
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
use Application\Form\Generic;
use Application\Form\Factory;
```

Next, define the wrappers using the `$wrappers` array defined in the first recipe. You can also use the `$statusList` array defined in the second recipe.

See if there is any status input from `$_POST`. Any input will become the selected key. Otherwise, the selected key is the default.

```
$email    = $_POST['email']   ?? '';
$checked0 = $_POST['status0'] ?? 'U';
$checked1 = $_POST['status1'] ?? 'U';
$checked2 = $_POST['status2'] ?? ['U'];
$checked3 = $_POST['status3'] ?? ['U'];
```

Now you can define the overall form configuration. The `name` and `attributes` parameters are used to configure the `form` tag itself. The other two parameters represent form-level and row-level wrappers. Lastly, we provide a `form_tag_inside_wrapper` flag to indicate that the form tag should *not* appear inside the wrapper (that is, `<table>`). If the wrapper was `<div>`, we would set this flag to `TRUE`:

```
$formConfig = [
  'name'         => 'status_form',
  'attributes'   =>
['id'=>'statusForm','method'=>'post',
'action'=>'chap_06_form_factory.php'],
  'row_wrapper'  => ['type' => 'tr', 'class' =>
'row'],
  'form_wrapper' =>
['type'=>'table','class'=>'table',
'id'=>'statusTable',

'class'=>'display','cellspacing'=>'0'],
                     'form_tag_inside_wrapper' =>
FALSE,
];
```

Next, define an array that holds parameters for each form element to be created by the factory. The array key becomes the name of the form element, and must be unique:

```
$config = [
  'email' => [
    'class'      => 'Application\Form\Generic',
    'type'       => Generic::TYPE_EMAIL,
    'label'      => 'Email',
    'wrappers'   => $wrappers,
    'attributes'=>
['id'=>'email','maxLength'=>128, 'title'=>'Enter
address',

'required'=>'','value'=>strip_tags($email)]
  ],
  'password' => [
    'class'       => 'Application\Form\Generic',
    'type'        => Generic::TYPE_PASSWORD,
    'label'       => 'Password',
    'wrappers'    => $wrappers,
    'attributes'  => ['id'=>'password',
    'title'       => 'Enter your password',
    'required'    => '']
  ],
  // etc.
];
```

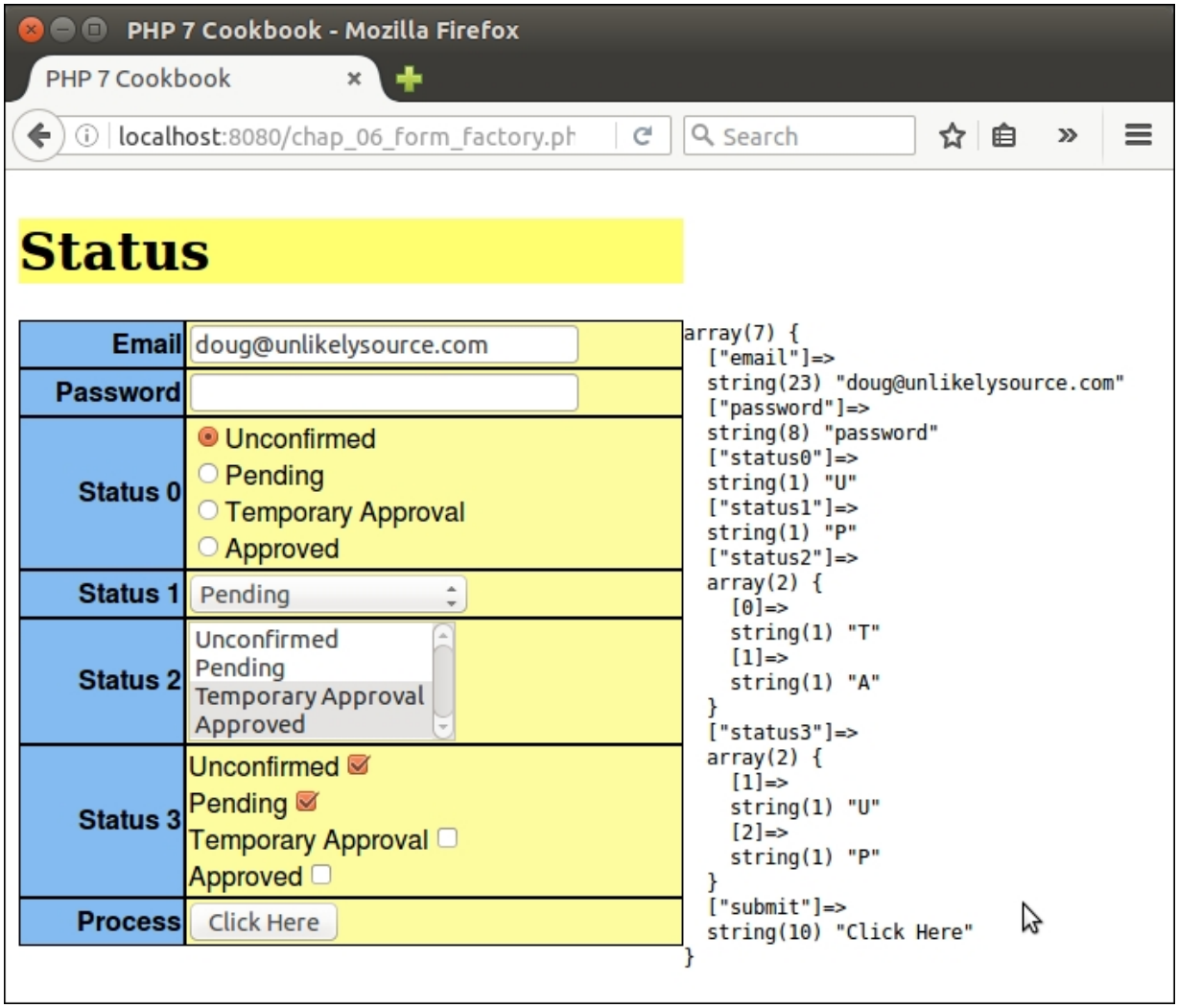
Lastly, be sure to generate the form:

```
$form = Factory::generate($config);
```

The actual display logic is extremely simple, as we simply call the form level `render()` method:

```
<?= $form->render($form, $formConfig); ?>
```

Here is the actual output:


PHP 7 Cookbook - Mozilla Firefox
PHP 7 Cookbook
localhost:8080/chap_06_form_factory.ph
Search
Status
Email
doug@unlikelysource.com
Password
Status 0
Unconfirmed
Pending
Temporary Approval
Approved
Status 1
Pending
Status 2
Unconfirmed
Pending
Temporary Approval
Approved
Status 3
Unconfirmed
Pending
Temporary Approval
Approved
Process
Click Here
array(7) {
["email"]=>
string(23) "doug@unlikelysource.com"
["password"]=>
string(8) "password"
["status0"]=>
string(1) "U"
["status1"]=>
string(1) "P"
["status2"]=>
array(2) {
[0]=>
string(1) "T"
[1]=>
string(1) "A"
}
["status3"]=>
array(2) {
[1]=>
string(1) "U"
[2]=>
string(1) "P"
}
["submit"]=>
string(10) "Click Here"
}

Chaining $_POST filters

Proper filtering and validation is a common problem when processing data submitted by users from an online form. It is arguably also the number one security vulnerability for a website. Furthermore, it can be quite awkward to have the filters and validators scattered all over the application. A chaining mechanism would resolve these issues neatly, and would also allow you to exert control over the order in which the filters and validators are processed.

How to do it...

1. There is a little-known PHP function, `filter_input_array()`, that, at first glance, seems well suited for this task. Looking more deeply into its functionality, however, it soon becomes apparent that this function was designed in the early days, and is not up to modern requirements for protection against attack and flexibility. Accordingly, we will instead present a much more flexible mechanism based on an array of callbacks performing filtering and validation.

   > **Note**
   >
   > The difference between *filtering* and *validation* is that filtering can potentially remove or transform values. Validation, on the other hand, tests data using criteria appropriate to the nature of the data, and returns a boolean result.

2. In order to increase flexibility, we will make our base filter and validation classes relatively light. By this, we mean *not* defining any specific filters or validation methods. Instead, we will operate entirely on the basis of a configuration array of

callbacks. In order to ensure compatibility in filtering and validation results, we will also define a specific result object, `Application\Filter\Result`.

3. The primary function of the `Result` class will be to hold a `$item` value, which would be the filtered value or a boolean result of validation. Another property, `$messages`, will hold an array of messages populated during the filtering or validation operation. In the constructor, the value supplied for `$messages` is formulated as an array. You might observe that both properties are defined `public`. This is to facilitate ease of access:

```
namespace Application\Filter;

class Result
{

  public $item;  // (mixed) filtered data |
(bool) result of validation
  public $messages = array();  // [(string)
message, (string) message ]

  public function __construct($item,
$messages)
  {
    $this->item = $item;
    if (is_array($messages)) {
        $this->messages = $messages;
    } else {
        $this->messages = [$messages];
    }
  }
```

4. We also define a method that allows us to merge this `Result` instance with another. This is important as at some point we will be processing the same value through a chain of filters. In such a case, we want the newly filtered value to overwrite the existing one, but we want the messages to be merged:

```
public function mergeResults(Result $result)
{
  $this->item = $result->item;
  $this->mergeMessages($result);
}
```

```
public function mergeMessages(Result $result)
{
  if (isset($result->messages) &&
is_array($result->messages)) {
    $this->messages = array_merge($this-
>messages, $result->messages);
  }
}
```

5. Finally, to finish the methods for this class, we add a method that merges validation results. The important consideration here is that *any* value of `FALSE`, up or down the validation chain, must cause the *entire* result to be `FALSE`:

```
public function mergeValidationResults(Result
$result)
{
  if ($this->item === TRUE) {
    $this->item = (bool) $result->item;
  }
  $this->mergeMessages($result);
  }

}
```

6. Next, to make sure that the callbacks produce compatible results, we will define an `Application\Filter\CallbackInterface` interface. You will note that we are taking advantage of the PHP 7 ability to data type the return value to ensure that we are getting a `Result` instance in return:

```
namespace Application\Filter;
interface CallbackInterface
{
  public function __invoke ($item, $params) :
Result;
}
```

7. Each callback should reference the same set of messages. Accordingly, we define a `Application\Filter\Messages` class with a series of static properties. We provide methods to set all messages, or just one message. The `$messages` property has been made `public` for easier access:

```
namespace Application\Filter;
class Messages
{
  const MESSAGE_UNKNOWN = 'Unknown';
  public static $messages;
  public static function setMessages(array
$messages)
  {
    self::$messages = $messages;
  }
  public static function setMessage($key,
$message)
  {
    self::$messages[$key] = $message;
  }
  public static function getMessage($key)
  {
    return self::$messages[$key] ??
self::MESSAGE_UNKNOWN;
  }
}
```

8. We are now in a position to define a `Application\Web\AbstractFilter` class that implements core functionality. As mentioned previously, this class will be relatively *lightweight* and we do not need to worry about specific filters or validators as they will be supplied through configuration. We use the `UnexpectedValueException` class, provided as part of the PHP 7 **Standard PHP Library** (**SPL**), in order to throw a descriptive exception in case one of the callbacks does not implement `CallbackInterface`:

```
namespace Application\Filter;
use UnexpectedValueException;
abstract class AbstractFilter
{
  // code described in the next several
bullets
```

9. First, we define useful class constants that hold various *housekeeping* values. The last four shown here control the format of messages to be displayed, and how to describe *missing* data:

```
const BAD_CALLBACK = 'Must implement
CallbackInterface';
const DEFAULT_SEPARATOR = '<br>' . PHP_EOL;
const MISSING_MESSAGE_KEY = 'item.missing';
const DEFAULT_MESSAGE_FORMAT = '%20s : %60s';
const DEFAULT_MISSING_MESSAGE = 'Item
Missing';
```

10. Next, we define core properties. `$separator` is used in conjunction with filtering and validation messages. `$callbacks` represents the array of callbacks that perform filtering and validation. `$assignments` map data fields to filters and/or validators. `$missingMessage` is represented as a property so that it can be overwritten (that is, for multi-language websites). Finally, `$results` is an array of `Application\Filter\Result` objects and is populated by the filtering or validation operation:

```
protected $separator;    // used for message
display
protected $callbacks;
protected $assignments;
protected $missingMessage;
protected $results = array();
```

11. At this point, we can build the `__construct()` method. Its main function is to set the array of callbacks and assignments. It also either sets values or accepts defaults for the separator (used in message display), and the *missing* message:

```
public function __construct(array $callbacks,
array $assignments,
                            $separator = NULL,
$message = NULL)
{
  $this->setCallbacks($callbacks);
  $this->setAssignments($assignments);
  $this->setSeparator($separator ??
self::DEFAULT_SEPARATOR);
  $this->setMissingMessage($message
                           ??
self::DEFAULT_MISSING_MESSAGE);
}
```

12. Next, we define a series of methods that allow us to set or remove callbacks. Notice that we allow the getting and setting of a single callback. This is useful if you have a generic set of callbacks, and need to modify just one. You will also note that `setOneCall()` checks to see if the callback implements `CallbackInterface`. If it does not, an `UnexpectedValueException` is thrown:

```
public function getCallbacks()
{
  return $this->callbacks;
}

public function getOneCallback($key)
{
  return $this->callbacks[$key] ?? NULL;
}

public function setCallbacks(array $callbacks)
{
  foreach ($callbacks as $key => $item) {
    $this->setOneCallback($key, $item);
  }
}

public function setOneCallback($key, $item)
{
  if ($item instanceof CallbackInterface) {
      $this->callbacks[$key] = $item;
  } else {
      throw new
UnexpectedValueException(self::BAD_CALLBACK);
  }
}

public function removeOneCallback($key)
{
  if (isset($this->callbacks[$key]))
  unset($this->callbacks[$key]);
}
```

13. Methods for results processing are quite simple. For convenience, we added `getItemsAsArray()`, otherwise `getResults()` will return an array of `Result` objects:

```
public function getResults()
{
  return $this->results;
}

public function getItemsAsArray()
{
  $return = array();
  if ($this->results) {
    foreach ($this->results as $key => $item)
    $return[$key] = $item->item;
  }
  return $return;
}
```

14. Retrieving messages is just a matter of looping through the array of `$this ->results` and extracting the `$messages` property. For convenience, we also added `getMessageString()` with some formatting options. To easily produce an array of messages, we use the PHP 7 `yield from` syntax. This has the effect of turning `getMessages()` into a **delegating generator**. The array of messages becomes a **sub-generator**:

```
public function getMessages()
{
  if ($this->results) {
      foreach ($this->results as $key =>
$item)
      if ($item->messages) yield from $item-
>messages;
  } else {
      return array();
  }
}

public function getMessageString($width = 80,
$format = NULL)
{
  if (!$format)
  $format = self::DEFAULT_MESSAGE_FORMAT .
$this->separator;
  $output = '';
  if ($this->results) {
    foreach ($this->results as $key => $value)
{
```

```
            if ($value->messages) {
              foreach ($value->messages as $message)
{
                $output .= sprintf(
                  $format, $key, trim($message));
              }
            }
          }
        }
        return $output;
}
```

15. Lastly, we define a mixed group of useful getters and setters:

```
        public function setMissingMessage($message)
        {
          $this->missingMessage = $message;
        }
        public function setSeparator($separator)
        {
          $this->separator = $separator;
        }
        public function getSeparator()
        {
          return $this->separator;
        }
        public function getAssignments()
        {
          return $this->assignments;
        }
        public function setAssignments(array
$assignments)
        {
          $this->assignments = $assignments;
        }
        // closing bracket for class AbstractFilter
}
```

16. Filtering and validation, although often performed together, are just as often performed separately. Accordingly, we define discrete classes for each. We'll start with `Application\Filter\Filter`. We make this class extend `AbstractFilter` in order to provide the core functionality described previously:

```
namespace Application\Filter;
class Filter extends AbstractFilter
{
  // code
}
```

17. Within this class we define a core `process()` method that scans an array of data and applies filters as per the array of assignments. If there are no assigned filters for this data set, we simply return `NULL`:

```
public function process(array $data)
{
  if (!(isset($this->assignments)
      && count($this->assignments))) {
        return NULL;
  }
```

18. Otherwise, we initialize `$this->results` to an array of `Result` objects where the `$item` property is the original value from `$data`, and the `$messages` property is an empty array:

```
foreach ($data as $key => $value) {
  $this->results[$key] = new Result($value,
array());
}
```

19. We then make a copy of `$this->assignments` and check to see if there are any *global* filters (identified by the '`*`' key. If so, we run `processGlobal()` and then unset the '`*`' key:

```
$toDo = $this->assignments;
if (isset($toDo['*'])) {
  $this->processGlobalAssignment($toDo['*'],
$data);
  unset($toDo['*']);
}
```

20. Finally, we loop through any remaining assignments, calling `processAssignment()`:

```
foreach ($toDo as $key => $assignment) {
  $this->processAssignment($assignment, $key);
}
```

21. As you will recall, each *assignment* is keyed to the data field, and represents an array of callbacks for that field. Thus, in `processGlobalAssignment()` we need to loop through the array of callbacks. In this case, however, because these assignments are *global*, we also need to loop through the *entire* data set, and apply each global filter in turn:

```
protected function
processGlobalAssignment($assignment, $data)
{
  foreach ($assignment as $callback) {
    if ($callback === NULL) continue;
    foreach ($data as $k => $value) {
      $result = $this-
>callbacks[$callback['key']]($this-
>results[$k]->item,
      $callback['params']);
      $this->results[$k]-
>mergeResults($result);
    }
  }
}
```

Note

The tricky bit is this line of code:

```
$result = $this-
>callbacks[$callback['key']]($this -
>results[$k]->item, $callback['params']);
```

Remember, each callback is actually an anonymous class that defines the PHP magic `__invoke()` method. The arguments supplied are the actual data item to be filtered, and an array of parameters. By running `$this->callbacks[$callback['key']]()` we are in fact magically calling `__invoke()`.

22. When we define `processAssignment()`, in a manner akin to `processGlobalAssignment()`, we need to execute each remaining callback assigned to each data key:

```
  protected function
processAssignment($assignment, $key)
  {
    foreach ($assignment as $callback) {
      if ($callback === NULL) continue;
      $result = $this-
>callbacks[$callback['key']]($this-
>results[$key]->item,

$callback['params']);
      $this->results[$key]-
>mergeResults($result);
    }
  }
}  // closing brace for
Application\Filter\Filter
```

> **Note**
>
> It is important that any filtering operation that alters the original user-supplied data should display a message indicating that a change was made. This can become part of an audit trail to safeguard you against potential legal liability when a change is made without user knowledge or consent.

How it works...

Create an `Application\Filter` folder. In this folder, create the following class files, using code from the preceding steps:

Application\Filter* class file	Code described in these steps

Application\Filter* class file	Code described in these steps
Result.php	3 - 5
CallbackInterface.php	6
Messages.php	7
AbstractFilter.php	8 - 15
Filter.php	16 - 22

Next, take the code discussed in step 5, and use it to configure an array of messages in a chap_06_post_data_config_messages.php file. Each callback references the Messages::$messages property. Here is a sample configuration:

```
<?php
use Application\Filter\Messages;
Messages::setMessages(
  [
    'length_too_short' => 'Length must be at least
%d',
    'length_too_long'  => 'Length must be no more
than %d',
    'required'         => 'Please be sure to enter
a value',
    'alnum'            => 'Only letters and
numbers allowed',
    'float'            => 'Only numbers or decimal
point',
    'email'            => 'Invalid email address',
```

```
    'in_array'          => 'Not found in the list',
    'trim'              => 'Item was trimmed',
    'strip_tags'        => 'Tags were removed from
this item',
    'filter_float'      => 'Converted to a decimal
number',
    'phone'             => 'Phone number is [+n]
nnn-nnn-nnnn',
    'test'              => 'TEST',
    'filter_length'     => 'Reduced to specified
length',
  ]
);
```

Next, create a `chap_06_post_data_config_callbacks.php` callback configuration file that contains configuration for filtering callbacks, as described in step 4. Each callback should follow this generic template:

```
'callback_key' => new class () implements
CallbackInterface
{
  public function __invoke($item, $params) :
Result
  {
    $changed  = array();
    $filtered = /* perform filtering operation on
$item */
    if ($filtered !== $item) $changed =
Messages::$messages['callback_key'];
    return new Result($filtered, $changed);
  }
}
```

The callbacks themselves must implement the interface and return a `Result` instance. We can take advantage of the PHP 7 **anonymous class** capability by having our callbacks return an anonymous class that implements `CallbackInterface`. Here is how an array of filtering callbacks might look:

```
use Application\Filter\ { Result, Messages,
CallbackInterface };
$config = [ 'filters' => [
```

```
  'trim' => new class () implements
CallbackInterface
  {
    public function __invoke($item, $params) :
Result
    {
      $changed  = array();
      $filtered = trim($item);
      if ($filtered !== $item)
      $changed = Messages::$messages['trim'];
      return new Result($filtered, $changed);
    }
  },
  'strip_tags' => new class ()
  implements CallbackInterface
  {
    public function __invoke($item, $params) :
Result
    {
      $changed  = array();
      $filtered = strip_tags($item);
      if ($filtered !== $item)
      $changed =
Messages::$messages['strip_tags'];
      return new Result($filtered, $changed);
    }
  },
  // etc.
]
];
```

For test purposes, we will use the prospects table as a target. Instead of providing data from `$_POST`, we will construct an array of *good* and *bad* data:

Field name	Type	Allow nulls?
id	int(11)	No
first_name	varchar(128)	No
last_name	varchar(128)	No
address	varchar(256)	Yes
city	varchar(64)	Yes
state_province	varchar(32)	Yes
postal_code	char(16)	No
phone	varchar(16)	No
country	char(2)	No
email	varchar(250)	No
status	char(8)	Yes
budget	decimal(10,2)	Yes
last_updated	datetime	Yes

You can now create a `chap_06_post_data_filtering.php` script that sets up autoloading, includes the messages and callbacks configuration files:

```
<?php
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
include __DIR__ .
'/chap_06_post_data_config_messages.php';
include __DIR__ .
'/chap_06_post_data_config_callbacks.php';
```

You then need to define *assignments* that represent a mapping between the data fields and filter callbacks. Use the `*` key to define a *global* filter that applies to all data:

```
$assignments = [
  '*'    => [ ['key' => 'trim', 'params' => []],
              ['key' => 'strip_tags', 'params' => []]
],
  'first_name'  => [ ['key' => 'length',
   'params' => ['length' => 128]] ],
```

```
  'last_name'  => [ ['key' => 'length',
   'params' => ['length' => 128]] ],
  'city'          => [ ['key' => 'length',
   'params' => ['length' => 64]] ],
  'budget'     => [ ['key' => 'filter_float',
'params' => []] ],
];
```

Next, define *good* and *bad* test data:

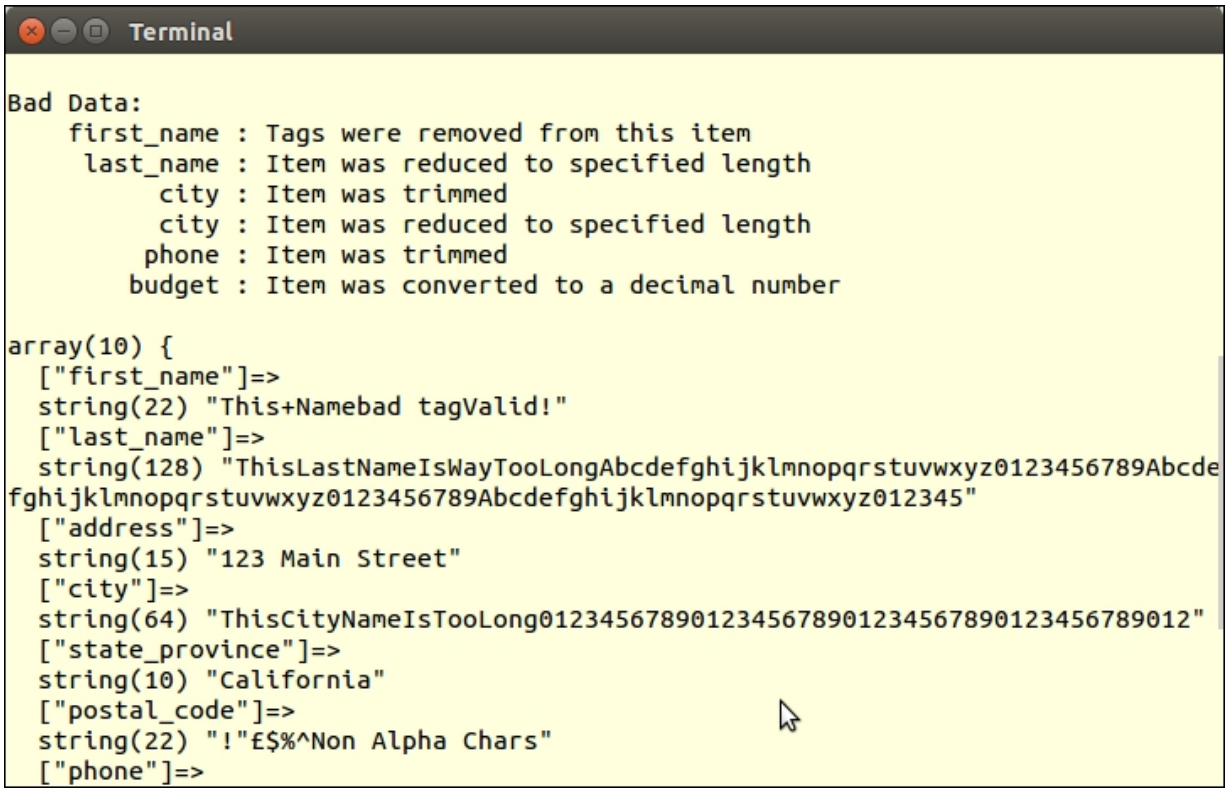
```
$goodData = [
  'first_name'      => 'Your Full',
  'last_name'       => 'Name',
  'address'         => '123 Main Street',
  'city'            => 'San Francisco',
  'state_province'  => 'California',
  'postal_code'     => '94101',
  'phone'           => '+1 415-555-1212',
  'country'         => 'US',
  'email'           => 'your@email.address.com',
  'budget'          => '123.45',
];
$badData = [
  'first_name'      => 'This+Name<script>bad
tag</script>Valid!',
  'last_name'       =>
'ThisLastNameIsWayTooLongAbcdefghijklmnopqrstuvwxy
z0123456789Abcdefghijklmnopqrstuvwxyz0123456789Abc
defghijklmnopqrstuvwxyz0123456789Abcdefghijklmnopq
rstuvwxyz0123456789',
  //'address'       => '',    // missing
  'city'            => '
ThisCityNameIsTooLong012345678901234567890123456678
901234567890123456789012345678  ',
  //'state_province'=> '',    // missing
  'postal_code'     => '!"£$%^Non Alpha Chars',
  'phone'           => ' 12345 ',
  'country'         => 'XX',
  'email'           => 'this.is@not@an.email',
  'budget'          => 'XXX',
];
```

Finally, you can create an `Application\Filter\Filter` instance, and test the data:

```
$filter = new Application\Filter\Filter(
$config['filters'], $assignments);
$filter->setSeparator(PHP_EOL);
  $filter->process($goodData);
echo $filter->getMessageString();
  var_dump($filter->getItemsAsArray());

$filter->process($badData);
echo $filter->getMessageString();
var_dump($filter->getItemsAsArray());
```

Processing *good* data produces no messages other than one indicating that the value for the *float* field was converted from string to `float`. The *bad* data, on the other hand, produces the following output:



You will also notice that tags were removed from `first_name`, and that both `last_name` and `city` were truncated.

There's more...

The `filter_input_array()` function takes two arguments: the input source (in the form of a pre-defined constant used to indicate one of the `$_*` PHP super-globals, that is, `$_POST`), and an array of matching field definitions as keys and filters or validators as values. This function performs not only filtering operations, but validation as well. The flags labeled *sanitize* are actually filters.

See also

Documentation and examples of `filter_input_array()` can be found at http://php.net/manual/en/function.filter-input-array.php. You might also have a look at the different types of *filters* that are available on http://php.net/manual/en/filter.filters.php.

Chaining $_POST validators

The *heavy lifting* for this recipe has already been accomplished in the preceding recipe. Core functionality is defined by `Application\Filter\AbstractFilter`. The actual validation is performed by an array of validating callbacks.

How to do it...

1. Look over the preceding recipe, *Chaining $_POST filters*. We will be using all of the classes and configuration files in this recipe, except where noted here.
2. To begin, we define a configuration array of validation callbacks. As with the preceding recipe, each callback should implement `Application\Filter\CallbackInterface`, and should return an instance of `Application\Filter\Result`. Validators would take this generic form:

```
use Application\Filter\ { Result, Messages,
CallbackInterface };
$config = [
  // validator callbacks
  'validators' => [
    'key' => new class () implements
CallbackInterface
    {
      public function __invoke($item, $params)
: Result
      {
        // validation logic goes here
        return new Result($valid, $error);
      }
    },
    // etc.
```

3. Next, we define a `Application\Filter\Validator` class, which loops through the array of assignments, testing each data item against its assigned validator callbacks. We make this class

extend `AbstractFilter` in order to provide the core functionality described previously:

```
namespace Application\Filter;
class Validator extends AbstractFilter
{
  // code
}
```

4. Within this class, we define a core `process()` method that scans an array of data and applies validators as per the array of assignments. If there are no assigned validators for this data set, we simply return the current status of `$valid` (which is `TRUE`):

```
public function process(array $data)
{
  $valid = TRUE;
  if (!(isset($this->assignments)
      && count($this->assignments))) {
        return $valid;
  }
```

5. Otherwise, we initialize `$this->results` to an array of `Result` objects where the `$item` property is set to `TRUE`, and the `$messages` property is an empty array:

```
foreach ($data as $key => $value) {
  $this->results[$key] = new Result(TRUE,
array());
}
```

6. We then make a copy of `$this->assignments` and check to see if there are any *global* filters (identified by the '`*`' key). If so, we run `processGlobal()` and then unset the '`*`' key:

```
$toDo = $this->assignments;
if (isset($toDo['*'])) {
  $this->processGlobalAssignment($toDo['*'],
$data);
  unset($toDo['*']);
}
```

7. Finally, we loop through any remaining assignments, calling `processAssignment()`. This is an ideal place to check to see if

any fields present in the assignments array is missing from the data. Note that we set `$valid` to `FALSE` if any validation callback returns `FALSE`:

```
foreach ($toDo as $key => $assignment) {
  if (!isset($data[$key])) {
      $this->results[$key] =
      new Result(FALSE, $this-
>missingMessage);
  } else {
      $this->processAssignment(
        $assignment, $key, $data[$key]);
  }
  if (!$this->results[$key]->item) $valid =
FALSE;
  }
  return $valid;
}
```

8. As you will recall, each *assignment* is keyed to the data field, and represents an array of callbacks for that field. Thus, in `processGlobalAssignment()`, we need to loop through the array of callbacks. In this case, however, because these assignments are *global*, we also need to loop through the *entire* data set, and apply each global filter in turn.
9. In contrast to the equivalent `Application\Filter\Fiter::processGlobalAssignment()` method, we need to call `mergeValidationResults()`. The reason for this is that if the value of `$result->item` is already `FALSE`, we need to ensure that it does not subsequently get overwritten by a value of `TRUE`. Any validator in the chain that returns `FALSE` must overwrite any other validation result:

```
protected function
processGlobalAssignment($assignment, $data)
{
  foreach ($assignment as $callback) {
    if ($callback === NULL) continue;
    foreach ($data as $k => $value) {
      $result = $this-
>callbacks[$callback['key']]
      ($value, $callback['params']);
      $this->results[$k]-
```

```
>mergeValidationResults($result);
    }
  }
}
```

10. When we define `processAssignment()`, in a manner akin to `processGlobalAssignment()`, we need to execute each remaining callback assigned to each data key, again calling `mergeValidationResults()`:

```
protected function
processAssignment($assignment, $key, $value)
{
  foreach ($assignment as $callback) {
    if ($callback === NULL) continue;
        $result = $this-
>callbacks[$callback['key']]
       ($value, $callback['params']);
        $this->results[$key]-
>mergeValidationResults($result);
    }
  }
```

How it works...

As with the preceding recipe, be sure to define the following classes:

- `Application\Filter\Result`
- `Application\Filter\CallbackInterface`
- `Application\Filter\Messages`
- `Application\Filter\AbstractFilter`

You can use the `chap_06_post_data_config_messages.php` file, also described in the previous recipe.

Next, create a `Validator.php` file in the `Application\Filter` folder. Place the code described in step 3 to 10.

Next, create a `chap_06_post_data_config_callbacks.php` callback configuration file that contains configurations for validation callbacks,

as described in step 2. Each callback should follow this generic template:

```
'validation_key' => new class () implements
CallbackInterface
{
  public function __invoke($item, $params) :
Result
  {
    $error = array();
    $valid = /* perform validation operation on
$item */
    if (!$valid)
    $error[] =
Messages::$messages['validation_key'];
    return new Result($valid, $error);
  }
}
```

Now you can create a `chap_06_post_data_validation.php` calling script that initializes autoloading and includes the configuration scripts:

```
<?php
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
include __DIR__ .
'/chap_06_post_data_config_messages.php';
include __DIR__ .
'/chap_06_post_data_config_callbacks.php';
```

Next, define an array of assignments, mapping data fields to validator callback keys:

```
$assignments = [
  'first_name'       => [ ['key' => 'length',
  'params'   => ['min' => 1, 'max' => 128]],
                ['key' => 'alnum',
  'params'   => ['allowWhiteSpace' => TRUE]],
                ['key'   => 'required','params' =>
[]] ],
```

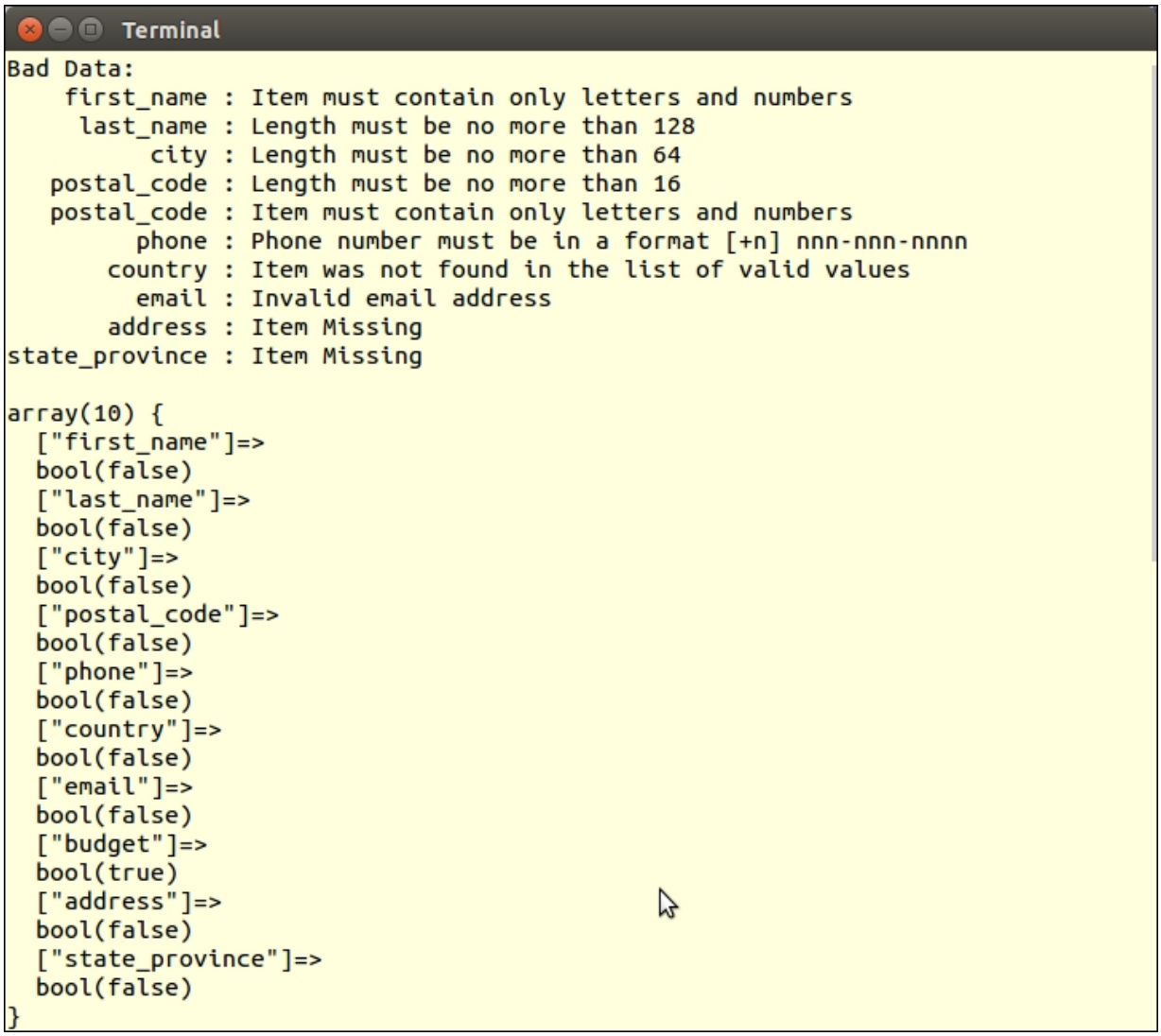
```
    'last_name'=> [ ['key' => 'length',
    'params'   => ['min'   => 1, 'max' => 128]],
                  ['key'   => 'alnum',
    'params'   => ['allowWhiteSpace' => TRUE]],
                  ['key'   => 'required','params' =>
[]] ],
    'address'       => [ ['key' => 'length',
    'params'        => ['max' => 256]] ],
    'city'          => [ ['key' => 'length',
    'params'        => ['min' => 1, 'max' => 64]] ],
    'state_province'=> [ ['key' => 'length',
    'params'        => ['min' => 1, 'max' => 32]] ],
    'postal_code'   => [ ['key' => 'length',
    'params'        => ['min' => 1, 'max' => 16] ],
                       ['key' => 'alnum',
    'params'        => ['allowWhiteSpace' => TRUE]],
                       ['key' => 'required','params'
=> [] ],
    'phone'         => [ ['key' => 'phone', 'params'
=> [] ],
    'country'       => [ ['key' => 'in_array',
    'params'        => $countries ],
                       ['key' => 'required','params'
=> [] ],
    'email'         => [ ['key' => 'email', 'params'
=> [] ],
                       ['key' => 'length',
    'params'        => ['max' => 250] ],
                       ['key' => 'required','params'
=> [] ] ],
    'budget'        => [ ['key' => 'float', 'params'
=> [] ]
];
```

For test data, use the same *good* and *bad* data defined in the `chap_06_post_data_filtering.php` file described in the previous recipe. After that, you are in a position to create an `Application\Filter\Validator` instance, and test the data:

```
$validator = new
Application\Filter\Validator($config['validators']
, $assignments);
$validator->setSeparator(PHP_EOL);
$validator->process($badData);
```

```
echo $validator->getMessageString(40, '%14s :
%-26s' . PHP_EOL);
var_dump($validator->getItemsAsArray());
$validator->process($goodData);
echo $validator->getMessageString(40, '%14s :
%-26s' . PHP_EOL);
var_dump($validator->getItemsAsArray());
```

As expected, the *good* data does not produce any validation errors. The *bad* data, on the other hand, generates the following output:



Notice that the *missing* fields, `address` and `state_province` validate `FALSE`, and return the missing item message.

Tying validation to a form

When a form is first rendered, there is little value in having a form class (such as `Application\Form\Factory`, described in the previous recipe) tied to a class that can perform filtering or validation (such as the `Application\Filter\*` described in the previous recipe). Once the form data has been submitted, however, interest grows. If the form data fails validation, the values can be filtered, and then re-displayed. Validation error messages can be tied to form elements, and rendered next to form fields.

How to do it...

1. First of all, be sure to implement the classes defined in the *Implementing a Form Factory*, *Chaining $_POST Filters*, and *Chaining $_POST Validators* recipes.
2. We will now turn our attention to the `Application\Form\Factory` class, and add properties and setters that allow us to attach instances of `Application\Filter\Filter` and `Application\Filter\Validator`. We also need define `$data`, which will be used to retain the filtered and/or validated data:

```
const DATA_NOT_FOUND = 'Data not found. Run
setData()';
const FILTER_NOT_FOUND = 'Filter not found.
Run setFilter()';
const VALIDATOR_NOT_FOUND = 'Validator not
found. Run setValidator()';

protected $filter;
protected $validator;
protected $data;

public function setFilter(Filter $filter)
{
  $this->filter = $filter;
}

public function setValidator(Validator
$validator)
```

```
{
  $this->validator = $validator;
}

public function setData($data)
{
  $this->data = $data;
}
```

3. Next, we define a `validate()` method that calls the `process()` method of the embedded `Application\Filter\Validator` instance. We check to see if `$data` and `$validator` exist. If not, the appropriate exceptions are thrown with instructions on which method needs to be run first:

```
public function validate()
{
  if (!$this->data)
  throw new
RuntimeException(self::DATA_NOT_FOUND);

  if (!$this->validator)
  throw new
RuntimeException(self::VALIDATOR_NOT_FOUND);
```

4. After calling the `process()` method, we associate validation result messages with form element messages. Note that the `process()` method returns a boolean value that represents the overall validation status of the data set. When the form is re-displayed following failed validation, error messages will appear next to each element:

```
$valid = $this->validator->process($this-
>data);

foreach ($this->elements as $element) {
  if (isset($this->validator->getResults()
      [$element->getName()])) {
        $element->setErrors($this->validator-
>getResults()
        [$element->getName()]->messages);
      }
    }
    return $valid;
  }
```

5. In a similar manner, we define a `filter()` method that calls the `process()` method of the embedded `Application\Filter\Filter` instance. As with the `validate()` method described in step 3, we need to check for the existence of `$data` and `$filter`. If either is missing, we throw a `RuntimeException` with the appropriate message:

```
public function filter()
{
  if (!$this->data)
  throw new
RuntimeException(self::DATA_NOT_FOUND);

  if (!$this->filter)
  throw new
RuntimeException(self::FILTER_NOT_FOUND);
```

6. We then run the `process()` method, which produces an array of `Result` objects where the `$item` property represents the end result of the filter chain. We then loop through the results, and, if the corresponding `$element` key matches, set the `value` attribute to the filtered value. We also add any messages resulting from the filtering process. When the form is then re-displayed, all value attributes will display filtered results:

```
$this->filter->process($this->data);
foreach ($this->filter->getResults() as $key
=> $result) {
  if (isset($this->elements[$key])) {
    $this->elements[$key]
    ->setSingleAttribute('value', $result-
>item);
    if (isset($result->messages)
        && count($result->messages)) {
      foreach ($result->messages as $message)
{
        $this->elements[$key]-
>addSingleError($message);
      }
    }
  }
}
}
```

How it works...

You can start by making the changes to `Application\Form\Factory` as described above. For a test target you can use the prospects database table shown in the *How it works...* section of the *Chaining $_POST filters* recipe. The various column settings should give you an idea of which form elements, filters, and validators to define.

As an example, you can define a `chap_06_tying_filters_to_form_definitions.php` file, which will contain definitions for form wrappers, elements, and filter assignments. Here are some examples:

```
<?php
use Application\Form\Generic;

define('VALIDATE_SUCCESS', 'SUCCESS: form
submitted ok!');
define('VALIDATE_FAILURE', 'ERROR: validation
errors detected');

$wrappers = [
  Generic::INPUT  => ['type' => 'td', 'class' =>
'content'],
  Generic::LABEL  => ['type' => 'th', 'class' =>
'label'],
  Generic::ERRORS => ['type' => 'td', 'class' =>
'error']
];

$elements = [
  'first_name' => [
     'class'     => 'Application\Form\Generic',
     'type'      => Generic::TYPE_TEXT,
     'label'     => 'First Name',
     'wrappers'  => $wrappers,
     'attributes'=>
['maxLength'=>128,'required'=>'']
  ],
  'last_name'   => [
    'class'     => 'Application\Form\Generic',
    'type'      => Generic::TYPE_TEXT,
    'label'     => 'Last Name',
```

```
    'wrappers'  => $wrappers,
    'attributes'=>
['maxLength'=>128,'required'=>'']
  ],
    // etc.
];

// overall form config
$formConfig = [
  'name'       => 'prospectsForm',
  'attributes' => [
'method'=>'post',
'action'=>'chap_06_tying_filters_to_form.php'
],
  'row_wrapper'  => ['type' => 'tr', 'class' =>
'row'],
  'form_wrapper' => [
    'type'=>'table',
    'class'=>'table',
    'id'=>'prospectsTable',
    'class'=>'display','cellspacing'=>'0'
  ],
  'form_tag_inside_wrapper' => FALSE,
];

$assignments = [
  'first_name'    => [ ['key' => 'length',
  'params'        => ['min' => 1, 'max' => 128]],
                     ['key' => 'alnum',
  'params'        => ['allowWhiteSpace' => TRUE]],
                     ['key' => 'required','params'
=> []] ],
  'last_name'     => [ ['key' => 'length',
  'params'        => ['min' => 1, 'max' => 128]],
                     ['key' => 'alnum',
  'params'        => ['allowWhiteSpace' => TRUE]],
                     ['key' => 'required','params'
=> []] ],
  'address'       => [ ['key' => 'length',
  'params'        => ['max' => 256]] ],
  'city'          => [ ['key' => 'length',
  'params'        => ['min' => 1, 'max' => 64]] ],
  'state_province'=> [ ['key' => 'length',
  'params'        => ['min' => 1, 'max' => 32]] ],
  'postal_code'   => [ ['key' => 'length',
  'params'        => ['min' => 1, 'max' => 16] ],
                     ['key' => 'alnum',
  'params'        => ['allowWhiteSpace' => TRUE]],
```

```
                    ['key' => 'required','params'
=> [] ],
  'phone'          => [ ['key' => 'phone',
'params' => [] ],
  'country'        => [ ['key' => 'in_array',
  'params'         => $countries ],
                    ['key' => 'required','params'
=> [] ],
  'email'          => [ ['key' => 'email',
'params' => [] ],
                    ['key' => 'length',
  'params'         => ['max' => 250] ],
                    ['key' => 'required','params'
=> [] ] ],
  'budget'         => [ ['key' => 'float',
'params' => [] ]
];
```

You can use the already existing `chap_06_post_data_config_callbacks.php` and `chap_06_post_data_config_messages.php` files described in the previous recipes. Finally, define a `chap_06_tying_filters_to_form.php` file that sets up autoloading and includes these three configuration files:

```
<?php
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
include __DIR__ .
'/chap_06_post_data_config_messages.php';
include __DIR__ .
'/chap_06_post_data_config_callbacks.php';
include __DIR__ .
'/chap_06_tying_filters_to_form_definitions.php';
```

Next, you can create instances of the form factory, filter, and validator classes:

```
use Application\Form\Factory;
use Application\Filter\ { Validator, Filter };
```

```
$form = Factory::generate($elements);
$form->setFilter(new Filter($callbacks['filters'],
$assignments['filters']));
$form->setValidator(new
Validator($callbacks['validators'],
$assignments['validators']));
```

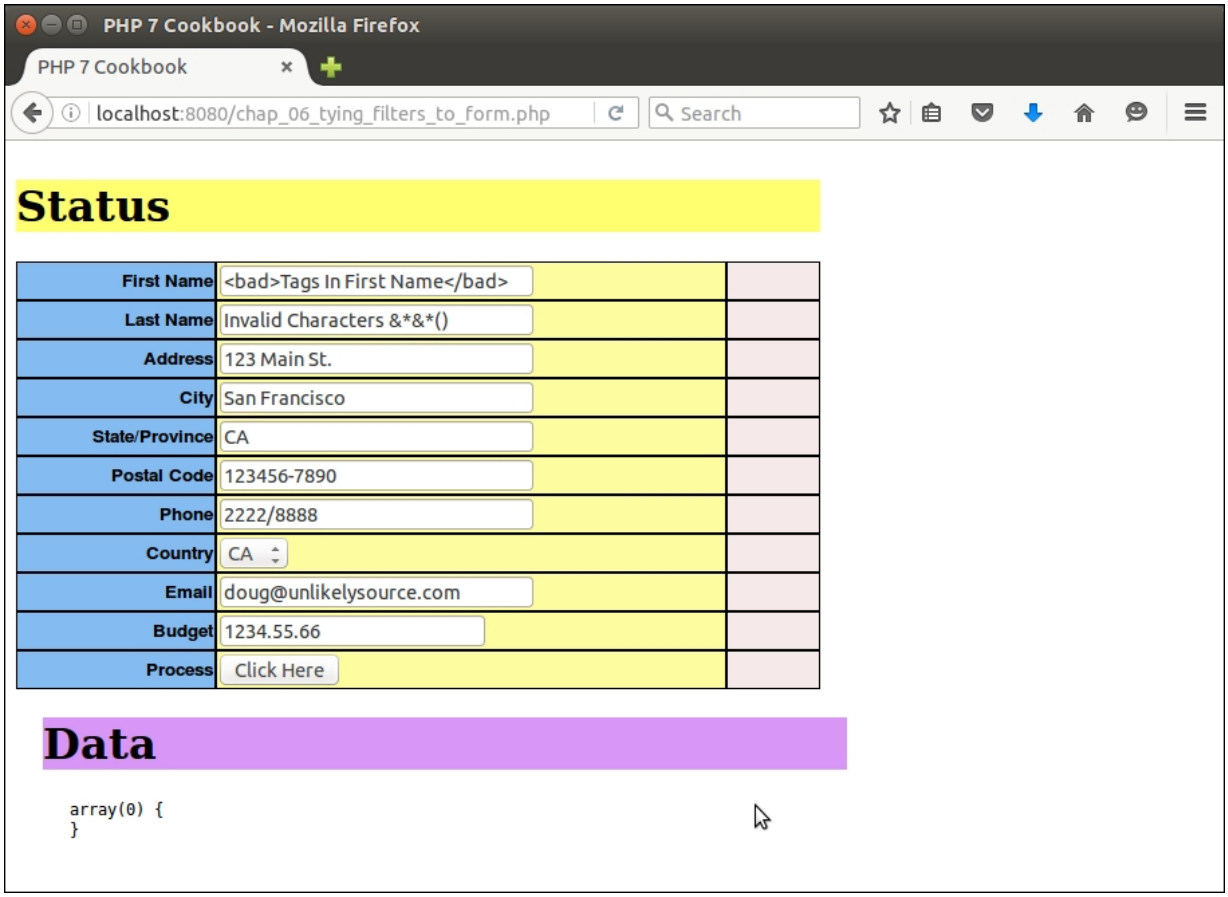
You can then check to see if there is any `$_POST` data. If so, perform validation and filtering:

```
$message = '';
if (isset($_POST['submit'])) {
  $form->setData($_POST);
  if ($form->validate()) {
    $message = VALIDATE_SUCCESS;
  } else {
    $message = VALIDATE_FAILURE;
  }
  $form->filter();
}
?>
```

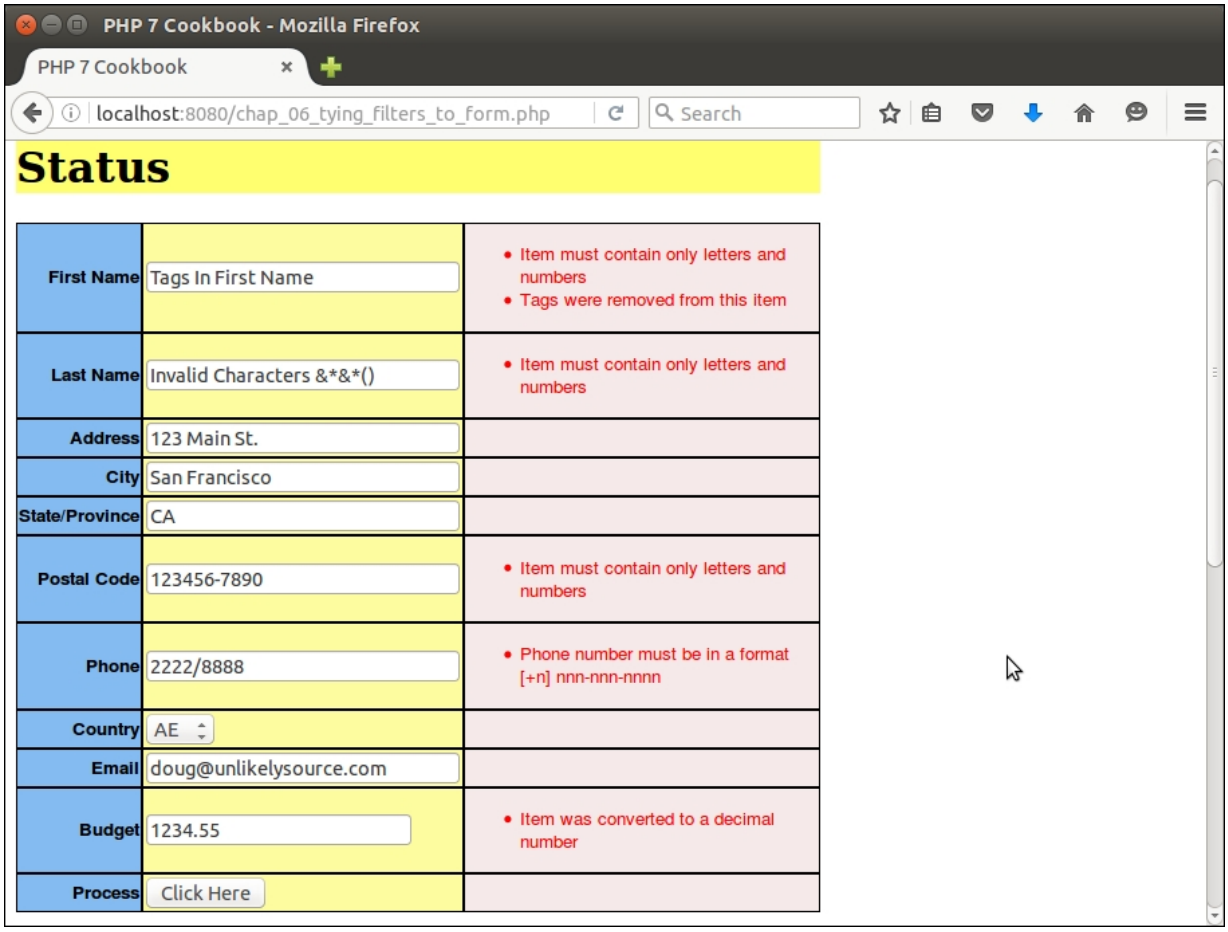
The view logic is extremely simple: just render the form. Any validation messages and values for the various elements will be assigned as part of validation and filtering:

```
<?= $form->render($form, $formConfig); ?>
```

Here is an example using bad form data:



Notice the filtering and validation messages. Also notice the bad tags:


PHP 7 Cookbook - Mozilla Firefox
PHP 7 Cookbook
localhost:8080/chap_06_tying_filters_to_form.php
Search
Status
First Name
Tags In First Name
Item must contain only letters and numbers
Tags were removed from this item
Last Name
Invalid Characters &*&*()
Item must contain only letters and numbers
Address
123 Main St.
City
San Francisco
State/Province
CA
Postal Code
123456-7890
Item must contain only letters and numbers
Phone
2222/8888
Phone number must be in a format [+n] nnn-nnn-nnnn
Country
AE
Email
doug@unlikelysource.com
Budget
1234.55
Item was converted to a decimal number
Process
Click Here

Chapter 7. Accessing Web Services

In this chapter, we will cover the following topics:

- Converting between PHP and XML
- Creating a simple REST client
- Creating a simple REST server
- Creating a simple SOAP client
- Creating a simple SOAP server

Introduction

Making background queries to external web services is becoming an ever-increasing part of any PHP web practice. The ability to provide appropriate, timely, and plentiful data means more business for your customers and the websites you develop. We start with a couple of recipes aimed at data conversion between **eXtensible Markup Language** (**XML**) and native PHP. Next, we show you how to implement a simple **Representational State Transfer** (**REST**) client and server. After that, we turn our attention to **SOAP** clients and servers.

Converting between PHP and XML

When considering a conversion between PHP native data types and XML, we would normally consider an array as the primary target. With this in mind, the process of converting from a PHP array to XML differs radically from the approach needed to do the reverse.

> **Note**
>
> Objects could also be considered for conversion; however, it is difficult to render object methods in XML. Properties can be represented, however, by using the `get_object_vars()` function, which reads object properties into an array.

How to do it...

1. First, we define an `Application\Parse\ConvertXml` class. This class will holdthe methods that will convert from XML to a PHP array, and vice versa. We will need both the `SimpleXMLElement` and `SimpleXMLIterator` classes from the SPL:

```
namespace Application\Parse;
use SimpleXMLIterator;
use SimpleXMLElement;
class ConvertXml
{
}
```

2. Next, we define a `xmlToArray()` method that will accept a `SimpleXMLIterator` instance as an argument. It will be called recursively and will produce a PHP array from an XML document. We take advantage of the `SimpleXMLIterator` ability

to advance through the XML document, using the `key()`, `current()`, `next()`, and `rewind()` methods to navigate:

```
public function xmlToArray(SimpleXMLIterator
$xml) : array
{
  $a = array();
  for( $xml->rewind(); $xml->valid(); $xml-
>next() ) {
    if(!array_key_exists($xml->key(), $a)) {
      $a[$xml->key()] = array();
    }
    if($xml->hasChildren()){
      $a[$xml->key()][] = $this-
>xmlToArray($xml->current());
    }
    else{
      $a[$xml->key()] = (array) $xml-
>current()->attributes();
      $a[$xml->key()]['value'] = strval($xml-
>current());
    }
  }
  return $a;
}
```

3. For the reverse process, also called recursively, we define two methods. The first method, `arrayToXml()`, sets up an initial `SimpleXMLElement` instance, and then calls the second method, `phpToXml()`:

```
public function arrayToXml(array $a)
{
  $xml = new SimpleXMLElement(
  '<?xml version="1.0" standalone="yes"?>
<root></root>');
  $this->phpToXml($a, $xml);
  return $xml->asXML();
}
```

4. Note that in the second method, we use `get_object_vars()` in case one of the array elements is an object. You'll also note that numbers alone are not allowed as XML tags, which means adding some text in front of the number:

```
protected function phpToXml($value, &$xml)
{
  $node = $value;
  if (is_object($node)) {
    $node = get_object_vars($node);
  }
  if (is_array($node)) {
    foreach ($node as $k => $v) {
      if (is_numeric($k)) {
        $k = 'number' . $k;
      }
      if (is_array($v)) {
          $newNode = $xml->addChild($k);
          $this->phpToXml($v, $newNode);
      } elseif (is_object($v)) {
          $newNode = $xml->addChild($k);
          $this->phpToXml($v, $newNode);
      } else {
          $xml->addChild($k, $v);
      }
    }
  } else  {
      $xml->addChild(self::UNKNOWN_KEY,
$node);
  }
}
```

How it works...

As a sample XML document, you can use the **Web Services Definition Language** (**WSDL**) for the United States National Weather Service. This is an XML document that describes a SOAP service, and can be found at http://graphical.weather.gov/xml/SOAP_server/ndfdXMLserver.php?wsdl.

We will use the `SimpleXMLIterator` class to provide an iteration mechanism. You can then configure autoloading, and get an instance of `Application\Parse\ConvertXml`, using `xmlToArray()` to convert the WSDL to a PHP array:

```
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
use Application\Parse\ConvertXml;
$wsdl = 'http://graphical.weather.gov/xml/'
. 'SOAP_server/ndfdXMLserver.php?wsdl';
$xml = new SimpleXMLIterator($wsdl, 0, TRUE);
$convert = new ConvertXml();
var_dump($convert->xmlToArray($xml));
```

The resulting array is shown here:

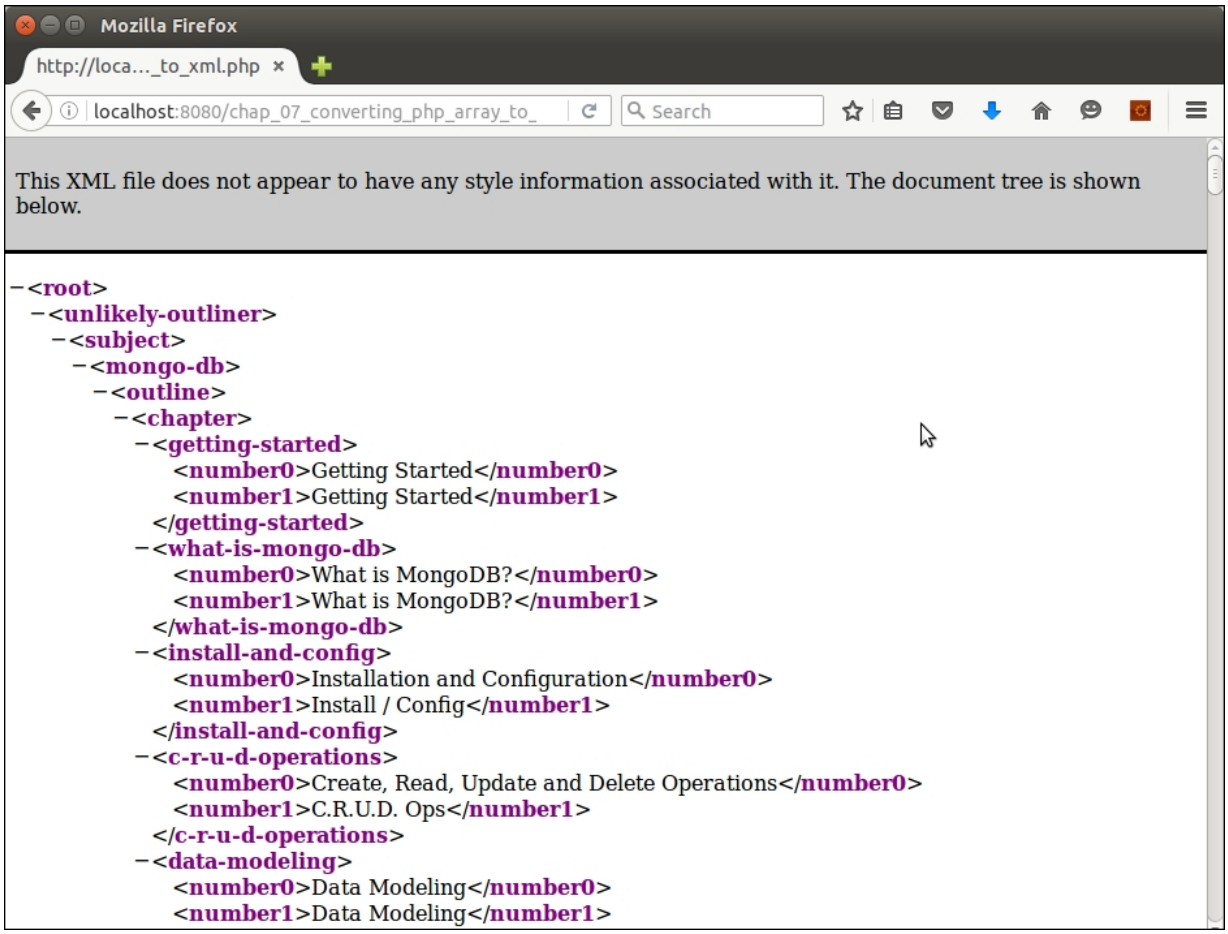


To do the reverse, use the `arrayToXml()` method described in this recipe. As a source document, you can use a

`source/data/mongo.db.global.php` file that contains an outline for a training video on MongoDB available through O'Reilly Media (disclaimer: by this author!). Using the same autoloader configuration and instance of `Application\Parse\ConvertXml`, here is the sample code you could use:

```
$convert = new ConvertXml();
header('Content-Type: text/xml');
echo $convert->arrayToXml(include CONFIG_FILE);
```

Here is the output in a browser:



Creating a simple REST client

REST clients use **HyperText Transfer Protocol** (**HTTP**) to generate requests to external web services. By changing the HTTP method, we can cause the external service to perform different operations. Although there are quite a few methods (or verbs) available, we will only focus on `GET` and `POST`. In this recipe, we will use the **Adapter** software design pattern to present two different ways of implementing a REST client.

How to do it...

1. Before we can define REST client adapters, we need to define common classes to represent request and response information. First, we will start with an abstract class that has methods and properties needed for either a request or response:

```
namespace Application\Web;

class AbstractHttp
{
```

2. Next, we define class constants that represent HTTP information:

```
const METHOD_GET = 'GET';
const METHOD_POST = 'POST';
const METHOD_PUT = 'PUT';
const METHOD_DELETE = 'DELETE';
const CONTENT_TYPE_HTML = 'text/html';
const CONTENT_TYPE_JSON = 'application/json';
const CONTENT_TYPE_FORM_URL_ENCODED =
  'application/x-www-form-urlencoded';
const HEADER_CONTENT_TYPE = 'Content-Type';
const TRANSPORT_HTTP = 'http';
const TRANSPORT_HTTPS = 'https';
const STATUS_200 = '200';
const STATUS_401 = '401';
const STATUS_500 = '500';
```

3. We then define properties that are needed for either a request or a response:

```
protected $uri;      // i.e.
http://xxx.com/yyy
protected $method;    // i.e. GET, PUT, POST,
DELETE
protected $headers;  // HTTP headers
protected $cookies;  // cookies
protected $metaData;  // information about the
transmission
protected $transport;  // i.e. http or https
protected $data = array();
```

4. It logically follows to define getters and setters for these properties:

```
public function setMethod($method)
{
  $this->method = $method;
}
public function getMethod()
{
  return $this->method ?? self::METHOD_GET;
}
// etc.
```

5. Some properties require access by key. For this purpose, we define `getXxxByKey()` and `setXxxByKey()` methods:

```
public function setHeaderByKey($key, $value)
{
  $this->headers[$key] = $value;
}
public function getHeaderByKey($key)
{
  return $this->headers[$key] ?? NULL;
}
public function getDataByKey($key)
{
  return $this->data[$key] ?? NULL;
}
public function getMetaDataByKey($key)
{
  return $this->metaData[$key] ?? NULL;
}
```

6. In some cases, the request will require parameters. We will assume that the parameters will be in the form of a PHP array stored in the `$data` property. We can then build the request URL using the `http_build_query()` function:

```
public function setUri($uri, array $params =
NULL)
{
  $this->uri = $uri;
  $first = TRUE;
  if ($params) {
    $this->uri .= '?' .
http_build_query($params);
  }
}
public function getDataEncoded()
{
  return http_build_query($this->getData());
}
```

7. Finally, we set `$transport` based on the original request:

```
public function setTransport($transport =
NULL)
{
  if ($transport) {
      $this->transport = $transport;
  } else {
      if (substr($this->uri, 0, 5) ==
self::TRANSPORT_HTTPS) {
          $this->transport =
self::TRANSPORT_HTTPS;
      } else {
          $this->transport =
self::TRANSPORT_HTTP;
      }
    }
  }
```

8. In this recipe, we will define a `Application\Web\Request` class that can accept parameters when we wish to generate a request, or, alternatively, populate properties with incoming request information when implementing a server that accepts requests:

```
namespace Application\Web;
class Request extends AbstractHttp
{
  public function __construct(
    $uri = NULL, $method = NULL, array
$headers = NULL,
    array $data = NULL, array $cookies = NULL)
    {
      if (!$headers) $this->headers = $_SERVER
?? array();
      else $this->headers = $headers;
      if (!$uri) $this->uri = $this-
>headers['PHP_SELF'] ?? '';
      else $this->uri = $uri;
      if (!$method) $this->method =
        $this->headers['REQUEST_METHOD'] ??
self::METHOD_GET;
      else $this->method = $method;
      if (!$data) $this->data = $_REQUEST ??
array();
      else $this->data = $data;
      if (!$cookies) $this->cookies = $_COOKIE
?? array();
      else $this->cookies = $cookies;
      $this->setTransport();
    }
}
```

9. Now we can turn our attention to a response class. In this case, we will define an `Application\Web\Received` class. The name reflects the fact that we are re-packaging data received from the external web service:

```
namespace Application\Web;
class Received extends AbstractHttp
{
  public function __construct(
    $uri = NULL, $method = NULL, array
$headers = NULL,
    array $data = NULL, array $cookies = NULL)
  {
    $this->uri = $uri;
    $this->method = $method;
    $this->headers = $headers;
    $this->data = $data;
    $this->cookies = $cookies;
```

```
            $this->setTransport();
        }
    }
```

Creating a streams-based REST client

We are now ready to consider two different ways to implement a REST client. The first approach is to use an underlying PHP I/O layer referred to as **Streams**. This layer provides a series of wrappers that provide access to external streaming resources. By default, any of the PHP file commands will use the file wrapper, which gives access to the local filesystem. We will use the `http://` or `https://` wrappers to implement the `Application\Web\Client\Streams` adapter:

1. First, we define a `Application\Web\Client\Streams` class:

```
namespace Application\Web\Client;
use Application\Web\ { Request, Received };
class Streams
{
  const BYTES_TO_READ = 4096;
```

2. Next, we define a method to send the request to the external web service. In the case of `GET`, we add the parameters to the URI. In the case of `POST`, we create a stream context that contains metadata instructing the remote service that we are supplying data. Using PHP Streams, making a request is just a matter of composing the URI, and, in the case of `POST`, setting the stream context. We then use a simple `fopen()`:

```
public static function send(Request $request)
{
  $data = $request->getDataEncoded();
  $received = new Received();
  switch ($request->getMethod()) {
    case Request::METHOD_GET :
      if ($data) {
        $request->setUri($request->getUri() .
'?' . $data);
      }
      $resource = fopen($request->getUri(),
```

```
'r');
      break;
    case Request::METHOD_POST :
      $opts = [
        $request->getTransport() =>
        [
          'method'  => Request::METHOD_POST,
          'header'  =>
Request::HEADER_CONTENT_TYPE
          . ': ' .
Request::CONTENT_TYPE_FORM_URL_ENCODED,
          'content' => $data
        ]
      ];
      $resource = fopen($request->getUri(),
'w',
      stream_context_create($opts));
      break;
    }
    return self::getResults($received,
$resource);
}
```

3. Finally, we have a look at retrieving and packaging results into a `Received` object. You will notice that we added a provision to decode data received in JSON format:

```
protected static function getResults(Received
$received, $resource)
{
  $received-
>setMetaData(stream_get_meta_data($resource));
  $data = $received-
>getMetaDataByKey('wrapper_data');
  if (!empty($data) && is_array($data)) {
    foreach($data as $item) {
      if (preg_match('!^HTTP/\d\.\d (\d+?) .*?
$!',
          $item, $matches)) {
          $received->setHeaderByKey('status',
$matches[1]);
      } else {
          list($key, $value) = explode(':',
$item);
          $received->setHeaderByKey($key,
trim($value));
      }
```

```
    }
  }
  $payload = '';
  while (!feof($resource)) {
    $payload .= fread($resource,
self::BYTES_TO_READ);
  }
  if ($received-
>getHeaderByKey(Received::HEADER_CONTENT_TYPE)
) {
    switch (TRUE) {
      case stripos($received->getHeaderByKey(

Received::HEADER_CONTENT_TYPE),

Received::CONTENT_TYPE_JSON) !== FALSE:
        $received-
>setData(json_decode($payload));
        break;
      default :
        $received->setData($payload);
        break;
          }
    }
    return $received;
}
```

Defining a cURL-based REST client

We will now have a look at our second approach for a REST client, one of which is based on the cURL extension:

1. For this approach, we will assume the same request and response classes. The initial class definition is much the same as for the Streams client discussed previously:

```
namespace Application\Web\Client;
use Application\Web\ { Request, Received };
class Curl
{
```

2. The `send()` method is quite a bit simpler than when using Streams. All we need to do is to define an array of options, and let cURL do the rest:

```
public static function send(Request $request)
{
  $data = $request->getDataEncoded();
  $received = new Received();
  switch ($request->getMethod()) {
    case Request::METHOD_GET :
      $uri = ($data)
        ? $request->getUri() . '?' . $data
        : $request->getUri();
          $options = [
            CURLOPT_URL => $uri,
            CURLOPT_HEADER => 0,
            CURLOPT_RETURNTRANSFER => TRUE,
            CURLOPT_TIMEOUT => 4
          ];
          break;
```

3. `POST` requires slightly different cURL parameters:

```
case Request::METHOD_POST :
  $options = [
    CURLOPT_POST => 1,
    CURLOPT_HEADER => 0,
    CURLOPT_URL => $request->getUri(),
    CURLOPT_FRESH_CONNECT => 1,
    CURLOPT_RETURNTRANSFER => 1,
    CURLOPT_FORBID_REUSE => 1,
    CURLOPT_TIMEOUT => 4,
    CURLOPT_POSTFIELDS => $data
  ];
  break;
}
```

4. We then execute a series of cURL functions and run the results through `getResults()`:

```
$ch = curl_init();
curl_setopt_array($ch, ($options));
if( ! $result = curl_exec($ch))
{
  trigger_error(curl_error($ch));
}
$received->setMetaData(curl_getinfo($ch));
curl_close($ch);
return self::getResults($received, $result);
}
```

5. The `getResults()` method packages results into a `Received` object:

```
protected static function getResults(Received
$received, $payload)
{
  $type = $received-
>getMetaDataByKey('content_type');
  if ($type) {
    switch (TRUE) {
      case stripos($type,
          Received::CONTENT_TYPE_JSON) !==
FALSE):
          $received-
>setData(json_decode($payload));
          break;
      default :
          $received->setData($payload);
          break;
    }
  }
  return $received;
}
```

How it works...

Be sure to copy all the preceding code into these classes:

- `Application\Web\AbstractHttp`
- `Application\Web\Request`
- `Application\Web\Received`
- `Application\Web\Client\Streams`
- `Application\Web\Client\Curl`

For this illustration, you can make a REST request to the Google Maps API to obtain driving directions between two points. You also need to create an API key for this purpose by following the directions given at https://developers.google.com/maps/documentation/directions/get-api-key.

You can then define a `chap_07_simple_rest_client_google_maps_curl.php` calling script that issues a request using the `Curl` client. You might also consider define a `chap_07_simple_rest_client_google_maps_streams.php` calling script that issues a request using the `Streams` client:

```
<?php
define('DEFAULT_ORIGIN', 'New York City');
define('DEFAULT_DESTINATION', 'Redondo Beach');
define('DEFAULT_FORMAT', 'json');
$apiKey = include __DIR__ . '/google_api_key.php';
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
use Application\Web\Request;
use Application\Web\Client\Curl;
```

You can then get the origin and destination:

```
$start = $_GET['start'] ?? DEFAULT_ORIGIN;
$end   = $_GET['end'] ?? DEFAULT_DESTINATION;
$start = strip_tags($start);
$end   = strip_tags($end);
```

You are now in a position to populate the `Request` object, and use it to generate the request:

```
$request = new Request(

'https://maps.googleapis.com/maps/api/directions/j
son',
  Request::METHOD_GET,
  NULL,
  ['origin' => $start, 'destination' => $end,
'key' => $apiKey],
  NULL
);

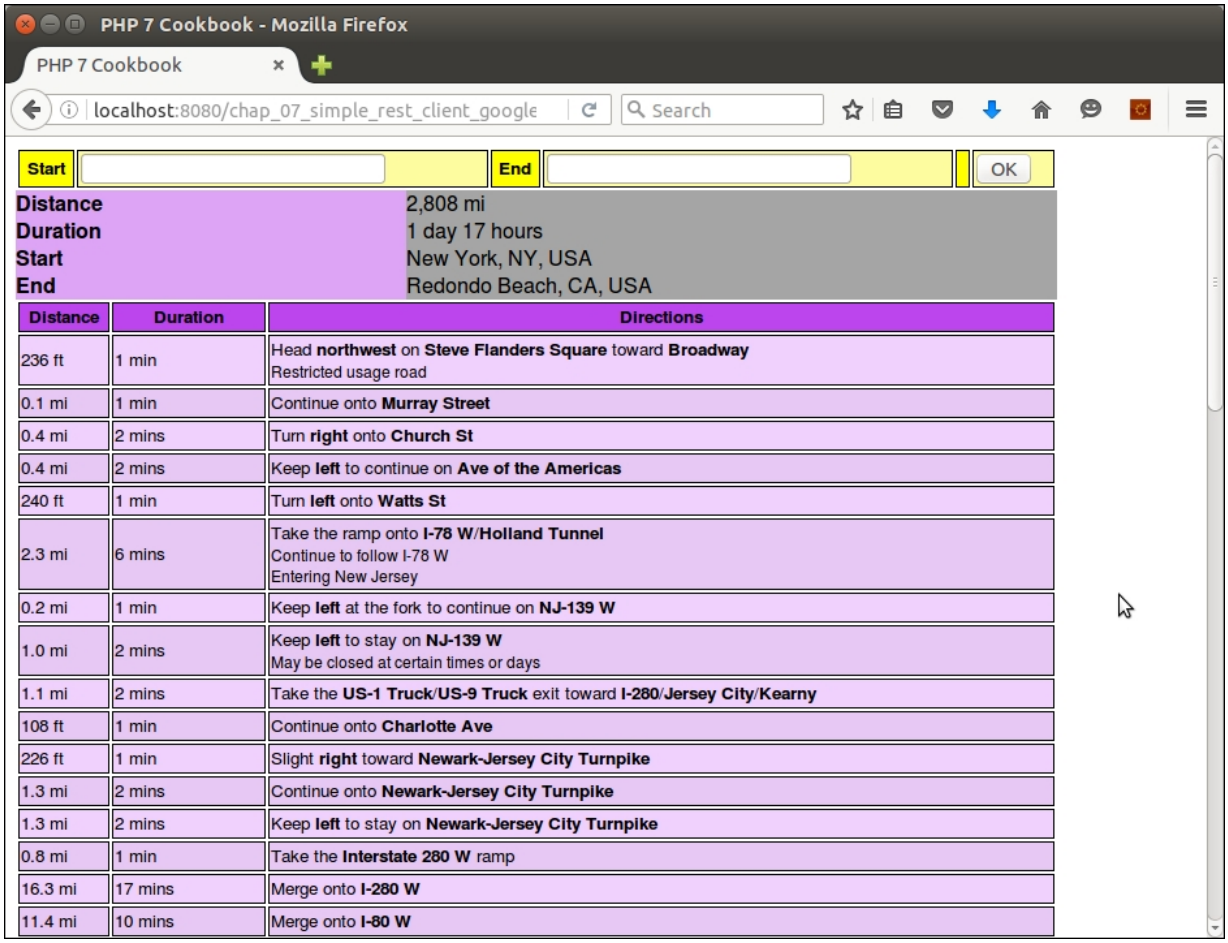
$received = Curl::send($request);
$routes   = $received->getData()->routes[0];
include __DIR__ .
```

```
'/chap_07_simple_rest_client_google_maps_template.
php';
```

For the purposes of illustration, you could also define a template that represents view logic to display the results of the request:

```
<?php foreach ($routes->legs as $item) : ?>
  <!-- Trip Info -->
  <br>Distance: <?= $item->distance->text; ?>
  <br>Duration: <?= $item->duration->text; ?>
  <!-- Driving Directions -->
  <table>
    <tr>
    <th>Distance</th><th>Duration</th>
<th>Directions</th>
    </tr>
    <?php foreach ($item->steps as $step) : ?>
    <?php $class = ($count++ & 01) ? 'color1' :
'color2'; ?>
    <tr>
    <td class="<?= $class ?>"><?= $step->distance-
>text ?></td>
    <td class="<?= $class ?>"><?= $step->duration-
>text ?></td>
    <td class="<?= $class ?>">
    <?= $step->html_instructions ?></td>
    </tr>
    <?php endforeach; ?>
  </table>
<?php endforeach; ?>
```

Here are the results of the request as seen in a browser:



There's more...

PHP Standards Recommendations (**PSR-7**) precisely defines request and response objects to be used when making requests between PHP applications. This is covered extensively in Appendix, *Defining PSR-7 Classes* .

See also

For more information on `Streams`, see this PHP documentation page http://php.net/manual/en/book.stream.php. An often asked question is "what is the difference between HTTP PUT and POST?" for an excellent discussion on this topic please refer to http://stackoverflow.com/questions/107390/whats-the-difference-

between-a-post-and-a-put-http-request. For more information on obtaining an API key from Google, please refer to these web pages:

https://developers.google.com/maps/documentation/directions/get-api-key

https://developers.google.com/maps/documentation/directions/intro#Introduction

Creating a simple REST server

There are several considerations when implementing a REST server. The answers to these three questions will then let you define your REST service:

- How is the raw request captured?
- What **Application Programming Interface** (**API**) do you want to publish?
- How do you plan to map HTTP verbs (for example, `GET`, `PUT`, `POST`, and `DELETE`) to API methods?

How to do it...

1. We will implement our REST server by building onto the request and response classes defined in the previous recipe, *Creating a simple REST client*. Review the classes discussed in the previous recipe, including the following:

    - `Application\Web\AbstractHttp`
    - `Application\Web\Request`
    - `Application\Web\Received`

2. We will also need to define a formal `Application\Web\Response` response class, based on `AbstractHttp`. The primary difference between this class and the others is that it accepts an instance of `Application\Web\Request` as an argument. The primary work is accomplished in the `__construct()` method. It's also important to set the `Content-Type` header and status:

```
namespace Application\Web;
class Response extends AbstractHttp
{

  public function __construct(Request $request
= NULL,
                              $status = NULL,
$contentType = NULL)
```

```
    {
      if ($request) {
        $this->uri = $request->getUri();
        $this->data = $request->getData();
        $this->method = $request->getMethod();
        $this->cookies = $request->getCookies();
        $this->setTransport();
      }
      $this->processHeaders($contentType);
      if ($status) {
        $this->setStatus($status);
      }
    }
    protected function
  processHeaders($contentType)
    {
      if (!$contentType) {
        $this-
  >setHeaderByKey(self::HEADER_CONTENT_TYPE,
          self::CONTENT_TYPE_JSON);
      } else {
        $this-
  >setHeaderByKey(self::HEADER_CONTENT_TYPE,
          $contentType);
      }
    }
    public function setStatus($status)
    {
      $this->status = $status;
    }
    public function getStatus()
    {
      return $this->status;
    }
  }
```

3. We are now in a position to define the `Application\Web\Rest\Server` class. You may be surprised at how simple it is. The real work is done in the associated API class:

Note

Note the use of the PHP 7 group use syntax:

```
  use Application\Web\ {
  Request,Response,Received }
```

```
namespace Application\Web\Rest;
use Application\Web\ { Request, Response,
Received };
class Server
{
  protected $api;
  public function __construct(ApiInterface
$api)
  {
    $this->api = $api;
  }
```

4. Next, we define a `listen()` method that serves as a target for the request. The heart of the server implementation is this line of code:

```
$jsonData =
json_decode(file_get_contents('php://input'),t
rue);
```

5. This captures raw input, which is assumed to be in JSON format:

```
public function listen()
{
  $request  = new Request();
  $response = new Response($request);
  $getPost  = $_REQUEST ?? array();
  $jsonData = json_decode(
    file_get_contents('php://input'),true);
  $jsonData = $jsonData ?? array();
  $request-
>setData(array_merge($getPost,$jsonData));
```

Note

We have also added a provision for authentication. Otherwise, anybody could make requests and obtain

potentially sensitive data. You will note that we do not have the server class performing authentication; rather, we leave it to the API class:

```
if (!$this->api->authenticate($request)) {
    $response-
>setStatus(Request::STATUS_401);
    echo $this->api::ERROR;
    exit;
}
```

6. We then map API methods to the primary HTTP methods `GET`, `PUT`, `POST`,and `DELETE`:

```
$id = $request->getData()[$this-
>api::ID_FIELD] ?? NULL;
switch (strtoupper($request->getMethod())) {
  case Request::METHOD_POST :
    $this->api->post($request, $response);
    break;
  case Request::METHOD_PUT :
    $this->api->put($request, $response);
    break;
  case Request::METHOD_DELETE :
    $this->api->delete($request, $response);
    break;
  case Request::METHOD_GET :
  default :
    // return all if no params
  $this->api->get($request, $response);
}
```

7. Finally, we package the response and send it out, JSON-encoded:

```
  $this->processResponse($response);
  echo json_encode($response->getData());
}
```

8. The `processResponse()` method sets headers and makes sure the result is packaged as an `Application\Web\Response` object:

```
protected function processResponse($response)
{
```

```
    if ($response->getHeaders()) {
      foreach ($response->getHeaders() as $key
=> $value) {
        header($key . ': ' . $value, TRUE,
               $response->getStatus());
      }
    }
    header(Request::HEADER_CONTENT_TYPE
    . ': ' . Request::CONTENT_TYPE_JSON, TRUE);
    if ($response->getCookies()) {
      foreach ($response->getCookies() as $key
=> $value) {
        setcookie($key, $value);
      }
    }
}
```

9. As mentioned earlier, the real work is done by the API class. We start by defining an abstract class that ensures the primary methods `get()`, `put()`, and so on are represented, and that all such methods accept request and response objects as arguments. You might notice that we have added a `generateToken()` method that uses the PHP 7 `random_bytes()` function to generate a truly random series of 16 bytes:

```
namespace Application\Web\Rest;
use Application\Web\ { Request, Response };
abstract class AbstractApi implements
ApiInterface
{
  const TOKEN_BYTE_SIZE  = 16;
  protected $registeredKeys;
  abstract public function get(Request
$request,
                               Response
$response);
  abstract public function put(Request
$request,
                               Response
$response);
  abstract public function post(Request
$request,
                                Response
$response);
  abstract public function delete(Request
$request,
```

```
                                        Response
$response);
  abstract public function
authenticate(Request $request);
  public function __construct($registeredKeys,
$tokenField)
  {
    $this->registeredKeys = $registeredKeys;
  }
  public static function generateToken()
  {
    return
bin2hex(random_bytes(self::TOKEN_BYTE_SIZE));
  }
}
```

10. We also define a corresponding interface that can be used for architectural and design purposes, as well as code development control:

```
namespace Application\Web\Rest;
use Application\Web\ { Request, Response };
interface ApiInterface
{
  public function get(Request $request,
Response $response);
  public function put(Request $request,
Response $response);
  public function post(Request $request,
Response $response);
  public function delete(Request $request,
Response $response);
  public function authenticate(Request
$request);
}
```

11. Here, we present a sample API based on `AbstractApi`. This class leverages database classes defined in Chapter 5, *Interacting with a Database*:

```
namespace Application\Web\Rest;
use Application\Web\ { Request, Response,
Received };
use Application\Entity\Customer;
use Application\Database\ { Connection,
```

```
CustomerService };

class CustomerApi extends AbstractApi
{
  const ERROR = 'ERROR';
  const ERROR_NOT_FOUND = 'ERROR: Not Found';
  const SUCCESS_UPDATE = 'SUCCESS: update
succeeded';
  const SUCCESS_DELETE = 'SUCCESS: delete
succeeded';
  const ID_FIELD = 'id';      // field name of
primary key
  const TOKEN_FIELD = 'token';  // field used
for authentication
  const LIMIT_FIELD = 'limit';
  const OFFSET_FIELD = 'offset';
  const DEFAULT_LIMIT = 20;
  const DEFAULT_OFFSET = 0;

  protected $service;

  public function __construct($registeredKeys,
                              $dbparams,
$tokenField = NULL)
  {
    parent::__construct($registeredKeys,
$tokenField);
    $this->service = new CustomerService(
      new Connection($dbparams));
  }
```

12. All methods receive request and response as arguments. You will notice the use of `getDataByKey()` to retrieve data items. The actual database interaction is performed by the service class. You might also notice that in all cases, we set an HTTP status code to inform the client of success or failure. In the case of `get()`, we look for an ID parameter. If received, we deliver information on a single customer only. Otherwise, we deliver a list of all customers using limit and offset:

```
public function get(Request $request, Response
$response)
{
  $result = array();
  $id = $request->getDataByKey(self::ID_FIELD)
```

```
?? 0;
  if ($id > 0) {
      $result = $this->service->
        fetchById($id)->entityToArray();
  } else {
    $limit  = $request-
>getDataByKey(self::LIMIT_FIELD)
      ?? self::DEFAULT_LIMIT;
    $offset = $request-
>getDataByKey(self::OFFSET_FIELD)
      ?? self::DEFAULT_OFFSET;
    $result = [];
    $fetch = $this->service->fetchAll($limit,
$offset);
    foreach ($fetch as $row) {
      $result[] = $row;
    }
  }
  if ($result) {
      $response->setData($result);
      $response-
>setStatus(Request::STATUS_200);
  } else {
      $response-
>setData([self::ERROR_NOT_FOUND]);
      $response-
>setStatus(Request::STATUS_500);
  }
}
```

13. The `put()` method is used to insert customer data:

```
public function put(Request $request, Response
$response)
{
  $cust = Customer::arrayToEntity($request-
>getData(),
                                  new
Customer());
  if ($newCust = $this->service->save($cust))
{
      $response->setData(['success' =>
self::SUCCESS_UPDATE,
                          'id' => $newCust-
>getId()]);
      $response-
>setStatus(Request::STATUS_200);
```

```
  } else {
      $response->setData([self::ERROR]);
      $response-
>setStatus(Request::STATUS_500);
  }
}
```

14. The `post()` method is used to update existing customer entries:

```
public function post(Request $request,
Response $response)
{
  $id = $request->getDataByKey(self::ID_FIELD)
?? 0;
  $reqData = $request->getData();
  $custData = $this->service->
    fetchById($id)->entityToArray();
  $updateData = array_merge($custData,
$reqData);
  $updateCust =
Customer::arrayToEntity($updateData,
  new Customer());
  if ($this->service->save($updateCust)) {
      $response->setData(['success' =>
self::SUCCESS_UPDATE,
                          'id' => $updateCust-
>getId()]);
      $response-
>setStatus(Request::STATUS_200);
  } else {
      $response->setData([self::ERROR]);
      $response-
>setStatus(Request::STATUS_500);
  }
}
```

15. As the name implies, `delete()` removes a customer entry:

```
public function delete(Request $request,
Response $response)
{
  $id = $request->getDataByKey(self::ID_FIELD)
?? 0;
  $cust = $this->service->fetchById($id);
  if ($cust && $this->service->remove($cust))
{
      $response->setData(['success' =>
```

```
self::SUCCESS_DELETE,
                              'id' => $id]);
      $response-
>setStatus(Request::STATUS_200);
  } else {
      $response-
>setData([self::ERROR_NOT_FOUND]);
      $response-
>setStatus(Request::STATUS_500);
  }
}
```

16. Finally, we define `authenticate()` to provide, in this example, a low-level mechanism to protect API usage:

```
public function authenticate(Request $request)
{
  $authToken = $request-
>getDataByKey(self::TOKEN_FIELD)
    ?? FALSE;
  if (in_array($authToken, $this-
>registeredKeys, TRUE)) {
      return TRUE;
  } else {
      return FALSE;
  }
}
}
```

How it works...

Define the following classes, which were discussed in the previous recipe:

- `Application\Web\AbstractHttp`
- `Application\Web\Request`
- `Application\Web\Received`

You can then define the following classes, described in this recipe, summarized in this table:

Class Application\Web*	Discussed in these steps
`Response`	2
`Rest\Server`	3 - 8
`Rest\AbstractApi`	9
`Rest\ApiInterface`	10
`Rest\CustomerApi`	11 - 16

You are now free to develop your own API class. If you choose to follow the illustration `Application\Web\Rest\CustomerApi`, however, you will need to also be sure to implement these classes, covered in Chapter 5, *Interacting with a Database*:

- `Application\Entity\Customer`
- `Application\Database\Connection`
- `Application\Database\CustomerService`

You can now define a `chap_07_simple_rest_server.php` script that invokes the REST server:

```
<?php
$dbParams = include __DIR__ .
'/../../config/db.config.php';
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
```

```
use Application\Web\Rest\Server;
use Application\Web\Rest\CustomerApi;
$apiKey = include __DIR__ . '/api_key.php';
$server = new Server(new CustomerApi([$apiKey],
$dbParams, 'id'));
$server->listen();
```

You can then use the built-in PHP 7 development server to listen on port `8080` for REST requests:

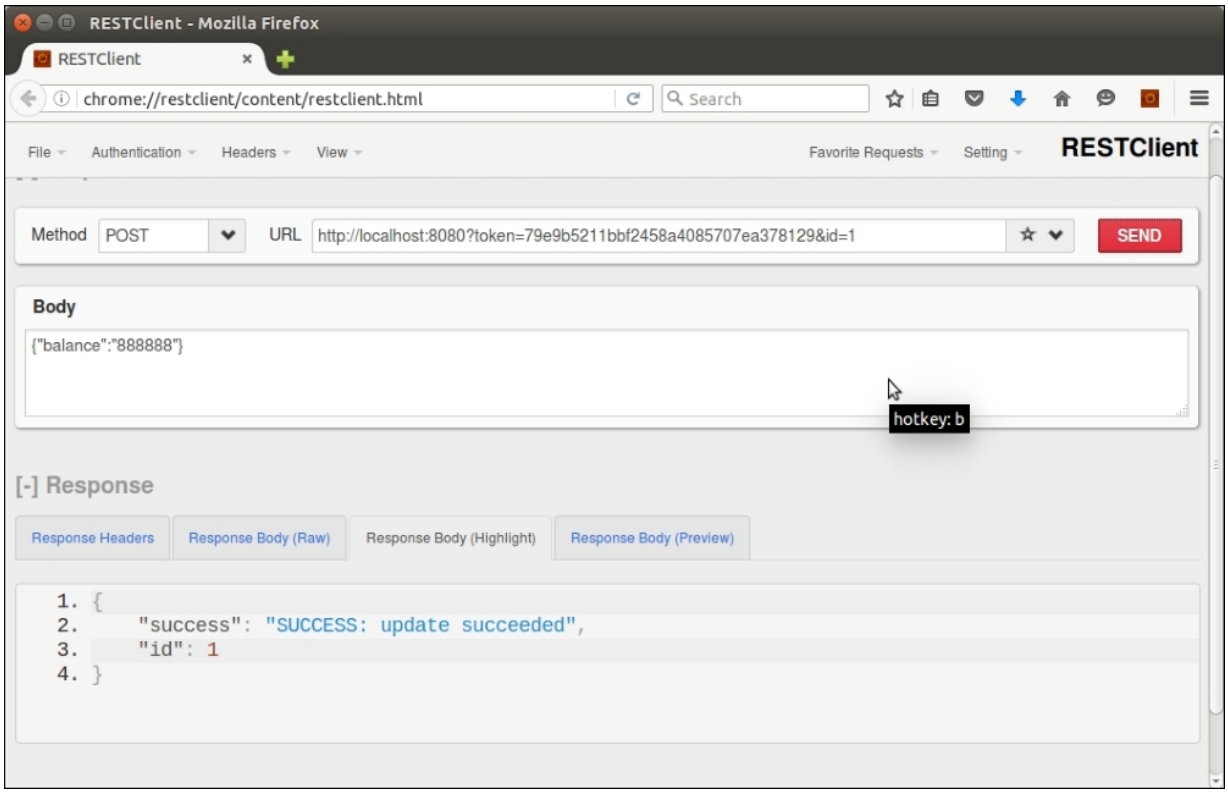
```
php -S localhost:8080
chap_07_simple_rest_server.php
```

To test your API, use the `Application\Web\Rest\AbstractApi::generateToken()` method to generate an authentication token that you can place in an `api_key.php` file, something like this:

```
<?php return '79e9b5211bbf2458a4085707ea378129';
```

You can then use a generic API client (such as the one described in the previous recipe), or a browser plugin such as RESTClient by Chao Zhou (see http://restclient.net/ for more information) to generate sample requests. Make sure you include the token for your request, otherwise the API as defined will reject the request.

Here is an example of a `POST` request for `ID 1`, which sets the `balance` field to a value of `888888`:



There's more...

There are a number of libraries that help you implement a REST server. One of my favorites is an example implementing a REST server in a single file: https://www.leaseweb.com/labs/2015/10/creating-a-simple-rest-api-in-php/

Various frameworks, such as CodeIgniter and Zend Framework, also have REST server implementations.

Creating a simple SOAP client

Using SOAP, in contrast to the process of implementing a REST client or server, is quite easy as there is a PHP SOAP extension that provides both capabilities.

> ## Note
>
> A frequently asked question is "what is the difference between SOAP and REST?" SOAP uses XML internally as its data format. SOAP uses HTTP but only for transport, and otherwise has no awareness of other HTTP methods. REST directly operates HTTP, and can use anything for data formats, but JSON is preferred. Another key difference is that SOAP can operate in conjunction with a WSDL, which makes the service self-describing, thus more publicly available. Thus, SOAP services are often offered by public institutions such as national health organizations.

How to do it...

For this example, we will make a SOAP request for an existing SOAP service offered by the United States National Weather service:

1. The first consideration is to identify the **WSDL** document. The WSDL is an XML document that describes the service:

```
$wsdl =
'http://graphical.weather.gov/xml/SOAP_server/
'
  . 'ndfdXMLserver.php?wsdl';
```

2. Next, we create a `soap client` instance using the WSDL:

```
$soap = new SoapClient($wsdl, array('trace' =>
TRUE));
```

3. We are then free to initialize some variables in anticipation of a weather forecast request:

```
$units = 'm';
$params = '';
$numDays = 7;
$weather = '';
$format = '24 hourly';
$startTime = new DateTime();
```

4. We can then make a `LatLonListCityNames()` SOAP request, identified as an operation in the WSDL, for a list of cities supported by the service. The request is returned in XML format, which suggests creating a `SimpleXLMElement` instance:

```
$xml = new SimpleXMLElement($soap-
>LatLonListCityNames(1));
```

5. Unfortunately, the list of cities and their corresponding latitude and longitude are in separate XML nodes. Accordingly, we use the `array_combine()` PHP function to create an associative array where latitude/longitude is the key, and the city name is the value. We can then later use this to present an HTML `SELECT` drop-down list, using `asort()` to alphabetize the list:

```
$cityNames = explode('|', $xml->cityNameList);
$latLonCity = explode(' ', $xml->latLonList);
$cityLatLon = array_combine($latLonCity,
$cityNames);
asort($cityLatLon);
```

6. We can then get city data from a web request as follows:

```
$currentLatLon = (isset($_GET['city'])) ?
strip_tags(urldecode($_GET['city'])) : '';
```

7. The SOAP call we wish to make is `NDFDgenByDay()`. We can determine the nature of the parameters supplied to the SOAP server by examining the WSDL:

```
<message name="NDFDgenByDayRequest">
<part name="latitude" type="xsd:decimal"/>
<part name="longitude" type="xsd:decimal"/>
<part name="startDate" type="xsd:date"/>
<part name="numDays" type="xsd:integer"/>
<part name="Unit" type="xsd:string"/>
<part name="format" type="xsd:string"/>
</message>
```

8. If the value of `$currentLatLon` is set, we can process the request. We wrap the request in a `try {} catch {}` block in case any exceptions are thrown:

```
if ($currentLatLon) {
  list($lat, $lon) = explode(',',
$currentLatLon);
  try {
      $weather = $soap-
>NDFDgenByDay($lat,$lon,
        $startTime->format('Y-m-
d'),$numDays,$unit,$format);
  } catch (Exception $e) {
      $weather .= PHP_EOL;
      $weather .= 'Latitude: ' . $lat . ' |
Longitude: ' . $lon;
      $weather .= 'ERROR' . PHP_EOL;
      $weather .= $e->getMessage() . PHP_EOL;
      $weather .= $soap->__getLastResponse() .
PHP_EOL;
  }
}
?>
```

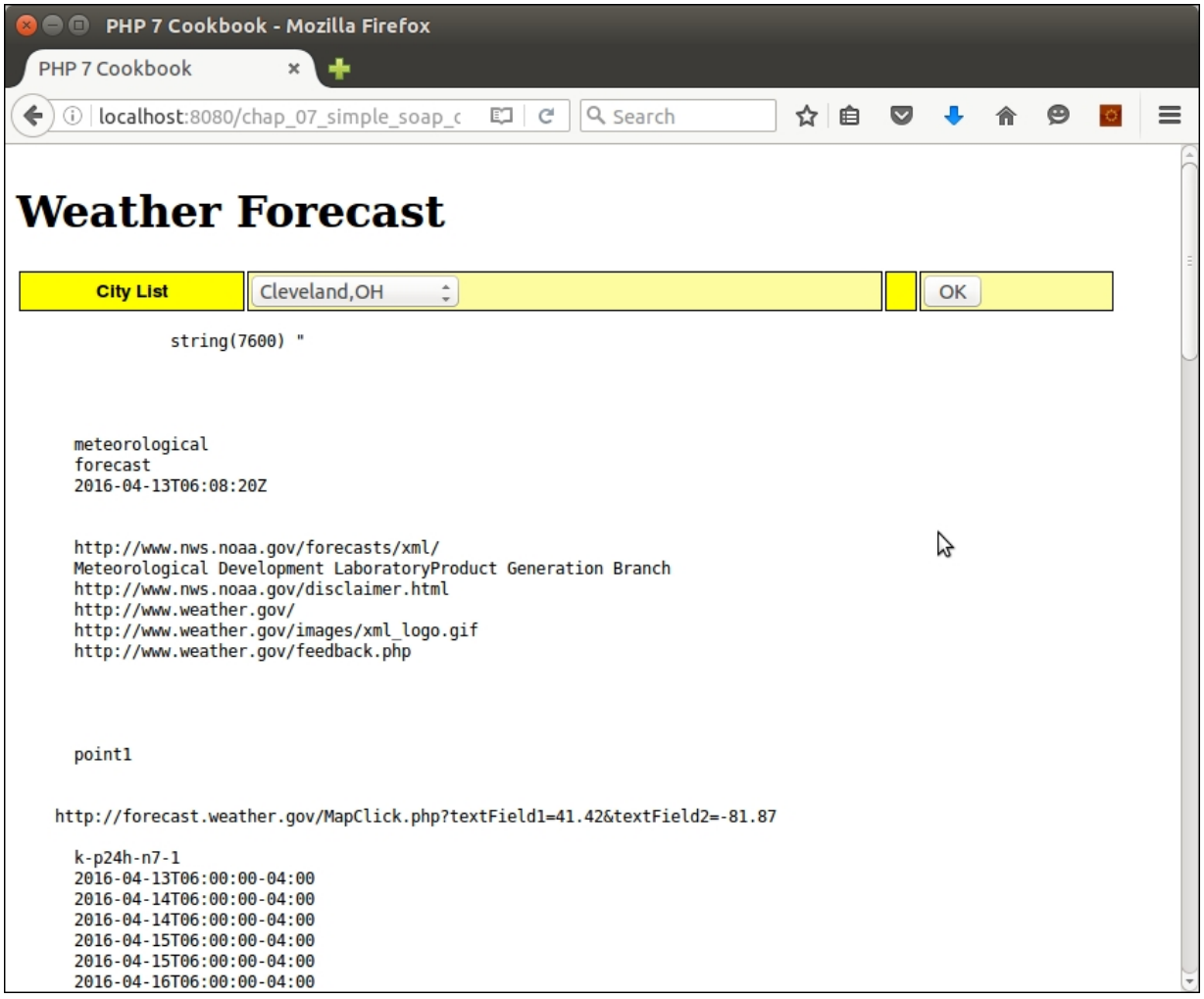
How it works...

Copy all the preceding code into a `chap_07_simple_soap_client_weather_service.php` file. You can then add view logic that displays a form with the list of cities, as well as the results:

```
<form method="get" name="forecast">
<br> City List:
<select name="city">
```

```
<?php foreach ($cityLatLon as $latLon => $city) :
?>
<?php $select = ($currentLatLon == $latLon) ? '
selected' : ''; ?>
<option value="<?= urlencode($latLon) ?>" <?=
$select ?>>
<?= $city ?></option>
<?php endforeach; ?>
</select>
<br><input type="submit" value="OK"></td>
</form>
<pre>
<?php var_dump($weather); ?>
</pre>
```

Here is the result, in a browser, of requesting the weather forecast for Cleveland, Ohio:



See also

For a good discussion on the difference between SOAP and REST, refer to the article present at http://stackoverflow.com/questions/209905/representational-state-transfer-rest-and-simple-object-access-protocol-soap?lq=1.

Creating a simple SOAP server

As with the SOAP client, we can use the PHP SOAP extension to implement a SOAP server. The most difficult part of the implementation will be generating the WSDL from the API class. We do not cover that process here as there are a number of good WSDL generators available.

How to do it...

1. First, you need an API that will be handled by the SOAP server. For this example, we define an `Application\Web\Soap\ProspectsApi` class that allows us to create, read, update, and delete the `prospects` table:

```
namespace Application\Web\Soap;
use PDO;
class ProspectsApi
{
  protected $registerKeys;
  protected $pdo;

  public function __construct($pdo,
$registeredKeys)
  {
    $this->pdo = $pdo;
    $this->registeredKeys = $registeredKeys;
  }
}
```

2. We then define methods that correspond to create, read, update, and delete. In this example, the methods are named `put()`, `get()`, `post()`, and `delete()`. These, in turn, call methods that generate SQL requests that are executed from a PDO instance. An example for `get()` is as follows:

```
public function get(array $request, array
$response)
{
  if (!$this->authenticate($request)) return
```

```
FALSE;
  $result = array();
  $id = $request[self::ID_FIELD] ?? 0;
  $email = $request[self::EMAIL_FIELD] ?? 0;
  if ($id > 0) {
      $result = $this->fetchById($id);
      $response[self::ID_FIELD] = $id;
  } elseif ($email) {
      $result = $this->fetchByEmail($email);
      $response[self::ID_FIELD] =
$result[self::ID_FIELD] ?? 0;
  } else {
      $limit = $request[self::LIMIT_FIELD]
        ?? self::DEFAULT_LIMIT;
      $offset = $request[self::OFFSET_FIELD]
        ?? self::DEFAULT_OFFSET;
      $result = [];
      foreach ($this->fetchAll($limit,
$offset) as $row) {
        $result[] = $row;
      }
  }
  $response = $this->processResponse(
    $result, $response, self::SUCCESS,
self::ERROR);
    return $response;
  }

  protected function processResponse($result,
$response,

$success_code, $error_code)
  {
    if ($result) {
        $response['data'] = $result;
        $response['code'] = $success_code;
        $response['status'] =
self::STATUS_200;
    } else {
        $response['data'] = FALSE;
        $response['code'] =
self::ERROR_NOT_FOUND;
        $response['status'] =
self::STATUS_500;
    }
    return $response;
  }
```

3. You can then generate a WSDL from your API. There are quite a few PHP-based WSDL generators available (see the *There's more...* section). Most require that you add `phpDocumentor` tags before the methods that will be published. In our example, the two arguments are both arrays. Here is the full WSDL for the API discussed earlier:

```
<?xml version="1.0" encoding="UTF-8"?>
  <wsdl:definitions xmlns:tns="php7cookbook"
targetNamespace="php7cookbook"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/so
ap/"
xmlns:s="http://www.w3.org/2001/XMLSchema"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap
/encoding/">
  <wsdl:message name="getSoapIn">
    <wsdl:part name="request" type="tns:array"
/>
    <wsdl:part name="response"
type="tns:array" />
  </wsdl:message>
  <wsdl:message name="getSoapOut">
    <wsdl:part name="return" type="tns:array"
/>
  </wsdl:message>
  <!—some nodes removed to conserve space -->
  <wsdl:portType name="CustomerApiSoap">
  <!—some nodes removed to conserve space -->
  <wsdl:binding name="CustomerApiSoap"
type="tns:CustomerApiSoap">
  <soap:binding
transport="http://schemas.xmlsoap.org/soap/htt
p" style="rpc" />
    <wsdl:operation name="get">
      <soap:operation
soapAction="php7cookbook#get" />
        <wsdl:input>
          <soap:body use="encoded"
encodingStyle=
"http://schemas.xmlsoap.org/soap/encoding/"
namespace="php7cookbook" parts="request
response" />
        </wsdl:input>
        <wsdl:output>
          <soap:body use="encoded"
```

```
encodingStyle=
"http://schemas.xmlsoap.org/soap/encoding/"
namespace="php7cookbook" parts="return" />
        </wsdl:output>
    </wsdl:operation>
  <!—some nodes removed to conserve space -->
  </wsdl:binding>
  <wsdl:service name="CustomerApi">
    <wsdl:port name="CustomerApiSoap"
binding="tns:CustomerApiSoap">
    <soap:address
location="http://localhost:8080/" />
    </wsdl:port>
  </wsdl:service>
  </wsdl:definitions>
```

4. Next, create a `chap_07_simple_soap_server.php` file, which will execute the SOAP server. Start by defining the location of the WSDL and any other necessary files (in this case, one for database configuration). If the `wsdl` parameter is set, deliver the WSDL rather than attempting to process the request. In this example, we use a simple API key to authenticate requests. We then create a SOAP server instance, assign an instance of our API class, and run `handle()`:

```
<?php
define('DB_CONFIG_FILE',
'/../config/db.config.php');
define('WSDL_FILENAME', __DIR__ .
'/chap_07_wsdl.xml');

if (isset($_GET['wsdl'])) {
    readfile(WSDL_FILENAME);
    exit;
}
$apiKey = include __DIR__ . '/api_key.php';
require __DIR__ .
'/../Application/Web/Soap/ProspectsApi.php';
require __DIR__ .
'/../Application/Database/Connection.php';
use Application\Database\Connection;
use Application\Web\Soap\ProspectsApi;
$connection = new
Application\Database\Connection(
  include __DIR__ . DB_CONFIG_FILE);
$api = new Application\Web\Soap\ProspectsApi(
```

```
    $connection->pdo, [$apiKey]);
$server = new SoapServer(WSDL_FILENAME);
$server->setObject($api);
echo $server->handle();
```

Note

Depending on the settings for your `php.ini` file, you may need to disable the WSDL cache, as follows:

```
ini_set('soap.wsdl_cache_enabled', 0);
```

If you have problems with incoming `POST` data, you can adjust this parameter as follows:

```
ini_set('always_populate_raw_post_data',
-1);
```

How it works...

You can easily test this recipe by first creating your target API class, and then generating a WSDL. You can then use the built-in PHP webserver to deliver the SOAP service with this command:

```
php -S localhost:8080
chap_07_simple_soap_server.php
```

You can then use the SOAP client discussed in the previous recipe to make a call to test the SOAP service:

```
<?php
define('WSDL_URL', 'http://localhost:8080?
wsdl=1');
$clientKey = include __DIR__ . '/api_key.php';
try {
  $client = new SoapClient(WSDL_URL);
```

```
    $response = [];
    $email = some_email_generated_by_test;
    $email = 'test5393@unlikelysource.com';
    echo "\nGet Prospect Info for Email: " . $email
. "\n";
    $request = ['token' => $clientKey, 'email' =>
$email];
    $result = $client->get($request,$response);
    var_dump($result);

} catch (SoapFault $e) {
    echo 'ERROR' . PHP_EOL;
    echo $e->getMessage() . PHP_EOL;
} catch (Throwable $e) {
    echo 'ERROR' . PHP_EOL;
    echo $e->getMessage() . PHP_EOL;
} finally {
    echo $client->__getLastResponse() . PHP_EOL;
}
```

Here is the output for email address `test5393@unlikelysource.com`:



See also

A simple Google search for WSDL generators for PHP came back with easily a dozen results. The one used to generate the WSDL for the `ProspectsApi` class was based on https://code.google.com/archive/p/php-wsdl-creator/. For more information on `phpDocumentor`, refer to the page at https://www.phpdoc.org/.

Chapter 8. Working with Date/Time and International Aspects

In this chapter, we will cover the following topics:

- Using emoticons or emoji in a view script
- Converting complex characters
- Getting the locale from browser data
- Formatting numbers by locale
- Handling currency by locale
- Formatting date/time by locale
- Creating an HTML international calendar generator
- Building a recurring events generator
- Handling translation without gettext

Introduction

We will start this chapter with two recipes that take advantage of a new **Unicode** escape syntax introduced with **PHP 7**. After that, we will cover how to determine a web visitor's **locale** from browser data. The next few recipes will cover the creation of a locale class, which will allow you to represent numbers, currency, dates, and time in a format specific to a locale. Finally, we will cover recipes that demonstrate how to generate an internationalized calendar, handle recurring events, and perform translation without having to use `gettext`.

Using emoticons or emoji in a view script

The word **emoticons** is a composite of *emotion* and *icon*. **Emoji**, originating from Japan, is another, larger, widely used set of icons. These icons are the little smiley faces, tiny ninjas, and rolling-on-the-floor-laughing icons that are so popular on any website that has a social networking aspect. Prior to PHP 7, however, producing these little beasties was an exercise in frustration.

How to do it...

1. First and foremost, you need to know the Unicode for the icon you wish to present. A quick search on the Internet will direct you to any one of several excellent charts. Here are the codes for the three *hear-no-evil*, *see-no-evil*, and *speak-no-evil* monkey icons:

   `U+1F648`, `U+1F649`, and `U+1F64A`

2. Any Unicode output to the browser must be properly identified. This is most often done by way of a `meta` tag. You should set the character set to UTF-8. Here is an example:

```
<head>
  <title>PHP 7 Cookbook</title>
  <meta http-equiv="content-type"
content="text/html;charset=utf-8" />
</head>
```

3. The traditional approach was to simply use HTML to display the icons. Thus, you could do something like this:

```
<table>
  <tr>
    <td>&#x1F648;</td>
    <td>&#x1F649;</td>
    <td>&#x1F64A;</td>
  </tr>
</table>
```

4. As of PHP 7, you can now construct full Unicode characters using this syntax: `"\u{xxx}"`. Here is an example with the same three icons as in the preceding bullet:

```
<table>
  <tr>
    <td><?php echo "\u{1F648}"; ?></td>
    <td><?php echo "\u{1F649}"; ?></td>
    <td><?php echo "\u{1F64A}"; ?></td>
  </tr>
</table>
```

> **Note**
>
> Your operating system and browser must both support Unicode and must also have the right set of fonts. In Ubuntu Linux, for example, you would need to install the `ttf-ancient-fonts` package to see emoji in your browser.

How it works...

In PHP 7, a new syntax was introduced that lets you render any Unicode character. Unlike other languages, the new PHP syntax allows for a variable number of hex digits. The basic format is this:

```
\u{xxxx}
```

The entire construct must be double quoted (or use **heredoc**). `xxxx` could be any combination of hex digits, 2, 4, 6, and above.

Create a file called `chap_08_emoji_using_html.php`. Be sure to include the `meta` tag that signals the browser that UTF-8 character encoding is being used:

```
<!DOCTYPE html>
<html>
  <head>
    <title>PHP 7 Cookbook</title>
    <meta http-equiv="content-type"
content="text/html;charset=utf-8" />
  </head>
```

Next, set up a basic HTML table, and display a row of emoticons/emoji:

```
  <body>
    <table>
      <tr>
        <td>&#x1F648;</td>
        <td>&#x1F649;</td>
        <td>&#x1F64A;</td>
      </tr>
    </table>
  </body>
</html>
```

Now add a row using PHP to emit emoticons/emoji:

```
  <tr>
    <td><?php echo "\u{1F648}"; ?></td>
    <td><?php echo "\u{1F649}"; ?></td>
    <td><?php echo "\u{1F64A}"; ?></td>
  </tr>
```

Here is the output seen from Firefox:



See also

- For a list of emoji codes, see http://unicode.org/emoji/charts/full-emoji-list.html

Converting complex characters

The ability to access the entire Unicode character set opens up many new possibilities for rendering complex characters, especially characters in alphabets other than Latin-1.

How to do it...

1. Some languages are read right-to-left instead of left-to-right. Examples include Hebrew and Arabic. In this example, we show you how to present *reverse* text using the `U+202E` Unicode character for right-to-left override. The following line of code prints `txet desreveR`:

```
echo "\u{202E}Reversed text";
echo "\u{202D}";    // returns output to left-
to-right
```

> **Note**
>
> Don't forget to invoke the left-to-right override character, `U+202D`, when finished!

2. Another consideration is the use of composed characters. One such example is `ñ` (the letter `n` with a tilde `~` floating above). This is used in words such as *mañana* (the Spanish word for morning or tomorrow, depending on the context). There is a *composed character* available, represented by Unicode code `U+00F1`. Here is an example of its use, which echoes `mañana`:

```
echo "ma\u{00F1}ana"; // shows mañana
```

3. This could potentially impact search possibilities, however. Imagine that your customers do not have a keyboard with this

composed character. If they start to type `man` in an attempt to search for `mañana`, they will be unsuccessful.

4. Having access to the *full* Unicode set offers other possibilities. Instead of using the *composed* character, you can use a combination of the original letter `n` along with the Unicode *combining* code, which places a floating tilde on top of the letter. In this `echo` command, the output is the same as previously. Only the way the word is formed differs:

```
echo "man\u{0303}ana"; // also shows mañana
```

5. A similar application could be made for accents. Consider the French word `élève` (student). You could render it using composed characters, or by using combining codes to float the accents above the letter. Consider the two following examples. Both examples produce the same output, but are rendered differently:

```
echo "\u{00E9}l\u{00E8}ve";
echo "e\u{0301}le\u{0300}ve";
```

How it works...

Create a file called `chap_08_control_and_combining_unicode.php`. Be sure to include the `meta` tag that signals the browser that UTF-8 character encoding is being used:

```
<!DOCTYPE html>
<html>
  <head>
    <title>PHP 7 Cookbook</title>
    <meta http-equiv="content-type"
content="text/html;charset=utf-8" />
  </head>
```

Next, set up basic PHP and HTML to display the examples discussed previously:

```
  <body>
    <pre>
      <?php
        echo "\u{202E}Reversed text"; // reversed
        //echo "\u{202D}"; // stops reverse
        echo "mañana";  // using pre-composed
characters
        echo "ma\u{00F1}ana"; // pre-composed
character
        echo "man\u{0303}ana"; // "n" with
combining ~ character (U+0303)
        echo "élève";
        echo "\u{00E9}l\u{00E8}ve"; // pre-
composed characters
        echo "e\u{0301}le\u{0300}ve"; // e +
combining characters
      ?>
    </pre>
</body>
</html>
```

Here is the output from a browser:



Getting the locale from browser data

In order to improve the user experience on a website, it's important to display information in a format that is acceptable in the user's locale. **Locale** is a generic term used to indicate an area of the world. An effort in the I.T. community has been made to codify locales using a two-part designation consisting of codes for both language and country. But when a person visits your website, how do you know their locale? Probably the most useful technique involves examining the HTTP language header.

How to do it...

1. In order to encapsulate locale functionality, we will assume a class, `Application\I18n\Locale`. We will have this class extend an existing class, `Locale`, which is part of the PHP `Intl` extension.

> ### Note
>
> **I18n** is a common abbreviation for **Internationalization**. (Count the number of letters!)

```
namespace Application\I18n;
use Locale as PhpLocale;
class Locale extends PhpLocale
{
  const FALLBACK_LOCALE = 'en';
  // some code
}
```

2. To get an idea of what an incoming request looks like, use `phpinfo(INFO_VARIABLES)`. Be sure to disable this function immediately after testing as it gives away too much information to potential attackers:

```
<?php phpinfo(INFO_VARIABLES); ?>
```

3. Locale information is stored in `$_SERVER['HTTP_ACCEPT_LANGUAGE']`. The value will take this general form: `ll-CC,rl;q=0.n, ll-CC,rl;q=0.n`, as defined in this table:

Abbreviation	Meaning
`ll`	Two-character lowercase code representing the language.
`-`	Separates language from country in the locale code `ll-CC`.
`CC`	Two-character uppercase code representing the country.
`,`	Separates locale code from fallback **root locale** code (usually the same as the language code).
`rl`	Two-character lowercase code representing the suggested root locale.

Abbreviation	Meaning
`;`	Separates locale information from quality. If quality is missing, default is `q=1` (100%) probability; this is preferred.
`q`	Quality.
`0.n`	Some value between 0.00 and 1.0. Multiply this value by 100 to get the percentage of probability that this is the actual language preferred by this visitor.

4. There can easily be more than one locale listed. For example, the website visitor could have multiple languages installed on their computer. It so happens that the PHP `Locale` class has a method, `acceptFromHttp()`, which reads the `Accept-language` header string and gives us the desired setting:

```
protected $localeCode;
public function
setLocaleCode($acceptLangHeader)
{
  $this->localeCode = $this-
>acceptFromHttp($acceptLangHeader);
}
```

5. We can then define the appropriate getters. The `getAcceptLanguage()` method returns the value from `$_SERVER['HTTP_ACCEPT_LANGUAGE']`:

```
public function getAcceptLanguage()
{
  return $_SERVER['HTTP_ACCEPT_LANGUAGE'] ??
```

```
self::FALLBACK_LOCALE;
}
public function getLocaleCode()
{
  return $this->localeCode;
}
```

6. Next we define a constructor that allows us to "manually" set the locale. Otherwise, the locale information is drawn from the browser:

```
public function __construct($localeString =
NULL)
{
  if ($localeString) {
    $this->setLocaleCode($localeString);
  } else {
    $this->setLocaleCode($this-
>getAcceptLanguage());
  }
}
```

7. Now comes the big decision: what to do with this information! This is covered in the next few recipes.

> ## Note
>
> Even though a visitor appears to accept one or more languages, that visitor does not necessarily want contents in the language/locale indicated by their browser. Accordingly, although you can certainly set the locale given this information, you should also provide them with a static list of alternative languages.

How it works...

In this illustration, let's take three examples:

- information derived from the browser
- a preset locale `fr-FR`

- a string taken from RFC 2616: `da, en-gb;q=0.8, en;q=0.7`

Place the code from steps 1 to 6 into a file, `Locale.php`, which is in the `Application\I18n` folder.

Next, create a file, `chap_08_getting_locale_from_browser.php`, which sets up autoloading and uses the new class:

```
<?php
  require __DIR__ .
'/../Application/Autoload/Loader.php';
  Application\Autoload\Loader::init(__DIR__ .
'/..');
  use Application\I18n\Locale;
```

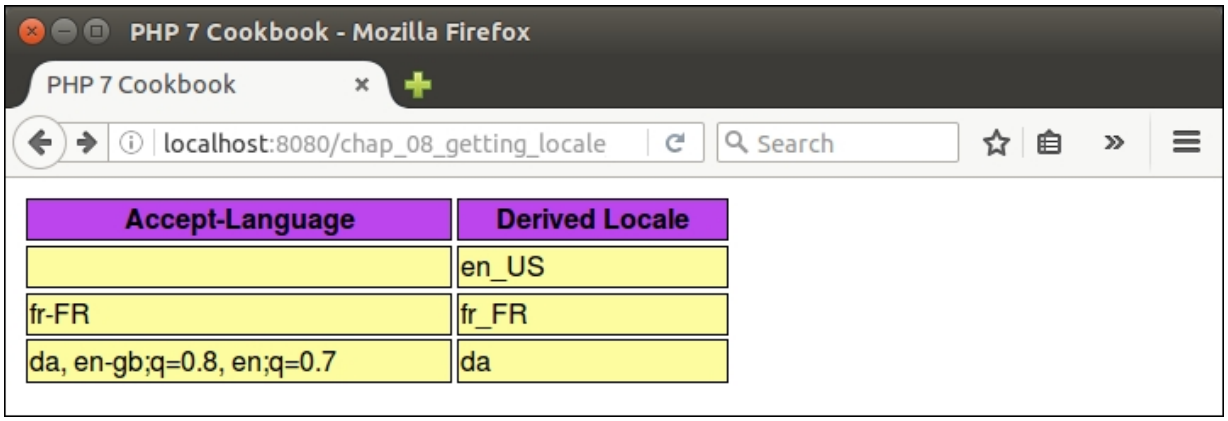
Now you can define an array with the three test locale strings:

```
$locale = [NULL, 'fr-FR', 'da, en-gb;q=0.8,
en;q=0.7'];
```

Finally, loop through the three locale strings, creating instances of the new class. Echo the value returned from `getLocaleCode()` to see what choice was made:

```
echo '<table>';
foreach ($locale as $code) {
  $locale = new Locale($code);
  echo '<tr>
    <td>' . htmlspecialchars($code) . '</td>
    <td>' . $locale->getLocaleCode() . '</td>
  </tr>';
}
echo '</table>';
```

Here is the result (with a little bit of styling):



See also

- For information on the PHP `Locale` class, see http://php.net/manual/en/class.locale.php
- For more information on the `Accept-Language` header, see section 14.4 of RFC 2616: https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html

Formatting numbers by locale

Numeric representations can vary by locale. As a simple example, in the UK one would see the number three million, eighty thousand, five hundred and twelve, and ninety-two one hundredths as follows:

```
3,080,512.92.
```

In France, however, the same number might appear like so:

```
3 080 512,92
```

How to do it...

Before you can represent a number in a locale-specific manner, you need to determine the locale. This can be accomplished using the `Application\I18n\Locale` class discussed in the previous recipe. The locale can be set manually or from header information.

1. Next, we will make use of the `format()` method of the `NumberFormatter` class, to both output and parse numbers in a locale-specific format. First we add a property that will contain an instance of the `NumberFormatter` class:

```
use NumberFormatter;
protected $numberFormatter;
```

> ### Note
>
> Our initial thought would be to consider using the PHP function `setlocale()` to produce numbers formatted according to locale. The problem with this legacy approach, however, is that *everything* will be considered based on this locale. This could introduce problems dealing with data that

is stored according to database specifications. Another issue with `setlocale()` is that it is based on outdated standards, including RFC 1766 and ISO 639. Finally, `setlocale()` is highly dependent on operating system locale support, which will make our code non-portable.

2. Normally, the next step would be to set `$numberFormatter` in the constructor. The problem with this approach, in the case of our `Application\I18n\Locale` class, is that we would end up with a top-heavy class, as we will also need to perform currency and date formatting as well. Accordingly, we add a `getter` that first checks to see whether an instance of `NumberFormatter` has already been created. If not, an instance is created and returned. The first argument in the new `NumberFormatter` is the locale code. The second argument, `NumberFormatter::DECIMAL`, represents what type of formatting we need:

```
public function getNumberFormatter()
{
  if (!$this->numberFormatter) {
    $this->numberFormatter = new
NumberFormatter($this->getLocaleCode(),
NumberFormatter::DECIMAL);
  }
  return $this->numberFormatter;
}
```

3. We then add a method that, given any number, will produce a string that represents that number formatted according to the locale:

```
public function formatNumber($number)
{
  return $this->getNumberFormatter()-
>format($number);
}
```

4. Next we add a method that can be used to parse numbers according to the locale, producing a native PHP numeric value. Please note that the result might not return `FALSE` on parse failure depending on the server's ICU version:

```
    public function parseNumber($string)
    {
      $result = $this->getNumberFormatter()-
    >parse($string);
      return ($result) ? $result :
    self::ERROR_UNABLE_TO_PARSE;
    }
```

How it works...

Make the additions to the `Application\I18n\Locale` class as discussed in the preceding bullet points. You can then create a `chap_08_formatting_numbers.php` file, which sets up autoloading and uses this class:

```
<?php
  require __DIR__ .
'/../Application/Autoload/Loader.php';
  Application\Autoload\Loader::init(__DIR__ .
'/..');
  use Application\I18n\Locale;
```

For this illustration, create two `Locale` instances, one for the UK, the other for France. You can also designate a large number to be used for testing:

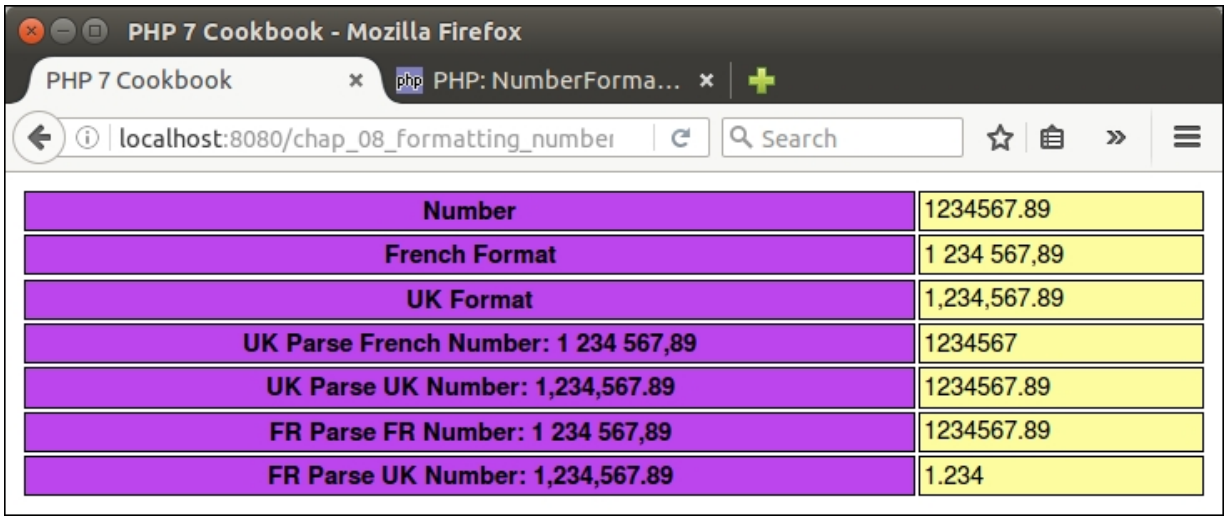
```
  $localeFr = new Locale('fr_FR');
  $localeUk = new Locale('en_GB');
  $number   = 1234567.89;
?>
```

Finally, you can wrap the `formatNumber()` and `parseNumber()` methods in the appropriate HTML display logic and view the results:

```
<!DOCTYPE html>
<html>
  <head>
    <title>PHP 7 Cookbook</title>
    <meta http-equiv="content-type"
content="text/html;charset=utf-8" />
```

```
    <link rel="stylesheet" type="text/css"
href="php7cookbook_html_table.css">
  </head>
  <body>
    <table>
      <tr>
        <th>Number</th>
        <td>1234567.89</td>
      </tr>
      <tr>
        <th>French Format</th>
        <td><?= $localeFr->formatNumber($number);
?></td>
      </tr>
      <tr>
        <th>UK Format</th>
        <td><?= $localeUk->formatNumber($number);
?></td>
      </tr>
      <tr>
        <th>UK Parse French Number: <?= $localeFr-
>formatNumber($number) ?></th>
        <td><?= $localeUk->parseNumber($localeFr-
>formatNumber($number)); ?></td>
      </tr>
      <tr>
        <th>UK Parse UK Number: <?= $localeUk-
>formatNumber($number) ?></th>
        <td><?= $localeUk->parseNumber($localeUk-
>formatNumber($number)); ?></td>
      </tr>
      <tr>
        <th>FR Parse FR Number: <?= $localeFr-
>formatNumber($number) ?></th>
        <td><?= $localeFr->parseNumber($localeFr-
>formatNumber($number)); ?></td>
      </tr>
      <tr>
        <th>FR Parse UK Number: <?= $localeUk-
>formatNumber($number) ?></th>
        <td><?= $localeFr->parseNumber($localeUk-
>formatNumber($number)); ?></td>
      </tr>
    </table>
  </body>
</html>
```

Here is the result as seen from a browser:



Note

Note that if the locale is set to `fr_FR`, a UK formatted number, when parsed, does not return the correct value. Likewise, when the locale is set to `en_GB`, a French formatted number does not return the correct value upon parsing. Accordingly, you might want to consider adding a validation check before attempting to parse the number.

See also

- For more information on the use and abuse of `setlocale()` please refer to this page: http://php.net/manual/en/function.setlocale.php.
- For a brief note on why number formatting will produce an error on some servers, but not others, check the **ICU** (**International Components for Unicode**) version. See the comments on this page: http://php.net/manual/en/numberformatter.parse.php. For more info on ICU formatting, see http://userguide.icu-project.org/formatparse.

Handling currency by locale

The technique for handling currency is similar to that for numbers. We will even use the same `NumberFormatter` class! There is one major difference, however, and it is a *show stopper*: in order to properly format currency, you will need to have on hand the currency code.

How to do it...

1. The first order of business is to have the currency codes available in some format. One possibility is to simply add the currency code as an `Application\I18n\Locale` class constructor argument:

```
const FALLBACK_CURRENCY = 'GBP';
protected $currencyCode;
public function __construct($localeString =
NULL, $currencyCode = NULL)
{
  // add this to the existing code:
  $this->currencyCode = $currencyCode ??
self::FALLBACK_CURRENCY;
}
```

> ### Note
>
> This approach, although obviously solid and workable, tends to fall into the category called *halfway measures* or *the easy way out*! This approach would also tend to eliminate full automation as the currency code is not available from the HTTP header. As you have probably gathered from other recipes in this book, we do not shy away from a more complex solution so, as the saying goes, *strap on your seat belts*!

2. We will first need to establish some sort of lookup mechanism, where, given a country code, we can obtain its predominant currency code. For this illustration, we will use the Adapter software design pattern. According to this pattern, we should be able to create different classes, which could potentially operate in entirely different ways, but which produce the same result. Accordingly, we need to define the desired result. For this purpose, we introduce a class, `Application\I18n\IsoCodes`. As you can see, this class has all the pertinent properties, along with a sort-of universal constructor:

```
namespace Application\I18n;
class IsoCodes
{
  public $name;
  public $iso2;
  public $iso3;
  public $iso_numeric;
  public $iso_3166;
  public $currency_name;
  public $currency_code;
  public $currency_number;
  public function __construct(array $data)
  {
    $vars = get_object_vars($this);
    foreach ($vars as $key => $value) {
      $this->$key = $data[$key] ?? NULL;
    }
  }
}
```

3. Next we define an interface that has the method we require to perform the *country-code-to-currency-code* lookup. In this case, we introduce `Application\I18n\IsoCodesInterface`:

```
namespace Application\I18n;

interface IsoCodesInterface
{
  public function
getCurrencyCodeFromIso2CountryCode($iso2) :
IsoCodes;
}
```

4. Now we are ready to build a lookup adapter class, which we will call `Application\I18n\IsoCodesDb`. It implements the abovementioned interface, and accepts an `Application\Database\Connection` instance (see Chapter 1, *Building a Foundation*), which is used to perform the lookup. The constructor sets up the required information, including the connection, the lookup table name, and the column that represents the ISO2 code. The lookup method required by the interface then issues an SQL statement and returns an array, which is then used to build an `IsoCodes` instance:

```
namespace Application\I18n;

use PDO;
use Application\Database\Connection;

class IsoCodesDb implements IsoCodesInterface
{
  protected $isoTableName;
  protected $iso2FieldName;
  protected $connection;
  public function __construct(Connection
$connection, $isoTableName, $iso2FieldName)
  {
    $this->connection = $connection;
    $this->isoTableName = $isoTableName;
    $this->iso2FieldName = $iso2FieldName;
  }
  public function
getCurrencyCodeFromIso2CountryCode($iso2) :
IsoCodes
  {
    $sql = sprintf('SELECT * FROM %s WHERE %s
= ?', $this->isoTableName, $this-
>iso2FieldName);
    $stmt = $this->connection->pdo-
>prepare($sql);
    $stmt->execute([$iso2]);
    return new IsoCodes($stmt-
>fetch(PDO::FETCH_ASSOC);
  }
}
```

5. Now we turn our attention back to the `Application\I18n\Locale` class. We first add a couple of new properties and class constants:

```
const ERROR_UNABLE_TO_PARSE = 'ERROR: Unable
to parse';
const FALLBACK_CURRENCY = 'GBP';

protected $currencyFormatter;
protected $currencyLookup;
protected $currencyCode;
```

6. We add new method that retrieves the country code from the locale string. We can leverage the `getRegion()` method, which comes from the PHP `Locale` class (which we extend). Just in case it's needed, we also add a method, `getCurrencyCode()`:

```
public function getCountryCode()
{
  return $this->getRegion($this-
>getLocaleCode());
}
public function getCurrencyCode()
{
  return $this->currencyCode;
}
```

7. As with formatting numbers, we define a `getCurrencyFormatter(I)`, much as we did `getNumberFormatter()` (shown previously). Notice that `$currencyFormatter` is defined using `NumberFormatter`, but with a different second parameter:

```
public function getCurrencyFormatter()
{
  if (!$this->currencyFormatter) {
    $this->currencyFormatter = new
NumberFormatter($this->getLocaleCode(),
NumberFormatter::CURRENCY);
  }
  return $this->currencyFormatter;
}
```

8. We then add a currency code lookup to the class constructor if the lookup class has been defined:

```
public function __construct($localeString =
NULL, IsoCodesInterface $currencyLookup =
NULL)
{
  // add this to the existing code:
  $this->currencyLookup = $currencyLookup;
  if ($this->currencyLookup) {
    $this->currencyCode = $this-
>currencyLookup-
>getCurrencyCodeFromIso2CountryCode($this-
>getCountryCode())->currency_code;
  } else {
    $this->currencyCode =
self::FALLBACK_CURRENCY;
  }
}
```

9. Then add the appropriate currency format and parse methods. Note that parsing currency, unlike parsing numbers, will return `FALSE` if the parsing operation is not successful:

```
public function formatCurrency($currency)
{
  return $this->getCurrencyFormatter()-
>formatCurrency($currency, $this-
>currencyCode);
}
public function parseCurrency($string)
{
  $result = $this->getCurrencyFormatter()-
>parseCurrency($string, $this->currencyCode);
  return ($result) ? $result :
self::ERROR_UNABLE_TO_PARSE;
}
```

How it works...

Create the following classes, as covered in the first several bullet points:

Class	Bullet point discussed
`Application\I18n\IsoCodes`	3
`Application\I18n\IsoCodesInterface`	4
`Application\I18n\IsoCodesDb`	5

We will assume, for the purposes of this illustration, that we have a populated MySQL database table, `iso_country_codes`, which has this structure:

```
CREATE TABLE `iso_country_codes` (
  `name` varchar(128) NOT NULL,
  `iso2` varchar(2) NOT NULL,
  `iso3` varchar(3) NOT NULL,
  `iso_numeric` int(11) NOT NULL AUTO_INCREMENT,
  `iso_3166` varchar(32) NOT NULL,
  `currency_name` varchar(32) DEFAULT NULL,
  `currency_code` char(3) DEFAULT NULL,
  `currency_number` int(4) DEFAULT NULL,
  PRIMARY KEY (`iso_numeric`)
) ENGINE=InnoDB AUTO_INCREMENT=895 DEFAULT
CHARSET=utf8;
```

Make the additions to the `Application\I18n\Locale` class, as discussed in bullet points 6 to 9 previously. You can then create a `chap_08_formatting_currency.php` file, which sets up autoloading and uses the appropriate classes:

```
<?php
define('DB_CONFIG_FILE', __DIR__ .
'/../config/db.config.php');
require __DIR__ .
```

```
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
use Application\I18n\Locale;
use Application\I18n\IsoCodesDb;
use Application\Database\Connection;
use Application\I18n\Locale;
```

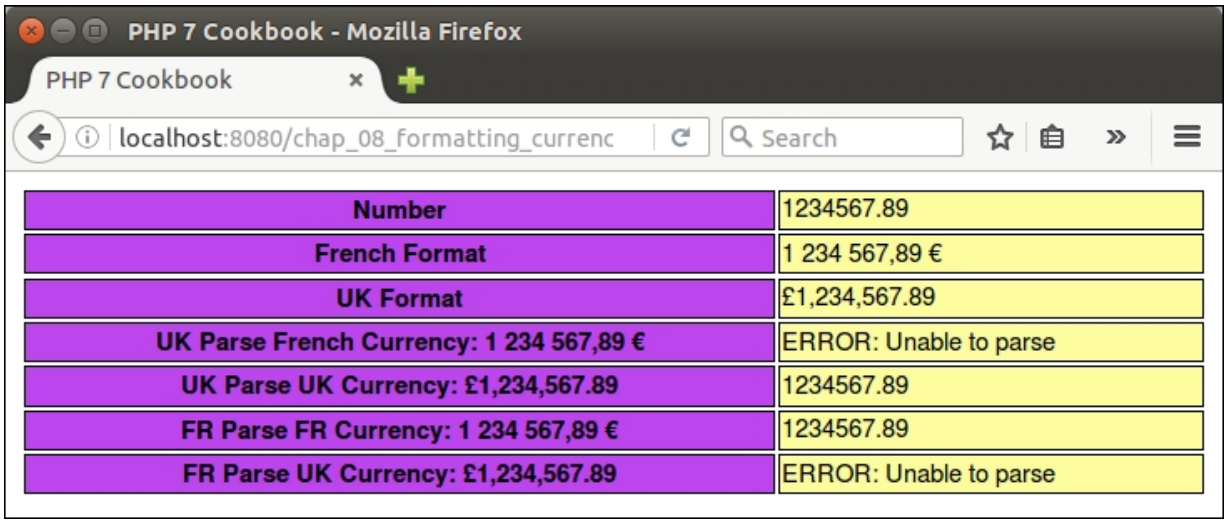
Next, we create instances of the `Connection` and `IsoCodesDb` classes:

```
$connection = new Connection(include
DB_CONFIG_FILE);
$isoLookup = new IsoCodesDb($connection,
'iso_country_codes', 'iso2');
```

For this illustration, create two `Locale` instances, one for the UK, the other for France. You can also designate a large number to be used for testing:

```
$localeFr = new Locale('fr-FR', $isoLookup);
$localeUk = new Locale('en_GB', $isoLookup);
$number   = 1234567.89;
?>
```

Finally, you can wrap the `formatCurrency()` and `parseCurrency()` methods in the appropriate HTML display logic and view the results. Base your view logic on that presented in the *How it works...* section of the previous recipe (not repeated here to save trees!). Here is the final output:



See also

- The most up-to-date list of currency codes is maintained by **ISO** (**International Standards Organization**). You can obtain this list in either **XML** or **XLS** (that is, **Microsoft Excel** spreadsheet format). Here is the page where these lists can be found: http://www.currency-iso.org/en/home/tables/table-a1.html.

Formatting date/time by locale

The formatting of date and time varies region to region. As a classic example, consider the year 2016, month April, day 15 and a time in the evening. The format preferred by denizens of the United States would be 7:23 PM, 4/15/2016, whereas in China you would most likely see 2016-04-15 19:23. As mentioned with number and currency formatting, it would also be important to display (and parse) dates in a format acceptable to your web visitors.

How to do it...

1. First of all, we need to modify `Application\I18n\Locale`, adding statements to use date formatting classes:

```
use IntlCalendar;
use IntlDateFormatter;
```

2. Next, we add a property to represent an `IntlDateFormatter` instance, as well as a series of predefined constants:

```
const DATE_TYPE_FULL   =
IntlDateFormatter::FULL;
const DATE_TYPE_LONG   =
IntlDateFormatter::LONG;
const DATE_TYPE_MEDIUM =
IntlDateFormatter::MEDIUM;
const DATE_TYPE_SHORT  =
IntlDateFormatter::SHORT;

const ERROR_UNABLE_TO_PARSE = 'ERROR: Unable
to parse';
const ERROR_UNABLE_TO_FORMAT = 'ERROR: Unable
to format date';
const ERROR_ARGS_STRING_ARRAY = 'ERROR: Date
must be string YYYY-mm-dd HH:ii:ss or
array(y,m,d,h,i,s)';
const ERROR_CREATE_INTL_DATE_FMT = 'ERROR:
Unable to create international date
formatter';
```

```
protected $dateFormatter;
```

3. After that, we are in a position to define a method, `getDateFormatter()`, which returns an `IntlDateFormatter` instance. The value of `$type` matches one of the `DATE_TYPE_*` constants defined previously:

```
public function getDateFormatter($type)
{
  switch ($type) {
    case self::DATE_TYPE_SHORT :
      $formatter = new
IntlDateFormatter($this->getLocaleCode(),
        IntlDateFormatter::SHORT,
IntlDateFormatter::SHORT);
      break;
    case self::DATE_TYPE_MEDIUM :
      $formatter = new
IntlDateFormatter($this->getLocaleCode(),
IntlDateFormatter::MEDIUM,
IntlDateFormatter::MEDIUM);
      break;
    case self::DATE_TYPE_LONG :
      $formatter = new
IntlDateFormatter($this->getLocaleCode(),
IntlDateFormatter::LONG,
IntlDateFormatter::LONG);
      break;
    case self::DATE_TYPE_FULL :
      $formatter = new
IntlDateFormatter($this->getLocaleCode(),
IntlDateFormatter::FULL,
IntlDateFormatter::FULL);
      break;
    default :
      throw new
InvalidArgumentException(self::ERROR_CREATE_IN
TL_DATE_FMT);
  }
  $this->dateFormatter = $formatter;
  return $this->dateFormatter;
}
```

4. Next we define a method that produces a locale formatted date. Defining the format of the incoming `$date` is a bit tricky. It cannot

be locale-specific, otherwise we will need to parse it according to locale rules, with unpredictable results. A better strategy would be to accept an array of values that represent year, month, day, and so on as integers. As a fallback, we will accept a string but only in this format: `YYYY-mm-dd HH:ii:ss`. Time zone is optional, and can be set separately. First we initialize variables:

```
public function formatDate($date, $type,
$timeZone = NULL)
{
  $result   = NULL;
  $year     = date('Y');
  $month    = date('m');
  $day      = date('d');
  $hour     = 0;
  $minutes  = 0;
  $seconds  = 0;
```

5. After that we produce a breakdown of values that represent year, month, day, and so on:

```
if (is_string($date)) {
  list($dateParts, $timeParts) = explode(' ',
$date);
  list($year,$month,$day) = explode('-
',$dateParts);
  list($hour,$minutes,$seconds) =
explode(':',$timeParts);
} elseif (is_array($date)) {

list($year,$month,$day,$hour,$minutes,$seconds
) = $date;
} else {
  throw new
InvalidArgumentException(self::ERROR_ARGS_STRI
NG_ARRAY);
}
```

6. Next we create an `IntlCalendar` instance, which will serve as an argument when running `format()`. We set the date using the discreet integer values:

```
$intlDate =
IntlCalendar::createInstance($timeZone, $this-
```

```
>getLocaleCode());
$intlDate-
>set($year,$month,$day,$hour,$minutes,$seconds
);
```

7. Finally, we obtain the date formatter instance, and produce the result:

```
  $formatter = $this->getDateFormatter($type);
  if ($timeZone) {
    $formatter->setTimeZone($timeZone);
  }
  $result = $formatter->format($intlDate);
  return $result ??
self::ERROR_UNABLE_TO_FORMAT;
}
```

8. The `parseDate()` method is actually simpler than formatting. The only complication is what to do if the type is not specified (which will be the most likely case). All we need to do is to loop through all possible types (of which there are only four) until a result is produced:

```
public function parseDate($string, $type =
NULL)
{
 if ($type) {
  $result = $this->getDateFormatter($type)-
>parse($string);
 } else {
  $tryThese = [self::DATE_TYPE_FULL,
    self::DATE_TYPE_LONG,
    self::DATE_TYPE_MEDIUM,
    self::DATE_TYPE_SHORT];
  foreach ($tryThese as $type) {
  $result = $this->getDateFormatter($type)-
>parse($string);
    if ($result) {
      break;
    }
  }
 }
 return ($result) ? $result :
self::ERROR_UNABLE_TO_PARSE;
}
```

How it works...

Code the changes to `Application\I18n\Locale`, discussed previously. You can then create a test file, `chap_08_formatting_date.php`, which sets up autoloading, and creates two instances of the `Locale` class, one for the USA, the other for France:

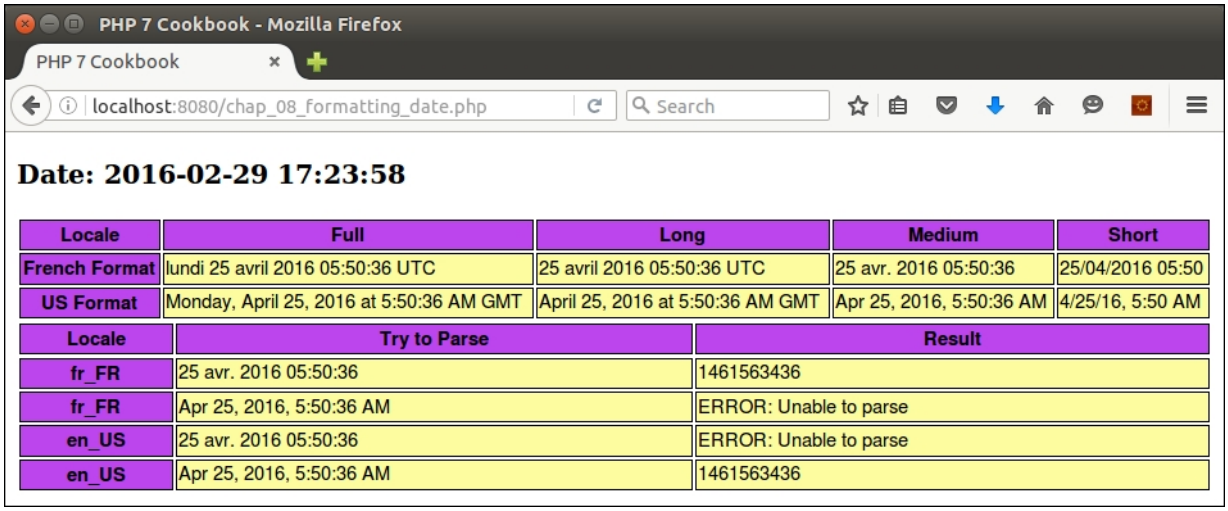
```
<?php
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
use Application\I18n\Locale;

$localeFr = new Locale('fr-FR');
$localeUs = new Locale('en_US');
$date     = '2016-02-29 17:23:58';
?>
```

Next, with suitable styling, run a test of `formatDate()` and `parseDate()`:

```
echo $localeFr->formatDate($date,
Locale::DATE_TYPE_FULL);
echo $localeUs->formatDate($date,
Locale::DATE_TYPE_MEDIUM);
$localeUs->parseDate($localeFr->formatDate($date,
Locale::DATE_TYPE_MEDIUM));
// etc.
```

An example of the output is shown here:



See also

- ISO 8601 gives precise definitions for all aspects of date and time. There is also an RFC that discusses the impact of ISO 8601 on the Internet. For reference, see https://tools.ietf.org/html/rfc3339. For a good overview of date formats by country, see https://en.wikipedia.org/wiki/Date_format_by_country.

Creating an HTML international calendar generator

Creating a program to display a calendar is something you would most likely do as a student at secondary school. A nested `for()` loop, where the inside loop generates a list of seven days, will generally suffice. Even the problem of how many days there are in the month is easily solved in the form of a simple array. Where it starts to get tricky is when you need to figure out, for any given year, on what day of the week does the 1st of January fall. Also, what if you want to represent the months and days of the week in a language and format acceptable to a specific locale? As you have probably guessed, we will build a solution using the previously discussed `Application\I18n\Locale` class.

How to do it...

1. First we need to create a generic class that will hold information for a single day. Initially it will only hold an integer value, `$dayOfMonth`. Later, in the next recipe, we'll expand it to include events. As the primary purpose of this class will be to yield `$dayOfMonth`, we'll incorporate this value into its constructor, and define `__invoke()` to return this value as well:

```
namespace Application\I18n;

class Day
{
  public $dayOfMonth;
  public function __construct($dayOfMonth)
  {
    $this->dayOfMonth = $dayOfMonth;
  }
  public function __invoke()
  {
    return $this->dayOfMonth ?? '';
  }
}
```

2. Create a new class that will hold the appropriate calendar-generation methods. It will accept an instance of `Application\I18n\Locale`, and will define a couple of class constants and properties. The format codes, such as `EEEEE` and `MMMM`, are drawn from ICU date formats:

```
namespace Application\I18n;

use IntlCalendar;

class Calendar
{

  const DAY_1 = 'EEEEE';  // T
  const DAY_2 = 'EEEEEE'; // Tu
  const DAY_3 = 'EEE';   // Tue
  const DAY_FULL = 'EEEE'; // Tuesday
  const MONTH_1 = 'MMMMM'; // M
  const MONTH_3 = 'MMM';  // Mar
  const MONTH_FULL = 'MMMM';  // March
  const DEFAULT_ACROSS = 3;
  const HEIGHT_FULL = '150px';
  const HEIGHT_SMALL = '60px';

  protected $locale;
  protected $dateFormatter;
  protected $yearArray;
  protected $height;

  public function __construct(Locale $locale)
  {
    $this->locale = $locale;
  }

     // other methods are discussed in the
following bullets

}
```

3. Then we define a method that returns an `IntlDateFormatter` instance from our `locale` class. This is stored in a class property, as it will be used frequently:

```
protected function getDateFormatter()
{
```

```
 if (!$this->dateFormatter) {
  $this->dateFormatter = $this->locale-
>getDateFormatter(Locale::DATE_TYPE_FULL);
 }
 return $this->dateFormatter;
}
```

4. Next we define a core method, `buildMonthArray()`, which creates a multi-dimensional array where the outer key is the week of the year, and the inner array is seven elements representing the days of the week. We accept the year, month, and optional time zone as arguments. Note, as part of variable initialization, we subtract 1 from the month. This is because the `IntlCalendar::set()` method expects a 0-based value for the month, where 0 represents January, 1 is February, and so on:

```
public function buildMonthArray($year, $month,
$timeZone = NULL)
{
$month -= 1;
//IntlCalendar months are 0 based; Jan==0,
Feb==1 and so on
  $day = 1;
  $first = TRUE;
  $value = 0;
  $monthArray = array();
```

5. We then create an `IntlCalendar` instance, and use it to determine how many days are in this month:

```
$cal = IntlCalendar::createInstance($timeZone,
$this->locale->getLocaleCode());
$cal->set($year, $month, $day);
$maxDaysInMonth = $cal-
>getActualMaximum(IntlCalendar::FIELD_DAY_OF_M
ONTH);
```

6. After that we use our `IntlDateFormatter` instance to determine what day of the week equates to the 1st of this month. After that, we set the pattern to `w`, which will subsequently give us the week number:

```
$formatter = $this->getDateFormatter();
$formatter->setPattern('e');
```

```
$firstDayIsWhatDow = $formatter->format($cal);
```

7. We are now ready to loop through all days in the month with nested loops. An outer `while()` loop ensures we don't go past the end of the month. The inner loop represents the days of the week. You will note that we take advantage of `IntlCalendar::get()`, which allows us to retrieve values from a wide range of predefined fields. We also adjust the week of the year value to 0 if it exceeds 52:

```
while ($day <= $maxDaysInMonth) {
  for ($dow = 1; $dow <= 7; $dow++) {
    $cal->set($year, $month, $day);
    $weekOfYear = $cal-
>get(IntlCalendar::FIELD_WEEK_OF_YEAR);
    if ($weekOfYear > 52) $weekOfYear = 0;
```

8. We then check to see whether `$first` is still set `TRUE`. If so, we start adding day numbers to the array. Otherwise, the array value is set to `NULL`. We then close all open statements and return the array. Note that we also need to make sure the inner loop doesn't go past the number of days in the month, hence the extra `if()` statement in the outer `else` clause.

> ## Note
>
> Note that instead of just storing the value for the day of the month, we use the newly defined `Application\I18n\Day` class.

```
if ($first) {
  if ($dow == $firstDayIsWhatDow) {
    $first = FALSE;
    $value = $day++;
  } else {
    $value = NULL;
  }
} else {
  if ($day <= $maxDaysInMonth) {
```

```
            $value = $day++;
          } else {
            $value = NULL;
          }
        }
        $monthArray[$weekOfYear][$dow] = new
Day($value);
      }
    }
    return $monthArray;
}
```

Refining internationalized output

1. First, a series of small methods, starting with one that extracts the internationally formatted day based on type. The type determines whether we deliver the full name of the day, an abbreviation, or just a single letter, all appropriate for that locale:

```
protected function getDay($type, $cal)
{
  $formatter = $this->getDateFormatter();
  $formatter->setPattern($type);
  return $formatter->format($cal);
}
```

2. Next we need a method that returns an HTML row of day names, calling the newly defined `getDay()` method. As mentioned previous, the type dictates the appearance of the days:

```
protected function getWeekHeaderRow($type,
$cal, $year, $month, $week)
{
  $output = '<tr>';
  $width  = (int) (100/7);
  foreach ($week as $day) {
    $cal->set($year, $month, $day());
    $output .= '<th style="vertical-
align:top;" width="' . $width . '%">' . $this-
>getDay($type, $cal) . '</th>';
  }
  $output .= '</tr>' . PHP_EOL;
```

```
    return $output;
}
```

3. After that, we define a very simple method to return a row of week dates. Note that we take advantage of `Day::__invoke()` using: `$day()`:

```
protected function getWeekDaysRow($week)
{
  $output = '<tr style="height:' . $this-
>height . ';">';
  $width  = (int) (100/7);
  foreach ($week as $day) {
    $output .= '<td style="vertical-
align:top;" width="' . $width . '%">' . $day()
.  '</td>';
  }
  $output .= '</tr>' . PHP_EOL;
  return $output;
}
```

4. And finally, a method that puts the smaller methods together to generate a calendar for a single month. First we build the month array, but only if `$yearArray` is not already available:

```
public function calendarForMonth($year,
    $month,
    $timeZone = NULL,
    $dayType = self::DAY_3,
    $monthType = self::MONTH_FULL,
    $monthArray = NULL)
{
  $first = 0;
  if (!$monthArray)
    $monthArray = $this->yearArray[$year]
[$month]
    ?? $this->buildMonthArray($year, $month,
$timeZone);
```

5. The month needs to be decremented by `1` as `IntlCalendar` months are 0-based: Jan = 0, Feb = 1, and so on. We then build an `IntlCalendar` instance using the time zone (if any), and the locale. We next create a `IntlDateFormatter` instance to retrieve the month name and other information according to locale:

```
    $month--;
    $cal =
IntlCalendar::createInstance($timeZone, $this-
>locale->getLocaleCode());
    $cal->set($year, $month, 1);
    $formatter = $this->getDateFormatter();
    $formatter->setPattern($monthType);
```

6. We then loop through the month array, and call the smaller methods just mentioned to build the final output:

```
    $this->height = ($dayType == self::DAY_FULL)
       ? self::HEIGHT_FULL : self::HEIGHT_SMALL;
    $html = '<h1>' . $formatter->format($cal) .
'</h1>';
    $header = '';
    $body   = '';
    foreach ($monthArray as $weekNum => $week) {
      if ($first++ == 1) {
        $header .= $this-
>getWeekHeaderRow($dayType, $cal, $year,
$month, $week);
      }
      $body .= $this->getWeekDaysRow($dayType,
$week);
    }
    $html .= '<table>' . $header . $body .
'</table>' . PHP_EOL;
    return $html;
}
```

7. In order to generate a calendar for the entire year, it's a simple matter of looping through months 1 to 12. To facilitate outside access, we first define a method that builds a year array:

```
public function buildYearArray($year,
$timeZone = NULL)
{
    $this->yearArray = array();
    for ($month = 1; $month <= 12; $month++) {
      $this->yearArray[$year][$month] = $this-
>buildMonthArray($year, $month, $timeZone);
    }
    return $this->yearArray;
}
```

```
public function getYearArray()
{
  return $this->yearArray;
}
```

8. To generate a calendar for a year, we define a method, `calendarForYear()`. If the year array has not been build, we call `buildYearArray()`. We take into account how many monthly calendars we wish to display across and then call `calendarForMonth()`:

```
public function calendarForYear($year,
  $timeZone = NULL,
  $dayType = self::DAY_1,
  $monthType = self::MONTH_3,
  $across = self::DEFAULT_ACROSS)
{
  if (!$this->yearArray) $this-
>buildYearArray($year, $timeZone);
  $yMax = (int) (12 / $across);
  $width = (int) (100 / $across);
  $output = '<table>' . PHP_EOL;
  $month = 1;
  for ($y = 1; $y <= $yMax; $y++) {
    $output .= '<tr>';
    for ($x = 1; $x <= $across; $x++) {
      $output .= '<td style="vertical-
align:top;" width="' . $width . '%">' . $this-
>calendarForMonth($year, $month, $timeZone,
$dayType, $monthType, $this->yearArray[$year]
[$month++]) . '</td>';
    }
    $output .= '</tr>' . PHP_EOL;
  }
  $output .= '</table>';
  return $output;
}
```

How it works...

First of all, make sure you build the `Application\I18n\Locale` class as defined in the previous recipe. After that, create a new file,

`Calendar.php`, in the `Application\I18n` folder, with all the methods described in this recipe.

Next, define a calling program, `chap_08_html_calendar.php`, which sets up autoloading and creates `Locale` and `Calendar` instances. Also be sure to define the year and month:

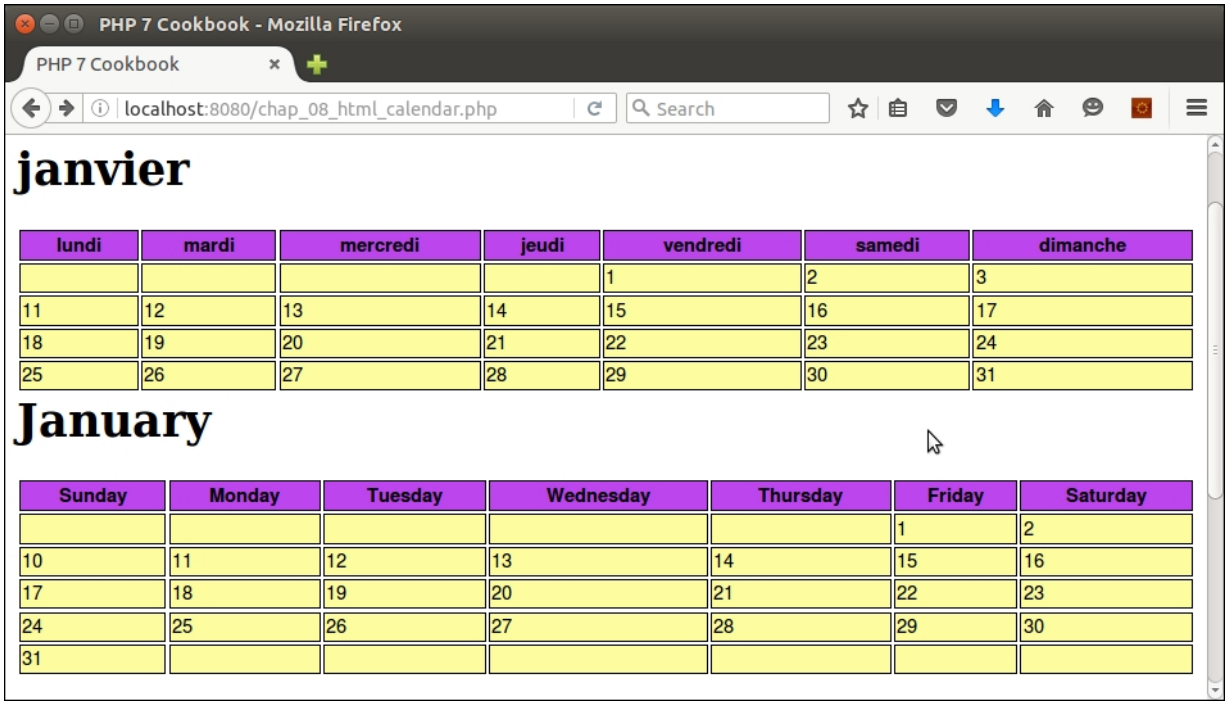
```
<?php
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
use Application\I18n\Locale;
use Application\I18n\Calendar;

$localeFr = new Locale('fr-FR');
$localeUs = new Locale('en_US');
$localeTh = new Locale('th_TH');
$calendarFr = new Calendar($localeFr);
$calendarUs = new Calendar($localeUs);
$calendarTh = new Calendar($localeTh);
$year = 2016;
$month = 1;
?>
```

You can then develop appropriate view logic to display the different calendars. For example, you can include parameters to display the full month and day names:

```
<!DOCTYPE html>
<html>
  <head>
  <title>PHP 7 Cookbook</title>
  <meta http-equiv="content-type"
content="text/html;charset=utf-8" />
  <link rel="stylesheet" type="text/css"
href="php7cookbook_html_table.css">
  </head>
  <body>
    <h3>Year: <?= $year ?></h3>
    <?= $calendarFr->calendarForMonth($year,
$month, NULL, Calendar::DAY_FULL); ?>
    <?= $calendarUs->calendarForMonth($year,
$month, NULL, Calendar::DAY_FULL); ?>
```

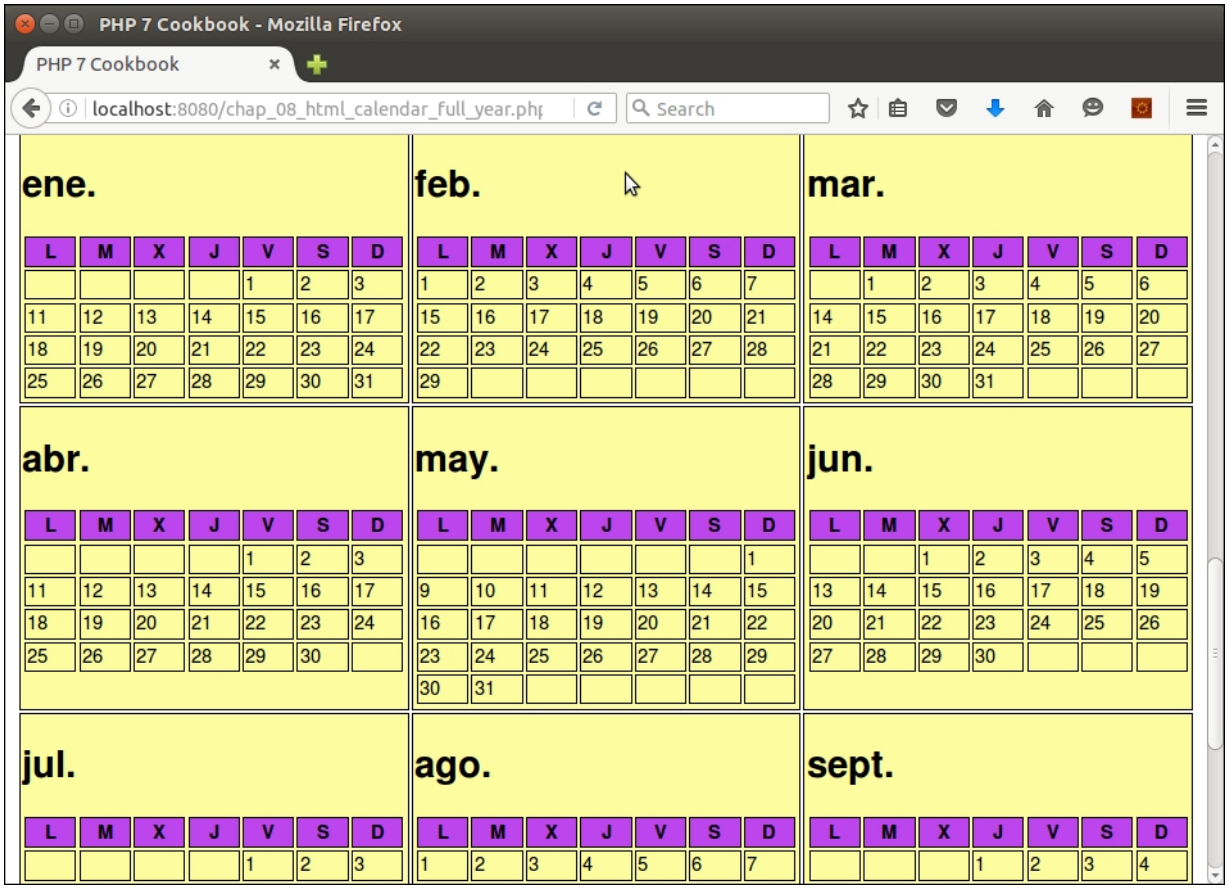
```
    <?= $calendarTh->calendarForMonth($year,
$month, NULL, Calendar::DAY_FULL); ?>
  </body>
</html>
```



With a couple of modifications, you can also display a calendar for the entire year:

```
$localeTh = new Locale('th_TH');
$localeEs = new Locale('es_ES');
$calendarTh = new Calendar($localeTh);
$calendarEs = new Calendar($localeEs);
$year = 2016;
echo $calendarTh->calendarForYear($year);
echo $calendarEs->calendarForYear($year);
```

Here is the browser output showing a full year calendar in Spanish:



See also

- For more information on codes used by `IntlDateFormatter::setPattern()`, see this article: http://userguide.icu-project.org/formatparse/datetime

Building a recurring events generator

A very common need related to generating a calendar is the scheduling of events. Events can be in the form of *one-off* events, which take place on one day, or on a weekend. There is a much greater need, however, to track events that are *recurring*. We need to account for the start date, the recurring interval (daily, weekly, monthly), and the number of occurrences or a specific end date.

How to do it...

1. Before anything else, it would be an excellent idea to create a class that represents an event. Ultimately you'll probably end up storing the data in such a class in a database. For this illustration, however, we will simply define the class, and leave the database aspect to your imagination. You will notice that we will use a number of classes included in the `DateTime` extension admirably suited to event generation:

```
namespace Application\I18n;

use DateTime;
use DatePeriod;
use DateInterval;
use InvalidArgumentException;

class Event
{
  // code
}
```

2. Next, we define a series of useful class constants and properties. You will notice that we defined most of the properties `public` in order to economize on the number of getters and setters needed. The intervals are defined as `sprintf()` format strings; `%d` will be substituted for a value:

```
const INTERVAL_DAY = 'P%dD';
const INTERVAL_WEEK = 'P%dW';
const INTERVAL_MONTH = 'P%dM';
const FLAG_FIRST = 'FIRST';    // 1st of the
month
const ERROR_INVALID_END  = 'Need to supply
either # occurrences or an end date';
const ERROR_INVALID_DATE = 'String i.e. YYYY-
mm-dd or DateTime instance only';
const ERROR_INVALID_INTERVAL = 'Interval must
take the form "P\d+(D | W | M)"';

public $id;
public $flag;
public $value;
public $title;
public $locale;
public $interval;
public $description;
public $occurrences;
public $nextDate;
protected $endDate;
protected $startDate;
```

3. Next we turn our attention to the constructor. We need to collect and set all information pertinent to an event. The variable names are self-explanatory.

> ## Note
>
> `$value` is not quite so clear. This parameter will ultimately be substituted for the value in the interval format string. So, for example, if the user selects `$interval` as `INTERVAL_DAY`, and `$value` as `2`, the resulting interval string will be `P2D`, which means every other day (or every 2nd day).

```
public function __construct($title,
    $description,
    $startDate,
    $interval,
    $value,
```

```
        $occurrences = NULL,
        $endDate = NULL,
        $flag = NULL)
    {
```

4. We then initialize variables. Note that the ID is pseudo-randomly generated, but might ultimately end up being the primary key in a database `events` table. Here we use `md5()` not for security purposes, but rather to quickly generate a hash so that IDs have a consistent appearance:

```
$this->id = md5($title . $interval . $value) .
sprintf('%04d', rand(0,9999));
$this->flag = $flag;
$this->value = $value;
$this->title = $title;
$this->description = $description;
$this->occurrences = $occurrences;
```

5. As mentioned previously, the interval parameter is a `sprintf()` pattern used to construct a proper `DateInterval` instance:

```
try {
  $this->interval = new
DateInterval(sprintf($interval, $value));
  } catch (Exception $e) {
  error_log($e->getMessage());
  throw new
InvalidArgumentException(self::ERROR_INVALID_I
NTERVAL);
}
```

6. To initialize `$startDate`, we call `stringOrDate()`. We then attempt to generate a value for `$endDate` by calling either `stringOrDate()` or `calcEndDateFromOccurrences()`. If we have neither an end date nor a number of occurrences, an exception is thrown:

```
  $this->startDate = $this-
>stringOrDate($startDate);
  if ($endDate) {
    $this->endDate = $this-
>stringOrDate($endDate);
  } elseif ($occurrences) {
    $this->endDate = $this-
>calcEndDateFromOccurrences();
```

```
    } else {
    throw new
InvalidArgumentException(self::ERROR_INVALID_E
ND);
    }
    $this->nextDate = $this->startDate;
}
```

7. The `stringOrDate()` method consists of a few lines of code that check the data type of the date variable, and return a `DateTime` instance or `NULL`:

```
protected function stringOrDate($date)
{
    if ($date === NULL) {
      $newDate = NULL;
    } elseif ($date instanceof DateTime) {
      $newDate = $date;
    } elseif (is_string($date)) {
      $newDate = new DateTime($date);
    } else {
      throw new
InvalidArgumentException(self::ERROR_INVALID_E
ND);
    }
    return $newDate;
}
```

8. We call the `calcEndDateFromOccurrences()` method from the constructor if `$occurrences` is set so that we'll know the end date for this event. We take advantage of the `DatePeriod` class, which provides an iteration based on a start date, `DateInterval`, and number of occurrences:

```
protected function
calcEndDateFromOccurrences()
{
    $endDate = new DateTime('now');
    $period = new DatePeriod(
$this->startDate, $this->interval, $this-
>occurrences);
    foreach ($period as $date) {
      $endDate = $date;
    }
    return $endDate;
}
```

9. Next we throw in a `__toString()` magic method, which simple echoes the title of the event:

```
public function __toString()
{
  return $this->title;
}
```

10. The last method we need to define for our `Event` class is `getNextDate()`, which is used when generating a calendar:

```
public function  getNextDate(DateTime $today)
{
  if ($today > $this->endDate) {
    return FALSE;
  }
  $next = clone $today;
  $next->add($this->interval);
  return $next;
}
```

11. Next we turn our attention to the `Application\I18n\Calendar` class described in the previous recipe. With a bit of minor surgery, we are ready to tie our newly defined `Event` class into the calendar. First we add a new property, `$events`, and a method to add events in the form of an array. We use the `Event::$id` property to make sure events are merged and not overwritten:

```
protected $events = array();
public function addEvent(Event $event)
{
  $this->events[$event->id] = $event;
}
```

12. Next we add a method, `processEvents()`, which adds an `Event` instance to a `Day` object when building the year calendar. First we check to see whether there are any events, and whether or not the `Day` object is `NULL`. As you may recall, it's likely that the first day of the month doesn't fall on the first day of the week, and thus the need to set the value of a `Day` object to `NULL`. We certainly do not want to add events to a non-operative day! We

then call `Event::getNextDate()` and see whether the dates match. If so, we store the `Event` into `Day::$events[]` and set the next date on the `Event` object:

```
protected function processEvents($dayObj,
$cal)
{
  if ($this->events && $dayObj()) {
    $calDateTime = $cal->toDateTime();
    foreach ($this->events as $id =>
$eventObj) {
      $next = $eventObj-
>getNextDate($eventObj->nextDate);
      if ($next) {
        if ($calDateTime->format('Y-m-d') ==
            $eventObj->nextDate->format('Y-m-
d')) {
          $dayObj->events[$eventObj->id] =
$eventObj;
          $eventObj->nextDate = $next;
        }
      }
    }
  }
  return $dayObj;
}
```

Note

Note that we do not do a direct comparison of the two objects. Two reasons for this: first of all, one is a `DateTime` instance, the other is an `IntlCalendar` instance. The other, more compelling reason, is that it's possible that hours:minutes:seconds were included when the `DateTime` instance was obtained, resulting in actual value differences between the two objects.

13. Now we need to add a call to `processEvents()` in the `buildMonthArray()` method so that it looks like this:

```
  while ($day <= $maxDaysInMonth) {
    for ($dow = 1; $dow <= 7; $dow++) {
      // add this to the existing code:
      $dayObj = $this->processEvents(new
Day($value), $cal);
      $monthArray[$weekOfYear][$dow] =
$dayObj;
    }
  }
```

14. Finally, we need to modify `getWeekDaysRow()`, adding the necessary code to output event information inside the box along with the date:

```
protected function getWeekDaysRow($type,
$week)
{
  $output = '<tr style="height:' . $this-
>height . ';">';
  $width  = (int) (100/7);
  foreach ($week as $day) {
    $events = '';
    if ($day->events) {
      foreach ($day->events as $single) {
        $events .= '<br>' . $single->title;
        if ($type == self::DAY_FULL) {
          $events .= '<br><i>' . $single-
>description . '</i>';
        }
      }
    }
    $output .= '<td style="vertical-
align:top;" width="' . $width . '%">'
  . $day() . $events . '</td>';
  }
  $output .= '</tr>' . PHP_EOL;
  return $output;
}
```

How it works...

To tie events to the calendar, first code the `Application\I18n\Event` class described in steps 1 to 10. Next, modify `Application\I18n\Calendar` as described in steps 11 to 14. You can

then create a test script, `chap_08_recurring_events.php`, which sets up autoloading and creates `Locale` and `Calendar` instances. For the purposes of illustration, go ahead and use '`es_ES`' as a locale:

```
<?php
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
use Application\I18n\ { Locale, Calendar, Event };

try {
  $year = 2016;
  $localeEs = new Locale('es_ES');
  $calendarEs = new Calendar($localeEs);
```

Now we can start defining and adding events to the calendar. The first example adds an event that lasts 3 days and starts on 8 January 2016:

```
  // add event: 3 days
  $title = 'Conf';
  $description = 'Special 3 day symposium on eco-
waste';
  $startDate = '2016-01-08';
  $event = new Event($title, $description,
$startDate,
                     Event::INTERVAL_DAY, 1, 2);
  $calendarEs->addEvent($event);
```

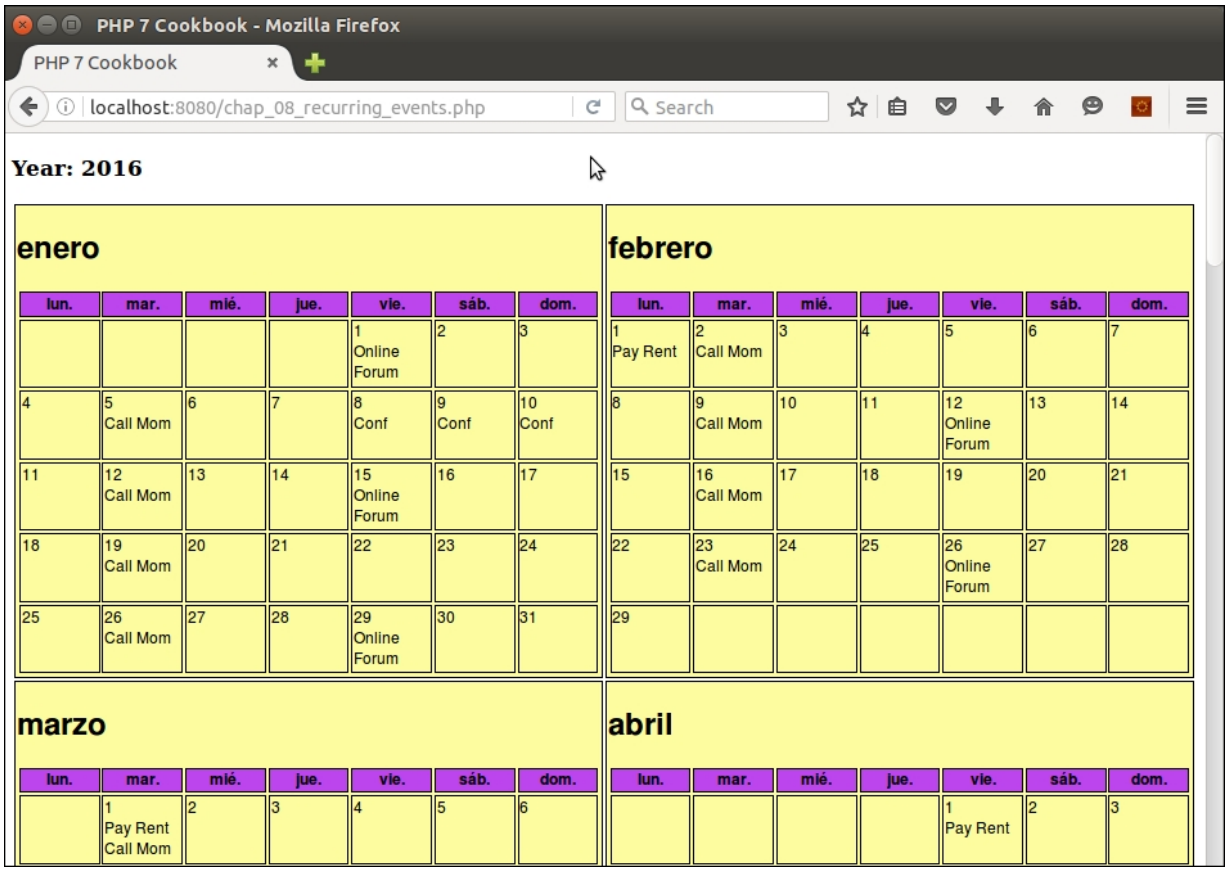
Here is another example, an event that occurs on the first of every month until September 2017:

```
  $title = 'Pay Rent';
  $description = 'Sent rent check to landlord';
  $startDate = new DateTime('2016-02-01');
  $event = new Event($title, $description,
$startDate,
    Event::INTERVAL_MONTH, 1, '2017-09-01', NULL,
Event::FLAG_FIRST);
  $calendarEs->addEvent($event);
```

You can then add sample weekly, bi-weekly, monthly, and so on events as desired. You can then close the `try...catch` block, and produce suitable display logic:

```
} catch (Throwable $e) {
  $message = $e->getMessage();
}
?>
<!DOCTYPE html>
<head>
  <title>PHP 7 Cookbook</title>
  <meta http-equiv="content-type"
content="text/html;charset=utf-8" />
  <link rel="stylesheet" type="text/css"
href="php7cookbook_html_table.css">
</head>
<body>
<h3>Year: <?= $year ?></h3>
<?= $calendarEs->calendarForYear($year,
'Europe/Berlin',
    Calendar::DAY_3, Calendar::MONTH_FULL, 2); ?>
<?= $calendarEs->calendarForMonth($year, 1  ,
'Europe/Berlin',
    Calendar::DAY_FULL); ?>
</body>
</html>
```

Here is the output showing the first few months of the year:



See also

- For more information on `IntlCalendar` field constants that can be used with `get()`, please refer to this page: http://php.net/manual/en/class.intlcalendar.php#intlcalendar.constants

Handling translation without gettext

Translation is an important part of making your website accessible to an international customer base. One way this is accomplished it to use the PHP `gettext` functions, which are based on the **GNU** `gettext` operating system tools installed on the local server. `gettext` is well documented and well supported, but uses a legacy approach and has distinct disadvantages. Accordingly, in this recipe, we present an alternative approach to translation where you can build your own *adapter*.

Something important to recognize is that the programmatic translation tools available to PHP are primarily designed to provide limited translation of a word or phrase, referred to as the **msgid** (**message ID**). The translated equivalent is referred to as the **msgstr** (**message string**). Accordingly, incorporating translation typically only involves relatively unchanging items such as menus, forms, error or success messages, and so on. For the purposes of this recipe, we will assume that you have the actual web page translations stored as blocks of text.

> ## Note
>
> If you need to translate entire pages of content, you might consider using the *Google Translate API*. This is, however, a paid service. Alternatively, you could outsource the translation to individuals with multi-lingual skills cheaply using *Amazon Mechanical Turk*. See the *See Also* section at the end of this recipe for the URLs.

How to do it...

1. We will once again use the Adapter software design pattern, in this case to provide alternatives to the translation source. In this recipe, we will demonstrate adapters for `.ini` files, `.csv` files, and databases.
2. To begin, we will define an interface that will later be used to identify a translation adapter. The requirements for a translation adapter are quite simple, we only need to return a message string for a given message ID:

```
namespace Application\I18n\Translate\Adapter;
interface TranslateAdapterInterface
{
  public function translate($msgid);
}
```

3. Next we define a trait that matches the interface. The trait will contain the actual code required. Note that if we fail to find the message string, we simply return the message ID:

```
namespace Application\I18n\Translate\Adapter;

trait TranslateAdapterTrait
{
  protected $translation;
  public function translate($msgid)
  {
    return $this->translation[$msgid] ??
$msgid;
  }
}
```

4. Now we're ready to define our first adapter. In this recipe, we'll start with an adapter that uses an `.ini` file as the source of translations. The first thing you'll notice is that we use the trait defined previously. The constructor method will vary between adapters. In this case, we use `parse_ini_file()` to produce an array of key/value pairs where the key is the message ID. Notice that we use the `$filePattern` parameter to substitute the locale, which then allows us to load the appropriate translation file:

```
namespace Application\I18n\Translate\Adapter;

use Exception;
```

```
use Application\I18n\Locale;

class Ini implements TranslateAdapterInterface
{
  use TranslateAdapterTrait;
  const ERROR_NOT_FOUND = 'Translation file
not found';
  public function __construct(Locale $locale,
$filePattern)
  {
    $translateFileName = sprintf($filePattern,
$locale->getLocaleCode());
    if (!file_exists($translateFileName)) {
      error_log(self::ERROR_NOT_FOUND . ':' .
$translateFileName);
      throw new
Exception(self::ERROR_NOT_FOUND);
    } else {
      $this->translation =
parse_ini_file($translateFileName);
    }
  }
}
```

5. The next adapter, `Application\I18n\Translate\Adapter\Csv`, is identical, except that we open the translation file and loop through using `fgetcsv()` to retrieve the message ID / message string key pairs. Here we show only the difference in the constructor:

```
public function __construct(Locale $locale,
$filePattern)
{
  $translateFileName = sprintf($filePattern,
$locale->getLocaleCode());
  if (!file_exists($translateFileName)) {
    error_log(self::ERROR_NOT_FOUND . ':' .
$translateFileName);
    throw new
Exception(self::ERROR_NOT_FOUND);
  } else {
    $fileObj = new
SplFileObject($translateFileName, 'r');
    while ($row = $fileObj->fgetcsv()) {
      $this->translation[$row[0]] = $row[1];
    }
```

```
    }
}
```

> **Note**
>
> The big disadvantage of both of these adapters is that we need to preload the entire translation set, which puts a strain on memory if there is a large number of translations. Also, the translation file needs to be opened and parsed, which drags down performance.

6. We now present the third adapter, which performs a database lookup and avoids the problems of the other two adapters. We use a `PDO` prepared statement which is sent to the database in the beginning, and only one time. We then execute as many times as needed, supplying the message ID as an argument. You will also notice that we needed to override the `translate()` method defined in the trait. Finally, you might have noticed the use of `PDOStatement::fetchColumn()` as we only need the one value:

```
namespace Application\I18n\Translate\Adapter;

use Exception;
use Application\Database\Connection;
use Application\I18n\Locale;

class Database implements
TranslateAdapterInterface
{
  use TranslateAdapterTrait;
  protected $connection;
  protected $statement;
  protected $defaultLocaleCode;
  public function __construct(Locale $locale,
                              Connection
$connection,
                              $tableName)
  {
```

```
        $this->defaultLocaleCode = $locale-
>getLocaleCode();
        $this->connection = $connection;
        $sql = 'SELECT msgstr FROM ' . $tableName
            . ' WHERE localeCode = ? AND msgid =
?';
        $this->statement = $this->connection->pdo-
>prepare($sql);
    }
    public function translate($msgid,
$localeCode = NULL)
    {
        if (!$localeCode) $localeCode = $this-
>defaultLocaleCode;
        $this->statement->execute([$localeCode,
$msgid]);
        return $this->statement->fetchColumn();
    }
}
```

7. We are now ready to define the core `Translation` class, which is tied to one (or more) adapters. We assign a class constant to represent the default locale, and properties for the locale, adapter, and text file pattern (explained later):

```
namespace Application\I18n\Translate;

use Application\I18n\Locale;
use
Application\I18n\Translate\Adapter\TranslateAd
apterInterface;

class Translation
{
    const DEFAULT_LOCALE_CODE = 'en_GB';
    protected $defaultLocaleCode;
    protected $adapter = array();
    protected $textFilePattern = array();
```

8. In the constructor, we determine the locale, and set the initial adapter to this locale. In this manner, we are able to host multiple adapters:

```
public function
__construct(TranslateAdapterInterface
$adapter,
```

```
                $defaultLocaleCode = NULL,
                $textFilePattern = NULL)
{
  if (!$defaultLocaleCode) {
    $this->defaultLocaleCode =
self::DEFAULT_LOCALE_CODE;
  } else {
    $this->defaultLocaleCode =
$defaultLocaleCode;
  }
  $this->adapter[$this->defaultLocaleCode] =
$adapter;
  $this->textFilePattern[$this-
>defaultLocaleCode] = $textFilePattern;
}
```

9. Next we define a series of setters, which gives us more flexibility:

```
public function setAdapter($localeCode,
TranslateAdapterInterface $adapter)
{
  $this->adapter[$localeCode] = $adapter;
}
public function
setDefaultLocaleCode($localeCode)
{
  $this->defaultLocaleCode = $localeCode;
}
public function
setTextFilePattern($localeCode, $pattern)
{
  $this->textFilePattern[$localeCode] =
$pattern;
}
```

10. We then define the PHP magic method `__invoke()`, which lets us make a direct call to the translator instance, returning the message string given the message ID:

```
public function __invoke($msgid, $locale =
NULL)
{
  if ($locale === NULL) $locale = $this-
>defaultLocaleCode;
  return $this->adapter[$locale]-
```

```
>translate($msgid);
}
```

11. Finally, we also add a method that can return translated blocks of text from text files. Bear in mind that this could be modified to use a database instead. We did not include this functionality in the adapter, as its purpose is completely different; we just want to return large blocks of code given a key, which could conceivably be the filename of the translated text file:

```
public function text($key, $localeCode = NULL)
{
  if ($localeCode === NULL) $localeCode =
$this->defaultLocaleCode;
  $contents = $key;
  if (isset($this-
>textFilePattern[$localeCode])) {
    $fn = sprintf($this-
>textFilePattern[$localeCode], $localeCode,
$key);
    if (file_exists($fn)) {
      $contents = file_get_contents($fn);
    }
  }
  return $contents;
}
```

How it works...

First you will need to define a directory structure to house the translation files. For the purposes of this illustration, you can make a directory ,`/path/to/project/files/data/languages`. Under this directory structure, create sub-directories that represent different locales. For this illustration, you could use these: `de_DE`, `fr_FR`, `en_GB`, and `es_ES`, representing German, French, English, and Spanish.

Next you will need to create the different translation files. As an example, here is a representative `data/languages/es_ES/translation.ini` file in Spanish:

```
Welcome=Bienvenido
About Us=Sobre Nosotros
Contact Us=Contáctenos
Find Us=Encontrarnos
click=clic para más información
```

Likewise, to demonstrate the CSV adapter, create the same thing as a CSV file, `data/languages/es_ES/translation.csv`:

```
"Welcome","Bienvenido"
"About Us","Sobre Nosotros"
"Contact Us","Contáctenos"
"Find Us","Encontrarnos"
"click","clic para más información"
```

Finally, create a database table, `translation`, and populate it with the same data. The main difference is that the database table will have three fields: `msgid`, `msgstr`, and `locale_code`:

```
CREATE TABLE `translation` (
  `msgid` varchar(255) NOT NULL,
  `msgstr` varchar(255) NOT NULL,
  `locale_code` char(6) NOT NULL DEFAULT '',
  PRIMARY KEY (`msgid`,`locale_code`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

Next, define the classes mentioned previously, using the code shown in this recipe:

- `Application\I18n\Translate\Adapter\TranslateAdapterInterface`
- `Application\I18n\Translate\Adapter\TranslateAdapterTrait`
- `Application\I18n\Translate\Adapter\Ini`
- `Application\I18n\Translate\Adapter\Csv`
- `Application\I18n\Translate\Adapter\Database`
- `Application\I18n\Translate\Translation`

Now you can create a test file, `chap_08_translation_database.php`, to test the database translation adapter. It should implement

autoloading, use the appropriate classes, and create a `Locale` and `Connection` instance. Note that the `TEXT_FILE_PATTERN` constant is a `sprintf()` pattern in which the locale code and filename are substituted:

```
<?php
define('DB_CONFIG_FILE',
'/../config/db.config.php');
define('TEXT_FILE_PATTERN', __DIR__ .
'/../data/languages/%s/%s.txt');
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
use Application\I18n\Locale;
use Application\I18n\Translate\ { Translation,
Adapter\Database };
use Application\Database\Connection;

$conn = new Connection(include __DIR__ .
DB_CONFIG_FILE);
$locale = new Locale('fr_FR');
```

Next, create a translation adapter instance and use that to create a `Translation` instance:

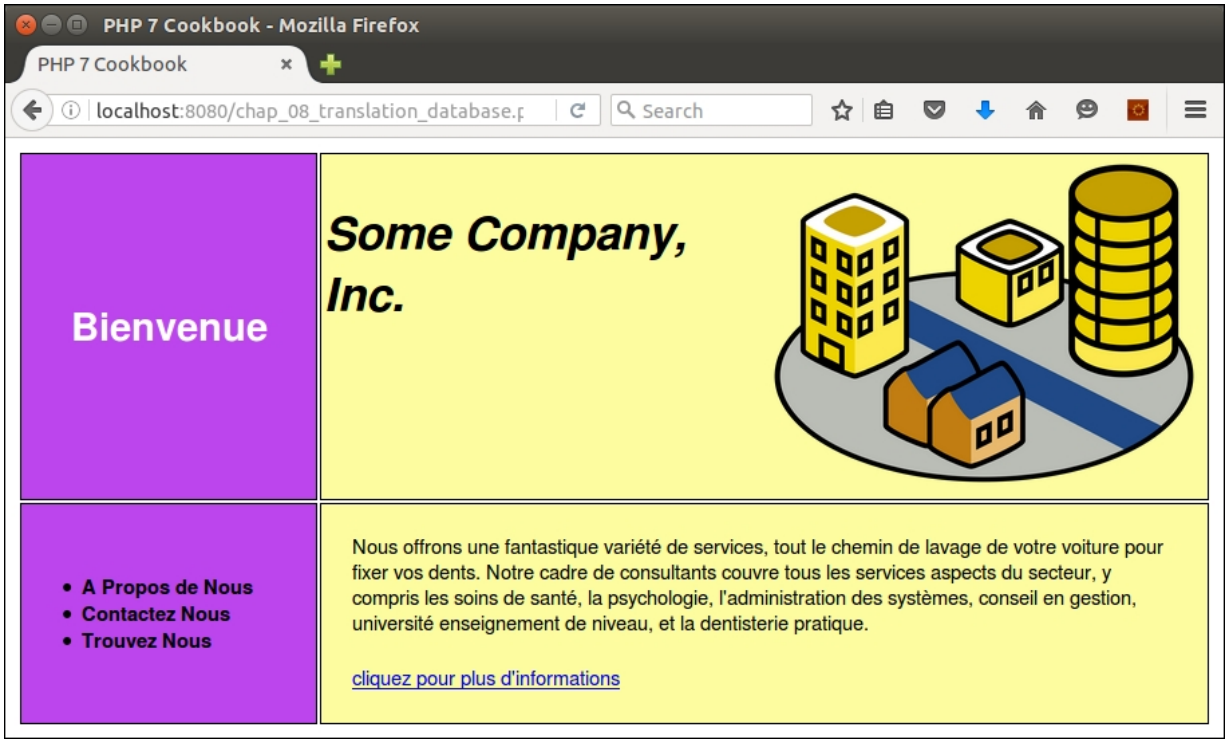
```
$adapter = new Database($locale, $conn,
'translation');
$translate = new Translation($adapter, $locale-
>getLocaleCode(), TEXT_FILE_PATTERN);
?>
```

Finally, create display logic that uses the `$translate` instance:

```
<!DOCTYPE html>
<head>
  <title>PHP 7 Cookbook</title>
  <meta http-equiv="content-type"
content="text/html;charset=utf-8" />
  <link rel="stylesheet" type="text/css"
href="php7cookbook_html_table.css">
</head>
<body>
```

```
<table>
<tr>
  <th><h1 style="color:white;"><?=
$translate('Welcome') ?></h1></th>
  <td>
    <div style="float:left;width:50%;vertical-
align:middle;">
    <h3 style="font-size:24pt;"><i>Some Company,
Inc.</i></h3>
    </div>
    <div style="float:right;width:50%;">
    <img src="jcartier-city.png" width="300px"/>
    </div>
  </td>
</tr>
<tr>
  <th>
    <ul>
      <li><?= $translate('About Us') ?></li>
      <li><?= $translate('Contact Us') ?></li>
      <li><?= $translate('Find Us') ?></li>
    </ul>
  </th>
  <td>
    <p>
    <?= $translate->text('main_page'); ?>
    </p>
    <p>
    <a href="#"><?= $translate('click') ?></a>
    </p>
  </td>
</tr>
</table>
</body>
</html>
```

You can then perform additional similar tests, substituting a new locale to get a different language, or using another adapter to test a different data source. Here is an example of output using a locale of `fr_FR` and the database translation adapter:



See also

- For more information on the Google Translation API, see https://cloud.google.com/translate/v2/translating-text-with-rest.
- For more information on Amazon Mechanical Turk, see https://www.mturk.com/mturk/welcome. For more information on `gettext`, see http://www.gnu.org/software/gettext/manual/gettext.html.

Chapter 9. Developing Middleware

In this chapter, we will cover the following topics:

- Authenticating with middleware
- Using middleware to implement access control
- Improving performance using the cache
- Implementing routing
- Making inter-framework system calls
- Using middleware to cross languages

Introduction

As often happens in the IT industry, terms get invented, and then used and abused. The term **middleware** is no exception. Arguably the first use of the term came out of the **Internet Engineering Task Force** (**IETF**) in the year 2000. Originally, the term was applied to any software which operates between the transport (that is, TCP/IP) and the application layer. More recently, especially with the acceptance of **PHP Standard Recommendation number 7** (**PSR-7**), middleware, specifically in the PHP world, has been applied to the web client-server environment.

> ## Note
>
> The recipes in this section will make use of the concrete classes defined in Appendix, *Defining PSR-7 Classes*.

Authenticating with middleware

One very important usage of middleware is to provide authentication. Most web-based applications need the ability to verify a visitor via username and password. By incorporating PSR-7 standards into an authentication class, you will make it generically useful across the board, so to speak, being secure enough that it can be used in any framework that provides PSR-7-compliant request and response objects.

How to do it...

1. We begin by defining an `Application\Acl\AuthenticateInterface` class. We use this interface to support the Adapter software design pattern, making our `Authenticate` class more generically useful by allowing a variety of adapters, each of which can draw authentication from a different source (for example, from a file, using OAuth2, and so on). Note the use of the PHP 7 ability to define the return value data type:

```
namespace Application\Acl;
use Psr\Http\Message\ { RequestInterface,
ResponseInterface };
interface AuthenticateInterface
{
  public function login(RequestInterface
$request) :
    ResponseInterface;
}
```

> **Note**
>
> Note that by defining a method that requires a PSR-7-compliant request, and produces a PSR-7-compliant

response, we have made this interface universally applicable.

2. Next, we define the adapter that implements the `login()` method required by the interface. We make sure to use the appropriate classes, and define fitting constants and properties. The constructor makes use of `Application\Database\Connection`, defined in Chapter 5, *Interacting with a Database*:

```
namespace Application\Acl;
use PDO;
use Application\Database\Connection;
use Psr\Http\Message\ { RequestInterface,
ResponseInterface };
use Application\MiddleWare\ { Response,
TextStream };
class DbTable  implements
AuthenticateInterface
{
  const ERROR_AUTH = 'ERROR: authentication
error';
  protected $conn;
  protected $table;
  public function __construct(Connection
$conn, $tableName)
  {
    $this->conn = $conn;
    $this->table = $tableName;
  }
```

3. The core `login()` method extracts the username and password from the request object. We then do a straightforward database lookup. If there is a match, we store user information in the response body, JSON-encoded:

```
public function login(RequestInterface
$request) :
  ResponseInterface
{
  $code = 401;
  $info = FALSE;
  $body = new TextStream(self::ERROR_AUTH);
  $params = json_decode($request->getBody()-
>getContents());
```

```
        $response = new Response();
        $username = $params->username ?? FALSE;
        if ($username) {
            $sql = 'SELECT * FROM ' . $this->table
              . ' WHERE email = ?';
            $stmt = $this->conn->pdo->prepare($sql);
            $stmt->execute([$username]);
            $row = $stmt->fetch(PDO::FETCH_ASSOC);
            if ($row) {
                if (password_verify($params-
>password,
                    $row['password'])) {
                      unset($row['password']);
                      $body =
                      new
TextStream(json_encode($row));
                      $response->withBody($body);
                      $code = 202;
                      $info = $row;
                    }
                  }
                }
                return $response->withBody($body)-
>withStatus($code);
              }
            }
```

Tip

Best practice

Never store passwords in clear text. When you need to do a password match, use `password_verify()`, which negates the need to reproduce the password hash.

4. The `Authenticate` class is a wrapper for an adapter class that implements `AuthenticationInterface`. Accordingly, the constructor takes an adapter class as an argument, as well as a string that serves as the key, in which authentication information is stored in `$_SESSION`:

```
namespace Application\Acl;
use Application\MiddleWare\ { Response,
TextStream };
use Psr\Http\Message\ { RequestInterface,
ResponseInterface };
class Authenticate
{
  const ERROR_AUTH = 'ERROR: invalid token';
  const DEFAULT_KEY = 'auth';
  protected $adapter;
  protected $token;
  public function __construct(
  AuthenticateInterface $adapter, $key)
  {
    $this->key = $key;
    $this->adapter = $adapter;
  }
```

5. In addition, we provide a login form with a security token, which helps prevent **Cross Site Request Forgery** (**CSRF**) attacks:

```
public function getToken()
{
  $this->token = bin2hex(random_bytes(16));
  $_SESSION['token'] = $this->token;
  return $this->token;
}
public function matchToken($token)
{
  $sessToken = $_SESSION['token'] ??
date('Ymd');
  return ($token == $sessToken);
}
public function getLoginForm($action = NULL)
{
  $action = ($action) ? 'action="' . $action .
'" ' : '';
  $output = '<form method="post" ' . $action .
'>';
  $output .= '<table><tr><th>Username</th>
<td>';
  $output .= '<input type="text"
name="username" /></td>';
  $output .= '</tr><tr><th>Password</th><td>';
  $output .= '<input type="password"
name="password" />';
  $output .= '</td></tr><tr><th> </th>';
```

```
    $output .= '<td><input type="submit" />
</td>';
    $output .= '</tr></table>';
    $output .= '<input type="hidden"
name="token" value="';
    $output .= $this->getToken() . '" />';
    $output .= '</form>';
    return $output;
}
```

6. Finally, the `login()` method in this class checks whether the token is valid. If not, a 400 response is returned. Otherwise, the `login()` method of the adapter is called:

```
public function login(
RequestInterface $request) : ResponseInterface
{
    $params = json_decode($request->getBody()-
>getContents());
    $token = $params->token ?? FALSE;
    if (!($token && $this->matchToken($token)))
{
        $code = 400;
        $body = new
TextStream(self::ERROR_AUTH);
        $response = new Response($code, $body);
    } else {
        $response = $this->adapter-
>login($request);
    }
    if ($response->getStatusCode() >= 200
        && $response->getStatusCode() < 300) {
        $_SESSION[$this->key] =
          json_decode($response->getBody()-
>getContents());
    } else {
        $_SESSION[$this->key] = NULL;
    }
    return $response;
}

}
```

How it works...

First of all, be sure to follow the recipes defined in Appendix, *Defining PSR-7 Classes*. Next, go ahead and define the classes presented in this recipe, summarized in the following table:

Class	Discussed in these steps
`Application\Acl\AuthenticateInterface`	1
`Application\Acl\DbTable`	2 - 3
`Application\Acl\Authenticate`	4 - 6

You can then define a `chap_09_middleware_authenticate.php` calling program that sets up autoloading and uses the appropriate classes:

```
<?php
session_start();
define('DB_CONFIG_FILE', __DIR__ .
'/../config/db.config.php');
define('DB_TABLE', 'customer_09');
define('SESSION_KEY', 'auth');
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');

use Application\Database\Connection;
use Application\Acl\ { DbTable, Authenticate };
use Application\MiddleWare\ { ServerRequest,
Request, Constants, TextStream };
```

You are now in a position to set up the authentication adapter and core class:

```
$conn   = new Connection(include DB_CONFIG_FILE);
$dbAuth = new DbTable($conn, DB_TABLE);
$auth   = new Authenticate($dbAuth, SESSION_KEY);
```

Be sure to initialize the incoming request, and set up the request to be made to the authentication class:

```
$incoming = new ServerRequest();
$incoming->initialize();
$outbound = new Request();
```

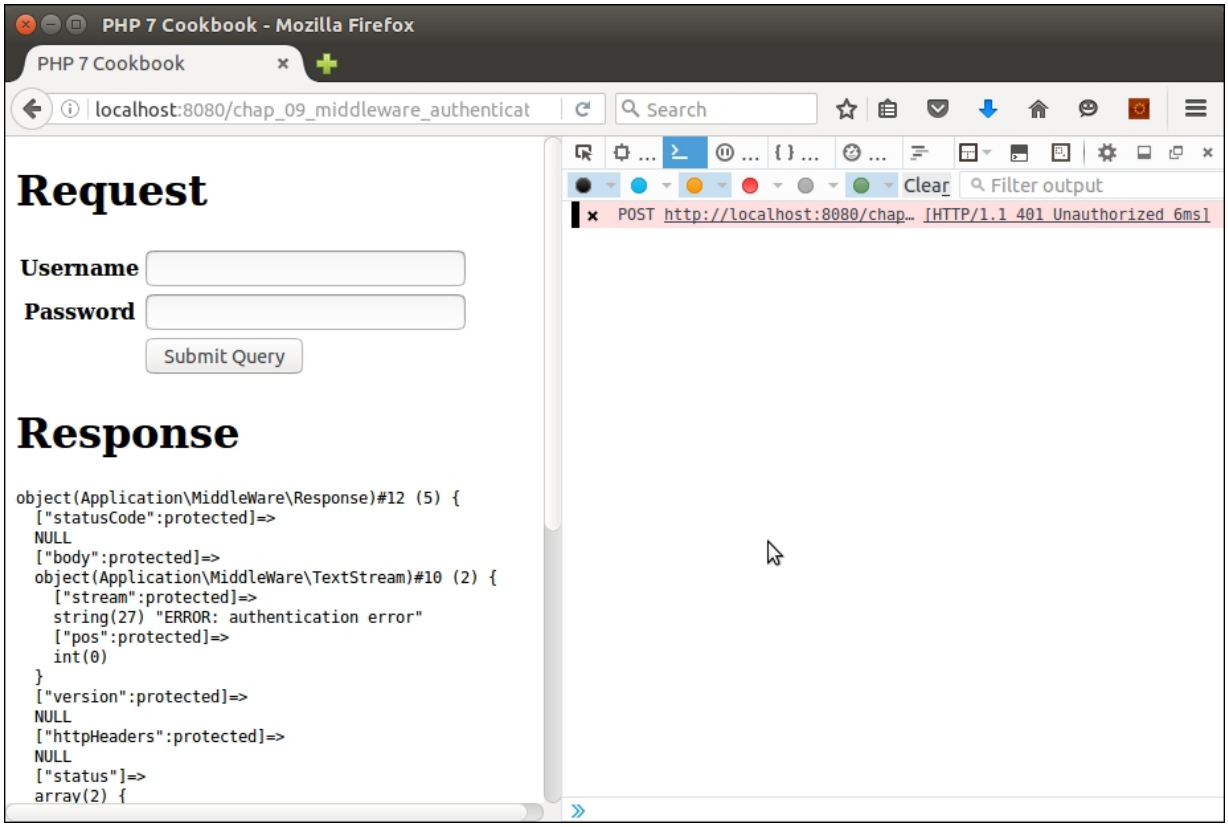
Check the incoming class method to see if it is `POST`. If so, pass a request to the authentication class:

```
if ($incoming->getMethod() ==
Constants::METHOD_POST) {
  $body = new TextStream(json_encode(
  $incoming->getParsedBody()));
  $response = $auth->login($outbound-
>withBody($body));
}
$action = $incoming->getServerParams()
['PHP_SELF'];
?>
```

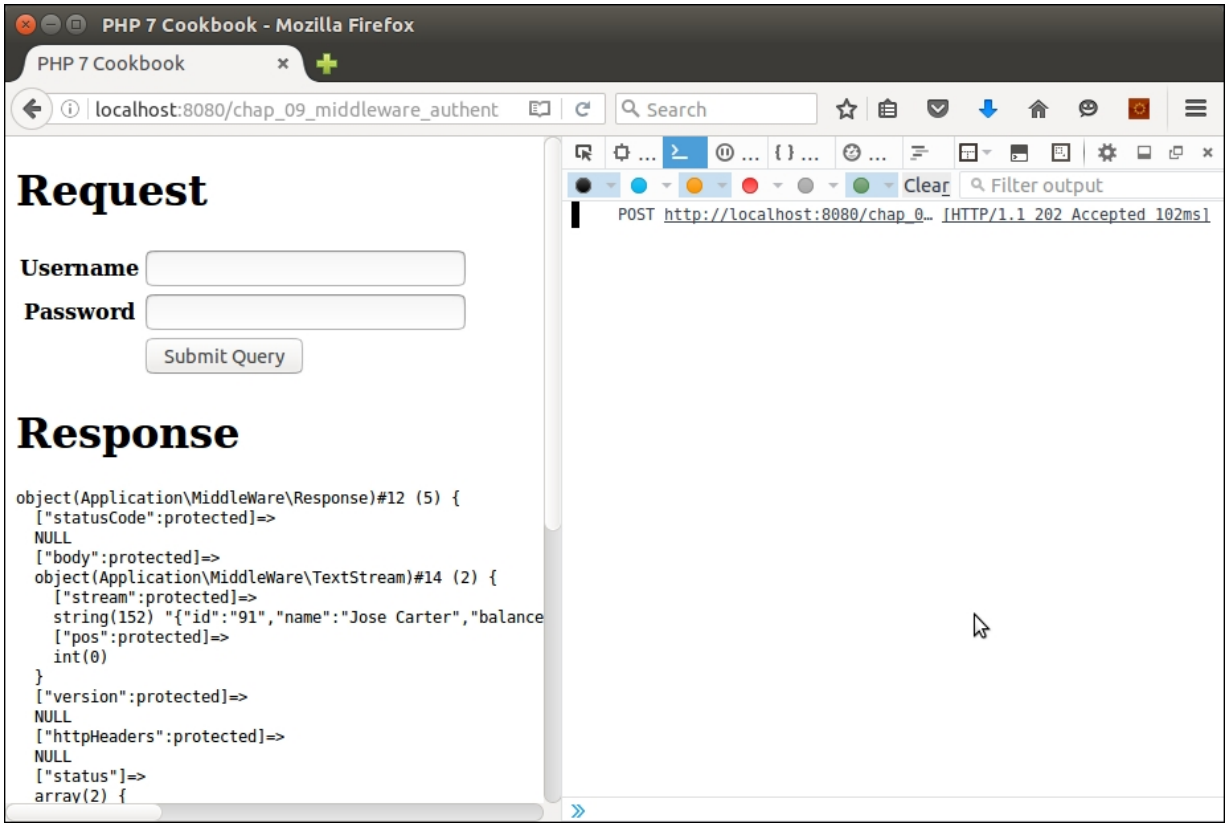
The display logic looks like this:

```
<?= $auth->getLoginForm($action) ?>
```

Here is the output from an invalid authentication attempt. Notice the `401` status code on the right. In this illustration, you could add a `var_dump()` of the response object:



Here is a successful authentication:



See also

For guidance on how to avoid CSRF and other attacks, please see Chapter 12, *Improving Web Security*.

Using middleware to implement access control

As the name implies, middleware sits in the middle of a sequence of function or method calls. Accordingly, middleware is well suited for the task of "gate keeper". You can easily implement an **Access Control List** (**ACL**) mechanism with a middleware class that reads the ACL, and allows or denies access to the next function or method call in the sequence.

How to do it...

1. Probably the most difficult part of the process is determining which factors to include in the ACL. For the purposes of illustration, let's say that our users are all assigned a `level` and a `status`. In this illustration, the level is defined as follows:

```
'levels' => [0, 'BEG', 'INT', 'ADV']
```

2. The status could indicate how far they are in the membership signup process. For example, a status of `0` could indicate they've initiated the membership signup process, but have not yet been confirmed. A status of `1` could indicate their e-mail address is confirmed, but they have not paid the monthly fee, and so on.
3. Next, we need to define the resources we plan to control. In this case, we will assume there is a need to control access to a series of web pages on the site. Accordingly, we need to define an array of such resources. In the ACL, we can then refer to the key:

```
'pages'  => [0 => 'sorry', 'logout' =>
'logout', 'login'  => 'auth',
             1 => 'page1', 2 => 'page2', 3 =>
'page3',
             4 => 'page4', 5 => 'page5', 6 =>
'page6',
```

```
                7 => 'page7', 8 => 'page8', 9 =>
'page9']
```

4. Finally, the most important piece of configuration is to make assignments to pages according to `level` and `status`. The generic template used in the configuration array might look like this:

```
status => ['inherits' => <key>, 'pages' =>
[level => [pages allowed], etc.]]
```

5. Now we are in a position to define the `Acl` class. As before, we use a few classes, and define constants and properties appropriate for access control:

```
namespace Application\Acl;

use InvalidArgumentException;
use Psr\Http\Message\RequestInterface;
use Application\MiddleWare\ { Constants,
Response, TextStream };

class Acl
{
  const DEFAULT_STATUS = '';
  const DEFAULT_LEVEL  = 0;
  const DEFAULT_PAGE   = 0;
  const ERROR_ACL = 'ERROR: authorization
error';
  const ERROR_APP = 'ERROR: requested page not
listed';
  const ERROR_DEF =
    'ERROR: must assign keys "levels", "pages"
and "allowed"';
  protected $default;
  protected $levels;
  protected $pages;
  protected $allowed;
```

6. In the `__construct()` method, we break up the assignments array into `$pages`, the resources to be controlled, `$levels`, and `$allowed`, which are the actual assignments. If the array does not include one of these three sub-components, an exception is thrown:

```
public function __construct(array
$assignments)
{
  $this->default = $assignments['default']
    ?? self::DEFAULT_PAGE;
  $this->pages   = $assignments['pages'] ??
FALSE;
  $this->levels  = $assignments['levels'] ??
FALSE;
  $this->allowed = $assignments['allowed'] ??
FALSE;
  if (!($this->pages && $this->levels &&
$this->allowed)) {
      throw new
InvalidArgumentException(self::ERROR_DEF);
  }
}
```

7. You may have noticed that we allow inheritance. In `$allowed`, the `inherits` key can be set to another key within the array. If so, we need to merge its values with the values currently under examination. We iterate through `$allowed` in reverse, merging any inherited values each time through the loop. This method, incidentally, also only isolates rules that apply to a certain `status` and `level`:

```
protected function mergeInherited($status,
$level)
{
  $allowed = $this->allowed[$status]['pages']
[$level]
    ?? array();
  for ($x = $status; $x > 0; $x--) {
    $inherits = $this->allowed[$x]
['inherits'];
    if ($inherits) {
        $subArray =
          $this->allowed[$inherits]['pages']
[$level]
          ?? array();
        $allowed = array_merge($allowed,
$subArray);
    }
  }
```

```
    return $allowed;
}
```

8. When processing authorization, we initialize a few variables, and then extract the page requested from the original request URI. If the page parameter doesn't exist, we set a `400` code:

```
public function isAuthorized(RequestInterface
$request)
{
  $code = 401;    // unauthorized
  $text['page'] = $this->pages[$this-
>default];
  $text['authorized'] = FALSE;
  $page = $request->getUri()->getQueryParams()
['page']
    ?? FALSE;
  if ($page === FALSE) {
      $code = 400;    // bad request
```

9. Otherwise, we decode the request body contents, and acquire the `status` and `level`. We are then in a position to call `mergeInherited()`, which returns an array of pages accessible to this `status` and `level`:

```
} else {
    $params = json_decode(
      $request->getBody()->getContents());
    $status = $params->status ??
self::DEFAULT_LEVEL;
    $level  = $params->level  ?? '*';
    $allowed = $this->mergeInherited($status,
$level);
```

10. If the requested page is in the `$allowed` array, we set the status code to a happy `200`, and return an authorized setting along with the web page that corresponds to the page code requested:

```
if (in_array($page, $allowed)) {
    $code = 200;    // OK
    $text['authorized'] = TRUE;
    $text['page'] = $this->pages[$page];
} else {
    $code = 401;            }
}
```

11. We then return the response, JSON-encoded, and we are done:

```
$body = new TextStream(json_encode($text));
return (new Response())->withStatus($code)
->withBody($body);
}

}
```

How it works...

After that, you will need to define `Application\Acl\Acl`, which is discussed in this recipe. Now move to the `/path/to/source/for/this/chapter` folder and create two directories: `public` and `pages`. In `pages`, create a series of PHP files, such as `page1.php`, `page2.php`, and so on. Here is an example of how one of these pages might look:

```
<?php // page 1 ?>
<h1>Page 1</h1>
<hr>
<p>Lorem ipsum dolor sit amet, consectetur
adipiscing elit. etc.</p>
```

You can also define a `menu.php` page, which could be included in the output:

```
<?php // menu ?>
<a href="?page=1">Page 1</a>
<a href="?page=2">Page 2</a>
<a href="?page=3">Page 3</a>
// etc.
```

The `logout.php` page should destroy the session:

```
<?php
  $_SESSION['info'] = FALSE;
  session_destroy();
?>
<a href="/">BACK</a>
```

The `auth.php` page will display a login screen (as described in the previous recipe):

```
<?= $auth->getLoginForm($action) ?>
```

You can then create a configuration file that allows access to web pages depending on level and status. For the sake of illustration, call it `chap_09_middleware_acl_config.php` and return an array that might look like this:

```
<?php
$min = [0, 'logout'];
return [
  'default' => 0,     // default page
  'levels' => [0, 'BEG', 'INT', 'ADV'],
  'pages'  => [0 => 'sorry',
  'logout' => 'logout',
  'login' => 'auth',
               1 => 'page1', 2 => 'page2', 3 =>
'page3',
               4 => 'page4', 5 => 'page5', 6 =>
'page6',
               7 => 'page7', 8 => 'page8', 9 =>
'page9'],
  'allowed' => [
               0 => ['inherits' => FALSE,
                     'pages' => [ '*' => $min,
'BEG' => $min,
                     'INT' => $min,'ADV' =>
$min]],
               1 => ['inherits' => FALSE,
                     'pages' => ['*' =>
['logout'],
                    'BEG' => [1, 'logout'],
                    'INT' => [1,2, 'logout'],
                    'ADV' => [1,2,3, 'logout']]],
               2 => ['inherits' => 1,
                     'pages' => ['BEG' => [4],
                     'INT' => [4,5],
                     'ADV' => [4,5,6]]],
               3 => ['inherits' => 2,
                     'pages' => ['BEG' => [7],
                     'INT' => [7,8],
                     'ADV' => [7,8,9]]]
```

```
        ]
    ];
```

Finally, in the `public` folder, define `index.php`, which sets up autoloading, and ultimately calls up both the `Authenticate` and `Acl` classes. As with other recipes, define configuration files, set up autoloading, and use certain classes. Also, don't forget to start the session:

```
<?php
session_start();
session_regenerate_id();
define('DB_CONFIG_FILE', __DIR__ .
'/../../config/db.config.php');
define('DB_TABLE', 'customer_09');
define('PAGE_DIR', __DIR__ . '/../pages');
define('SESSION_KEY', 'auth');
require __DIR__ .
'/../../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/../..');

use Application\Database\Connection;
use Application\Acl\ { Authenticate, Acl };
use Application\MiddleWare\ { ServerRequest,
Request, Constants, TextStream };
```

Tip

Best practice

It is a best practice to protect your sessions. An easy way to help protect a session is to use `session_regenerate_id()`, which invalidates the existing PHP session identifier and generates a new one. Thus, if an attacker were to obtain the session identifier through illegal means, the window of time in which any given session identifier is valid is kept to a minimum.

You can now pull in the ACL configuration, and create instances for `Authenticate` as well as `Acl`:

```
$config = require __DIR__ .
'/../chap_09_middleware_acl_config.php';
$acl    = new Acl($config);
$conn   = new Connection(include DB_CONFIG_FILE);
$dbAuth = new DbTable($conn, DB_TABLE);
$auth   = new Authenticate($dbAuth, SESSION_KEY);
```

Next, define incoming and outbound request instances:

```
$incoming = new ServerRequest();
$incoming->initialize();
$outbound = new Request();
```

If the incoming request method was `post`, process the authentication calling the `login()` method:

```
if (strtolower($incoming->getMethod()) ==
Constants::METHOD_POST) {
    $body = new TextStream(json_encode(
    $incoming->getParsedBody()));
    $response = $auth->login($outbound-
>withBody($body));
}
```

If the session key defined for authentication is populated, that means the user has been successfully authenticated. If not, we program an anonymous function, called **later**, which includes the authentication login page:

```
$info = $_SESSION[SESSION_KEY] ?? FALSE;
if (!$info) {
    $execute = function () use ($auth) {
      include PAGE_DIR . '/auth.php';
    };
```

Otherwise, you can proceed with the ACL check. You first need to find, from the original query, which web page the user wants to visit,

however:

```
} else {
    $query = $incoming->getServerParams()
['QUERY_STRING'] ?? '';
```

You can then reprogram the `$outbound` request to include this information:

```
$outbound->withBody(new
TextStream(json_encode($info)));
$outbound->getUri()->withQuery($query);
```

Next, you'll be in a position to check authorization, supplying the outbound request as an argument:

```
$response = $acl->isAuthorized($outbound);
```

You can then examine the return response for the `authorized` parameter, and program an anonymous function to include the return `page` parameter if OK, and the `sorry` page otherwise:

```
$params   = json_decode($response->getBody()-
>getContents());
$isAllowed = $params->authorized ?? FALSE;
if ($isAllowed) {
    $execute = function () use ($response,
$params) {
      include PAGE_DIR .'/' . $params->page .
'.php';
      echo '<pre>', var_dump($response), '</pre>';
      echo '<pre>',
var_dump($_SESSION[SESSION_KEY]);
      echo '</pre>';
    };
} else {
    $execute = function () use ($response) {
      include PAGE_DIR .'/sorry.php';
      echo '<pre>', var_dump($response), '</pre>';
      echo '<pre>',
var_dump($_SESSION[SESSION_KEY]);
```

```
        echo '</pre>';
    };
}
}
```

Now all you need to do is to set the form action and wrap the anonymous function in HTML:

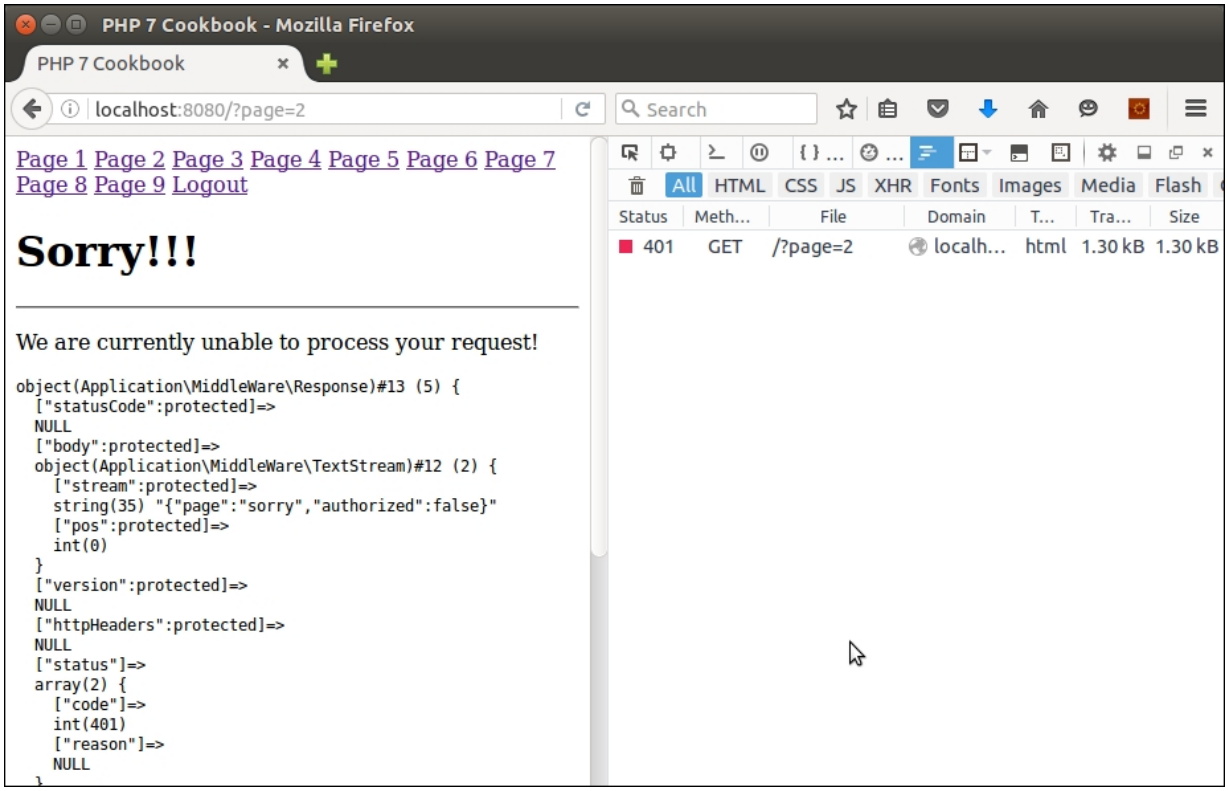
```
$action = $incoming->getServerParams()
['PHP_SELF'];
?>
<!DOCTYPE html>
<head>
  <title>PHP 7 Cookbook</title>
  <meta http-equiv="content-type"
content="text/html;charset=utf-8" />
</head>
<body>
  <?php $execute(); ?>
</body>
</html>
```

To test it, you can use the built-in PHP web server, but you will need to use the `-t` flag to indicate that the document root is `public`:

```
cd /path/to/source/for/this/chapter
php -S localhost:8080 -t public
```

From a browser, you can access the `http://localhost:8080/` URL.

If you try to access any page, you will simply be redirected back to the login page. As per the configuration, a user with status = `1`, and level = `BEG` can only access page `1` and log out. If, when logged in as this user, you try to access page 2, here is the output:



See also

This example relies on `$_SESSION` as the sole means of user authentication once they have logged in. For good examples of how you can protect PHP sessions, please see Chapter 12, *Improving Web Security*, specifically the recipe entitled *Safeguarding the PHP session*.

Improving performance using the cache

The cache software design pattern is where you store a result that takes a long time to generate. This could take the form of a lengthy view script or a complex database query. The storage destination needs to be highly performant, of course, if you wish to improve the user experience of website visitors. As different installations will have different potential storage targets, the cache mechanism lends itself to the adapter pattern as well. Examples of potential storage destinations include memory, a database, and the filesystem.

How to do it...

1. As with a couple of other recipes in this chapter, as there are shared constants, we define a discreet `Application\Cache\Constants` class:

```
<?php
namespace Application\Cache;

class Constants
{
  const DEFAULT_GROUP  = 'default';
  const DEFAULT_PREFIX = 'CACHE_';
  const DEFAULT_SUFFIX = '.cache';
  const ERROR_GET      = 'ERROR: unable to
retrieve from cache';
  // not all constants are shown to conserve
space
}
```

2. Seeing as we are following the adapter design pattern, we define an interface next:

```
namespace Application\Cache;
interface  CacheAdapterInterface
{
  public function hasKey($key);
```

```
  public function getFromCache($key, $group);
  public function saveToCache($key, $data,
$group);
  public function removeByKey($key);
  public function removeByGroup($group);
}
```

3. Now we are ready to define our first cache adapter, in this illustration, by using a MySQL database. We need to define properties that will hold column names as well as prepared statements:

```
namespace Application\Cache;
use PDO;
use Application\Database\Connection;
class Database implements
CacheAdapterInterface
{
  protected $sql;
  protected $connection;
  protected $table;
  protected $dataColumnName;
  protected $keyColumnName;
  protected $groupColumnName;
  protected $statementHasKey       = NULL;
  protected $statementGetFromCache = NULL;
  protected $statementSaveToCache  = NULL;
  protected $statementRemoveByKey  = NULL;
  protected $statementRemoveByGroup= NULL;
```

4. The constructor allows us to provide key column names as well as an `Application\Database\Connection` instance and the name of the table used for the cache:

```
public function __construct(Connection
$connection,
  $table,
  $idColumnName,
  $keyColumnName,
  $dataColumnName,
  $groupColumnName = Constants::DEFAULT_GROUP)
  {
    $this->connection  = $connection;
    $this->setTable($table);
    $this->setIdColumnName($idColumnName);
    $this->setDataColumnName($dataColumnName);
```

```
        $this->setKeyColumnName($keyColumnName);
        $this-
>setGroupColumnName($groupColumnName);
    }
```

5. The next few methods prepare statements, and are called when we access the database. We do not show all the methods, but present enough to give you the idea:

```
public function prepareHasKey()
{
  $sql = 'SELECT `' . $this->idColumnName . '`
'
  . 'FROM `'   . $this->table . '` '
  . 'WHERE `'  . $this->keyColumnName . '` =
:key ';
  $this->sql[__METHOD__] = $sql;
  $this->statementHasKey =
  $this->connection->pdo->prepare($sql);
}
public function prepareGetFromCache()
{
  $sql = 'SELECT `' . $this->dataColumnName .
'` '
  . 'FROM `'   . $this->table . '` '
  . 'WHERE `'  . $this->keyColumnName . '` =
:key '
  . 'AND `'    . $this->groupColumnName . '` =
:group';
  $this->sql[__METHOD__] = $sql;
  $this->statementGetFromCache =
  $this->connection->pdo->prepare($sql);
}
```

6. Now we define a method that determines whether data for a given key exists:

```
public function hasKey($key)
{
  $result = 0;
  try {
      if (!$this->statementHasKey) $this-
>prepareHasKey();
          $this->statementHasKey-
>execute(['key' => $key]);
  } catch (Throwable $e) {
      error_log(__METHOD__ . ':' . $e-
```

```
>getMessage());
      throw new
Exception(Constants::ERROR_REMOVE_KEY);
  }
  return (int) $this->statementHasKey
  ->fetch(PDO::FETCH_ASSOC)[$this-
>idColumnName];
}
```

7. The core methods are ones that read from and write to the cache. Here is the method that retrieves from the cache. All we need to do is to execute the prepared statement, which performs a `SELECT`, with a `WHERE` clause, which incorporates the key and group:

```
public function getFromCache(
$key, $group = Constants::DEFAULT_GROUP)
{
  try {
      if (!$this->statementGetFromCache)
          $this->prepareGetFromCache();
          $this->statementGetFromCache-
>execute(
             ['key' => $key, 'group' =>
$group]);
          while ($row = $this-
>statementGetFromCache
            ->fetch(PDO::FETCH_ASSOC)) {
            if ($row && count($row)) {
                yield unserialize($row[$this-
>dataColumnName]);
            }
          }
  } catch (Throwable $e) {
      error_log(__METHOD__ . ':' . $e-
>getMessage());
      throw new
Exception(Constants::ERROR_GET);
  }
}
```

8. When writing to the cache, we first determine whether an entry for this cache key exists. If so, we perform an `UPDATE`; otherwise, we perform an `INSERT`:

```
public function saveToCache($key, $data,
$group = Constants::DEFAULT_GROUP)
{
  $id = $this->hasKey($key);
  $result = 0;
  try {
      if ($id) {
          if (!$this->statementUpdateCache)
              $this->prepareUpdateCache();
              $result = $this-
>statementUpdateCache
              ->execute(['key' => $key,
              'data' => serialize($data),
              'group' => $group,
              'id' => $id]);
          } else {
              if (!$this-
>statementSaveToCache)
              $this->prepareSaveToCache();
              $result = $this-
>statementSaveToCache
              ->execute(['key' => $key,
              'data' => serialize($data),
              'group' => $group]);
          }
      } catch (Throwable $e) {
          error_log(__METHOD__ . ':' . $e-
>getMessage());
          throw new
Exception(Constants::ERROR_SAVE);
      }
      return $result;
   }
```

9. We then define two methods that remove the cache either by key or by group. Removal by group provides a convenient mechanism if there are a large number of items that need to be deleted:

```
public function removeByKey($key)
{
  $result = 0;
  try {
      if (!$this->statementRemoveByKey)
      $this->prepareRemoveByKey();
      $result = $this->statementRemoveByKey-
```

```
>execute(
        ['key' => $key]);
  } catch (Throwable $e) {
      error_log(__METHOD__ . ':' . $e-
>getMessage());
      throw new
Exception(Constants::ERROR_REMOVE_KEY);
  }
  return $result;
}

public function removeByGroup($group)
{
  $result = 0;
  try {
      if (!$this->statementRemoveByGroup)
          $this->prepareRemoveByGroup();
          $result = $this-
>statementRemoveByGroup->execute(
            ['group' => $group]);
      } catch (Throwable $e) {
          error_log(__METHOD__ . ':' . $e-
>getMessage());
          throw new
Exception(Constants::ERROR_REMOVE_GROUP);
      }
      return $result;
  }
```

10. Lastly, we define getters and setters for each of the properties. Not all are shown here to conserve space:

```
public function setTable($name)
{
  $this->table = $name;
}
public function getTable()
{
  return $this->table;
}
// etc.
}
```

11. The filesystem cache adapter defines the same methods as defined earlier. Note the use of `md5()`, not for security, but as a way of quickly generating a text string from the key:

```
namespace Application\Cache;
use RecursiveIteratorIterator;
use RecursiveDirectoryIterator;
class File implements CacheAdapterInterface
{
  protected $dir;
  protected $prefix;
  protected $suffix;
  public function __construct(
    $dir, $prefix = NULL, $suffix = NULL)
  {
    if (!file_exists($dir)) {
        error_log(__METHOD__ . ':' .
Constants::ERROR_DIR_NOT);
        throw new
Exception(Constants::ERROR_DIR_NOT);
    }
    $this->dir = $dir;
    $this->prefix = $prefix ??
Constants::DEFAULT_PREFIX;
    $this->suffix = $suffix ??
Constants::DEFAULT_SUFFIX;
  }

  public function hasKey($key)
  {
    $action = function ($name, $md5Key,
&$item) {
      if (strpos($name, $md5Key) !== FALSE) {
        $item ++;
      }
    };

    return $this->findKey($key, $action);
  }

  public function getFromCache($key, $group =
Constants::DEFAULT_GROUP)
  {
    $fn = $this->dir . '/' . $group . '/'
    . $this->prefix . md5($key) . $this-
>suffix;
    if (file_exists($fn)) {
        foreach (file($fn) as $line) { yield
$line; }
    } else {
        return array();
```

```
    }
  }

  public function saveToCache(
    $key, $data, $group =
Constants::DEFAULT_GROUP)
  {
    $baseDir = $this->dir . '/' . $group;
    if (!file_exists($baseDir))
mkdir($baseDir);
    $fn = $baseDir . '/' . $this->prefix .
md5($key)
    . $this->suffix;
    return file_put_contents($fn,
json_encode($data));
  }

  protected function findKey($key, callable
$action)
  {
    $md5Key = md5($key);
    $iterator = new RecursiveIteratorIterator(
      new RecursiveDirectoryIterator($this-
>dir),
      RecursiveIteratorIterator::SELF_FIRST);
      $item = 0;
    foreach ($iterator as $name => $obj) {
      $action($name, $md5Key, $item);
    }
    return $item;
  }

  public function removeByKey($key)
  {
    $action = function ($name, $md5Key,
&$item) {
      if (strpos($name, $md5Key) !== FALSE) {
        unlink($name);
        $item++;
      }
    };
    return $this->findKey($key, $action);
  }

  public function removeByGroup($group)
  {
    $removed = 0;
    $baseDir = $this->dir . '/' . $group;
```

```
        $pattern = $baseDir . '/' . $this->prefix
. '*'
        . $this->suffix;
        foreach (glob($pattern) as $file) {
          unlink($file);
          $removed++;
        }
        return $removed;
      }
    }
```

12. Now we are ready to present the core cache mechanism. In the constructor, we accept a class that implements `CacheAdapterInterface` as an argument:

```
namespace Application\Cache;
use Psr\Http\Message\RequestInterface;
use Application\MiddleWare\ { Request,
Response, TextStream };
class Core
{
  public function
__construct(CacheAdapterInterface $adapter)
  {
    $this->adapter = $adapter;
  }
```

13. Next are a series of wrapper methods that call methods of the same name from the adapter, but accept a `Psr\Http\Message\RequestInterface` class an an argument, and return a `Psr\Http\Message\ResponseInterface` as a response. We start with a simple one: `hasKey()`. Note how we extract the `key` from the request parameters:

```
public function hasKey(RequestInterface
$request)
{
  $key = $request->getUri()->getQueryParams()
['key'] ?? '';
  $result = $this->adapter->hasKey($key);
}
```

14. To retrieve information from the cache, we need to pull the key and group parameters from the request object, and then call the same method from the adapter. If no results are obtained, we

set a `204` code, which indicates the request was a success, but no content was produced. Otherwise, we set a `200` (success) code, and iterate through the results. Everything is then stuffed into a response object, which is returned:

```
public function getFromCache(RequestInterface
$request)
{
  $text = array();
  $key = $request->getUri()->getQueryParams()
['key'] ?? '';
  $group = $request->getUri()-
>getQueryParams()['group']
    ?? Constants::DEFAULT_GROUP;
  $results = $this->adapter-
>getFromCache($key, $group);
  if (!$results) {
      $code = 204;
  } else {
      $code = 200;
      foreach ($results as $line) $text[] =
$line;
  }
  if (!$text || count($text) == 0) $code =
204;
  $body = new TextStream(json_encode($text));
  return (new Response())->withStatus($code)
                         ->withBody($body);
}
```

15. Strangely, writing to the cache is almost identical, except that the results are expected to be either a number (that is, the number of rows affected), or a Boolean result:

```
public function saveToCache(RequestInterface
$request)
{
  $text = array();
  $key = $request->getUri()->getQueryParams()
['key'] ?? '';
  $group = $request->getUri()-
>getQueryParams()['group']
    ?? Constants::DEFAULT_GROUP;
  $data = $request->getBody()->getContents();
  $results = $this->adapter->saveToCache($key,
$data, $group);
```

```
  if (!$results) {
      $code = 204;
  } else {
      $code = 200;
      $text[] = $results;
  }
      $body = new
TextStream(json_encode($text));
      return (new Response())-
>withStatus($code)
                                -
>withBody($body);
  }
```

16. The remove methods are, as expected, quite similar to each other:

```
public function removeByKey(RequestInterface
$request)
{
  $text = array();
  $key = $request->getUri()->getQueryParams()
['key'] ?? '';
  $results = $this->adapter-
>removeByKey($key);
  if (!$results) {
      $code = 204;
  } else {
      $code = 200;
      $text[] = $results;
  }
  $body = new TextStream(json_encode($text));
  return (new Response())->withStatus($code)
                         ->withBody($body);
}

public function removeByGroup(RequestInterface
$request)
{
  $text = array();
  $group = $request->getUri()-
>getQueryParams()['group']
    ?? Constants::DEFAULT_GROUP;
  $results = $this->adapter-
>removeByGroup($group);
  if (!$results) {
      $code = 204;
```

```
        } else {
            $code = 200;
            $text[] = $results;
        }
        $body = new TextStream(json_encode($text));
        return (new Response())->withStatus($code)
                               ->withBody($body);
        }
    } // closing brace for class Core
```

How it works...

In order to demonstrate the use of the `Acl` class, you will need to define the classes described in this recipe, summarized here:

Class	Discussed in these steps
`Application\Cache\Constants`	1
`Application\Cache\CacheAdapterInterface`	2
`Application\Cache\Database`	3 - 10
`Application\Cache\File`	11
`Application\Cache\Core`	12 - 16

Next, define a test program, which you could call `chap_09_middleware_cache_db.php`. In this program, as usual, define constants for necessary files, set up autoloading, use the appropriate classes, oh... and write a function that produces prime numbers (you're probably re-reading that last little bit at this point. Not to worry, we can help you with that!):

```
<?php
define('DB_CONFIG_FILE', __DIR__ .
'/../config/db.config.php');
define('DB_TABLE', 'cache');
define('CACHE_DIR', __DIR__ . '/cache');
define('MAX_NUM', 100000);
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
use Application\Database\Connection;
use Application\Cache\{ Constants, Core, Database,
File };
use Application\MiddleWare\ { Request, TextStream
};
```

Well, a function that takes a long time to run is needed, so prime number generator, here we go! The numbers 1, 2, and 3 are given as primes. We use the PHP 7 `yield from` syntax to produce these first three. then, we skip right to 5, and proceed up to the maximum value requested:

```
function generatePrimes($max)
{
  yield from [1,2,3];
  for ($x = 5; $x < $max; $x++)
  {
    if($x & 1) {
        $prime = TRUE;
        for($i = 3; $i < $x; $i++) {
            if(($x % $i) === 0) {
                $prime = FALSE;
                break;
            }
        }
        if ($prime) yield $x;
```

```
    }
  }
}
```

You can then set up a database cache adapter instance, which serves as an argument for the core:

```
$conn    = new Connection(include DB_CONFIG_FILE);
$dbCache = new Database(
  $conn, DB_TABLE, 'id', 'key', 'data', 'group');
$core    = new Core($dbCache);
```

Alternatively, if you wish to use the file cache adapter instead, here is the appropriate code:

```
$fileCache = new File(CACHE_DIR);
$core    = new Core($fileCache);
```

If you wanted to clear the cache, here is how it might be done:

```
$uriString = '/?group=' .
Constants::DEFAULT_GROUP;
$cacheRequest = new Request($uriString, 'get');
$response = $core->removeByGroup($cacheRequest);
```

You can use `time()` and `microtime()` to see how long this script runs with and without the cache:

```
$start = time() + microtime(TRUE);
echo "\nTime: " . $start;
```

Next, generate a cache request. A status code of `200` indicates you were able to obtain a list of primes from the cache:

```
$uriString = '/?key=Test1';
$cacheRequest = new Request($uriString, 'get');
$response = $core->getFromCache($cacheRequest);
$status   = $response->getStatusCode();
if ($status == 200) {
```

```
        $primes = json_decode($response->getBody()-
>getContents());
```

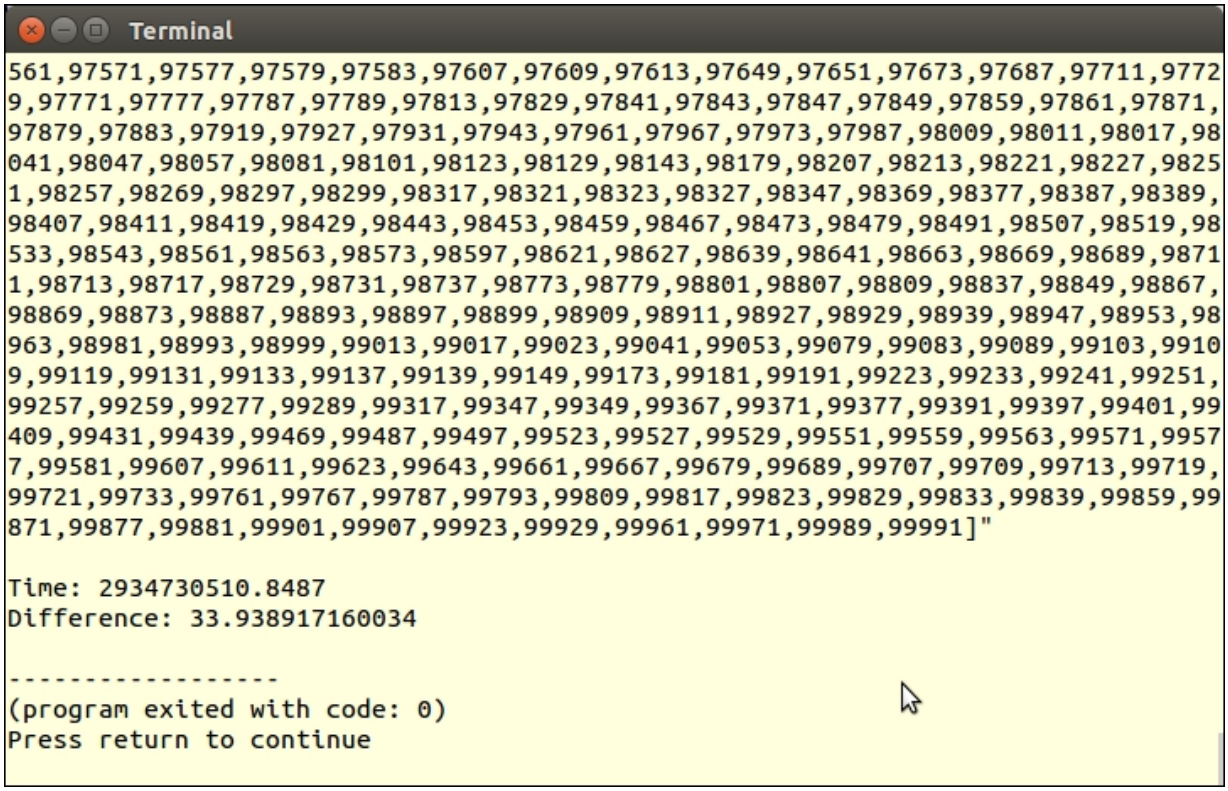
Otherwise, you can assume nothing was obtained from the cache, which means you need to generate prime numbers, and save the results to the cache:

```
} else {
    $primes = array();
    foreach (generatePrimes(MAX_NUM) as $num) {
        $primes[] = $num;
    }
    $body = new TextStream(json_encode($primes));
    $response = $core->saveToCache(
    $cacheRequest->withBody($body));
}
```

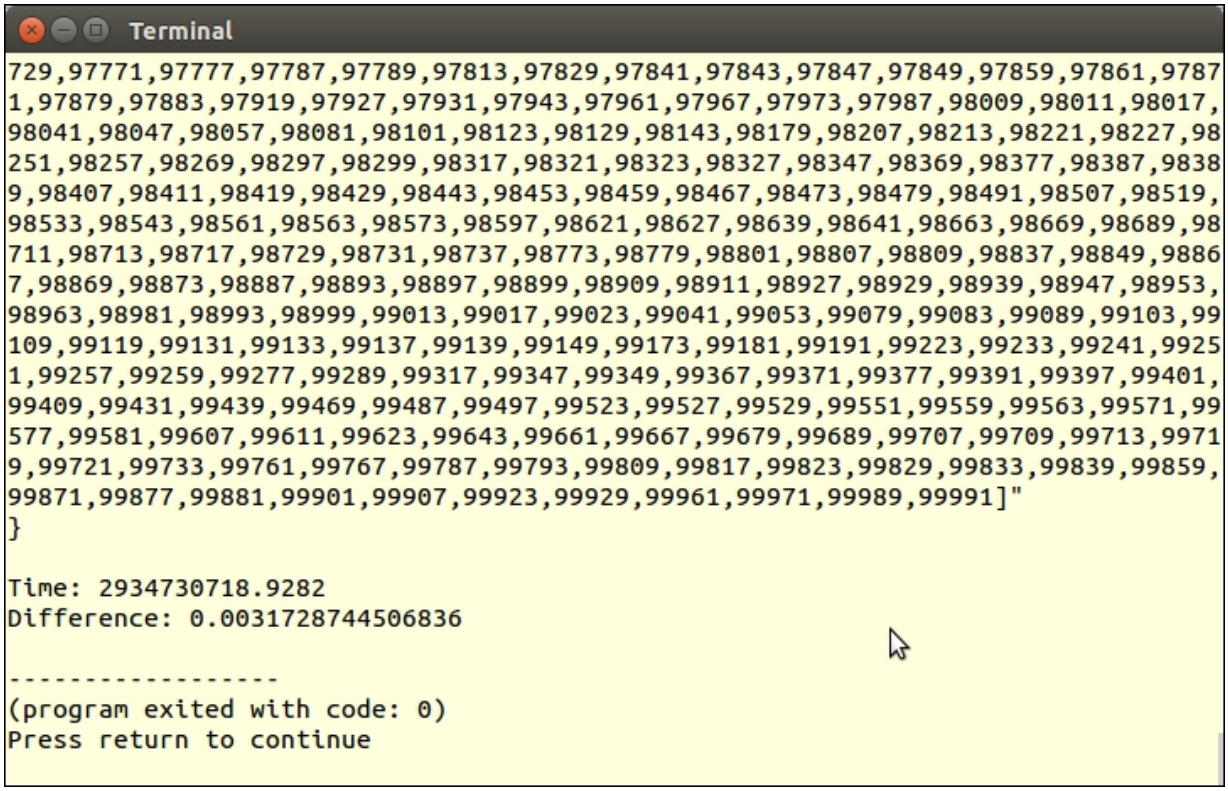
You can then check the stop time, calculate the difference, and have a look at your new list of primes:

```
$time = time() + microtime(TRUE);
$diff = $time - $start;
echo "\nTime: $time";
echo "\nDifference: $diff";
var_dump($primes);
```

Here is the expected output before values were stored in the cache:



You can now run the same program again, this time retrieving from the cache:



Allowing for the fact that our little prime number generator is not the world's most efficient, and also that the demonstration was run on a laptop, the time went from over 30 seconds down to milliseconds.

There's more...

Another possible cache adapter could be built around commands that are part of the **Alternate PHP Cache** (**APC**) extension. This extension includes such functions as `apc_exists()`, `apc_store()`, `apc_fetch()`, and `apc_clear_cache()`. These functions are perfect for our `hasKey()`, `saveToCache()`, `getFromCache()`, and `removeBy*()` functions.

See also

You might consider making slight changes to the cache adapter classes described previously following PSR-6, which is a standards recommendation directed towards the cache. There is not the same

level of acceptance of this standard as with PSR-7, however, so we decided to not follow this standard exactly in the recipe presented here. For more information on PSR-6, please refer to http://www.php-fig.org/psr/psr-6/.

Implementing routing

Routing refers to the process of accepting user-friendly URLs, dissecting the URL into its component parts, and then making a determination as to which class and method should be dispatched. The advantage of such an implementation is that not only can you make your URLs **Search Engine Optimization** (**SEO**)-friendly, but you can also create rules, incorporating regular expression patterns, which can extract values of parameters.

How to do it...

1. Probably the most popular approach is to take advantage of a web server that supports **URL rewriting**. An example of this is an Apache web server configured to use `mod_rewrite`. You then define rewriting rules that allow graphic file requests and requests for CSS and JavaScript to pass untouched. Otherwise, the request would be funneled through a routing method.
2. Another potential approach is to simply have your web server virtual host definition point to a specific routing script, which then invokes the routing class, make routing decisions, and redirect appropriately.
3. The first code to consider is how to define routing configuration. The obvious answer is to construct an array, where each key would point to a regular expression against which the URI path would match, and some form of action. An example of such configuration is shown in the following code snippet. In this example, we have three routes defined: `home`, `page`, and the default. The default should be last as it will match anything not matched previously. The action is in the form of an anonymous function that will be executed if a route match occurs:

```
$config = [
  'home' => [
    'uri' => '!^/$!',
    'exec' => function ($matches) {
      include PAGE_DIR . '/page0.php'; }
```

```
  ],
  'page' => [
    'uri' => '!^/(page)/(\d+)$!',
      'exec' => function ($matches) {
        include PAGE_DIR . '/page' .
$matches[2] . '.php'; }
  ],
  Router::DEFAULT_MATCH => [
    'uri' => '!.*!',
    'exec' => function ($matches) {
      include PAGE_DIR . '/sorry.php'; }
  ],
];
```

4. Next, we define our `Router` class. We first define constants and properties that will be of use during the process of examining and matching a route:

```
namespace Application\Routing;
use InvalidArgumentException;
use Psr\Http\Message\ServerRequestInterface;
class Router
{
  const DEFAULT_MATCH = 'default';
  const ERROR_NO_DEF  = 'ERROR: must supply a
default match';
  protected $request;
  protected $requestUri;
  protected $uriParts;
  protected $docRoot;
  protected $config;
  protected $routeMatch;
```

5. The constructor accepts a `ServerRequestInterface` compliant class, the path to the document root, and the configuration file mentioned earlier. Note that we throw an exception if the default configuration is not supplied:

```
public function
__construct(ServerRequestInterface $request,
$docRoot, $config)
{
  $this->config = $config;
  $this->docRoot = $docRoot;
  $this->request = $request;
  $this->requestUri =
```

```
        $request->getServerParams()
['REQUEST_URI'];
    $this->uriParts = explode('/', $this-
>requestUri);
    if (!isset($config[self::DEFAULT_MATCH])) {
        throw new InvalidArgumentException(
          self::ERROR_NO_DEF);
    }
}
```

6. Next, we have a series of getters that allow us to retrieve the original request, document root, and final route match:

```
public function getRequest()
{
  return $this->request;
}
public function getDocRoot()
{
  return $this->docRoot;
}
public function getRouteMatch()
{
  return $this->routeMatch;
}
```

7. The `isFileOrDir()` method is used to determine whether we are trying to match against a CSS, JavaScript, or graphic request (among other possibilities):

```
public function isFileOrDir()
{
  $fn = $this->docRoot . '/' . $this-
>requestUri;
  $fn = str_replace('//', '/', $fn);
  if (file_exists($fn)) {
      return $fn;
  } else {
      return '';
  }
}
```

8. Finally we define `match()`, which iterates through the configuration array and runs the `uri` parameter through `preg_match()`. If positive, the configuration key and `$matches` array populated by `preg_match()` are stored in `$routeMatch`, and

the callback is returned. If there is no match, the default callback is returned:

```
public function match()
{
  foreach ($this->config as $key => $route) {
    if (preg_match($route['uri'],
        $this->requestUri, $matches)) {
        $this->routeMatch['key'] = $key;
        $this->routeMatch['match'] = $matches;
        return $route['exec'];
    }
  }
  return $this->config[self::DEFAULT_MATCH]
['exec'];
}
}
```

How it works...

First, change to `/path/to/source/for/this/chapter` and create a directory called `routing`. Next, define a file, `index.php`, which sets up autoloading and uses the right classes. You can define a constant `PAGE_DIR` that points to the `pages` directory created in the previous recipe:

```
<?php
define('DOC_ROOT', __DIR__);
define('PAGE_DIR', DOC_ROOT . '/../pages');

require_once __DIR__ .
'/../../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/../..');
use Application\MiddleWare\ServerRequest;
use Application\Routing\Router;
```

Next, add the configuration array discussed in step 3 of this recipe. Note that you could add `(/)?` at the end of the pattern to account for an optional trailing slash. Also, for the `home` route, you could offer two options: either `/` or `/home`:

```
$config = [
  'home' => [
    'uri' => '!^(/|/home)$!',
    'exec' => function ($matches) {
      include PAGE_DIR . '/page0.php'; }
  ],
  'page' => [
    'uri' => '!^/(page)/(\d+)(/)?$!',
    'exec' => function ($matches) {
      include PAGE_DIR . '/page' . $matches[2] .
'.php'; }
  ],
  Router::DEFAULT_MATCH => [
    'uri' => '!.*!',
    'exec' => function ($matches) {
      include PAGE_DIR . '/sorry.php'; }
  ],
];
```

You can then define a router instance, supplying an initialized `ServerRequest` instance as the first argument:

```
$router = new Router((new ServerRequest())
  ->initialize(), DOC_ROOT, $config);
$execute = $router->match();
$params  = $router->getRouteMatch()['match'];
```

You then need to check to see whether the request is a file or directory, and also whether the route match is `/`:

```
if ($fn = $router->isFileOrDir()
    && $router->getRequest()->getUri()->getPath()
!= '/') {
    return FALSE;
} else {
    include DOC_ROOT . '/main.php';
}
```

Next, define `main.php`, something like this:

```
<?php // demo using middleware for routing ?>
<!DOCTYPE html>
```

```
<head>
  <title>PHP 7 Cookbook</title>
  <meta http-equiv="content-type"
  content="text/html;charset=utf-8" />
</head>
<body>
    <?php include PAGE_DIR . '/route_menu.php'; ?>
    <?php $execute($params); ?>
</body>
</html>
```

And finally, a revised menu that uses user-friendly routing is required:

```
<?php // menu for routing ?>
<a href="/home">Home</a>
<a href="/page/1">Page 1</a>
<a href="/page/2">Page 2</a>
<a href="/page/3">Page 3</a>
<!-- etc. -->
```

To test the configuration using Apache, define a virtual host definition that points to `/path/to/source/for/this/chapter/routing`. In addition, define a `.htaccess` file that directs any request that is not a file, directory, or link to `index.php`. Alternatively, you could just use the built-in PHP webserver. In a terminal window or command prompt, type this command:

```
cd /path/to/source/for/this/chapter/routing
php -S localhost:8080
```

In a browser, the output when requesting `http://localhost:8080/home` is something like this:



See also

For information on rewriting using the **NGINX** web server, have a look at this article: http://nginx.org/en/docs/http/ngx_http_rewrite_module.html. There are plenty of sophisticated PHP routing libraries available that introduce far greater functionality than the simple router presented here. These include Altorouter (http://altorouter.com/), TreeRoute (https://github.com/baryshev/TreeRoute), FastRoute (https://github.com/nikic/FastRoute), and Aura.Router. (https://github.com/auraphp/Aura.Router). In addition, most frameworks (for example, Zend Framework 2 or CodeIgniter) have their own routing capabilities.

Making inter-framework system calls

One of the primary reasons for the development of PSR-7 (and middleware) was a growing need to make calls between frameworks. It is of interest to note that the main documentation for PSR-7 is hosted by **PHP Framework Interop Group** (**PHP-FIG**).

How to do it...

1. The primary mechanism used in middleware inter-framework calls is to create a driver program that executes framework calls in succession, maintaining a common request and response object. The request and response objects are expected to represent `Psr\Http\Message\ServerRequestInterface` and `Psr\Http\Message\ResponseInterface`, respectively.
2. For the purposes of this illustration, we define a middleware session validator. The constants and properties reflect the session `thumbprint`, which is a term we use to incorporate factors such as the website visitor's IP address, browser, and language settings:

```
namespace Application\MiddleWare\Session;
use InvalidArgumentException;
use Psr\Http\Message\ {
  ServerRequestInterface, ResponseInterface };
use Application\MiddleWare\ { Constants,
Response, TextStream };
class Validator
{
  const KEY_TEXT = 'text';
  const KEY_SESSION = 'thumbprint';
  const KEY_STATUS_CODE = 'code';
  const KEY_STATUS_REASON = 'reason';
  const KEY_STOP_TIME = 'stop_time';
  const ERROR_TIME = 'ERROR: session has
exceeded stop time';
  const ERROR_SESSION = 'ERROR: thumbprint
```

```
does not match';
  const SUCCESS_SESSION = 'SUCCESS: session
validates OK';
  protected $sessionKey;
  protected $currentPrint;
  protected $storedPrint;
  protected $currentTime;
  protected $storedTime;
```

3. The constructor takes a `ServerRequestInterface` instance and the session as arguments. If the session is an array (such as `$_SESSION`), we wrap it in a class. The reason why we do this is in case we are passed a session object, such as `JSession` used in Joomla. We then create the thumbprint using the previously mentioned factors. If the stored thumbprint is not available, we assume this is the first time, and store the current print as well as stop time, if this parameter is set. We used `md5()` because it's a fast hash, is not exposed externally, and is therefore useful to this application:

```
public function __construct(
  ServerRequestInterface $request, $stopTime =
NULL)
{
  $this->currentTime  = time();
  $this->storedTime   =
$_SESSION[self::KEY_STOP_TIME] ?? 0;
  $this->currentPrint =
    md5($request->getServerParams()
['REMOTE_ADDR']
      . $request->getServerParams()
['HTTP_USER_AGENT']
      . $request->getServerParams()
['HTTP_ACCEPT_LANGUAGE']);
        $this->storedPrint  =
$_SESSION[self::KEY_SESSION]
      ?? NULL;
  if (empty($this->storedPrint)) {
      $this->storedPrint = $this-
>currentPrint;
      $_SESSION[self::KEY_SESSION] = $this-
>storedPrint;
      if ($stopTime) {
          $this->storedTime = $stopTime;
          $_SESSION[self::KEY_STOP_TIME] =
```

```
$stopTime;
        }
    }
}
```

4. It's not required to define `__invoke()`, but this magic method is quite convenient for standalone middleware classes. As is the convention, we accept `ServerRequestInterface` and `ResponseInterface` instances as arguments. In this method, we simply check to see whether the current thumbprint matches the one stored. The first time, of course, they will match. But on subsequent requests, the chances are an attacker intent on session hijacking will be caught out. In addition, if the session time exceeds the stop time (if set), likewise, a `401` code will be sent:

```
public function __invoke(
  ServerRequestInterface $request, Response
$response)
{
  $code = 401;  // unauthorized
  if ($this->currentPrint != $this-
>storedPrint) {
      $text[self::KEY_TEXT] =
self::ERROR_SESSION;
      $text[self::KEY_STATUS_REASON] =
        Constants::STATUS_CODES[401];
  } elseif ($this->storedTime) {
      if ($this->currentTime > $this-
>storedTime) {
          $text[self::KEY_TEXT] =
self::ERROR_TIME;
          $text[self::KEY_STATUS_REASON] =
            Constants::STATUS_CODES[401];
      } else {
          $code = 200; // success
      }
  }
  if ($code == 200) {
      $text[self::KEY_TEXT] =
self::SUCCESS_SESSION;
      $text[self::KEY_STATUS_REASON] =
        Constants::STATUS_CODES[200];
  }
  $text[self::KEY_STATUS_CODE] = $code;
```

```
    $body = new TextStream(json_encode($text));
    return $response->withStatus($code)-
>withBody($body);
}
```

5. We can now put our new middleware class to use. The main problems with inter-framework calls, at least at this point, are summarized here. Accordingly, how we implement middleware depends heavily on the last point:

    - Not all PHP frameworks are PSR-7-compliant
    - Existing PSR-7 implementations are not complete
    - All frameworks want to be the "boss"

6. As an example, have a look at the configuration files for **Zend Expressive**, which is a self-proclaimed *PSR7 Middleware Microframework*. Here is the file, `middleware-pipeline.global.php`, which is located in the `config/autoload` folder in a standard Expressive application. The dependencies key is used to identify the middleware wrapper classes that will be activated in the pipeline:

```
<?php
use
Zend\Expressive\Container\ApplicationFactory;
use Zend\Expressive\Helper;
return [
  'dependencies' => [
     'factories' => [
        Helper\ServerUrlMiddleware::class =>

Helper\ServerUrlMiddlewareFactory::class,
        Helper\UrlHelperMiddleware::class =>

Helper\UrlHelperMiddlewareFactory::class,
        // insert your own class here
     ],
  ],
```

7. Under the `middleware_pipline` key, you can identify classes that will be executed before or after the routing process occurs. Optional parameters include `path`, `error`, and `priority`:

```
'middleware_pipeline' => [
   'always' => [
      'middleware' => [
         Helper\ServerUrlMiddleware::class,
      ],
      'priority' => 10000,
   ],
   'routing' => [
      'middleware' => [

ApplicationFactory::ROUTING_MIDDLEWARE,
         Helper\UrlHelperMiddleware::class,
         // insert reference to middleware
here

ApplicationFactory::DISPATCH_MIDDLEWARE,
      ],
      'priority' => 1,
   ],
   'error' => [
      'middleware' => [
         // Add error middleware here.
      ],
      'error'    => true,
      'priority' => -10000,
    ],
  ],
];
```

8. Another technique is to modify the source code of an existing framework module, and make a request to a PSR-7-compliant middleware application. Here is an example modifying a **Joomla!** installation to include a middleware session validator.
9. Next, add this code the end of the `index.php` file in the `/path/to/joomla` folder. Since Joomla! uses Composer, we can leverage the Composer autoloader:

```
session_start();    // to support use of
$_SESSION
$loader = include __DIR__ .
'/libraries/vendor/autoload.php';
$loader->add('Application', __DIR__ .
'/libraries/vendor');
$loader->add('Psr', __DIR__ .
'/libraries/vendor');
```

10. We can then create an instance of our middleware session validator, and make a validation request just before `$app = JFactory::getApplication('site');`:

```
$session = JFactory::getSession();
$request =
  (new
Application\MiddleWare\ServerRequest())-
>initialize();
$response = new
Application\MiddleWare\Response();
$validator = new
Application\Security\Session\Validator(
  $request, $session);
$response = $validator($request, $response);
if ($response->getStatusCode() != 200) {
  // take some action
}
```

How it works...

First, create the `Application\MiddleWare\Session\Validator` test middleware class described in steps 2-5. Then you will need to go to https://getcomposer.org/ and follow the directions to obtain Composer. Download it to the `/path/to/source/for/this/chapter` folder. Next, build a basic Zend Expressive application, as shown next. Be sure to select `No` when prompted for minimal skeleton:

```
cd /path/to/source/for/this/chapter
php composer.phar create-project
zendframework/zend-expressive-skeleton expressive
```

This will create a `folder /path/to/source/for/this/chapter/expressive`. Change to this directory. Modify `public/index.php` as follows:

```
<?php
if (php_sapi_name() === 'cli-server'
    && is_file(__DIR__ . parse_url(
$_SERVER['REQUEST_URI'], PHP_URL_PATH))
) {
```

```
    return false;
}
chdir(dirname(__DIR__));
session_start();
$_SESSION['time'] = time();
$appDir = realpath(__DIR__ . '/../../..');
$loader = require 'vendor/autoload.php';
$loader->add('Application', $appDir);
$container = require 'config/container.php';
$app = $container-
>get(\Zend\Expressive\Application::class);
$app->run();
```

You will then need to create a wrapper class that invokes our session validator middleware. Create a `SessionValidateAction.php` file that needs to go in the `/path/to/source/for/this/chapter/expressive/src/App/Action` folder. For the purposes of this illustration, set the stop time parameter to a short duration. In this case, `time() + 10` gives you 10 seconds:

```
namespace App\Action;
use Application\MiddleWare\Session\Validator;
use Zend\Diactoros\ { Request, Response };
use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface;
class SessionValidateAction
{
  public function __invoke(ServerRequestInterface
$request,
  ResponseInterface $response, callable $next =
null)
  {
    $inbound   = new Response();
    $validator = new Validator($request,
time()+10);
    $inbound   = $validator($request, $response);
    if ($inbound->getStatusCode() != 200) {
        session_destroy();
        setcookie('PHPSESSID', 0, time()-300);
        $params = json_decode(
          $inbound->getBody()->getContents(),
TRUE);
        echo
'<h1>',$params[Validator::KEY_TEXT],'</h1>';
```

```
            echo '<pre>',var_dump($inbound),'</pre>';
            exit;
        }
        return $next($request,$response);
    }
}
```

You will now need to add the new class to the middleware pipeline. Modify `config/autoload/middleware-pipeline.global.php` as follows. Modifications are shown in **bold**:

```
<?php
use Zend\Expressive\Container\ApplicationFactory;
use Zend\Expressive\Helper;
return [
  'dependencies' => [
     'invokables' => [
        App\Action\SessionValidateAction::class =>
        App\Action\SessionValidateAction::class,
     ],
   'factories' => [
      Helper\ServerUrlMiddleware::class =>
      Helper\ServerUrlMiddlewareFactory::class,
      Helper\UrlHelperMiddleware::class =>
      Helper\UrlHelperMiddlewareFactory::class,
    ],
  ],
  'middleware_pipeline' => [
      'always' => [
         'middleware' => [
            Helper\ServerUrlMiddleware::class,
         ],
         'priority' => 10000,
      ],
      'routing' => [
         'middleware' => [

ApplicationFactory::ROUTING_MIDDLEWARE,
            Helper\UrlHelperMiddleware::class,

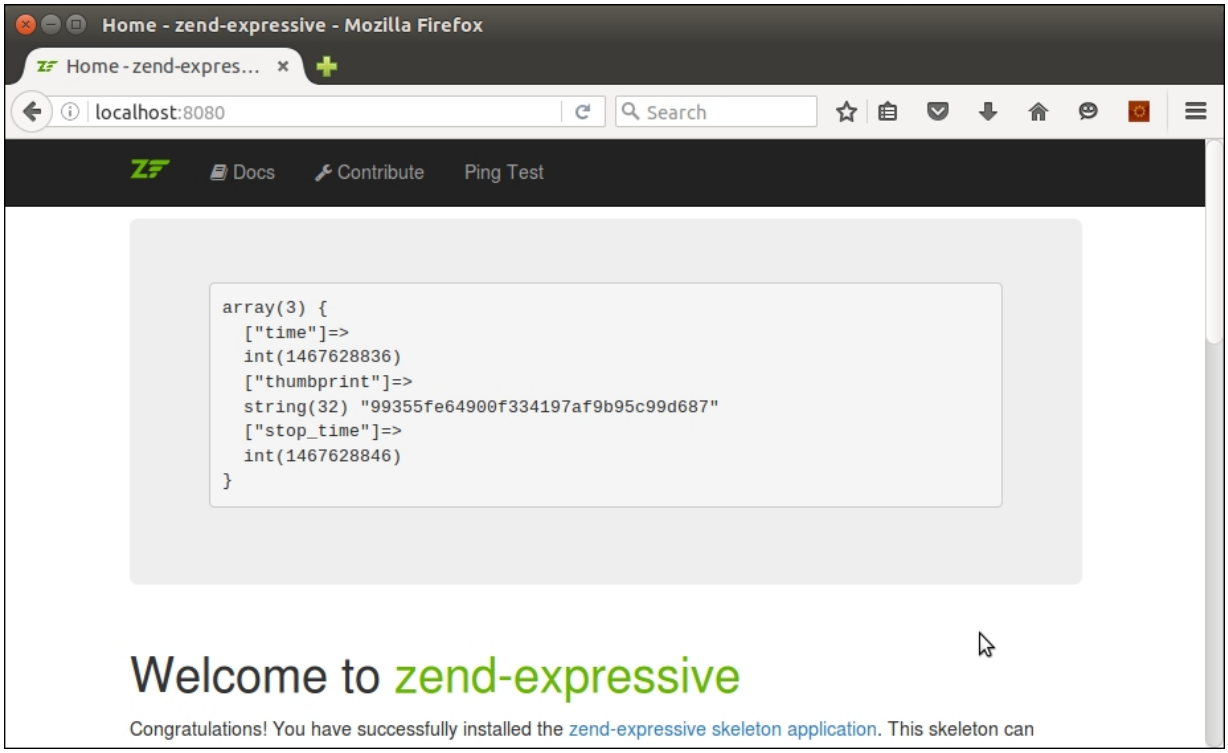
App\Action\SessionValidateAction::class,

ApplicationFactory::DISPATCH_MIDDLEWARE,
         ],
         'priority' => 1,
      ],
```

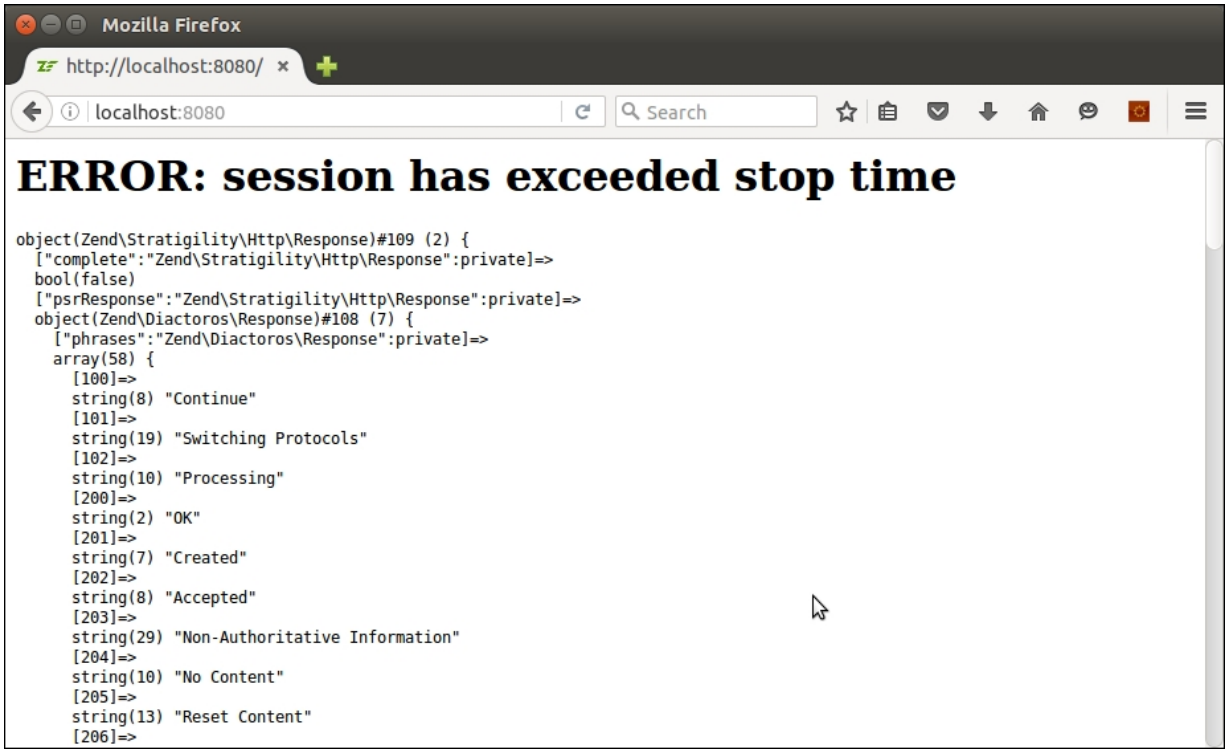
```
        'error' => [
            'middleware' => [
                // Add error middleware here.
            ],
            'error'    => true,
            'priority' => -10000,
        ],
    ],
];
```

You might also consider modifying the home page template to show the status of `$_SESSION`. The file in question is `/path/to/source/for/this/chapter/expressive/templates/app/home-page.phtml`. Simply adding `var_dump($_SESSION)` should suffice.

Initially, you should see something like this:



After 10 seconds, refresh the browser. You should now see this:


Mozilla Firefox
http://localhost:8080/
localhost:8080
Search
ERROR: session has exceeded stop time
object(Zend\Stratigility\Http\Response)#109 (2) {
["complete":"Zend\Stratigility\Http\Response":private]=>
bool(false)
["psrResponse":"Zend\Stratigility\Http\Response":private]=>
object(Zend\Diactoros\Response)#108 (7) {
["phrases":"Zend\Diactoros\Response":private]=>
array(58) {
[100]=>
string(8) "Continue"
[101]=>
string(19) "Switching Protocols"
[102]=>
string(10) "Processing"
[200]=>
string(2) "OK"
[201]=>
string(7) "Created"
[202]=>
string(8) "Accepted"
[203]=>
string(29) "Non-Authoritative Information"
[204]=>
string(10) "No Content"
[205]=>
string(13) "Reset Content"
[206]=>

Using middleware to cross languages

Except in cases where you are trying to communicate between different versions of PHP, PSR-7 middleware will be of minimal use. Recall what the acronym stands for: **PHP Standards Recommendations**. Accordingly, if you need to make a request to an application written in another language, treat it as you would any other web service HTTP request.

How to do it...

1. In the case of PHP 4, you actually have a chance in that there is limited support for object-oriented programming. Accordingly, the best approach would be to downgrade the basic PSR-7 classes described in the first three recipes. There is not enough space to cover all the changes, but we present a potential PHP 4 version of `Application\MiddleWare\ServerRequest`. The first thing to note is that there are no namespaces! Accordingly, we use a classname with underscores, _, in place of namespace separators:

```
class Application_MiddleWare_ServerRequest
extends Application_MiddleWare_Request
implements
Psr_Http_Message_ServerRequestInterface
{
```

2. All properties are identified in PHP 4 using the key word `var`:

```
var $serverParams;
var $cookies;
var $queryParams;
// not all properties are shown
```

3. The `initialize()` method is almost the same, except that syntax such as `$this->getServerParams()['REQUEST_URI']` was

not allowed in PHP 4. Accordingly, we need to split this out into a separate variable:

```
function initialize()
{
  $params = $this->getServerParams();
  $this->getCookieParams();
  $this->getQueryParams();
  $this->getUploadedFiles;
  $this->getRequestMethod();
  $this->getContentType();
  $this->getParsedBody();
  return $this-
>withRequestTarget($params['REQUEST_URI']);
}
```

4. All of the `$_XXX` super-globals were present in later versions of PHP 4:

```
function getServerParams()
{
  if (!$this->serverParams) {
      $this->serverParams = $_SERVER;
  }
  return $this->serverParams;
}
// not all getXXX() methods are shown to
conserve space
```

5. The null coalesce operator was only introduced in PHP 7. We need to use `isset(XXX) ? XXX : '';` instead:

```
function getRequestMethod()
{
  $params = $this->getServerParams();
  $method = isset($params['REQUEST_METHOD'])
    ? $params['REQUEST_METHOD'] : '';
  $this->method = strtolower($method);
  return $this->method;
}
```

6. The JSON extension was not introduced until PHP 5. Accordingly, we need to be satisfied with raw input. We could also possibly use `serialize()` or `unserialize()` in place of `json_encode()` and `json_decode()`:

```
function getParsedBody()
{
  if (!$this->parsedBody) {
      if (($this->getContentType() ==

Constants::CONTENT_TYPE_FORM_ENCODED
           || $this->getContentType() ==
           Constants::CONTENT_TYPE_MULTI_FORM)
           && $this->getRequestMethod() ==
           Constants::METHOD_POST)
      {
          $this->parsedBody = $_POST;
      } elseif ($this->getContentType() ==
                Constants::CONTENT_TYPE_JSON
                || $this->getContentType() ==

Constants::CONTENT_TYPE_HAL_JSON)
      {
          ini_set("allow_url_fopen", true);
          $this->parsedBody =
            file_get_contents('php://stdin');
      } elseif (!empty($_REQUEST)) {
          $this->parsedBody = $_REQUEST;
      } else {
          ini_set("allow_url_fopen", true);
          $this->parsedBody =
            file_get_contents('php://stdin');
      }
  }
  return $this->parsedBody;
}
```

7. The `withXXX()` methods work pretty much the same in PHP 4:

```
function withParsedBody($data)
{
  $this->parsedBody = $data;
  return $this;
}
```

8. Likewise, the `withoutXXX()` methods work the same as well:

```
function withoutAttribute($name)
{
  if (isset($this->attributes[$name])) {
      unset($this->attributes[$name]);
  }
```

```
    return $this;
  }

  }
```

9. For websites using other languages, we could use the PSR-7 classes to formulate requests and responses, but would then need to use an HTTP client to communicate with the other website. As an example, recall the demonstration of a `Request` discussed in the recipe *Developing a PSR-7 request class* from this chapter. Here is the example from the *How it works...* section:

```
$request = new Request(
  TARGET_WEBSITE_URL,
  Constants::METHOD_POST,
  new TextStream($contents),
  [Constants::HEADER_CONTENT_TYPE =>
  Constants::CONTENT_TYPE_FORM_ENCODED,
  Constants::HEADER_CONTENT_LENGTH => $body-
>getSize()]
);

$data = http_build_query(['data' =>
$request->getBody()->getContents()]);

$defaults = array(
  CURLOPT_URL => $request->getUri()-
>getUriString(),
  CURLOPT_POST => true,
  CURLOPT_POSTFIELDS => $data,
);
$ch = curl_init();
curl_setopt_array($ch, $defaults);
$response = curl_exec($ch);
curl_close($ch);
```

Chapter 10. Looking at Advanced Algorithms

In this chapter, we will cover:

- Using getters and setters
- Implementing a linked list
- Building a bubble sort
- Implementing a stack
- Building a binary search class
- Implementing a search engine
- Displaying a multi-dimensional array and accumulating totals

Introduction

In this chapter, we cover recipes that implement various advanced algorithms such as linked list, bubble sort, stacks, and binary search. In addition, we cover getters and setters, as well as implementing a search engine and displaying values from a multi-dimensional array with accumulated totals.

Using getters and setters

At first glance, it would seemingly make sense to define classes with `public` properties, which can then be directly read or written. It is considered a best practice, however, to make properties `protected`, and to then define a **getter** and **setter** for each. As the name implies, a *getter* retrieves the value of a property. A *setter* is used to set the value.

> ## Tip
>
> **Best practice**
>
> Define properties as `protected` to prevent accidental *outside* access. Use `public` get* and set* methods to provide access to these properties. In this manner, not only can you more precisely control access, but you can also make formatting and data type changes to the properties while getting and setting them.

How to do it...

1. Getters and setters provide additional flexibility when getting or setting values. You are able to add an additional layer of logic if needed, something which would not be possible if you were to directly read or write a public property. All you need to do is to create a public method with a prefix of either `get` or `set`. The name of the property becomes the suffix. It is a convention to make the first letter of the variable uppercase. Thus, if the property is `$testValue`, the getter would be `getTestValue()`.
2. In this example, we define a class with a protected property, `$date`. Notice that the `get` and `set` methods allow for treatment as either a `DateTime` object or as a string. The value is actually stored in any event as a `DateTime` instance:

```
$a = new class() {
  protected $date;
  public function setDate($date)
  {
    if (is_string($date)) {
        $this->date = new DateTime($date);
    } else {
        $this->date = $date;
    }
  }
  public function getDate($asString = FALSE)
  {
    if ($asString) {
        return $this->date->format('Y-m-d
H:i:s');
    } else {
        return $this->date;
    }
  }
};
```

3. Getters and setters allow you to filter or sanitize the data coming in or going out. In the following example, there are two properties, `$intVal` and `$arrVal`, which are set to a default initial value of `NULL`. Notice that not only are the return values for the getters data-typed, but they also provide defaults. The setters also either enforce the incoming data-type, or type-cast the incoming value to a certain data-type:

```
<?php
class GetSet
{
  protected $intVal = NULL;
  protected $arrVal = NULL;
  // note the use of the null coalesce
operator to return a default value
  public function getIntVal() : int
  {
    return $this->intVal ?? 0;
  }
  public function getArrVal() : array
  {
    return $this->arrVal ?? array();
  }
  public function setIntVal($val)
```

```
    {
      $this->intVal = (int) $val ?? 0;
    }
    public function setArrVal(array $val)
    {
      $this->arrVal = $val ?? array();
    }
}
```

4. If you have a class with lots and lots of properties, it might become tedious to define a distinct getter and setter for each property. In this case, you can define a kind of *fallback* using the magic method `__call()`. The following class defines nine different properties. Instead of having to define nine getters and nine setters, we define a single method, `__call()`, which makes a determination whether or not the usage is `get` or `set`. If `get`, it retrieves the key from an internal array. If `set`, it stores the value in the internal array.

> ## Note
>
> The `__call()` method is a magic method which is executed if an application makes a call to a non-existent method.

```
<?php
class LotsProps
{
  protected $firstName  = NULL;
  protected $lastName   = NULL;
  protected $addr1      = NULL;
  protected $addr2      = NULL;
  protected $city       = NULL;
  protected $state      = NULL;
  protected $province   = NULL;
  protected $postalCode = NULL;
  protected $country    = NULL;
  protected $values     = array();

  public function __call($method, $params)
  {
```

```
        preg_match('/^(get|set)(.*?)$/i', $method,
    $matches);
        $prefix = $matches[1] ?? '';
        $key    = $matches[2] ?? '';
        $key    = strtolower($key);
        if ($prefix == 'get') {
            return $this->values[$key] ?? '---';
        } else {
            $this->values[$key] = $params[0];
        }
      }
    }
```

How it works...

Copy the code mentioned in step 1 into a new file, `chap_10_oop_using_getters_and_setters.php`. To test the class, add the following:

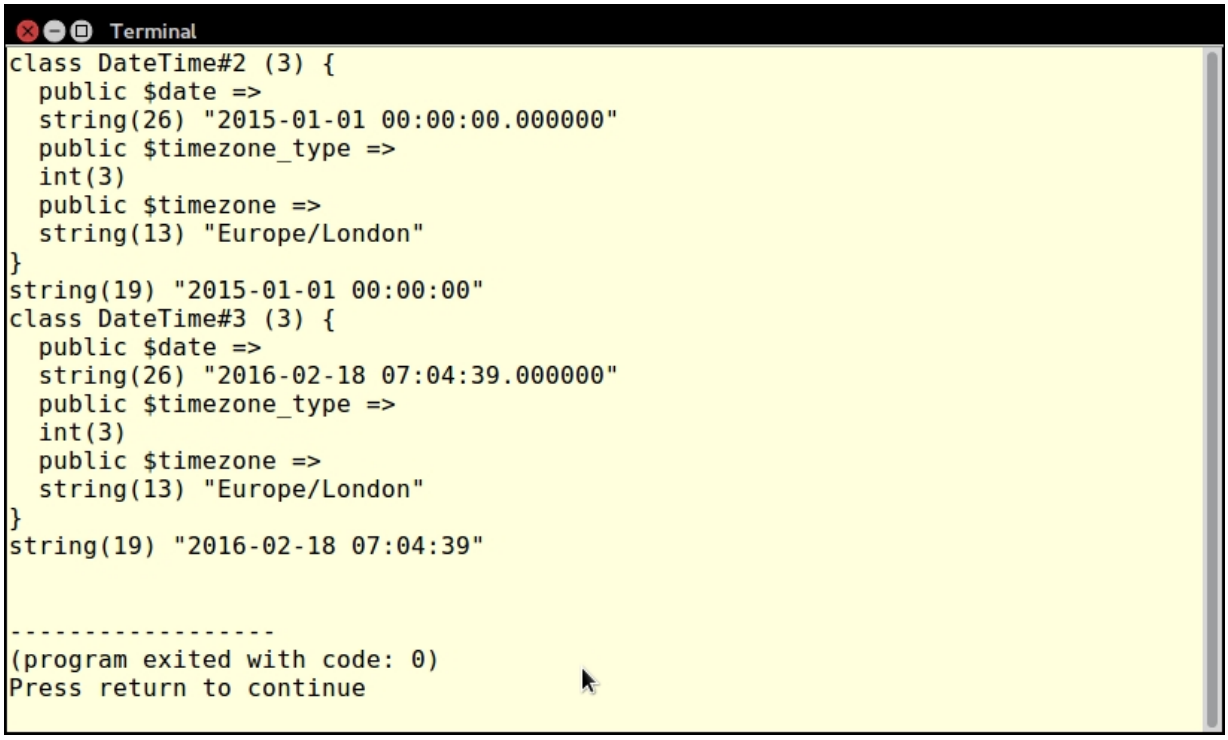
```
// set date using a string
$a->setDate('2015-01-01');
var_dump($a->getDate());

// retrieves the DateTime instance
var_dump($a->getDate(TRUE));

// set date using a DateTime instance
$a->setDate(new DateTime('now'));
var_dump($a->getDate());

// retrieves the DateTime instance
var_dump($a->getDate(TRUE));
```

In the output (shown next), you can see that the `$date` property can be set using either a `string` or an actual `DateTime` instance. When `getDate()` is executed, you can return either a `string` or a `DateTime` instance, depending on the value of the `$asString` flag:



Next, have a look at the code defined in step 2. Copy this code into a file, `chap_10_oop_using_getters_and_setters_defaults.php`, and add the following:

```
// create the instance
$a = new GetSet();

// set a "proper" value
$a->setIntVal(1234);
echo $a->getIntVal();
echo PHP_EOL;

// set a bogus value
$a->setIntVal('some bogus value');
echo $a->getIntVal();
echo PHP_EOL;

// NOTE: boolean TRUE == 1
$a->setIntVal(TRUE);
echo $a->getIntVal();
echo PHP_EOL;

// returns array() even though no value was set
var_dump($a->getArrVal());
echo PHP_EOL;
```

```
// sets a "proper" value
$a->setArrVal(['A','B','C']);
var_dump($a->getArrVal());
echo PHP_EOL;

try {
    $a->setArrVal('this is not an array');
    var_dump($a->getArrVal());
    echo PHP_EOL;
} catch (TypeError $e) {
    echo $e->getMessage();
}

echo PHP_EOL;
```

As you can see from the following output, setting a *proper* integer value works as expected. A non-numeric value defaults to `0`. Interestingly, if you supply a Boolean `TRUE` as an argument to `setIntVal()`, it is interpolated to `1`.

If you call `getArrVal()` without setting a value, the default is an empty array. Setting an array value works as expected. However, if you supply a non-array value as an argument, the type hint of the array causes a `TypeError` to be thrown, which can be caught as shown here:



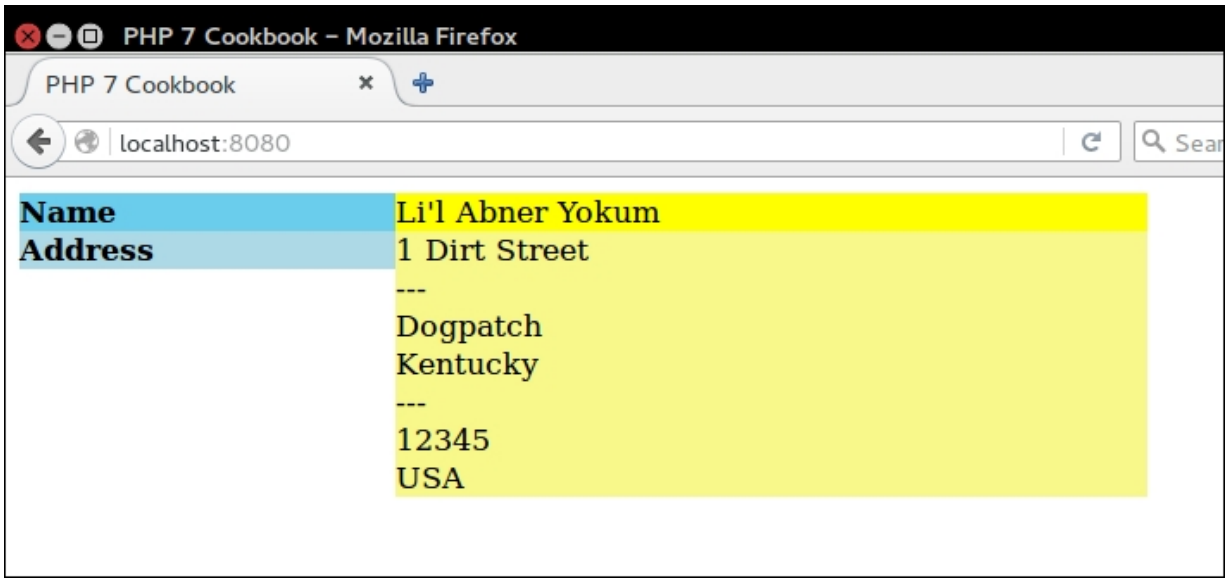
Finally, take the `LotsProps` class defined in step 3 and place it in a separate file, `chap_10_oop_using_getters_and_setters_magic_call.php`. Now add code to set values. What will happen, of course, is that the magic method `__call()` is invoked. After running `preg_match()`, the remainder of the non-existent property, after the letters `set`, will become a key in the internal array `$values`:

```
$a = new LotsProps();
$a->setFirstName('Li\'l Abner');
$a->setLastName('Yokum');
$a->setAddr1('1 Dirt Street');
$a->setCity('Dogpatch');
$a->setState('Kentucky');
$a->setPostalCode('12345');
$a->setCountry('USA');
?>
```

You can then define HTML that displays the values using the corresponding `get` methods. These will in turn return keys from the internal array:

```
<div class="container">
<div class="left blue1">Name</div>
<div class="right yellow1">
<?= $a->getFirstName() . ' ' . $a->getLastName() ?
></div>
</div>
<div class="left blue2">Address</div>
<div class="right yellow2">
    <?= $a->getAddr1() ?>
    <br><?= $a->getAddr2() ?>
    <br><?= $a->getCity() ?>
    <br><?= $a->getState() ?>
    <br><?= $a->getProvince() ?>
    <br><?= $a->getPostalCode() ?>
    <br><?= $a->getCountry() ?>
</div>
</div>
```

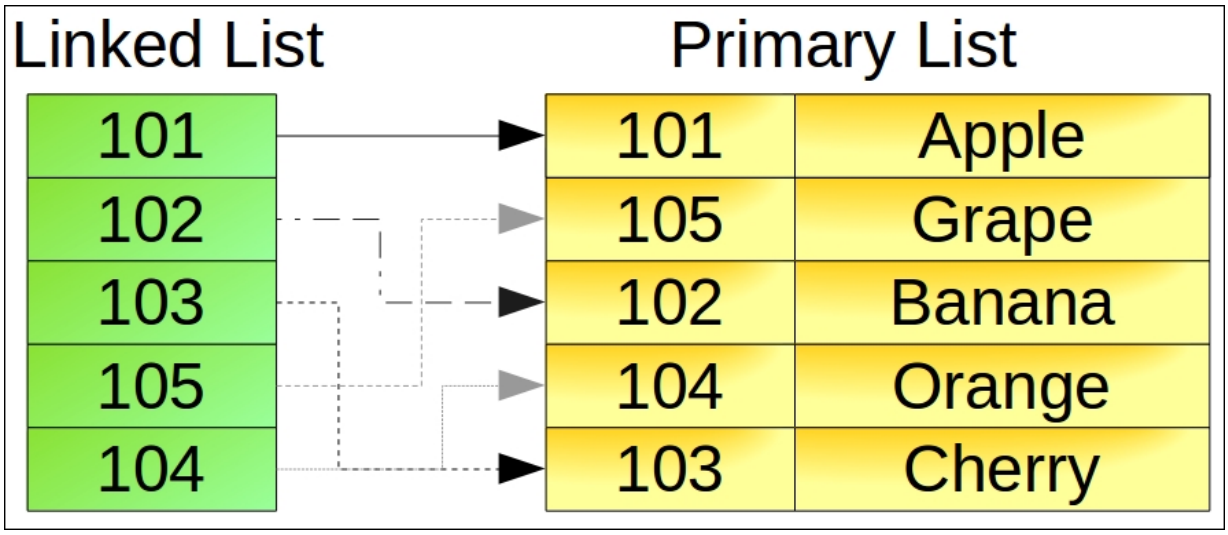
Here is the final output:



Implementing a linked list

A linked list is where one list contains keys that point to keys in another list. An analogy, in database terms, would be where you have a table that contains data, and a separate index that points to the data. One index might produce a list of items by ID. Another index might yield a list according to title and so on. The salient feature of the linked list is that you do not have to touch the original list of items.

For example, in the diagram shown next, the primary list contains ID numbers and the names of fruits. If you were to directly output the primary list, the fruit names would display in this order: **Apple**, **Grape**, **Banana**, **Orange**, **Cherry**. If you were to use the linked list as an index, on the other hand, the resulting output of fruit names would be **Apple**, **Banana**, **Cherry**, **Grape**, and **Orange**:



How to do it...

1. One of the primary uses of a linked list is to produce a display of items in a different order. One approach would be to create an iteration of key value pairs, where the key represents the new

order, and the value contains the value of the key in the primary list. Such a function might look like this:

```
function buildLinkedList(array $primary,
                         callable $makeLink)
{
  $linked = new ArrayIterator();
  foreach ($primary as $key => $row) {
    $linked->offsetSet($makeLink($row), $key);
  }
  $linked->ksort();
  return $linked;
}
```

2. We use an anonymous function to generate the new key in order to provide extra flexibility. You will also notice that we do a sort by key (`ksort()`) so that the linked list iterates in key order.
3. All we need to do to use the linked list is to iterate through it, but produce results from the primary list, `$customer` in this example:

```
foreach ($linked as $key => $link) {
  $output .= printRow($customer[$link]);
}
```

4. Note that in no way do we touch the primary list. This allows us to generate multiple linked lists, each representing a different order, while retaining our original set of data.
5. Another important use of a linked list is for the purposes of filtering. The technique is similar to that shown previously. The only difference is that we expand the `buildLinkedList()` function, adding a filter column and filter value:

```
function buildLinkedList(array $primary,
                         callable $makeLink,
                         $filterCol = NULL,
                         $filterVal = NULL)
{
  $linked = new ArrayIterator();
  $filterVal = trim($filterVal);
  foreach ($primary as $key => $row) {
    if ($filterCol) {
      if (trim($row[$filterCol]) ==
$filterVal) {
        $linked->offsetSet($makeLink($row),
```

```
$key);
        }
      } else {
        $linked->offsetSet($makeLink($row),
$key);
      }
    }
    $linked->ksort();
    return $linked;
}
```

6. We only include items in the linked list where the value represented by `$filterCol` in the primary list matches `$filterVal`. The iteration logic is the same as that shown in step 2.
7. Finally, another form of linked list is the *doubly* linked list. In this case, the list is constructed in such a manner that the iteration can occur in either a forward or reverse direction. In the case of PHP, we are fortunate to have an SPL class, `SplDoublyLinkedList`, which neatly does the trick. Here is a function that builds a doubly linked list:

```
function buildDoublyLinkedList(ArrayIterator
$linked)
{
  $double = new SplDoublyLinkedList();
  foreach ($linked as $key => $value) {
    $double->push($value);
  }
  return $double;
}
```

Note

The terminology for `SplDoublyLinkedList` can be misleading. `SplDoublyLinkedList::top()` actually points to the *end* of the list, whereas `SplDoublyLinkedList::bottom()` points to the *beginning*!

How it works...

Copy the code shown in the first bullet into a file, `chap_10_linked_list_include.php`. In order to demonstrate the use of a linked list, you will need a source of data. For this illustration, you can make use of the `customer.csv` file that was mentioned in earlier recipes. It is a CSV file with the following columns:

```
"id","name","balance","email","password","status",
"security_question",
"confirm_code","profile_id","level"
```

You can add the following functions to the include file mentioned previously to generate a primary list of customers, and to display information about them. Note that we use the first column, id as the primary key:

```
function readCsv($fn, &$headers)
{
  if (!file_exists($fn)) {
    throw new Error('File Not Found');
  }
  $fileObj = new SplFileObject($fn, 'r');
  $result = array();
  $headers = array();
  $firstRow = TRUE;
  while ($row = $fileObj->fgetcsv()) {
    // store 1st row as headers
    if ($firstRow) {
      $firstRow = FALSE;
      $headers = $row;
    } else {
      if ($row && $row[0] !== NULL && $row[0] !==
0) {
        $result[$row[0]] = $row;
      }
    }
  }
  return $result;
}

function printHeaders($headers)
```

```
{
  return sprintf('%4s : %18s : %8s : %32s : %4s' .
PHP_EOL,
                 ucfirst($headers[0]),
                 ucfirst($headers[1]),
                 ucfirst($headers[2]),
                 ucfirst($headers[3]),
                 ucfirst($headers[9]));
}

function printRow($row)
{
  return sprintf('%4d : %18s : %8.2f : %32s : %4s'
. PHP_EOL,
                 $row[0], $row[1], $row[2],
$row[3], $row[9]);
}

function printCustomer($headers, $linked,
$customer)
{
  $output = '';
  $output .= printHeaders($headers);
  foreach ($linked as $key => $link) {
    $output .= printRow($customer[$link]);
  }
  return $output;
}
```

You can then define a calling program, `chap_10_linked_list_in_order.php`, which includes the file defined previously, and reads `customer.csv`:

```
<?php
define('CUSTOMER_FILE', __DIR__ .
'/../data/files/customer.csv');
include __DIR__ .
'/chap_10_linked_list_include.php';
$headers = array();
$customer = readCsv(CUSTOMER_FILE, $headers);
```

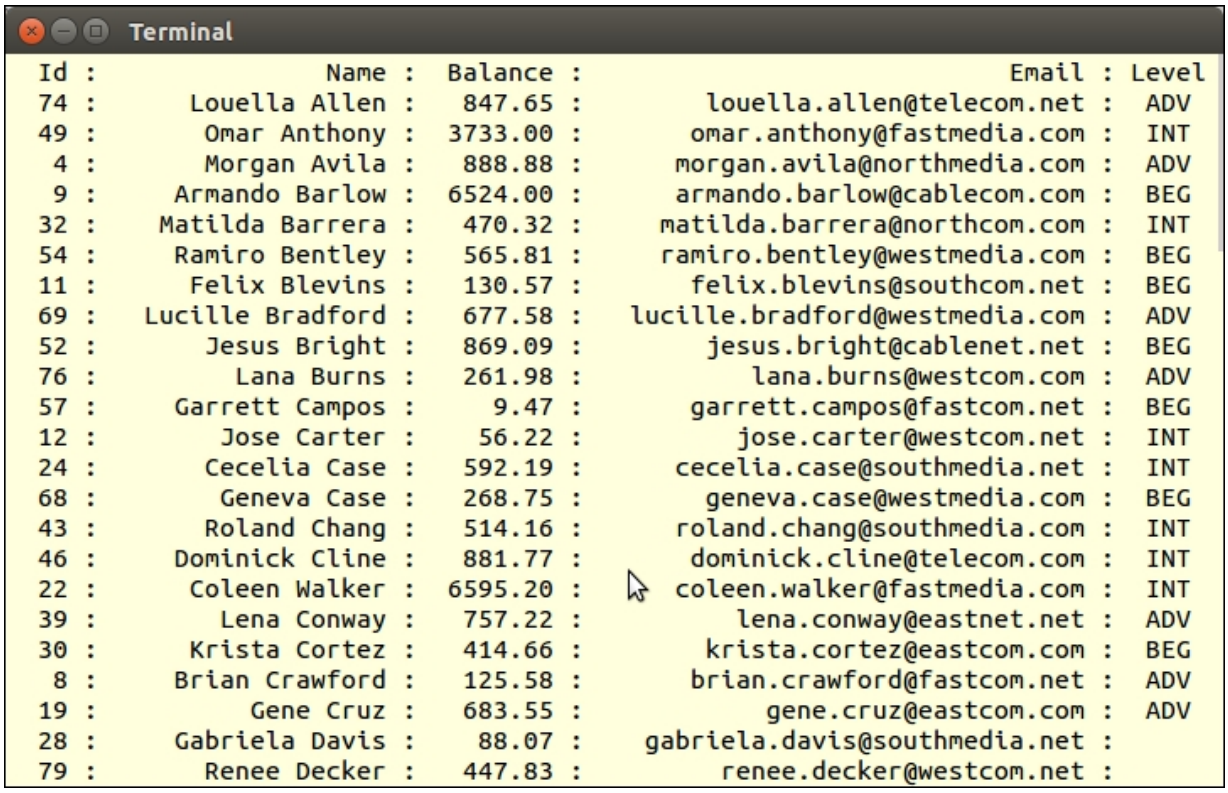
You can then define an anonymous function that will produce a key in the linked list. In this illustration, define a function that breaks down column 1 (name) into first and last names:

```
$makeLink = function ($row) {
  list($first, $last) = explode(' ', $row[1]);
  return trim($last) . trim($first);
};
```

You can then call the function to build the linked list, and use `printCustomer()` to display the results:

```
$linked = buildLinkedList($customer, $makeLink);
echo printCustomer($headers, $linked, $customer);
```

Here is how the output might appear:



To produce a filtered result, modify `buildLinkedList()` as discussed in step 4. You can then add logic that checks to see whether the value of the filter column matches the value in the filter:

```
define('LEVEL_FILTER', 'INT');

$filterCol = 9;
```

```
$filterVal = LEVEL_FILTER;
$linked = buildLinkedList($customer, $makeLink,
$filterCol, $filterVal);
```

There's more...

PHP 7.1 introduced the use of `[ ]` as an alternative to `list()`. If you look at the anonymous function mentioned previously, you could rewrite this in PHP 7.1 as follows:

```
$makeLink = function ($row) {
  [$first, $last] = explode(' ', $row[1]);
  return trim($last) . trim($first);
};
```

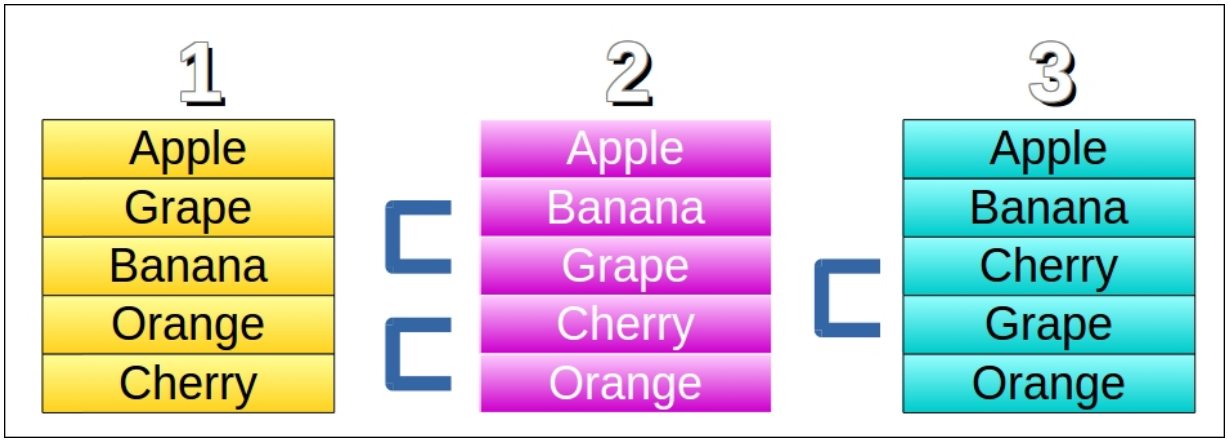
For more information, see https://wiki.php.net/rfc/short_list_syntax.

Building a bubble sort

The classic **bubble sort** is an exercise often assigned to university students. Nonetheless, it's important to master this algorithm as there are many occasions where built-in PHP sorting functions do not apply. An example would be sorting a multi-dimensional array where the sort key is not the first column.

The way the bubble sort works is to recursively iterate through the list and swap the current value with the next value. If you want items to be in ascending order, the swap occurs if the next item is less than the current item. For descending order, the swap occurs if the reverse is true. The sort is concluded when no more swaps occur.

In the following diagram, after the first pass, **Grape** and **Banana** are swapped, as are **Orange** and **Cherry**. After the 2nd pass, **Grape** and **Cherry** are swapped. No more swaps occur on the last pass, and the bubble sort ends:



How to do it...

1. We do not want to actually *move* the values around in the array; that would be horribly expensive in terms of resource usage. Instead, we will use a **linked list**, discussed in the previous recipe.

2. First we build a linked list using the `buildLinkedList()` function discussed in the previous recipe.
3. We then define a new function, `bubbleSort()`, which accepts the linked list by reference, the primary list, a sort field, and a parameter that represents sort order (ascending or descending):

```
function bubbleSort(&$linked, $primary,
$sortField, $order = 'A')
{
```

4. The variables needed include one that represents the number of iterations, the number of swaps, and an iterator based upon the linked list:

```
  static $iterations = 0;
  $swaps = 0;
  $iterator = new ArrayIterator($linked);
```

5. In the `while()` loop, we only proceed if the iteration is still `valid`, which is to say still in progress. We then obtain the current key and value, and the next key and value. Note the extra `if()` statement to ensure the iteration is still valid (that is, to make sure we don't drop off the end of the list!):

```
while ($iterator->valid()) {
  $currentLink = $iterator->current();
  $currentKey  = $iterator->key();
  if (!$iterator->valid()) break;
  $iterator->next();
  $nextLink = $iterator->current();
  $nextKey  = $iterator->key();
```

6. Next we check to see whether the sort is to be ascending or descending. Depending on the direction, we check to see whether the next value is greater than, or less than, the current value. The result of the comparison is stored in `$expr`:

```
if ($order == 'A') {
    $expr = $primary[$linked->offsetGet
            ($currentKey)][$sortField] >
            $primary[$linked-
>offsetGet($nextKey)][$sortField];
} else {
    $expr = $primary[$linked->offsetGet
```

```
                ($currentKey)][$sortField] <
                $primary[$linked-
>offsetGet($nextKey)][$sortField];
}
```

7. If the value of `$expr` is `TRUE`, and we have valid current and next keys, the values are swapped in the linked list. We also increment `$swaps`:

```
if ($expr && $currentKey && $nextKey
    && $linked->offsetExists($currentKey)
    && $linked->offsetExists($nextKey)) {
    $tmp = $linked->offsetGet($currentKey);
    $linked->offsetSet($currentKey,
    $linked->offsetGet($nextKey));
    $linked->offsetSet($nextKey, $tmp);
    $swaps++;
  }
}
```

8. Finally, if any swaps have occurred, we need to run through the iteration again, until there are no more swaps. Accordingly, we make a recursive call to the same method:

```
if ($swaps) bubbleSort($linked, $primary,
$sortField, $order);
```

9. The *real* return value is the re-organized linked list. We also return the number of iterations just for reference:

```
  return ++$iterations;
}
```

How it works...

Add the `bubbleSort()` function discussed previously to the include file created in the previous recipe. You can use the same logic discussed in the previous recipe to read the `customer.csv` file, producing a primary list:

```
<?php
define('CUSTOMER_FILE', __DIR__ .
```

```
'/../data/files/customer.csv');
include __DIR__ .
'/chap_10_linked_list_include.php';
$headers = array();
$customer = readCsv(CUSTOMER_FILE, $headers);
```

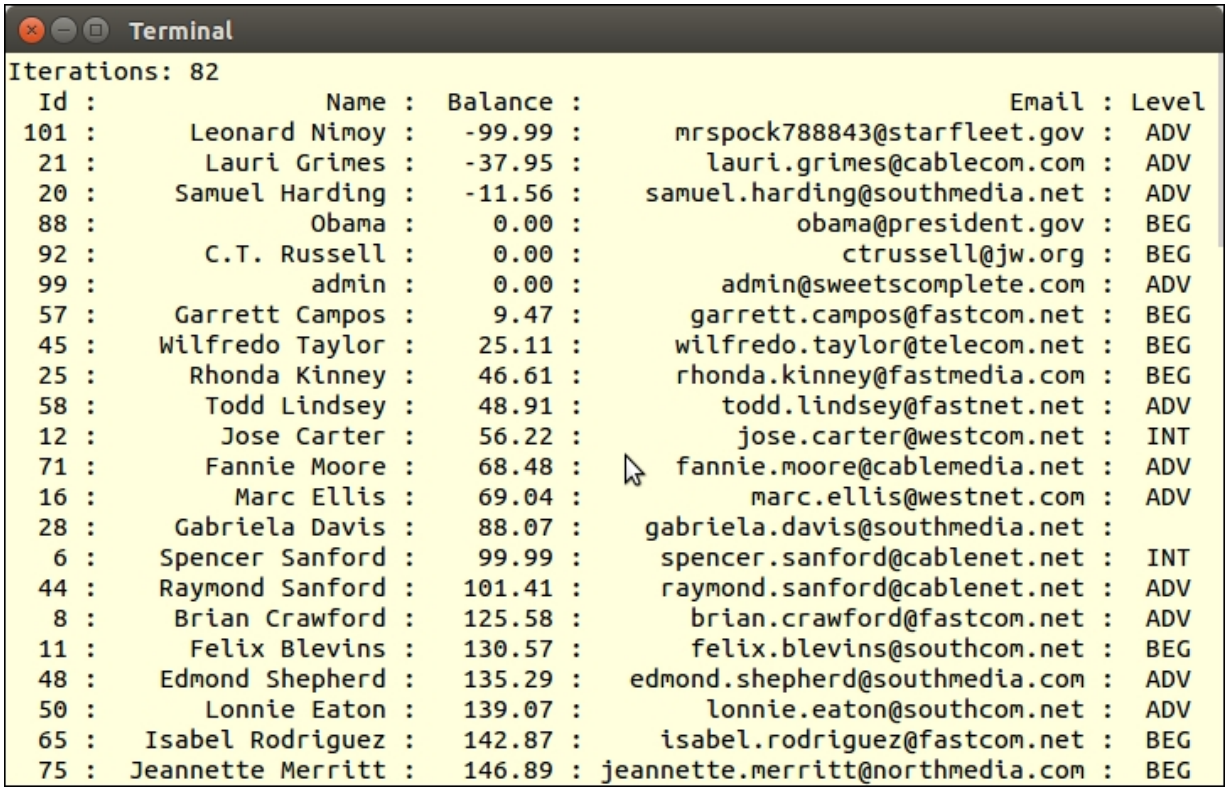
You can then produce a linked list using the first column as a sort key:

```
$makeLink = function ($row) {
  return $row[0];
};
$linked = buildLinkedList($customer, $makeLink);
```

Finally, call the `bubbleSort()` function, providing the linked list and customer list as arguments. You can also provide a sort column, in this illustration column 2, that represents the account balance, using the letter `'A'` to indicate ascending order. The `printCustomer()` function can be used to display output:

```
echo 'Iterations: ' . bubbleSort($linked,
$customer, 2, 'A') . PHP_EOL;
echo printCustomer($headers, $linked, $customer);
```

Here is an example of the output:


Terminal
Iterations: 82
Id : Name : Balance : Email : Level
101 : Leonard Nimoy : -99.99 : mrspock788843@starfleet.gov : ADV
21 : Lauri Grimes : -37.95 : lauri.grimes@cablecom.com : ADV
20 : Samuel Harding : -11.56 : samuel.harding@southmedia.net : ADV
88 : Obama : 0.00 : obama@president.gov : BEG
92 : C.T. Russell : 0.00 : ctrussell@jw.org : BEG
99 : admin : 0.00 : admin@sweetscomplete.com : ADV
57 : Garrett Campos : 9.47 : garrett.campos@fastcom.net : BEG
45 : Wilfredo Taylor : 25.11 : wilfredo.taylor@telecom.net : BEG
25 : Rhonda Kinney : 46.61 : rhonda.kinney@fastmedia.com : BEG
58 : Todd Lindsey : 48.91 : todd.lindsey@fastnet.net : ADV
12 : Jose Carter : 56.22 : jose.carter@westcom.net : INT
71 : Fannie Moore : 68.48 : fannie.moore@cablemedia.net : ADV
16 : Marc Ellis : 69.04 : marc.ellis@westnet.com : ADV
28 : Gabriela Davis : 88.07 : gabriela.davis@southmedia.net :
6 : Spencer Sanford : 99.99 : spencer.sanford@cablenet.net : INT
44 : Raymond Sanford : 101.41 : raymond.sanford@cablenet.net : ADV
8 : Brian Crawford : 125.58 : brian.crawford@fastcom.net : ADV
11 : Felix Blevins : 130.57 : felix.blevins@southcom.net : BEG
48 : Edmond Shepherd : 135.29 : edmond.shepherd@southmedia.com : ADV
50 : Lonnie Eaton : 139.07 : lonnie.eaton@southcom.net : ADV
65 : Isabel Rodriguez : 142.87 : isabel.rodriguez@fastcom.net : BEG
75 : Jeannette Merritt : 146.89 : jeannette.merritt@northmedia.com : BEG

Implementing a stack

A **stack** is a simple algorithm normally implemented as **Last In First Out** (**LIFO**). Think of a stack of books sitting on a library table. When the librarian goes to restore the books to their place, the topmost book is processed first, and so on in order, until the book at the bottom of the stack has been replaced. The topmost book was the last one to be placed on the stack, thus last in first out.

In programming terms, a stack is used to temporarily store information. The retrieval order facilitates retrieving the most recent item first.

How to do it...

1. First we define a class, `Application\Generic\Stack`. The core logic is encapsulated in an SPL class, `SplStack`:

```
namespace Application\Generic;
use SplStack;
class Stack
{
  // code
}
```

2. Next we define a property to represent the stack, and set up an `SplStack` instance:

```
protected $stack;
public function __construct()
{
  $this->stack = new SplStack();
}
```

3. After that we define methods to add and remove from the stack, the classic `push()` and `pop()` methods:

```
public function push($message)
{
  $this->stack->push($message);
```

```
}
public function pop()
{
  return $this->stack->pop();
}
```

4. We also throw in an implementation of `__invoke()` that returns an instance of the `stack` property. This allows us to use the object in a direct function call:

```
public function __invoke()
{
  return $this->stack;
}
```

How it works...

One possible use for a stack is to store messages. In the case of messages, it is usually desirable to retrieve the latest first, thus it is a perfect use case for a stack. Define the `Application\Generic\Stack` class as discussed in this recipe. Next, define a calling program that sets up autoloading and creates an instance of the `stack`:

```
<?php
// setup class autoloading
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
use Application\Generic\Stack;
$stack = new Stack();
```

To do something with the stack, store a series of messages. As you would most likely store messages at different points in your application, you can use `sleep()` to simulate other code running:

```
echo 'Do Something ... ' . PHP_EOL;
$stack->push('1st Message: ' . date('H:i:s'));
sleep(3);

echo 'Do Something Else ... ' . PHP_EOL;
```

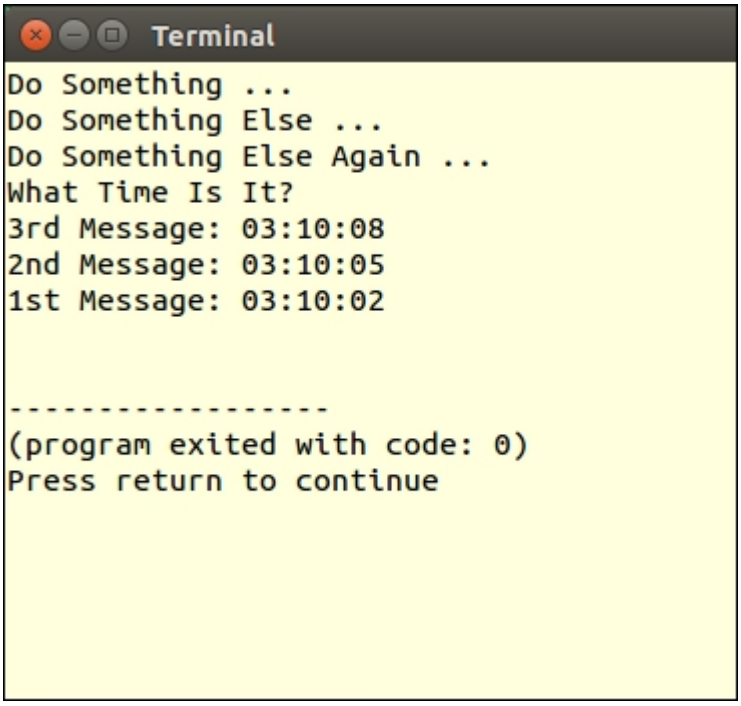
```
$stack->push('2nd Message: ' . date('H:i:s'));
sleep(3);

echo 'Do Something Else Again ... ' . PHP_EOL;
$stack->push('3rd Message: ' . date('H:i:s'));
sleep(3);
```

Finally, simply iterate through the stack to retrieve messages. Note that you can call the stack object as if it were a function, which returns the `SplStack` instance:

```
echo 'What Time Is It?' . PHP_EOL;
foreach ($stack() as $item) {
  echo $item . PHP_EOL;
}
```

Here is the expected output:



Building a binary search class

Conventional searches often proceed through the list of items in a sequential manner. This means that the maximum possible number of items to be searched could be the same as the length of the list! This is not very efficient. If you need to expedite a search, consider implementing a *binary* search.

The technique is quite simple: you find the midpoint in the list, and determine whether the search item is less than, equal to, or greater than the midpoint item. If less, you set the upper limit to the midpoint, and search only the first half of the list. If greater, set the lower limit to the midpoint, and search only the last half of the list. You would then proceed to divide the list into 1/4, 1/8, 1/16, and so on, until the search item is found (or not).

> ## Note
>
> It's important to note that although the maximum number of comparisons is considerably smaller than a sequential search (*log n + 1* where *n* is the number of elements in the list, and *log* is the binary logarithm), the list involved in the search must first be sorted, which of course downgrades performance.

How to do it...

1. We first construct a search class, `Application\Generic\Search`, which accepts the primary list as an argument. As a control, we also define a property, `$iterations`:

```
namespace Application\Generic;
class Search
{
  protected $primary;
  protected $iterations;
```

```
  public function __construct($primary)
  {
    $this->primary = $primary;
  }
```

2. Next we define a method, `binarySearch()`, which sets up the search infrastructure. The first order of business is to build a separate array, `$search`, where the key is a composite of the columns included in the search. We then sort by key:

```
public function binarySearch(array $keys,
$item)
{
  $search = array();
  foreach ($this->primary as $primaryKey =>
$data) {
    $searchKey = function ($keys, $data) {
      $key = '';
      foreach ($keys as $k) $key .= $data[$k];

      return $key;
    };
    $search[$searchKey($keys, $data)] =
$primaryKey;
  }
  ksort($search);
```

3. We then pull out the keys into another array, `$binary`, so that we can perform the binary sort based on numeric keys. We then call `doBinarySearch()`, which results in a key from our intermediary array `$search`, or a Boolean, `FALSE`:

```
  $binary = array_keys($search);
  $result = $this->doBinarySearch($binary,
$item);
  return $this->primary[$search[$result]] ??
FALSE;
}
```

4. The first `doBinarySearch()` initializes a series of parameters. `$iterations`, `$found`, `$loop`, `$done`, and `$max` are all used to prevent an endless loop. `$upper` and `$lower` represent the slice of the list to be examined:

```
public function doBinarySearch($binary, $item)
{
  $iterations = 0;
  $found = FALSE;
  $loop  = TRUE;
  $done  = -1;
  $max   = count($binary);
  $lower = 0;
  $upper = $max - 1;
```

5. We then implement a `while()` loop and set the midpoint:

```
  while ($loop && !$found) {
    $mid = (int) (($upper - $lower) / 2) +
$lower;
```

6. We now get to use the new PHP 7 **spaceship operator**, which gives us, in a single comparison, less than, equal to, or greater than. If less, we set the upper limit to the midpoint. If greater, the lower limit is adjusted to the midpoint. If equal, we're done and home free:

```
switch ($item <=> $binary[$mid]) {
  // $item < $binary[$mid]
  case -1 :
  $upper = $mid;
  break;
  // $item == $binary[$mid]
  case 0 :
  $found = $binary[$mid];
  break;
  // $item > $binary[$mid]
  case 1 :
  default :
  $lower = $mid;
}
```

7. Now for a bit of loop control. We increment the number of iterations and make sure it does not exceed the size of the list. If so, something is definitely wrong and we need to bail out. Otherwise, we check to see whether the upper and lower limits are the same more than twice in a row, in which case the search item has not been found. Then we store the number of iterations and return whatever was found (or not):

```
            $loop = (($iterations++ < $max) && ($done
        < 1));
            $done += ($upper == $lower) ? 1 : 0;
          }
          $this->iterations = $iterations;
          return $found;
        }
```

How it works...

First, implement the `Application\Generic\Search` class defining the methods described in this recipe. Next, define a calling program, `chap_10_binary_search.php`, which sets up autoloading and reads the `customer.csv` file as a search target (as discussed in the previous recipe):

```
<?php
define('CUSTOMER_FILE', __DIR__ .
'/../data/files/customer.csv');
include __DIR__ .
'/chap_10_linked_list_include.php';
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
use Application\Generic\Search;
$headers = array();
$customer = readCsv(CUSTOMER_FILE, $headers);
```

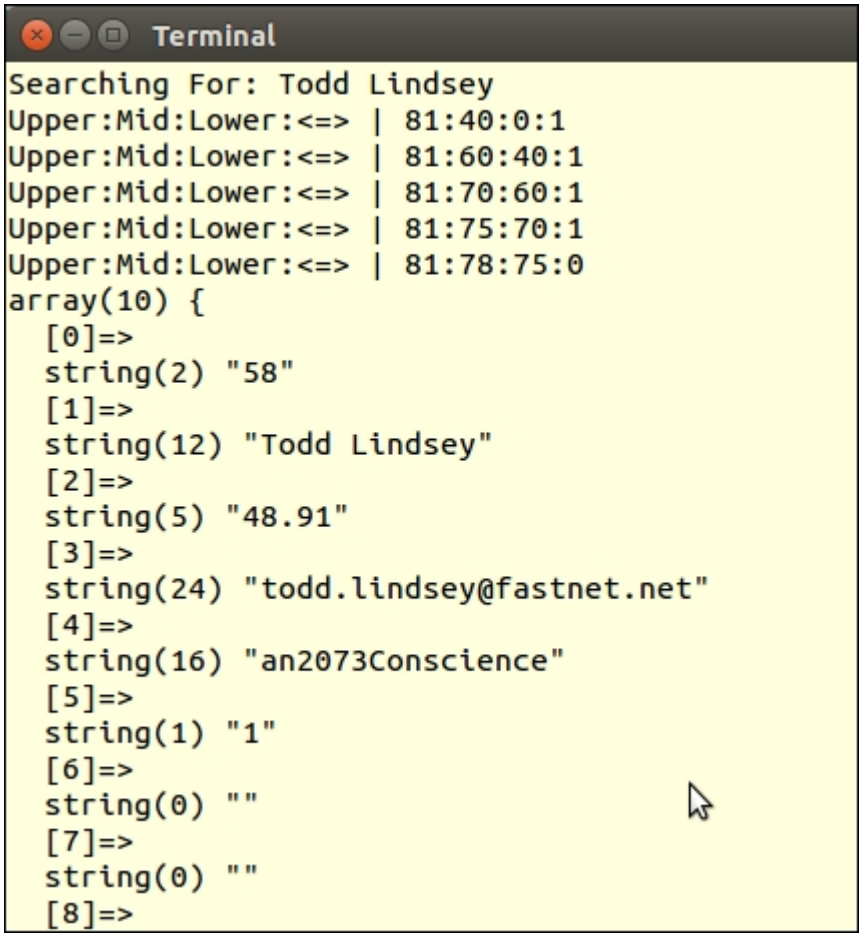
You can then create a new `Search` instance, and specify an item somewhere in the middle of the list. In this illustration, the search is based on column 1, customer name, and the item is `Todd Lindsey`:

```
$search = new Search($customer);
$item = 'Todd Lindsey';
$cols = [1];
echo "Searching For: $item\n";
var_dump($search->binarySearch($cols, $item));
```

For illustration, add this line just before `switch()` in `Application\Generic\Search::doBinarySearch()`:

```
echo 'Upper:Mid:Lower:<=> | ' . $upper . ':' .
$mid . ':' . $lower . ':' . ($item <=>
$binary[$mid]);
```

The output is shown here. Notice how the upper, middle, and lower limits adjust until the item is found:



See also

For more information on binary search, there is an excellent article on Wikipedia that goes through the basic math at https://en.wikipedia.org/wiki/Binary_search_algorithm.

Implementing a search engine

In order to implement a search engine, we need to make provision for multiple columns to be included in the search. In addition, it's important to recognize that the search item might be found in the middle of the field, and that very rarely will users provide enough information for an exact match. Accordingly, we will rely heavily on the SQL `LIKE %value%` clause.

How to do it...

1. First, we define a basic class to hold search criteria. The object contains three properties: the key, which ultimately represents a database column; the operator (`LIKE`, `<`, `>`, and so on); and optionally an item. The reason why an item is optional is that some operators, such as `IS NOT NULL`, do not require specific data:

```
namespace Application\Database\Search;
class Criteria
{
  public $key;
  public $item;
  public $operator;
  public function __construct($key, $operator,
$item = NULL)
  {
    $this->key  = $key;
    $this->operator = $operator;
    $this->item = $item;
  }
}
```

2. Next we need to define a class, `Application\Database\Search\Engine`, and provide the necessary class constants and properties. The difference between `$columns` and `$mapping` is that `$columns` holds information that will ultimately appear in an HTML `SELECT` field (or the equivalent). For security reasons, we do not want to

expose the actual names of the database columns, thus the need for another array `$mapping`:

```
namespace Application\Database\Search;
use PDO;
use Application\Database\Connection;
class Engine
{
  const ERROR_PREPARE = 'ERROR: unable to
prepare statement';
  const ERROR_EXECUTE = 'ERROR: unable to
execute statement';
  const ERROR_COLUMN  = 'ERROR: column name
not on list';
  const ERROR_OPERATOR= 'ERROR: operator not
on list';
  const ERROR_INVALID = 'ERROR: invalid search
criteria';

  protected $connection;
  protected $table;
  protected $columns;
  protected $mapping;
  protected $statement;
  protected $sql = '';
```

3. Next, we define a set of operators we are willing to support. The key represents actual SQL. The value is what will appear in the form:

```
  protected $operators = [
      'LIKE'     => 'Equals',
      '<'        => 'Less Than',
      '>'        => 'Greater Than',
      '<>'       => 'Not Equals',
      'NOT NULL' => 'Exists',
  ];
```

4. The constructor accepts a database connection instance as an argument. For our purposes, we will use `Application\Database\Connection`, defined in Chapter 5, *Interacting with a Database*. We also need to provide the name of the database table, as well as `$columns`, an array of arbitrary column keys and labels, which will appear in the HTML form.

This will reference `$mapping`, where the key matches `$columns`, but where the value represents actual database column names:

```
public function __construct(Connection
$connection,
                            $table, array
$columns, array $mapping)
{
  $this->connection  = $connection;
  $this->setTable($table);
  $this->setColumns($columns);
  $this->setMapping($mapping);
}
```

5. After the constructor, we provide a series of useful getters and setters:

```
public function setColumns($columns)
{
  $this->columns = $columns;
}
public function getColumns()
{
  return $this->columns;
}
// etc.
```

6. Probably the most critical method is the one that builds the SQL statement to be prepared. After the initial `SELECT` setup, we add a `WHERE` clause, using `$mapping` to add the actual database column name. We then add the operator and implement `switch()` which, based on the operator, may or may not add a named placeholder that will represent the search item:

```
public function prepareStatement(Criteria
$criteria)
{
  $this->sql = 'SELECT * FROM ' . $this->table
. ' WHERE ';
  $this->sql .= $this->mapping[$criteria->key]
. ' ';
  switch ($criteria->operator) {
    case 'NOT NULL' :
      $this->sql .= ' IS NOT NULL OR ';
      break;
```

```
        default :
          $this->sql .= $criteria->operator . ' :'
          . $this->mapping[$criteria->key] . ' OR
';
        }
```

7. Now that the core `SELECT` has been defined, we remove any trailing `OR` keywords, and add a clause that causes the result to be sorted according to the search column. The statement is then sent to the database to be prepared:

```
      $this->sql = substr($this->sql, 0, -4)
        . ' ORDER BY ' . $this->mapping[$criteria-
>key];
      $statement = $this->connection->pdo-
>prepare($this->sql);
      return $statement;
}
```

8. We are now ready to move on to the main show, the `search()` method. We accept an `Application\Database\Search\Criteria` object as an argument. This ensures that we have an item key and operator at a minimum. To be on the safe side, we add an `if()` statement to check these properties:

```
public function search(Criteria $criteria)
{
  if (empty($criteria->key) ||
empty($criteria->operator)) {
    yield ['error' => self::ERROR_INVALID];
    return FALSE;
  }
```

9. We then call `prepareStatement()` using `try` / `catch` to trap errors:

```
try {
    if (!$statement = $this-
>prepareStatement($criteria)) {
      yield ['error' => self::ERROR_PREPARE];
      return FALSE;
}
```

10. Next we build an array of parameters that will be supplied to `execute()`. The key represents the database column name that

was used as a placeholder in the prepared statement. Note that instead of using `=`, we use the `LIKE %value% construct`:

```
$params = array();
switch ($criteria->operator) {
  case 'NOT NULL' :
    // do nothing: already in statement
    break;
    case 'LIKE' :
    $params[$this->mapping[$criteria->key]] =
    '%' . $criteria->item . '%';
    break;
    default :
    $params[$this->mapping[$criteria->key]] =
    $criteria->item;
}
```

11. The statement is executed, and the results returned using the `yield` keywords, which effectively turns this method into a generator:

```
    $statement->execute($params);
    while ($row = $statement-
>fetch(PDO::FETCH_ASSOC)) {
      yield $row;
    }
  } catch (Throwable $e) {
    error_log(__METHOD__ . ':' . $e-
>getMessage());
    throw new Exception(self::ERROR_EXECUTE);
  }
  return TRUE;
}
```

How it works...

Place the code discussed in this recipe in the files `Criteria.php` and `Engine.php` under `Application\Database\Search`. You can then define a calling script, `chap_10_search_engine.php`, which sets up autoloading. You can take advantage of the `Application\Database\Connection` class discussed in Chapter 5, *Interacting with a Database*, and the form element classes covered in Chapter 6, *Building Scalable Websites*:

```
<?php
define('DB_CONFIG_FILE',
'/../config/db.config.php');
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');

use Application\Database\Connection;
use Application\Database\Search\ { Engine,
Criteria };
use Application\Form\Generic;
use Application\Form\Element\Select;
```

You can now define which database columns will appear in the form, and a matching mapping file:

```
$dbCols = [
  'cname' => 'Customer Name',
  'cbal' => 'Account Balance',
  'cmail' => 'Email Address',
  'clevel' => 'Level'
];

$mapping = [
  'cname' => 'name',
  'cbal' => 'balance',
  'cmail' => 'email',
  'clevel' => 'level'
];
```

You can now set up the database connection and create the search engine instance:

```
$conn = new Connection(include __DIR__ .
DB_CONFIG_FILE);
$engine = new Engine($conn, 'customer', $dbCols,
$mapping);
```

In order to display the appropriate drop-down `SELECT` elements, we define wrappers and elements based on `Application\Form\*` classes:

```
$wrappers = [
  Generic::INPUT => ['type' => 'td', 'class' =>
'content'],
  Generic::LABEL => ['type' => 'th', 'class' =>
'label'],
  Generic::ERRORS => ['type' => 'td', 'class' =>
'error']
];

// define elements
$fieldElement = new Select('field',
                Generic::TYPE_SELECT,
                'Field',
                $wrappers,
                ['id' => 'field']);
                $opsElement = new Select('ops',
                Generic::TYPE_SELECT,
                'Operators',
                $wrappers,
                ['id' => 'ops']);
                $itemElement = new Generic('item',
                Generic::TYPE_TEXT,
                'Searching For ...',
                $wrappers,
                ['id' => 'item','title' => 'If
more than one item, separate with commas']);
                $submitElement = new
Generic('submit',
                Generic::TYPE_SUBMIT,
                'Search',
                $wrappers,
                ['id' => 'submit','title' =>
'Click to Search', 'value' => 'Search']);
```

We then get input parameters (if defined), set form element options, create search criteria, and run the search:

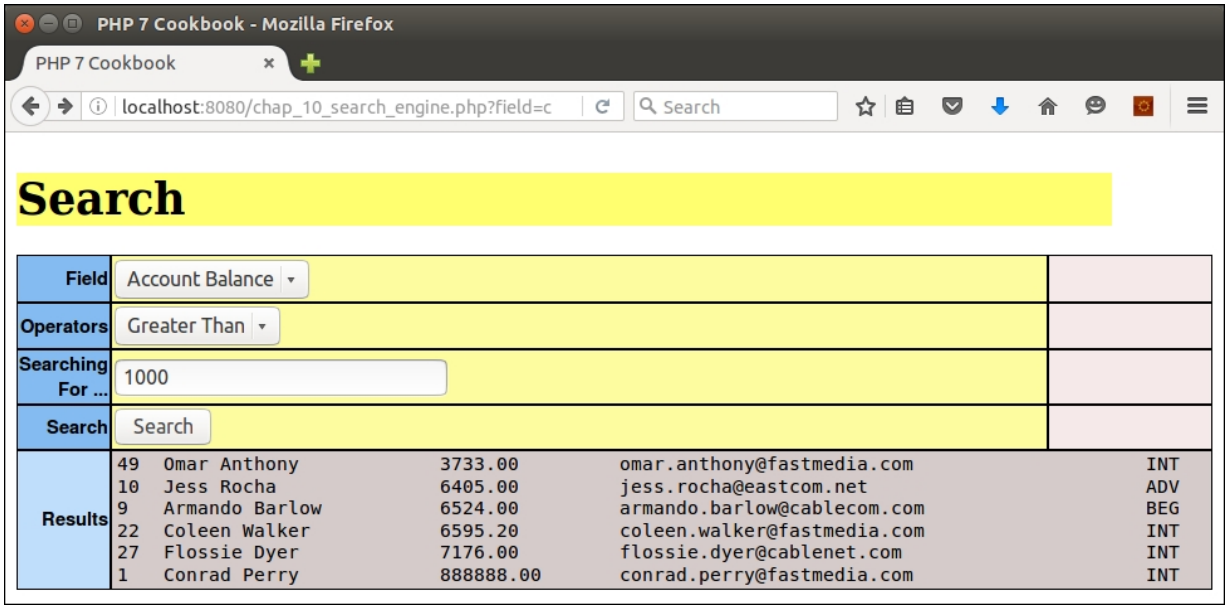
```
$key  = (isset($_GET['field']))
? strip_tags($_GET['field']) : NULL;
$op   = (isset($_GET['ops'])) ? $_GET['ops'] :
NULL;
$item = (isset($_GET['item'])) ?
strip_tags($_GET['item']) : NULL;
$fieldElement->setOptions($dbCols, $key);
$itemElement->setSingleAttribute('value', $item);
```

```
$opsElement->setOptions($engine->getOperators(),
$op);
$criteria = new Criteria($key, $op, $item);
$results = $engine->search($criteria);
?>
```

The display logic mainly orients towards rendering the form. A more thorough presentation is discussed in Chapter 6, *Building Scalable Websites*, but we show the core logic here:

```
  <form name="search" method="get">
  <table class="display" cellspacing="0"
width="100%">
    <tr><?= $fieldElement->render(); ?></tr>
    <tr><?= $opsElement->render(); ?></tr>
    <tr><?= $itemElement->render(); ?></tr>
    <tr><?= $submitElement->render(); ?></tr>
    <tr>
    <th class="label">Results</th>
      <td class="content" colspan=2>
      <span style="font-size: 10pt;font-
family:monospace;">
      <table>
      <?php foreach ($results as $row) : ?>
        <tr>
          <td><?= $row['id'] ?></td>
          <td><?= $row['name'] ?></td>
          <td><?= $row['balance'] ?></td>
          <td><?= $row['email'] ?></td>
          <td><?= $row['level'] ?></td>
        </tr>
      <?php endforeach; ?>
      </table>
      </span>
      </td>
    </tr>
  </table>
  </form>
```

Here is sample output from a browser:


PHP 7 Cookbook - Mozilla Firefox
PHP 7 Cookbook
localhost:8080/chap_10_search_engine.php?field=c
Search
Search
Field
Account Balance
Operators
Greater Than
Searching For ...
1000
Search
Results
49 Omar Anthony 3733.00 omar.anthony@fastmedia.com INT
10 Jess Rocha 6405.00 jess.rocha@eastcom.net ADV
9 Armando Barlow 6524.00 armando.barlow@cablecom.com BEG
22 Coleen Walker 6595.20 coleen.walker@fastmedia.com INT
27 Flossie Dyer 7176.00 flossie.dyer@cablenet.com INT
1 Conrad Perry 888888.00 conrad.perry@fastmedia.com INT

Displaying a multi-dimensional array and accumulating totals

How to properly display data from a multi-dimensional array has been a classic problem for any web developer. For illustration, assume you wish to display a list of customers and their purchases. For each customer, you wish to show their name, phone number, account balance, and so on. This already represents a two dimensional array where the *x* axis represents customers and the *y* axis represents data for that customer. Now add in purchases and you have a third axis! How can you represent a 3D model on a 2D screen? One possible solution would be to incorporate "hidden" division tags with a simple JavaScript visibility toggle.

How to do it...

1. First we need to generate a 3D array from a SQL statement that uses a number of `JOIN` clauses. We will use the `Application/Database/Connection` class introduced in Chapter 1, *Building a Foundation,* to formulate an appropriate SQL query. We leave two parameters open, `min` and `max`, in order to support pagination. Unfortunately, we cannot use a simple `LIMIT` and `OFFSET` in this case, as the number of rows will vary depending on the number of purchases for any given customer. Accordingly, we can restrict the number of rows by placing restrictions on the customer ID that presumably (hopefully) is incremental. To make this work properly, we also need to set the primary `ORDER` to customer ID:

```
define('ITEMS_PER_PAGE', 6);
define('SUBROWS_PER_PAGE', 6);
define('DB_CONFIG_FILE',
'/../config/db.config.php');
include __DIR__ .
'/../Application/Database/Connection.php';
use Application\Database\Connection;
```

```
$conn = new Connection(include __DIR__ .
DB_CONFIG_FILE);
$sql  = 'SELECT
c.id,c.name,c.balance,c.email,f.phone, '
  .
'u.transaction,u.date,u.quantity,u.sale_price,
r.title '
  . 'FROM customer AS c '
  . 'JOIN profile AS f '
  . 'ON f.id = c.id '
  . 'JOIN purchases AS u '
  . 'ON u.customer_id = c.id '
  . 'JOIN products AS r '
  . 'ON u.product_id = r.id '
  . 'WHERE c.id >= :min AND c.id < :max '
  . 'ORDER BY c.id ASC, u.date DESC ';
```

2. Next we can implement a form of pagination, based on restrictions on the customer ID, using simple `$_GET` parameters. Note that we add an extra check to make sure the value of `$prev` does not go below zero. You might consider adding another control that ensures the value of `$next` does not go beyond the last customer ID. In this illustration, we just allow it to increment:

```
$page = $_GET['page'] ?? 1;
$page = (int) $page;
$next = $page + 1;
$prev = $page - 1;
$prev = ($prev >= 0) ? $prev : 0;
```

3. We then calculate the values for `$min` and `$max`, and prepare and execute the SQL statement:

```
$min  = $prev * ITEMS_PER_PAGE;
$max  = $page * ITEMS_PER_PAGE;
$stmt = $conn->pdo->prepare($sql);
$stmt->execute(['min' => $min, 'max' =>
$max]);
```

4. A `while()` loop can be used to fetch results. We use a simple fetch mode of `PDO::FETCH_ASSOC` for the purpose of this example. Using the customer ID as a key, we store basic customer information as array parameters. We then store an array of purchase information in a sub-array, `$results[$key]`

`['purchases'][]`. When the customer ID changes, it's a signal to store the same information for the next customer. Note that we accumulate totals per customer in an array key total:

```
$custId = 0;
$result = array();
$grandTotal = 0.0;
while ($row = $stmt->fetch(PDO::FETCH_ASSOC))
{
  if ($row['id'] != $custId) {
    $custId = $row['id'];
    $result[$custId] = [
      'name'    => $row['name'],
      'balance' => $row['balance'],
      'email'   => $row['email'],
      'phone'   => $row['phone'],
    ];
    $result[$custId]['total'] = 0;
  }
  $result[$custId]['purchases'][] = [
    'transaction' => $row['transaction'],
    'date'        => $row['date'],
    'quantity'    => $row['quantity'],
    'sale_price'  => $row['sale_price'],
    'title'       => $row['title'],
  ];
  $result[$custId]['total'] +=
$row['sale_price'];
  $grandTotal += $row['sale_price'];
}
?>
```

5. Next we implement the view logic. First, we start with a block that displays primary customer information:

```
<div class="container">
<?php foreach ($result as $key => $data) : ?>
<div class="mainLeft color0">
    <?= $data['name'] ?> [<?= $key ?>]
</div>
<div class="mainRight">
  <div class="row">
    <div class="left">Balance</div>
          <div class="right"><?=
$data['balance']; ?></div>
  </div>
```

```
  <div class="row">
    <div class="left color2">Email</div>
           <div class="right"><?=
$data['email']; ?></div>
  </div>
  <div class="row">
    <div class="left">Phone</div>
           <div class="right"><?=
$data['phone']; ?></div>
    </div>
  <div class="row">
        <div class="left color2">Total
Purchases</div>
    <div class="right">
<?= number_format($data['total'],2); ?>
</div>
  </div>
```

6. Next comes the logic to display a list of purchases for this customer:

```
<!-- Purchases Info -->
<table>
  <tr>
  <th>Transaction</th><th>Date</th>
<th>Qty</th>
   <th>Price</th><th>Product</th>
  </tr>
  <?php $count  = 0; ?>
  <?php foreach ($data['purchases'] as
$purchase) : ?>
  <?php $class = ($count++ & 01) ? 'color1' :
'color2'; ?>
  <tr>
  <td class="<?= $class ?>"><?=
$purchase['transaction'] ?></td>
  <td class="<?= $class ?>"><?=
$purchase['date'] ?></td>
  <td class="<?= $class ?>"><?=
$purchase['quantity'] ?></td>
  <td class="<?= $class ?>"><?=
$purchase['sale_price'] ?></td>
  <td class="<?= $class ?>"><?=
$purchase['title'] ?></td>
  </tr>
  <?php endforeach; ?>
</table>
```

7. For the purposes of pagination, we then add buttons to represent *previous* and *next*:

```
<?php endforeach; ?>
<div class="container">
  <a href="?page=<?= $prev ?>">
        <input type="button" value="Previous">
</a>
  <a href="?page=<?= $next ?>">
        <input type="button" value="Next"
class="buttonRight"></a>
</div>
<div class="clearRow"></div>
</div>
```

8. The result so far, unfortunately, is nowhere near neat and tidy! Accordingly we add a simple JavaScript function to toggle the visibility of a `<div>` tag based on its `id` attribute:

```
<script type="text/javascript">
function showOrHide(id) {
  var div = document.getElementById(id);
  div.style.display = div.style.display ==
"none" ? "block" : "none";
}
</script>
```

9. Next we wrap the purchases table inside an initially invisible `<div>` tag. Then, we can place a limit of how many sub-rows are initially visible, and add a link that *reveals* the remaining purchase data:

```
<div class="row" id="<?= 'purchase' . $key ?>"
style="display:none;">
  <table>
    <tr>
      <th>Transaction</th><th>Date</th>
<th>Qty</th>
                  <th>Price</th>
<th>Product</th>
    </tr>
  <?php $count  = 0; ?>
  <?php $first  = TRUE; ?>
  <?php foreach ($data['purchases'] as
$purchase) : ?>
    <?php if ($count > SUBROWS_PER_PAGE &&
```

```
$first) : ?>
    <?php     $first = FALSE; ?>
    <?php     $subId = 'subrow' . $key; ?>
    </table>
    <a href="#" onClick="showOrHide('<?=
$subId ?>')">More</a>
    <div id="<?= $subId ?>"
style="display:none;">
    <table>
    <?php endif; ?>
  <?php $class = ($count++ & 01) ? 'color1' :
'color2'; ?>
  <tr>
  <td class="<?= $class ?>"><?=
$purchase['transaction'] ?></td>
  <td class="<?= $class ?>"><?=
$purchase['date'] ?></td>
  <td class="<?= $class ?>"><?=
$purchase['quantity'] ?></td>
  <td class="<?= $class ?>"><?=
$purchase['sale_price'] ?></td>
  <td class="<?= $class ?>"><?=
$purchase['title'] ?></td>
  </tr>
  <?php endforeach; ?>
  </table>
  <?php if (!$first) : ?></div><?php endif; ?>
</div>
```

10. We then add a button that, when clicked, reveals the hidden `<div>` tag:

```
<input type="button" value="Purchases"
class="buttonRight"
    onClick="showOrHide('<?= 'purchase' . $key
?>')">
```

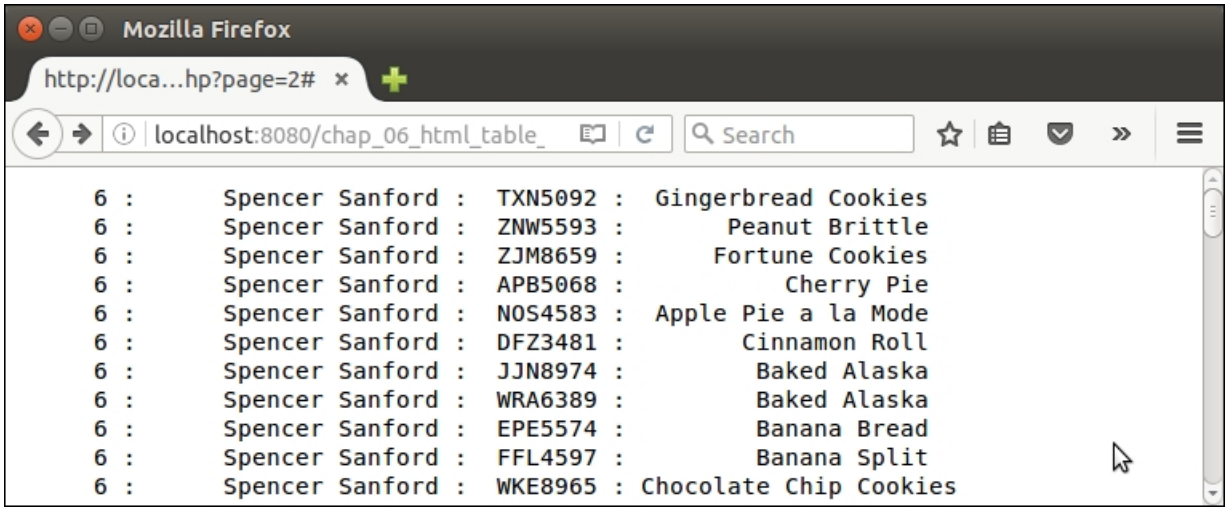
How it works...

Place the code described in steps 1 to 5 into a file, `chap_10_html_table_multi_array_hidden.php`.

Just inside the `while()` loop, add the following:

```
printf('%6s : %20s : %8s : %20s' . PHP_EOL,
    $row['id'], $row['name'], $row['transaction'],
$row['title']);
```

Just after the `while()` loop, add an `exit` command. Here is the output:



You will notice that the basic customer information, such as the ID and name, repeats for each result row, but purchase information, such as transaction and product title, varies. Go ahead and remove the `printf()` statement.

Replace the `exit` command with the following:

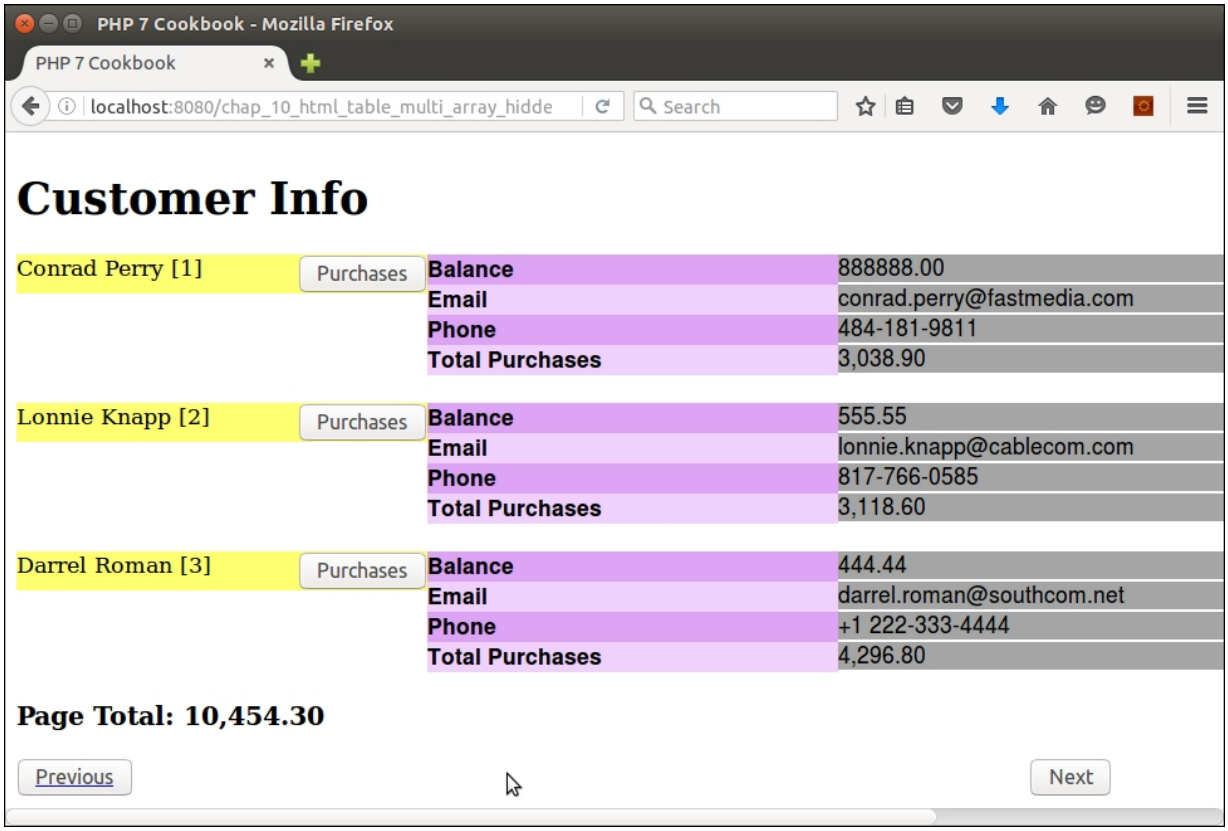
```
echo '<pre>', var_dump($result), '</pre>'; exit;
```

Here is how the newly composed 3D array looks:



```
array(6) {
  [6]=>
  array(5) {
    ["name"]=>
    string(15) "Spencer Sanford"
    ["balance"]=>
    string(5) "99.99"
    ["email"]=>
    string(28) "spencer.sanford@cablenet.net"
    ["phone"]=>
    string(12) "451-815-7386"
    ["purchases"]=>
    array(92) {
      [0]=>
      array(5) {
        ["transaction"]=>
        string(7) "TXN5092"
        ["date"]=>
        string(19) "2016-09-12 05:46:16"
        ["quantity"]=>
        string(2) "44"
        ["sale_price"]=>
        string(5) "10.50"
        ["title"]=>
        string(19) "Gingerbread Cookies"
      }
      [1]=>
      array(5) {
        ["transaction"]=>
        string(7) "ZNW5593"
        ["date"]=>
        string(19) "2015-09-18 03:58:26"
```

You can now add the display logic shown in steps 5 to 7. As mentioned, although you are now showing all data, the visual display is not helpful. Now go ahead and add the refinements mentioned in the remaining steps. Here is how the initial output might appear:



When the **Purchases** button is clicked, initial purchase info appears. If the link to **More** is clicked, the remaining purchase information shows:

Balance	888888.00
Email	conrad.perry@fastmedia.com
Phone	484-181-9811
Total Purchases	3,038.90

Transaction	Date	Qty	Price	Product
OJM3659	2016-10-27 03:27:28	2	8.40	Peanut Butter Cups
RLI3437	2016-08-10 14:16:22	17	10.20	Neapolitan Ice Cream
EOV1192	2016-05-15 08:56:28	3	17.10	Chocolate Eclair
ZAM6290	2015-10-21 23:11:22	18	72.00	Ice Cream Cake
ZNW5593	2015-09-18 03:58:26	15	75.00	Pumpkin Ice Cream

More

Chapter 11. Implementing Software Design Patterns

In this chapter, we will cover the following topics:

- Creating an array to object hydrator
- Building an object to array hydrator
- Implementing a strategy pattern
- Defining a mapper
- Implementing object-relational mapping
- Implementing the Pub/Sub design pattern

Introduction

The idea of incorporating **software design patterns** into **object-oriented programming** (**OOP**) code was first discussed in a seminal work entitled *Design Patterns: Elements of Reusable Object-Oriented Software*, authored by the famous Gang of Four (E. Gamma, R. Helm, R. Johnson, and J. Vlissides) in 1994. Defining neither standards nor protocols, this work identified common generic software designs that have proven useful over the years. The patterns discussed in this book are generally thought to fall into three categories: creational, structural, and behavioral.

Examples of many of these patterns have already been presented in this book. Here is a brief summary:

Design pattern	**Chapter**	**Recipe**
Singleton	2	Defining visibility

Design pattern	Chapter	Recipe
Factory	6	Implementing a form factory
Adapter	8	Handling translation without `gettext()`
Proxy	7	Creating a simple REST client Creating a simple SOAP client
Iterator	2 3	Recursive directory iterator Using iterators

In this chapter, we will examine a number of additional design patterns, focusing primarily on Concurrency and Architectural patterns.

Creating an array to object hydrator

The **Hydrator** pattern is a variation of the **Data Transfer Object** design pattern. Its design principle is quite simple: moving data from one place to another. In this illustration, we will define classes to move data from an array to an object.

How to do it...

1. First, we define a `Hydrator` class that is able to use getters and setters. For this illustration we will use `Application\Generic\Hydrator\GetSet`:

```
namespace Application\Generic\Hydrator;
class GetSet
{
  // code
}
```

2. Next, we define a `hydrate()` method, which takes both an array and an object as arguments. It then calls the `setXXX()` methods on the object to populate it with values from the array. We use `get_class()` to determine the object's class, and then `get_class_methods()` to get a list of all methods. `preg_match()` is used to match the method prefix and its suffix, which is subsequently assumed to be the array key:

```
public static function hydrate(array $array,
$object)
{
  $class = get_class($object);
  $methodList = get_class_methods($class);
  foreach ($methodList as $method) {
    preg_match('/^(set)(.*?)$/i', $method,
$matches);
    $prefix = $matches[1] ?? '';
    $key    = $matches[2] ?? '';
    $key    = strtolower(substr($key, 0, 1)) .
```

```
        substr($key, 1);
            if ($prefix == 'set' &&
        !empty($array[$key])) {
                $object->$method($array[$key]);
            }
          }
          return $object;
        }
```

How it works...

To demonstrate how the array to hydrator object is used, first define the `Application\Generic\Hydrator\GetSet` class as described in the *How to do it...* section. Next, define an entity class that can be used to test the concept. For the purposes of this illustration, create a `Application\Entity\Person` class, with the appropriate properties and methods. Be sure to define getters and setters for all properties. Not all such methods are shown here:

```
namespace Application\Entity;
class Person
{
  protected $firstName  = '';
  protected $lastName   = '';
  protected $address    = '';
  protected $city       = '';
  protected $stateProv  = '';
  protected $postalCode = '';
  protected $country    = '';

  public function getFirstName()
  {
    return $this->firstName;
  }

  public function setFirstName($firstName)
  {
    $this->firstName = $firstName;
  }

  // etc.
}
```

You can now create a calling program called `chap_11_array_to_object.php`, which sets up autoloading, and uses the appropriate classes:

```
<?php
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
use Application\Entity\Person;
use Application\Generic\Hydrator\GetSet;
```

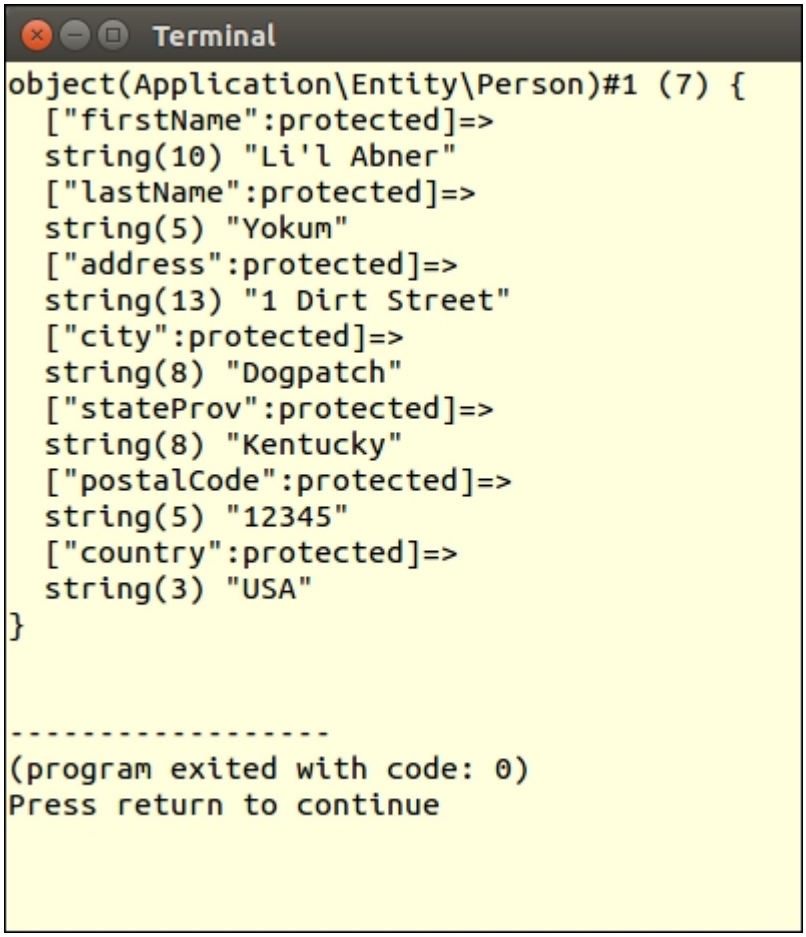
Next, you can define a test array with values that will be added to a new `Person` instance:

```
$a['firstName'] = 'Li\'l Abner';
$a['lastName']  = 'Yokum';
$a['address']   = '1 Dirt Street';
$a['city']      = 'Dogpatch';
$a['stateProv'] = 'Kentucky';
$a['postalCode']= '12345';
$a['country']   = 'USA';
```

You can now call `hydrate()` and `extract()` in a static manner:

```
$b = GetSet::hydrate($a, new Person());
var_dump($b);
```

The results are shown in the following screenshot:


Terminal
object(Application\Entity\Person)#1 (7) {
["firstName":protected]=>
string(10) "Li'l Abner"
["lastName":protected]=>
string(5) "Yokum"
["address":protected]=>
string(13) "1 Dirt Street"
["city":protected]=>
string(8) "Dogpatch"
["stateProv":protected]=>
string(8) "Kentucky"
["postalCode":protected]=>
string(5) "12345"
["country":protected]=>
string(3) "USA"
}

(program exited with code: 0)
Press return to continue

Building an object to array hydrator

This recipe is the converse of the *Creating an array to object hydrator* recipe. In this case, we need to pull values from object properties and return an associative array where the key will be the column name.

How to do it...

1. For this illustration we will build upon the `Application\Generic\Hydrator\GetSet` class defined in the previous recipe:

```
namespace Application\Generic\Hydrator;
class GetSet
{
  // code
}
```

2. After the `hydrate()` method defined in the previous recipe, we define an `extract()` method, which takes an object as an argument. The logic is similar to that used with `hydrate()`, except this time we're searching for `getXXX()` methods. Again, `preg_match()` is used to match the method prefix and its suffix, which is subsequently assumed to be the array key:

```
public static function extract($object)
{
  $array = array();
  $class = get_class($object);
  $methodList = get_class_methods($class);
  foreach ($methodList as $method) {
    preg_match('/^(get)(.*?)$/i', $method,
$matches);
    $prefix = $matches[1] ?? '';
    $key    = $matches[2] ?? '';
    $key    = strtolower(substr($key, 0, 1)) .
substr($key, 1);
```

```
            if ($prefix == 'get') {
              $array[$key] = $object->$method();
            }
          }
          return $array;
        }
        }
```

> **Note**
>
> Note that we have defined `hydrate()` and `extract()` as static methods for convenience.

How it works...

Define a calling program called `chap_11_object_to_array.php`, which sets up autoloading, and uses the appropriate classes:

```
<?php
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
use Application\Entity\Person;
use Application\Generic\Hydrator\GetSet;
```

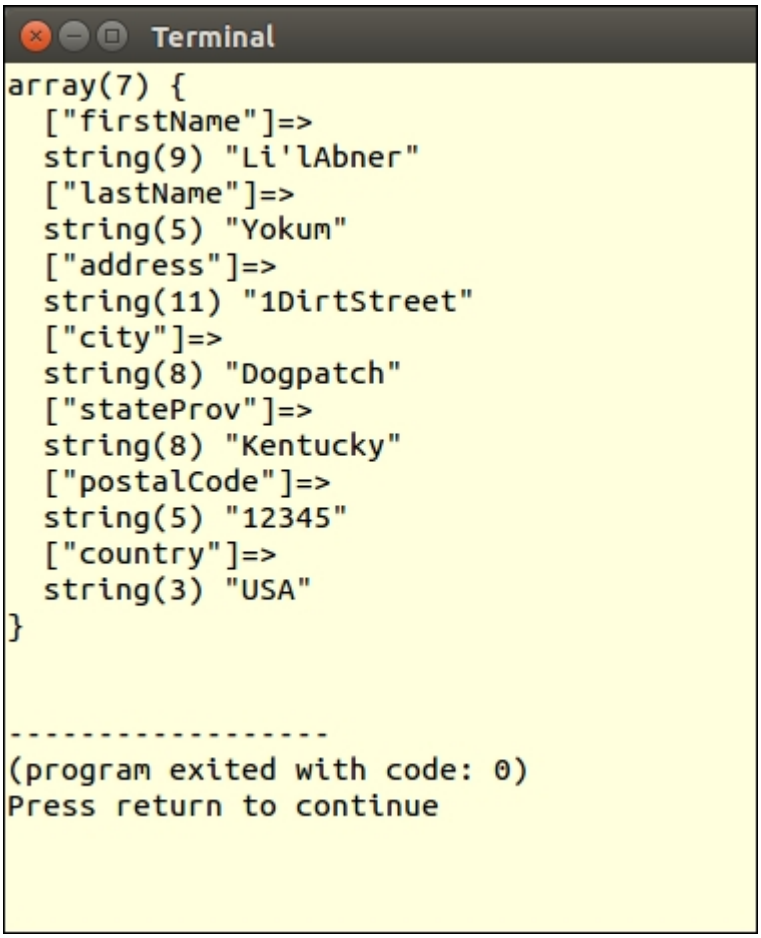
Next, define an instance of `Person`, setting values for its properties:

```
$obj = new Person();
$obj->setFirstName('Li\'lAbner');
$obj->setLastName('Yokum');
$obj->setAddress('1DirtStreet');
$obj->setCity('Dogpatch');
$obj->setStateProv('Kentucky');
$obj->setPostalCode('12345');
$obj->setCountry('USA');
```

Finally, call the new `extract()` method in a static manner:

```
$a = GetSet::extract($obj);
var_dump($a);
```

The output is shown in the following screenshot:



Implementing a strategy pattern

It is often the case that runtime conditions force the developer to define several ways of doing the same thing. Traditionally, this involved a massive `if/elseif/else` block of commands. You would then either have to define large blocks of logic inside the `if` statement, or create a series of functions or methods to enable the different approaches. The strategy pattern attempts to formalize this process by having the primary class encapsulate a series of sub-classes that represent different approaches to solve the same problem.

How to do it...

1. In this illustration, we will use the `GetSet` hydrator class defined previously as a strategy. We will define a primary `Application\Generic\Hydrator\Any` class, which will then consume strategy classes in the `Application\Generic\Hydrator\Strategy` namespace, including `GetSet`, `PublicProps`, and `Extending`.
2. We first define class constants that reflect the built-in strategies that are available:

```
namespace Application\Generic\Hydrator;
use InvalidArgumentException;
use Application\Generic\Hydrator\Strategy\ {
GetSet, PublicProps, Extending };
class Any
{
  const STRATEGY_PUBLIC  = 'PublicProps';
  const STRATEGY_GET_SET = 'GetSet';
  const STRATEGY_EXTEND  = 'Extending';
  protected $strategies;
  public $chosen;
```

3. We then define a constructor that adds all built-in strategies to the `$strategies` property:

```
public function __construct()
{
  $this->strategies[self::STRATEGY_GET_SET] =
new GetSet();
  $this->strategies[self::STRATEGY_PUBLIC] =
new PublicProps();
  $this->strategies[self::STRATEGY_EXTEND] =
new Extending();
}
```

4. We also add an `addStrategy()` method that allows us to overwrite or add new strategies without having to recode the class:

```
public function addStrategy($key,
HydratorInterface $strategy)
{
  $this->strategies[$key] = $strategy;
}
```

5. The `hydrate()` and `extract()` methods simply call those of the chosen strategy:

```
public function hydrate(array $array, $object)
{
  $strategy = $this->chooseStrategy($object);
  $this->chosen = get_class($strategy);
  return $strategy::hydrate($array, $object);
}

public function extract($object)
{
  $strategy = $this->chooseStrategy($object);
  $this->chosen = get_class($strategy);
  return $strategy::extract($object);
}
```

6. The tricky bit is figuring out which hydration strategy to choose. For this purpose we define `chooseStrategy()`, which takes an object as an argument. We first perform some detective work by way of getting a list of class methods. We then scan through the list to see if we have any `getXXX()` or `setXXX()` methods. If so, we choose the `GetSet` hydrator as our chosen strategy:

```
public function chooseStrategy($object)
{
  $strategy = NULL;
  $methodList =
get_class_methods(get_class($object));
  if (!empty($methodList) &&
is_array($methodList)) {
      $getSet = FALSE;
      foreach ($methodList as $method) {
        if (preg_match('/^get|set.*$/i',
$method)) {
            $strategy = $this-
>strategies[self::STRATEGY_GET_SET];
      break;
    }
  }
}
```

7. Still within our `chooseStrategy()` method, if there are no getters or setters, we next use `get_class_vars()` to determine if there are any available properties. If so, we choose `PublicProps` as our hydrator:

```
if (!$strategy) {
    $vars =
get_class_vars(get_class($object));
    if (!empty($vars) && count($vars)) {
        $strategy = $this-
>strategies[self::STRATEGY_PUBLIC];
    }
}
```

8. If all else fails, we fall back to the `Extending` hydrator, which returns a new class that simply extends the object class, thus making any `public` or `protected` properties available:

```
if (!$strategy) {
    $strategy = $this-
>strategies[self::STRATEGY_EXTEND];
}
return $strategy;
}
}
```

9. Now we turn our attention to the strategies themselves. First, we define a new `Application\Generic\Hydrator\Strategy` namespace.
10. In the new namespace, we define an interface that allows us to identify any strategies that can be consumed by `Application\Generic\Hydrator\Any`:

```
namespace
Application\Generic\Hydrator\Strategy;
interface HydratorInterface
{
  public static function hydrate(array $array,
$object);
  public static function extract($object);
}
```

11. The `GetSet` hydrator is exactly as defined in the previous two recipes, with the only addition being that it will implement the new interface:

```
namespace
Application\Generic\Hydrator\Strategy;
class GetSet implements HydratorInterface
{

  public static function hydrate(array $array,
$object)
  {
    // defined in the recipe:
    // "Creating an Array to Object Hydrator"
  }

  public static function extract($object)
  {
    // defined in the recipe:
    // "Building an Object to Array Hydrator"
  }
}
```

12. The next hydrator simply reads and writes public properties:

```
namespace
Application\Generic\Hydrator\Strategy;
class PublicProps implements HydratorInterface
{
```

```
  public static function hydrate(array $array,
$object)
  {
    $propertyList= array_keys(
      get_class_vars(get_class($object)));
    foreach ($propertyList as $property) {
      $object->$property = $array[$property]
?? NULL;
    }
    return $object;
  }

  public static function extract($object)
  {
    $array = array();
    $propertyList = array_keys(
      get_class_vars(get_class($object)));
    foreach ($propertyList as $property) {
      $array[$property] = $object->$property;
    }
    return $array;
  }
}
```

13. Finally, `Extending`, the Swiss Army knife of hydrators, extends the object class, thus providing direct access to properties. We further define magic getters and setters to provide access to properties.
14. The `hydrate()` method is the most difficult as we are assuming no getters or setters are defined, nor are the properties defined with a visibility level of `public`. Accordingly, we need to define a class that extends the class of the object to be hydrated. We do this by first defining a string that will be used as a template to build the new class:

```
namespace
Application\Generic\Hydrator\Strategy;
class Extending implements HydratorInterface
{
  const UNDEFINED_PREFIX = 'undefined';
  const TEMP_PREFIX = 'TEMP_';
  const ERROR_EVAL = 'ERROR: unable to
evaluate object';
  public static function hydrate(array $array,
$object)
```

```
        {
          $className = get_class($object);
          $components = explode('\\', $className);
          $realClass  = array_pop($components);
          $nameSpace  = implode('\\', $components);
          $tempClass = $realClass .
    self::TEMP_SUFFIX;
          $template = 'namespace '
            . $nameSpace . '{'
            . 'class ' . $tempClass
            . ' extends ' . $realClass . ' '
```

15. Continuing in the `hydrate()` method, we define a `$values` property, and a constructor that assigns the array to be hydrated into the object as an argument. We loop through the array of values, assigning values to properties. We also define a useful `getArrayCopy()` method, which returns these values if needed, as well as a magic `__get()` method to simulate direct property access:

```
. '{ '
. '  protected $values; '
. '  public function __construct($array) '
. '  { $this->values = $array; '
. '    foreach ($array as $key => $value) '
. '       $this->$key = $value; '
. '  } '
. '  public function getArrayCopy() '
. '  { return $this->values; } '
```

16. For convenience we define a magic `__get()` method, which simulates direct variable access as if they were public:

```
. '  public function __get($key) '
. '  { return $this->values[$key] ?? NULL; } '
```

17. Still in the template for the new class, we define also a magic `__call()` method, which simulates getters and setters:

```
. '  public function __call($method, $params)
'
. '  { '
. '    preg_match("/^(get|set)(.*?)$/i", '
. '        $method, $matches); '
. '    $prefix = $matches[1] ?? ""; '
```

```
. '    $key    = $matches[2] ?? ""; '
. '    $key    = strtolower(substr($key, 0,
1)) '
. '              substr($key, 1); '
. '    if ($prefix == "get") { '
. '        return $this->values[$key] ?? NULL;
'
. '     } else { '
. '        $this->values[$key] = $params[0]; '
. '     } '
. '  } '
. '} '
. '} // ends namespace ' . PHP_EOL
```

18. Finally, still in the template for the new class, we add a function, in the global namespace, that builds and returns the class instance:

```
. 'namespace { '
. 'function build($array) '
. '{ return new ' . $nameSpace . '\\'
.    $tempClass . '($array); } '
. '} // ends global namespace '
. PHP_EOL;
```

19. Still in the `hydrate()` method, we execute the completed template using `eval()`. We then run the `build()` method defined just at the end of the template. Note that as we are unsure of the namespace of the class to be populated, we define and call `build()` from the global namespace:

```
try {
    eval($template);
} catch (ParseError $e) {
    error_log(__METHOD__ . ':' . $e-
>getMessage());
    throw new Exception(self::ERROR_EVAL);
}
return \build($array);
}
```

20. The `extract()` method is much easier to define as our choices are extremely limited. Extending a class and populating it from an array using magic methods is easily accomplished. The reverse is not the case. If we were to extend the class, we would

lose all the property values, as we are extending the class, not the object instance. Accordingly, our only option is to use a combination of getters and public properties:

```
public static function extract($object)
{
  $array = array();
  $class = get_class($object);
  $methodList = get_class_methods($class);
  foreach ($methodList as $method) {
    preg_match('/^(get)(.*?)$/i', $method,
$matches);
    $prefix = $matches[1] ?? '';
    $key    = $matches[2] ?? '';
    $key    = strtolower(substr($key, 0, 1))
    . substr($key, 1);
    if ($prefix == 'get') {
        $array[$key] = $object->$method();
    }
  }
  $propertyList=
array_keys(get_class_vars($class));
  foreach ($propertyList as $property) {
    $array[$property] = $object->$property;
  }
  return $array;
  }
}
```

How it works...

You can begin by defining three test classes with identical properties: `firstName`, `lastName`, and so on. The first, `Person`, should have protected properties along with getters and setters. The second, `PublicPerson`, will have public properties. The third, `ProtectedPerson`, has protected properties but no getters nor setters:

```
<?php
namespace Application\Entity;
class Person
{
  protected $firstName  = '';
  protected $lastName   = '';
```

```
  protected $address    = '';
  protected $city       = '';
  protected $stateProv  = '';
  protected $postalCode = '';
  protected $country    = '';

    public function getFirstName()
    {
      return $this->firstName;
    }

    public function setFirstName($firstName)
    {
      $this->firstName = $firstName;
    }

  // be sure to define remaining getters and
setters

}

<?php
namespace Application\Entity;
class PublicPerson
{
  private $id = NULL;
  public $firstName  = '';
  public $lastName   = '';
  public $address    = '';
  public $city       = '';
  public $stateProv  = '';
  public $postalCode = '';
  public $country    = '';
}

<?php
namespace Application\Entity;

class ProtectedPerson
{
  private $id = NULL;
  protected $firstName  = '';
  protected $lastName   = '';
  protected $address    = '';
  protected $city       = '';
  protected $stateProv  = '';
  protected $postalCode = '';
```

```
    protected $country    = '';
}
```

You can now define a calling program called `chap_11_strategy_pattern.php`, which sets up autoloading and uses the appropriate classes:

```
<?php
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
use Application\Entity\ { Person, PublicPerson,
ProtectedPerson };
use Application\Generic\Hydrator\Any;
use Application\Generic\Hydrator\Strategy\ {
GetSet, Extending, PublicProps };
```

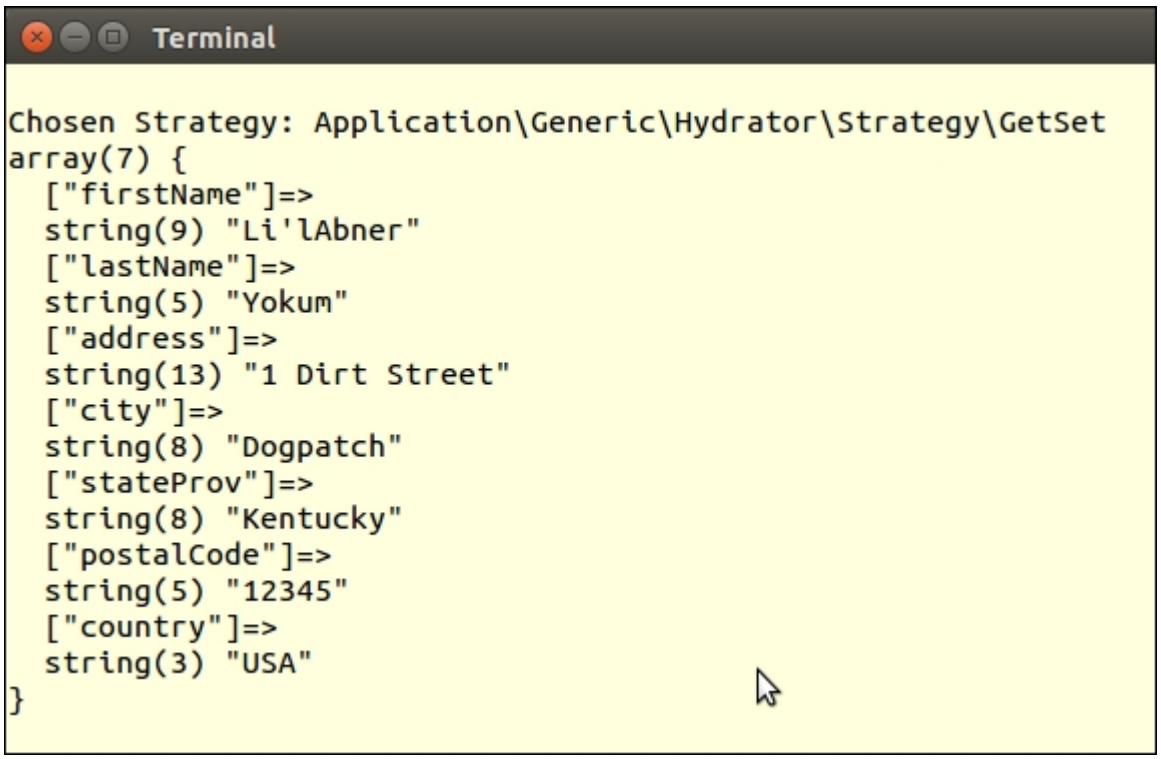
Next, create an instance of `Person` and run the setters to define values for properties:

```
$obj = new Person();
$obj->setFirstName('Li\'lAbner');
$obj->setLastName('Yokum');
$obj->setAddress('1 Dirt Street');
$obj->setCity('Dogpatch');
$obj->setStateProv('Kentucky');
$obj->setPostalCode('12345');
$obj->setCountry('USA');
```

Next, create an instance of the `Any` hydrator, call `extract()`, and use `var_dump()` to view the results:

```
$hydrator = new Any();
$b = $hydrator->extract($obj);
echo "\nChosen Strategy: " . $hydrator->chosen .
"\n";
var_dump($b);
```

Observe, in the following output, that the `GetSet` strategy was chosen:



Note

Note that the `id` property is not set as its visibility level is `private`.

Next, you can define an array with the same values. Call `hydrate()` on the `Any` instance, and supply a new `PublicPerson` instance as an argument:

```
$a = [
  'firstName'  => 'Li\'lAbner',
  'lastName'   => 'Yokum',
  'address'    => '1 Dirt Street',
  'city'       => 'Dogpatch',
  'stateProv'  => 'Kentucky',
  'postalCode' => '12345',
  'country'    => 'USA'
];

$p = $hydrator->hydrate($a, new PublicPerson());
echo "\nChosen Strategy: " . $hydrator->chosen .
```

```
"\n";
var_dump($p);
```

Here is the result. Note that the `PublicProps` strategy was chosen in this case:



Finally, call `hydrate()` again, but this time supply an instance of `ProtectedPerson` as the object argument. We then call `getFirstName()` and `getLastName()` to test the magic getters. We also access first and last names as direct variable access:

```
$q = $hydrator->hydrate($a, new
ProtectedPerson());
echo "\nChosen Strategy: " . $hydrator->chosen .
"\n";
echo "Name: {$q->getFirstName()} {$q-
>getLastName()}\n";
echo "Name: {$q->firstName} {$q->lastName}\n";
var_dump($q);
```

Here is the last output, showing that the `Extending` strategy was chosen. You'll also note that the instance is a new `ProtectedPerson_TEMP` class, and that the protected properties are fully populated:



Defining a mapper

A **mapper** or **data mapper** works in much the same manner as a hydrator: converting data from one model, be it array or object, into another. A critical difference is that the hydrator is generic and does not need to have object property names pre-programmed, whereas the mapper is the opposite: it needs precise information on property names for both models. In this recipe we will demonstrate the use of a mapper to convert data from one database table into another.

How to do it...

1. We first define a `Application\Database\Mapper\FieldConfig` class, which contains mapping instructions for individual fields. We also define appropriate class constants:

```
namespace Application\Database\Mapper;
use InvalidArgumentException;
class FieldConfig
{
  const ERROR_SOURCE =
    'ERROR: need to specify destTable and/or
source';
  const ERROR_DEST   = 'ERROR: need to specify
either '
    . 'both destTable and destCol or neither';
```

2. Key properties are defined along with the appropriate class constants. `$key` is used to identify the object. `$source` represents the column from the source database table. `$destTable` and `$destCol` represent the target database table and column. `$default`, if defined, contains a default value or a callback that produces the appropriate value:

```
public $key;
public $source;
public $destTable;
public $destCol;
public $default;
```

3. We now turn our attention to the constructor, which assigns default values, builds the key, and checks to see that either or both `$source` or `$destTable` and `$destCol` are defined:

```
public function __construct($source = NULL,
                            $destTable = NULL,
                            $destCol   = NULL,
                            $default   = NULL)
{
  // generate key from source + destTable +
destCol
  $this->key = $source . '.' . $destTable .
'.' . $destCol;
  $this->source = $source;
  $this->destTable = $destTable;
  $this->destCol = $destCol;
  $this->default = $default;
  if (($destTable && !$destCol) ||
      (!$destTable && $destCol)) {
      throw new
InvalidArgumentException(self::ERROR_DEST);
  }
  if (!$destTable && !$source) {
      throw new InvalidArgumentException(
        self::ERROR_SOURCE);
  }
}
```

> ## Note
>
> Note that we allow source and destination columns to be `NULL`. The reason for this is that we might have a source column that has no place in the destination table. Likewise, there might be mandatory columns in the destination table that are not represented in the source table.

4. In the case of defaults, we need to check to see if the value is a callback. If so, we run the callback; otherwise, we return the direct value. Note that the callbacks should be defined so that they accept a database table row as an argument:

```
public function getDefault()
{
  if (is_callable($this->default)) {
      return call_user_func($this->default,
$row);
  } else {
      return $this->default;
  }
}
```

5. Finally, to wrap up this class, we define getters and setters for each of the five properties:

```
public function getKey()
{
  return $this->key;
}

public function setKey($key)
{
  $this->key = $key;
}

// etc.
```

6. Next, we define a `Application\Database\Mapper\Mapping` mapping class, which accepts the name of the source and destination tables as well as an array of `FieldConfig` objects as an argument. You will see later that we allow the destination table property to be an array, as the mapping might be to two or more destination tables:

```
namespace Application\Database\Mapper;
class Mapping
{
  protected $sourceTable;
  protected $destTable;
  protected $fields;
  protected $sourceCols;
  protected $destCols;

  public function __construct(
    $sourceTable, $destTable, $fields = NULL)
  {
    $this->sourceTable = $sourceTable;
```

```
    $this->destTable = $destTable;
    $this->fields = $fields;
  }
```

7. We then define getters and setters for these properties:

```
public function getSourceTable()
{
  return $this->sourceTable;
}
public function setSourceTable($sourceTable)
{
  $this->sourceTable = $sourceTable;
}
// etc.
```

8. For field configuration, we also need to provide the ability to add an individual field. There is no need to supply the key as a separate argument as this can be obtained from the `FieldConfig` instance:

```
public function addField(FieldConfig $field)
{
  $this->fields[$field->getKey()] = $field;
  return $this;
}
```

9. It is extremely important to obtain an array of source column names. The problem is that the source column name is a property buried in a `FieldConfig` object. Accordingly, when this method is called, we loop through the array of `FieldConfig` objects and invoke `getSource()` on each one to obtain the source column name:

```
public function getSourceColumns()
{
  if (!$this->sourceCols) {
      $this->sourceCols = array();
      foreach ($this->getFields() as $field) {
        if (!empty($field->getSource())) {
            $this->sourceCols[$field-
>getKey()] =
                $field->getSource();
        }
      }
```

```
    }
    return $this->sourceCols;
  }
```

10. We use a similar approach for `getDestColumns()`. The big difference compared to getting a list of source columns is that we only want the columns for one specific destination table, which is critical if there's more than one such table is defined. We do not need to check to see if `$destCol` is set as this is already taken care of in the constructor for `FieldConfig`:

```
  public function getDestColumns($table)
  {
    if (empty($this->destCols[$table])) {
        foreach ($this->getFields() as $field) {
          if ($field->getDestTable()) {
            if ($field->getDestTable() ==
$table) {
                $this->destCols[$table][$field-
>getKey()] =
                  $field->getDestCol();
            }
          }
        }
    }
    return $this->destCols[$table];
  }
```

11. Finally, we define a method that accepts as a first argument an array representing one row of data from the source table. The second argument is the name of the destination table. The method produces an array of data ready to be inserted into the destination table.
12. We had to make a decision as to which would take precedence: the default value (which could be provided by a callback), or data from the source table. We decided to test for a default value first. If the default comes back `NULL`, data from the source is used. Note that if further processing is required, the default should be defined as a callback.

```
  public function mapData($sourceData,
$destTable)
  {
```

```
    $dest = array();
    foreach ($this->fields as $field) {
      if ($field->getDestTable() == $destTable)
{
          $dest[$field->getDestCol()] = NULL;
          $default = $field-
>getDefault($sourceData);
          if ($default) {
              $dest[$field->getDestCol()] =
$default;
          } else {
              $dest[$field->getDestCol()] =
                    $sourceData[$field-
>getSource()];
          }
      }
    }
    return $dest;
}
}
```

Note

Note that some columns will appear in the destination insert that are not present in the source row. In this case, the `$source` property of the `FieldConfig` object is left as `NULL`, and a default value is supplied, either as a scalar value or as a callback.

13. We are now ready to define two methods that will generate SQL. The first such method will generate an SQL statement to read from the source table. The statement will include placeholders to be prepared (for example, using `PDO::prepare()`):

```
public function getSourceSelect($where = NULL)
{
  $sql = 'SELECT '
  . implode(',', $this->getSourceColumns()) .
' ';
```

```
    $sql .= 'FROM ' . $this->getSourceTable() .
' ';
    if ($where) {
      $where = trim($where);
      if (stripos($where, 'WHERE') !== FALSE) {
          $sql .= $where;
      } else {
          $sql .= 'WHERE ' . $where;
      }
    }
    return trim($sql);
  }
```

14. The other SQL generation method produces a statement to be prepared for a specific destination table. Notice that the placeholders are the same as the column names preceded by ":":

```
  public function getDestInsert($table)
  {
    $sql = 'INSERT INTO ' . $table . ' ';
    $sql .= '( '
    . implode(',', $this-
>getDestColumns($table))
    . ' ) ';
    $sql .= ' VALUES ';
    $sql .= '( :'
    . implode(',:', $this-
>getDestColumns($table))
    . ' ) ';
    return trim($sql);
  }
```

How it works...

Use the code shown in steps 1 to 5 to produce an `Application\Database\Mapper\FieldConfig` class. Place the code shown in steps 6 to 14 into a second `Application\Database\Mapper\Mapping` class.

Before defining a calling program that performs mapping, it's important to consider the source and destination database tables. The definition for the source table, `prospects_11`, is as follows:

```
CREATE TABLE `prospects_11` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `first_name` varchar(128) NOT NULL,
  `last_name` varchar(128) NOT NULL,
  `address` varchar(256) DEFAULT NULL,
  `city` varchar(64) DEFAULT NULL,
  `state_province` varchar(32) DEFAULT NULL,
  `postal_code` char(16) NOT NULL,
  `phone` varchar(16) NOT NULL,
  `country` char(2) NOT NULL,
  `email` varchar(250) NOT NULL,
  `status` char(8) DEFAULT NULL,
  `budget` decimal(10,2) DEFAULT NULL,
  `last_updated` datetime DEFAULT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `UNIQ_35730C06E7927C74` (`email`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

In this example, you can use two destination tables, `customer_11` and `profile_11`, between which there is a 1:1 relationship:

```
CREATE TABLE `customer_11` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(256) CHARACTER SET latin1
    COLLATE latin1_general_cs NOT NULL,
  `balance` decimal(10,2) NOT NULL,
  `email` varchar(250) NOT NULL,
  `password` char(16) NOT NULL,
  `status` int(10) unsigned NOT NULL DEFAULT '0',
  `security_question` varchar(250) DEFAULT NULL,
  `confirm_code` varchar(32) DEFAULT NULL,
  `profile_id` int(11) DEFAULT NULL,
  `level` char(3) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `UNIQ_81398E09E7927C74` (`email`)
) ENGINE=InnoDB AUTO_INCREMENT=80 DEFAULT
CHARSET=utf8 COMMENT='Customers';

CREATE TABLE `profile_11` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `address` varchar(256) NOT NULL,
  `city` varchar(64) NOT NULL,
  `state_province` varchar(32) NOT NULL,
  `postal_code` varchar(10) NOT NULL,
  `country` varchar(3) NOT NULL,
```

```
  `phone` varchar(16) NOT NULL,
  `photo` varchar(128) NOT NULL,
  `dob` datetime NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=80 DEFAULT
CHARSET=utf8 COMMENT='Customers';
```

You can now define a calling program called `chap_11_mapper.php`, which sets up autoloading and uses the two classes mentioned previously. You can also use the `Connection` class defined in Chapter 5, *Interacting with a Database*:

```
<?php
define('DB_CONFIG_FILE',
'/../config/db.config.php');
define('DEFAULT_PHOTO', 'person.gif');
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
use Application\Database\Mapper\ { FieldConfig,
Mapping };
use Application\Database\Connection;
$conn = new Connection(include __DIR__ .
DB_CONFIG_FILE);
```

For demonstration purposes, after having made sure the two destination tables exist, you can truncate both tables so that any data that appears is clean:

```
$conn->pdo->query('DELETE FROM customer_11');
$conn->pdo->query('DELETE FROM profile_11');
```

You are now ready to build the `Mapping` instance and populate it with `FieldConfig` objects. Each `FieldConfig` object represents a mapping between source and destination. In the constructor, supply the name of the source table and the two destination tables in the form of an array:

```
$mapper = new Mapping('prospects_11',
['customer_11','profile_11']);
```

You can start simply by mapping fields between `prospects_11` and `customer_11` where there are no defaults:

```
$mapper>addField(new
FieldConfig('email','customer_11','email'))
```

Note that `addField()` returns the current mapping instance so there is no need to keep specifying `$mapper->addField()`. This technique is referred to as the **fluent interface**.

The name field is tricky, as in the `prospects_11` table it's represented by two columns, but only one column in the `customer_11` table. Accordingly, you can add a callback as default for `first_name` to combine the two fields into one. You will also need to define an entry for `last_name` but where there is no destination mapping:

```
->addField(new
FieldConfig('first_name','customer_11','name',
  function ($row) { return
trim(($row['first_name'] ?? '')
. ' ' .  ($row['last_name'] ?? ''));}))
->addField(new FieldConfig('last_name'))
```

The `customer_11::status` field can use the null coalesce operator (`??`) to determine if it's set or not:

```
->addField(new
FieldConfig('status','customer_11','status',
  function ($row) { return $row['status'] ??
'Unknown'; }))
```

The `customer_11::level` field is not represented in the source table, thus you can make a `NULL` entry for the source field, but make sure the destination table and column are set. Likewise, `customer_11::password` is not present in the source table. In this case, the callback uses the phone number as a temporary password:

```
->addField(new
FieldConfig(NULL,'customer_11','level','BEG'))
->addField(new
FieldConfig(NULL,'customer_11','password',
  function ($row) { return $row['phone']; }))
```

You can also set mappings from `prospects_11` to `profile_11` as follows. Note that as the source photo and date of birth columns are not present in `prospects_11`, you can set any appropriate default:

```
->addField(new
FieldConfig('address','profile_11','address'))
->addField(new
FieldConfig('city','profile_11','city'))
->addField(new
FieldConfig('state_province','profile_11',
'state_province', function ($row) {
  return $row['state_province'] ?? 'Unknown'; }))
->addField(new
FieldConfig('postal_code','profile_11',
'postal_code'))
->addField(new
FieldConfig('phone','profile_11','phone'))
->addField(new
FieldConfig('country','profile_11','country'))
->addField(new
FieldConfig(NULL,'profile_11','photo',
DEFAULT_PHOTO))
->addField(new
FieldConfig(NULL,'profile_11','dob',
date('Y-m-d')));
```

In order to establish the 1:1 relationship between the `profile_11` and `customer_11` tables, we set the values of `customer_11::id`, `customer_11::profile_id` and `profile_11::id` to the value of `$row['id']` using a callback:

```
$idCallback = function ($row) { return $row['id'];
};
$mapper->addField(new
FieldConfig('id','customer_11','id',
$idCallback))
->addField(new
```

```
FieldConfig(NULL,'customer_11','profile_id',
$idCallback))
->addField(new
FieldConfig('id','profile_11','id',$idCallback));
```

You can now call the appropriate methods to generate three SQL statements, one to read from the source table, and two to insert into the two destination tables:

```
$sourceSelect  = $mapper->getSourceSelect();
$custInsert    = $mapper-
>getDestInsert('customer_11');
$profileInsert = $mapper-
>getDestInsert('profile_11');
```

These three statements can immediately be prepared for later execution:

```
$sourceStmt  = $conn->pdo->prepare($sourceSelect);
$custStmt    = $conn->pdo->prepare($custInsert);
$profileStmt = $conn->pdo-
>prepare($profileInsert);
```

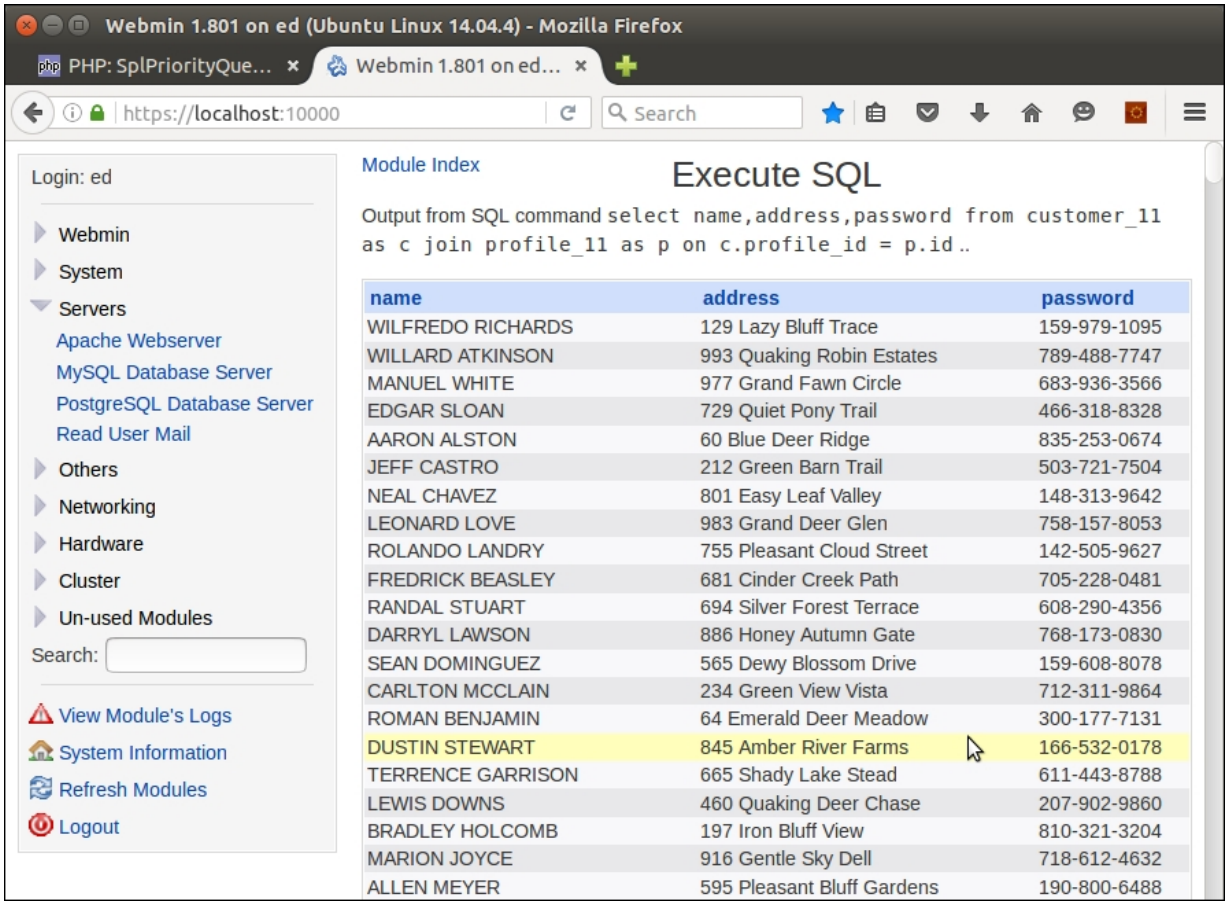
We then execute the `SELECT` statement, which produces rows from the source table. In a loop we then generate `INSERT` data for each destination table, and execute the appropriate prepared statements:

```
$sourceStmt->execute();
while ($row = $sourceStmt-
>fetch(PDO::FETCH_ASSOC)) {
  $custData = $mapper->mapData($row,
'customer_11');
  $custStmt->execute($custData);
  $profileData = $mapper->mapData($row,
'profile_11');
  $profileStmt->execute($profileData);
  echo "Processing: {$custData['name']}\n";
}
```

Here are the three SQL statements produced:



We can then view the data directly from the database using SQL `JOIN` to ensure the relationship has been maintained:


Webmin 1.801 on ed (Ubuntu Linux 14.04.4) - Mozilla Firefox
PHP: SplPriorityQue...
Webmin 1.801 on ed...
https://localhost:10000
Search
Login: ed
Webmin
System
Servers
Apache Webserver
MySQL Database Server
PostgreSQL Database Server
Read User Mail
Others
Networking
Hardware
Cluster
Un-used Modules
Search:
View Module's Logs
System Information
Refresh Modules
Logout
Module Index
Execute SQL
Output from SQL command select name,address,password from customer_11 as c join profile_11 as p on c.profile_id = p.id ..
name address password
WILFREDO RICHARDS 129 Lazy Bluff Trace 159-979-1095
WILLARD ATKINSON 993 Quaking Robin Estates 789-488-7747
MANUEL WHITE 977 Grand Fawn Circle 683-936-3566
EDGAR SLOAN 729 Quiet Pony Trail 466-318-8328
AARON ALSTON 60 Blue Deer Ridge 835-253-0674
JEFF CASTRO 212 Green Barn Trail 503-721-7504
NEAL CHAVEZ 801 Easy Leaf Valley 148-313-9642
LEONARD LOVE 983 Grand Deer Glen 758-157-8053
ROLANDO LANDRY 755 Pleasant Cloud Street 142-505-9627
FREDRICK BEASLEY 681 Cinder Creek Path 705-228-0481
RANDAL STUART 694 Silver Forest Terrace 608-290-4356
DARRYL LAWSON 886 Honey Autumn Gate 768-173-0830
SEAN DOMINGUEZ 565 Dewy Blossom Drive 159-608-8078
CARLTON MCCLAIN 234 Green View Vista 712-311-9864
ROMAN BENJAMIN 64 Emerald Deer Meadow 300-177-7131
DUSTIN STEWART 845 Amber River Farms 166-532-0178
TERRENCE GARRISON 665 Shady Lake Stead 611-443-8788
LEWIS DOWNS 460 Quaking Deer Chase 207-902-9860
BRADLEY HOLCOMB 197 Iron Bluff View 810-321-3204
MARION JOYCE 916 Gentle Sky Dell 718-612-4632
ALLEN MEYER 595 Pleasant Bluff Gardens 190-800-6488

Implementing object-relational mapping

There are two primary techniques to achieve a relational mapping between objects. The first technique involves pre-loading the related child objects into the parent object. The advantage to this approach is that it is easy to implement, and all parent-child information is immediately available. The disadvantage is that large amounts of memory are potentially consumed, and the performance curve is skewed.

The second technique is to embed a secondary lookup into the parent object. In this latter approach, when you need to access the child objects, you would run a getter that would perform the secondary lookup. The advantage of this approach is that performance demands are spread out throughout the request cycle, and memory usage is (or can be) more easily managed. The disadvantage of this approach is that there are more queries generated, which means more work for the database server.

> **Note**
>
> Please note, however, that we will show how the use of **prepared statements** can be used to greatly offset this disadvantage.

How to do it...

Let's have a look at two techniques to implement object-relational mapping.

Technique #1 - pre-loading all child information

First, we will discuss how to implement object relational mapping by pre-loading all child information into the parent class. For this illustration, we will use three related database tables, `customer`, `purchases`, and `products`:

1. We will use the existing `Application\Entity\Customer` class (defined in Chapter 5, *Interacting with a Database*, in the *Defining entity classes to match database tables recipe*) as a model to develop an `Application\Entity\Purchase` class. As before, we will use the database definition as the basis of the entity class definition. Here is the database definition for the `purchases` table:

```
CREATE TABLE `purchases` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `transaction` varchar(8) NOT NULL,
  `date` datetime NOT NULL,
  `quantity` int(10) unsigned NOT NULL,
  `sale_price` decimal(8,2) NOT NULL,
  `customer_id` int(11) DEFAULT NULL,
  `product_id` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `IDX_C3F3` (`customer_id`),
  KEY `IDX_665A` (`product_id`),
  CONSTRAINT `FK_665A` FOREIGN KEY
(`product_id`) REFERENCES `products` (`id`),
  CONSTRAINT `FK_C3F3` FOREIGN KEY
(`customer_id`) REFERENCES `customer` (`id`)
);
```

2. Based on the customer entity class, here is how `Application\Entity\Purchase` might look. Note that not all getters and setters are shown:

```
namespace Application\Entity;

class Purchase extends Base
{
```

```
  const TABLE_NAME = 'purchases';
  protected $transaction = '';
  protected $date = NULL;
  protected $quantity = 0;
  protected $salePrice = 0.0;
  protected $customerId = 0;
  protected $productId = 0;

  protected $mapping = [
    'id'            => 'id',
    'transaction'   => 'transaction',
    'date'          => 'date',
    'quantity'      => 'quantity',
    'sale_price'    => 'salePrice',
    'customer_id'   => 'customerId',
    'product_id'    => 'productId',
  ];

  public function getTransaction() : string
  {
    return $this->transaction;
  }
  public function setTransaction($transaction)
  {
    $this->transaction = $transaction;
  }
  // NOTE: other getters / setters are not
shown here
}
```

3. We are now ready to define `Application\Entity\Product`. Here is the database definition for the `products` table:

```
CREATE TABLE `products` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `sku` varchar(16) DEFAULT NULL,
  `title` varchar(255) NOT NULL,
  `description` varchar(4096) DEFAULT NULL,
  `price` decimal(10,2) NOT NULL,
  `special` int(11) NOT NULL,
  `link` varchar(128) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `UNIQ_38C4` (`sku`)
);
```

4. Based on the customer entity class, here is how `Application\Entity\Product` might look:

```
namespace Application\Entity;

class Product extends Base
{

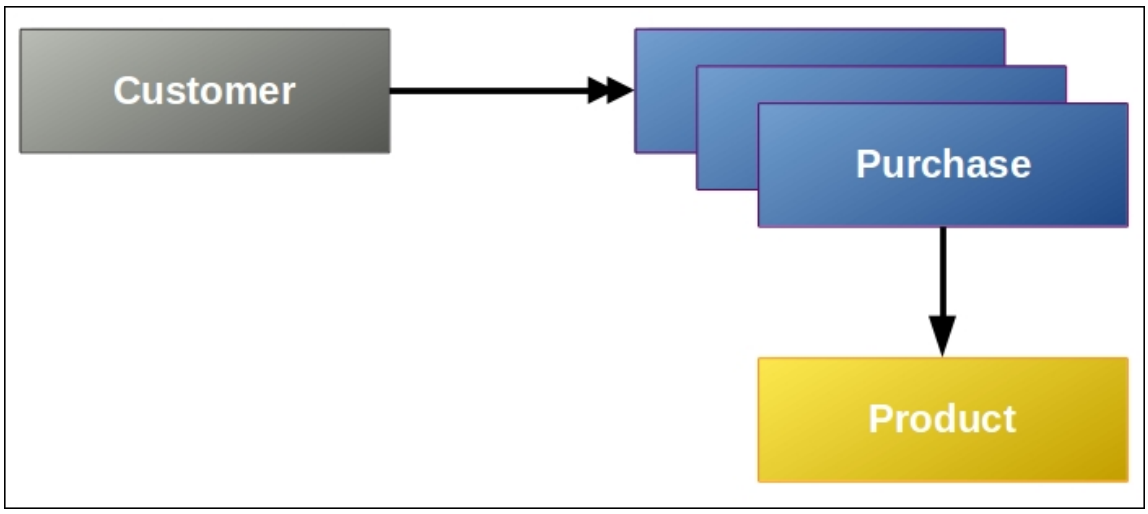
  const TABLE_NAME = 'products';
  protected $sku = '';
  protected $title = '';
  protected $description = '';
  protected $price = 0.0;
  protected $special = 0;
  protected $link = '';

  protected $mapping = [
    'id'          => 'id',
    'sku'         => 'sku',
    'title'       => 'title',
    'description' => 'description',
    'price'       => 'price',
    'special'     => 'special',
    'link'        => 'link',
  ];

  public function getSku() : string
  {
    return $this->sku;
  }
  public function setSku($sku)
  {
    $this->sku = $sku;
  }
  // NOTE: other getters / setters are not
shown here
}
```

5. Next, we need to implement a way to embed related objects. We will start with the `Application\Entity\Customer` parent class. For this section, we will assume the following relationships, illustrated in the following diagram:

    - One customer, many purchases
    - One purchase, one product



6. Accordingly, we define a getter and setter that process purchases in the form of an array of objects:

```
protected $purchases = array();
public function addPurchase($purchase)
{
  $this->purchases[] = $purchase;
}
public function getPurchases()
{
  return $this->purchases;
}
```

7. Now we turn our attention to `Application\Entity\Purchase`. In this case, there is a 1:1 relationship between a purchase and a product, so there's no need to process an array:

```
protected $product = NULL;
public function getProduct()
{
  return $this->product;
}
public function setProduct(Product $product)
{
  $this->product = $product;
}
```

Note

> Notice that in both entity classes, we do not alter the `$mapping` array. This is because implementing object relational mapping has no bearing on the mapping between entity property names and database column names.

8. Since the core functionality of obtaining basic customer information is still needed, all we need to do is to extend the `Application\Database\CustomerService` class described in Chapter 5, *Interacting with a Database*, in the *Tying entity classes to RDBMS queries* recipe. We can create a new `Application\Database\CustomerOrmService_1` class, which extends `Application\Database\CustomerService`:

```
namespace Application\Database;
use PDO;
use PDOException;
use Application\Entity\Customer;
use Application\Entity\Product;
use Application\Entity\Purchase;
class CustomerOrmService_1 extends
CustomerService
{
  // add methods here
}
```

9. We then add a method to the new service class that performs a lookup and embeds the results, in the form of `Product` and `Purchase` entities, into the core customer entity. This method performs a lookup in the form of a `JOIN`. This is possible because there is a 1:1 relationship between purchase and product. Because the `id` column has the same name in both tables, we need to add the purchase ID column as an alias. We then loop through the results, creating `Product` and `Purchase` entities. After overriding the ID, we can then embed the `Product` entity into the `Purchase` entity, and then add the `Purchase` entity to the array in the `Customer` entity:

```
protected function
fetchPurchasesForCustomer(Customer $cust)
{
```

```
    $sql = 'SELECT u.*,r.*,u.id AS purch_id '
      . 'FROM purchases AS u '
      . 'JOIN products AS r '
      . 'ON r.id = u.product_id '
      . 'WHERE u.customer_id = :id '
      . 'ORDER BY u.date';
    $stmt = $this->connection->pdo-
>prepare($sql);
    $stmt->execute(['id' => $cust->getId()]);
    while ($result = $stmt-
>fetch(PDO::FETCH_ASSOC)) {
      $product = Product::arrayToEntity($result,
new Product());
      $product->setId($result['product_id']);
      $purch = Purchase::arrayToEntity($result,
new Purchase());
      $purch->setId($result['purch_id']);
      $purch->setProduct($product);
      $cust->addPurchase($purch);
    }
    return $cust;
}
```

10. Next, we provide a wrapper for the original `fetchById()` method. This block of code needs to not only get the original `Customer` entity, but needs to look up and embed `Product and Purchase` entities. We can call the new `fetchByIdAndEmbedPurchases()` method and accept a customer ID as an argument:

```
public function
fetchByIdAndEmbedPurchases($id)
{
  return $this->fetchPurchasesForCustomer(
    $this->fetchById($id));
}
```

Technique #2 - embedding secondary lookups

Now we will cover embedding secondary lookups into the related entity classes. We will continue to use the same illustration as above, using the entity classes defined that correspond to three related database tables, `customer`, `purchases`, and `products`:

1. The mechanics of this approach are quite similar to those described in the preceding section. The main difference is that instead of doing the database lookup, and producing entity classes right away, we will embed a series of anonymous functions that will do the same thing, but called from the view logic.
2. We need to add a new method to the `Application\Entity\Customer` class that adds a single entry to the `purchases` property. Instead of an array of `Purchase` entities, we will be supplying an anonymous function:

```
public function setPurchases(Closure
$purchaseLookup)
{
  $this->purchases = $purchaseLookup;
}
```

3. Next, we will make a copy of the `Application\Database\CustomerOrmService_1` class, and call it `Application\Database\CustomerOrmService_2`:

```
namespace Application\Database;
use PDO;
use PDOException;
use Application\Entity\Customer;
use Application\Entity\Product;
use Application\Entity\Purchase;
class CustomerOrmService_2 extends
CustomerService
{
  // code
}
```

4. We then define a `fetchPurchaseById()` method, which looks up a single purchase based on its ID and produces a `Purchase` entity. Because we will ultimately be making a series of repetitive requests for single purchases in this approach, we can regain database efficiency by working off the same prepared statement, in this case, a property called `$purchPreparedStmt`:

```
public function fetchPurchaseById($purchId)
{
  if (!$this->purchPreparedStmt) {
```

```
        $sql = 'SELECT * FROM purchases WHERE id
= :id';
        $this->purchPreparedStmt =
        $this->connection->pdo->prepare($sql);
    }
    $this->purchPreparedStmt->execute(['id' =>
$purchId]);
    $result = $this->purchPreparedStmt-
>fetch(PDO::FETCH_ASSOC);
    return Purchase::arrayToEntity($result, new
Purchase());
}
```

5. After that, we need a `fetchProductById()` method that looks up a single product based on its ID and produces a `Product` entity. Given that a customer may have purchased the same product several times, we can introduce an additional level of efficiency by storing acquired product entities in a `$products` array. In addition, as with purchases, we can perform lookups on the same prepared statement:

```
public function fetchProductById($prodId)
{
    if (!isset($this->products[$prodId])) {
        if (!$this->prodPreparedStmt) {
            $sql = 'SELECT * FROM products WHERE
id = :id';
            $this->prodPreparedStmt =
            $this->connection->pdo-
>prepare($sql);
        }
        $this->prodPreparedStmt->execute(['id'
=> $prodId]);
        $result = $this->prodPreparedStmt
        ->fetch(PDO::FETCH_ASSOC);
        $this->products[$prodId] =
          Product::arrayToEntity($result, new
Product());
    }
    return $this->products[$prodId];
}
```

6. We can now rework the `fetchPurchasesForCustomer()` method to have it embed an anonymous function that makes calls to both `fetchPurchaseById()` and `fetchProductById()`, and then assigns

the resulting product entity to the newly found purchase entity. In this example, we do an initial lookup that just returns the IDs of all purchases for this customer. We then embed a sequence of anonymous functions in the `Customer::$purchases` property, storing the purchase ID as the array key, and the anonymous function as its value:

```
public function
fetchPurchasesForCustomer(Customer $cust)
{
  $sql = 'SELECT id '
    . 'FROM purchases AS u '
    . 'WHERE u.customer_id = :id '
    . 'ORDER BY u.date';
  $stmt = $this->connection->pdo-
>prepare($sql);
  $stmt->execute(['id' => $cust->getId()]);
  while ($result = $stmt-
>fetch(PDO::FETCH_ASSOC)) {
    $cust->addPurchaseLookup(
    $result['id'],
    function ($purchId, $service) {
      $purchase = $service-
>fetchPurchaseById($purchId);
      $product  = $service->fetchProductById(
                  $purchase->getProductId());
      $purchase->setProduct($product);
      return $purchase; }
    );
  }
  return $cust;
}
```

How it works...

Define the following classes based on the steps from this recipe as follows:

Class	**Technique #1 steps**

Class	Technique #1 steps
`Application\Entity\Purchase`	1 - 2, 7
`Application\Entity\Product`	3 - 4
`Application\Entity\Customer`	6, 16, + described in Chapter 5, *Interacting with a Database*.
`Application\Database\CustomerOrmService_1`	8 - 10

The second approach to this would be as follows:

Class	Technique #2 steps
`Application\Entity\Customer`	2
`Application\Database\CustomerOrmService_2`	3 - 6

In order to implement approach #1, where entities are embedded, define a calling program called `chap_11_orm_embedded.php`, which sets up autoloading and uses the appropriate classes:

```
<?php
define('DB_CONFIG_FILE',
'/../config/db.config.php');
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
use Application\Database\Connection;
use Application\Database\CustomerOrmService_1;
```

Next, create an instance of the service, and look up a customer using a random ID:

```
$service = new CustomerOrmService_1(new
Connection(include __DIR__ . DB_CONFIG_FILE));
$id   = rand(1,79);
$cust = $service->fetchByIdAndEmbedPurchases($id);
```

In the view logic, you will have acquired a fully populated `Customer` entity by way of the `fetchByIdAndEmbedPurchases()` method. Now all you need to do is to call the right getters to display information:

```
  <!-- Customer Info -->
  <h1><?= $cust->getname() ?></h1>
  <div class="row">
    <div class="left">Balance</div><div
class="right">
      <?= $cust->getBalance(); ?></div>
  </div>
    <!-- etc. -->
```

The logic needed to display purchase information would then look something like the following HTML. Notice that `Customer::getPurchases()` returns an array of `Purchase` entities. To get product information from the `Purchase` entity, inside the loop, call `Purchase::getProduct()`, which produces a `Product` entity. You can then call any of the `Product` getters, in this example, `Product::getTitle()`:

```
  <!-- Purchases Info -->
  <table>
  <?php foreach ($cust->getPurchases() as
$purchase) : ?>
  <tr>
  <td><?= $purchase->getTransaction() ?></td>
  <td><?= $purchase->getDate() ?></td>
  <td><?= $purchase->getQuantity() ?></td>
  <td><?= $purchase->getSalePrice() ?></td>
  <td><?= $purchase->getProduct()->getTitle() ?>
</td>
  </tr>
  <?php endforeach; ?>
</table>
```

Turning your attention to the second approach, which uses secondary lookups, define a calling program called `chap_11_orm_secondary_lookups.php`, which sets up autoloading and uses the appropriate classes:

```
<?php
define('DB_CONFIG_FILE',
'/../config/db.config.php');
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
use Application\Database\Connection;
use Application\Database\CustomerOrmService_2;
```

Next, create an instance of the service, and look up a customer using a random ID:

```
$service = new CustomerOrmService_2(new
Connection(include __DIR__ . DB_CONFIG_FILE));
$id   = rand(1,79);
```

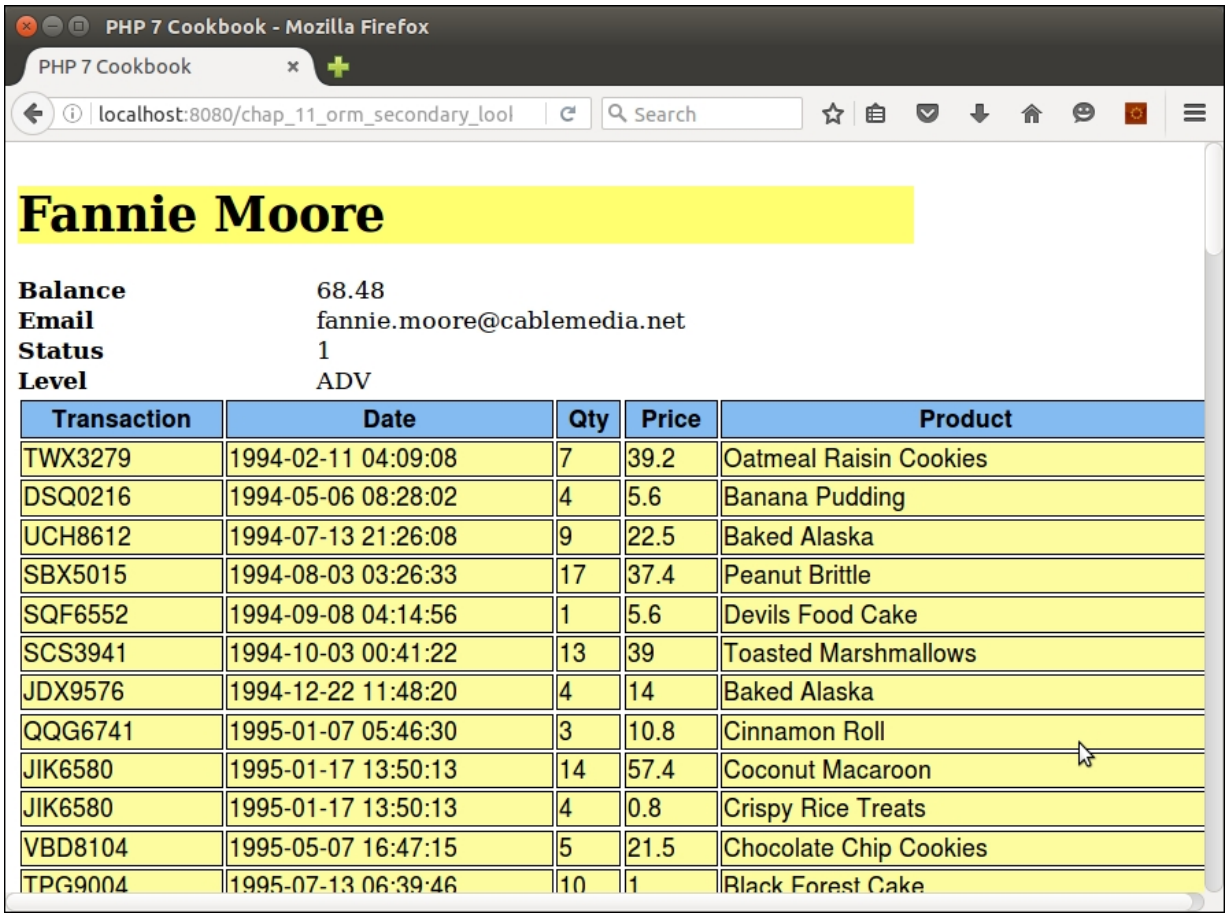
You can now retrieve an `Application\Entity\Customer` instance and call `fetchPurchasesForCustomer()` for this customer, which embeds the sequence of anonymous functions:

```
$cust = $service->fetchById($id);
$cust = $service-
>fetchPurchasesForCustomer($cust);
```

The view logic for displaying core customer information remains the same as described previously. The logic needed to display purchase information would then look something like the following HTML code snippet. Notice that `Customer::getPurchases()` returns an array of anonymous functions. Each function call returns one specific purchase and related products:

```
<table>
  <?php foreach($cust->getPurchases() as $purchId
=> $function) : ?>
  <tr>
  <?php $purchase = $function($purchId, $service);
?>
  <td><?= $purchase->getTransaction() ?></td>
  <td><?= $purchase->getDate() ?></td>
  <td><?= $purchase->getQuantity() ?></td>
  <td><?= $purchase->getSalePrice() ?></td>
  <td><?= $purchase->getProduct()->getTitle() ?>
</td>
  </tr>
  <?php endforeach; ?>
</table>
```

Here is an example of the output:



Tip

Best practice

Although each iteration of the loop represents two independent database queries (one for purchase, one for product), efficiency is retained by the use of *prepared statements*. Two statements are prepared in advance: one that looks up a specific purchase, and one that looks up a specific product. These prepared statements are then executed multiple times. Also, each product retrieval is independently stored in an array, resulting in even greater efficiency.

See also

Probably the best example of a library that implements object-relational mapping is Doctrine. Doctrine uses an embedded approach that its documentation refers to as a proxy. For more information, please refer to http://www.doctrine-project.org/projects/orm.html.

You might also consider reviewing a training video on *Learning Doctrine*, available from O'Reilly Media at http://shop.oreilly.com/product/0636920041382.do. (Disclaimer: this is a shameless plug by the author of both this book and this video!)

Implementing the Pub/Sub design pattern

The **Publish/Subscribe** (**Pub/Sub**) design pattern often forms the basis of software event-driven programming. This methodology allows **asynchronous** communications between different software applications, or different software modules within a single application. The purpose of the pattern is to allow a method or function to publish a signal when an action of significance has taken place. One or more classes would then subscribe and take action if a certain signal has been published.

Example of such actions are when the database is modified, or when a user has logged in. Another common use for this design pattern is when an application delivers news feeds. If an urgent news item has been posted, the application would publish this fact, allowing client subscribers to refresh their news listings.

How to do it...

1. First, we define our publisher class, `Application\PubSub\Publisher`. You'll notice that we are making use of two useful **Standard PHP Library** (**SPL**) interfaces, `SplSubject` and `SplObserver`:

```
namespace Application\PubSub;
use SplSubject;
use SplObserver;
class Publisher implements SplSubject
{
  // code
}
```

2. Next, we add properties to represent the publisher name, data to be passed to subscribers, and an array of subscribers (also referred to as listeners). You will also note that we will use a

linked list (described in Chapter 10, *Looking at Advanced Algorithms*) to allow for priority:

```
protected $name;
protected $data;
protected $linked;
protected $subscribers;
```

3. The constructor initializes these properties. We also throw in `__toString()` in case we need quick access to the name of this publisher:

```
public function __construct($name)
{
  $this->name = $name;
  $this->data = array();
  $this->subscribers = array();
  $this->linked = array();
}

public function __toString()
{
  return $this->name;
}
```

4. In order to associate a subscriber with this publisher, we define `attach()`, which is specified in the `SplSubject` interface. We accept an `SplObserver` instance as an argument. Note that we need to add entries to both the `$subscribers` and `$linked` properties. `$linked` is then sorted by value, represented by the priority, using `arsort()`, which sorts in reverse and maintains the key:

```
public function attach(SplObserver
$subscriber)
{
  $this->subscribers[$subscriber->getKey()] =
$subscriber;
  $this->linked[$subscriber->getKey()] =
    $subscriber->getPriority();
  arsort($this->linked);
}
```

5. The interface also requires us to define `detach()`, which removes the subscriber from the list:

```
public function detach(SplObserver
$subscriber)
{
  unset($this->subscribers[$subscriber-
>getKey()]);
  unset($this->linked[$subscriber->getKey()]);
}
```

6. Also required by the interface, we define `notify()`, which calls `update()` on all the subscribers. Note that we loop through the linked list to ensure the subscribers are called in order of priority:

```
public function notify()
{
  foreach ($this->linked as $key => $value)
  {
    $this->subscribers[$key]->update($this);
  }
}
```

7. Next, we define the appropriate getters and setters. We don't show them all here to conserve space:

```
public function getName()
{
  return $this->name;
}

public function setName($name)
{
  $this->name = $name;
}
```

8. Finally, we need to provide a means of setting data items by key, which will then be available to subscribers when `notify()` is invoked:

```
public function setDataByKey($key, $value)
{
  $this->data[$key] = $value;
}
```

9. Now we can have a look at `Application\PubSub\Subscriber`. Typically, we would define multiple subscribers for each publisher. In this case, we implement the `SplObserver` interface:

```
namespace Application\PubSub;
use SplSubject;
use SplObserver;
class Subscriber implements SplObserver
{
  // code
}
```

10. Each subscriber needs a unique identifier. In this case, we create the key using `md5()` and date/time information, combined with a random number. The constructor initializes the properties as follows. The actual logical functionality performed by the subscriber is in the form of a callback:

```
protected $key;
protected $name;
protected $priority;
protected $callback;
public function __construct(
  string $name, callable $callback, $priority
= 0)
{
  $this->key = md5(date('YmdHis') .
rand(0,9999));
  $this->name = $name;
  $this->callback = $callback;
  $this->priority = $priority;
}
```

11. The `update()` function is called when `notifiy()` on the publisher is invoked. We pass a publisher instance as an argument, and call the callback defined for this subscriber:

```
public function update(SplSubject $publisher)
{
  call_user_func($this->callback, $publisher);
}
```

12. We also need to define getters and setters for convenience. Not all are shown here:

```
    public function getKey()
    {
      return $this->key;
    }

    public function setKey($key)
    {
      $this->key = $key;
    }

    // other getters and setters not shown
```

How it works...

For the purposes of this illustration, define a calling program called `chap_11_pub_sub_simple_example.php`, which sets up autoloading and uses the appropriate classes:

```
<?php
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
use Application\PubSub\ { Publisher, Subscriber };
```

Next, create a publisher instance and assign data:

```
$pub = new Publisher('test');
$pub->setDataByKey('1', 'AAA');
$pub->setDataByKey('2', 'BBB');
$pub->setDataByKey('3', 'CCC');
$pub->setDataByKey('4', 'DDD');
```

Now you can create test subscribers that read data from the publisher and echo the results. The first parameter is the name, the second the callback, and the last is the priority:

```
$sub1 = new Subscriber(
  '1',
  function ($pub) {
```

```
    echo '1:' . $pub->getData()[1] . PHP_EOL;
  },
  10
);
$sub2 = new Subscriber(
  '2',
  function ($pub) {
    echo '2:' . $pub->getData()[2] . PHP_EOL;
  },
  20
);
$sub3 = new Subscriber(
  '3',
  function ($pub) {
    echo '3:' . $pub->getData()[3] . PHP_EOL;
  },
  99
);
```

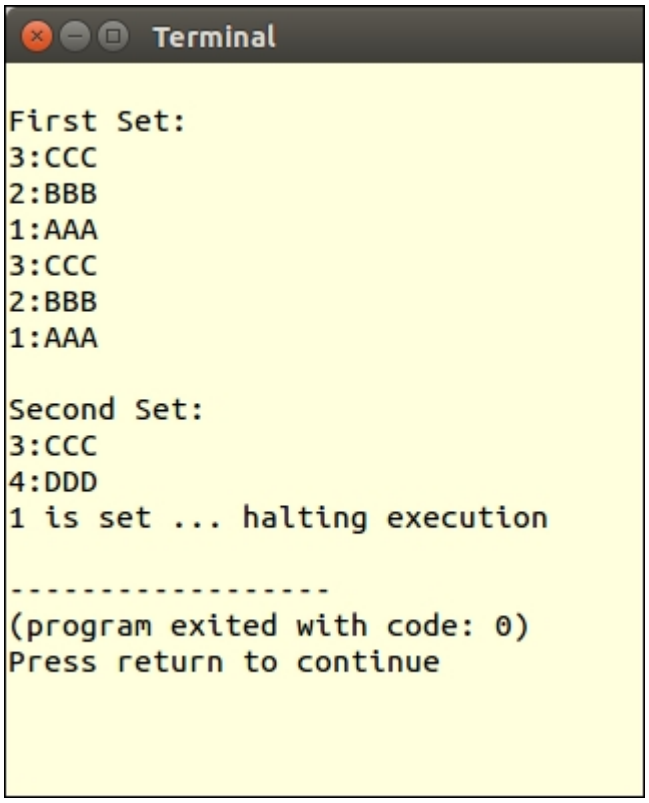
For test purposes, attach the subscribers out of order, and call `notify()` twice:

```
$pub->attach($sub2);
$pub->attach($sub1);
$pub->attach($sub3);
$pub->notify();
$pub->notify();
```

Next, define and attach another subscriber that looks at the data for subscriber 1 and exits if it's not empty:

```
$sub4 = new Subscriber(
  '4',
  function ($pub) {
    echo '4:' . $pub->getData()[4] . PHP_EOL;
    if (!empty($pub->getData()[1]))
      die('1 is set ... halting execution');
  },
  25
);
$pub->attach($sub4);
$pub->notify();
```

Here is the output. Note that the output is in order of priority (where higher priority goes first), and that the second block of output is interrupted:



There's more...

A closely related software design pattern is **Observer**. The mechanism is similar but the generally agreed difference is that Observer operates in a synchronous manner, where all observer methods are called when a signal (often also referred to as message or event) is received. The Pub/Sub pattern, in contrast, operates asynchronously, typically using a message queue. Another difference is that in the Pub/Sub pattern, publishers do not need to be aware of subscribers.

See also

For a good discussion on the difference between the Observer and Pub/Sub patterns, refer to the article at http://stackoverflow.com/questions/15594905/difference-between-observer-pub-sub-and-data-binding.

Chapter 12. Improving Web Security

In this chapter, we will cover the following topics:

- Filtering `$_POST` data
- Validating `$_POST` data
- Safeguarding the PHP session
- Securing forms with a token
- Building a secure password generator
- Safeguarding forms with a CAPTCHA
- Encrypting/decrypting without `mcrypt`

Introduction

In this chapter, we will show you how to set up a simple yet effective mechanism for filtering and validating a block of post data. Then, we will cover how to protect your PHP sessions from potential session hijacking and other forms of attack. The next recipe shows how to protect forms from **Cross Site Request Forgery** (**CSRF**) attacks using a randomly generated token. The recipe on password generation shows you how to incorporate PHP 7 true randomization to generate secure passwords. We then show you two forms of **CAPTCHA**: one that is text based, the other using a distorted image. Finally, there is a recipe that covers strong encryption without using the discredited and soon-to-be-deprecated `mcrypt` extension.

Filtering $_POST data

The process of filtering data can encompass any or all of the following:

- Removing unwanted characters (that is, removing `<script>` tags)
- Performing transformations on the data (that is, converting a quote to `"`)
- Encrypting or decrypting the data

Encryption is covered in the last recipe of this chapter. Otherwise, we will present a basic mechanism that can be used to filter `$_POST` data arriving following form submission.

How to do it...

1. First of all, you need to have an awareness of the data that will be present in `$_POST`. Also, perhaps more importantly, you will need to be aware of the restrictions imposed by the database table in which the form data will presumably be stored. As an example, have a look at the database structure for the `prospects` table:

```
COLUMN          TYPE            NULL
DEFAULT
first_name      varchar(128)    No      None
NULL
last_name       varchar(128)    No      None
NULL
address         varchar(256)    Yes     None
NULL
city            varchar(64)     Yes     None
NULL
state_province  varchar(32)     Yes     None
NULL
postal_code     char(16)        No      None
NULL
phone           varchar(16)     No      None
```

```
NULL
country         char(2)           No     None
NULL
email           varchar(250)      No     None
NULL
status          char(8)           Yes    None
NULL
budget          decimal(10,2)     Yes    None
NULL
last_updated    datetime          Yes    None
NULL
```

2. Once you have completed an analysis of the data to be posted and stored, you can determine what type of filtering is to occur, and which PHP functions will serve this purpose.
3. As an example, if you need to get rid of leading and trailing white space, which is completely possible from user supplied form data, you can use the PHP `trim()` function. All of the character data has length limits according to the database structure. Accordingly, you might consider using `substr()` to ensure the length is not exceeded. If you wanted to remove non-alphabetical characters, you might consider using `preg_replace()` with the appropriate pattern.
4. We can now group the set of desired PHP functions into a single array of callbacks. Here is an example based on the filtering needs for the form data that will eventually be stored in the `prospects` table:

```
$filter = [
  'trim' => function ($item) { return
trim($item); },
  'float' => function ($item) { return (float)
$item; },
  'upper' => function ($item) { return
strtoupper($item); },
  'email' => function ($item) {
     return filter_var($item,
FILTER_SANITIZE_EMAIL); },
  'alpha' => function ($item) {
     return preg_replace('/[^A-Za-z]/', '',
$item); },
  'alnum' => function ($item) {
     return preg_replace('/[^0-9A-Za-z ]/',
'', $item); },
```

```
  'length' => function ($item, $length) {
     return substr($item, 0, $length); },
  'stripTags' => function ($item) { return
strip_tags($item); },
];
```

5. Next, we define an array that matches the field names expected in `$_POST`. In this array, we specify the key in the `$filter` array, along with any parameters. Note the first key, `*`. We will use that as a wildcard to be applied to all fields:

```
$assignments = [
  '*'              => ['trim' => NULL,
'stripTags' => NULL],
  'first_name'     => ['length' => 32, 'alnum'
=> NULL],
  'last_name'      => ['length' => 32, 'alnum'
=> NULL],
  'address'        => ['length' => 64, 'alnum'
=> NULL],
  'city'           => ['length' => 32],
  'state_province'=> ['length' => 20],
  'postal_code'    => ['length' => 12, 'alnum'
=> NULL],
  'phone'          => ['length' => 12],
  'country'        => ['length' => 2, 'alpha'
=> NULL,
                       'upper' => NULL],
  'email'          => ['length' => 128, 'email'
=> NULL],
  'budget'         => ['float' => NULL],
];
```

6. We then loop through the data set (that is, coming from `$_POST`) and apply the callbacks in turn. We first run all callbacks assigned to the wildcard (`*`) key.

> ## Note
>
> It is important to implement a wildcard filter to avoid redundant settings. In the preceding example, we wish to

apply filters that represent the PHP functions `strip_tags()` and `trim()` for every item.

7. Next, we run through all callbacks assigned to a particular data field. When we're done, all values in `$data` will be filtered:

```
foreach ($data as $field => $item) {
  foreach ($assignments['*'] as $key =>
$option) {
    $item = $filter[$key]($item, $option);
  }
  foreach ($assignments[$field] as $key =>
$option) {
    $item = $filter[$key]($item, $option);
  }
}
```

How it works...

Place the code shown in steps 4 through 6 into a file called `chap_12_post_data_filtering_basic.php`. You will also need to define an array to simulate data that would be present in `$_POST`. In this case, you could define two arrays, one with *good* data, and one with *bad* data:

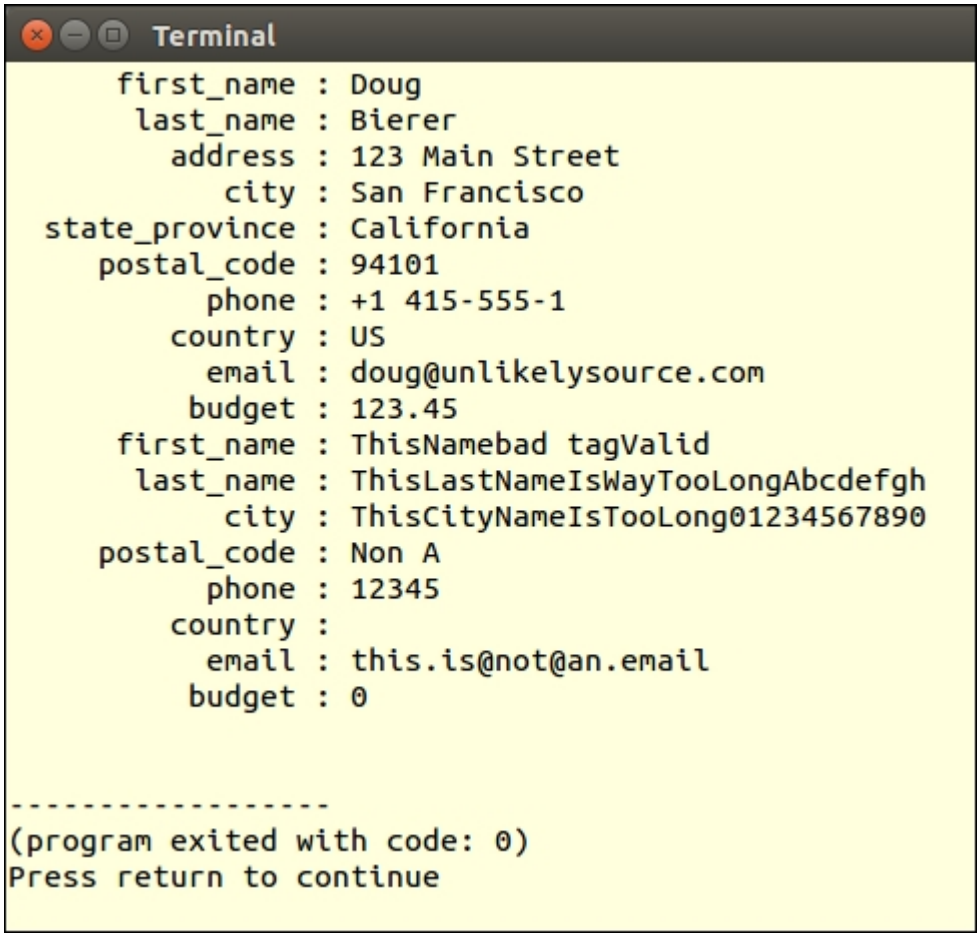
```
$testData = [
  'goodData'   => [
    'first_name'     => 'Doug',
    'last_name'      => 'Bierer',
    'address'        => '123 Main Street',
    'city'           => 'San Francisco',
    'state_province'=> 'California',
    'postal_code'    => '94101',
    'phone'          => '+1 415-555-1212',
    'country'        => 'US',
    'email'          => 'doug@unlikelysource.com',
    'budget'         => '123.45',
  ],
  'badData' => [
    'first_name' => 'This+Name<script>bad
tag</script>Valid!',
```

```
    'last_name'         =>
'ThisLastNameIsWayTooLongAbcdefghijklmnopqrstuvwxy
z0123456789Abcdefghijklmnopqrstuvwxyz0123456789Abc
defghijklmnopqrstuvwxyz0123456789Abcdefghijklmnopq
rstuvwxyz0123456789',
    //'address'         => '',    // missing
    'city'      =>
'ThisCityNameIsTooLong012345678901234567890123456
8901234567890123456789012345678901234567890  ',
    //'state_province'=> '',    // missing
    'postal_code'     => '!"£$%^Non Alpha Chars',
    'phone'           => ' 12345 ',
    'country'         => '12345',
    'email'           => 'this.is@not@an.email',
    'budget'          => 'XXX',
  ]
];
```

Finally, you will need to loop through the filter assignments, presenting the good and bad data:

```
foreach ($testData as $data) {
  foreach ($data as $field => $item) {
    foreach ($assignments['*'] as $key => $option)
{
      $item = $filter[$key]($item, $option);
    }
    foreach ($assignments[$field] as $key =>
$option) {
      $item = $filter[$key]($item, $option);
    }
    printf("%16s : %s\n", $field, $item);
  }
}
```

Here's how the output might appear for this example:



Note that the names were truncated and tags were removed. You will also note that although the e-mail address was filtered, it is still not a valid address. It's important to note that for proper treatment of data, it might be necessary to *validate* as well as to filter.

See also

In Chapter 6, *Building Scalable Websites*, the recipe entitled *Chaining $_POST filters*, discusses how to incorporate the basic filtering concepts covered here into a comprehensive filter chaining mechanism.

Validating $_POST data

The primary difference between filtering and validation is that the latter does not alter the original data. Another difference is in intent. The purpose of validation is to confirm that the data matches certain criteria established according to the needs of your customer.

How to do it...

1. The basic validation mechanism we will present here is identical to that shown in the preceding recipe. As with filtering, it is vital to have an idea of the nature of the data to be validated, how it fits your customer's requirements, and also whether it matches the criteria enforced by the database. For example, if in the database, the maximum width of the column is 128, the validation callback could use `strlen()` to confirm that the length of the data submitted is less than or equal to 128 characters. Likewise, you could use `ctype_alnum()` to confirm that the data only contains letters and numbers, as appropriate.
2. Another consideration for validation is to present an appropriate validation failure message. The validation process, in a certain sense, is also a *confirmation* process, where somebody presumably will review the validation to confirm success or failure. If the validation fails, that person will need to know the reason why.
3. For this illustration, we will again focus on the `prospects` table. We can now group the set of desired PHP functions into a single array of callbacks. Here is an example based on the validation needs for the form data, which will eventually be stored in the `prospects` table:

```
$validator = [
  'email' => [
    'callback' => function ($item) {
      return filter_var($item,
FILTER_VALIDATE_EMAIL); },
    'message'  => 'Invalid email address'],
```

```
  'alpha' => [
    'callback' => function ($item) {
      return ctype_alpha(str_replace(' ', '',
$item)); },
    'message'  => 'Data contains non-alpha
characters'],
  'alnum' => [
    'callback' => function ($item) {
      return ctype_alnum(str_replace(' ', '',
$item)); },
    'message'  => 'Data contains characters
which are '
       . 'not letters or numbers'],
  'digits' => [
    'callback' => function ($item) {
      return preg_match('/[^0-9.]/', $item);
},
    'message'  => 'Data contains characters
which '
      . 'are not numbers'],
  'length' => [
    'callback' => function ($item, $length) {
      return strlen($item) <= $length; },
    'message'  => 'Item has too many
characters'],
  'upper' => [
    'callback' => function ($item) {
      return $item == strtoupper($item); },
    'message'  => 'Item is not upper case'],
  'phone' => [
    'callback' => function ($item) {
      return preg_match('/[^0-9() -+]/',
$item); },
    'message'  => 'Item is not a valid phone
number'],
];
```

Note

Notice, for the alpha and alnum callbacks, we allow for whitespace by first removing it using `str_replace()`. We can then call `ctype_alpha()` or `ctype_alnum()`, which will determine whether any disallowed characters are present.

4. Next, we define an array of assignments that matches the field names expected in `$_POST`. In this array, we specify the key in the `$validator` array, along with any parameters:

```
$assignments = [
  'first_name'    => ['length' => 32, 'alpha'
=> NULL],
  'last_name'     => ['length' => 32, 'alpha'
=> NULL],
  'address'       => ['length' => 64, 'alnum'
=> NULL],
  'city'          => ['length' => 32, 'alnum'
=> NULL],
  'state_province'=> ['length' => 20, 'alpha'
=> NULL],
  'postal_code'   => ['length' => 12, 'alnum'
=> NULL],
  'phone'         => ['length' => 12, 'phone'
=> NULL],
  'country'       => ['length' => 2, 'alpha'
=> NULL,
                      'upper' => NULL],
  'email'         => ['length' => 128, 'email'
=> NULL],
  'budget'        => ['digits' => NULL],
];
```

5. We then use nested `foreach()` loops to iterate through the block of data one field at a time. For each field, we loop through the callbacks assigned to that field:

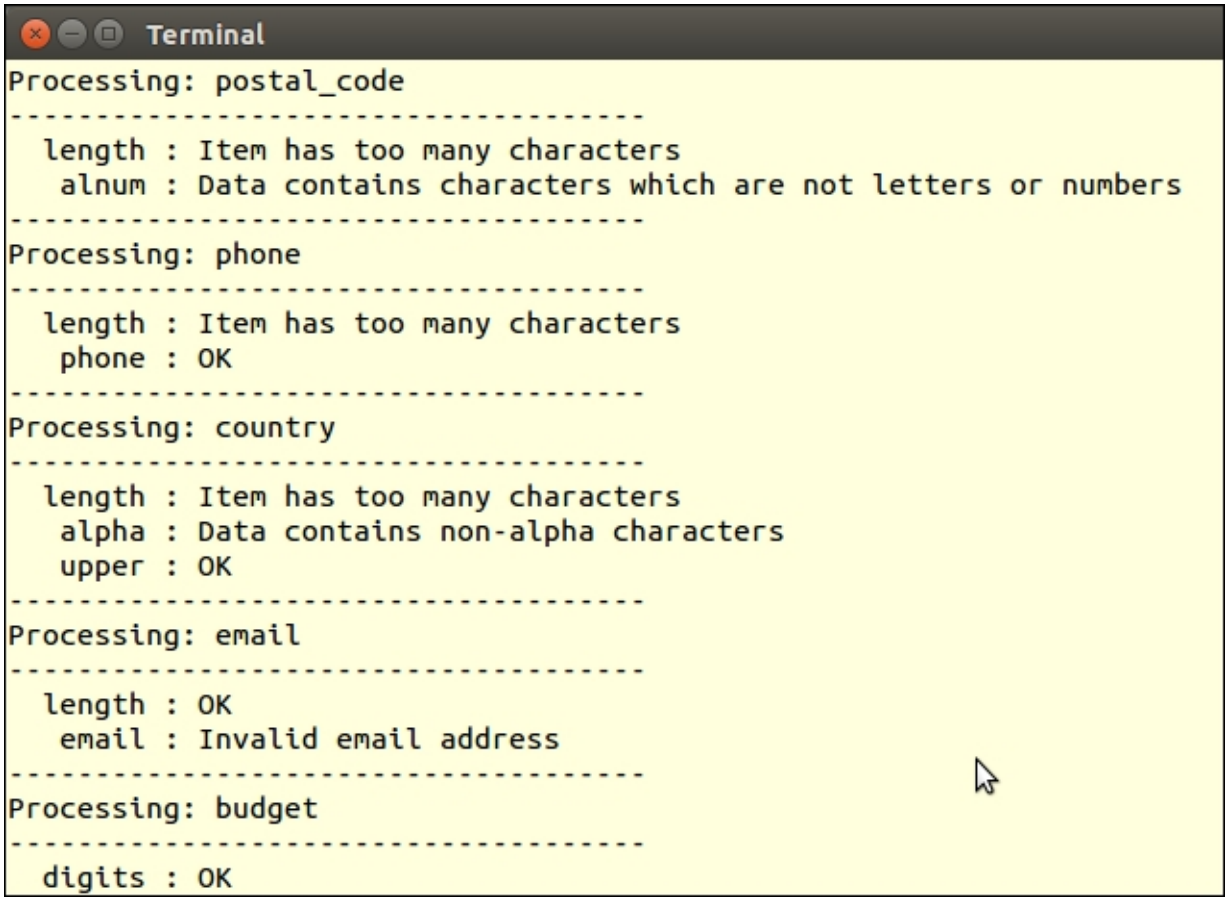
```
foreach ($data as $field => $item) {
  echo 'Processing: ' . $field . PHP_EOL;
  foreach ($assignments[$field] as $key =>
$option) {
    if ($validator[$key]['callback']($item,
$option)) {
        $message = 'OK';
    } else {
        $message = $validator[$key]
['message'];
    }
    printf('%8s : %s' . PHP_EOL, $key,
$message);
```

```
        }
    }
```

> **Tip**
>
> Instead of echoing the output directly, as shown, you might log the validation success/failure to be presented to the reviewer at a later time. Also, as shown in Chapter 6, *Building Scalable Websites*, you can work the validation mechanism into the form, displaying validation messages next to their matching form elements.

How it works...

Place the code shown in steps 3 through 5 into a file called `chap_12_post_data_validation_basic.php`. You will also need to define an array of data that simulates data that would be present in `$_POST`. In this case, you use the two arrays mentioned in the preceding recipe, one with *good* data, and one with *bad* data. The final output should look something like this:



See also

- In Chapter 6, *Building Scalable Websites*, the recipe entitled *Chaining $_POST validators* discusses how to incorporate the basic validation concepts covered here into a comprehensive filter chaining mechanism.

Safeguarding the PHP session

The PHP session mechanism is quite simple. Once the session is started using `session_start()` or the `php.ini session.autostart` setting, the PHP engine generates a unique token that is, by default, conveyed to the user by way of a cookie. On subsequent requests, while the session is still considered active, the user's browser (or equivalent) presents the session identifier, again usually by way of a cookie, for inspection. The PHP engine then uses this identifier to locate the appropriate file on the server, populating `$_SESSION` with the stored information. There are tremendous security concerns when the session identifier is the sole means of identifying a returning website visitor. In this recipe, we will present several techniques that will help you to safeguard your sessions, which, in turn, will vastly improve the overall security of the website.

How to do it...

1. First of all, it's important to recognize how using the session as the sole means of authentication can be dangerous. Imagine for a moment that when a valid user logs in to your website, that you set a `loggedIn` flag in `$_SESSION`:

```
session_start();
$loggedIn = $_SESSION['isLoggedIn'] ?? FALSE;
if (isset($_POST['login'])) {
  if ($_POST['username'] == // username lookup
      && $_POST['password'] == // password
lookup) {
      $loggedIn = TRUE;
      $_SESSION['isLoggedIn'] = TRUE;
  }
}
```

2. In your program logic, you allow the user to see sensitive information if `$_SESSION['isLoggedIn']` is set to `TRUE`:

```
<br>Secret Info
<br><?php if ($loggedIn) echo // secret
```

```
information; ?>
```

3. If an attacker were to obtain the session identifier, for example, by means of a successfully executed **Cross-site scripting** (**XSS**) attack, all he/she would need to do would be to set the value of the `PHPSESSID` cookie to the illegally obtained one, and they are now viewed by your application as a valid user.
4. One quick and easy way to narrow the window of time during which the `PHPSESSID` is valid is to use `session_regenerate_id()`. This very simple command generates a new session identifier, invalidates the old one, maintains session data intact, and has a minimal impact on performance. This command can only be executed after the session has started:

```
session_start();
session_regenerate_id();
```

5. Another often overlooked technique is to ensure that web visitors have a logout option. It is important, however, to not only destroy the session using `session_destroy()`, but also to unset `$_SESSION` data and to expire the session cookie:

```
session_unset();
session_destroy();
setcookie('PHPSESSID', 0, time() - 3600);
```

6. Another easy technique that can be used to prevent session hijacking is to develop a finger-print or thumb-print of the website visitor. One way to implement this technique is to collect information unique to the website visitor over and above the session identifier. Such information includes the user agent (that is, the browser), languages accepted, and remote IP address. You can derive a simple hash from this information, and store the hash on the server in a separate file. The next time the user visits the website, if you have determined they are logged in based on session information, you can then perform a secondary verification by matching finger-prints:

```
$remotePrint = md5($_SERVER['REMOTE_ADDR']
                   .
$_SERVER['HTTP_USER_AGENT']
```

```
                    .
    $_SERVER['HTTP_ACCEPT_LANGUAGE']);
    $printsMatch = file_exists(THUMB_PRINT_DIR .
    $remotePrint);
    if ($loggedIn && !$printsMatch) {
        $info = 'SESSION INVALID!!!';
        error_log('Session Invalid: ' . date('Y-m-
    d H:i:s'), 0);
        // take appropriate action
    }
```

> **Note**
>
> We are using `md5()` as it's a fast hashing algorithm and is well suited for internal usage. It is *not recommended* to use `md5()` for any external use as it is subject to brute-force attacks.

How it works...

To demonstrate how a session is vulnerable, code a simple login script that sets a `$_SESSION['isLoggedIn'] flag` upon successful login. You could call the file `chap_12_session_hijack.php`:

```
session_start();
$loggedUser = $_SESSION['loggedUser'] ?? '';
$loggedIn = $_SESSION['isLoggedIn'] ?? FALSE;
$username = 'test';
$password = 'password';
$info = 'You Can Now See Super Secret
Information!!!';

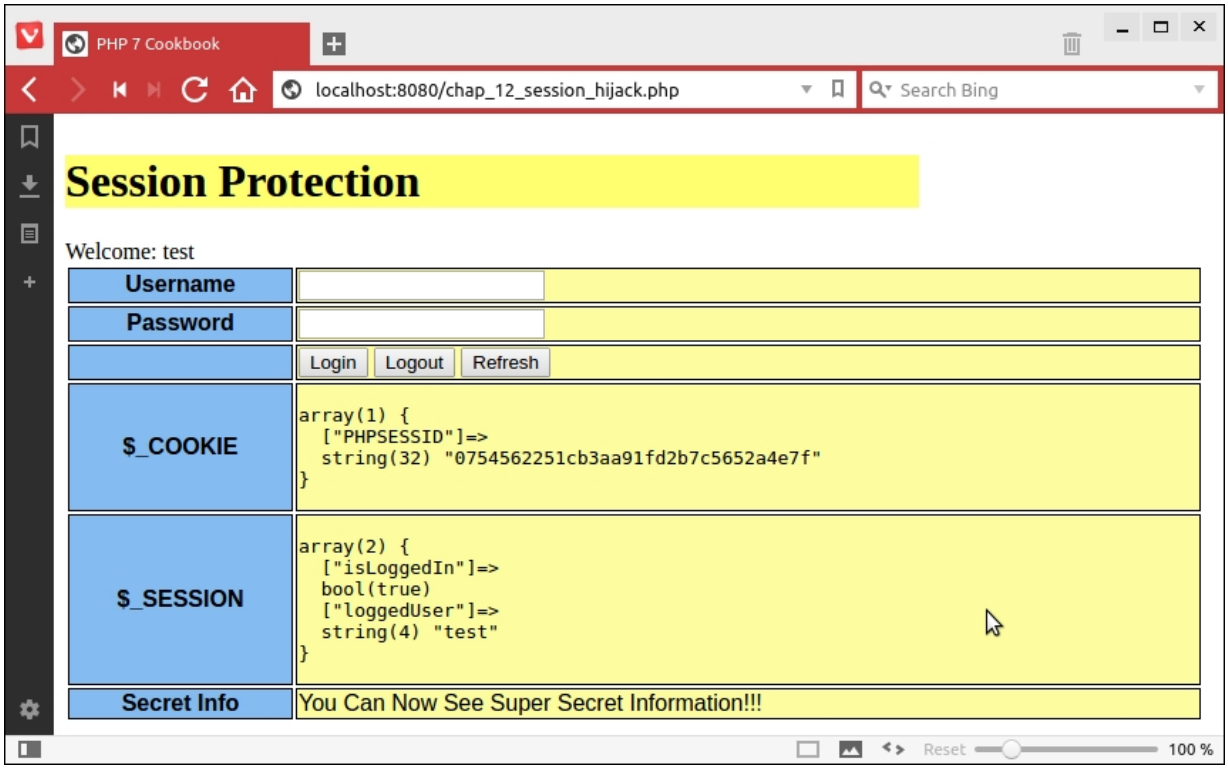
if (isset($_POST['login'])) {
  if ($_POST['username'] == $username
      && $_POST['password'] == $password) {
        $loggedIn = TRUE;
        $_SESSION['isLoggedIn'] = TRUE;
        $_SESSION['loggedUser'] = $username;
        $loggedUser = $username;
```

```
    }
  } elseif (isset($_POST['logout'])) {
    session_destroy();
  }
```

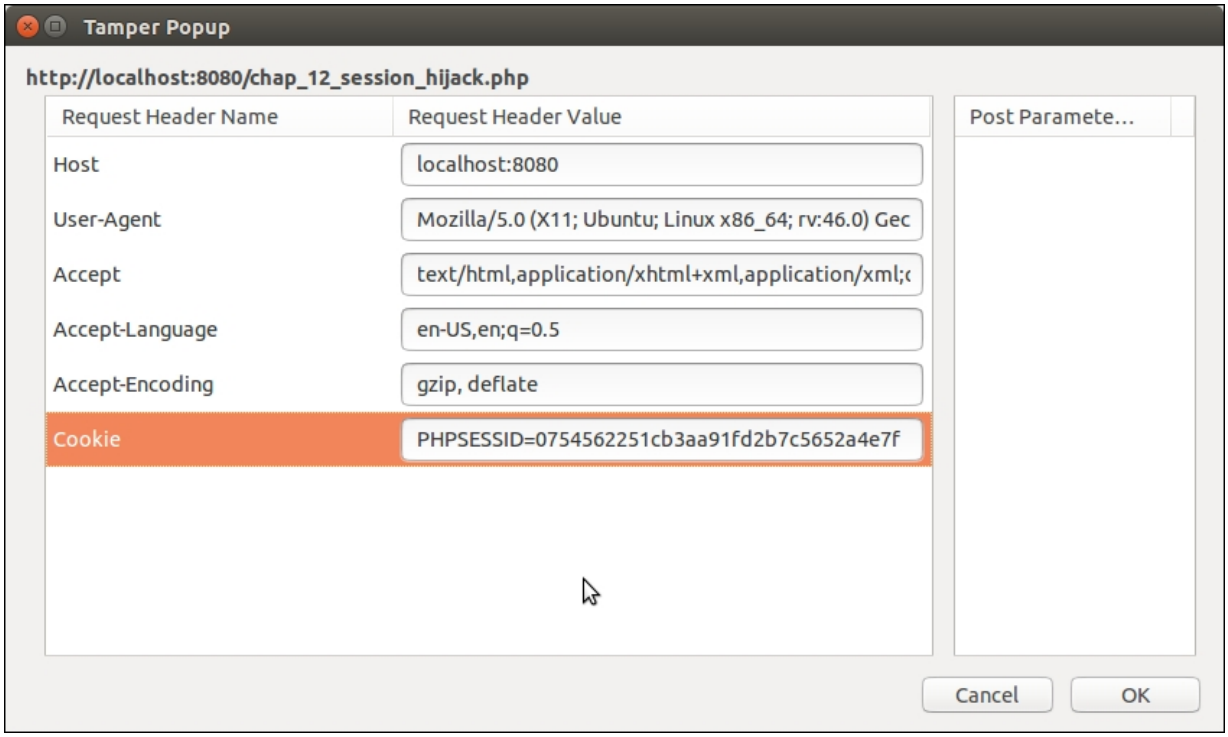
You can then add code that displays a simple login form. To test for session vulnerability, follow this procedure using the `chap_12_session_hijack.php` file we just created:

1. Change to the directory containing the file.
2. Run the `php -S localhost:8080` command.
3. Using one browser, open the URL `http://localhost:8080/<filename>`.
4. Login as user `test` with a password as `password`.
5. You should be able to see **You Can Now See Super Secret Information!!!**.
6. Refresh the page: each time, you should see a new session identifier.
7. Copy the value of the `PHPSESSID` cookie.
8. Open another browser to the same web page.
9. Modify the cookie sent by the browser by copying the value of `PHPSESSID`.

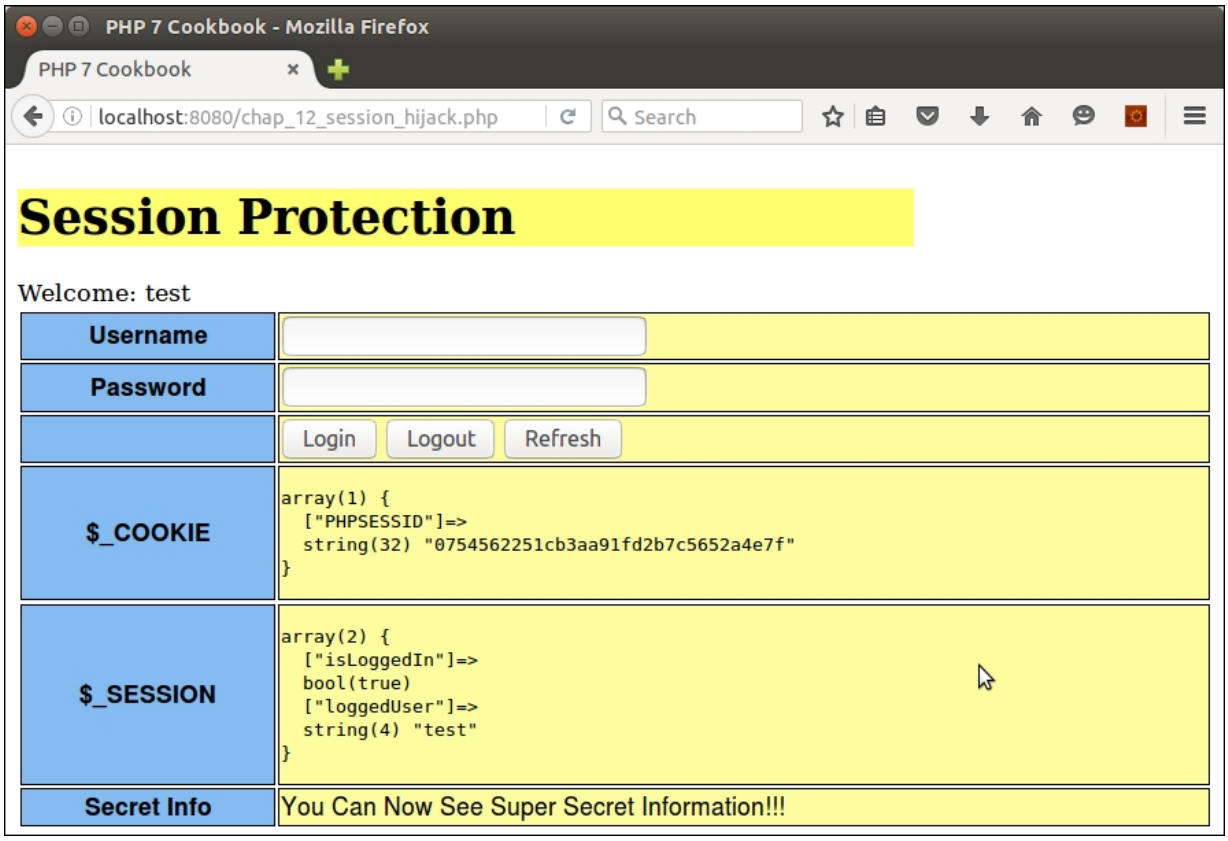
For illustration, we are also showing the value of `$_COOKIE` and `$_SESSION`, shown in the following screenshot using the Vivaldi browser:



We then copy the value of `PHPSESSID`, open a Firefox browser, and use a tool called Tamper Data to modify the value of the cookie:



You can see in the next screenshot that we are now an authenticated user without entering the username or password:



You can now implement the changes discussed in the preceding steps. Copy the file created previously to `chap_12_session_protected.php`. Now go ahead and regenerate the session ID:

```
<?php
define('THUMB_PRINT_DIR', __DIR__ . '/../data/');
session_start();
session_regenerate_id();
```

Next, initialize variables and determine the logged in status (as before):

```
$username = 'test';
$password = 'password';
$info = 'You Can Now See Super Secret
```

```
Information!!!';
$loggedIn = $_SESSION['isLoggedIn'] ?? FALSE;
$loggedUser = $_SESSION['user'] ?? 'guest';
```

You can add a session thumb-print using the remote address, user agent, and language settings:

```
$remotePrint = md5($_SERVER['REMOTE_ADDR']
  . $_SERVER['HTTP_USER_AGENT']
  . $_SERVER['HTTP_ACCEPT_LANGUAGE']);
$printsMatch = file_exists(THUMB_PRINT_DIR .
$remotePrint);
```

If the login is successful, we store thumb-print info and login status in the session:

```
if (isset($_POST['login'])) {
  if ($_POST['username'] == $username
      && $_POST['password'] == $password) {
        $loggedIn = TRUE;
        $_SESSION['user'] = strip_tags($username);
        $_SESSION['isLoggedIn'] = TRUE;
        file_put_contents(
          THUMB_PRINT_DIR . $remotePrint,
$remotePrint);
  }
```

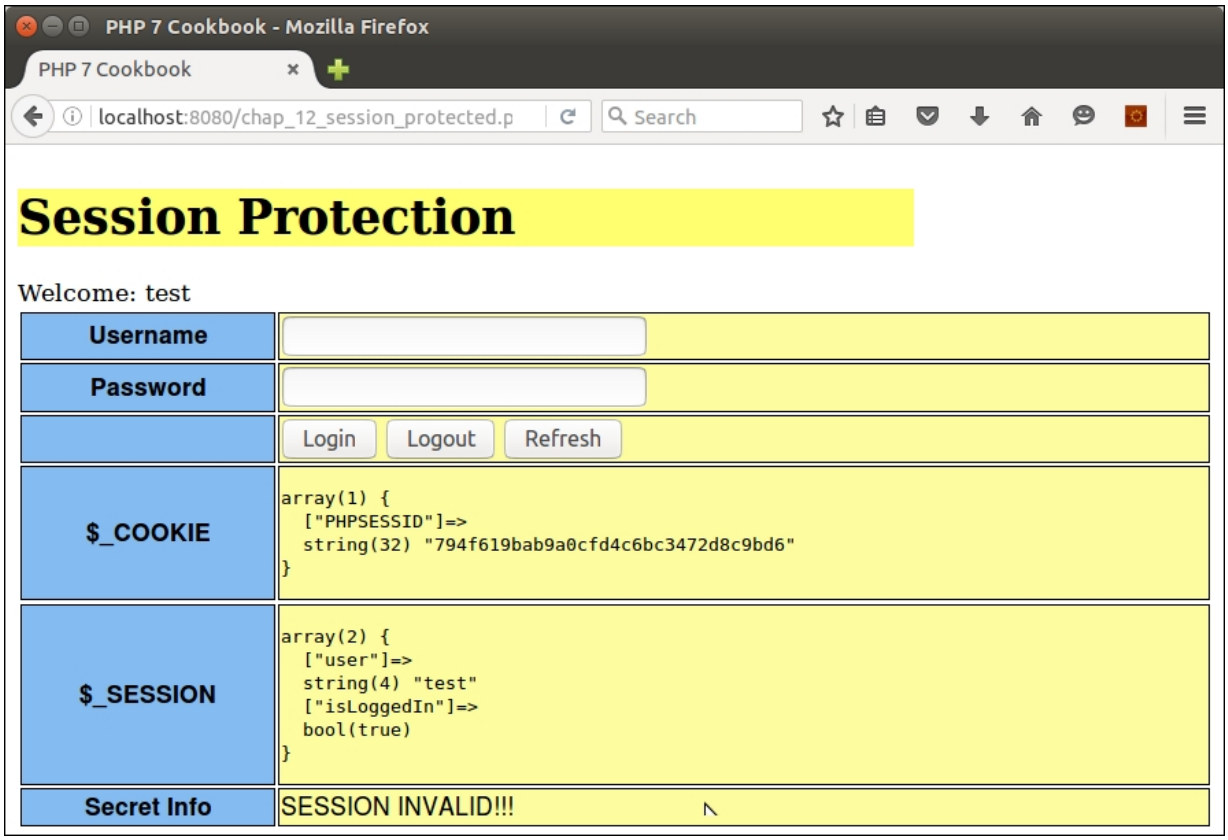
You can also check for the logout option and implement a proper logout procedure: unset `$_SESSION` variables, invalidate the session, and expire the cookie. You can also remove the thumb-print file and implement a redirect:

```
} elseif (isset($_POST['logout'])) {
  session_unset();
  session_destroy();
  setcookie('PHPSESSID', 0, time() - 3600);
  if (file_exists(THUMB_PRINT_DIR . $remotePrint))
    unlink(THUMB_PRINT_DIR . $remotePrint);
    header('Location: ' . $_SERVER['REQUEST_URI']
);
  exit;
```

Otherwise, if the operation is not login or logout, you can check to see whether the user is considered logged in, and if the thumb-print doesn't match, the session is considered invalid, and the appropriate action is taken:

```
} elseif ($loggedIn && !$printsMatch) {
    $info = 'SESSION INVALID!!!';
    error_log('Session Invalid: ' . date('Y-m-d
H:i:s'), 0);
    // take appropriate action
}
```

You can now run the same procedure as mentioned previously using the new `chap_12_session_protected.php` file. The first thing you will notice is that the session is now considered invalid. The output will look something like this:



The reason for this is that the thumb-print does not match as you are now using a different browser. Likewise, if you refresh the page of

the first browser, the session identifier is regenerated, making any previously copied identifier obsolete. Finally, the logout button will completely clear session information.

See also

For an excellent overview of website vulnerabilities, please refer to the article present at https://www.owasp.org/index.php/Category:Vulnerability. For information on session hijacking, refer to https://www.owasp.org/index.php/Session_hijacking_attack.

Securing forms with a token

This recipe presents another very simple technique that will safeguard your forms against **Cross Site Request Forgery** (**CSRF**) attacks. Simply put, a CSRF attack is possible when, possibly using other techniques, an attacker is able to infect a web page on your website. In most cases, the infected page will then start issuing requests (that is, using JavaScript to purchase items, or make settings changes) using the credentials of a valid, logged-in user. It's extremely difficult for your application to detect such activity. One measure that can easily be taken is to generate a random token that is included in every form to be submitted. Since the infected page will not have access to the token, nor have the ability to generate one that matches, form validation will fail.

How to do it...

1. First, to demonstrate the problem, we create a web page that simulates an infected page that generates a request to post an entry to the database. For this illustration, we will call the file `chap_12_form_csrf_test_unprotected.html`:

```
<!DOCTYPE html>
  <body onload="load()">
  <form action="/chap_12_form_unprotected.php"
    method="post" id="csrf_test"
name="csrf_test">
    <input name="name" type="hidden" value="No
Goodnick" />
    <input name="email" type="hidden"
value="malicious@owasp.org" />
    <input name="comments" type="hidden"
       value="Form is vulnerable to CSRF
attacks!" />
    <input name="process" type="hidden"
value="1" />
  </form>
  <script>
    function load() {
document.forms['csrf_test'].submit(); }
```

```
  </script>
</body>
</html>
```

2. Next, we create a script called `chap_12_form_unprotected.php` that responds to the form posting. As with other calling programs in this book, we set up autoloading and use the `Application\Database\Connection` class covered in Chapter 5, *Interacting with a Database*:

```
<?php
define('DB_CONFIG_FILE',
'/../config/db.config.php');
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
use Application\Database\Connection;
$conn = new Connection(include __DIR__ .
DB_CONFIG_FILE);
```

3. We then check to see the process button has been pressed, and even implement a filtering mechanism, as covered in the *Filtering $_POST data* recipe in this chapter. This is to prove that a CSRF attack is easily able to bypass filters:

```
if ($_POST['process']) {
    $filter = [
      'trim' => function ($item) { return
trim($item); },
      'email' => function ($item) {
        return filter_var($item,
FILTER_SANITIZE_EMAIL); },
      'length' => function ($item, $length) {
        return substr($item, 0, $length); },
      'stripTags' => function ($item) {
      return strip_tags($item); },
  ];

  $assignments = [
    '*'          => ['trim' => NULL,
'stripTags' => NULL],
    'email'    => ['length' => 249, 'email' =>
NULL],
    'name'     => ['length' => 128],
```

```
    'comments'=> ['length' => 249],
  ];

  $data = $_POST;
  foreach ($data as $field => $item) {
    foreach ($assignments['*'] as $key =>
$option) {
      $item = $filter[$key]($item, $option);
    }
    if (isset($assignments[$field])) {
      foreach ($assignments[$field] as $key =>
$option) {
        $item = $filter[$key]($item, $option);
      }
      $filteredData[$field] = $item;
    }
  }
```

4. Finally, we insert the filtered data into the database using a prepared statement. We then redirect to another script, called `chap_12_form_view_results.php`, which simply dumps the contents of the `visitors` table:

```
try {
    $filteredData['visit_date'] = date('Y-m-d
H:i:s');
    $sql = 'INSERT INTO visitors '
        . ' (email,name,comments,visit_date) '
        . 'VALUES
(:email,:name,:comments,:visit_date)';
    $insertStmt = $conn->pdo->prepare($sql);
    $insertStmt->execute($filteredData);
} catch (PDOException $e) {
    echo $e->getMessage();
}
}
header('Location:
/chap_12_form_view_results.php');
exit;
```

5. The result, of course, is that the attack is allowed, despite filtering and the use of prepared statements.
6. Implementing the form protection token is actually quite easy! First of all, you need to generate the token and store it in the session. We take advantage of the new `random_bytes()` PHP 7

function to generate a truly random token, one which will be difficult, if not impossible, for an attacker to match:

```
session_start();
$token =
urlencode(base64_encode((random_bytes(32))));
$_SESSION['token'] = $token;
```

> **Note**
>
> The output of `random_bytes()` is binary. We use `base64_encode()` to convert it into a usable string. We then further process it using `urlencode()` so that it is properly rendered in an HTML form.

7. When we render the form, we then present the token as a hidden field:

```
<input type="hidden" name="token" value="<?=
$token ?>" />
```

8. We then copy and alter the `chap_12_form_unprotected.php` script mentioned previously, adding logic to first check to see whether the token matches the one stored in the session. Note that we unset the current token to make it invalid for future use. We call the new script `chap_12_form_protected_with_token.php`:

```
if ($_POST['process']) {
    $sessToken = $_SESSION['token'] ?? 1;
    $postToken = $_POST['token'] ?? 2;
    unset($_SESSION['token']);
    if ($sessToken != $postToken) {
        $_SESSION['message'] = 'ERROR: token
mismatch';
    } else {
        $_SESSION['message'] = 'SUCCESS: form
processed';
        // continue with form processing
    }
}
```

How it works...

To test how an infected web page might launch a CSRF attack, create the following files, as shown earlier in the recipe:

- `chap_12_form_csrf_test_unprotected.html`
- `chap_12_form_unprotected.php`

You can then define a file called `chap_12_form_view_results.php`, which dumps the `visitors` table:

```
<?php
session_start();
define('DB_CONFIG_FILE',
'/../config/db.config.php');
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
use Application\Database\Connection;
$conn = new Connection(include __DIR__ .
DB_CONFIG_FILE);
$message = $_SESSION['message'] ?? '';
unset($_SESSION['message']);
$stmt = $conn->pdo->query('SELECT * FROM
visitors');
?>
<!DOCTYPE html>
<body>
<div class="container">
  <h1>CSRF Protection</h1>
  <h3>Visitors Table</h3>
  <?php while ($row = $stmt-
>fetch(PDO::FETCH_ASSOC)) : ?>
  <pre><?php echo implode(':', $row); ?></pre>
  <?php endwhile; ?>
  <?php if ($message) : ?>
  <b><?= $message; ?></b>
  <?php endif; ?>
</div>
</body>
</html>
```

From a browser, launch `chap_12_form_csrf_test_unprotected.html`. Here is how the output might appear:

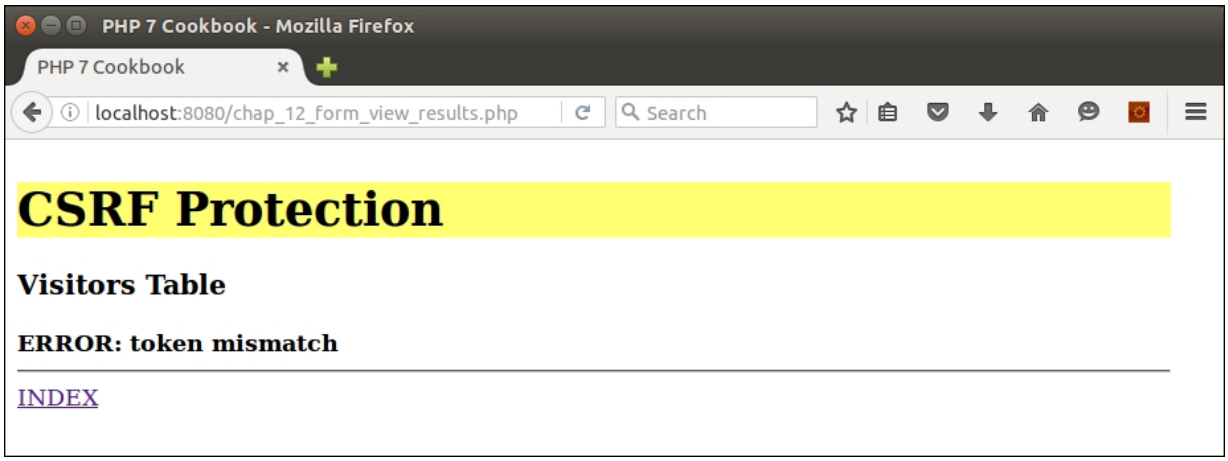


As you can see, the attack was successful despite filtering and the use of prepared statements!

Next, copy the `chap_12_form_unprotected.php` file to `chap_12_form_protected.php`. Make the change indicated in step 8 in the recipe. You will also need to alter the test HTML file, copying `chap_12_form_csrf_test_unprotected.html` to `chap_12_form_csrf_test_protected.html`. Change the value for the action parameter in the `FORM` tag as follows:

```
<form
action="/chap_12_form_protected_with_token.php"
  method="post" id="csrf_test" name="csrf_test">
```

When you run the new HTML file from a browser, it calls `chap_12_form_protected.php`, which looks for a token that does not exist. Here is the expected output:



Finally, go ahead and define a file called `chap_12_form_protected.php` that generates a token and displays it as a hidden element:

```
<?php
session_start();
$token =
urlencode(base64_encode((random_bytes(32))));
$_SESSION['token'] = $token;
?>
<!DOCTYPE html>
<body onload="load()">
<div class="container">
<h1>CSRF Protected Form</h1>
<form
action="/chap_12_form_protected_with_token.php"
     method="post" id="csrf_test"
name="csrf_test">
<table>
<tr><th>Name</th><td><input name="name"
type="text" /></td></tr>
<tr><th>Email</th><td><input name="email"
type="text" /></td></tr>
<tr><th>Comments</th><td>
<input name="comments" type="textarea" rows=4
cols=80 />
</td></tr>
<tr><th> </th><td>
<input name="process" type="submit"
value="Process" />
</td></tr>
</table>
<input type="hidden" name="token" value="<?=
$token ?>" />
```

```
</form>
<a href="/chap_12_form_view_results.php">
    CLICK HERE</a> to view results
</div>
</body>
</html>
```

When we display and submit data from the form, the token is validated and the data insertion is allowed to continue, as shown here:



See also

For more information on CSFR attacks, please refer to https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF).

Building a secure password generator

A common misconception is that the only way attackers crack hashed passwords is by using **brute force attacks** and **rainbow tables**. Although this is often the first pass in an attack sequence, attackers will use much more sophisticated attacks on a second, third, or fourth pass. Other attacks include *combination*, *dictionary*, *mask*, and rules-based. Dictionary attacks use a database of words literally from the dictionary to guess passwords. Combination is where dictionary words are combined. Mask attacks are similar to brute force, but more selective, thus cutting down the time to crack. Rules-based attacks will detect things such as substituting the number 0 for the letter o.

The good news is that by simply increasing the length of the password beyond the magic length of six characters exponentially increases the time to crack the hashed password. Other factors, such as interspersing uppercase with lowercase letters randomly, random digits, and special characters, will also have an exponential impact on the time to crack. At the end of the day, we need to bear in mind that a human being will eventually need to enter the passwords created, which means that need to be at least marginally memorable.

Tip

Best practice

Passwords should be stored as a hash, and never as plain text. MD5 and SHA* are no longer considered secure (although SHA* is much better than MD5). Using a utility such as `oclHashcat`, an attacker can generate an average of 55 billion attempts per second on a password hashed using MD5 that has been made

available through an exploit (that is, a successful SQL injection attack).

How to do it...

1. First, we define a `Application\Security\PassGen` class that will hold the methods needed for password generation. We also define certain class constants and properties that will be used as part of the process:

```
namespace Application\Security;
class PassGen
{
  const SOURCE_SUFFIX = 'src';
  const SPECIAL_CHARS =
    '\`¬|!"£$%^&*()_-+={}[]:@~;\'#<>?,./|\\';
  protected $algorithm;
  protected $sourceList;
  protected $word;
  protected $list;
```

2. We then define low-level methods that will be used for password generation. As the names suggest, `digits()` produces random digits, and `special()` produces a single character from the `SPECIAL_CHARS` class constant:

```
public function digits($max = 999)
{
  return random_int(1, $max);
}

public function special()
{
  $maxSpecial = strlen(self::SPECIAL_CHARS) -
1;
  return self::SPECIAL_CHARS[random_int(0,
$maxSpecial)];
}
```

> ## Note
>
> Notice that we are frequently using the new PHP 7 function `random_int()` in this example. Although marginally slower, this method offers true **Cryptographically Secure Pseudo Random Number Generator** (**CSPRNG**) capabilities compared to the more dated `rand()` function.

3. Now comes the tricky part: generating a hard-to-guess word. This is where the `$wordSource` constructor parameter comes into play. It is an array of websites from which our word base will be derived. Accordingly, we need a method that will pull a unique list of words from the sources indicated, and store the results in a file. We accept the `$wordSource` array as an argument, and loop through each URL. We use `md5()` to produce a hash of the website name, which is then built into a filename. The newly produced filename is then stored in `$sourceList`:

```
public function processSource(
$wordSource, $minWordLength, $cacheDir)
{
  foreach ($wordSource as $html) {
    $hashKey = md5($html);
    $sourceFile = $cacheDir . '/' . $hashKey .
'.'
    . self::SOURCE_SUFFIX;
    $this->sourceList[] = $sourceFile;
```

4. If the file doesn't exist, or is zero-byte, we process the contents. If the source is HTML, we only accept content inside the `<body>` tag. We then use `str_word_count()` to pull a list of words out of the string, also employing `strip_tags()` to remove any markup:

```
if (!file_exists($sourceFile) ||
filesize($sourceFile) == 0) {
    echo 'Processing: ' . $html . PHP_EOL;
    $contents = file_get_contents($html);
    if (preg_match('/<body>(.*)<\/body>/i',
        $contents, $matches)) {
```

```
            $contents = $matches[1];
        }
        $list =
str_word_count(strip_tags($contents), 1);
```

5. We then remove any words that are too short, and use `array_unique()` to get rid of duplicates. The final result is stored in a file:

```
        foreach ($list as $key => $value) {
          if (strlen($value) < $minWordLength) {
            $list[$key] = 'xxxxxx';
          } else {
            $list[$key] = trim($value);
          }
        }
        $list = array_unique($list);
        file_put_contents($sourceFile,
implode("\n",$list));
    }
  }
  return TRUE;
}
```

6. Next, we define a method that *flips* random letters in the word to uppercase:

```
public function flipUpper($word)
{
  $maxLen   = strlen($word);
  $numFlips = random_int(1, $maxLen - 1);
  $flipped  = strtolower($word);
  for ($x = 0; $x < $numFlips; $x++) {
       $pos = random_int(0, $maxLen - 1);
       $word[$pos] = strtoupper($word[$pos]);
  }
  return $word;
}
```

7. Finally, we are ready to define a method that chooses a word from our source. We choose a word source at random, and use the `file()` function to read from the appropriate cached file:

```
public function word()
{
  $wsKey    = random_int(0, count($this-
```

```
>sourceList) - 1);
  $list     = file($this->sourceList[$wsKey]);
  $maxList  = count($list) - 1;
  $key      = random_int(0, $maxList);
  $word     = $list[$key];
  return $this->flipUpper($word);
}
```

8. So that we do not always produce passwords of the same pattern, we define a method that allows us to place the various components of a password in different positions in the final password string. The algorithms are defined as an array of method calls available within this class. So, for example, an algorithm of `['word', 'digits', 'word', 'special']` might end up looking like `hElLo123aUTo!`:

```
public function initAlgorithm()
{
  $this->algorithm = [
    ['word', 'digits', 'word', 'special'],
    ['digits', 'word', 'special', 'word'],
    ['word', 'word', 'special', 'digits'],
    ['special', 'word', 'special', 'digits'],
    ['word', 'special', 'digits', 'word',
'special'],
    ['special', 'word', 'special', 'digits',
    'special', 'word', 'special'],
  ];
}
```

9. The constructor accepts the word source array, minimum word length, and location of the cache directory. It then processes the source files and initializes the algorithms:

```
public function __construct(
  array $wordSource, $minWordLength,
$cacheDir)
{
  $this->processSource($wordSource,
$minWordLength, $cacheDir);
  $this->initAlgorithm();
}
```

10. Finally, we are able to define the method that actually generates the password. All it needs to do is to select an algorithm at

random, and then loop through, calling the appropriate methods:

```
    public function generate()
    {
      $pwd = '';
      $key = random_int(0, count($this->algorithm)
    - 1);
      foreach ($this->algorithm[$key] as $method)
    {
        $pwd .= $this->$method();
      }
      return str_replace("\n", '', $pwd);
    }

    }
```

How it works...

First, you will need to place the code described in the previous recipe into a file called `PassGen.php` in the `Application\Security` folder. Now you can create a calling program called `chap_12_password_generate.php` that sets up autoloading, uses `PassGen`, and defines the location of the cache directory:

```
    <?php
    define('CACHE_DIR', __DIR__ . '/cache');
    require __DIR__ .
    '/../Application/Autoload/Loader.php';
    Application\Autoload\Loader::init(__DIR__ .
    '/..');
    use Application\Security\PassGen;
```

Next, you will need to define an array of websites that will be used as a source for the word-base to be used in password generation. In this illustration, we will choose from the Project Gutenberg texts *Ulysses* (J. Joyce), *War and Peace* (L. Tolstoy), and *Pride and Prejudice* (J. Austen):

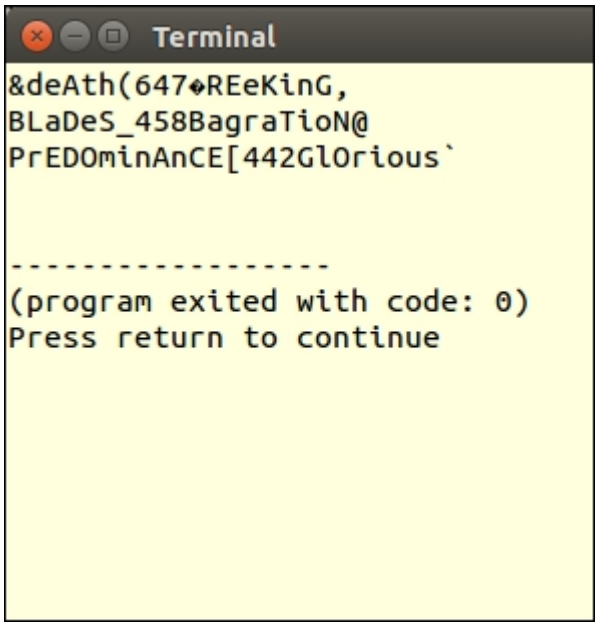
```
    $source = [
      'https://www.gutenberg.org/files/4300/4300-
    0.txt',
```

```
  'https://www.gutenberg.org/files/2600/2600-
h/2600-h.htm',
  'https://www.gutenberg.org/files/1342/1342-
h/1342-h.htm',
];
```

Next, we create the `PassGen` instance, and run `generate()`:

```
$passGen = new PassGen($source, 4, CACHE_DIR);
echo $passGen->generate();
```

Here are a few example passwords produced by `PassGen`:



See also

An excellent article on how an attacker would approach cracking a password can be viewed at http://arstechnica.com/security/2013/05/how-crackers-make-minced-meat-out-of-your-passwords/. To find out more about brute force attacks you can refer to https://www.owasp.org/index.php/Brute_force_attack. For information on `oclHashcat`, see this page: http://hashcat.net/oclhashcat/.

Safeguarding forms with a CAPTCHA

CAPTCHA is actually an acronym for **Completely Automated Public Turing Test to Tell Computers and Humans Apart**. The technique is similar to the one presented in the preceding recipe, *Securing forms with a token*. The difference is that instead of storing the token in a hidden form input field, the token is rendered into a graphic that is difficult for an automated attack system to decipher. Also, the intent of a CAPTCHA is slightly different from a form token: it is designed to confirm that the web visitor is a human being, and not an automated system.

How to do it...

1. There are several approaches to CAPTCHA: presenting a question based on knowledge only a human would possess, text tricks, and a graphics image that needs to be interpreted.
2. The image approach presents web visitors with an image with heavily distorted letters and/or numbers. This approach can be complicated, however, in that it relies on the GD extension, which may not be available on all servers. The GD extension can be difficult to compile, and has heavy dependencies on various libraries that must be present on the host server.
3. The text approach is to present a series of letters and/or numbers, and give the web visitor a simple instruction such as *please type this backwards*. Another variation is to use ASCII "art" to form characters that a human web visitor is able to interpret.
4. Finally, you might have a question/answer approach with questions such as *The head is attached to the body by what body part*, and have answers such as *Arm*, *Leg*, and *Neck*. The downside to this approach is that an automated attack system will have a 1 in 3 chance of passing the test.

Generating a text CAPTCHA

1. For this illustration, we will start with the text approach, and follow with the image approach. In either case, we first need to define a class that generates the phrase to be presented (and decoded by the web visitor). For this purpose, we define an `Application\Captcha\Phrase` class. We also define properties and class constants used in the phrase generation process:

```
namespace Application\Captcha;
class Phrase
{
  const DEFAULT_LENGTH   = 5;
  const DEFAULT_NUMBERS  = '0123456789';
  const DEFAULT_UPPER    =
'ABCDEFGHJKLMNOPQRSTUVWXYZ';
  const DEFAULT_LOWER    =
'abcdefghijklmnopqrstuvwxyz';
  const DEFAULT_SPECIAL  =
    '¬\`|!"£$%^&*()_-+={}[]:;@\'~#<,>.?/|\\';
  const DEFAULT_SUPPRESS = ['O','l'];

  protected $phrase;
  protected $includeNumbers;
  protected $includeUpper;
  protected $includeLower;
  protected $includeSpecial;
  protected $otherChars;
  protected $suppressChars;
  protected $string;
  protected $length;
```

2. The constructor, as you would expect, accepts values for the various properties, with defaults assigned so that an instance can be created without having to specify any parameters. The `$include*` flags are used to signal which character sets will be present in the base string from which the phrase will be generated. For example, if you wish to only have numbers, `$includeUpper` and `$includeLower` would both be set to `FALSE`. `$otherChars` is provided for extra flexibility. Finally, `$suppressChars` represents an array of characters that will be

removed from the base string. The default removes uppercase `O` and lowercase `l`:

```
public function __construct(
  $length = NULL,
  $includeNumbers = TRUE,
  $includeUpper= TRUE,
  $includeLower= TRUE,
  $includeSpecial = FALSE,
  $otherChars = NULL,
  array $suppressChars = NULL)
  {
    $this->length = $length ??
self::DEFAULT_LENGTH;
    $this->includeNumbers = $includeNumbers;
    $this->includeUpper = $includeUpper;
    $this->includeLower = $includeLower;
    $this->includeSpecial = $includeSpecial;
    $this->otherChars = $otherChars;
    $this->suppressChars = $suppressChars
      ?? self::DEFAULT_SUPPRESS;
    $this->phrase = $this->generatePhrase();
  }
```

3. We then define a series of getters and setters, one for each property. Please note that we only show the first two in order to conserve space.

```
public function getString()
{
  return $this->string;
}

public function setString($string)
{
  $this->string = $string;
}

// other getters and setters not shown
```

4. We next need to define a method that initializes the base string. This consists of a series of simple if statements that check the various `$include*` flags and append to the base string as appropriate. At the end, we use `str_replace()` to remove the characters represented in `$suppressChars`:

```
public function initString()
{
  $string = '';
  if ($this->includeNumbers) {
      $string .= self::DEFAULT_NUMBERS;
  }
  if ($this->includeUpper) {
      $string .= self::DEFAULT_UPPER;
  }
  if ($this->includeLower) {
      $string .= self::DEFAULT_LOWER;
  }
  if ($this->includeSpecial) {
      $string .= self::DEFAULT_SPECIAL;
  }
  if ($this->otherChars) {
      $string .= $this->otherChars;
  }
  if ($this->suppressChars) {
      $string = str_replace(
        $this->suppressChars, '', $string);
  }
  return $string;
}
```

Tip

Best practice

Get rid of letters that can be confused with numbers (that is, the letter `O` can be confused with the number `0`, and a lowercase `l` can be confused with the number `1`.

5. We are now ready to define the core method that generates the random phrase that the CAPTCHA presents to website visitors. We set up a simple `for()` loop, and use the new PHP 7 `random_int()` function to jump around in the base string:

```
public function generatePhrase()
{
```

```
    $phrase = '';
    $this->string = $this->initString();
    $max = strlen($this->string) - 1;
    for ($x = 0; $x < $this->length; $x++) {
      $phrase .= substr(
        $this->string, random_int(0, $max), 1);
    }
    return $phrase;
  }
}
```

6. Now we turn our attention away from the phrase and onto the class that will produce a text CAPTCHA. For this purpose, we first define an interface so that, in the future, we can create additional CAPTCHA classes that all make use of `Application\Captcha\Phrase`. Note that `getImage()` will return text, text art, or an actual image, depending on which class we decide to use:

```
namespace Application\Captcha;
interface CaptchaInterface
{
  public function getLabel();
  public function getImage();
  public function getPhrase();
}
```

7. For a text CAPTCHA, we define a `Application\Captcha\Reverse` class. The reason for this name is that this class produces not just text, but text in reverse. The `__construct()` method builds an instance of `Phrase`. Note that `getImage()` returns the phrase in reverse:

```
namespace Application\Captcha;
class Reverse implements CaptchaInterface
{
  const DEFAULT_LABEL = 'Type this in
reverse';
  const DEFAULT_LENGTH = 6;
  protected $phrase;
  public function __construct(
    $label  = self::DEFAULT_LABEL,
    $length = self:: DEFAULT_LENGTH,
    $includeNumbers = TRUE,
    $includeUpper   = TRUE,
```

```
      $includeLower   = TRUE,
      $includeSpecial = FALSE,
      $otherChars     = NULL,
      array $suppressChars = NULL)
    {
      $this->label  = $label;
      $this->phrase = new Phrase(
        $length,
        $includeNumbers,
        $includeUpper,
        $includeLower,
        $includeSpecial,
        $otherChars,
        $suppressChars);
      }

    public function getLabel()
    {
      return $this->label;
    }

    public function getImage()
    {
      return strrev($this->phrase->getPhrase());
    }

    public function getPhrase()
    {
      return $this->phrase->getPhrase();
    }

  }
```

Generating an image CAPTCHA

1. The image approach, as you can well imagine, is much more complicated. The phrase generation process is the same. The main difference is that not only do we need to imprint the phrase on a graphic, but we also need to distort each letter differently and introduce noise in the form of random dots.
2. We define a `Application\Captcha\Image` class that implements `CaptchaInterface`. The class constants and properties include not only those needed for phrase generation, but what is needed for image generation as well:

```
namespace Application\Captcha;
use DirectoryIterator;
class Image implements CaptchaInterface
{

  const DEFAULT_WIDTH = 200;
  const DEFAULT_HEIGHT = 50;
  const DEFAULT_LABEL = 'Enter this phrase';
  const DEFAULT_BG_COLOR = [255,255,255];
  const DEFAULT_URL = '/captcha';
  const IMAGE_PREFIX = 'CAPTCHA_';
  const IMAGE_SUFFIX = '.jpg';
  const IMAGE_EXP_TIME = 300;    // seconds
  const ERROR_REQUIRES_GD = 'Requires the GD
extension + '
    .  ' the JPEG library';
  const ERROR_IMAGE = 'Unable to generate
image';

  protected $phrase;
  protected $imageFn;
  protected $label;
  protected $imageWidth;
  protected $imageHeight;
  protected $imageRGB;
  protected $imageDir;
  protected $imageUrl;
```

3. The constructor needs to accept all the arguments required for phrase generation, as described in the previous steps. In addition, we need to accept arguments required for image generation. The two mandatory parameters are `$imageDir` and `$imageUrl`. The first is where the graphic will be written. The second is the base URL, after which we will append the generated filename. `$imageFont` is provided in case we want to provide TrueType fonts, which will produce a more secure CAPTCHA. Otherwise, we're limited to the default fonts which, to quote a line in a famous movie, *ain't a pretty sight*:

```
public function __construct(
  $imageDir,
  $imageUrl,
  $imageFont = NULL,
  $label = NULL,
  $length = NULL,
```

```
  $includeNumbers = TRUE,
  $includeUpper= TRUE,
  $includeLower= TRUE,
  $includeSpecial = FALSE,
  $otherChars = NULL,
  array $suppressChars = NULL,
  $imageWidth = NULL,
  $imageHeight = NULL,
  array $imageRGB = NULL
)
{
```

4. Next, still in the constructor, we check to see whether the `imagecreatetruecolor` function exists. If this comes back as `FALSE`, we know the GD extension is not available. Otherwise, we assign parameters to properties, generate the phrase, remove old images, and write out the CAPTCHA graphic:

```
if (!function_exists('imagecreatetruecolor'))
{
    throw new
\Exception(self::ERROR_REQUIRES_GD);
}
$this->imageDir   = $imageDir;
$this->imageUrl   = $imageUrl;
$this->imageFont  = $imageFont;
$this->label      = $label ??
self::DEFAULT_LABEL;
$this->imageRGB   = $imageRGB ??
self::DEFAULT_BG_COLOR;
$this->imageWidth = $imageWidth ??
self::DEFAULT_WIDTH;
$this->imageHeight= $imageHeight ??
self::DEFAULT_HEIGHT;
if (substr($imageUrl, -1, 1) == '/') {
    $imageUrl = substr($imageUrl, 0, -1);
}
$this->imageUrl = $imageUrl;
if (substr($imageDir, -1, 1) ==
DIRECTORY_SEPARATOR) {
    $imageDir = substr($imageDir, 0, -1);
}

$this->phrase = new Phrase(
  $length,
  $includeNumbers,
```

```
    $includeUpper,
    $includeLower,
    $includeSpecial,
    $otherChars,
    $suppressChars);
  $this->removeOldImages();
  $this->generateJpg();
  }
```

5. The process of removing old images is extremely important; otherwise we will end up with a directory filled with expired CAPTCHA images! We use the `DirectoryIterator` class to scan the designated directory and check the access time. We calculate an old image file as one that is the current time minus the value specified by `IMAGE_EXP_TIME`:

```
public function removeOldImages()
{
  $old = time() - self::IMAGE_EXP_TIME;
  foreach (new DirectoryIterator($this-
>imageDir)
            as $fileInfo) {
    if($fileInfo->isDot()) continue;
    if ($fileInfo->getATime() < $old) {
      unlink($this->imageDir .
DIRECTORY_SEPARATOR
             . $fileInfo->getFilename());
    }
  }
}
```

6. We are now ready to move on to the main show. First, we split the `$imageRGB` array into `$red`, `$green`, and `$blue`. We use the core `imagecreatetruecolor()` function to generate the base graphic with the width and height specified. We use the RGB values to colorize the background:

```
public function generateJpg()
{
  try {
      list($red,$green,$blue) = $this-
>imageRGB;
      $im = imagecreatetruecolor(
        $this->imageWidth, $this-
>imageHeight);
```

```
        $black = imagecolorallocate($im, 0, 0,
0);
        $imageBgColor = imagecolorallocate(
          $im, $red, $green, $blue);
        imagefilledrectangle($im, 0, 0, $this-
>imageWidth,
          $this->imageHeight, $imageBgColor);
```

7. Next, we define *x* and *y* margins based on image width and height. We then initialize variables to be used to write the phrase onto the graphic. We then loop a number of times that matches the length of the phrase:

```
$xMargin = (int) ($this->imageWidth * .1 +
.5);
$yMargin = (int) ($this->imageHeight * .3 +
.5);
$phrase = $this->getPhrase();
$max = strlen($phrase);
$count = 0;
$x = $xMargin;
$size = 5;
for ($i = 0; $i < $max; $i++) {
```

8. If `$imageFont` is specified, we are able to write each character with a different size and angle. We also need to adjust the *x* axis (that is, horizontal) value according to the size:

```
if ($this->imageFont) {
    $size = rand(12, 32);
    $angle = rand(0, 30);
    $y = rand($yMargin + $size, $this-
>imageHeight);
    imagettftext($im, $size, $angle, $x, $y,
$black,
      $this->imageFont, $phrase[$i]);
    $x += (int) ($size  + rand(0,5));
```

9. Otherwise, we're stuck with the default fonts. We use the largest size of `5`, as smaller sizes are unreadable. We provide a low level of distortion by alternating between `imagechar()`, which writes the image normally, and `imagecharup()`, which writes it sideways:

```
} else {
    $y = rand(0, ($this->imageHeight -
$yMargin));
    if ($count++ & 1) {
        imagechar($im, 5, $x, $y, $phrase[$i],
$black);
    } else {
        imagecharup($im, 5, $x, $y,
$phrase[$i], $black);
    }
    $x += (int) ($size * 1.2);
  }
} // end for ($i = 0; $i < $max; $i++)
```

10. Next we need to add noise in the form of random dots. This is necessary in order to make the image harder for automated systems to detect. It is also recommended that you add code to draw a few lines as well:

```
$numDots = rand(10, 999);
for ($i = 0; $i < $numDots; $i++) {
  imagesetpixel($im, rand(0, $this-
>imageWidth),
    rand(0, $this->imageHeight), $black);
}
```

11. We then create a random image filename using our old friend `md5()` with the date and a random number from `0` to `9999` as arguments. Note that we can safely use `md5()` as we are not trying to hide any secret information; we're merely interested in generating a unique filename quickly. We wipe out the image object as well to conserve memory:

```
$this->imageFn = self::IMAGE_PREFIX
. md5(date('YmdHis') . rand(0,9999))
. self::IMAGE_SUFFIX;
imagejpeg($im, $this->imageDir .
DIRECTORY_SEPARATOR
. $this->imageFn);
imagedestroy($im);
```

12. The entire construct is in a `try/catch` block. If an error or exception is thrown, we log the message and take the appropriate action:

```
} catch (\Throwable $e) {
    error_log(__METHOD__ . ':' . $e-
>getMessage());
    throw new \Exception(self::ERROR_IMAGE);
}
}
```

13. Finally, we define the methods required by the interface. Note that `getImage()` returns an HTML `<img>` tag, which can then be immediately displayed:

```
public function getLabel()
{
  return $this->label;
}

public function getImage()
{
  return sprintf('<img src="%s/%s" />',
    $this->imageUrl, $this->imageFn);
}

public function getPhrase()
{
  return $this->phrase->getPhrase();
}

}
```

How it works...

Be sure to define the classes discussed in this recipe, summarized in the following table:

Class	Subsection	The steps it appears in

Class	Subsection	The steps it appears in
`Application\Captcha\Phrase`	Generating a text CAPTCHA	1 - 5
`Application\Captcha\CaptchaInterface`		6
`Application\Captcha\Reverse`		7
`Application\Captcha\Image`	Generating an image CAPTCHA	2 - 13

Next, define a calling program called `chap_12_captcha_text.php` that implements a text CAPTCHA. You first need to set up autoloading and use the appropriate classes:

```
<?php
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
use Application\Captcha\Reverse;
```

After that, be sure to start the session. You would use appropriate measures to protect the session as well. To conserve space, we only show one simple measure, `session_regenerate_id()`:

```
session_start();
session_regenerate_id();
```

Next, you can define a function that creates the CAPTCHA; retrieves the phrase, label, and image (in this case, reverse text); and stores the value in the session:

```
function setCaptcha(&$phrase, &$label, &$image)
{
  $captcha = new Reverse();
  $phrase  = $captcha->getPhrase();
  $label   = $captcha->getLabel();
  $image   = $captcha->getImage();
  $_SESSION['phrase'] = $phrase;
}
```

Now is a good time to initialize variables and determine the `loggedIn` status:

```
$image      = '';
$label      = '';
$phrase     = $_SESSION['phrase'] ?? '';
$message    = '';
$info       = 'You Can Now See Super Secret
Information!!!';
$loggedIn   = $_SESSION['isLoggedIn'] ?? FALSE;
$loggedUser = $_SESSION['user'] ?? 'guest';
```

You can then check to see whether the login button has been pressed. If so, check to see whether the CAPTCHA phrase has been entered. If not, initialize a message informing the user they need to enter the CAPTCHA phrase:

```
if (!empty($_POST['login'])) {
  if (empty($_POST['captcha'])) {
    $message = 'Enter Captcha Phrase and Login
Information';
```

If the CAPTCHA phrase is present, check to see whether it matches what is stored in the session. If it doesn't match, proceed as if the

form is invalid. Otherwise, process the login as you would have otherwise. For the purposes of this illustration, you can simulate a login by using hard-coded values for the username and password:

```
} else {
    if ($_POST['captcha'] == $phrase) {
        $username = 'test';
        $password = 'password';
        if ($_POST['user'] == $username
            && $_POST['pass'] == $password) {
            $loggedIn = TRUE;
            $_SESSION['user'] =
strip_tags($username);
            $_SESSION['isLoggedIn'] = TRUE;
        } else {
            $message = 'Invalid Login';
        }
    } else {
        $message = 'Invalid Captcha';
    }
}
```

You might also want to add code for a logout option, as described in the *Safeguarding the PHP session* recipe:

```
} elseif (isset($_POST['logout'])) {
  session_unset();
  session_destroy();
  setcookie('PHPSESSID', 0, time() - 3600);
  header('Location: ' . $_SERVER['REQUEST_URI'] );
  exit;
}
```

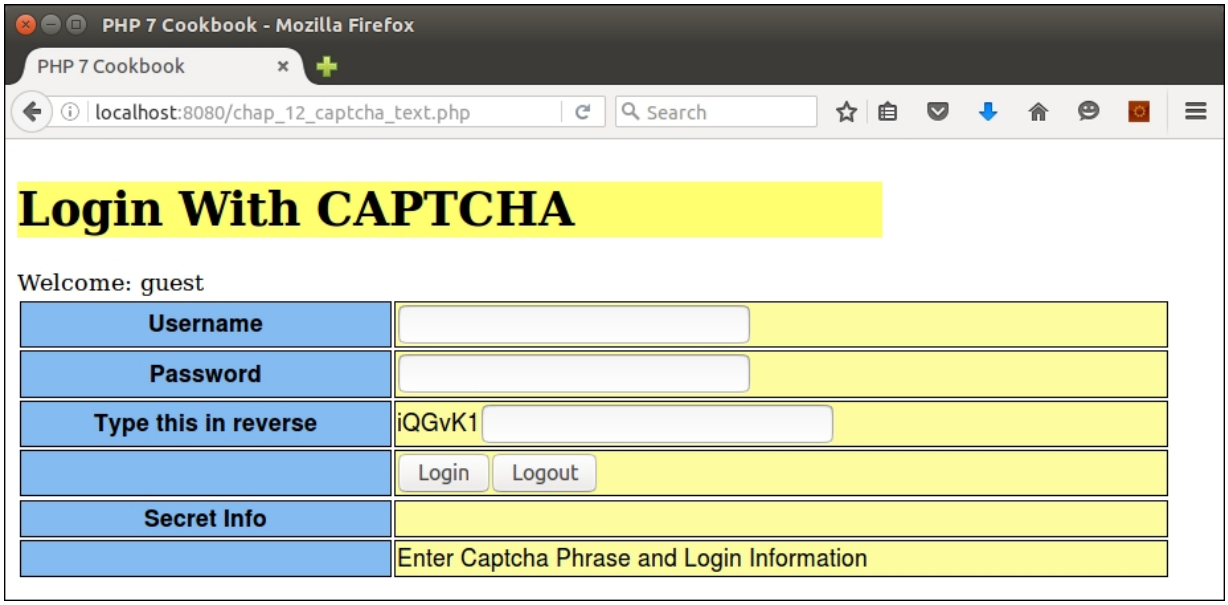
You can then run `setCaptcha()`:

```
setCaptcha($phrase, $label, $image);
```

Lastly, don't forget the view logic, which, in this example, presents a basic login form. Inside the form tag, you'll need to add view logic to display the CAPTCHA and label:

```
<tr>
  <th><?= $label; ?></th>
  <td><?= $image; ?><input type="text"
name="captcha" /></td>
</tr>
```

Here is the resulting output:



To demonstrate how to use the image CAPTCHA, copy the code from `chap_12_captcha_text.php` to `cha_12_captcha_image.php`. We define constants that represent the location of the directory in which we will write the CAPTCHA images. (Be sure to create this directory!) Otherwise, the autoloading and use statement structure is similar. Note that we also define a TrueType font. Differences are noted in **bold**:

```
<?php
define('IMAGE_DIR', __DIR__ . '/captcha');
define('IMAGE_URL', '/captcha');
define('IMAGE_FONT', __DIR__ .
'/FreeSansBold.ttf');
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
use Application\Captcha\Image;
```

```
session_start();
session_regenerate_id();
```

Tip

Important!

Fonts can potentially be protected under copyright, trademark, patent, or other intellectual property laws. If you use a font for which you are not licensed, you and your customer could be held liable in court! Use an open source font, or one that is available on the web server for which you have a valid license.

Of course, in the `setCaptcha()` function, we use the `Image` class instead of `Reverse`:

```
function setCaptcha(&$phrase, &$label, &$image)
{
  $captcha = new Image(IMAGE_DIR, IMAGE_URL,
IMAGE_FONT);
  $phrase  = $captcha->getPhrase();
  $label   = $captcha->getLabel();
  $image   = $captcha->getImage();
  $_SESSION['phrase'] = $phrase;
  return $captcha;
}
```

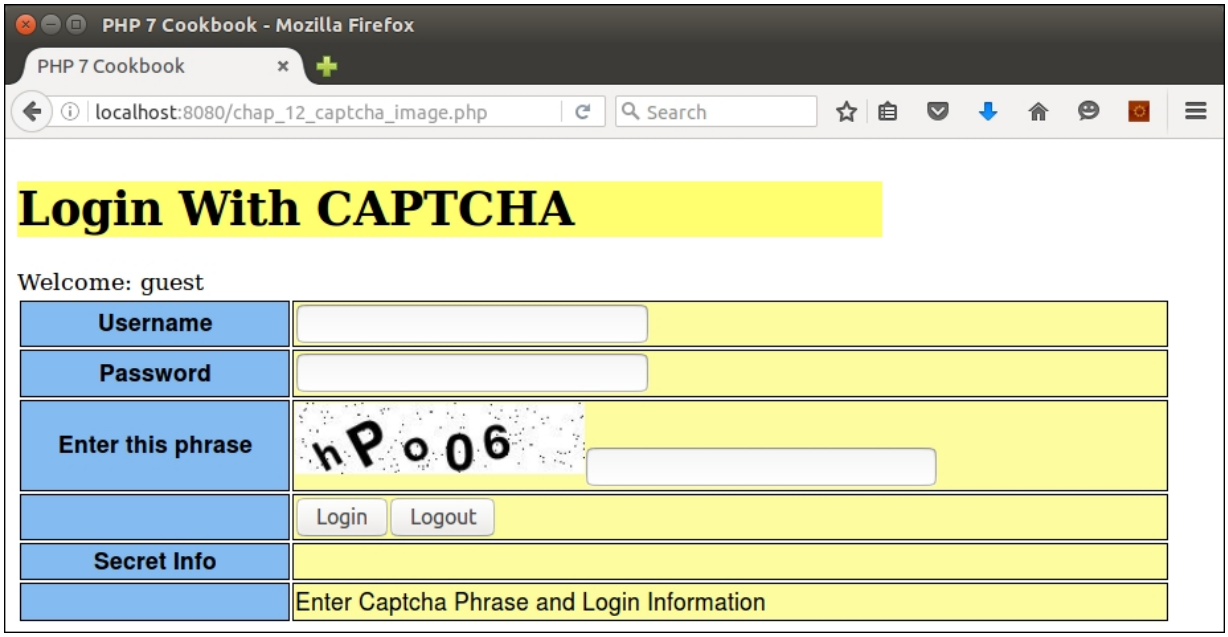
Variable initialization is the same as the previous script, and login processing is identical to the previous script:

```
$image      = '';
$label      = '';
$phrase     = $_SESSION['phrase'] ?? '';
$message    = '';
$info       = 'You Can Now See Super Secret
Information!!!';
$loggedIn   = $_SESSION['isLoggedIn'] ?? FALSE;
$loggedUser = $_SESSION['user'] ?? 'guest';
```

```
if (!empty($_POST['login'])) {

  // etc.  -- identical to
chap_12_captcha_text.php
```

Even the view logic remains the same, as we are using `getImage()`, which, in the case of the image CAPTCHA, returns directly usable HTML. Here is the output using a TrueType font:



There's more...

If you are not inclined to use the preceding code to generate your own in-house CAPTCHA, there are plenty of libraries available. Most popular frameworks have this ability. Zend Framework, for example, has its Zend\Captcha component class. There is also reCAPTCHA, which is generally invoked as a service in which your application makes a call to an external website that generates the CAPTCHA and token for you. A good place to start looking is http://www.captcha.net/ website.

See also

For more information on the protection of fonts as intellectual property, refer to the article present at https://en.wikipedia.org/wiki/Intellectual_property_protection_of_typefaces.

Encrypting/decrypting without mcrypt

It is a little-known fact among members of the general PHP community that the `mcrypt` extension, the core of most PHP-based encryption considered secure, is anything but secure. One of the biggest issues, from a security perspective, is that the `mcrypt` extension requires advanced knowledge of cryptography to successfully operate, which few programmers have. This leads to gross misuse and ultimately problems such as a 1 in 256 chance of data corruption. Not good odds. Furthermore, developer support for `libmcrypt`, the core library upon which the `mcrypt` extension is based, was *abandoned* in 2007, which means the code base is out-of-date, bug-ridden, and has no mechanism to apply patches. Accordingly, it is extremely important to understand how to perform strong encryption/decryption *without* using `mcrypt`!

How to do it...

1. The solution to the problem posed previously, in case you're wondering, is to use `openssl`. This extension is well maintained, and has modern and very strong encryption/decryption capabilities.

> ### Tip
>
> **Important**
>
> In order to use any `openssl*` functions, the `openssl` PHP extension must be compiled and enabled! In addition, you will need to install the latest OpenSSL package on your web server.

2. First, you will need to determine which cipher methods are available on your installation. For this purpose, you can use the `openssl_get_cipher_methods()` command. Examples will include algorithms based on **Advanced Encryption Standard** (**AES**), **BlowFish** (**BF**), **CAMELLIA**, **CAST5**, **Data Encryption Standard** (**DES**), **Rivest Cipher** (**RC**) (also affectionately known as **Ron's Code**), and **SEED**. You will note that this method shows cipher methods duplicated in upper and lowercase.
3. Next, you will need to figure out which method is most appropriate for your needs. Here is a table that gives a quick summary of the various methods:

Method	Published	Key size (bits)	Key block size (bytes)	Notes
`camellia`	2000	128, 192, 256	16	Developed by Mitsubishi and NTT
`aes`	1998	128, 192, 256	16	Developed by Joan Daemen and Vincent Rijmen. Originally submitted as Rijndael
`seed`	1998	128	16	Developed by the Korea Information Security Agency

Method	Published	Key size (bits)	Key block size (bytes)	Notes
cast5	1996	40 to 128	8	Developed by Carlisle Adams and Stafford Tavares
bf	1993	1 to 448	8	Designed by Bruce Schneier
rc2	1987	8 to 1,024 defaults to 64	8	Designed by Ron Rivest (one of the core founders of RSA)
des	1977	56 (+8 parity bits)	8	Developed by IBM, based on work done by Horst Feistel

4. Another consideration is what your preferred block cipher **mode of operation is**. Common choices are summarized in this table:

Mode	Stands For	Notes

Mode	Stands For	Notes
ECB	Electronic Code Book	Does not require **initialization vector** (**IV**); supports parallelization for both encryption and decryption; simple and fast; does not hide data patterns; not recommended!!!
CBC	Cipher Block Chaining	Requires IV; subsequent blocks, even if identical, are XOR'ed with previous block, resulting in better overall encryption; if the IVs are predictable, the first block can be decoded, leaving remaining message exposed; message must be padded to a multiple of the cipher block size; supports parallelization only for decryption
CFB	Cipher Feedback	Close relative of CBC, except that encryption is performed in reverse
OFB	Output Feedback	Very symmetrical: encrypt and decrypt are the same; does not supports parallelization at all
CTR	Counter	Similar in operation to OFB; supports parallelization for both encryption and decryption

Mode	Stands For	Notes
CCM	Counter with CBC-MAC	Derivative of CTR; only designed for block length of 128 bits; provides authentication and confidentiality; **CBC-MAC** stands for **Cipher Block Chaining - Message Authentication Code**
GCM	Galois/Counter Mode	Based on CTR mode; should use a different IV for each stream to be encrypted; exceptionally high throughput (compared to other modes); supports parallelization for both encryption and decryption
XTS	XEX-based Tweaked-codebook mode with ciphertext Stealing	Relatively new (2010) and fast; uses two keys; increases the amount of data that can be securely encrypted as one block

5. Before choosing a cipher method and mode, you will also need to determine whether the encrypted contents needs to be unencrypted outside of your PHP application. For example, if you are storing database credentials encrypted into a standalone text file, do you need to have the ability to decrypt from the command line? If so, make sure that the cipher method and operation mode you choose are supported by the target operating system.

6. The number of bytes supplied for the **IV** varies according to the cipher method chosen. For best results, use `random_bytes()` (new in PHP 7), which returns a true **CSPRNG** sequence of bytes. The length of the IV varies considerably. Try a size of 16 to start with. If a *warning* is generated, the correct number of bytes to be supplied for that algorithm will be shown, so adjust the size accordingly:

```
$iv  = random_bytes(16);
```

7. To perform encryption, use `openssl_encrypt()`. Here are the parameters that should be passed:

Parameter	**Notes**
Data	Plain text you need to encrypt.
Method	One of the methods you identified using `openssl_get_cipher_methods()`. identified as follows: *method - key_size - cipher_mode* So, for example, if you want a method of AES, a key size of 256, and GCM mode, you would enter `aes-256-gcm`.
Password	Although documented as *password*, this parameter can be viewed as a *key*. Use `random_bytes()` to generate a key with a number of bytes to match the desired key size.

Parameter	Notes
Options	Until you gain more experience with `openssl` encryption, it is recommended you stick with the default value of `0`.
IV	Use `random_bytes()` to generate an IV with a number of bytes to match the cipher method.

8. As an example, suppose you wanted to choose the AES cipher method, a key size of 256, and XTS mode. Here is the code used to encrypt:

```
$plainText = 'Super Secret Credentials';
$key = random_bytes(16);
$method = 'aes-256-xts';
$cipherText = openssl_encrypt($plainText,
$method, $key, 0, $iv);
```

9. To decrypt, use the same values for `$key` and `$iv`, along with the `openssl_decrypt()` function:

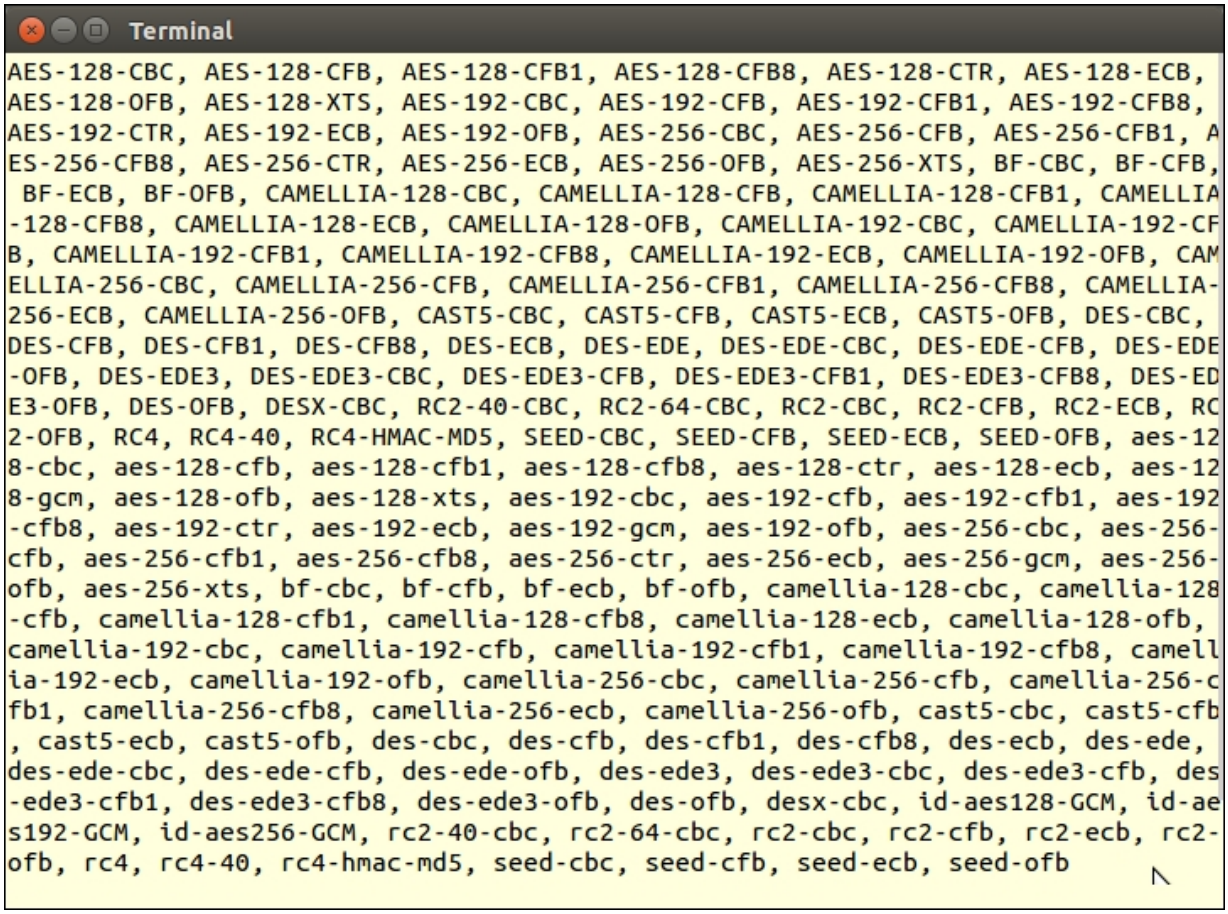
```
$plainText = openssl_decrypt($cipherText,
$method, $key, 0, $iv);
```

How it works...

In order to see which cipher methods are available, create a PHP script called `chap_12_openssl_encryption.php` and run this command:

```
<?php
echo implode(', ', openssl_get_cipher_methods());
```

The output should look something like this:



Next, you can add values for the plain text to be encrypted, the method, key, and IV. As an example, try AES, with a key size of 256, using the XTS operating mode:

```
$plainText = 'Super Secret Credentials';
$method = 'aes-256-xts';
$key = random_bytes(16);
$iv  = random_bytes(16);
```

To encrypt, you can use `openssl_encrypt()`, specifying the parameters configured previously:

```
$cipherText = openssl_encrypt($plainText, $method,
$key, 0, $iv);
```

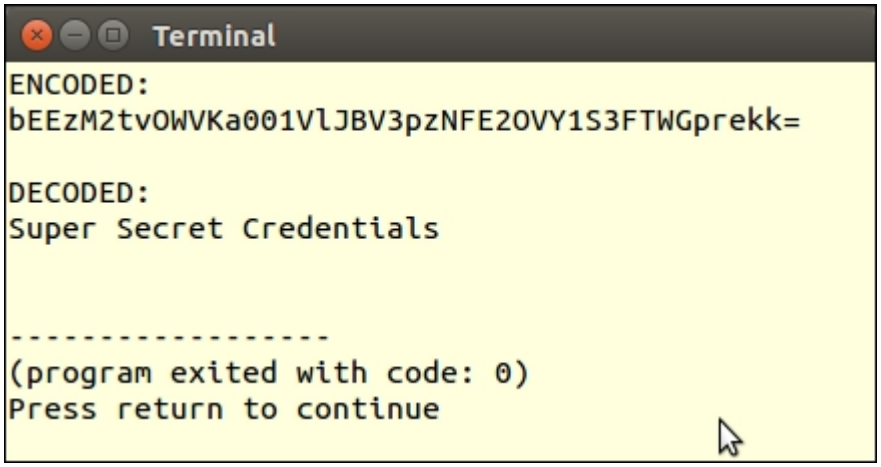
You might also want to base 64-encode the result to make it more usable:

```
$cipherText = base64_encode($cipherText);
```

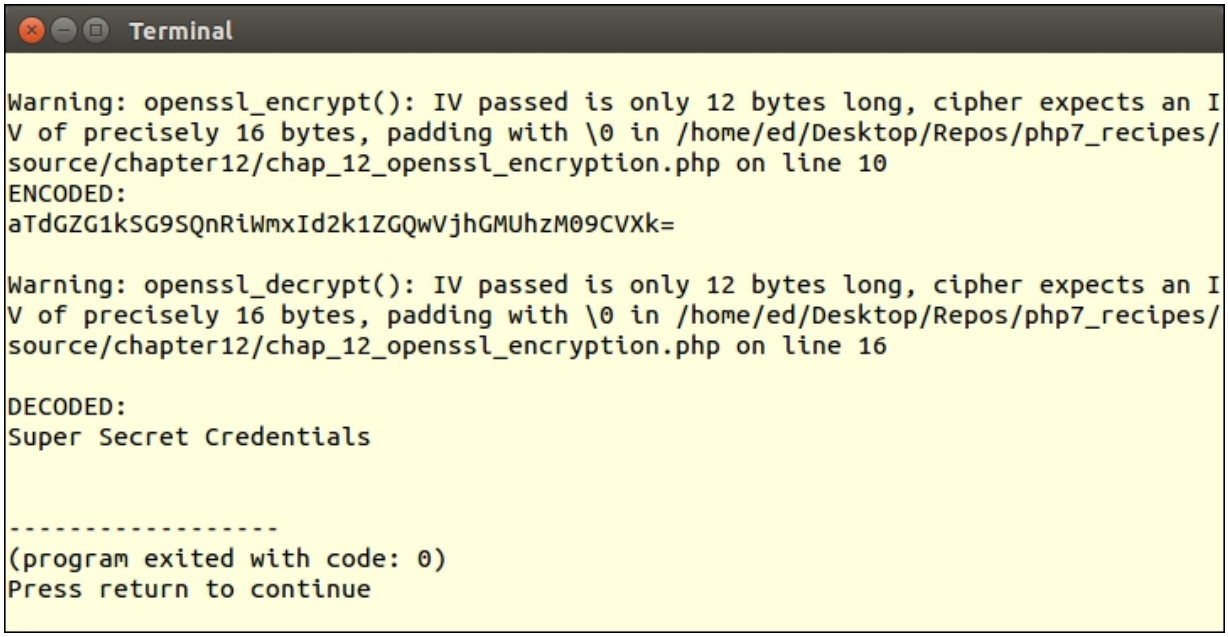
To decrypt, use the same `$key` and `$iv` values. Don't forget to un-encode the base 64 value first:

```
$plainText =
openssl_decrypt(base64_decode($cipherText),
$method, $key, 0, $iv);
```

Here is the output showing the base 64-encoded cipher text, followed by the decrypted plain text:



If you supply an incorrect number of bytes for the IV, for the cipher method chosen, a warning message will be shown:



There's more...

In PHP 7, there was a problem when using `open_ssl_encrypt()` and `open_ssl_decrypt()` and the **Authenticated Encrypt with Associated Data** (**AEAD**) modes supported: GCM and CCM. Accordingly, in PHP 7.1, three extra parameters have been added to these functions, as follows:

Parameter	Description
`$tag`	Authentication tag passed by reference; variable value remains the same if authentication fails
`$aad`	Additional authentication data

Parameter	Description
`$tag_length`	4 to 16 for GCM mode; no limits for CCM mode; only for `open_ssl_encrypt()`

For more information, you can refer to https://wiki.php.net/rfc/openssl_aead.

See also

For an excellent discussion on why the `mcrypt` extension is being deprecated in PHP 7.1, please refer to the article at https://wiki.php.net/rfc/mcrypt-viking-funeral. For a good description of block cipher, which forms the basis for the various cipher methods, refer to the article present at https://en.wikipedia.org/wiki/Block_cipher. For an excellent description of AES, refer to https://en.wikipedia.org/wiki/Advanced_Encryption_Standard. A good article that describes encryption operation modes can be seen at https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation.

Note

For some of the newer modes, if the data to be encrypted is less than the block size, `openssl_decrypt()` will return no value. If you *pad* the data to be at least the block size, the problem goes away. Most of the modes implement internal padding so this is not an issue. With some of the newer modes (that is, `xts`) you might see this problem. Be sure to conduct tests on short strings of data less than eight characters before putting your code into production.

Chapter 13. Best Practices, Testing, and Debugging

In this chapter, we will cover the following topics:

- Using Traits and Interfaces
- Universal exception handler
- Universal error handler
- Writing a simple test
- Writing a test suite
- Generating fake test data
- Customizing sessions using `session_start` parameters

Introduction

In this chapter, we will show you how traits and interfaces work together. Then, we turn our attention to the design of a fallback mechanism that will catch errors and exceptions in situations where you were not able (or forgot) to define specific `try/catch` blocks. We will then venture into the world of unit testing, showing you first how to write simple tests, and then how to group those tests together into test suites. Next, we define a class that lets you create any amount of generic test data. We close the chapter with a discussion of how to easily manage sessions using new PHP 7 features.

Using Traits and Interfaces

It is considered a best practice to make use of interfaces as a means of establishing the classification of a set of classes, and to guarantee the existence of certain methods. Traits and Interfaces often work together, and are an important aspect of implementation. Wherever you have a frequently used Interface that defines a method where the code does not change (such as a setter or getter), it is useful to also define a Trait that contains the actual code implementation.

How to do it...

1. For this example, we will use `ConnectionAwareInterface`, first presented in Chapter 4, *Working with PHP Object-Oriented Programming*. This interface defines a `setConnection()` method that sets a `$connection` property. Two classes in the `Application\Generic` namespace, `CountryList` and `CustomerList`, contain redundant code, which matches the method defined in the interface.
2. Here is what `CountryList` looks like before the change:

```
class CountryList
{
  protected $connection;
  protected $key   = 'iso3';
  protected $value = 'name';
  protected $table = 'iso_country_codes';

  public function setConnection(Connection
$connection)
  {
    $this->connection = $connection;
  }
  public function list()
  {
    $list = [];
    $sql  = sprintf('SELECT %s,%s FROM %s',
$this->key,
                    $this->value, $this-
>table);
```

```
        $stmt = $this->connection->pdo-
>query($sql);
        while ($item = $stmt-
>fetch(PDO::FETCH_ASSOC)) {
          $list[$item[$this->key]] =  $item[$this-
>value];
        }
        return $list;
      }

}
```

3. We will now move `list()` into a trait called `ListTrait`:

```
trait ListTrait
{
  public function list()
  {
    $list = [];
    $sql  = sprintf('SELECT %s,%s FROM %s',
                    $this->key, $this->value,
$this->table);
    $stmt = $this->connection->pdo-
>query($sql);
    while ($item = $stmt-
>fetch(PDO::FETCH_ASSOC)) {
           $list[$item[$this->key]] =
$item[$this->value];
    }
    return $list;
  }
}
```

4. We can then insert the code from `ListTrait` into a new class, `CountryListUsingTrait`, as shown next:

```
class CountryListUsingTrait
{
  use ListTrait;
  protected $connection;
  protected $key   = 'iso3';
  protected $value = 'name';
  protected $table = 'iso_country_codes';
  public function setConnection(Connection
$connection)
  {
    $this->connection = $connection;
```

```
  }

}
```

5. Next, we observe that many classes need to set a connection instance. Again, this calls for a trait. This time, however, we place the trait in the `Application\Database` namespace. Here is the new trait:

```
namespace Application\Database;
trait ConnectionTrait
{
  protected $connection;
  public function setConnection(Connection
$connection)
  {
    $this->connection = $connection;
  }
}
```

6. Traits are often used to avoid duplication of code. It is often the case that you also need to identify the class that uses the trait. A good way to do this is to develop an interface that matches the trait. In this example, we will define `Application\Database\ConnectionAwareInterface`:

```
namespace Application\Database;
use Application\Database\Connection;
interface ConnectionAwareInterface
{
  public function setConnection(Connection
$connection);
}
```

7. And here is the revised `CountryListUsingTrait` class. Note that as the new trait is affected by its location in the namespace, we needed to add a `use` statement at the top of the class. You will also note that we implement `ConnectionAwareInterface` to identify the fact that this class requires the method defined in the trait. Notice that we are taking advantage of the new PHP 7 group use syntax:

```
namespace Application\Generic;
use PDO;
```

```
use Application\Database\ {
Connection, ConnectionTrait,
ConnectionAwareInterface
};
class CountryListUsingTrait implements
ConnectionAwareInterface
{
  use ListTrait;
  use ConnectionTrait;

  protected $key   = 'iso3';
  protected $value = 'name';
  protected $table = 'iso_country_codes';

}
```

How it works...

First of all, make sure the classes developed in Chapter 4, *Working with PHP Object-Oriented Programming*, have been created. These include the `Application\Generic\CountryList` and `Application\Generic\CustomerList` classes discussed in Chapter 4, *Working with PHP Object-Oriented Programming*, in the recipe *Using interfaces*. Save each class in a new file in the `Application\Generic` folder as `CountryListUsingTrait.php` and `CustomerListUsingTrait.php`. Be sure to change the class names to match the new names of the files!

As discussed in step 3, remove the `list()` method from both `CountryListUsingTrait.php` and `CustomerListUsingTrait.php`. Add `use ListTrait;` in place of the method removed. Place the removed code into a separate file, in the same folder, called `ListTrait.php`.

You will also notice further duplication of code between the two list classes, in this case the `setConnection()` method. This calls for another trait!

Cut the `setConnection()` method out of both `CountryListUsingTrait.php` and `CustomerListUsingTrait.php` list classes, and place the removed code into a separate file called

`ConnectionTrait.php`. As this trait is logically related to `ConnectionAwareInterface` and `Connection`, it makes sense to place the file in the `Application\Database` folder, and to specify its namespace accordingly.

Finally, define `Application\Database\ConnectionAwareInterface` as discussed in step 6. Here is the final `Application\Generic\CustomerListUsingTrait` class after all changes:

```
<?php
namespace Application\Generic;
use PDO;
use Application\Database\Connection;
use Application\Database\ConnectionTrait;
use Application\Database\ConnectionAwareInterface;
class CustomerListUsingTrait implements
ConnectionAwareInterface
{

  use ListTrait;
  use ConnectionTrait;

  protected $key   = 'id';
  protected $value = 'name';
  protected $table = 'customer';
}
```

You can now copy the `chap_04_oop_simple_interfaces_example.php` file mentioned in Chapter 4, *Working with PHP Object-Oriented Programming*, to a new file called `chap_13_trait_and_interface.php`. Change the reference from `CountryList` to `CountryListUsingTrait`. Likewise, change the reference from `CustomerList` to `CustomerListUsingTrait`. Otherwise, the code can remain the same:

```
<?php
define('DB_CONFIG_FILE',
'/../config/db.config.php');
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
```

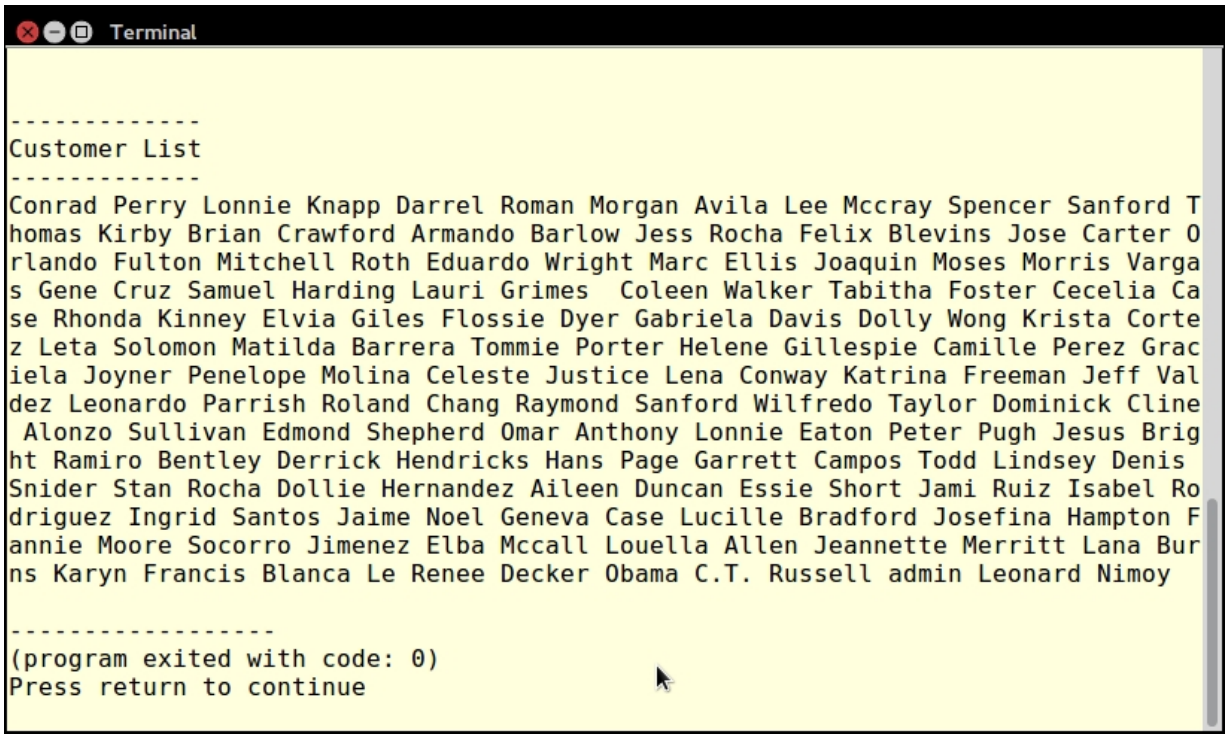
```
$params = include __DIR__ . DB_CONFIG_FILE;
try {
    $list =
Application\Generic\ListFactory::factory(
      new
Application\Generic\CountryListUsingTrait(),
$params);
    echo 'Country List' . PHP_EOL;
    foreach ($list->list() as $item) echo $item .
' ';
    $list =
Application\Generic\ListFactory::factory(
      new
Application\Generic\CustomerListUsingTrait(),
      $params);
    echo 'Customer List' . PHP_EOL;
    foreach ($list->list() as $item) echo $item .
' ';

} catch (Throwable $e) {
    echo $e->getMessage();
}
```

The output will be exactly as described in the *Using interfaces* recipe of Chapter 4, *Working with Object-Oriented Programming*. You can see the country list portion of the output in the following screenshot:



The next image displays the customer list portion of the output:



Universal exception handler

Exceptions are especially useful when used in conjunction with code in a `try/catch` block. Using this construct, however, can be awkward in some situations, making code virtually unreadable. Another consideration is that many classes end up throwing exceptions that you have not anticipated. In such cases, it would be highly desirable to have some sort of fallback exception handler.

How to do it...

1. First, we define a generic exception handling class, `Application\Error\Handler`:

```
namespace Application\Error;
class Handler
{
  // code goes here
}
```

2. We define properties that represents a log file. If the name is not supplied, it is named after the year, month, and day. In the constructor, we use `set_exception_handler()` to assign the `exceptionHandler()` method (in this class) as the fallback handler:

```
protected $logFile;
public function __construct(
  $logFileDir = NULL, $logFile = NULL)
{
  $logFile = $logFile    ?? date('Ymd') .
'.log';
  $logFileDir = $logFileDir ?? __DIR__;
  $this->logFile = $logFileDir . '/' .
$logFile;
  $this->logFile = str_replace('//', '/',
$this->logFile);

set_exception_handler([$this,'exceptionHandler
']);
}
```

3. Next, we define the `exceptionHandler()` method, which takes an `Exception` object as an argument. We record the date and time, the class name of the exception, and its message in the log file:

```
public function exceptionHandler($ex)
{
  $message = sprintf('%19s : %20s : %s' .
PHP_EOL,
    date('Y-m-d H:i:s'), get_class($ex), $ex-
>getMessage());
  file_put_contents($this->logFile, $message,
FILE_APPEND);
}
```

4. If we specifically put a `try/catch` block in our code, this will override our universal exception handler. If, on the other hand, we do not use try/catch and an exception is thrown, the universal exception handler will come into play.

> ## Tip
>
> **Best practice**
>
> You should always use try/catch to trap exceptions and possibly continue in your application. The exception handler described here is only designed to allow your application to end "gracefully" in situations where exceptions thrown have not been caught.

How it works...

First, place the code shown in the preceding recipe into a `Handler.php` file in the `Application\Error` folder. Next, define a test class that will throw an exception. For the purposes of illustration, create an `Application\Error\ThrowsException` class that will throw an exception. As an example, set up a PDO instance with the error

mode set to `PDO::ERRMODE_EXCEPTION`. You then craft an SQL statement that is guaranteed to fail:

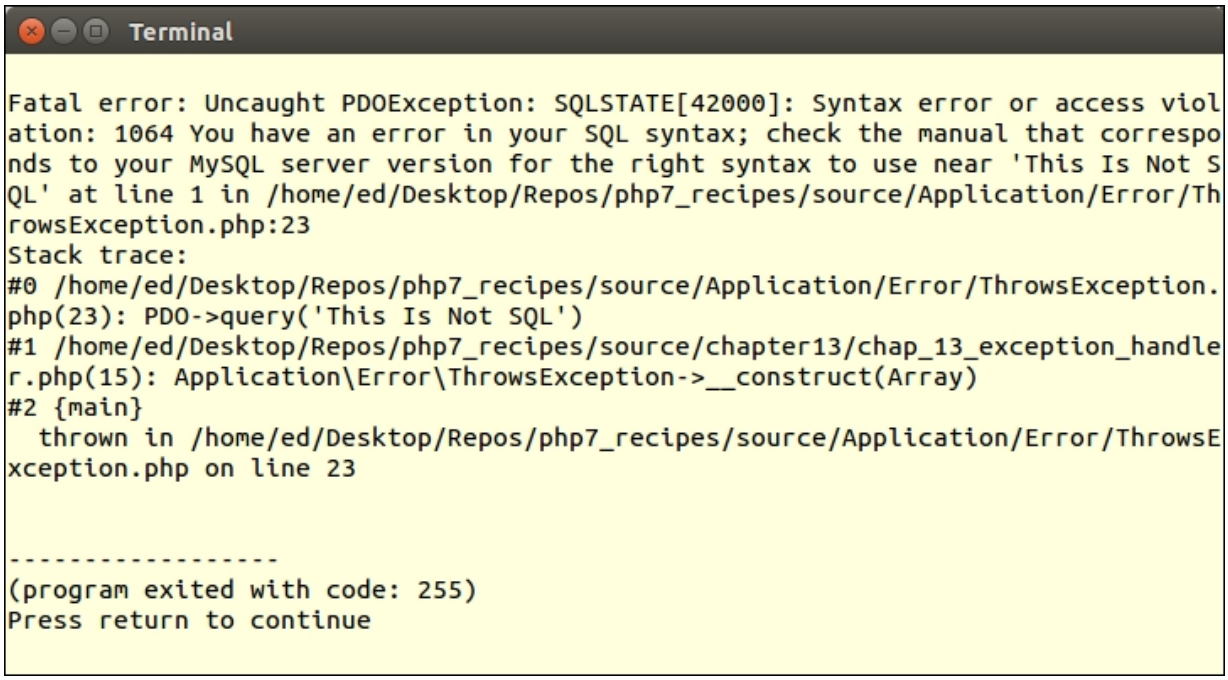
```
namespace Application\Error;
use PDO;
class ThrowsException
{
  protected $result;
  public function __construct(array $config)
  {
    $dsn = $config['driver'] . ':';
    unset($config['driver']);
    foreach ($config as $key => $value) {
      $dsn .= $key . '=' . $value . ';';
    }
    $pdo = new PDO(
      $dsn,
      $config['user'],
      $config['password'],
      [PDO::ATTR_ERRMODE =>
PDO::ERRMODE_EXCEPTION]);
      $stmt = $pdo->query('This Is Not SQL');
      while ($row = $stmt-
>fetch(PDO::FETCH_ASSOC)) {
        $this->result[] = $row;
      }
  }
}
```

Next, define a calling program called `chap_13_exception_handler.php` that sets up autoloading, uses the appropriate classes:

```
<?php
define('DB_CONFIG_FILE', __DIR__ .
'/../config/db.config.php');
$config = include DB_CONFIG_FILE;
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
use Application\Error\ { Handler, ThrowsException
};
```

At this point, if you create a `ThrowsException` instance without implementing the universal handler, a `Fatal Error` is generated as an exception has been thrown but not caught:

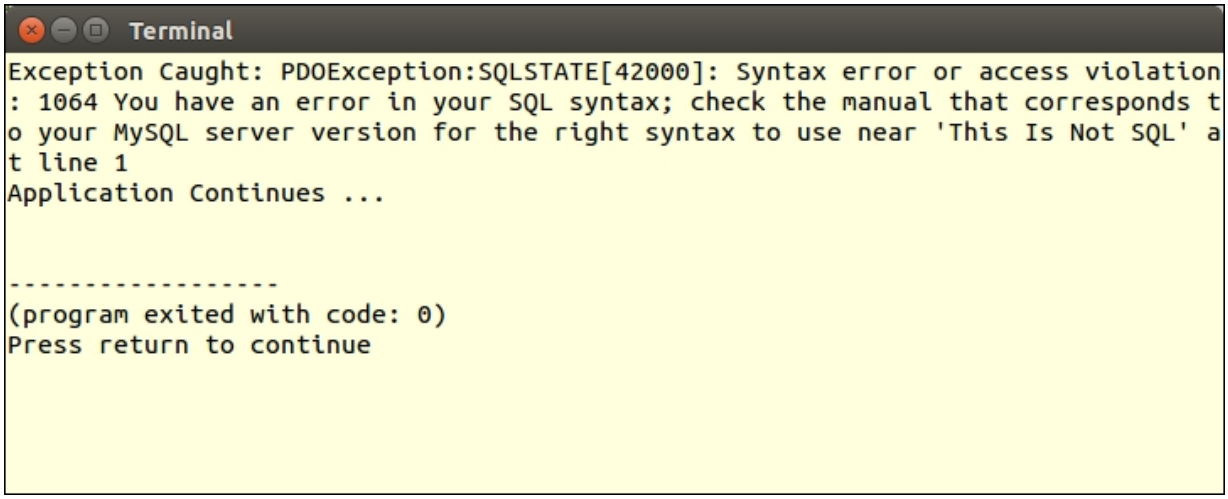
```
$throws1 = new ThrowsException($config);
```



If, on the other hand, you use a `try/catch` block, the exception will be caught and your application is allowed to continue, if it is stable enough:

```
try {
    $throws1 = new ThrowsException($config);
} catch (Exception $e) {
    echo 'Exception Caught: ' . get_class($e) .
':' . $e->getMessage() . PHP_EOL;
}
echo 'Application Continues ...' . PHP_EOL;
```

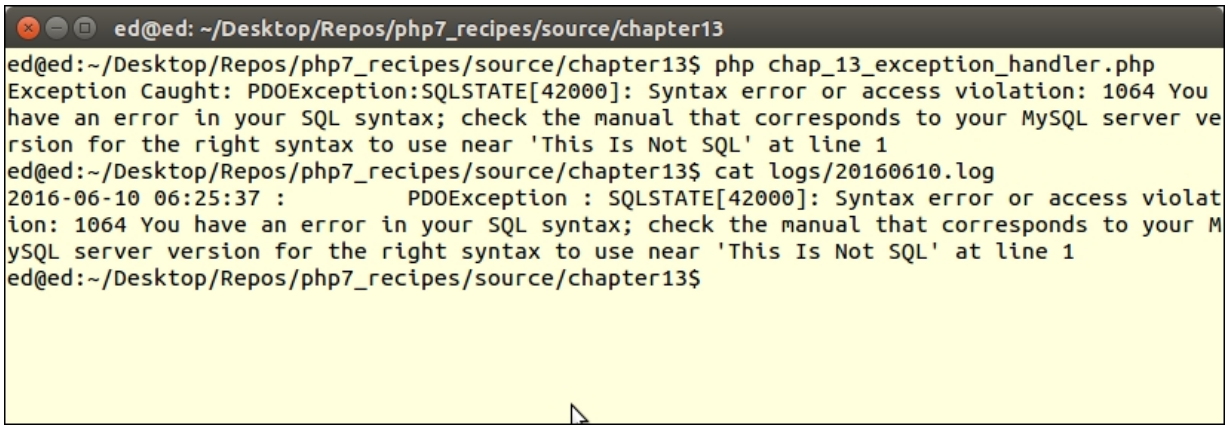
You will observe the following output:



To demonstrate use of the exception handler, define a `Handler` instance, passing a parameter that represents the directory to contain log files, before the `try/catch` block. After `try/catch`, outside the block, create another instance of `ThrowsException`. When you run this sample program, you will notice that the first exception is caught inside the `try/catch` block, and the second exception is caught by the handler. You will also note that after the handler, the application ends:

```
$handler = new Handler(__DIR__ . '/logs');
try {
    $throws1 = new ThrowsException($config);
} catch (Exception $e) {
    echo 'Exception Caught: ' . get_class($e) .
':'
      . $e->getMessage() . PHP_EOL;
}
$throws1 = new ThrowsException($config);
echo 'Application Continues ...' . PHP_EOL;
```

Here is the output from the completed example program, along with the contents of the log file:



See also

- It might be a good idea to review the documentation on the `set_exception_handler()` function. Have a look, especially, at the comment (posted 7 years ago, but still pertinent) by Anonymous that clarifies how this function works: http://php.net/manual/en/function.set-exception-handler.php.

Universal error handler

The process of developing a universal error handler is quite similar to the preceding recipe. There are certain differences, however. First of all, in PHP 7, some errors are thrown and can be caught, whereas others simply stop your application dead in its tracks. To further confuse matters, some errors are treated like exceptions, whereas others are derived from the new PHP 7 `Error` class. Fortunately for us, in PHP 7, both `Error` and `Exception` implement a new interface called `Throwable`. Accordingly, if you are not sure whether your code will throw an `Exception` or an `Error`, simply catch an instance of `Throwable` and you'll catch both.

How to do it...

1. Modify the `Application\Error\Handler` class defined in the preceding recipe. In the constructor, set a new `errorHandler()` method as the default error handler:

```
public function __construct($logFileDir =
NULL, $logFile = NULL)
{
  $logFile    = $logFile    ?? date('Ymd') .
'.log';
  $logFileDir = $logFileDir ?? __DIR__;
  $this->logFile = $logFileDir . '/' .
$logFile;
  $this->logFile = str_replace('//', '/',
$this->logFile);

set_exception_handler([$this,'exceptionHandler
']);
  set_error_handler([$this, 'errorHandler']);
}
```

2. We then define the new method, using the documented parameters. As with our exception handler, we log information to a log file:

```
public function errorHandler($errno, $errstr,
$errfile, $errline)
{
  $message = sprintf('ERROR: %s : %d : %s : %s
: %s' . PHP_EOL,
    date('Y-m-d H:i:s'), $errno, $errstr,
$errfile, $errline);
  file_put_contents($this->logFile, $message,
FILE_APPEND);
}
```

3. Also, just to be able to distinguish errors from exceptions, add `EXCEPTION` to the message sent to the log file in the `exceptionHandler()` method:

```
public function exceptionHandler($ex)
{
  $message = sprintf('EXCEPTION: %19s : %20s :
%s' . PHP_EOL,
    date('Y-m-d H:i:s'), get_class($ex), $ex-
>getMessage());
  file_put_contents($this->logFile, $message,
FILE_APPEND);
}
```

How it works...

First, make the changes to `Application\Error\Handler` as defined previously. Next, create a class that throws an error that, for this illustration, could be defined as `Application\Error\ThrowsError`. For example, you could have a method that attempts a divide by zero operation, and another that attempts to parse non-PHP code using `eval()`:

```
<?php
namespace Application\Error;
class ThrowsError
{
  const NOT_PARSE = 'this will not parse';
  public function divideByZero()
  {
    $this->zero = 1 / 0;
```

```
    }
    public function willNotParse()
    {
      eval(self::NOT_PARSE);
    }
}
```

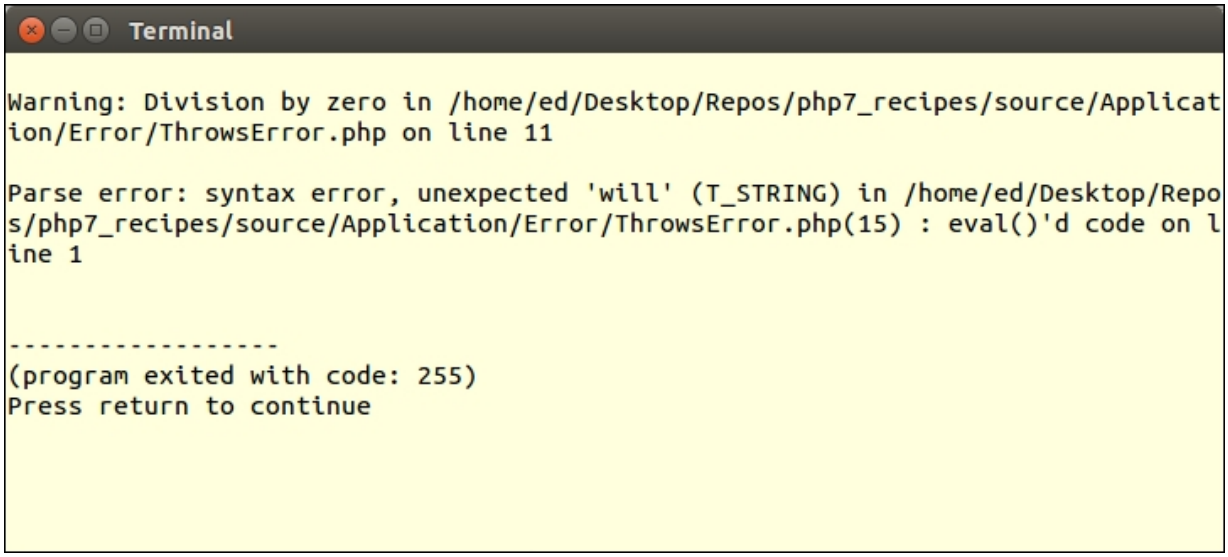
You can then define a calling program called `chap_13_error_throwable.php` that sets up autoloading, uses the appropriate classes, and creates an instance of `ThrowsError`:

```
<?php
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
use Application\Error\ { Handler, ThrowsError };
$error = new ThrowsError();
```

If you then call the two methods, without a try/catch block and without defining the universal error handler, the first method generates a `Warning`, whereas the second throws a `ParseError`:

```
$error->divideByZero();
$error->willNotParse();
echo 'Application continues ... ' . PHP_EOL;
```

Because this is an error, program execution stops, and you will not see `Application continues ...`:



If you wrap the method calls in `try/catch` blocks and catch `Throwable`, the code execution continues:

```
try {
    $error->divideByZero();
} catch (Throwable $e) {
    echo 'Error Caught: ' . get_class($e) . ':'
      . $e->getMessage() . PHP_EOL;
}
try {
    $error->willNotParse();
} catch (Throwable $e) {
    echo 'Error Caught: ' . get_class($e) . ':'
    . $e->getMessage() . PHP_EOL;
}
echo 'Application continues ... ' . PHP_EOL;
```

From the following output, you will also note that the program exits with `code 0`, which tells us all is OK:

```
Terminal

Warning: Division by zero in /home/ed/Desktop/Repos/php7_recipes/source/Applicat
ion/Error/ThrowsError.php on line 11
Error Caught: ParseError:syntax error, unexpected 'will' (T_STRING)
Application continues ...


------------------
(program exited with code: 0)
Press return to continue
```

Finally, after the `try/catch` blocks, run the errors again, moving the echo statement to the end. You will see in the output that the errors were caught, but in the log file, notice that `DivisionByZeroError` is caught by the exception handler, whereas the `ParseError` is caught by the error hander:

```
$handler = new Handler(__DIR__ . '/logs');
$error->divideByZero();
$error->willNotParse();
echo 'Application continues ... ' . PHP_EOL;
```

```
ed@ed: ~/Desktop/Repos/php7_recipes/source/chapter13
ed@ed:~/Desktop/Repos/php7_recipes/source/chapter13$ php chap_13_error_throwable.php

Warning: Division by zero in /home/ed/Desktop/Repos/php7_recipes/source/Application/Error/T
hrowsError.php on line 11
Error Caught: ParseError:syntax error, unexpected 'will' (T_STRING)
ed@ed:~/Desktop/Repos/php7_recipes/source/chapter13$ cat logs/20160610.log
ERROR    : 2016-06-10 07:16:00 : 2 : Division by zero : /home/ed/Desktop/Repos/php7_recipes
/source/Application/Error/ThrowsError.php : 11
EXCEPTION: 2016-06-10 07:16:00 :          ParseError : syntax error, unexpected identifier
 (T_STRING)
ed@ed:~/Desktop/Repos/php7_recipes/source/chapter13$
```

See also

- PHP 7.1 allows you to specify more than one class in the `catch ()` clause. So, instead of a single `Throwable` you could say `catch`

```
(Exception | Error $e) { xxx }
```

Writing a simple test

The primary means of testing PHP code is to use **PHPUnit**, which is based on a methodology called **Unit Testing**. The philosophy behind unit testing is quite simple: you break down your code into the smallest possible logical units. You then test each unit in isolation to confirm that it performs as expected. These expectations are codified into a series of **assertions**. If all assertions return `TRUE`, then the unit has passed the test.

> **Note**
>
> In the case of procedural PHP, a unit is a function. For OOP PHP, the unit is a method within a class.

How to do it...

1. The first order of business is to either install PHPUnit directly onto your development server, or download the source code, which is available in the form of a single **phar** (**PHP archive**) file. A quick visit to the official website for PHPUnit (https://phpunit.de/) lets us download right from the main page.
2. It is a best practice, however, to use a package manager to both install and maintain PHPUnit. For this purpose, we will use a package management program called **Composer**. To install Composer, visit the main website, https://getcomposer.org/, and follow the instructions on the download page. The current procedure, at the time of writing, is as follows. Note that you need to substitute the hash of the current version in place of `<hash>`:

```
php -r
"copy('https://getcomposer.org/installer',
'composer-setup.php');"
php -r "if (hash_file('SHA384', 'composer-
```

```
setup.php') === '<hash>') {
    echo 'Installer verified';
} else {
    echo 'Installer corrupt';
unlink('composer-setup.php');
} echo PHP_EOL;"
php composer-setup.php
php -r "unlink('composer-setup.php');"
```

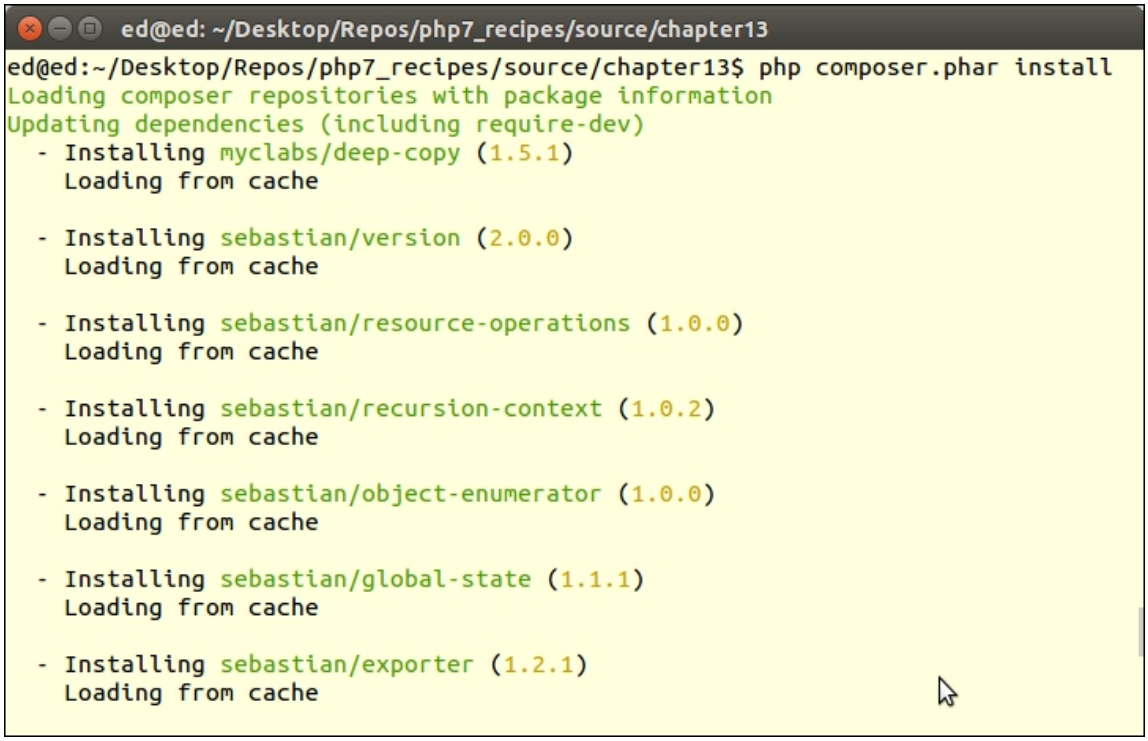
> ## Tip
>
> **Best practice**
>
> The advantage of using a package management program such as Composer is that it will not only install, but can also be used to update any external software (such as PHPUnit) used by your application.

3. Next, we use Composer to install PHPUnit. This is accomplished by creating a `composer.json` file that contains a series of directives outlining project parameters and dependencies. A full description of these directives is beyond the scope of this book; however, for the purposes of this recipe, we create a minimal set of directives using the key parameter `require`. You will also note that the contents of the file are in **JavaScript Object Notation** (**JSON**) format:

```
{
  "require-dev": {
    "phpunit/phpunit": "*"
  }
}
```

4. To perform the installation from the command line, we run the following command. The output is shown just after:

```
php composer.phar install
```



5. PHPUnit and its dependencies are placed in a `vendor` folder that Composer will create if it does not already exist. The primary command to invoke PHPUnit is then symbolically linked into the `vendor/bin` folder. If you place this folder in your `PATH`, all you need do is to run this command, which checks the version and incidentally confirms the installation:

```
phpunit --version
```

Running simple tests

1. For the purposes of this illustration, let's assume we have a `chap_13_unit_test_simple.php` file that contains the `add()` function:

```
<?php
function add($a = NULL, $b = NULL)
{
  return $a + $b;
}
```

2. Tests are then written as classes that extend `PHPUnit\Framework\TestCase`. If you are testing a library of functions, at the beginning of the test class, include the file that contains function definitions. You would then write methods that start with the word `test`, usually followed by the name of the function you are testing, and possibly some additional CamelCase words to further describe the test. For the purposes of this recipe, we will define a `SimpleTest` test class:

```
<?php
use PHPUnit\Framework\TestCase;
require_once __DIR__ .
'/chap_13_unit_test_simple.php';
class SimpleTest extends TestCase
{
  // testXXX() methods go here
}
```

3. Assertions form the heart of any set of tests. The `See also` section gives you the documentation reference for the complete list of assertions. An assertion is a PHPUnit method that compares a known value against a value produced by that which you wish to test. An example is `assertEquals()`, which checks to see whether the first argument equals the second. The following example tests a method called `add()` and confirms **2** is the return value for `add(1,1)`:

```
public function testAdd()
{
  $this->assertEquals(2, add(1,1));
}
```

4. You can also test to see whether something is *not* true. This example asserts that 1 + 1 does not equal 3:

```
$this->assertNotEquals(3, add(1,1));
```

5. An assertion that is extremely useful when used to test a string is `assertRegExp()`. Assume, for this illustration, that we are testing a function that produces an HTML table out of a multidimensional array:

```
function table(array $a)
{
  $table = '<table>';
  foreach ($a as $row) {
    $table .= '<tr><td>';
    $table .= implode('</td><td>', $row);
    $table .= '</td></tr>';
  }
  $table .= '</table>';
  return $table;
}
```

6. We can construct a simple test that confirms that the output contains `<table>`, one or more characters, followed by `</table>`. Further, we wish to confirm that a `<td>B</td>` element exists. When writing the test, we build a test array that consists of three sub-arrays containing the letters A—C, D—F, and G—I. We then pass the test array to the function, and run assertions against the result:

```
public function testTable()
{
  $a = [range('A', 'C'),range('D',
'F'),range('G','I')];
  $table = table($a);
  $this->assertRegExp('!^<table>.+</table>$!',
$table);
  $this->assertRegExp('!<td>B</td>!', $table);
}
```

7. To test a class, instead of including a library of functions, simply include the file that defines the class to be tested. For the sake of illustration, let's take the library of functions shown previously and move them into a `Demo` class:

```
<?php
class Demo
{
  public function add($a, $b)
  {
    return $a + $b;
  }

  public function sub($a, $b)
```

```
    {
      return $a - $b;
    }
    // etc.
}
```

8. In our `SimpleClassTest` test class, instead of including the library file, we include the file that represents the `Demo` class. We need an instance of `Demo` in order to run tests. For this purpose, we use a specially designed `setup()` method, which is run before each test. Also, you will note a `teardown()` method, which is run immediately after each test:

```
<?php
use PHPUnit\Framework\TestCase;
require_once __DIR__ . '/Demo.php';
class SimpleClassTest extends TestCase
{
  protected $demo;
  public function setup()
  {
    $this->demo = new Demo();
  }
  public function teardown()
  {
    unset($this->demo);
  }
  public function testAdd()
  {
    $this->assertEquals(2, $this->demo-
>add(1,1));
  }
  public function testSub()
  {
    $this->assertEquals(0, $this->demo-
>sub(1,1));
  }
  // etc.
}
```

Note

The reason why `setup()` and `teardown()` are run before and after each test is to ensure a fresh test environment. That way, the results of one test will not influence the results of another test.

Testing database Model classes

1. When testing a class, such as a Model class, that has database access, other considerations come into play. The main consideration is that you should run tests against a test database, not the real database used in production. A final point is that by using a test database, you can populate it in advance with appropriate, controlled data. `setup()` and `teardown()` could also be used to add or remove test data.
2. As an example of a class that uses the database, we will define a class `VisitorOps`. The new class will include methods to add, remove, and find visitors. Note that we've also added a method to return the latest SQL statement executed:

```
<?php
require __DIR__ .
'/../Application/Database/Connection.php';
use Application\Database\Connection;
class VisitorOps
{

const TABLE_NAME = 'visitors';
protected $connection;
protected $sql;

public function __construct(array $config)
{
  $this->connection = new Connection($config);
}

public function getSql()
{
  return $this->sql;
}

public function findAll()
```

```
{
  $sql = 'SELECT * FROM ' . self::TABLE_NAME;
  $stmt = $this->runSql($sql);
  while ($row = $stmt-
>fetch(PDO::FETCH_ASSOC)) {
    yield $row;
  }
}

public function findById($id)
{
  $sql = 'SELECT * FROM ' . self::TABLE_NAME;
  $sql .= ' WHERE id = ?';
  $stmt = $this->runSql($sql, [$id]);
  return $stmt->fetch(PDO::FETCH_ASSOC);
}

public function removeById($id)
{
  $sql = 'DELETE FROM ' . self::TABLE_NAME;
  $sql .= ' WHERE id = ?';
  return $this->runSql($sql, [$id]);
}

public function addVisitor($data)
{
  $sql = 'INSERT INTO ' . self::TABLE_NAME;
  $sql .= ' (' .
implode(',',array_keys($data)) . ') ';
  $sql .= ' VALUES ';
  $sql .= ' ( :' .
implode(',:',array_keys($data)) . ') ';
  $this->runSql($sql, $data);
  return $this->connection->pdo-
>lastInsertId();
}

public function runSql($sql, $params = NULL)
{
  $this->sql = $sql;
  try {
      $stmt = $this->connection->pdo-
>prepare($sql);
      $result = $stmt->execute($params);
  } catch (Throwable $e) {
      error_log(__METHOD__ . ':' . $e-
>getMessage());
      return FALSE;
```

```
        }
        return $stmt;
    }
}
```

3. For tests that involve a database, it is recommended that you use a test database instead of the live production database. Accordingly, you will need an extra set of database connection parameters that can be used to establish a database connection in the `setup()` method.
4. It's possible that you wish to establish a consistent block of sample data. This could be inserted into the test database in the `setup()` method.
5. Finally, you may wish to reset the test database after each test, which is accomplished in the `teardown()` method.

Using mock classes

1. In some cases, the test will access complex components that require external resources. An example is a service class that needs access to a database. It is a best practice to minimize database access in a test suite. Another consideration is that we are not testing database access; we are only testing the functionality of one specific class. Accordingly, it is sometimes necessary to define **mock** classes that mimic the behavior of the their parent class, but that restrict access to external resources.

> ### Tip
>
> **Best practice**
>
> Limit actual database access in your tests to the Model (or equivalent) classes. Otherwise, the time it takes to run the entire set of tests could become excessive.

2. In this case, for illustration, define a service class, `VisitorService`, which makes use of the `VisitorOps` class discussed earlier:

```
<?php
require_once __DIR__ . '/VisitorOps.php';
require_once __DIR__ .
'/../Application/Database/Connection.php';
use Application\Database\Connection;
class VisitorService
{
  protected $visitorOps;
  public function __construct(array $config)
  {
    $this->visitorOps = new
VisitorOps($config);
  }
  public function showAllVisitors()
  {
    $table = '<table>';
    foreach ($this->visitorOps->findAll() as
$row) {
      $table .= '<tr><td>';
      $table .= implode('</td><td>', $row);
      $table .= '</td></tr>';
    }
    $table .= '</table>';
    return $table;
  }
```

3. For test purposes, we add a getter and setter for the `$visitorOps` property. This allows us to insert a mock class in place of the real `VisitorOps` class:

```
public function getVisitorOps()
{
  return $this->visitorOps;
}

public function setVisitorOps(VisitorOps
$visitorOps)
{
  $this->visitorOps = $visitorOps;
}
} // closing brace for VisitorService
```

4. Next, we define a `VisitorOpsMock` mock class that mimics the functionality of its parent class. Class constants and properties are inherited. We then add mock test data, and a getter in case we need access to the test data later:

```
<?php
require_once __DIR__ . '/VisitorOps.php';
class VisitorOpsMock extends VisitorOps
{
  protected $testData;
  public function __construct()
  {
    $data = array();
    for ($x = 1; $x <= 3; $x++) {
      $data[$x]['id'] = $x;
      $data[$x]['email'] = $x .
'test@unlikelysource.com';
      $data[$x]['visit_date'] =
        '2000-0' . $x . '-0' . $x . '
00:00:00';
      $data[$x]['comments'] = 'TEST ' . $x;
      $data[$x]['name'] = 'TEST ' . $x;
    }
    $this->testData = $data;
  }
  public function getTestData()
  {
    return $this->testData;
  }
```

5. Next, we override `findAll()` to return test data using `yield`, just as in the parent class. Note that we still build the SQL string, as this is what the parent class does:

```
public function findAll()
{
  $sql = 'SELECT * FROM ' . self::TABLE_NAME;
  foreach ($this->testData as $row) {
    yield $row;
  }
}
```

6. To mock `findById()` we simply return that array key from `$this->testData`. For `removeById()`, we unset the array key supplied as a parameter from `$this->testData`:

```
public function findById($id)
{
  $sql = 'SELECT * FROM ' . self::TABLE_NAME;
  $sql .= ' WHERE id = ?';
  return $this->testData[$id] ?? FALSE;
}
public function removeById($id)
{
  $sql = 'DELETE FROM ' . self::TABLE_NAME;
  $sql .= ' WHERE id = ?';
  if (empty($this->testData[$id])) {
      return 0;
  } else {
      unset($this->testData[$id]);
      return 1;
  }
}
```

7. Adding data is slightly more complicated in that we need to emulate the fact that the `id` parameter might not be supplied, as the database would normally auto-generate this for us. To get around this, we check for the `id` parameter. If not set, we find the largest array key and increment:

```
public function addVisitor($data)
{
  $sql = 'INSERT INTO ' . self::TABLE_NAME;
  $sql .= ' (' .
implode(',',array_keys($data)) . ') ';
  $sql .= ' VALUES ';
  $sql .= ' ( :' .
implode(',:',array_keys($data)) . ') ';
  if (!empty($data['id'])) {
      $id = $data['id'];
  } else {
      $keys = array_keys($this->testData);
      sort($keys);
      $id = end($keys) + 1;
      $data['id'] = $id;
  }
    $this->testData[$id] = $data;
    return 1;
  }

} // ending brace for the class VisitorOpsMock
```

Using anonymous classes as mock objects

1. A nice variation on mock objects involves the use of the new PHP 7 anonymous class in place of creating a formal class that defines mock functionality. The advantage of using an anonymous class is that you can extend an existing class, which makes the object appear legitimate. This approach is especially useful if you only need to override one or two methods.
2. For this illustration, we will modify `VisitorServiceTest.php` presented previously, calling it `VisitorServiceTestAnonClass.php`:

```
<?php
use PHPUnit\Framework\TestCase;
require_once __DIR__ . '/VisitorService.php';
require_once __DIR__ . '/VisitorOps.php';
class VisitorServiceTestAnonClass extends
TestCase
{
  protected $visitorService;
  protected $dbConfig = [
    'driver'   => 'mysql',
    'host'     => 'localhost',
    'dbname'   => 'php7cookbook_test',
    'user'     => 'cook',
    'password' => 'book',
    'errmode'  => PDO::ERRMODE_EXCEPTION,
  ];
    protected $testData;
```

3. You will notice that in `setup()`, we define an anonymous class that extends `VisitorOps`. We only need to override the `findAll()` method:

```
public function setup()
{
  $data = array();
  for ($x = 1; $x <= 3; $x++) {
    $data[$x]['id'] = $x;
    $data[$x]['email'] = $x .
'test@unlikelysource.com';
    $data[$x]['visit_date'] =
      '2000-0' . $x . '-0' . $x . ' 00:00:00';
    $data[$x]['comments'] = 'TEST ' . $x;
```

```
    $data[$x]['name'] = 'TEST ' . $x;
  }
  $this->testData = $data;
  $this->visitorService =
    new VisitorService($this->dbConfig);
  $opsMock =
    new class ($this->testData) extends
VisitorOps {
      protected $testData;
      public function __construct($testData)
      {
        $this->testData = $testData;
      }
      public function findAll()
      {
        return $this->testData;
      }
    };
    $this->visitorService-
>setVisitorOps($opsMock);
}
```

4. Note that in `testShowAllVisitors()`, when `$this->visitorService->showAllVisitors()` is executed, the anonymous class is called by the visitor service, which in turn calls the overridden `findAll()`:

```
public function teardown()
{
  unset($this->visitorService);
}
public function testShowAllVisitors()
{
  $result = $this->visitorService-
>showAllVisitors();
  $this->assertRegExp('!^<table>.+</table>$!',
$result);
  foreach ($this->testData as $key => $value)
{
    $dataWeWant = '!<td>' . $key . '</td>!';
    $this->assertRegExp($dataWeWant, $result);
  }
}
}
```

Using Mock Builder

1. Another technique is to use `getMockBuilder()`. Although this approach does not allow a great deal of finite control over the mock object produced, it's extremely useful in situations where you only need to confirm that an object of a certain class is returned, and when a specified method is run, this method returns some expected value.
2. In the following example, we copied `VisitorServiceTestAnonClass`; the only difference is in how an instance of `VisitorOps` is supplied in `setup()`, in this case, using `getMockBuilder()`. Note that although we did not use `with()` in this example, it is used to feed controlled parameters to the mocked method:

```
<?php
use PHPUnit\Framework\TestCase;
require_once __DIR__ . '/VisitorService.php';
require_once __DIR__ . '/VisitorOps.php';
class VisitorServiceTestAnonMockBuilder
extends TestCase
{
  // code is identical to
VisitorServiceTestAnon
  public function setup()
  {
    $data = array();
    for ($x = 1; $x <= 3; $x++) {
      $data[$x]['id'] = $x;
      $data[$x]['email'] = $x .
'test@unlikelysource.com';
      $data[$x]['visit_date'] =
        '2000-0' . $x . '-0' . $x . '
00:00:00';
      $data[$x]['comments'] = 'TEST ' . $x;
      $data[$x]['name'] = 'TEST ' . $x;
  }
  $this->testData = $data;
    $this->visitorService =
      new VisitorService($this->dbConfig);
    $opsMock = $this-
>getMockBuilder(VisitorOps::class)
                    ->setMethods(['findAll'])
```

```
                        -
>disableOriginalConstructor()
                        ->getMock();
                        $opsMock->expects($this-
>once())
                        ->method('findAll')
                        ->with()
                        ->will($this-
>returnValue($this->testData));
                        $this->visitorService-
>setVisitorOps($opsMock);
    }
    // remaining code is the same
}
```

> **Note**
>
> We have shown how to create simple one-off tests. In most cases, however, you will have many classes that need to be tested, preferably all at once. This is possible by developing a *test suite*, discussed in more detail in the next recipe.

How it works...

First, you need to install PHPUnit, as discussed in steps 1 to 5. Be sure to include `vendor/bin` in your PATH so that you can run PHPUnit from the command line.

Running simple tests

Next, define a `chap_13_unit_test_simple.php` program file with a series of simple functions, such as `add()`, `sub()` and so on, as discussed in step 1. You can then define a simple test class contained in `SimpleTest.php` as mentioned in steps 2 and 3.

Assuming `phpunit` is in your `PATH`, from a terminal window, change to the directory containing the code developed for this recipe, and run

the following command:

```
phpunit SimpleTest SimpleTest.php
```

You should see the following output:

```
ed@ed: ~/Desktop/Repos/php7_recipes/source/chapter13
ed@ed:~/Desktop/Repos/php7_recipes/source/chapter13$ phpunit SimpleTest SimpleTest.php
PHPUnit 5.4.3 by Sebastian Bergmann and contributors.

.....                                                               5 / 5 (100%)

Time: 33 ms, Memory: 4.00MB

OK (5 tests, 9 assertions)
ed@ed:~/Desktop/Repos/php7_recipes/source/chapter13$
```

Make a change in `SimpleTest.php` so that the test will fail (step 4):

```
public function testDiv()
{
  $this->assertEquals(2, div(4, 2));
  $this->assertEquals(99, div(4, 0));
}
```

Here is the revised output:

```
ed@ed: ~/Desktop/Repos/php7_recipes/source/chapter13
ed@ed:~/Desktop/Repos/php7_recipes/source/chapter13$ phpunit SimpleTest SimpleTest.php
PHPUnit 5.4.3 by Sebastian Bergmann and contributors.

...F.                                                               5 / 5 (100%)

Time: 25 ms, Memory: 4.00MB

There was 1 failure:

1) SimpleTest::testDiv
Failed asserting that 0 matches expected 99.

/home/ed/Desktop/Repos/php7_recipes/source/chapter13/SimpleTest.php:28

FAILURES!
Tests: 5, Assertions: 9, Failures: 1.
ed@ed:~/Desktop/Repos/php7_recipes/source/chapter13$
```

Next, add the `table()` function to `chap_13_unit_test_simple.php` (step 5), and `testTable()` to `SimpleTest.php` (step 6). Re-run the unit test and observe the results.

To test a class, copy the functions developed in `chap_13_unit_test_simple.php` to a `Demo` class (step 7). After making the modifications to `SimpleTest.php` suggested in step 8, re-run the simple test and observe the results.

Testing database model classes

First, create an example class to be tested, `VisitorOps`, shown in step 2 in this subsection. You can now define a class we will call `SimpleDatabaseTest` to test `VisitorOps`. First of all, use `require_once` to load the class to test. (We will discuss how to incorporate autoloading in the next recipe!) Then define key properties, including test database configuration and test data. You could use `php7cookbook_test` as the test database:

```
<?php
use PHPUnit\Framework\TestCase;
require_once __DIR__ . '/VisitorOps.php';
class SimpleDatabaseTest extends TestCase
{
  protected $visitorOps;
  protected $dbConfig = [
    'driver'   => 'mysql',
    'host'     => 'localhost',
    'dbname'   => 'php7cookbook_test',
    'user'     => 'cook',
    'password' => 'book',
    'errmode'  => PDO::ERRMODE_EXCEPTION,
  ];
  protected $testData = [
    'id' => 1,
    'email' => 'test@unlikelysource.com',
    'visit_date' => '2000-01-01 00:00:00',
    'comments' => 'TEST',
    'name' => 'TEST'
  ];
}
```

Next, define `setup()`, which inserts the test data, and confirms that the last SQL statement was `INSERT`. You should also check to see whether the return value was positive:

```
public function setup()
{
  $this->visitorOps = new VisitorOps($this-
>dbConfig);
  $this->visitorOps->addVisitor($this->testData);
  $this->assertRegExp('/INSERT/', $this-
>visitorOps->getSql());
}
```

After that, define `teardown()`, which removes the test data and confirms that the query for `id = 1` comes back as `FALSE`:

```
public function teardown()
{
  $result = $this->visitorOps->removeById(1);
  $result = $this->visitorOps->findById(1);
  $this->assertEquals(FALSE, $result);
  unset($this->visitorOps);
}
```

The first test is for `findAll()`. First, confirm the data type of the result. You could take the topmost element using `current()`. We confirm there are five elements, that one of them is `name`, and that the value is the same as that in the test data:

```
public function testFindAll()
{
  $result = $this->visitorOps->findAll();
  $this->assertInstanceOf(Generator::class,
$result);
  $top = $result->current();
  $this->assertCount(5, $top);
  $this->assertArrayHasKey('name', $top);
  $this->assertEquals($this->testData['name'],
$top['name']);
}
```

The next test is for `findById()`. It is almost identical to `testFindAll()`:

```
public function testFindById()
{
  $result = $this->visitorOps->findById(1);
  $this->assertCount(5, $result);
  $this->assertArrayHasKey('name', $result);
  $this->assertEquals($this->testData['name'],
$result['name']);
}
```

You do not need to bother with a test for `removeById()` as this is already done in `teardown()`. Likewise, there is no need to test `runSql()` as this is done as part of the other tests.

Using mock classes

First, define a `VisitorService` service class as described in steps 2 and 3 in this subsection. Next, define a `VisitorOpsMock` mock class, which is discussed in steps 4 to 7.

You are now in a position to develop a test, `VisitorServiceTest`, for the service class. Note that you need provide your own database configuration as it is a best practice to use a test database instead of the production version:

```
<?php
use PHPUnit\Framework\TestCase;
require_once __DIR__ . '/VisitorService.php';
require_once __DIR__ . '/VisitorOpsMock.php';

class VisitorServiceTest extends TestCase
{
  protected $visitorService;
  protected $dbConfig = [
    'driver'   => 'mysql',
    'host'     => 'localhost',
    'dbname'   => 'php7cookbook_test',
    'user'     => 'cook',
    'password' => 'book',
    'errmode'  => PDO::ERRMODE_EXCEPTION,
```

```
    ];
  }
```

In `setup()`, create an instance of the service, and insert `VisitorOpsMock` in place of the original class:

```
  public function setup()
  {
    $this->visitorService = new
  VisitorService($this->dbConfig);
    $this->visitorService->setVisitorOps(new
  VisitorOpsMock());
  }
  public function teardown()
  {
    unset($this->visitorService);
  }
```

In our test, which produces an HTML table from the list of visitors, you can then look for certain elements, knowing what to expect in advance as you have control over the test data:

```
  public function testShowAllVisitors()
  {
    $result = $this->visitorService-
  >showAllVisitors();
    $this->assertRegExp('!^<table>.+</table>$!',
  $result);
    $testData = $this->visitorService-
  >getVisitorOps()->getTestData();
    foreach ($testData as $key => $value) {
      $dataWeWant = '!<td>' . $key . '</td>!';
      $this->assertRegExp($dataWeWant, $result);
    }
  }
  }
```

You might then wish to experiment with the variations suggested in the last two subsections, *Using Anonymous Classes as Mock Objects*, and *Using Mock Builder*.

There's more...

Other assertions test operations on numbers, strings, arrays, objects, files, JSON, and XML, as summarized in the following table:

Category	Assertions
General	`assertEquals()`, `assertFalse()`, `assertEmpty()`, `assertNull()`, `assertSame()`, `assertThat()`, `assertTrue()`
Numeric	`assertGreaterThan()`, `assertGreaterThanOrEqual()`, `assertLessThan()`, `assertLessThanOrEqual()`, `assertNan()`, `assertInfinite()`
String	`assertStringEndsWith()`, `assertStringEqualsFile()`, `assertStringStartsWith()`, `assertRegExp()`, `assertStringMatchesFormat()`, `assertStringMatchesFormatFile()`
Array/iterator	`assertArrayHasKey()`, `assertArraySubset()`, `assertContains()`, `assertContainsOnly()`, `assertContainsOnlyInstancesOf()`, `assertCount()`
File	`assertFileEquals()`, `assertFileExists()`

Category	Assertions
Objects	`assertClassHasAttribute()`, `assertClassHasStaticAttribute()`, `assertInstanceOf()`, `assertInternalType()`, `assertObjectHasAttribute()`
JSON	`assertJsonFileEqualsJsonFile()`, `assertJsonStringEqualsJsonFile()`, `assertJsonStringEqualsJsonString()`
XML	`assertEqualXMLStructure()`, `assertXmlFileEqualsXmlFile()`, `assertXmlStringEqualsXmlFile()`, `assertXmlStringEqualsXmlString()`

See also...

- For a good discussion on unit testing, have a look here: https://en.wikipedia.org/wiki/Unit_testing.
- For more information on `composer.json` file directives, see https://getcomposer.org/doc/04-schema.md.
- For a complete list of assertions, have a look at this PHPUnit documentation page: https://phpunit.de/manual/current/en/phpunit-book.html#appendixes.assertions.
- The PHPUnit documentation also goes into using `getMockBuilder()` in detail here: https://phpunit.de/manual/current/en/phpunit-book.html#test-doubles.mock-objects

Writing a test suite

You may have noticed after having read through the previous recipe that it can quickly become tedious to have to manually run `phpunit` and specify test classes and PHP filenames. This is especially true when dealing with applications that employ dozens or even hundreds of classes and files. The PHPUnit project has a built-in capability to handle running multiple tests with a single command. Such a set of tests is referred to as a **test suite**.

How to do it...

1. At its simplest, all you need to do is to move all the tests into a single folder:

```
mkdir tests
cp *Test.php tests
```

2. You'll need to adjust commands that include or require external files to account for the new location. The example shown (`SimpleTest`) was developed in the preceding recipe:

```
<?php
use PHPUnit\Framework\TestCase;
require_once __DIR__ .
'/../chap_13_unit_test_simple.php';

class SimpleTest extends TestCase
{
  // etc.
```

3. You can then simply run `phpunit` with the directory path as an argument. PHPUnit will then automatically run all tests in that folder. In this example, we assume there is a `tests` subdirectory:

```
phpunit tests
```

4. You can use the `--bootstrap` option to specify a file that is executed prior to running the tests. A typical use for this option is to initiate autoloading:

```
phpunit --boostrap
tests_with_autoload/bootstrap.php tests
```

5. Here is the sample `bootstrap.php` file that implements autoloading:

```
<?php
require __DIR__ .
'/../../Application/Autoload/Loader.php';
Application\Autoload\Loader::init([__DIR__]);
```

6. Another possibility is to define one or more sets of tests using an XML configuration file. Here is an example that runs only the Simple* tests:

```
<phpunit>
  <testsuites>
    <testsuite name="simple">
      <file>SimpleTest.php</file>
      <file>SimpleDbTest.php</file>
      <file>SimpleClassTest.php</file>
    </testsuite>
  </testsuites>
</phpunit>
```

7. Here is another example that runs a test based on a directory and also specifies a bootstrap file:

```
<phpunit bootstrap="bootstrap.php">
  <testsuites>
    <testsuite name="visitor">
      <directory>Simple</directory>
    </testsuite>
  </testsuites>
</phpunit>
```

How it works...

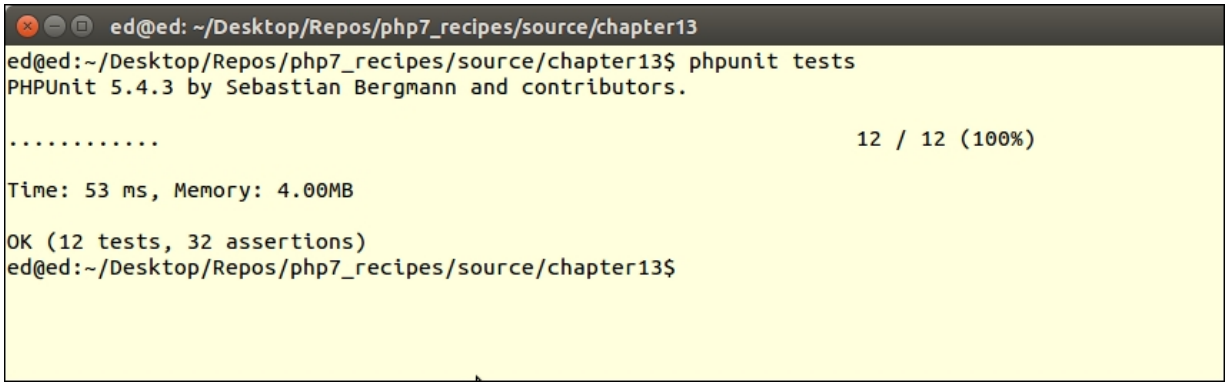
Make sure all the tests discussed in the previous recipe, *Writing a simple test*, have been defined. You can then create a `tests` folder and move or copy all the `*Test.php` files into this folder. You'll then

need to adjust the path in the `require_once` statements, as shown in step 2.

In order to demonstrate how PHPUnit can run all tests in a folder, from the directory containing the source code you defined for this chapter, run the following command:

```
phpunit tests
```

You should see the following output:



To demonstrate the use of a autoloading via a bootstrap file, create a new `tests_with_autoload` directory. In this folder, define a `bootstrap.php` file with the code shown in step 5. Create two directories in `tests_with_autoload`: `Demo` and `Simple`.

From the directory containing the source code for this chapter, copy the file (discussed in step 12 of the previous recipe) into `tests_with_autoload/Demo/Demo.php`. After the opening `<?php` tag, add this line:

```
namespace Demo;
```

Next, copy the `SimpleTest.php` file to `tests_with_autoload/Simple/ClassTest.php`. (Notice the filename change!). You will need to change the first few lines to the following:

```
<?php
namespace Simple;
use Demo\Demo;
use PHPUnit\Framework\TestCase;

class ClassTest extends TestCase
{
  protected $demo;
  public function setup()
  {
    $this->demo = new Demo();
  }
// etc.
```

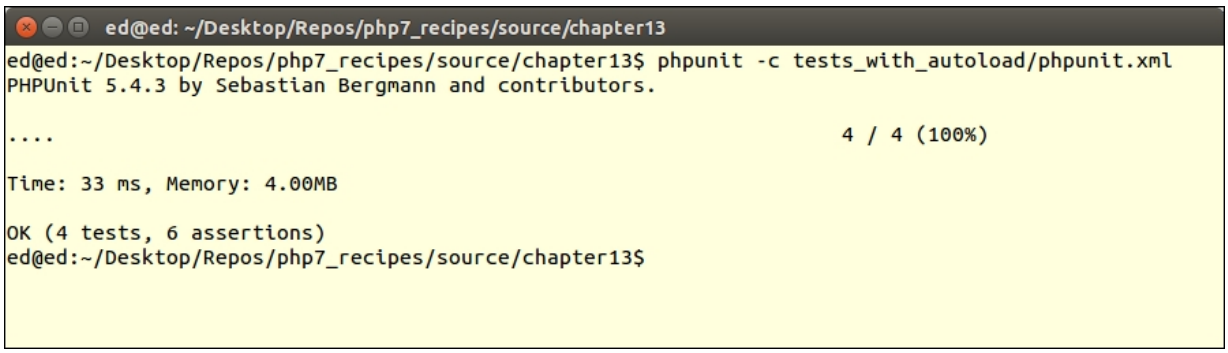
After that, create a `tests_with_autoload/phpunit.xml` file that pulls everything together:

```
<phpunit bootstrap="bootstrap.php">
  <testsuites>
    <testsuite name="visitor">
      <directory>Simple</directory>
    </testsuite>
  </testsuites>
</phpunit>
```

Finally, change to the directory that contains the code for this chapter. You can now run a unit test that incorporates a bootstrap file, along with autoloading and namespaces, as follows:

```
phpunit -c tests_with_autoload/phpunit.xml
```

The output should appear as follows:



See also...

- For more information on writing PHPUnit test suites, have a look at this documentation page: https://phpunit.de/manual/current/en/phpunit-book.html#organizing-tests.xml-configuration.

Generating fake test data

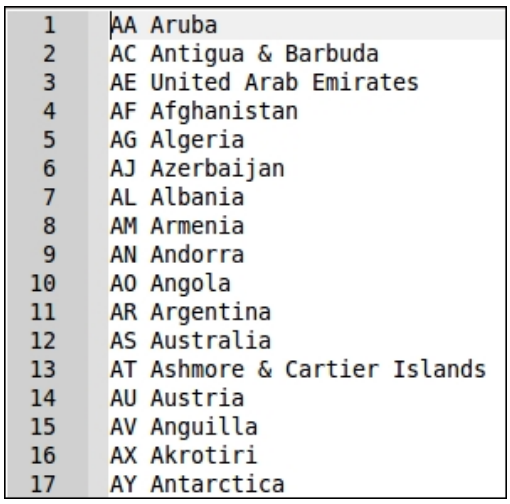
Part of the testing and debugging process involves incorporating realistic test data. In some cases, especially when testing database access and producing benchmarks, large amounts of test data are needed. One way in which this can be accomplished is to incorporate a process of scraping data from websites, and then putting the data together in realistic, yet random, combinations to be inserted into a database.

How to do it...

1. The first step is to determine what data is needed in order to test your application. Another consideration is dose the website address an international audience, or will the market be primarily from a single country?
2. In order to produce a consistent fake data tool, it's extremely important to move the data from its source into a usable digital format. The first choice is a series of database tables. Another, not as attractive, alternative is a CSV file.
3. You may end up converting the data in stages. For example, you could pull data from a web page that lists country codes and country names into a text file.

AF Afghanistan	AG Algeria	AJ Azerbaijan
AL Albania	AM Armenia	AN Andorra
AO Angola	AR Argentina	AS Australia
AT Ashmore & Cartier Islands	AU Austria	AV Anguilla
AX Akrotiri	AY Antarctica	BA Bahrain
BB Barbados	BC Botswana	BD Bermuda
BE Belgium	BF Bahamas, The	BG Bangladesh
BH Belize	BK Bosnia & Herzegovina	BL Bolivia
BM Burma	BN Benin	BO Belarus
BP Soloman Islands	BR Brazil	BS Bassas Da India
BT Bhutan	BU Bulgaria	BV Bouvet Island
BX Brunei	BY Burundi	CA Canada
CB Cambodia	CD Chad	CE Sri Lanka
CF Congo	CG Congo (Dem. Republic of The)-(Zaire)	CH China
CI Chile	CJ Cayman Islands	CK Cocos (Keeling) Islands
CM Cameroon	CN Comoros	CO Colombia
CR Coral Sea Islands	CS Costa Rica	CT Central African Republic
CU Cuba	CV Cape Verde	CW Cook Islands

4. Since this list is short, it's easy to literally cut and paste this into a text file.
5. We can then do a search for " " and replace with "`\n`", which gives us this:



```
 1  AA Aruba
 2  AC Antigua & Barbuda
 3  AE United Arab Emirates
 4  AF Afghanistan
 5  AG Algeria
 6  AJ Azerbaijan
 7  AL Albania
 8  AM Armenia
 9  AN Andorra
10  AO Angola
11  AR Argentina
12  AS Australia
13  AT Ashmore & Cartier Islands
14  AU Austria
15  AV Anguilla
16  AX Akrotiri
17  AY Antarctica
```

6. This can then be imported into a spreadsheet, which then lets you export to a CSV file. From there, it's a simple matter to import it into a database. phpMyAdmin, for example, has such a facility.
7. For the sake of this illustration, we will assume that we are generating data that will end up in the `prospects` table. Here is the SQL statement used to create this table:

```
CREATE TABLE 'prospects' (
  'id' int(11) NOT NULL AUTO_INCREMENT,
  'first_name' varchar(128) NOT NULL,
  'last_name' varchar(128) NOT NULL,
  'address' varchar(256) DEFAULT NULL,
  'city' varchar(64) DEFAULT NULL,
  'state_province' varchar(32) DEFAULT NULL,
  'postal_code' char(16) NOT NULL,
  'phone' varchar(16) NOT NULL,
  'country' char(2) NOT NULL,
  'email' varchar(250) NOT NULL,
  'status' char(8) DEFAULT NULL,
  'budget' decimal(10,2) DEFAULT NULL,
  'last_updated' datetime DEFAULT NULL,
  PRIMARY KEY ('id'),
  UNIQUE KEY 'UNIQ_35730C06E7927C74' ('email')
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

8. Now it's time to create a class that is capable of generating fake data. We will then create methods to generate data for each of the fields shown above, except for `id`, which is auto-generated:

```
namespace Application\Test;

use PDO;
use Exception;
use DateTime;
use DateInterval;
use PDOException;
use SplFileObject;
```

```
use InvalidArgumentsException;
use Application\Database\Connection;

class FakeData
{
  // data generation methods here
}
```

9. Next, we define constants and properties that will be used as part of the process:

```
const MAX_LOOKUPS     = 10;
const SOURCE_FILE     = 'file';
const SOURCE_TABLE    = 'table';
const SOURCE_METHOD   = 'method';
const SOURCE_CALLBACK = 'callback';
const FILE_TYPE_CSV   = 'csv';
const FILE_TYPE_TXT   = 'txt';
const ERROR_DB        = 'ERROR: unable to read source
table';
const ERROR_FILE      = 'ERROR: file not found';
const ERROR_COUNT     = 'ERROR: unable to ascertain count
or ID column missing';
const ERROR_UPLOAD    = 'ERROR: unable to upload file';
const ERROR_LOOKUP    = 'ERROR: unable to find any IDs in
the source table';

protected $connection;
protected $mapping;
protected $files;
protected $tables;
```

10. We then define properties that will be used to generate random letters, street names, and e-mail addresses. You can think of these arrays as seeds that can be modified and/or expanded to suite your needs. As an example, you might substitute street name fragments in Paris for a French audience:

```
protected $alpha = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
protected $street1 =
['Amber','Blue','Bright','Broad','Burning',
  'Cinder','Clear','Dewy','Dusty','Easy']; // etc.
protected $street2 =
['Anchor','Apple','Autumn','Barn','Beacon',
  'Bear','Berry','Blossom','Bluff','Cider','Cloud']; //
etc.
protected $street3 =
['Acres','Arbor','Avenue','Bank','Bend',
  'Canyon','Circle','Street'];
protected $email1 =
['northern','southern','eastern','western',
  'fast','midland','central'];
protected $email2 = ['telecom','telco','net','connect'];
protected $email3 = ['com','net'];
```

11. In the constructor, we accept a `Connection` object, used for database access, an array of mappings to the fake data:

```
public function __construct(Connection $conn, array
$mapping)
{
  $this->connection = $conn;
  $this->mapping = $mapping;
}
```

12. To generate street names, rather than attempt to create a database table, it might be more efficient to use a set of seed arrays to generate random combinations. Here is an example of how this might work:

```
public function getAddress($entry)
{
  return random_int(1,999)
   . ' ' . $this->street1[array_rand($this->street1)]
   . ' ' . $this->street2[array_rand($this->street2)]
   . ' ' . $this->street3[array_rand($this->street3)];
}
```

13. Depending on the level of realism desired, you could also build a database table that matches postal codes to cities. Postal codes could also be randomly generated. Here is an example that generates postal codes for the UK:

```
public function getPostalCode($entry, $pattern = 1)
{
  return $this->alpha[random_int(0,25)]
   . $this->alpha[random_int(0,25)]
   . random_int(1, 99)
   . ' '
   . random_int(1, 9)
   . $this->alpha[random_int(0,25)]
   . $this->alpha[random_int(0,25)];
}
```

14. Fake e-mail generation can likewise use a set of seed arrays to produce random results. We could also program it to receive an existing `$entry` array, with parameters, and use those parameters to create the name portion of the address:

```
public function getEmail($entry, $params = NULL)
{
  $first = $entry[$params[0]] ?? $this-
>alpha[random_int(0,25)];
  $last  = $entry[$params[1]] ?? $this-
>alpha[random_int(0,25)];
  return $first[0] . '.' . $last
   . '@'
   . $this->email1[array_rand($this->email1)]
   . $this->email2[array_rand($this->email2)]
   . '.'
```

```
        . $this->email3[array_rand($this->email3)];
    }
```

15. For date generation, one approach would be to accept as arguments an existing `$entry` array, with parameters. The parameters would be an array where the first value is a start date. The second parameter would be the maximum number of days to *subtract* from the start date. This effectively lets you return a random date from a range. Note that we use `DateTime::sub()` to subtract a random number of days. `sub()` requires a `DateInterval` instance, which we build using `P`, the random number of days, and then `'D'`:

```
public function getDate($entry, $params)
{
  list($fromDate, $maxDays) = $params;
  $date = new DateTime($fromDate);
  $date->sub(new DateInterval('P' . random_int(0,
$maxDays) . 'D'));
  return $date->format('Y-m-d H:i:s');
}
```

16. As mentioned at the beginning of this recipe, the data sources we will use for fake data generation will vary. In some cases, as shown in the previous few steps, we use seed arrays, and build the fake data. In other cases, we might want to use a text or CSV file as a data source. Here is how such a method might look:

```
public function getEntryFromFile($name, $type)
{
  if (empty($this->files[$name])) {
      $this->pullFileData($name, $type);
  }
  return $this->files[$name][
  random_int(0, count($this->files[$name]))];
}
```

17. You will note that we first need to pull the file data into an array, which forms the return value. Here is the method that does that for us. We throw an `Exception` if the specified file is not found. The file type is identified as one of our class constants: `FILE_TYPE_TEXT` or `FILE_TYPE_CSV`. Depending on the type, we use either `fgetcsv()` or `fgets()`:

```
public function pullFileData($name, $type)
{
  if (!file_exists($name)) {
      throw new Exception(self::ERROR_FILE);
  }
  $fileObj = new SplFileObject($name, 'r');
  if ($type == self::FILE_TYPE_CSV) {
      while ($data = $fileObj->fgetcsv()) {
        $this->files[$name][] = trim($data);
      }
  } else {
      while ($data = $fileObj->fgets()) {
```

```
            $this->files[$name][] = trim($data);
        }
    }
```

18. Probably the most complicated aspect of this process is drawing random data from a database table. We accept as arguments the table name, the name of the column that comprises the primary key, an array that maps between the database column name in the lookup table, and the target column name:

```
public function getEntryFromTable($tableName, $idColumn,
$mapping)
{
  $entry = array();
  try {
      if (empty($this->tables[$tableName])) {
        $sql  = 'SELECT ' . $idColumn . ' FROM ' .
$tableName
          . ' ORDER BY ' . $idColumn . ' ASC LIMIT 1';
        $stmt = $this->connection->pdo->query($sql);
        $this->tables[$tableName]['first'] =
          $stmt->fetchColumn();
        $sql  = 'SELECT ' . $idColumn . ' FROM ' .
$tableName
          . ' ORDER BY ' . $idColumn . ' DESC LIMIT 1';
        $stmt = $this->connection->pdo->query($sql);
        $this->tables[$tableName]['last'] =
          $stmt->fetchColumn();
    }
```

19. We are now in a position to set up the prepared statement and initialize a number of critical variables:

```
$result = FALSE;
$count = self::MAX_LOOKUPS;
$sql  = 'SELECT * FROM ' . $tableName
  . ' WHERE ' . $idColumn . ' = ?';
$stmt = $this->connection->pdo->prepare($sql);
```

20. The actual lookup we place inside a `do...while` loop. The reason for this is that we need to run the query at least once to achieve results. Only if we do not arrive at a result do we continue with the loop. We generate a random number between the lowest ID and the highest ID, and then use this in a parameter in the query. Notice that we also decrement a counter to prevent an endless loop. This is in case the IDs are not sequential, in which case we could accidentally generate an ID that does not exist. If we exceed the maximum attempts, still with no results, we throw an `Exception`:

```
do {
  $id = random_int($this->tables[$tableName]['first'],
    $this->tables[$tableName]['last']);
  $stmt->execute([$id]);
  $result = $stmt->fetch(PDO::FETCH_ASSOC);
} while ($count-- && !$result);
```

```
        if (!$result) {
            error_log(__METHOD__ . ':' . self::ERROR_LOOKUP);
            throw new Exception(self::ERROR_LOOKUP);
        }
    } catch (PDOException $e) {
        error_log(__METHOD__ . ':' . $e->getMessage());
        throw new Exception(self::ERROR_DB);
    }
```

21. We then use the mapping array to retrieve values from the source table using keys expected in the destination table:

```
    foreach ($mapping as $key => $value) {
        $entry[$value] = $result[$key] ?? NULL;
    }
    return $entry;
    }
```

22. The heart of this class is a `getRandomEntry()` method, which generates a single array of fake data. We loop through `$mapping` one entry at a time and examine the various parameters:

```
    public function getRandomEntry()
    {
        $entry = array();
        foreach ($this->mapping as $key => $value) {
            if (isset($value['source'])) {
                switch ($value['source']) {
```

23. The `source` parameter is used to implement what effectively serves as a Strategy Pattern. We support four different possibilities for `source`, all defined as class constants. The first one is `SOURCE_FILE`. In this case, we use the `getEntryFromFile()` method discussed previously:

```
                    case self::SOURCE_FILE :
                        $entry[$key] = $this->getEntryFromFile(
                        $value['name'], $value['type']);
                        break;
```

24. The callback option returns a value according to the callback supplied in the `$mapping` array:

```
                    case self::SOURCE_CALLBACK :
                        $entry[$key] = $value['name']();
                        break;
```

25. The `SOURCE_TABLE` option uses the database table defined in `$mapping` as a lookup. Note that `getEntryFromTable()`, discussed previously, is able to return an array of values, which means we need to use `array_merge()` to consolidate the results:

```
                    case self::SOURCE_TABLE :
                        $result = $this->getEntryFromTable(
```

```
$value['name'],$value['idCol'],$value['mapping']);
            $entry = array_merge($entry, $result);
          break;
```

26. The `SOURCE_METHOD` option, which is also the default, uses a method already included with this class. We check to see whether parameters are included, and, if so, add those to the method call. Note the use of `{}` to influence interpolation. If we made a `$this->$value['name']()` PHP 7 call, due to the Abstract Syntax Tree (AST) rewrite, it would interpolate like this, `${$this->$value}['name']()`, which is not what we want:

```
            case self::SOURCE_METHOD :
            default :
              if (!empty($value['params'])) {
                  $entry[$key] = $this->{$value['name']}(
                    $entry, $value['params']);
              } else {
                  $entry[$key] = $this->{$value['name']}
($entry);
              }
            }
        }
    }
    return $entry;
}
```

27. We define a method that loops through `getRandomEntry()` to produce multiple lines of fake data. We also add an option to insert to a destination table. If this option is enabled, we set up a prepared statement to insert, and also check to see whether we need to truncate any data currently in this table:

```
public function generateData(
$howMany, $destTableName = NULL, $truncateDestTable =
FALSE)
{
  try {
      if ($destTableName) {
        $sql = 'INSERT INTO ' . $destTableName
          . ' (' . implode(',', array_keys($this-
>mapping))
          . ') '. ' VALUES ' . ' (:'
          . implode(',:', array_keys($this->mapping)) .
')';
        $stmt = $this->connection->pdo->prepare($sql);
        if ($truncateDestTable) {
          $sql = 'DELETE FROM ' . $destTableName;
          $this->connection->pdo->query($sql);
        }
      }
  } catch (PDOException $e) {
      error_log(__METHOD__ . ':' . $e->getMessage());
      throw new Exception(self::ERROR_COUNT);
  }
```

28. Next, we loop through the number of lines of data requested, and run `getRandomEntry()`. If a database insert is requested, we execute the prepared statement in a `try/catch` block. In any event, we turn this method into a generator using the `yield` keyword:

```
for ($x = 0; $x < $howMany; $x++) {
  $entry = $this->getRandomEntry();
  if ($insert) {
    try {
        $stmt->execute($entry);
    } catch (PDOException $e) {
        error_log(__METHOD__ . ':' . $e->getMessage());
        throw new Exception(self::ERROR_DB);
    }
  }
  yield $entry;
}
}
```

> ## Tip
>
> **Best practice**
>
> If the amount of data to be returned is massive, it's much better to yield the data as it is produced, thus saving the memory required for an array.

How it works...

The first thing to do is to ensure you have the data ready for random data generation. In this recipe, we will presume that the destination table is `prospects`, which has the following SQL database definition shown in step 7.

As a data source for names, you could create text files for first names and surnames. In this illustration, we will reference the `data/files` directory, and the files `first_names.txt` and `surnames.txt`. For city, state or province, postal code, and country, it might be useful to download the data from a source such as http://www.geonames.org/, and upload to a `world_city_data` table. For the remaining fields, such as address, e-mail, status, and so on, you could either use methods built into `FakeData`, or define callbacks.

Next, be sure to define `Application\Test\FakeData`, adding the content discussed in steps 8 to 29. After you have finished, create a calling program called `chap_13_fake_data.php`, which sets up autoloading and uses the appropriate

classes. You should also define constants that match the path to the database configuration, and names files:

```
<?php
define('DB_CONFIG_FILE', __DIR__ .
'/../config/db.config.php');
define('FIRST_NAME_FILE', __DIR__ .
'/../data/files/first_names.txt');
define('LAST_NAME_FILE', __DIR__ .
'/../data/files/surnames.txt');
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\Test\FakeData;
use Application\Database\Connection;
```

Next, define a mapping array that uses the column names in the destination table (prospects) as a key. You need to then define sub-keys for `source`, `name`, and any other parameters that are required. For starters, '`first_name`' and '`last_name`' will both use a file as a source, 'name' points to the name of the file, and '`type`' indicates a file type of text:

```
$mapping = [
  'first_name'    => ['source' => FakeData::SOURCE_FILE,
  'name'          => FIRST_NAME_FILE,
  'type'          => FakeData::FILE_TYPE_TXT],
  'last_name'     => ['source' => FakeData::SOURCE_FILE,
  'name'          => LAST_NAME_FILE,
  'type'          => FakeData::FILE_TYPE_TXT],
```

The `'address'`, `'email'`, and `'last_updated'` all use built-in methods as a data source. The last two also define parameters to be passed:

```
  'address'       => ['source' => FakeData::SOURCE_METHOD,
  'name'          => 'getAddress'],
  'email'         => ['source' => FakeData::SOURCE_METHOD,
  'name'          => 'getEmail',
  'params'        => ['first_name','last_name']],
  'last_updated' => ['source' => FakeData::SOURCE_METHOD,
  'name'          => 'getDate',
  'params'        => [date('Y-m-d'), 365*5]]
```

The `'phone'`, `'status'` and `'budget'` could all use callbacks to provide fake data:

```
  'phone'         => ['source' => FakeData::SOURCE_CALLBACK,
  'name'          => function () {
                     return sprintf('%3d-%3d-%4d',
random_int(101,999),
                     random_int(101,999), random_int(0,9999));
}],
  'status'        => ['source' => FakeData::SOURCE_CALLBACK,
```

```
  'name'         => function () { $status =
['BEG','INT','ADV'];
                    return $status[rand(0,2)]; }],
  'budget'       => ['source' => FakeData::SOURCE_CALLBACK,
                     'name' => function() { return
random_int(0, 99999)
                     + (random_int(0, 99) * .01); }]
```

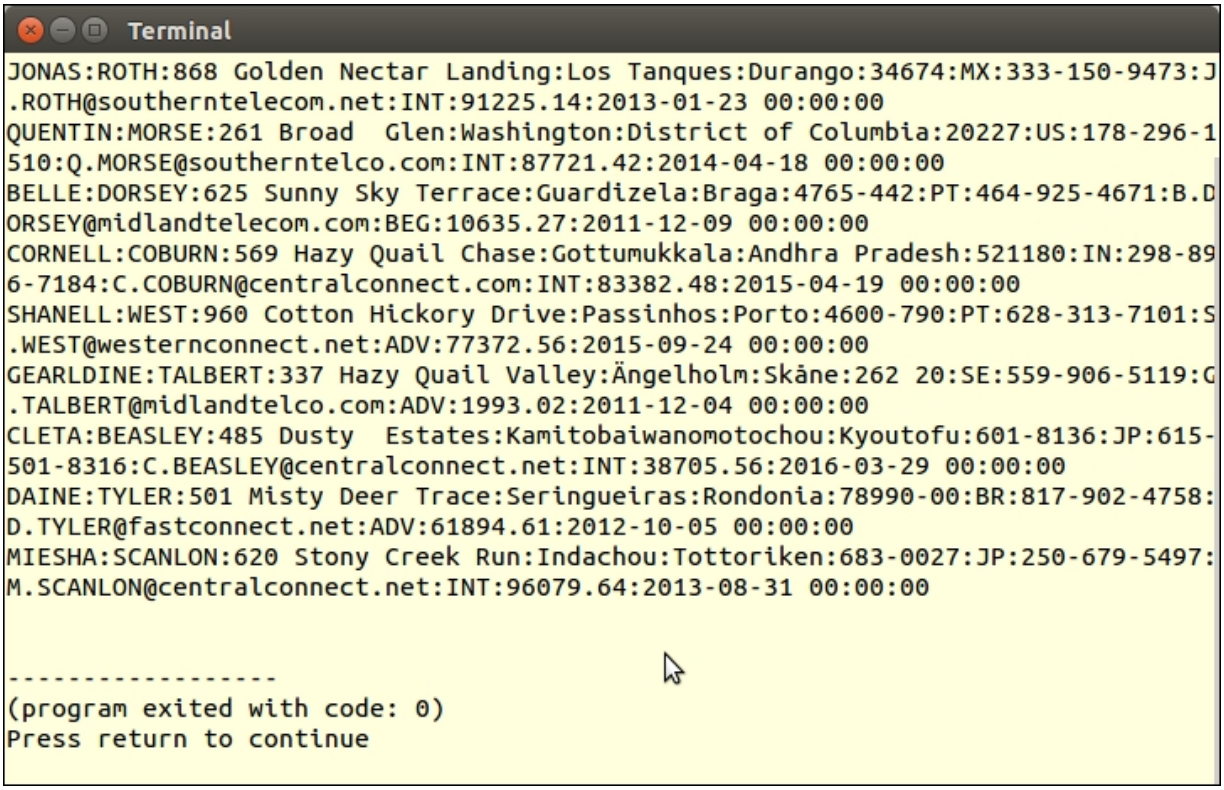
And finally, `'city'` draws its data from a lookup table, which also gives you data for the fields listed in the `'mapping'` parameter. You can then leave those keys undefined. Notice that you should also specify the column representing the primary key for the table:

```
'city' => ['source' => FakeData::SOURCE_TABLE,
'name' => 'world_city_data',
'idCol' => 'id',
'mapping' => [
'city' => 'city',
'state_province' => 'state_province',
'postal_code_prefix' => 'postal_code',
'iso2' => 'country']
],
  'state_province'=> [],
  'postal_code'  => [],
  'country'    => [],
];
```

You can then define the destination table, a `Connection` instance, and create the `FakeData` instance. A `foreach()` loop will suffice to display a given number of entries:

```
$destTableName = 'prospects';
$conn = new Connection(include DB_CONFIG_FILE);
$fake = new FakeData($conn, $mapping);
foreach ($fake->generateData(10) as $row) {
  echo implode(':', $row) . PHP_EOL;
}
```

The output, for 10 rows, would look something like this:



There's more...

Here is a summary of websites with various lists of data that could be of use when generating test data:

Type of Data	URL	Notes
Names	http://nameberry.com/	
	http://www.babynamewizard.com/international-names-lists-popular-names-from-around-the-world	
Raw Name Lists	http://deron.meranda.us/data/census-dist-female-first.txt	US female first names

Type of Data	URL	Notes
	http://deron.meranda.us/data/census-dist-male-first.txt	US male first names
	http://www.avss.ucsb.edu/NameFema.HTM	US female first names
	http://www.avss.ucsb.edu/namemal.htm	US male first names
Last Names	http://names.mongabay.com/data/1000.html	US surnames from census
	http://surname.sofeminine.co.uk/w/surnames/most-common-surnames-in-great-britain.html	British surnames
	https://gist.github.com/subodhghulaxe/8148971	List of US surnames in the form of a PHP array
	http://www.dutchgenealogy.nl/tng/surnames-all.php	Dutch surnames
	http://www.worldvitalrecords.com/browsesurnames.aspx?l=A	International surnames; just change the last letter(s) to get a list of names starting with that letter(s)

Type of Data	URL	Notes
Cities	http://www.travelgis.com/default.asp?framesrc=/cities/	World cities
	https://www.maxmind.com/en/free-world-cities-database	
	https://github.com/David-Haim/CountriesToCitiesJSON	
	http://www.fallingrain.com/world/index.html	
Postal Codes	https://boutell.com/zipcodes/	US only; includes cities, postal codes, latitude and longitude
	http://www.geonames.org/export/	International; city names, postal codes, EVERYTHING!; free download

Customizing sessions using session_start parameters

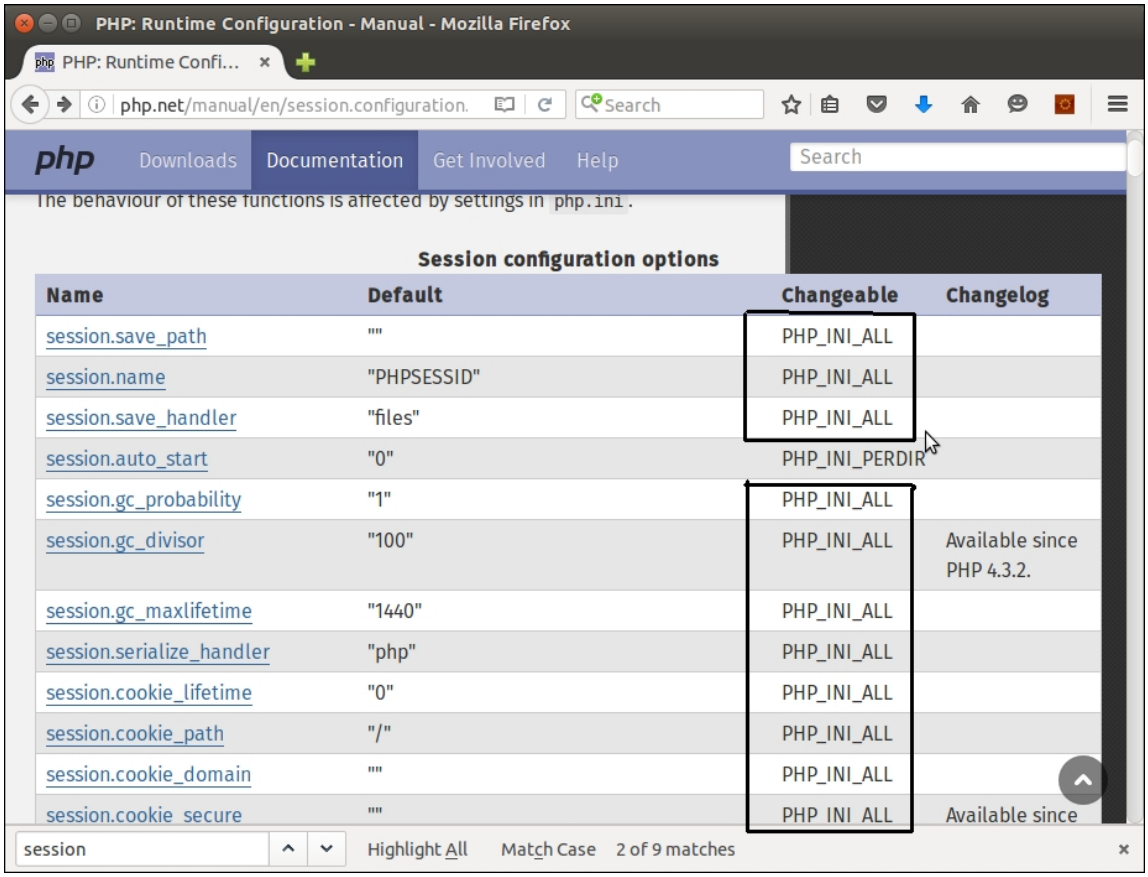
Up until PHP 7, in order to override `php.ini` settings for secure session management, you had to use a series of `ini_set()` commands. This approach is extremely annoying in that you also needed to know which settings were available, and being able to re-use the same settings in other applications was difficult. As of PHP 7, however, you can supply an array of parameters to the `session_start()` command, which immediately sets those values.

How to do it...

1. We start by developing an `Application\Security\SessOptions` class, which will hold session parameters and also have the ability to start the session. We also define a class constant in case invalid session options are passed:

```
namespace Application\Security;
use ReflectionClass;
use InvalidArgumentsException;
class SessOptions
{
  const ERROR_PARAMS = 'ERROR: invalid session
options';
```

2. Next we scan the list of `php.ini` session directives (documented at http://php.net/manual/en/session.configuration.php). We are specifically looking for directives that, in the `Changeable` column, are marked `PHP_INI_ALL`. Such directives can be overridden at runtime, and are thus available as arguments to `session_start()`:



3. We then define these as class constants, which will make this class more usable for development purposes. Most decent code editors will be able to scan the class and give you a list of constants, making it easy to manage session settings. Please note that not all settings are shown, in order to conserve space in the book:

```
const SESS_OP_NAME         = 'name';
const SESS_OP_LAZY_WRITE   = 'lazy_write';  //
AVAILABLE // SINCE PHP 7.0.0.
const SESS_OP_SAVE_PATH    = 'save_path';
const SESS_OP_SAVE_HANDLER = 'save_handler';
// etc.
```

4. We are then in a position to define the constructor, which accepts an array of `php.ini` session settings as an argument. We use `ReflectionClass` to get a list of class constants, and run the `$options` argument through a loop to confirm the setting is allowed. Also note the use of `array_flip()`, which flips keys and

values, so that the actual values for our class constants form the array key, and the name of the class constant becomes the value:

```
protected $options;
protected $allowed;
public function __construct(array $options)
{
  $reflect = new
ReflectionClass(get_class($this));
  $this->allowed = $reflect->getConstants();
  $this->allowed = array_flip($this->allowed);
  unset($this->allowed[self::ERROR_PARAMS]);
  foreach ($options as $key => $value) {
    if(!isset($this->allowed[$key])) {
      error_log(__METHOD__ . ':' .
self::ERROR_PARAMS);
      throw new InvalidArgumentsException(
      self::ERROR_PARAMS);
    }
  }
  $this->options = $options;
}
```

5. We then close with two more methods; one gives us outside access to the allowed parameters, while the other starts the session:

```
public function getAllowed()
{
  return $this->allowed;
}

public function start()
{
  session_start($this->options);
}
```

How it works...

Place all the code discussed in this recipe into a `SessOptions.php` file in the `Application\Security` directory. You can then define a calling

program called `chap_13_session_options.php` to test the new class, which sets up autoloading and uses the class:

```
<?php
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
use Application\Security\SessOptions;
```

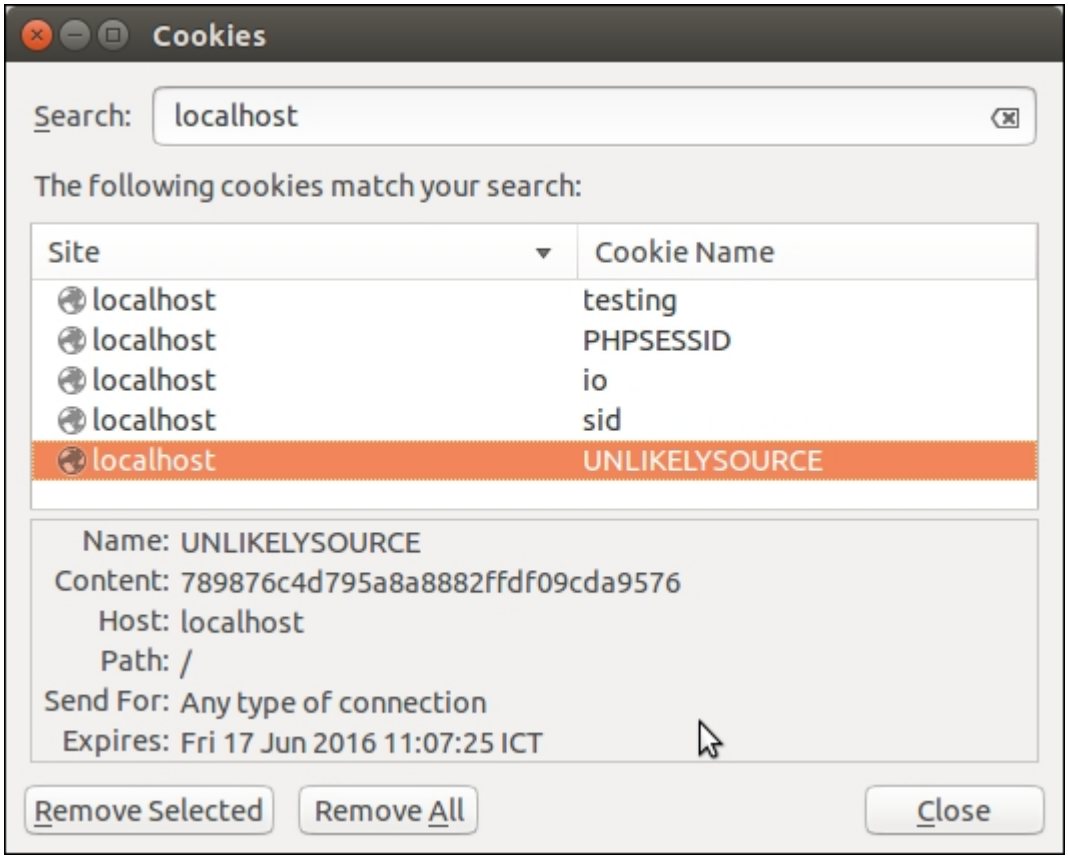
Next, define an array that uses the class constants as keys, with values as desired to manage the session. Note that in the example shown here, session information is stored in a subdirectory, `session`, which you need to create:

```
$options = [
  SessOptions::SESS_OP_USE_ONLY_COOKIES => 1,
  SessOptions::SESS_OP_COOKIE_LIFETIME => 300,
  SessOptions::SESS_OP_COOKIE_HTTPONLY => 1,
  SessOptions::SESS_OP_NAME => 'UNLIKELYSOURCE',
  SessOptions::SESS_OP_SAVE_PATH => __DIR__ .
'/session'
];
```

You can now create the `SessOptions` instance and run `start()` to start the session. You could use `phpinfo()` here to show some information on the session:

```
$sessOpt = new SessOptions($options);
$sessOpt->start();
$_SESSION['test'] = 'TEST';
phpinfo(INFO_VARIABLES);
```

If you look for information on cookies using your browser's developer tools, you will note the name is set to `UNLIKELYSOURCE` and the expiration time is 5 minutes from now:



If you do a scan of the session directory, you will see that the session information has been stored there:

```
ed@ed: ~/Desktop/Repos/php7_recipes/source/chapter13
ed@ed:~/Desktop/Repos/php7_recipes/source/chapter13$ ls -l session
total 4
-rw------- 1 ed ed 16 Jun 17 11:03 sess_789876c4d795a8a8882ffdf09cda9576
ed@ed:~/Desktop/Repos/php7_recipes/source/chapter13$
```

See also...

- For more information on session-related `php.ini` directives, see this summary: http://php.net/manual/en/session.configuration.php

Appendix A. Defining PSR-7 Classes

In this appendix, we will cover the following topics:

- Implementing PSR-7 value object classes
- Developing a PSR-7 Request class
- Defining a PSR-7 Response class

Introduction

PHP Standard Recommendation number 7 (**PSR-7**) defines a number of interfaces, but does not provide actual implementations. Accordingly, we need to define concrete code implementations in order to start creating custom middleware.

Implementing PSR-7 value object classes

In order to work with PSR-7 requests and responses, we first need to define a series of value objects. These are classes that represent logical objects used in web-based activities such as URIs, file uploads, and streaming request or response bodies.

Getting ready

The source code for the PSR-7 interfaces is available as a `Composer` package. It is considered a best practice to use `Composer` to manage external software, including PSR-7 interfaces.

How to do it...

1. First of all, go to the following URL to obtain the latest versions of the PSR-7 interface definitions: https://github.com/php-fig/http-message. The source code is also available. At the time of writing, the following definitions are available:

Interface	Extends	Notes	What the methods handle
`MessageInterface`		Defines methods common to HTTP messages	Headers, message body (that is, content), and protocol

---

Interface	Extends	Notes	What the methods handle
`RequestInterface`	`MessageInterface`	Represents requests generated by a client	The URI, HTTP method, and the request target
`ServerRequestInterface`	`RequestInterface`	Represents a request coming to a server from a client	Server and query parameters, cookies, uploaded files, and the parsed body
`ResponseInterface`	`MessageInterface`	Represents a response from the server to client	HTTP status code and reason
`StreamInterface`		Represents the data stream	Streaming behavior such as seek, tell, read, write, and so on

Interface	Extends	Notes	What the methods handle
`UriInterface`		Represents the URI	Scheme (that is, HTTP, HTTPS), host, port, username, password (that is, for FTP), query parameters, path, and fragment
`UploadedFileInterface`		Deals with uploaded files	File size, media type, moving the file, and filename

2. Unfortunately, we will need to create concrete classes that implement these interfaces in order to utilize PSR-7. Fortunately, the interface classes are extensively documented internally through a series of comments. We will start with a separate class that contains useful constants:

> **Tip**

Note that we take advantage of a new feature introduced in PHP 7 that allows us to define a constant as an array.

```
namespace Application\MiddleWare;
class Constants
{
  const HEADER_HOST   = 'Host';     // host
header
  const HEADER_CONTENT_TYPE = 'Content-Type';
  const HEADER_CONTENT_LENGTH = 'Content-Length';

  const METHOD_GET    = 'get';
  const METHOD_POST   = 'post';
  const METHOD_PUT    = 'put';
  const METHOD_DELETE = 'delete';
  const HTTP_METHODS  =
['get','put','post','delete'];

  const STANDARD_PORTS = [
    'ftp' => 21, 'ssh' => 22, 'http' => 80,
'https' => 443
  ];

  const CONTENT_TYPE_FORM_ENCODED =
    'application/x-www-form-urlencoded';
  const CONTENT_TYPE_MULTI_FORM   =
'multipart/form-data';
  const CONTENT_TYPE_JSON         =
'application/json';
  const CONTENT_TYPE_HAL_JSON     =
'application/hal+json';

  const DEFAULT_STATUS_CODE    = 200;
  const DEFAULT_BODY_STREAM    = 'php://input';
  const DEFAULT_REQUEST_TARGET = '/';

  const MODE_READ = 'r';
  const MODE_WRITE = 'w';

  // NOTE: not all error constants are shown to
conserve space
  const ERROR_BAD = 'ERROR: ';
  const ERROR_UNKNOWN = 'ERROR: unknown';

  // NOTE: not all status codes are shown here!
  const STATUS_CODES = [
    200 => 'OK',
```

```
      301 => 'Moved Permanently',
      302 => 'Found',
      401 => 'Unauthorized',
      404 => 'Not Found',
      405 => 'Method Not Allowed',
      418 => 'I_m A Teapot',
      500 => 'Internal Server Error',
    ];
}
```

> **Note**
>
> A complete list of HTTP status codes can be found here: https://tools.ietf.org/html/rfc7231#section-6.1.

3. Next, we will tackle classes that represent value objects used by other PSR-7 classes. For a start, here is the class that represents a URI. In the constructor, we accept a URI string as an argument, and break it down into its component parts using the `parse_url()` function:

```
namespace Application\MiddleWare;
use InvalidArgumentException;
use Psr\Http\Message\UriInterface;
class Uri implements UriInterface
{
  protected $uriString;
  protected $uriParts = array();

  public function __construct($uriString)
  {
    $this->uriParts = parse_url($uriString);
    if (!$this->uriParts) {
      throw new InvalidArgumentException(
        Constants::ERROR_INVALID_URI);
    }
    $this->uriString = $uriString;
  }
```

> **Note**

URI stands for **Uniform Resource Indicator**. This is what you would see at the top of your browser when making a request. For more information on what comprises a URI, have a look at http://tools.ietf.org/html/rfc3986.

4. Following the constructor, we define methods to access the component parts of the URI. The **scheme** represents a PHP wrapper (that is, HTTP, FTP, and so on):

```
public function getScheme()
{
  return strtolower($this->uriParts['scheme']) ??
'';
}
```

5. The **authority** represents the username (if present), the host, and optionally the port number:

```
public function getAuthority()
{
  $val = '';
  if (!empty($this->getUserInfo()))
  $val .= $this->getUserInfo() . '@';
  $val .= $this->uriParts['host'] ?? '';
  if (!empty($this->uriParts['port']))
  $val .= ':' . $this->uriParts['port'];
  return $val;
}
```

6. **User info** represents the username (if present) and optionally the password. An example of when a password is used is when accessing an FTP website such as `ftp://username:password@website.com:/path`:

```
public function getUserInfo()
{
  if (empty($this->uriParts['user'])) {
    return '';
  }
  $val = $this->uriParts['user'];
  if (!empty($this->uriParts['pass']))
    $val .= ':' . $this->uriParts['pass'];
  return $val;
}
```

7. **Host** is the DNS address included in the URI:

```
public function getHost()
{
  if (empty($this->uriParts['host'])) {
    return '';
  }
  return strtolower($this->uriParts['host']);
}
```

8. **Port** is the HTTP port, if present. You will note if a port is listed in our `STANDARD_PORTS` constant, the return value is `NULL`, according to the requirements of PSR-7:

```
public function getPort()
{
  if (empty($this->uriParts['port'])) {
      return NULL;
  } else {
      if ($this->getScheme()) {
          if ($this->uriParts['port'] ==
              Constants::STANDARD_PORTS[$this-
>getScheme()]) {
              return NULL;
          }
      }
      return (int) $this->uriParts['port'];
  }
}
```

9. **Path** is the part of the URI that follows the DNS address. According to PSR-7, this must be encoded. We use the `rawurlencode()` PHP function as it is compliant with RFC 3986. We cannot just encode the entire path, however, as the path separator (that is, `/`) would also get encoded! Accordingly, we need to first break it up using `explode()`, encode the parts, and then reassemble it:

```
public function getPath()
{
  if (empty($this->urlParts['path'])) {
    return '';
  }
  return implode('/', array_map("rawurlencode",
explode('/', $this->urlParts['path'])));
}
```

10. Next, we define a method to retrieve the `query` string (that is, from `$_GET`). These too must be URL-encoded. First, we define `getQueryParams()`, which breaks the query string into an associative array. You will note the reset option in case we wish to refresh the query parameters. We then define `getQuery()`, which takes the array and produces a proper URL-encoded string:

```
public function getQueryParams($reset = FALSE)
{
  if ($this->queryParams && !$reset) {
    return $this->queryParams;
  }
  $this->queryParams = [];
  if (!empty($this->uriParts['query'])) {
    foreach (explode('&', $this-
>uriParts['query']) as $keyPair) {
      list($param,$value) =
explode('=',$keyPair);
      $this->queryParams[$param] = $value;
    }
  }
  return $this->queryParams;
}

public function getQuery()
{
  if (!$this->getQueryParams()) {
    return '';
  }
  $output = '';
  foreach ($this->getQueryParams() as $key =>
$value) {
    $output .= rawurlencode($key) . '='
    . rawurlencode($value) . '&';
  }
  return substr($output, 0, -1);
}
```

11. After that, we provide a method to return the `fragment` (that is, a `#` in the URI), and any part following it:

```
public function getFragment()
{
  if (empty($this->urlParts['fragment'])) {
    return '';
  }
  return rawurlencode($this-
```

```
>urlParts['fragment']);
}
```

12. Next, we define a series of `withXXX()` methods, which match the `getXXX()` methods described above. These methods are designed to add, replace, or remove properties associated with the request class (scheme, authority, user info, and so on). In addition, these methods return the current instance that allows us to use these methods in a series of successive calls (often referred to as the **fluent interface**). We start with `withScheme()`:

> ## Note
>
> You will note that an empty argument, according to PSR-7, signals the removal of that property. You will also note that we do not allow a scheme that does not match what is defined in our `Constants::STANDARD_PORTS` array.

```
public function withScheme($scheme)
{
  if (empty($scheme) && $this->getScheme()) {
      unset($this->uriParts['scheme']);
  } else {
      if
(isset(STANDARD_PORTS[strtolower($scheme)])) {
          $this->uriParts['scheme'] = $scheme;
      } else {
          throw new
InvalidArgumentException(Constants::ERROR_BAD .
__METHOD__);
      }
  }
  return $this;
}
```

13. We then apply similar logic to methods that overwrite, add, or replace the user info, host, port, path, query, and fragment. Note that the `withQuery()` method resets the query parameters array. `withHost()`, `withPort()`, `withPath()`, and `withFragment()` use the same logic, but are not shown to conserve space:

```
public function withUserInfo($user, $password =
null)
{
  if (empty($user) && $this->getUserInfo()) {
      unset($this->uriParts['user']);
  } else {
      $this->urlParts['user'] = $user;
      if ($password) {
          $this->urlParts['pass'] = $password;
      }
  }
  return $this;
}
// Not shown:
withHost(),withPort(),withPath(),withFragment()

public function withQuery($query)
{
  if (empty($query) && $this->getQuery()) {
      unset($this->uriParts['query']);
  } else {
      $this->uriParts['query'] = $query;
  }
  // reset query params array
  $this->getQueryParams(TRUE);
  return $this;
}
```

14. Finally, we wrap up the `Application\MiddleWare\Uri` class with `__toString()`, which, when the object is used in a string context, returns a proper URI, assembled from `$uriParts`. We also define a convenience method, `getUriString()`, that simply calls `__toString()`:

```
public function __toString()
{
    $uri = ($this->getScheme())
      ? $this->getScheme() . '://' : '';
```

15. If the `authority` URI part is present, we add it. `authority` includes the user information, host, and port. Otherwise, we just append `host` and `port`:

```
if ($this->getAuthority()) {
    $uri .= $this->getAuthority();
} else {
    $uri .= ($this->getHost()) ? $this->getHost()
```

```
: '';
    $uri .= ($this->getPort())
      ? ':' . $this->getPort() : '';
}
```

16. Before adding `path`, we first check whether the first character is `/`. If not, we need to add this separator. We then add `query` and `fragment`, if present:

```
$path = $this->getPath();
if ($path) {
    if ($path[0] != '/') {
        $uri .= '/' . $path;
    } else {
        $uri .= $path;
    }
}
$uri .= ($this->getQuery())
  ? '?' . $this->getQuery() : '';
$uri .= ($this->getFragment())
  ? '#' . $this->getFragment() : '';
return $uri;
}

public function getUriString()
{
  return $this->__toString();
}

}
```

> ## Note
>
> Note the use of string dereferencing (that is, `$path[0]`), now part of PHP 7.

17. Next, we turn our attention to a class that represents the body of the message. As it is not known how large the body might be, PSR-7 recommends that the body should be treated as a **stream**. A stream is a resource that allows access to input and output sources in a linear fashion. In PHP, all file commands operate on top of the `Streams` sub-system, so this is a natural fit. PSR-7 formalizes this by

way of `Psr\Http\Message\StreamInterface` that defines such methods as `read()`, `write()`, `seek()`, and so on. We now present `Application\MiddleWare\Stream` that we can use to represent the body of incoming or outgoing requests and/or responses:

```
namespace Application\MiddleWare;
use SplFileInfo;
use Throwable;
use RuntimeException;
use Psr\Http\Message\StreamInterface;
class Stream implements StreamInterface
{
  protected $stream;
  protected $metadata;
  protected $info;
```

18. In the constructor, we open the stream using a simple `fopen()` command. We then use `stream_get_meta_data()` to get information on the stream. For other details, we create an `SplFileInfo` instance:

```
public function __construct($input, $mode =
self::MODE_READ)
{
  $this->stream = fopen($input, $mode);
  $this->metadata = stream_get_meta_data($this-
>stream);
  $this->info = new SplFileInfo($input);
}
```

> ## Note
>
> The reason why we chose `fopen()` over the more modern `SplFileObject` is that the latter does not allow direct access to the inner file resource object, and is therefore useless for this application.

19. We include two convenience methods that provide access to the resource, as well as access to the `SplFileInfo` instance:

```
public function getStream()
{
```

```
    return $this->stream;
  }

  public function getInfo()
  {
    return $this->info;
  }
```

20. Next, we define low-level core streaming methods:

```
  public function read($length)
  {
    if (!fread($this->stream, $length)) {
        throw new RuntimeException(self::ERROR_BAD
. __METHOD__);
    }
  }
  public function write($string)
  {
    if (!fwrite($this->stream, $string)) {
        throw new RuntimeException(self::ERROR_BAD
. __METHOD__);
    }
  }
  public function rewind()
  {
    if (!rewind($this->stream)) {
        throw new RuntimeException(self::ERROR_BAD
. __METHOD__);
    }
  }
  public function eof()
  {
    return eof($this->stream);
  }
  public function tell()
  {
    try {
        return ftell($this->stream);
    } catch (Throwable $e) {
        throw new RuntimeException(self::ERROR_BAD
. __METHOD__);
    }
  }
  public function seek($offset, $whence = SEEK_SET)
  {
    try {
        fseek($this->stream, $offset, $whence);
    } catch (Throwable $e) {
        throw new RuntimeException(self::ERROR_BAD
```

```
. __METHOD__);
    }
  }
  public function close()
  {
    if ($this->stream) {
      fclose($this->stream);
    }
  }
  public function detach()
  {
    return $this->close();
  }
```

21. We also need to define informational methods that tell us about the stream:

```
  public function getMetadata($key = null)
  {
    if ($key) {
        return $this->metadata[$key] ?? NULL;
    } else {
        return $this->metadata;
    }
  }
  public function getSize()
  {
    return $this->info->getSize();
  }
  public function isSeekable()
  {
    return boolval($this->metadata['seekable']);
  }
  public function isWritable()
  {
    return $this->stream->isWritable();
  }
  public function isReadable()
  {
    return $this->info->isReadable();
  }
```

22. Following PSR-7 guidelines, we then define `getContents()` and `__toString()` in order to dump the contents of the stream:

```
  public function __toString()
  {
    $this->rewind();
    return $this->getContents();
```

```
    }

    public function getContents()
    {
      ob_start();
      if (!fpassthru($this->stream)) {
        throw new RuntimeException(self::ERROR_BAD .
    __METHOD__);
      }
      return ob_get_clean();
    }
    }
```

23. An important variation of the `Stream` class shown previously is `TextStream` that is designed for situations where the body is a string (that is, an array encoded as JSON) rather than a file. As we need to make absolutely certain that the incoming `$input` value is of the string data type, we invoke PHP 7 strict types just after the opening tag. We also identify a `$pos` property (that is, position) that will emulate a file pointer, but instead point to a position within the string:

```
<?php
declare(strict_types=1);
namespace Application\MiddleWare;
use Throwable;
use RuntimeException;
use SplFileInfo;
use Psr\Http\Message\StreamInterface;

class TextStream implements StreamInterface
{
  protected $stream;
  protected $pos = 0;
```

24. Most of the methods are quite simple and self-explanatory. The `$stream` property is the input string:

```
public function __construct(string $input)
{
  $this->stream = $input;
}
public function getStream()
{
  return $this->stream;
}
  public function getInfo()
```

```
{
  return NULL;
}
public function getContents()
{
  return $this->stream;
}
public function __toString()
{
  return $this->getContents();
}
public function getSize()
{
  return strlen($this->stream);
}
public function close()
{
  // do nothing: how can you "close" string???
}
public function detach()
{
  return $this->close();  // that is, do nothing!
}
```

25. To emulate streaming behavior, `tell()`, `eof()`, `seek()`, and so on, work with `$pos`:

```
public function tell()
{
  return $this->pos;
}
public function eof()
{
  return ($this->pos == strlen($this->stream));
}
public function isSeekable()
{
  return TRUE;
}
public function seek($offset, $whence = NULL)
{
  if ($offset < $this->getSize()) {
      $this->pos = $offset;
  } else {
      throw new RuntimeException(
        Constants::ERROR_BAD . __METHOD__);
  }
}
public function rewind()
```

```
{
  $this->pos = 0;
}
public function isWritable()
{
  return TRUE;
}
```

26. The `read()` and `write()` methods work with `$pos` and substrings:

```
public function write($string)
{
  $temp = substr($this->stream, 0, $this->pos);
  $this->stream = $temp . $string;
  $this->pos = strlen($this->stream);
}

public function isReadable()
{
  return TRUE;
}
public function read($length)
{
  return substr($this->stream, $this->pos,
$length);
}
public function getMetadata($key = null)
{
  return NULL;
}

}
```

27. The last of the value objects to be presented is `Application\MiddleWare\UploadedFile`. As with the other classes, we first define properties that represent aspects of a file upload:

```
namespace Application\MiddleWare;
use RuntimeException;
use InvalidArgumentException;
use Psr\Http\Message\UploadedFileInterface;
class UploadedFile implements
UploadedFileInterface
{

  protected $field;   // original name of file
upload field
  protected $info;    // $_FILES[$field]
```

```
  protected $randomize;
  protected $movedName = '';
```

28. In the constructor, we allow the definition of the name attribute of the file upload form field, as well as the corresponding array in `$_FILES`. We add the last parameter to signal whether or not we want the class to generate a new random filename once the uploaded file is confirmed:

```
public function __construct($field, array $info,
$randomize = FALSE)
{
  $this->field = $field;
  $this->info = $info;
  $this->randomize = $randomize;
}
```

29. Next, we create a `Stream` class instance for the temporary or moved file:

```
public function getStream()
{
  if (!$this->stream) {
      if ($this->movedName) {
          $this->stream = new Stream($this-
>movedName);
      } else {
          $this->stream = new
Stream($info['tmp_name']);
      }
  }
  return $this->stream;
}
```

30. The `moveTo()` method performs the actual file movement. Note the extensive series of safety checks to help prevent an injection attack. If randomize is not enabled, we use the original user-supplied filename:

```
public function moveTo($targetPath)
{
  if ($this->moved) {
      throw new
Exception(Constants::ERROR_MOVE_DONE);
  }
  if (!file_exists($targetPath)) {
      throw new
```

```
InvalidArgumentException(Constants::ERROR_BAD_DIR
);
  }
  $tempFile = $this->info['tmp_name'] ?? FALSE;
  if (!$tempFile || !file_exists($tempFile)) {
      throw new
Exception(Constants::ERROR_BAD_FILE);
  }
  if (!is_uploaded_file($tempFile)) {
      throw new
Exception(Constants::ERROR_FILE_NOT);
  }
  if ($this->randomize) {
      $final = bin2hex(random_bytes(8)) . '.txt';
  } else {
      $final = $this->info['name'];
  }
  $final = $targetPath . '/' . $final;
  $final = str_replace('//', '/', $final);
  if (!move_uploaded_file($tempFile, $final)) {
      throw new
RuntimeException(Constants::ERROR_MOVE_UNABLE);
  }
  $this->movedName = $final;
  return TRUE;
}
```

31. We then provide access to the other parameters returned in `$_FILES` from the `$info` property. Please note that the return values from `getClientFilename()` and `getClientMediaType()` should be considered untrusted, as they originate from the outside. We also add a method to return the moved filename:

```
public function getMovedName()
{
  return $this->movedName ?? NULL;
}
public function getSize()
{
  return $this->info['size'] ?? NULL;
}
public function getError()
{
  if (!$this->moved) {
      return UPLOAD_ERR_OK;
  }
  return $this->info['error'];
}
```

```
    public function getClientFilename()
    {
      return $this->info['name'] ?? NULL;
    }
    public function getClientMediaType()
    {
      return $this->info['type'] ?? NULL;
    }

    }
```

How it works...

First of all, go to https://github.com/php-fig/http-message/tree/master/src, the GitHub repository for the PSR-7 interfaces, and download them. Create a directory called `Psr/Http/Message` in `/path/to/source` and places the files there. Alternatively, you can visit https://packagist.org/packages/psr/http-message and install the source code using `Composer`. (For instructions on how to obtain and use `Composer`, you can visit https://getcomposer.org/.)

Then, go ahead and define the classes discussed previously, summarized in this table:

Class	**Steps discussed in**
`Application\MiddleWare\Constants`	2
`Application\MiddleWare\Uri`	3 to 16
`Application\MiddleWare\Stream`	17 to 22

Class	Steps discussed in
`Application\MiddleWare\TextStream`	23 to 26
`Application\MiddleWare\UploadedFile`	27 to 31

Next, define a `chap_09_middleware_value_objects_uri.php` calling program that implements autoloading and uses the appropriate classes. Please note that if you use `Composer`, unless otherwise instructed, it will create a folder called `vendor`. `Composer` also adds its own autoloader, which you are free to use here:

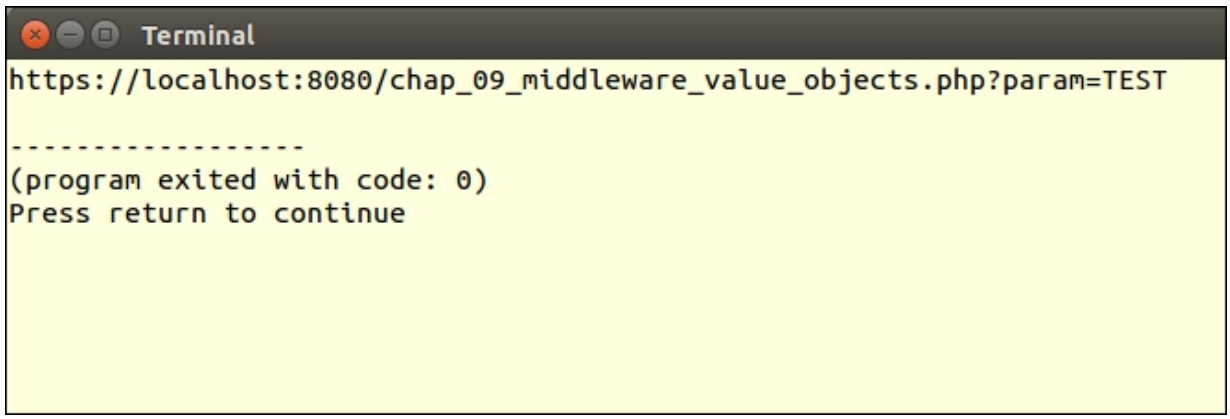
```
<?php
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\MiddleWare\Uri;
```

You can then create a `Uri` instance and use the `with` methods to add parameters. You can then echo the `Uri` instance directly as `__toString()` is defined:

```
$uri = new Uri();
$uri->withScheme('https')
    ->withHost('localhost')
    ->withPort('8080')
    -
>withPath('chap_09_middleware_value_objects_uri.php')
    ->withQuery('param=TEST');

echo $uri;
```

Here is the expected result:



Next, create a directory called `uploads` from `/path/to/source/for/this/chapter`. Go ahead and define another calling program, `chap_09_middleware_value_objects_file_upload.php`, that sets up autoloading and uses the appropriate classes:

```
<?php
define('TARGET_DIR', __DIR__ . '/uploads');
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\MiddleWare\UploadedFile;
```

Inside a `try...catch` block, check to see whether any files were uploaded. If so, loop through `$_FILES` and create `UploadedFile` instances where `tmp_name` is set. You can then use the `moveTo()` method to move the files to `TARGET_DIR`:

```
try {
    $message = '';
    $uploadedFiles = array();
    if (isset($_FILES)) {
        foreach ($_FILES as $key => $info) {
          if ($info['tmp_name']) {
              $uploadedFiles[$key] = new
UploadedFile($key, $info, TRUE);
              $uploadedFiles[$key]-
>moveTo(TARGET_DIR);
          }
        }
    }
} catch (Throwable $e) {
    $message =  $e->getMessage();
```

```
}
?>
```

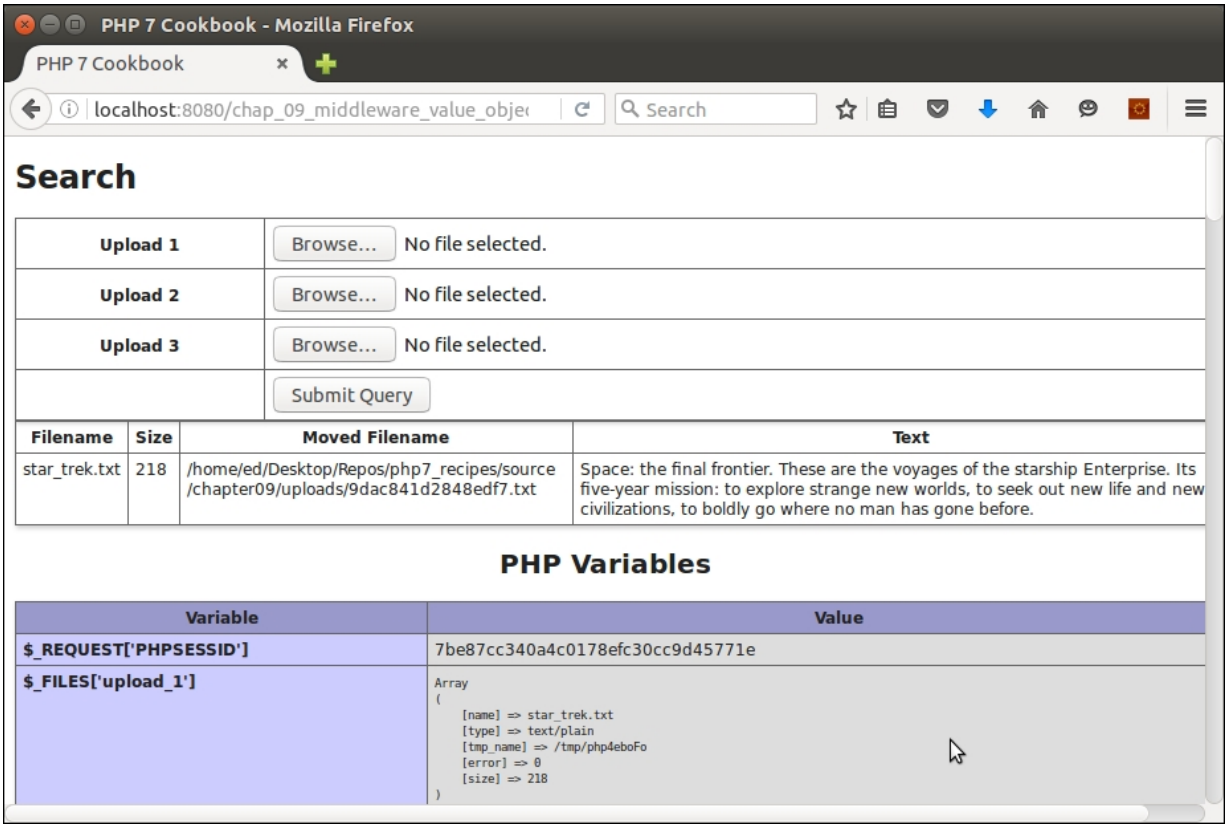
In the view logic, display a simple file upload form. You could also use `phpinfo()` to display information about what was uploaded:

```
<form name="search" method="post" enctype="<?=
Constants::CONTENT_TYPE_MULTI_FORM ?>">
<table class="display" cellspacing="0" width="100%">
    <tr><th>Upload 1</th><td><input type="file"
name="upload_1" /></td></tr>
    <tr><th>Upload 2</th><td><input type="file"
name="upload_2" /></td></tr>
    <tr><th>Upload 3</th><td><input type="file"
name="upload_3" /></td></tr>
    <tr><th> </th><td><input type="submit" />
</td></tr>
</table>
</form>
<?= ($message) ? '<h1>' . $message . '</h1>' : ''; ?>
```

Next, if there were any uploaded files, you can display information on each one. You can also use `getStream()` followed by `getContents()` to display each file (assuming you're using short text files):

```
<?php if ($uploadedFiles) : ?>
<table class="display" cellspacing="0" width="100%">
    <tr>
        <th>Filename</th><th>Size</th>
      <th>Moved Filename</th><th>Text</th>
    </tr>
    <?php foreach ($uploadedFiles as $obj) : ?>
        <?php if ($obj->getMovedName()) : ?>
        <tr>
            <td><?= htmlspecialchars($obj-
>getClientFilename()) ?></td>
            <td><?= $obj->getSize() ?></td>
            <td><?= $obj->getMovedName() ?></td>
            <td><?= $obj->getStream()->getContents()
?></td>
        </tr>
        <?php endif; ?>
    <?php endforeach; ?>
</table>
<?php endif; ?>
<?php phpinfo(INFO_VARIABLES); ?>
```

Here is how the output might appear:



See also

- For more information on PSR, please have a look at https://en.wikipedia.org/wiki/PHP_Standard_Recommendation
- For information on PSR-7 specifically, here is the official description: http://www.php-fig.org/psr/psr-7/
- For information on PHP streams, take a look at http://php.net/manual/en/book.stream.php

Developing a PSR-7 Request class

One of the key characteristics of PSR-7 middleware is the use of **Request** and **Response** classes. When applied, this enables different blocks of software to perform together without sharing any specific knowledge between them. In this context, a request class should encompass all aspects of the original user request, including such items as browser settings, the original URL requested, parameters passed, and so forth.

How to do it...

1. First, be sure to define classes to represent the `Uri`, `Stream`, and `UploadedFile` value objects, as described in the previous recipe.
2. Now we are ready to define the core `Application\MiddleWare\Message` class. This class consumes `Stream` and `Uri` and implements `Psr\Http\Message\MessageInterface`. We first define properties for the key value objects, including those representing the message body (that is, a `StreamInterface` instance), version, and HTTP headers:

```
namespace Application\MiddleWare;
use Psr\Http\Message\ {
  MessageInterface,
  StreamInterface,
  UriInterface
};
class Message implements MessageInterface
{
  protected $body;
  protected $version;
  protected $httpHeaders = array();
```

3. Next, we have the `getBody()` method that represents a `StreamInterface` instance. A companion method, `withBody()`,

returns the current `Message` instance and allows us to overwrite the current value of `body`:

```
public function getBody()
{
  if (!$this->body) {
      $this->body = new
Stream(self::DEFAULT_BODY_STREAM);
  }
  return $this->body;
}
public function withBody(StreamInterface
$body)
{
  if (!$body->isReadable()) {
      throw new
InvalidArgumentException(self::ERROR_BODY_UNRE
ADABLE);
  }
  $this->body = $body;
  return $this;
}
```

4. PSR-7 recommends that headers should be viewed as case-insensitive. Accordingly, we define a `findHeader()` method (not directly defined by `MessageInterface`) that locates a header using `stripos()`:

```
protected function findHeader($name)
{
  $found = FALSE;
  foreach (array_keys($this->getHeaders()) as
$header) {
    if (stripos($header, $name) !== FALSE) {
        $found = $header;
        break;
    }
  }
  return $found;
}
```

5. The next method, not defined by PSR-7, is designed to populate the `$httpHeaders` property. This property is assumed to be an associative array where the key is the header, and the value is the string representing the header value. If there is more than

one value, additional values separated by commas are appended to the string. There is an excellent `apache_request_headers()` PHP function from the Apache extension that produces headers if they are not already available in `$httpHeaders`:

```
protected function getHttpHeaders()
{
  if (!$this->httpHeaders) {
      if
(function_exists('apache_request_headers')) {
          $this->httpHeaders =
apache_request_headers();
      } else {
          $this->httpHeaders = $this-
>altApacheReqHeaders();
      }
  }
  return $this->httpHeaders;
}
```

6. If `apache_request_headers()` is not available (that is, the Apache extension is not enabled), we provide an alternative, `altApacheReqHeaders()`:

```
protected function altApacheReqHeaders()
{
  $headers = array();
  foreach ($_SERVER as $key => $value) {
    if (stripos($key, 'HTTP_') !== FALSE) {
        $headerKey = str_ireplace('HTTP_', '',
$key);
        $headers[$this-
>explodeHeader($headerKey)] = $value;
    } elseif (stripos($key, 'CONTENT_') !==
FALSE) {
        $headers[$this->explodeHeader($key)] =
$value;
    }
  }
  return $headers;
}
protected function explodeHeader($header)
{
  $headerParts = explode('_', $header);
  $headerKey = ucwords(implode(' ',
```

```
strtolower($headerParts)));
  return str_replace(' ', '-', $headerKey);
}
```

7. Implementing `getHeaders()` (required in PSR-7) is now a trivial loop through the `$httpHeaders` property produced by the `getHttpHeaders()` method discussed in step 4:

```
public function getHeaders()
{
  foreach ($this->getHttpHeaders() as $key =>
$value) {
    header($key . ': ' . $value);
  }
}
```

8. Again, we provide a series of `with` methods designed to overwrite or replace headers. Since there can be many headers, we also have a method that adds to the existing set of headers. The `withoutHeader()` method is used to remove a header instance. Notice the consistent use of `findHeader()`, mentioned in the previous step, to allow for case-insensitive handling of headers:

```
public function withHeader($name, $value)
{
  $found = $this->findHeader($name);
  if ($found) {
      $this->httpHeaders[$found] = $value;
  } else {
      $this->httpHeaders[$name] = $value;
  }
  return $this;
}

public function withAddedHeader($name, $value)
{
  $found = $this->findHeader($name);
  if ($found) {
      $this->httpHeaders[$found] .= $value;
  } else {
      $this->httpHeaders[$name] = $value;
  }
  return $this;
}
```

```
public function withoutHeader($name)
{
  $found = $this->findHeader($name);
  if ($found) {
      unset($this->httpHeaders[$found]);
  }
  return $this;
}
```

9. We then provide a series of useful header-related methods to confirm a header exists, retrieve a single header line, and retrieve a header in array form, as per PSR-7:

```
public function hasHeader($name)
{
  return boolval($this->findHeader($name));
}

public function getHeaderLine($name)
{
  $found = $this->findHeader($name);
  if ($found) {
      return $this->httpHeaders[$found];
  } else {
      return '';
  }
}

public function getHeader($name)
{
  $line = $this->getHeaderLine($name);
  if ($line) {
      return explode(',', $line);
  } else {
      return array();
  }
}
```

10. Finally, to round off header handling, we present `getHeadersAsString` that produces a single header string with the headers separated by `\r\n` for direct use with PHP stream contexts:

```
public function getHeadersAsString()
{
```

```
    $output = '';
    $headers = $this->getHeaders();
    if ($headers && is_array($headers)) {
        foreach ($headers as $key => $value) {
          if ($output) {
              $output .= "\r\n" . $key . ': ' .
$value;
          } else {
              $output .= $key . ': ' . $value;
          }
        }
    }
    return $output;
}
```

11. Still within the `Message` class, we now turn our attention to version handling. According to PSR-7, the return value for the protocol version (that is, HTTP/1.1) should only be the numerical part. For this reason, we also provide `onlyVersion()` that strips off any non-digit character, allowing periods:

```
public function getProtocolVersion()
{
    if (!$this->version) {
        $this->version = $this-
>onlyVersion($_SERVER['SERVER_PROTOCOL']);
    }
    return $this->version;
}

public function withProtocolVersion($version)
{
    $this->version = $this-
>onlyVersion($version);
    return $this;
}

protected function onlyVersion($version)
{
    if (!empty($version)) {
        return preg_replace('/[^0-9\.]/', '',
$version);
    } else {
        return NULL;
    }
}
```

```
}
```

12. Finally, almost as an anticlimax, we are ready to define our `Request` class. It must be noted here, however, that we need to consider both out-bound as well as in-bound requests. That is to say, we need a class to represent an outgoing request a client will make to a server, as well as a request *received* from a client by a server. Accordingly, we provide `Application\MiddleWare\Request` (requests a client will make to a server), and `Application\MiddleWare\ServerRequest` (requests received from a client by a server). The good news is that most of our work has already been done: notice that our `Request` class extends `Message`. We also provide properties to represent the URI and HTTP method:

```
namespace Application\MiddleWare;

use InvalidArgumentException;
use Psr\Http\Message\ { RequestInterface,
StreamInterface, UriInterface };

class Request extends Message implements
RequestInterface
{
  protected $uri;
  protected $method; // HTTP method
  protected $uriObj; //
Psr\Http\Message\UriInterface instance
```

13. All properties in the constructor default to `NULL`, but we leave open the possibility of defining the appropriate arguments right away. We use the inherited `onlyVersion()` method to sanitize the version. We also define `checkMethod()` to make sure any method supplied is on our list of supported HTTP methods, defined as a constant array in `Constants`:

```
public function __construct($uri = NULL,
                            $method = NULL,
                            StreamInterface
$body = NULL,
                            $headers = NULL,
                            $version = NULL)
```

```
{
  $this->uri = $uri;
  $this->body = $body;
  $this->method = $this->checkMethod($method);
  $this->httpHeaders = $headers;
  $this->version = $this-
>onlyVersion($version);
}
protected function checkMethod($method)
{
  if (!$method === NULL) {
      if (!in_array(strtolower($method),
Constants::HTTP_METHODS)) {
          throw new
InvalidArgumentException(Constants::ERROR_HTTP
_METHOD);
      }
  }
  return $method;
}
```

14. We are going to interpret the request target as the originally requested URI in the form of a string. Bear in mind that our `Uri` class has methods that will parse this into its component parts, hence our provision of the `$uriObj` property. In the case of `withRequestTarget()`, notice that we run `getUri()` that performs the aforementioned parsing process:

```
public function getRequestTarget()
{
  return $this->uri ??
Constants::DEFAULT_REQUEST_TARGET;
}

public function
withRequestTarget($requestTarget)
{
  $this->uri = $requestTarget;
  $this->getUri();
  return $this;
}
```

15. Our `get` and `with` methods, which represent the HTTP method, reveal no surprises. We use `checkMethod()`, used in the

constructor as well, to ensure the method matches those we plan to support:

```
public function getMethod()
{
  return $this->method;
}

public function withMethod($method)
{
  $this->method = $this->checkMethod($method);
  return $this;
}
```

16. Finally, we have a `get` and `with` method for the URI. As mentioned in step 14, we retain the original request string in the `$uri` property and the newly parsed `Uri` instance in `$uriObj`. Note the extra flag to preserve any existing `Host` header:

```
public function getUri()
{
  if (!$this->uriObj) {
      $this->uriObj = new Uri($this->uri);
  }
  return $this->uriObj;
}

public function withUri(UriInterface $uri,
$preserveHost = false)
{
  if ($preserveHost) {
    $found = $this-
>findHeader(Constants::HEADER_HOST);
    if (!$found && $uri->getHost()) {
      $this-
>httpHeaders[Constants::HEADER_HOST] = $uri-
>getHost();
    }
  } elseif ($uri->getHost()) {
      $this-
>httpHeaders[Constants::HEADER_HOST] = $uri-
>getHost();
  }
  $this->uri = $uri->__toString();
  return $this;
```

```
    }
}
```

17. The `ServerRequest` class extends `Request` and provides additional functionality to retrieve information of interest to a server handling an incoming request. We start by defining properties that will represent incoming data read from the various PHP `$_ super-globals` (that is, `$_SERVER`, `$_POST`, and so on):

```
namespace Application\MiddleWare;
use Psr\Http\Message\ {
ServerRequestInterface, UploadedFileInterface
} ;

class ServerRequest extends Request implements
ServerRequestInterface
{

  protected $serverParams;
  protected $cookies;
  protected $queryParams;
  protected $contentType;
  protected $parsedBody;
  protected $attributes;
  protected $method;
  protected $uploadedFileInfo;
  protected $uploadedFileObjs;
```

18. We then define a series of getters to pull super-global information. We do not show everything, to conserve space:

```
public function getServerParams()
{
  if (!$this->serverParams) {
      $this->serverParams = $_SERVER;
  }
  return $this->serverParams;
}
// getCookieParams() reads $_COOKIE
// getQueryParams() reads $_GET
// getUploadedFileInfo() reads $_FILES

public function getRequestMethod()
{
  $method = $this->getServerParams()
```

```
['REQUEST_METHOD'] ?? '';
  $this->method = strtolower($method);
  return $this->method;
}

public function getContentType()
{
  if (!$this->contentType) {
      $this->contentType = $this-
>getServerParams()['CONTENT_TYPE'] ?? '';
      $this->contentType = strtolower($this-
>contentType);
  }
  return $this->contentType;
}
```

19. As uploaded files are supposed to be represented as independent `UploadedFile` objects (presented in the previous recipe), we also define a method that takes `$uploadedFileInfo` and creates `UploadedFile` objects:

```
public function getUploadedFiles()
{
  if (!$this->uploadedFileObjs) {
      foreach ($this->getUploadedFileInfo() as
$field => $value) {
        $this->uploadedFileObjs[$field] = new
UploadedFile($field, $value);
      }
  }
  return $this->uploadedFileObjs;
}
```

20. As with the other classes defined previously, we provide `with` methods that add or overwrite properties and return the new instance:

```
public function withCookieParams(array
$cookies)
{
  array_merge($this->getCookieParams(),
$cookies);
  return $this;
}
public function withQueryParams(array $query)
{
```

```
  array_merge($this->getQueryParams(),
$query);
  return $this;
}
public function withUploadedFiles(array
$uploadedFiles)
{
  if (!count($uploadedFiles)) {
      throw new
InvalidArgumentException(Constant::ERROR_NO_UP
LOADED_FILES);
  }
  foreach ($uploadedFiles as $fileObj) {
    if (!$fileObj instanceof
UploadedFileInterface) {
        throw new
InvalidArgumentException(Constant::ERROR_INVAL
ID_UPLOADED);
    }
  }
  $this->uploadedFileObjs = $uploadedFiles;
}
```

21. One important aspect of PSR-7 messages is that the body should also be available in a parsed manner, that is to say, a sort of structured representation rather than just a raw stream. Accordingly, we define `getParsedBody()` and its accompanying `with` method. The PSR-7 recommendations are quite specific when it comes to form posting. Note the series of `if` statements that check the `Content-Type` header as well as the method:

```
public function getParsedBody()
{
  if (!$this->parsedBody) {
      if (($this->getContentType() ==
Constants::CONTENT_TYPE_FORM_ENCODED
          || $this->getContentType() ==
Constants::CONTENT_TYPE_MULTI_FORM)
          && $this->getRequestMethod() ==
Constants::METHOD_POST)
      {
          $this->parsedBody = $_POST;
      } elseif ($this->getContentType() ==
Constants::CONTENT_TYPE_JSON
                || $this->getContentType() ==
Constants::CONTENT_TYPE_HAL_JSON)
```

```
        {
            ini_set("allow_url_fopen", true);
            $this->parsedBody =
json_decode(file_get_contents('php://input'));
        } elseif (!empty($_REQUEST)) {
            $this->parsedBody = $_REQUEST;
        } else {
            ini_set("allow_url_fopen", true);
            $this->parsedBody =
file_get_contents('php://input');
        }
    }
    return $this->parsedBody;
}

public function withParsedBody($data)
{
    $this->parsedBody = $data;
    return $this;
}
```

22. We also allow for attributes that are not precisely defined in PSR-7. Rather, we leave this open so that the developer can provide whatever is appropriate for the application. Notice the use of `withoutAttributes()` that allows you to remove attributes at will:

```
public function getAttributes()
{
    return $this->attributes;
}
public function getAttribute($name, $default =
NULL)
{
    return $this->attributes[$name] ?? $default;
}
public function withAttribute($name, $value)
{
    $this->attributes[$name] = $value;
    return $this;
}
public function withoutAttribute($name)
{
    if (isset($this->attributes[$name])) {
        unset($this->attributes[$name]);
    }
```

```
        return $this;
    }

    }
```

23. Finally, in order to load the different properties from an in-bound request, we define `initialize()`, which is not in PSR-7, but is extremely convenient:

```
public function initialize()
{
    $this->getServerParams();
    $this->getCookieParams();
    $this->getQueryParams();
    $this->getUploadedFiles;
    $this->getRequestMethod();
    $this->getContentType();
    $this->getParsedBody();
    return $this;
}
```

How it works...

First, be sure to complete the preceding recipe, as the `Message` and `Request` classes consume `Uri`, `Stream`, and `UploadedFile` value objects. After that, go ahead and define the classes summarized in the following table:

Class	Steps they are discussed in
`Application\MiddleWare\Message`	2 to 9
`Application\MiddleWare\Request`	10 to 14

Class	Steps they are discussed in
`Application\MiddleWare\ServerRequest`	15 to 20

After that, you can define a server program, `chap_09_middleware_server.php`, which sets up autoloading and uses the appropriate classes. This script will pull the incoming request into a `ServerRequest` instance, initialize it, and then use `var_dump()` to show what information was received:

```
<?php
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
use Application\MiddleWare\ServerRequest;

$request = new ServerRequest();
$request->initialize();
echo '<pre>', var_dump($request), '</pre>';
```

To run the server program, first change to the `/path/to/source/for/this/chapter folder`. You can then run the following command:

```
php -S localhost:8080
chap_09_middleware_server.php'
```

As for the client, first create a calling program, `chap_09_middleware_request.php`, that sets up autoloading, uses the appropriate classes, and defines the target server and a local text file:

```
<?php
define('READ_FILE', __DIR__ . '/gettysburg.txt');
define('TEST_SERVER', 'http://localhost:8080');
require __DIR__ .
'/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ .
'/..');
use Application\MiddleWare\ { Request, Stream,
Constants };
```

Next, you can create a `Stream` instance using the text as a source. This will become the body of a new Request, which, in this case, mirrors what might be expected for a form posting:

```
$body = new Stream(READ_FILE);
```

You can then directly build a `Request` instance, supplying parameters as appropriate:

```
$request = new Request(
    TEST_SERVER,
    Constants::METHOD_POST,
    $body,
    [Constants::HEADER_CONTENT_TYPE =>
Constants::CONTENT_TYPE_FORM_ENCODED,Constants::HE
ADER_CONTENT_LENGTH => $body->getSize()]
);
```

Alternatively, you can use the fluent interface syntax to produce exactly the same results:

```
$uriObj = new Uri(TEST_SERVER);
$request = new Request();
$request->withRequestTarget(TEST_SERVER)
        ->withMethod(Constants::METHOD_POST)
        ->withBody($body)
        -
>withHeader(Constants::HEADER_CONTENT_TYPE,
Constants::CONTENT_TYPE_FORM_ENCODED)
        -
>withAddedHeader(Constants::HEADER_CONTENT_LENGTH,
$body->getSize());
```

You can then set up a cURL resource to simulate a form posting, where the data parameter is the contents of the text file. You can follow that with `curl_init()`, `curl_exec()`, and so on, echoing the results:

```
$data = http_build_query(['data' => $request-
>getBody()->getContents()]);
$defaults = array(
    CURLOPT_URL => $request->getUri()-
>getUriString(),
    CURLOPT_POST => true,
    CURLOPT_POSTFIELDS => $data,
);
$ch = curl_init();
curl_setopt_array($ch, $defaults);
$response = curl_exec($ch);
curl_close($ch);
```

Here is how the direct output might appear:



See also

- An excellent article that shows example usage written by *Matthew Weir O'Phinney*, the editor of PSR-7 (also the lead architect for Zend Framework 1, 2, and 3), is available here: https://mwop.net/blog/2015-01-26-psr-7-by-example.html

Defining a PSR-7 Response class

The Response class represents outbound information returned to whatever entity made the original request. HTTP headers play an important role in this context as we need to know that format is requested by the client, usually in the incoming `Accept` header. We then need to set the appropriate `Content-Type` header in the Response class to match that format. Otherwise, the actual body of the response will be HTML, JSON, or whatever else has been requested (and delivered).

How to do it...

1. The `Response` class is actually much easier to implement than the `Request` class as we are only concerned with returning the response from the server to the client. Additionally, it extends our `Application\MiddleWare\Message` class where most of the work has been done. So, all that remains to be done is to define an `Application\MiddleWare\Response` class. As you will note, the only unique property is `$statusCode`:

```
namespace Application\MiddleWare;
use Psr\Http\Message\ { Constants, ResponseInterface, StreamInterface };
class Response extends Message implements ResponseInterface
{
  protected $statusCode;
```

2. The constructor is not defined by PSR-7, but we provide it for convenience, allowing a developer to create a `Response` instance with all parts intact. We use methods from `Message` and constants from the `Constants` class to verify the arguments:

```
public function __construct($statusCode = NULL,
                            StreamInterface $body = NULL,
                            $headers = NULL,
                            $version = NULL)
{
  $this->body = $body;
  $this->status['code'] = $statusCode ?? Constants::DEFAULT_STATUS_CODE;
  $this->status['reason'] = Constants::STATUS_CODES[$statusCode] ?? '';
  $this->httpHeaders = $headers;
  $this->version = $this->onlyVersion($version);
  if ($statusCode) $this->setStatusCode();
}
```

3. We provide a nice way to set the HTTP status code, irrespective of any headers, using `http_response_code()`, available from PHP 5.4 onwards. As this work is on PHP 7, we are safe in the knowledge that this method exists:

```
public function setStatusCode()
{
  http_response_code($this->getStatusCode());
}
```

4. Otherwise, it is of interest to obtain the status code using the following method:

```
public function getStatusCode()
{
  return $this->status['code'];
}
```

5. As with the other PSR-7-based classes discussed in earlier recipes, we also define a `with` method that sets the status code and returns the current instance. Note the use of `STATUS_CODES` to confirm its existence:

```
public function withStatus($statusCode, $reasonPhrase = '')
{
  if (!isset(Constants::STATUS_CODES[$statusCode])) {
      throw new InvalidArgumentException(Constants::ERROR_INVALID_STATUS);
```

```
        }
        $this->status['code'] = $statusCode;
        $this->status['reason'] = ($reasonPhrase) ?
      Constants::STATUS_CODES[$statusCode] : NULL;
        $this->setStatusCode();
        return $this;
      }
```

6. Finally, we define a method that returns the reason for the HTTP status, which is a short text phrase, in this example, based on RFC 7231. Note the use of the PHP 7 null coalesce operator `??` that returns the first non-null item out of three possible choices:

```
      public function getReasonPhrase()
      {
        return $this->status['reason']
          ?? Constants::STATUS_CODES[$this->status['code']]
          ?? '';
        }
      }
```

How it works...

First of all, be sure to define the classes discussed in the previous two recipes. After that, you can create another simple server program, `chap_09_middleware_server_with_response.php`, which sets up autoloading and uses the appropriate classes:

```
<?php
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\MiddleWare\ { Constants, ServerRequest, Response, Stream };
```

You can then define an array with key/value pairs, where the value points to a text file in the current directory to be used as content:

```
$data = [
  1 => 'churchill.txt',
  2 => 'gettysburg.txt',
  3 => 'star_trek.txt'
];
```

Next, inside a `try...catch` block, you can initialize some variables, initialize the server request, and set up a temporary filename:

```
try {

    $body['text'] = 'Initial State';
    $request = new ServerRequest();
    $request->initialize();
    $tempFile = bin2hex(random_bytes(8)) . '.txt';
    $code = 200;
```

After that, check to see whether the method is GET or POST. If it's GET, check to see whether an `id` parameter was passed. If so, return the body of the matching text file. Otherwise, return a list of text files:

```
if ($request->getMethod() == Constants::METHOD_GET) {
    $id = $request->getQueryParams()['id'] ?? NULL;
    $id = (int) $id;
    if ($id && $id <= count($data)) {
        $body['text'] = file_get_contents(
        __DIR__ . '/' . $data[$id]);
    } else {
```

```
            $body['text'] = $data;
        }
```

Otherwise, return a response indicating a success code 204 and the size of the request body received:

```
    } elseif ($request->getMethod() == Constants::METHOD_POST) {
        $size = $request->getBody()->getSize();
        $body['text'] = $size . ' bytes of data received';
        if ($size) {
            $code = 201;
        } else {
            $code = 204;
        }
    }
```

You can then catch any exceptions and report them with a status code of 500:

```
    } catch (Exception $e) {
        $code = 500;
        $body['text'] = 'ERROR: ' . $e->getMessage();
    }
```

The response needs to be wrapped in a stream, so you can write the body out to the temp file and create it as `Stream`. You can also set the `Content-Type` header to `application/json` and run `getHeaders()`, which outputs the current set of headers. After that, echo the body of the response. For this illustration, you could also dump the `Response` instance to confirm it was constructed correctly:

```
    try {
        file_put_contents($tempFile, json_encode($body));
        $body = new Stream($tempFile);
        $header[Constants::HEADER_CONTENT_TYPE] = 'application/json';
        $response = new Response($code, $body, $header);
        $response->getHeaders();
        echo $response->getBody()->getContents() . PHP_EOL;
        var_dump($response);
```

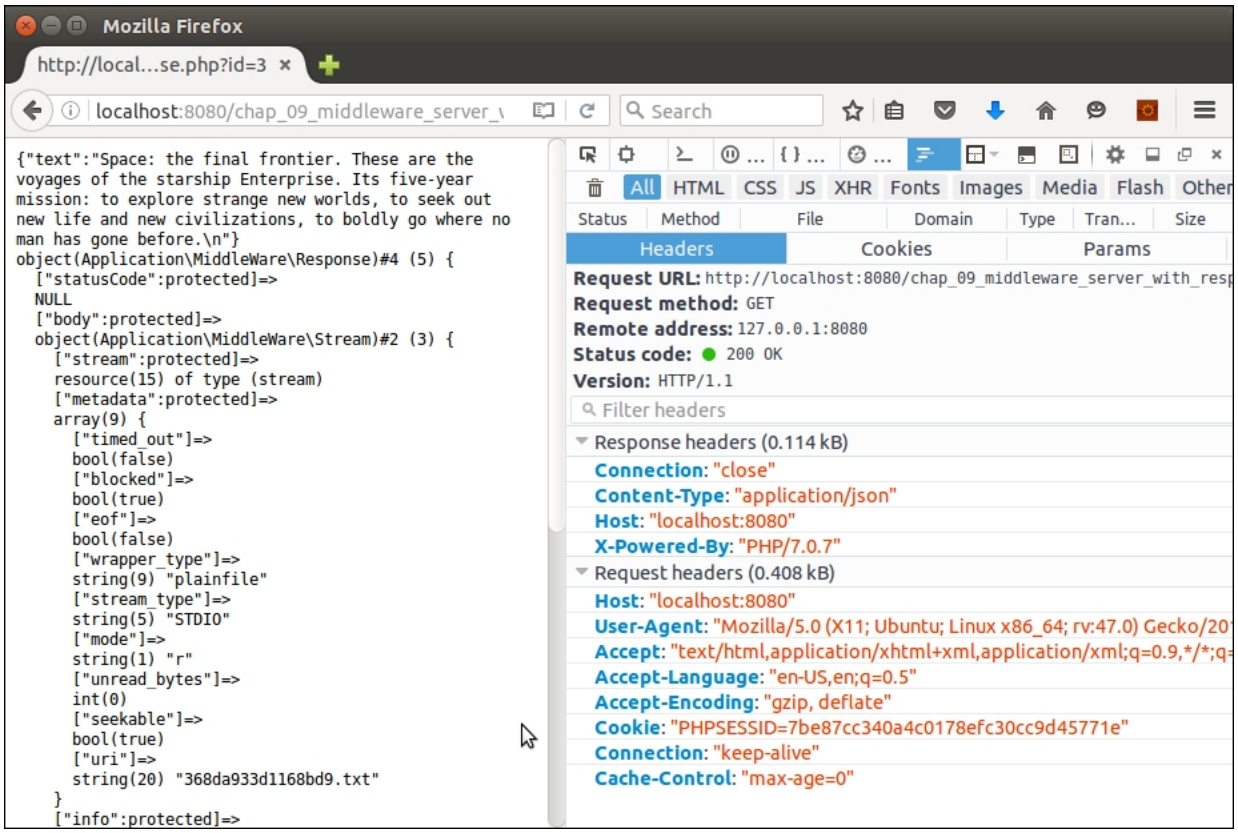
To wrap things up, catch any errors or exceptions using `Throwable`, and don't forget to delete the temp file:

```
    } catch (Throwable $e) {
        echo $e->getMessage();
    } finally {
       unlink($tempFile);
    }
```

To test, it's just a matter of opening a terminal window, changing to the `/path/to/source/for/this/chapter` directory, and running the following command:

```
    php -S localhost:8080
```

From a browser, you can then call this program, adding an `id` parameter. You might consider opening the developer tools to monitor the response header. Here is an example of the expected output. Note the content type of `application/json`:



See also

- For more information on PSR, please visit http://www.php-fig.org/psr/.
- The following table summarizes the state of PSR-7 compliance at the time of writing. The frameworks not included in this table either do not have PSR-7 support at all, or lack documentation for PSR-7.

Framework	Website	Notes
Slim	http://www.slimframework.com/docs/concepts/value-objects.html	High PSR-7 compliance
Laravel/Lumen	https://lumen.laravel.com/docs/5.2/requests	High PSR-7 compliance
Zend Framework 3/Expressive	https://framework.zend.com/blog/2016-06-28-zend-framework-3.html or https://zendframework.github.io/zend-expressive/ respectively	High PSR-7 compliance Also Diactoros, and Straigility

Framework	Website	Notes
Zend Framework 2	https://github.com/zendframework/zend-psr7bridge	PSR-7 bridge available
Symfony	http://symfony.com/doc/current/cookbook/psr7.html	PSR-7 bridge available
Joomla	https://www.joomla.org	Limited PSR-7 support
Cake PHP	http://mark-story.com/posts/view/psr7-bridge-for-cakephp	PSR-7 support is in the roadmap and will use the bridge approach

- There are a number of PSR-7 middleware classes already available. The following table summarizes some of the more popular ones:

Middleware	Website	Notes
Guzzle	https://github.com/guzzle/psr7	HTTP message library
Relay	http://relayphp.com/	Dispatcher
Radar	https://github.com/radarphp/Radar.Project	Action/domain/responder skeleton
NegotiationMiddleware	https://github.com/rszrama/negotiation-middleware	Content negotiation
psr7-csrf-middleware	https://packagist.org/packages/schnittstabil/psr7-csrf-middleware	Cross Site Request Forgery prevention
oauth2-server	http://alexbilbie.com/2016/04/league-oauth2-server-version-5-is-out	OAuth2 server which supports PSR-7
zend-diactoros	https://zendframework.github.io/zend-diactoros/	PSR-7 HTTP message implementation

Part 2. Module 2

Learning PHP 7 High Performance

Improve the performance of your PHP application to ensure the application users aren't left waiting

Chapter 1. Setting Up the Environment

PHP 7 has finally been released. For a long time, the PHP community was talking about it and has still not stopped. The main improvement in PHP 7 is its performance. For a long time, the PHP community faced performance issues in large-scale applications. Even some small applications with high traffic faced performance issues. Server resources were increased, but it did not help much because in the end the bottleneck was PHP itself. Different caching techniques were used, such as APC, and this helped a little. However, the community still needed a version of PHP that could boost the application's performance at its peak. And this is where PHPNG comes in.

PHPNG stands for **PHP next generation**. It is a completely separate branch and is mainly targeted for performance. Some people thought that PHPNG is **JIT** (**Just In Time**) compilation, but in reality, PHPNG is based on a refactored **Zend Engine**, which was highly optimized for performance. PHPNG is used as a base for PHP 7 development, and according to the official PHP wiki page, the PHPNG branch is now merged into the master branch.

Before starting to build an application, the development environment should be finalized and configured. In this chapter, we will discuss setting up the development environment on different systems, such as Windows and different flavors of Linux.

We will cover the following topics:

- Setting up Windows
- Setting up Ubuntu or Debian
- Setting up CentOS
- Setting up Vagrant

All other environments can be skipped, and we can set up the environment that we will use.

Setting up Windows

There are many tools available that have Apache, PHP, and MySQL bundled for Windows, provide easy installation, and are very easy to use. Most of these tools already provide support for PHP 7 with Apache, such as through XAMPP, WAMPP, and EasyPHP. EasyPHP is the only one that also provides support for **NGINX** and provides easy steps to changes webserver from NGINX to Apache or Apache to Nginx.

Note

XAMPP is also available for Linux and Mac OS X. However, WAMP and EasyPHP are only available for Windows. Any of these three can be used for this book, but we recommend EasyPHP as it supports NGINX, and for this book, we mostly use NGINX.

Any of the three tools can be used, but we require more control over every element of our web server tools, so we will also install NGINX, PHP 7, and MySQL individually and then connect them together.

Note

NGINX Windows binaries can be downloaded from http://nginx.org/en/download.html. We recommend using a stable version, though there is no problem with using a mainline version. PHP Windows binaries can be downloaded from http://windows.php.net/download/. Download either 32-bit or 64-

bit binaries of the *non-thread safe* version according to your system.

Perform the following steps:

1. Download NGINX and PHP Windows binaries mentioned in the information box. Copy NGINX to a suitable directory. For example, we have a completely separate D drive for development purposes. Copy NGINX to this development drive or any other directory. Now, copy PHP either to the NGINX directory or to any other secure folder location.
2. In the PHP directory, there will be two `.ini` files, `php.ini-development` and `php.ini-production`. Rename either one of them to `php.ini`. PHP will be using this configuration file.
3. Hold the *Shift* key and right click in the PHP directory to open the command-line window. The command-line window will be opened in the same location path. Issue the following command to start PHP:

   ```
   php-cgi -b 127.0.0.1:9000
   ```

   The `-b` option starts PHP and binds to the path for external **FastCGI** servers. The preceding command binds PHP to loop back the `127.0.0.1` IP on port `9000`. Now, PHP is accessible on this path.

4. To configure NGINX, open the `nginx_folder/conf/nginx.conf` file. The first thing to do is to add root and index to the server block, as follows:

   ```
   server {
     root html;
     index index.php index.html index.htm;
   ```

5. Now, we need to configure NGINX to use PHP as FastCGI on the path mentioned before on which it is started. In the `nginx.conf` file, uncomment the following location block for PHP:

```
location ~ \.php$ {
  fastcgi_pass    127.0.0.1:9000;
  fastcgi_param    SCRIPT_FILENAME
complete_path_webroot_folder$fastcgi_script_na
me;
include    fastcgi_params;
}
```

Note the `fastcgi_param` option. The highlighted `complete_path_webroot_folder` path should be the absolute path to the HTML directory inside the `nginx` folder. Let's say that your NGINX is placed at the `D:\nginx` path; then, the absolute path to the `HTML` folder will be `D:\nginx\html`. However, for the preceding `fastcgi_param` option, `\` should be replaced by `/`.

6. Now, restart NGINX by issuing the following command in the root of the NGINX folder:

```
nginx -s restart
```

7. After NGINX is restarted, open your browser and enter the IP or hostname of your Windows server or machine, and we will see the NGINX welcome message.
8. Now, to verify the PHP installation and its working with NGINX, create an `info.php` file in webroot and enter the following code in it:

```
<?php
  phpinfo();
?>
```

9. Now, in the browser, access your_ip/info.php, and we will be presented with a page full of PHP and server information. Congratulations! We configured NGINX and PHP to work perfectly together.

Note

On Windows and Mac OS X, we recommend that you use a virtual machine installed with all the tools on a Linux flavor to get the best performance out of the server. It is easy to manage everything in Linux. There are vagrant boxes available that have everything ready to use. Also, a custom virtual machine configuration with all the tools, including NGINX, Apache, PHP 7, Ubuntu, Debian, or CentOS, and other great ones, can be made at https://puphpet.com, which is an easy-to-use GUI. Another nice tool is Laravel Homestead, which is a **Vagrant** box with great tools.

Setting up Debian or Ubuntu

Ubuntu is derived from Debian, so the process is the same for both Ubuntu and Debian. We will use Debian 8 Jessie and Ubuntu 14.04 Server LTS. The same process can be applied to desktop versions for both.

First, add the repositories for both Debian and Ubuntu.

Debian

As of the time we're writing this book, Debian does not provide an official repository for PHP 7. So, for Debian, we will use `dotdeb` repositories to install NGINX and PHP 7. Perform the following steps:

1. Open the `/etc/apt/sources.list` file and add the following two lines at the end of the file:

```
deb http://packages.dotdeb.org jessie all
deb-src http://packages.dotdeb.org jessie all
```

2. Now, execute the following commands in the terminal:

```
wget https://www.dotdeb.org/dotdeb.gpg
sudo apt-key add dotdeb.gpg
sudo apt-get update
```

The first two commands will add `dotdeb` repo to Debian and the last command will refresh the cache for sources.

Ubuntu

As of the time of writing this book, Ubuntu also does not provide PHP 7 in their official repos, so we will use a third-party repo for the PHP 7 installation. Perform the following steps:

1. Run the following commands in the terminal:

```
sudo add-apt-repository ppa:ondrej/php
sudo apt-get update
```

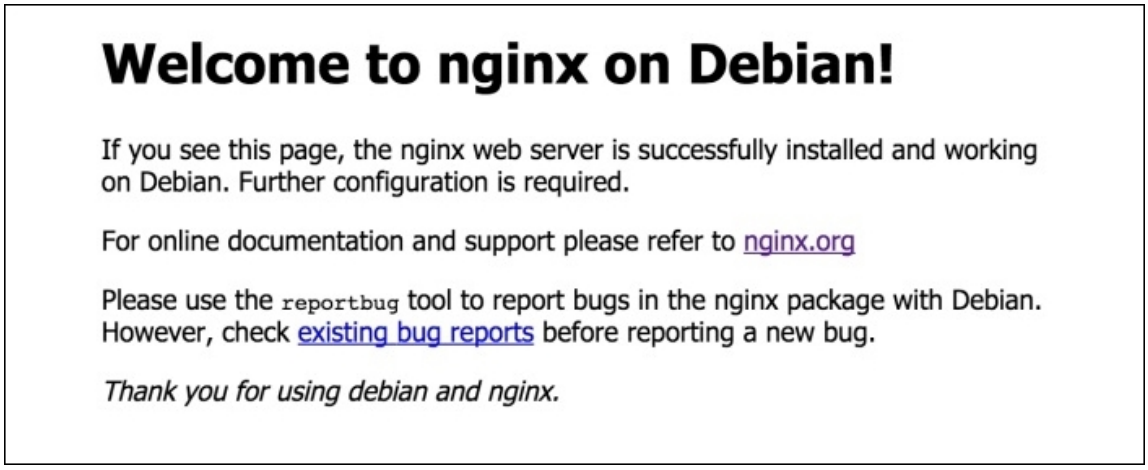
2. Now, the repositories are added. Let's install NGINX and PHP 7.

> **Note**
>
> The rest of the process is mostly the same for both Debian and Ubuntu, so we wont list them separately, as we did for the adding repositories section.

3. To install NGINX, run the following command in the terminal (Debian and Ubuntu):

```
sudo apt-get install nginx
```

4. After the installation is successful, it can be verified by entering the hostname and IP of the Debian or Ubuntu server. If we see something similar to the following screenshot, then our installation is successful:



The following is a list of three useful NGINX commands:

- `service nginx start`: This starts the NGINX server

- `service nginx restart`: This restarts the NGINX server
- `service nginx stop`: This stops the NGINX server

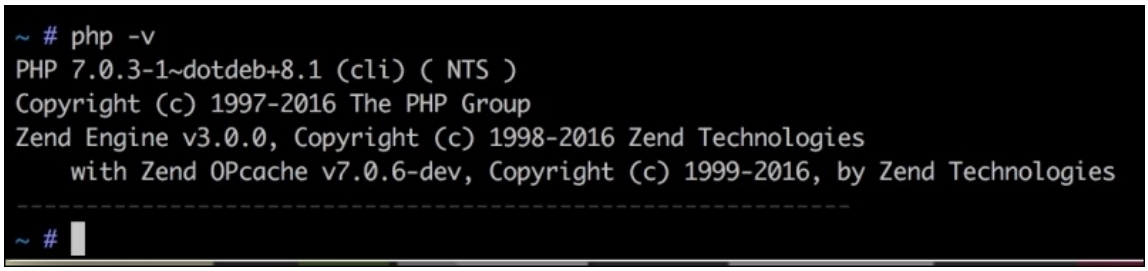
5. Now, it's time to install PHP 7 by issuing the following command:

```
sudo apt-get install php7.0 php7.0-fpm php7.0-
mysql php7.0-mcrypt php7.0-cli
```

   This will install PHP 7 along with the other modules mentioned. Also, we installed PHP Cli for the command-line purpose. To verify whether PHP 7 is properly installed, issue the following command in the terminal:

```
php -v
```

6. If it displays the PHP version along with some other details, as shown in the following screenshot, then PHP is properly installed:



7. Now, we need to configure NGINX to work with PHP 7. First, copy the NGINX default config file `/etc/nginx/sites-available/default` to `/etc/nginx/sites-available/www.packt.com.conf` using the following command in the terminal:

```
cd /etc/nginx/sites-available
sudo cp default www.packt.com.conf
sudo ln -s /etc/nginx /sites-
available/www.packt.com.conf /etc/
nginx/sites-enabled/www.packt.com.conf
```

   First, we copied the default configuration file, created another virtual host configuration file, `www.packt.com.conf`, and then

created a symbolic link file to this virtual host file in the sites-enabled folder.

> **Note**
>
> It is good practice to create a configuration file for each virtual host by the same name as of the domain so that it can easily be recognized by any other person.

8. Now, open the `/etc/nginx/sites-available/www.packt.com.conf` file and add or edit the highlighted code, as shown here:

```
server {
  server_name your_ip:80;
  root /var/www/html;
  index index.php index.html index.htm;
  location ~ \.php$ {
    fastcgi_pass unix:/var/run/php/php7.0-
fpm.sock;
      fastcgi_index index.php;
      include fastcgi_params;
  }
}
```

The preceding configuration is not a complete configuration file. We copied only those configuration options that are important and that we may want to change.

In the preceding code, our webroot path is `/var/www/html`, where our PHP files and other application files will be placed. In the index config option, add `index.php` so that if no file is provided in the URL, NGINX can look for and parse `index.php`.

We added a location block for PHP that includes a `fastcgi_pass` option, which has a path to the PHP7 FPM socket. Here, our PHP runs on a Unix socket, which is faster than that of TCP/IP.

9. After making these changes, restart NGINX. Now, to test whether PHP and NGINX are properly configured, create an `info.php` file at the root of the `webroot` folder and place the following code in it:

```
<?php
  phpinfo();
 ?>
```

10. Now, in the browser, type `server_ip/info.php`, and if you see a PHP configuration page, then congratulations! PHP and NGINX are both properly configured.

> ## Note
>
> If PHP and NGINX run on the same system, then PHP listens to the loopback IP at port `9000`. The port can be changed to any other port. In case, we want to run PHP on the TCP/IP port, then in `fastcgi_pass`, we will enter `127.0.0.1:9000`.

Now, let's install **Percona Server**. Percona Server is a fork of MySQL and is optimized for high performance. We will read more about Percona Server in Chapter 3, *Increasing PHP 7 Application Performance*. Now, let's install Percona Server on Debian/Ubuntu via the following steps:

1. First, let's add the Percona Server repository to our system by running the following command in the terminal:

```
sudo wget
https://repo.percona.com/apt/percona-
release_0.1-3.$(lsb_release -sc)_all.deb
sudo dpkg -i percona-release_0.1-
3.$(lsb_release -sc)_all.deb
```

The first command will download the repo packages from the Percona repo. The second command will install the downloaded packages and will create a `percona-release.list` file at `/etc/apt/sources.list.d/percona-release.list`.

2. Now, install Percona Server by executing the following command in the terminal:

```
sudo apt-get update
```

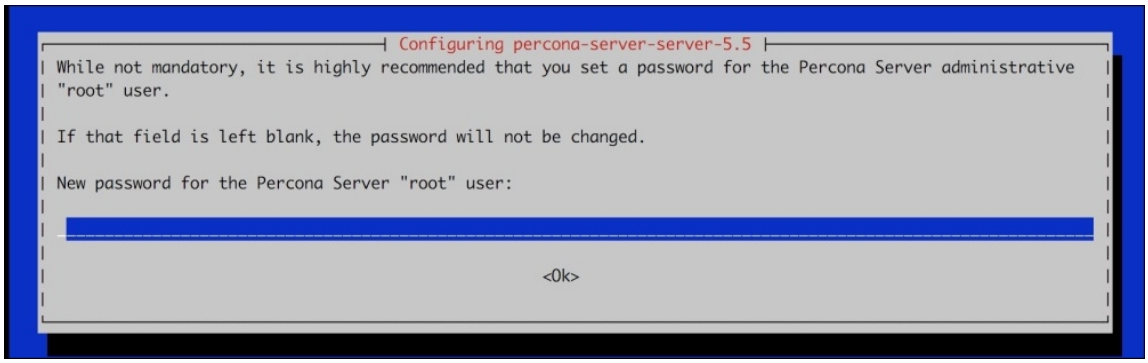
3. Now, issue the following command to install Percona Server:

```
sudo apt-get install percona-server-5.5
```

The installation process will start. It will take a while to download it.

> ## Note
>
> For the purpose of this book, we will install Percona Server 5.5. Percona Server 5.6 is also available, which can be installed without any issues.

During the installation, the password for the `root` user will be asked, as shown in the following screenshot:



It is optional but recommended to enter the password. After entering the password, re-enter the password on the next

screen. The installation process will continue.

4. After the installation is complete, the Percona Server installation can be verified by using the following command:

```
mysql --version
```

   It will display the version of Percona Server. As mentioned before, Percona Server is a fork of MySQL, so all the same MySQL commands, queries, and settings can be used.

Setting up CentOS

CentOS is a fork of **Red Hat Enterprise Linux** (**RHEL**) and stands for **Community Enterprise Operating System**. It is a widely used OS on servers specially used by hosting companies to provide shared hosting.

Let's start by configuring CentOS for our development environment. Perform the following steps:

Installing NGINX

1. First, we need to add NGINX RPM to our CentOS installation because CentOS does not provide any default repository for NGINX. Issue the following command in your terminal:

   ```
   sudo rpm -Uvh
   http://nginx.org/packages/centos/7/noarch/RPMS
   /nginx-release-centos-7-0.el7.ngx.noarch.rpm
   ```

   This will add the NGINX repo to CentOS.

2. Now, issue the following command to see which versions of NGINX are available to install:

   ```
   sudo yum --showduplicates list Nginx
   ```

   This will show you the latest stable releases. In our case, it displays NGINX 1.8.0 and NGINX 1.8.1.

3. Now, let's install NGINX using the following command:

   ```
   sudo yum install Nginx
   ```

   This will install NGINX.

4. On CentOS, NGINX won't start automatically after installation or restarting. So, first, we will enable NGINX to autostart after a system restarts using the following command:

```
systemctl enable Nginx.service
```

5. Now, let's start NGINX by issuing the following command:

```
systemctl start Nginx.service
```

6. Then, open your browser and enter the IP of the CentOS server or host name. If you see the same welcome screen as we saw in the figure earlier in the chapter for Debian, then NGINX is installed successfully.

   To check which version of NGINX is installed, issue the following command in the terminal:

```
Nginx -v
```

   On our server, the NGINX version installed is 1.8.1.

   Now, our web server is ready.

Installing PHP 7

1. The next step is to install PHP 7 FPM and configure both NGINX and PHP 7 to work together. As of the time of writing this book, PHP 7 is not packaged in official CentOS repositories. So, we have two choices to install PHP 7: either we build it from source, or we use third-party repositories. Building from source is a little bit difficult, so let's go the easy way and use third-party repositories.

   Note

> For this book, we will use webtatic repos for the PHP 7 installation as they provide quick updates for the new versions. There are some more repositories, and it is just the reader's choice to use any repository as long as it works.

2. Now, let's add a webtatic repository to our CentOS repo by issuing the following command:

```
rpm -Uvh
https://dl.fedoraproject.org/pub/epel/epel-
release-latest-7.noarch.rpm
rpm -Uvh
https://mirror.webtatic.com/yum/el7/webtatic-
release.rpm
```

3. After the repos are added successfully, issue the following command to see which version is available for installation:

```
sudo yum –showduplicates list php70w
```

   In our case, PHP 7.0.3 is available to install.

4. Now, issue the following command to install PHP 7 along with some modules that may be required:

```
sudo yum install php70w php70w-common php70w-
cli php70w-fpm php70w-mysql php70w-opcache
php70w-mcrypt
```

5. This will install core PHP 7 and some modules available for PHP 7. If any other module is required, it can be installed easily; however, first, search to check whether it is available or not. Issue the following command in the terminal to see all the available modules for PHP 7:

```
sudo yum search php70w-
```

   We will see a long list of all the available modules for PHP 7.

6. Now, let's say that we want to install the PHP 7 gd module; issue the following command:

```
sudo yum install php70w-gd
```

   This will install the gd module. Multiple modules can be installed using the same command and separating each module by a space, as we did in the initial installation of PHP.

   Now, to check which version of PHP is installed, issue the following command:

```
php -v
```

   In our case, PHP 7.0.3 is installed.

7. To start, stop, and restart PHP, issue the following commands in the terminal:

```
sudo systemctl start php-fpm
sudo systemctl restart php-fpm
sudo systemctl stop php-fpm
```

8. Now, let's configure NGINX to use PHP FPM. Open the default NGINX virtual host file located at `/etc/Nginx/conf.d/default.conf` using either `vi`, `nano`, or any other editor of your choice. Now, make sure that two options are set in the server block, as follows:

```
server {
    listen  80;
    server_name  localhost;
    root   /usr/share/nginx/html;
index  index.php index.html index.htm;
```

   The `root` option indicates the web document root where our website source code files will be placed. Index indicates the default files that will be loaded along with extensions. If any of these files are found, they will be executed by default, regardless of any file mentioned in the URLs.

9. The next configuration in NGINX is a location block for PHP. The following is the configuration for PHP:

```
location ~ \.php$ {
    try_files $uri =404;
    fastcgi_split_path_info ^(.+\.php)(/.+)$;
    fastcgi_pass 127.0.0.1:9000;
    fastcgi_index index.php;
    fastcgi_param
SCRIPT_FILENAME$document_root$fastcgi_script_n
ame;
      include fastcgi_params;
    }
```

   The preceding block is the most important configuration as it enables NGINX to communicate with PHP. The line `fastcgi_pass 127.0.0.1:9000` tells NGINX that PHP FPM can be accessed on the `127.0.0.1` loopback IP on port `9000`. The rest of the details are the same as those we discussed for Debian and Ubuntu.

10. Now, to test our installation, we will create a file named `info.php` with the following contents:

```
<?php
  phpinfo();
?>
```

    After saving the file, type `http://server_ip/info.php` or `http://hostname/info.php`, and we will get a page with complete information about PHP. If you see this page, congratulations! PHP runs alongside NGINX.

Installing Percona Server

1. Now, we will install Percona Server on CentOS. The installation process is the same, except that it has a separate repository. To add the Percona Server repo to CentOS, execute the following command in the terminal:

```
sudo yum install
http://www.percona.com/downloads/percona-
release/redhat/0.1-3/percona-release-0.1-
3.noarch.rpm
```

After the repo installation is completed, a message will be displayed stating the completion of the installation.

2. Now, to test the repo, issue the following command, and it will list all the available Percona packages:

```
sudo yum search percona
```

3. To install Percona Server 5.5, issue the following command in the terminal:

```
sudo yum install Percona-Server-server-55
```

The installation process will start. The rest of the process is the same as for Debian/Ubuntu.

4. After the installation is completed, we will see a completion message.

Setting up Vagrant

Vagrant is a tool used by developers for development environments. Vagrant provides an easy command-line interface to set up virtual machines with all the tools required. Vagrant uses boxes called Vagrant Boxes that can have a Linux operating system and other tools according to this box. Vagrant supports both Oracle VM VirtualBox and VMware. For the purpose of this book, we will use VirtualBox, which we assume is installed on your machine as well.

Vagrant has several boxes for PHP 7, including Laravel Homestead and Rasmus PHP7dev. So, let's get started by configuring the Rasmus PHP7dev box on Windows and Mac OS X.

Note

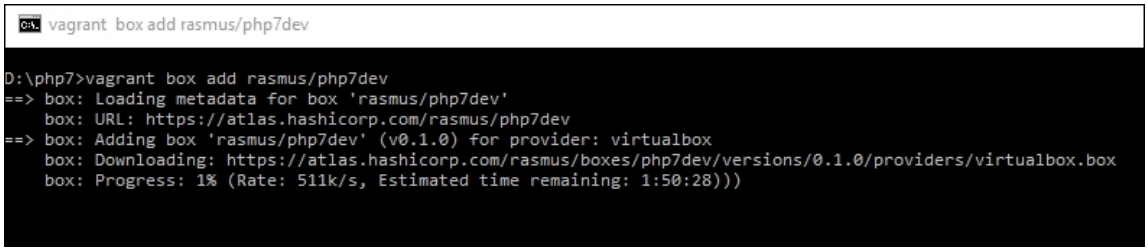
We assume that both VirutalBox and Vagrant are installed on our machine. VirtualBox can be downloaded from https://www.virtualbox.org/wiki/Downloads, and Vagrant can be downloaded from https://www.vagrantup.com/downloads.html for different platforms. Details about Rasmus PHP7dev VagrantBox can be found at https://github.com/rlerdorf/php7dev.

Perform the following steps:

1. Make a directory in one of the drives. For example, we made a `php7` directory in our `D` drive. Then, open the command line in this specific folder directly by holding the *Shift* key, right-clicking, and then selecting **Open command window here**.
2. Now, issue the following command in the command window:

```
vagrant box add rasmus/php7dev
```

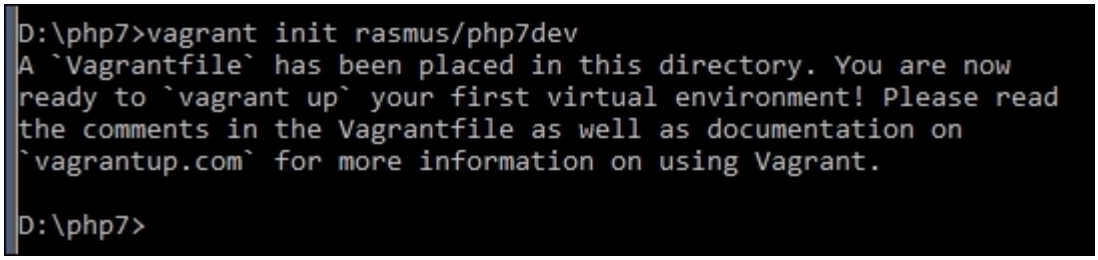
   It will start downloading the Vagrant box, as shown in the following screenshot:



3. Now, when the download is completed, we need to initialize it so that the box is configured and added to VirtualBox for us. Issue the following command in the command window:

```
vagrant init rasmus/php7dev
```

   This will start adding the box to VirtualBox and configuring it. When the process is completed, it will display a message, as in the following screenshot:



4. Now, issue the following command, which will completely set up the Vagrant box and start it up and running:

```
vagrant up
```

   This process will take a little bit of time. When it is completed, your box is ready and running and can be used.

5. Now, the first thing to do after it is up is to update everything. This box uses Ubuntu, so open the command window in the same `php7dev` directory and issue the following command:

```
vagrant ssh
```

   It will connect us with the virtual machines through SSH.

> **Note**
>
> In Windows, if SSH in not installed or not configured in the `PATH` variable, PuTTY can be used. It can be downloaded from http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html. For PuTTY, the host will be `127.0.0.1`, and the port will be `2222`. `Vagrant` is both the username and password for SSH.

6. When we are logged in to the box OS, issue the following commands to update the system:

```
sudo apt-get update
sudo apt-get upgrade
```

   This will update the core system, NGINX, MySQL, PHP 7, and other installed tools if new versions are available.

7. The box is now ready to use for development purposes. The box can be accessed in the browser by typing its IP address in the browser window. To find the IP address of the box, issue the following command in the SSH-connected command window:

```
sudo ifconfig
```

   This will display some details. Find out the IPv4 details there and take the IP of the box.

Summary

In this chapter, we configured different environments for the purpose of development. We installed NGINX and PHP 7 on the windows machine. We also configured Debian/Ubuntu and installed NGINX, PHP, and Percona Server 5.5. Then, we configured CentOS and installed NGINX, PHP, and Percona Server 5.5. Lastly, we discussed how to configure Vagrant Box on a Windows machine.

In the next chapter, we will study new features in PHP 7, such as type hints, namespace groupings and declarations, the Spaceship operator, and other features.

Chapter 2. New Features in PHP 7

PHP 7 has introduced new features that can help programmers write high-performing and effective code. Also, some old-fashioned features are completely removed, and PHP 7 will throw an error if used. Most of the fatal errors are now exceptions, so PHP won't show an ugly fatal error message any more; instead, it will go through an exception with the available details.

In this chapter, we will cover the following topics:

- Type hints
- Namespaces and group use declarations
- The anonymous classes
- Old-style constructor deprecation
- The Spaceship operator
- The null coalesce operator
- Uniform variable syntax
- Miscellaneous changes

OOP features

PHP 7 introduced a few new OOP features that will enable developers to write clean and effective code. In this section, we will discuss these features.

Type hints

Prior to PHP 7, there was no need to declare the data type of the arguments passed to a function or class method. Also, there was no need to mention the return data type. Any data type can be passed to and returned from a function or method. This is one of the huge

problems in PHP, in which it is not always clear which data types should be passed or received from a function or method. To fix this problem, PHP 7 introduced type hints. As of now, two type hints are introduced: scalar and return type hints. These are discussed in the following sections.

Type hints is a feature in both OOP and procedural PHP because it can be used for both procedural functions and object methods.

Scalar type hints

PHP 7 made it possible to use scalar type hints for integers, floats, strings, and Booleans for both functions and methods. Let's have a look at the following example:

```
class Person
{
  public function age(int $age)
  {
    return $age;
    }

  public function name(string $name)
  {
    return $name;
    }

  public function isAlive(bool $alive)
  {
    return $alive;
    }

}

$person = new Person();
echo $person->name('Altaf Hussain');
echo $person->age(30);
echo $person->isAlive(TRUE);
```

In the preceding code, we created a `Person` class. We have three methods, and each method receives different arguments whose data types are defined with them, as is highlighted in the preceding code.

If you run the preceding code, it will work fine as we will pass the desired data types for each method.

Age can be a float, such as `30.5` years; so, if we pass a float number to the `age` method, it will still work, as follows:

```
echo $person->age(30.5);
```

Why is that? It is because, by default, *scalar type hints are nonrestrictive*. This means that we can pass float numbers to a method that expects an integer number.

To make it more restrictive, the following single-line code can be placed at the top of the file:

```
declare(strict_types = 1);
```

Now, if we pass a float number to the `age` function, we will get an **Uncaught Type Error**, which is a fatal error that tells us that `Person::age` must be of the int type given the float. Similar errors will be generated if we pass a string to a method that is not of the string type. Consider the following example:

```
echo $person->isAlive('true');
```

The preceding code will generate the fatal error as the string is passed to it.

Return type hints

Another important feature of PHP 7 is the ability to define the return data type for a function or method. It behaves the same way scalar type hints behave. Let's modify our `Person` class a little to understand return type hints, as follows:

```
class Person
{
```

```
  public function age(float $age) : string
  {
    return 'Age is '.$age;
  }

  public function name(string $name) : string
  {
    return $name;
    }

  public function isAlive(bool $alive) : string
  {
    return ($alive) ? 'Yes' : 'No';
  }

}
```

The changes in the class are highlighted. The return type is defined using the `: data-type` syntax. It does not matter if the return type is the same as the scalar type. These can be different as long as they match their respective data types.

Now, let's try an example with the object return type. Consider the previous `Person` class and add a `getAddress` method to it. Also, we will add a new class, `Address`, to the same file, as shown in the following code:

```
class Address
{
  public function getAddress()
  {
  return ['street' => 'Street 1', 'country' =>
'Pak'];
  }
}

class Person
{
  public function age(float $age) : string
  {
    return 'Age is '.$age;
  }

  public function name(string $name) : string
```

```
  {
    return $name;
  }

  public function isAlive(bool $alive) : string
  {
    return ($alive) ? 'Yes' : 'No';
  }

  public function getAddress() : Address
  {
  return new Address();
  }
}
```

The additional code added to the `Person` class and the new `Address` class is highlighted. Now, if we call the `getAddress` method of the `Person` class, it will work perfectly and won't throw an error. However, let's suppose that we change the return statement, as follows:

```
public function getAddress() : Address
{
  return ['street' => 'Street 1', 'country' =>
'Pak'];
}
```

In this case, the preceding method will throw an *uncaught* exception similar to the following:

```
Fatal error: Uncaught TypeError: Return value of
Person::getAddress() must be an instance of
Address, array returned
```

This is because we return an array instead of an `Address` object. Now, the question is: why use type hints? The big advantage of using type hints is that it will always avoid accidentally passing or returning wrong and unexpected data to methods or functions.

As can be seen in the preceding examples, this makes the code clear, and by looking at the declarations of the methods, one can exactly know which data types should be passed to each of the

methods and what kind of data is returned by looking into the code of each method or comment, if any.

Namespaces and group use declaration

In a very large codebase, classes are divided into namespaces, which makes them easy to manage and work with. However, if there are too many classes in a namespace and we need to use 10 of them, then we have to type the complete use statement for all these classes.

> ## Note
>
> In PHP, it is not required to divide classes in subfolders according to their namespace, as is the case with other programming languages. Namespaces just provide a logical separation of classes. However, we are not limited to placing our classes in subfolders according to our namespaces.

For example, we have a `Publishers/Packt` namespace and the classes `Book`, `Ebook`, `Video`, and `Presentation`. Also, we have a `functions.php` file, which has our normal functions and is in the same `Publishers/Packt` namespace. Another file, `constants.php`, has the constant values required for the application and is in the same namespace. The code for each class and the `functions.php` and `constants.php` files is as follows:

```
//book.php
namespace Publishers\Packt;

class Book
{
  public function get() : string
  {
    return get_class();
```

```
  }
}
```

Now, the code for the `Ebook` class is as follows:

```
//ebook.php
namespace Publishers\Packt;

class Ebook
{
  public function get() : string
  {
    return get_class();
  }
}
```

The code for the `Video` class is as follows:

```
//presentation.php
namespace Publishers\Packt;

class Video
{
  public function get() : string
  {
    return get_class();
  }
}
```

Similarly, the code for the `presentation` class is as follows:

```
//presentation.php
namespace Publishers\Packt;

class Presentation
{
  public function get() : string
  {
    return get_class();
  }
}
```

All the four classes have the same methods, which return the classes' names using the PHP built-in `get_class()` function.

Now, add the following two functions to the `functions.php` file:

```
//functions.php

namespace Publishers\Packt;

function getBook() : string
{
  return 'PHP 7';
}
function saveBook(string $book) : string
{
  return $book.' is saved';
}
```

Now, let's add the following code to the `constants.php` file:

```
//constants.php

namespace Publishers/Packt;

const COUNT = 10;
const KEY = '123DGHtiop09847';
const URL = 'https://www.Packtpub.com/';
```

The code in both `functions.php` and `constants.php` is self-explanatory. Note that each file has a `namespace Publishers/Packt` line at the top, which makes these classes, functions, and constants belong to this namespace.

Now, there are three ways to use the classes, functions, and constants. Let's consider each one.

Take a look at the following code:

```
//Instantiate objects for each class in namespace

$book = new Publishers\Packt\Book();
```

```
$ebook = new Publishers\Packt\Ebook();
$video = new Publishers\Packt\Video();
$presentation = new
Publishers\Packt\Presentation();

//Use functions in namespace

echo Publishers/Packt/getBook();
echo Publishers/Packt/saveBook('PHP 7 High
Performance');

//Use constants

echo Publishers\Packt\COUNT;
echo Publishers\Packt\KEY;
```

In the preceding code, we used namespace names directly while creating objects or using functions and constants. The code looks fine, but it is cluttered. Namespace is everywhere, and if we have lots of namespaces, it will look very ugly, and the readability will be affected.

Note

We did not include class files in the previous code. Either the `include` statements or PHP's `__autoload` function can be used to include all the files.

Now, let's rewrite the preceding code to make it more readable, as follows:

```
use Publishers\Packt\Book;
use Publishers\Packt\Ebook;
use Publishers\Packt\Video;
use Publishers\Packt\Presentation;
use function Publishers\Packt\getBook;
use function Publishers\Packt\saveBook;
use const Publishers\Packt\COUNT;
use const Publishers\Packt\KEY;
```

```
$book = new Book();
$ebook = new Ebook(();
$video = new Video();
$pres = new Presentation();

echo getBook();
echo saveBook('PHP 7 High Performance');

echo COUNT;
echo KEY;
```

In the preceding code, at the top, we used PHP statements for specific classes, functions, and constants in a namespace. However, we still wrote duplicate lines of code for each class, function, and/or constant. This may lead to us have lots of use statements at the top of the file, and the overall verbosity would not be good.

To fix this problem, PHP 7 introduced group use declaration. There are three types of group use declarations:

- Non mixed use declarations
- Mixed use declarations
- Compound use declarations

Non mixed group use declarations

Consider that we have different types of features in a namespace, as we have classes, functions, and contacts in a namespace. In non mixed group use declarations, we declare them separately using a `use` statement. To better understand it, take a look at the following code:

```
use Publishers\Packt\{ Book, Ebook, Video,
Presentation };
use function Publishers\Packt\{ getBook, saveBook
};
use const Publishers\Packt\{ COUNT, KEY };
```

We have three types of features in a namespace: class, functions, and constants. So, we have used separate group `use` declaration

statements to use them. The code is now looking more cleaner, organized, and readable and doesn't require too much duplicate typing.

Mixed group use declarations

In this declaration, we combine all types into a single `use` statement. Take a look at the following code:

```
use Publishers\Packt\{
  Book,
  Ebook,
  Video,
  Presentation,
  function getBook,
  function saveBook,
  const COUNT,
  const KEY
};
```

The compound namespace declaration

To understand the compound namespace declaration, we will consider the following criteria.

Let's say we have a `Book` class in the `Publishers\Packt\Paper` namespace. Also, we have an `Ebook` class in the `Publishers\Packt\Electronic` namespace. The `Video` and `Presentation` classes are in the `Publishers\Packt\Media` namespace. So, to use these classes, we will use the code, as follows:

```
use Publishers\Packt\Paper\Book;
use Publishers\Packt\Electronic\Ebook;
use Publishers\Packt\Media\{Video,Presentation};
```

In the compound namespace declaration, we can use the preceding namespaces as follows:

```
use Publishers\Packt\{
  Paper\Book,
  Electronic\Ebook,
  Media\Video,
  Media\Presentation
};
```

It is more elegant and clear, and it doesn't require extra typing if the namespace names are long.

The anonymous classes

An anonymous class is a class that is declared and instantiated at the same time. It does not have a name and can have the full features of a normal class. These classes are useful when a single one-time small task is required to be performed and there is no need to write a full-blown class for it.

> ## Note
>
> While creating an anonymous class, it is not named, but it is named internally in PHP with a unique reference based on its address in the memory block. For example, the internal name of an anonymous class may be `class@0x4f6a8d124`.

The syntax of this class is the same as that of the named classes, but only the name of the class is missing, as shown in the following syntax:

```
new class(argument) { definition };
```

Let's look at a basic and very simple example of an anonymous class, as follows:

```
$name = new class() {
  public function __construct()
```

```
  {
    echo 'Altaf Hussain';
  }
};
```

The preceding code will just display the output as `Altaf Hussain`.

Arguments can also be passed to the *anonymous class constructor*, as shown in the following code:

```
$name = new class('Altaf Hussain') {
  public function __construct(string $name)
  {
    echo $name;
  }
};
```

This will give us the same output as the first example.

Anonymous classes can extend other classes and have the same parent-child classes functioning as normal named classes. Let's have another example; take a look at the following:

```
class Packt
{
  protected $number;

  public function __construct()
  {
    echo 'I am parent constructor';
  }

  public function getNumber() : float
  {
    return $this->number;
  }
}

$number = new class(5) extends packt
{
  public function __construct(float $number)
  {
    parent::__construct();
```

```
    $this->number = $number;
  }
};

echo $number->getNumber();
```

The preceding code will display `I am parent constructor` and `5`. As can be seen, we extended the `Packt` class the way we extend named classes. Also, we can access the `public` and `protected` properties and methods within the anonymous class and public properties and methods using anonymous class objects.

Anonymous classes can implement interfaces too, the same as named classes. Let's create an interface first. Run the following:

```
interface Publishers
{
  public function __construct(string $name, string
$address);
  public function getName();
  public function getAddress();
}
```

Now, let's modify our `Packt` class as follows. We added the highlighted code:

```
class Packt
{
  protected $number;
  protected $name;
  protected $address;
  public function …
}
```

The rest of the code is same as the first `Packt` class. Now, let's create our anonymous class, which will implement the `Publishers` interface created in the previous code and extend the new `Packt` class, as follows:

```
$info = new class('Altaf Hussain', 'Islamabad,
Pakistan')extends packt implements Publishers
{
  public function __construct(string $name, string
$address)
  {
    $this->name = $name;
    $this->address = $address;
  }

  public function getName() : string
  {
  return $this->name;
  }

  public function getAddress() : string
  {
  return $this->address;
  }
}

echo $info->getName(). ' '.$info->getAddress();
```

The preceding code is self-explanatory and will output `Altaf Hussain` along with the address.

It is possible to use anonymous classes within another class, as shown here:

```
class Math
{
  public $first_number = 10;
  public $second_number = 20;

  public function add() : float
  {
    return $this->first_number + $this-
>second_number;
  }

  public function multiply_sum()
  {
    return new class() extends Math
    {
      public function multiply(float
```

```
$third_number) : float
      {
        return $this->add() * $third_number;
      }
    };
  }
}

$math = new Math();
echo $math->multiply_sum()->multiply(2);
```

The preceding code will return `60`. How does this happen? The `Math` class has a `multiply_sum` method that returns the object of an anonymous class. This anonymous class is extended from the `Math` class and has a `multiply` method. So, our `echo` statement can be divided into two parts: the first is `$math->multiply_sum()`, which returns the object of the anonymous class, and the second is `->multiply(2)`, in which we chained this object to call the anonymous class's `multiply` method along with an argument of the value `2`.

In the preceding case, the `Math` class can be called the outer class, and the anonymous class can be called the inner class. However, remember that it is not required for the inner class to extend the outer class. In the preceding example, we extended it just to ensure that the inner classes could have access to the outer classes' properties and methods by extending the outer classes.

Old-style constructor deprecation

Back in PHP 4, the constructor of a class has the same name method as that of the class. It is still used and is valid until PHP's 5.6 version. However, now, in PHP 7, it is deprecated. Let's have an example, as shown here:

```
class Packt
{
  public function packt()
  {
    echo 'I am an old style constructor';
  }
```

```
}

$packt = new Packt();
```

The preceding code will display the output `I am an old style constructor` with a deprecated message, as follows:

```
Deprecated: Methods with the same name as their
class will not be constructors in a future version
of PHP; Packt has a deprecated constructor in…
```

However, the old style constructor is still called. Now, let's add the PHP `__construct` method to our class, as follows:

```
class Packt
{
  public function __construct()
  {
    echo 'I am default constructor';
  }

  public function packt()
  {
    echo 'I am just a normal class method';
  }
}

$packt = new Packt();
$packt->packt();
```

In the preceding code, when we instantiated the object of the class, the normal `__construct` constructor was called. The `packt()` method isn't considered a normal class method.

Note

Old-style constructors are deprecated, which means that they will still work in PHP 7 and a deprecated message will be displayed, but it will be removed in the upcoming versions. It is best practice to not use them.

The throwable interface

PHP 7 introduced a base interface that can be base for every object that can use the `throw` statement. In PHP, exceptions and errors can occur. Previously, exceptions could be handled, but it was not possible to handle errors, and thus, any fatal error caused the complete application or a part of the application to halt. To make errors (the most fatal errors) catchable as well, PHP 7 introduced the *throwable* interface, which is implemented by both the exception and error.

> **Note**
>
> The PHP classes we created can't implement the throwable interface. If required, these classes must extend an exception.

We all know exceptions, so in this topic, we will only discuss errors, which can handle the ugly, fatal errors.

Error

Almost all fatal errors can now throw an error instance, and similarly to exceptions, error instances can be caught using the `try/catch` block. Let's have a simple example:

```
function iHaveError($object)
{
  return $object->iDontExist();
  {

//Call the function
iHaveError(null);
echo "I am still running";
```

If the preceding code is executed, a fatal error will be displayed, the application will be halted, and the `echo` statement won't be executed in the end.

Now, let's place the function call in the `try/catch` block, as follows:

```
try
{
  iHaveError(null);
} catch(Error $e)
{
  //Either display the error message or log the
error message
  echo $e->getMessage();
}

echo 'I am still running';
```

Now, if the preceding code is executed, the `catch` body will be executed, and after this, the rest of the application will continue running. In the preceding case, the `echo` statement will be executed.

In most cases, the error instance will be thrown for the most fatal errors, but for some errors, a subinstance of error will be thrown, such as `TypeError`, `DivisionByZeroError`, `ParseError`, and so on.

Now, let's take a look at a `DivisionByZeroError` exception in the following example:

```
try
{
  $a = 20;
  $division = $a / 20;
} catch(DivisionByZeroError $e)
{
  echo $e->getMessage();
}
```

Before PHP 7, the preceding code would have issued a warning about the division by zero. However, now in PHP 7, it will throw a `DivisionByZeroError`, which can be handled.

New operators

PHP 7 introduced two interested operators. These operators can help write less and cleaner code, so the final code will be more readable as compared to the traditional operators in use. Let's have a look at them.

The Spaceship operator (<=>)

The Spaceship or Combined Comparison operator is useful to compare values (strings, integers, floats, and so on), arrays, and objects. This operator is just a wrapper and performs the same tasks as the three comparison operators `==`, `<`, and `>`. This operator can also be used to write clean and less code for callback functions for `usort`, `uasort`, and `uksort`. This operator works as follows:

- It returns 0 if both the operands on left- and right-hand sides are equal
- It returns -1 if the right operand is greater than the left operand
- It returns 1 if the left operand is greater than the right one

Let's take a look at a few examples by comparing integers, strings, objects, and arrays and note the result:

```
$int1 = 1;
$int2 = 2;
$int3 = 1;

echo $int1 <=> $int3; //Returns 0
echo '<br>';
echo $int1 <=> $int2; //Returns -1
echo '<br>';
echo $int2 <=> $int3; //Returns 1
```

Run the preceding code, and you will have an output similar to the following:

```
0
-1
1
```

In the first comparison, in which we compare `$int1` and `$int3`, both are equal, so it will return `0`. In the second comparison, in which `$int1` and `$int2` are compared, it will return `-1` because the right operand (`$int2`) in greater than the left operand (`$int1`). Finally, the third comparison will return `1` as the left operand (`$int2`) is greater than the right operand (`$int3`).

The preceding is a simple example in which we compared integers. We can check strings, objects, and arrays in the same way, and they are compared the same standard PHP way.

Note

Some examples for the `<=>` operator can be found at https://wiki.php.net/rfc/combined-comparison-operator. This is an RFC publication that has more useful details about its usage.

This operator can be more useful in sorting arrays. Take a look at the following code:

```
Function normal_sort($a, $b) : int
{
  if( $a == $b )
    return 0;
  if( $a < $b )
    return -1;
  return 1;
}

function space_sort($a, $b) : int
{
  return $a <=> $b;
}

$normalArray = [1,34,56,67,98,45];
```

```
//Sort the array in asc
usort($normalArray, 'normal_sort');

foreach($normalArray as $k => $v)
{
  echo $k.' => '.$v.'<br>';
}

$spaceArray = [1,34,56,67,98,45];

//Sort it by spaceship operator
usort($spaceArray, 'space_sort');

foreach($spaceArray as $key => $value)
{
  echo $key.' => '.$value.'<br>';
}
```

In the preceding code, we used two functions to sort the two different arrays with the same values. The `$normalArray` array is sorted by the `normal_sort` function, in which the `normal_sort` function uses `if` statements to compare the values. The second array `$spaceArray` has the same values as `$normalArray`, but this array is sorted by the `space_sort` function, which uses the Spaceship operator. The final result for both array sorts is the same, but the code in the callback functions is different. The `normal_sort` function has `if` statements and multiple lines of code, while the `space_sort` function has a single line of code—that's it! The `space_sort` function code is clearer and does not require multiple if statements.

The null coalesce operator(??)

We all know ternary operators, and we use them most of the time. Ternary operators are just a single-line replacement for *if-else* statements. For example, consider the following code:

```
$post = ($_POST['title']) ? $_POST['title'] :
NULL;
```

If `$_POST['title']` exists, then the `$post` variable will be assigned its value; otherwise, `NULL` will be assigned. However, if `$_POST` or `$_POST['title']` does not exist or is null, then PHP will issue a notice of *Undefined index*. To fix this notice, we need to use the `isset` function, as follows:

```
$post = isset($_POST['title']) ? $_POST['title'] :
NULL;
```

Mostly, it will seem fine, but it becomes very nasty when we have to check for values in multiple places, especially when using PHP as a templating language.

In PHP 7, the coalescence operator is introduced, which is simple and returns the value of its first operand (left operand) if it exists and is not null. Otherwise, it returns its second operand (right operand). Consider the following example:

```
$post = $_POST['title'] ?? NULL;
```

This example is exactly similar to the preceding code. The coalesce operator checks whether `$_POST['title']` exists. If it does, the operator returns it; otherwise, it returns `NULL`.

Another great feature of this operator is that it can be chained. Here's an example:

```
$title = $_POST['title'] ?? $_GET['title'] ?? 'No
POST or GET';
```

According to the definition, it will first check whether the first operand exists and return it; if it does not exist, it will return the second operand. Now, if there is another coalesce operator used on the second operand, the same rule will be applied, and the value on the left operand will be returned if it exists. Otherwise, the value of the right operand will be returned.

So, the preceding code is the same as the following:

```
If(isset($_POST['title']))
  $title = $_POST['title'];
elseif(isset($_GET['title']))
  $title = $_GET['title'];
else
  $title = 'No POST or GET';
```

As can be noted in the preceding examples, the coalesce operator can help write clean, concise, and less code.

Uniform variable syntax

Most of the time, we may face a situation in which the method, variable, or classes names are stored in other variables. Take a look at the following example:

```
$objects['class']->name;
```

In the preceding code, first, `$objects['class']` will be interpreted, and after this, the property name will be interpreted. As shown in the preceding example, variables are normally evaluated from left to right.

Now, consider the following scenario:

```
$first = ['name' => 'second'];
$second = 'Howdy';

echo $$first['name'];
```

In PHP 5.x, this code would be executed, and the output would be `Howdy`. However, this is not inconsistent with the left-to-right expression evaluation. This is because `$$first` should be evaluated first and then the index name, but in the preceding case, it is evaluated as `${$first['name']}`. It is clear that the variable syntax is not consistent and may create confusion. To avoid this inconsistency, PHP 7 introduced a new syntax called uniform variable syntax. Without using this syntax, the preceding example will bring it into notice, and the desired results won't be produced. To make it work in PHP 7, the curly brackets should be added, as follows:

```
echo ${$first['name']};
```

Now, let's have another example, as follows:

```
class Packt
{
  public $title = 'PHP 7';
  public $publisher = 'Packt Publisher';

  public function getTitle() : string
  {
    return $this->title;
  }

  public function getPublisher() : string
  {
    return $this->publisher;
  }
}

$mthods = ['title' => 'getTitle', 'publisher' =>
'getPublisher'];
$object = new Packt();
echo 'Book '.$object->$methods['title']().' is
published by '.$object->$methods['publisher']();
```

If the preceding code is executed in PHP 5.x, it will work fine and output our desired result. However, if we execute this code in PHP 7, it will give a fatal error. The error will be at the last line of the code, which is highlighted. PHP 7 will first try to evaluate `$object->$method.` After this, it will try to evaluate `['title']`; and so on; this is not correct.

To make it work in PHP 7, the curly brackets should be added, as in the following code:

```
echo 'Book '.$object->{$methods['title']}().' is
published by '.$object->{$methods['publisher']}();
```

After making the changes mentioned before, we will get our desired output.

Miscellaneous features and changes

PHP 7 also introduced some other new features with small changes, such as new syntax for array constants, multiple default cases in `switch` statement, options array in `session_start`, and so on. Let's have a look at these too.

Constant arrays

Starting with PHP 5.6, constant arrays can be initialized using the `const` keyword, as follows:

```
const STORES = ['en', 'fr', 'ar'];
```

Now, starting with PHP 7, constant arrays can be initialized using the `define` function, as follows:

```
define('STORES', ['en', 'fr', 'ar']);
```

Multiple default cases in the switch statement

Prior to PHP 7, multiple default cases in a switch statement were allowed. Check out the following example:

```
switch(true)
{
  default:
    echo 'I am first one';
    break;
  default:
    echo 'I am second one';
}
```

Before PHP 7, the preceding code was allowed, but in PHP 7, this will result in a fatal error similar to the following:

```
Fatal error: Switch statements may only contain
one default clause in…
```

The options array for session_start function

Before PHP 7, whenever we needed to start a session, we just used the `session_start()` function. This function did not take any arguments, and all the settings defined in `php.ini` were used. Now, starting with PHP 7, an optional array for options can be passed, which will override the session settings in the `php.ini` file.

A simple example is as follows:

```
session_start([
  'cookie_lifetime' => 3600,
  'read_and_close'  => true
]);
```

As can be seen in the preceding example, it is possible to override the `php.ini` settings for a session easily.

Filtered unserialize function

It is common practice to serialize and unserialize objects. However, the PHP `unserialize()` function was not secure because it did not have any filtering options and could unserialize objects of any type. PHP 7 introduced filtering in this function. The default filtering option is to unserialize objects of all classes or types. Its basic working is as follows:

```
$result = unserialize($object, ['allowed_classes'
=> ['Packt', 'Books', 'Ebooks']]);
```

Summary

In this chapter, we discussed new OOP features, such as type hints, anonymous classes, the throwable interface, group use declaration for namespaces, and two important new operators, the Spaceship or Combined Comparison operator and the null Coalesce operator. Also, we discussed the uniform variable syntax and a few other new features, such as new syntax for the contact array definition, options array for the `session_start()` function, and removal of multiple default cases in the switch statement.

In the next chapter, we will discuss how to improve the application's performance. We will discuss Apache and NGINX and different settings for them to improve performance.

We will discuss different settings for PHP to improve its performance. The Google page speed module, CSS/JavaScript combining and compression, CDN, and so on will also be discussed.

Chapter 3. Improving PHP 7 Application Performance

PHP 7 has been completely rewritten from the ground up based on the **PHP Next Generation** (**phpng** or **PHPNG**) targeting performance. However, there are always more ways to improve the performance of the application, including writing high performance code, using best practices, web server optimizations, caching, and so on. In this chapter, we will discuss such optimizations listed as follows:

- NGINX and Apache
- HTTP server optimization
- Content Delivery Network (CDN)
- JavaScript/CSS optimization
- Full page caching
- Varnish
- The infrastructure

NGINX and Apache

There are too many HTTP server software available, and each one has its pros and cons. The two most popular HTTP servers used are NGINX and Apache. Let's have a look at both of them and note which one is better for our needs.

Apache

Apache is the most widely used HTTP server and is loved by most administrators. It is selected by administrators because of its flexibility, widespread support, power, and modules for most of the interpreted languages, such as PHP. As Apache can process a vast number of interpreted languages, it does not need to communicate

with other software to fulfill the request. Apache can process requests in prefork (the processes are spawned across thread), worker (threads are spawned across processes), and event-driven (same as worker process, but it sets dedicated threads for *keep-alive* connections and separate threads for active connections); thus, it provides much more flexibility.

As discussed earlier, each request will be processed by a single thread or process, so Apache consumes too many resources. When it comes to high-traffic applications, Apache may slow down the application as it does not provide good support for concurrent processing.

NGINX

NGINX was built to solve the concurrency problems with high-traffic applications. NGINX provides asynchronous, event-driven, and nonblocking request handling. As requests are processed asynchronously, NGINX does not wait for a request to be completed to block the resource.

NGINX creates worker processes, and each individual worker process can handle thousands of connections. So, a few processes can handle high traffic at once.

NGINX does not provide any built-in support for any interpreted languages. It relies on external resources for this. This is also good because the processing is made outside NGINX, and NGINX only processes the connections and requests. Mostly, NGINX is considered faster than Apache. In some situations, such as with static content (serving images, `.css` and `.js` files, and so on), this can be true, but in current high performance servers, Apache is not the problem; PHP is the bottleneck.

Note

Both Apache and NGINX are available for all kinds of operations systems. For the purpose of this book, we will use Debian and Ubuntu, so all file paths will be mentioned according to these OSes

As mentioned before, we will use NGINX for this book.

HTTP server optimization

Each HTTP server provides certain features that can be used to optimize request handling and serving content. In this section, we will share some techniques for both Apache and NGINX that can be used to optimize the web server and provide the best performance and scalability. Mostly, when these optimizations are applied, a restart for Apache or NGINX is required.

Caching static files

Mostly, static files, such as images, `.css`, `.js`, and fonts don't change frequently. So, it is best practice to cache these static files on the end user machine. For this purpose, the web server adds special headers to the response, which tells the user browser to cache the static content for a certain amount of time. The following is the configuration code for both Apache and NGINX.

Apache

Let's have a look at the Apache configuration to cache the following static content:

```
<FilesMatch "\.
(ico|jpg|jpeg|png|gif|css|js|woff)$">
Header set Cache-Control "max-age=604800, public"

Apache Configuration

</FileMatch>

Should be

</FilesMatch>
</FileMatch>
```

In the preceding code that has to be placed in a `.htaccess` file, we used the Apache `FilesMatch` directive to match the extensions of files. If a desired extension file is requested, Apache sets the headers to cache control for seven days. The browser then caches these static files for seven days.

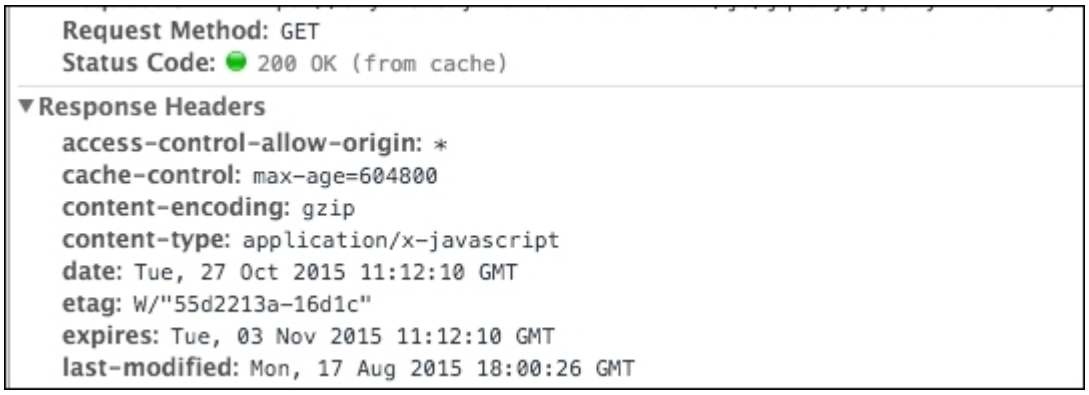
NGINX

The following configuration can be placed in `/etc/nginx/sites-available/your-virtual-host-conf-file`:

```
Location ~* .(ico|jpg|jpeg|png|gif|css|js|woff)$ {
  Expires 7d;
}
```

In the preceding code, we used the NGINX `Location` block with a case-insensitive modifier (`~*`) to set `Expires` for seven days. This code will set the cache-control header for seven days for all the defined file types.

After making these settings, the response headers for a request will be as follows:



In the preceding figure, it can be clearly seen that the `.js` file is loaded from cache. Its cache-control header is set to seven days or 604,800 seconds. The expiry date can also be noted clearly in the `expires` headers. After the expiry date, the browser will load this `.js`

file from the server and cache it again for the duration defined in the cache-control headers.

HTTP persistent connection

In HTTP persistent connection, or HTTP keep-alive, a single TCP/IP connection is used for multiple requests or responses. It has a huge performance improvement over the normal connection as it uses only a single connection instead of opening and closing connections for each and every single request or response. Some of the benefits of the HTTP keep-alive are as follows:

- The load on the CPU and memory is reduced because fewer TCP connections are opened at a time, and no new connections are opened for subsequent requests and responses as these TCP connections are used for them.
- Reduces latency in subsequent requests after the TCP connection is established. When a TCP connection is to be established, a three-way handshake communication is made between a user and the HTTP server. After successfully handshaking, a TCP connection is established. In case of keep-alive, the handshaking is performed only once for the initial request to establish a TCP connection, and no handshaking or TCP connection opening/closing is performed for the subsequent requests. This improves the performance of the requests/responses.
- Network congestion is reduced because only a few TCP connections are opened to the server at a time.

Besides these benefits, there are some side effects of keep-alive. Every server has a concurrency limit, and when this concurrency limit is reached or consumed, there can be a huge degradation in the application's performance. To overcome this issue, a time-out is defined for each connection, after which the HTTP keep-alive connection is closed automatically. Now, let's enable HTTP keep-alive on both Apache and NGINX.

Apache

In Apache, keep-alive can be enabled in two ways. You can enable it either in the `.htaccess` file or in the Apache config file.

To enable it in the `.htaccess` file, place the following configuration in the `.htaccess` file:

```
<ifModule mod_headers.c>
  Header set Connection keep-alive
</ifModule>
```

In the preceding configuration, we set the Connection header to keep-alive in the `.htaccess` file. As the `.htaccess` configuration overrides the configuration in the config files, this will override whatever configuration is made for keep-alive in the Apache config file.

To enable the keep-alive connection in the Apache config file, we have to modify three configuration options. Search for the following configuration and set the values to the ones in the example:

```
KeepAlive On
MaxKeepAliveRequests 100
KeepAliveTimeout 100
```

In the preceding configuration, we turned on the keep-alive configuration by setting the value of `KeepAlive` to `On`.

The next is `MaxKeepAliveRequests`, which defines the maximum number of keep-alive connections to the web server at the time. A value of 100 is the default in Apache, and it can be changed according to the requirements. For high performance, this value should be kept high. If set to 0, it will allow unlimited keep-alive connections, which is not recommended.

The last configuration is `KeepAliveTimeout`, which is set to 100 seconds. This defines the number of seconds to wait for the next request from the same client on the same TCP connection. If no request is made, then the connection is closed.

NGINX

HTTP keep-alive is part of the `http_core` module and is enabled by default. In the NGINX configuration file, we can edit a few options, such as timeout. Open the `nginx` config file, edit the following configuration options, and set its values to the following:

```
keepalive_requests 100
keepalive_timeout 100
```

The `keepalive_requests` config defines the maximum number of requests a single client can make on a single HTTP keep-alive connection.

The `keepalive_timeout` config is the number of seconds that the server needs to wait for the next request until it closes the keep-alive connection.

GZIP compression

Content compression provides a way to reduce the contents' size delivered by the HTTP server. Both Apache and NGINX provide support for GZIP compression, and similarly, most modern browsers support GZIP. When the GZIP compression is enabled, the HTTP server sends compressed HTML, CSS, JavaScript, and images that are small in size. This way, the contents are loaded fast.

A web server only compresses content via GZIP when the browser sends information about itself that it supports GZIP compression. Usually, a browser sends such information in *Request* headers.

The following are codes for both Apache and NGINX to enable GZIP compression.

Apache

The following code can be placed in the `.htaccess` file:

```
<IfModule mod_deflate.c>
SetOutputFilter DEFLATE
 #Add filters to different content types
AddOutputFilterByType DEFLATE text/html text/plain
text/xml    text/css text/javascript
application/javascript
    #Don't compress images
    SetEnvIfNoCase Request_URI \.(?:gif|jpe?
g|png)$ no-gzip dont-
    vary
</IfModule>
```

In the preceding code, we used the Apache `deflate` module to enable compression. We used filter by type to compress only certain types of files, such as `.html`, plain text, `.xml`, `.css`, and `.js`. Also, before ending the module, we set a case to not compress the images because compressing images can cause image quality degradation.

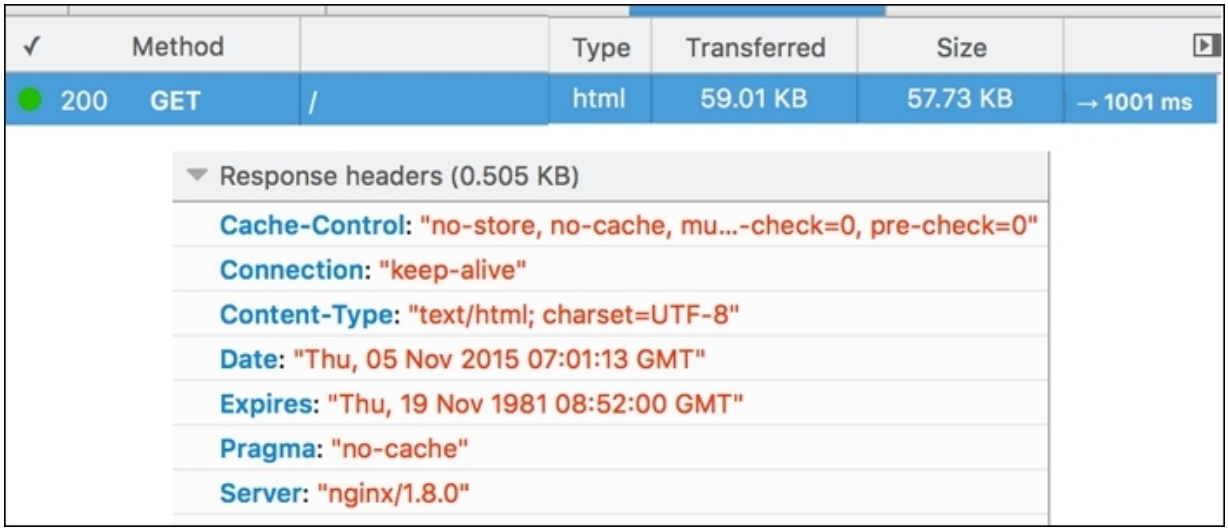
NGINX

As mentioned previously, you have to place the following code in your virtual host conf file for NGINX:

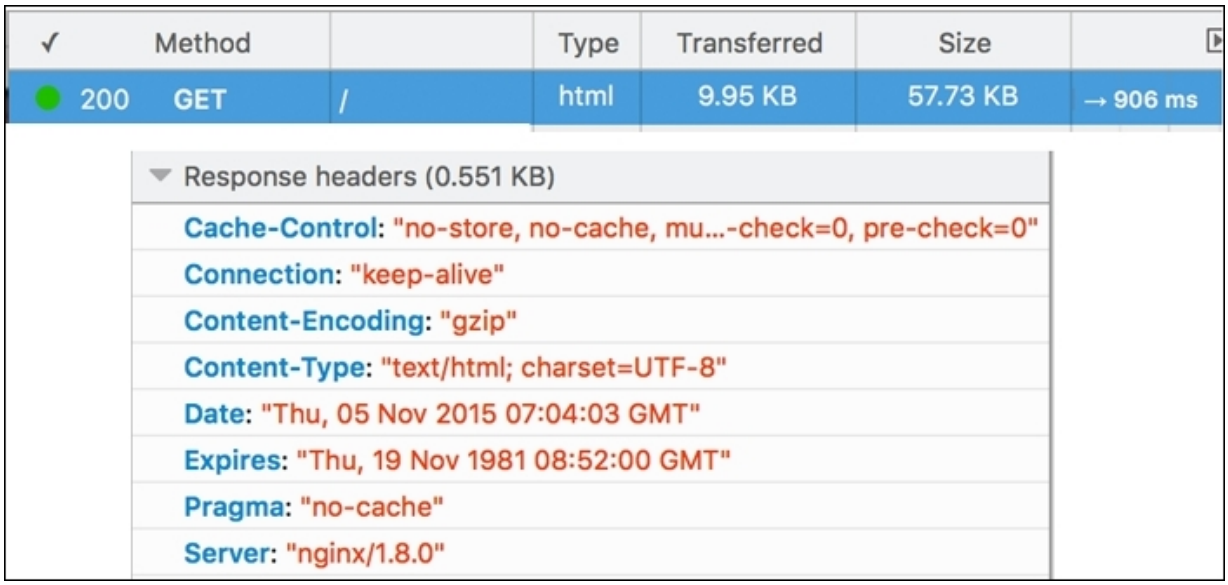
```
gzip on;
gzip_vary on;
gzip_types text/plain text/xml text/css
text/javascript application/x-javascript;
gzip_com_level 4;
```

In the preceding code, GZIP compression is activated by the `gzip on;` line. The `gzip_vary on;` line is used to enable varying headers. The `gzip_types` line is used to define the types of files to be compressed. Any file types can be added depending on the requirements. The `gzip_com_level 4;` line is used to set the compression level, but be careful with this value; you don't want to set it too high. Its range is from 1 to 9, so keep it in the middle.

Now, let's check whether the compression really works. In the following screenshot, the request is sent to a server that does not have GZIP compression enabled. The size of the final HTML page downloaded or transferred is 59 KB:



After enabling GZIP compression on the web server, the size of the transferred HTML page is reduced up to 9.95 KB, as shown in the following screenshot:



Also, it can be noted that the time to load the contents is also reduced. So, the smaller the size of your contents, the faster the page will load.

Using PHP as a separate service

Apache uses the `mod_php` module for PHP. This way, the PHP interpreter is integrated to Apache, and all processing is done by this Apache module, which eats up more server hardware resources. It is possible to use PHP-FPM with Apache, which uses the FastCGI protocol and runs in a separate process. This enables Apache to worry about HTTP request handlings, and the PHP processing is made by the PHP-FPM.

NGINX, on the other hand, does not provide any built-in support or any support by module for PHP processing. So, with NGINX, PHP is always used in a separate service.

Now, let's take a look at what happens when PHP runs as a separate service: the web server does not know how to process the dynamic content request and forwards the request to another external service, which reduces the processing load on the web server.

Disabling unused modules

Both Apache and NGINX come with lots of modules built into them. In most cases, you won't need some of these modules. It is good practice to disable these modules.

It is good practice to make a list of the modules that are enabled, disable those modules one by one, and restart the server. After this, check whether your application is working or not. If it works, go ahead; otherwise, enable the module(s) after which the application stopped working properly again.

This is because you may see that a certain module may not be required, but some other useful module depends on this module. So, it's best practice it to make a list and enable or disable the modules, as stated before.

Apache

To list all the modules that are loaded for Apache, issue the following command in the terminal:

```
sudo apachectl -M
```

This command will list all the loaded modules, as can be seen in the following screenshot:



Now, analyze all the loaded modules, check whether they are needed for the application, and disable them, as follows.

Open up the Apache config file and find the section where all the modules are loaded. A sample is included here:

```
LoadModule access_compat_module
modules/mod_access_compat.so
LoadModule actions_module modules/mod_actions.so
```

```
LoadModule alias_module modules/mod_alias.so
LoadModule allowmethods_module
modules/mod_allowmethods.so
LoadModule asis_module modules/mod_asis.so
LoadModule auth_basic_module
modules/mod_auth_basic.so
#LoadModule auth_digest_module
modules/mod_auth_digest.so
#LoadModule auth_form_module
modules/mod_auth_form.so
#LoadModule authn_anon_module
modules/mod_authn_anon.so
```

The modules that have a `#` sign in front of them are not loaded. So, to disable a module in the complete list, just place a `#` sign. The `#` sign will comment out the line, and the module won't be loaded anymore.

NGINX

To check which modules NGINX is compiled with, issue the following command in the terminal:

```
sudo Nginx -V
```

This will list complete information about the NGINX installation, including the version and modules with which NGINX is compiled. Have a look at the following screenshot:



Normally, NGINX enables only those modules that are required for NGINX to work. To enable any other module that is compiled with NGINX installed, we can place a little configuration for it in the `nginx.conf` file, but there is no single way to disable any NGINX module. So, it is good to search for this specific module and take a look at the module page on the NGINX website. There, we can find information about this specific module, and if available, we can find information about how to disable and configure this module.

Web server resources

Each web server comes with its own optimum settings for general use. However, these settings may be not optimum for your current server hardware. The biggest problem on the web server hardware is the RAM. The more RAM the server has, the more the web server will be able to handle requests.

NGINX

NGINX provides two variables to adjust the resources, which are `worker_processes` and `worker_connections`. The `worker_processes` settings decide how many NGINX processes should run.

Now, how many `worker_processes` resources should we use? This depends on the server. Usually, it is one worker processes per processor core. So, if your server processor has four cores, this value can be set to 4.

The value of `worker_connections` shows the number of connections per `worker_processes` setting per second. Simply speaking, `worker_connections` tells NGINX how many simultaneous requests can be handled by NGINX. The value of `worker_connections` depends on the system processor core. To find out the core's limitations on a Linux system (Debian/Ubuntu), issue the following command in the terminal:

```
Ulimit -n
```

This command will show you a number that should be used for `worker_connections`.

Now, let's say that our processor has four cores, and each core's limitation is 512. Then, we can set the values for these two variables in the NGINX main configuration file. On Debian/Ubuntu, it is located at `/etc/nginx/nginx.conf`.

Now, find out these two variables and set them as follows:

```
Worker_processes 4;
Worker_connections 512
```

The preceding values can be high, specially `worker_connections`, because server processor cores have high limitations.

Content Delivery Network (CDN)

Content Delivery Network is used to host static media files, such as images, `.css` and `.js` files, and audio and video files. These files are stored on a geographical network whose servers are located in different locations. Then, these files are served to requests from a specific server, depending on the request location.

CDN provides the following features:

- As the contents are static, which don't change frequently, CDN caches them in memory. When a request comes for a certain file, CDN sends the file directly from cache, which is faster than loading the file from disk and sending it to the browser.
- CDN servers are located in different locations. All the files are stored in each location, depending on your settings in CDN. When a browser request arrives to CDN, CDN sends the requested contents from the nearest location available to the requested location. For example, if the CDN has servers in London, New York, and Dubai and a request comes from Middle East, the CDN will send content from the Dubai server. This way, as a CDN delivers the contents from the nearest location, the response time is reduced.
- Each browser has limitations for sending simultaneous requests to a domain. Mostly, it's three requests. When a response arrives for a request, the browser sends more requests to the same domain, which causes a delay in complete page loading. CDN provides subdomains (either their own subdomains or your main domain's subdomains, using your main domain's DNS settings), which enables browsers to send more parallel requests for the same contents loading from different domains. This enables the browser to load the page content fast.
- Generally, there is a small amount of requests for dynamic content and more requests for static content. If your

application's static content is hosted on a separate CDN server, this will reduce the load on your server tremendously.

Using CDN

So, how do you use CDN in your application? In best practice, if your application has high traffic, creating different subdomains at your CDN for each content type is the best. For example, a separate domain for CSS and JavaScript files, a subdomain for images, and another separate subdomain for audio/videos files can be created. This way, the browser will send parallel requests for each content type. Let's say, we have the following URLs for each content type:

- **For CSS and JavaScript**: `http://css-js.yourcdn.com`
- **For images**: `http://images.yourcdn.com`
- **For other media**: `http://media.yourcdn.com`

Now, most open source applications provide settings at their admin control panel to set up CDN URLs, but in case you happened to use an open source framework or a custom-build application, you can define your own setting for CDN by placing the previous URLs either in the database or in a configuration file loaded globally.

For our example, we will place the preceding URLs in a config file and create three constants for them, as follows:

```
Constant('CSS_JS_URL', 'http://css-
js.yourcdn.com/');
Constant('IMAGES_URL',
'http://images.yourcdn.com/');
Constant('MEDiA_URL', 'http://css-
js.yourcdn.com/');
```

If we need to load a CSS file, it can be loaded as follows:

```
<script type="text/javascript" src="<?php echo
CSS_JS_URL ?>js/file.js"></script>
```

For a JavaScript file, it can be loaded as follows:

```
<link rel="stylesheet" type="text/css" href="<?php
echo CSS_JS_URL ?>css/file.css" />
```

If we load images, we can use the previous way in the `src` attribute of the `img` tag, as follows:

```
<img src="<?php echo IMAGES_URL ?
>images/image.png" />
```

In the preceding examples, if we don't need to use CDN or want to change the CDN URLs, it will be easy to change in just one place.

Most famous JavaScript libraries and templating engines host their static resources on their own personal CDN. Google hosts query libraries, fonts, and other JavaScript libraries on its own CDN, which can be used directly in applications.

Sometimes, we may not want to use CDN or be able to afford them. For this, we can use a technique called domain sharing. Using domain sharding, we can create subdomains or point out other domains to our resources' directories on the same server and application. The technique is the same as discussed earlier; the only difference is that we direct other domains or subdomains to our media, CSS, JavaScript, and image directories ourselves.

This may seem be fine, but it won't provide us with CDN's best performance. This is because CDN decides the geographical availability of content depending on the customer's location, extensive caching, and files optimization on the fly.

CSS and JavaScript optimization

Every web application has CSS and JavaScript files. Nowadays, it is common that most applications have lots of CSS and JavaScript files to make the application attractive and interactive. Each CSS and JavaScript file needs a browser to send a request to the server to fetch the file. So, the more the CSS and JavaScript files you have, the more requests the browser will need to send, thus affecting its performance.

Each file has a content size, and it takes time for the browser to download it. For example, if we have 10 CSS files of 10 KB each and 10 JavaScript files of 50 KB each, the total content size of the CSS files is 100 KB, and for JavaScript it is 500 KB—600 KB for both types of files. This is too much, and the browser will take time to download them.

> **Note**
>
> Performance plays a vital role in web applications. Even Google counts performance in its indexing. Don't think of a file that has a few KBs and takes a 1 ms to download because when it comes to performance, each millisecond is counted. The best thing is to optimize, compress, and cache everything.

In this section, we will discuss two ways to optimize our CSS and JS, which are as follows:

- Merging
- Minifying

Merging

In the merging process, we can merge all the CSS files into a single file, and the same process is carried out with JavaScript files, thus creating a single file for CSS and JavaScript. If we have 10 files for CSS, the browser sends 10 requests for all these files. However, if we merge them in a single file, the browser will send only one request, and thus, the time taken for nine requests is saved.

Minifying

In the minifying process, all the empty lines, comments, and extra spaces are removed from the CSS and JavaScript files. This way, the size of the file is reduced, and the file loads fast.

For example, let's say you have the following CSS code in a file:

```
.header {
  width: 1000px;
  height: auto;
  padding: 10px
}

/* move container to left */
.float-left {
  float: left;
}

/* Move container to right */
.float-right {
  float: right;
}
```

After minifying the file, we will have CSS code similar to the following:

```
.header{width:100px;height:auto;padding:10px}.floa
t-left{float:left}.float-right{float:right}
```

Similarly for JavaScript, let's consider that we have the following code in a JavaScript file:

```
/* Alert on page load */
$(document).ready(function() {
  alert("Page is loaded");
});

/* add three numbers */
function addNumbers(a, b, c) {
  return a + b + c;
}
```

Now, if the preceding file is minified, we will have the following code:

```
$(document).ready(function(){alert("Page is
loaded")});function addNumbers(a,b,c){return
a+b+c;}
```

It can be noted in the preceding examples that all the unnecessary white spaces and new lines are removed. Also, it places the complete file code in one single line. All code comments are removed. This way, the file size is reduced, which helps the file be loaded fast. Also, this file will consume less bandwidth, which is useful if the server resources are limited.

Most open source applications, such as Magento, Drupal, and WordPress, provide either built-in support or support the application by third-party plugins/modules. Here, we won't cover how to merge CSS or JavaScript files in these applications, but we will discuss a few tools that can merge CSS and JavaScript files.

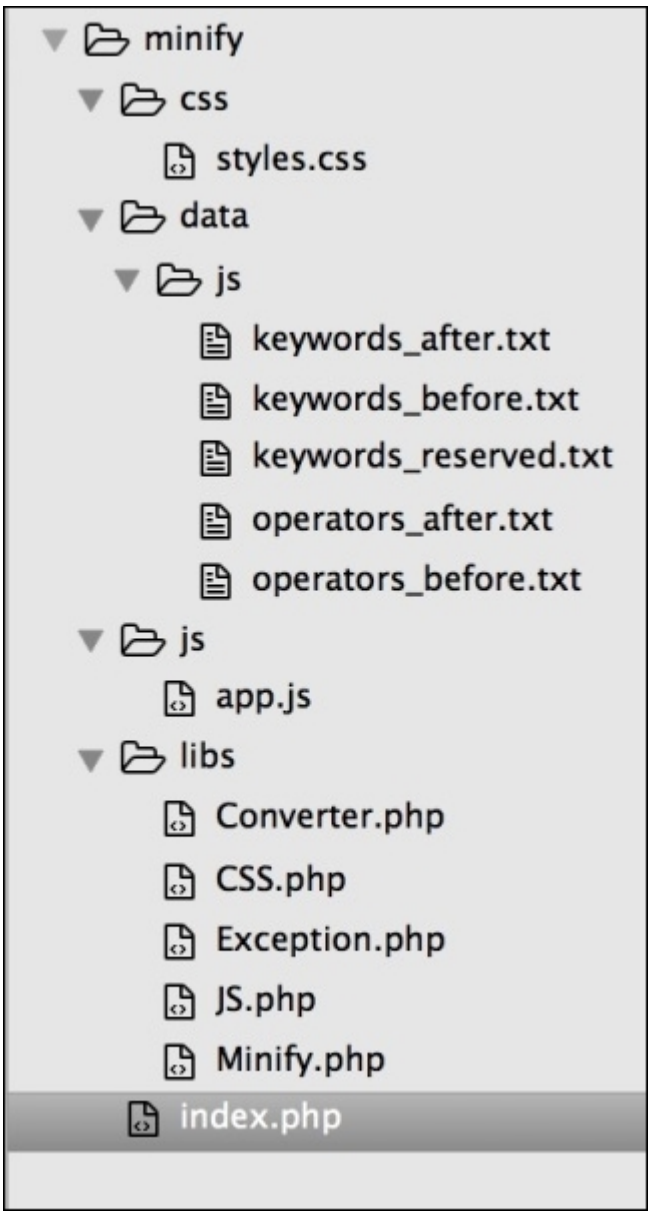
Minify

Minify is a set of libraries completely written in PHP. Minify supports both merging and minifying for both CSS and JavaScript files. Its code is completely object-oriented and namespaced, so it can be embedded into any current or proprietary framework.

Note

The Minify homepage is located at http://minifier.org. It is also hosted on GitHub at https://github.com/matthiasmullie/minify. It is important to note that the Minify library uses a path converter library, which is written by the same author. The path converter library can be downloaded from https://github.com/matthiasmullie/path-converter. Download this library and place it in the same folder as the minify libraries.

Now, let's create a small project that we will use to minify and merge CSS and JavaScript files. The folder structure of the project will be as in the following screenshot:



In the preceding screenshot, the complete project structure is shown. The project name is `minify`. The `css` folder has all of our CSS files, including the minified or merged ones. Similarly, the `js` folder has all our JavaScript files, including the minified or merged ones. The `libs` folder has the `Minify` library along with the `Converter` library. `Index.php` has our main code to minify and merge CSS and JavaScript files.

Note

The `data` folder in the project tree is related to JavaScript minification. As JavaScript has keywords that require a space before and after them, these `.txt` files are used to identify these operators.

So, let's start by minifying our CSS and JavaScript files using the following code in `index.php`:

```
include('libs/Converter.php');
include('libs/Minify.php');
include('libs/CSS.php');
include('libs/JS.php');
include('libs/Exception.php');

use MatthiasMullie\Minify;

/* Minify CSS */
$cssSourcePath = 'css/styles.css';
$cssOutputPath = 'css/styles.min.css';
$cssMinifier = new Minify\CSS($cssSourcePath);
$cssMinifier->minify($cssOutputPath);

/* Minify JS */
$jsSourcePath = 'js/app.js';
$jsOutputPath = 'js/app.min.js';
$jsMinifier = new Minify\JS($jsSourcePath);
$jsMinifier->minify($jsOutputPath);
```

The preceding code is simple. First, we included all our required libraries. Then, in the `Minify CSS` block, we created two path variables: `$cssSourcePath`, which has the path to the CSS file that we need to minify, and `$cssOutputPath`, which has path to the minified CSS file that will be generated.

After this, we instantiated an object of the `CSS.php` class and passed the CSS file that we need to minify. Finally, we called the minify method of the `CSS` class and passed the output path along with the filename, which will generate the required file for us.

The same explanation goes for the JS minifying process.

If we run the preceding PHP code, all the files are in place, and everything goes fine, then two new filenames will be created: `styles.min.css` and `app.min.js`. These are the new minified versions of their original files.

Now, let's use Minify to merge multiple CSS and JavaScript files. First, add some CSS and JavaScript files to the respective folders in the project. After this, we just need to add a little code to the current code. In the following code, I will skip including all the libraries, but these files have to be loaded whenever you need to use Minify:

```
/* Minify CSS */
$cssSourcePath = 'css/styles.css';
$cssOutputPath = 'css/styles.min.merged.css';
$cssMinifier = new Minify\CSS($cssSourcePath);
$cssMinifier->add('css/style.css');
$cssMinifier->add('css/forms.js');
$cssMinifier->minify($cssOutputPath);

/* Minify JS */
$jsSourcePath = 'js/app.js';
$jsOutputPath = 'js/app.min.merged.js';
$jsMinifier = new Minify\JS($jsSourcePath);
$jsMinifier->add('js/checkout.js');
$jsMinifier->minify($jsOutputPath);
```

Now, take a look at the highlighted code. In the CSS part, we saved the minified and merged file as `style.min.merged.css`, but naming is not important; it is all up to our own choice.

Now, we will simply use the add method of the `$cssMinifier` and `$jsMinifier` objects to add new files and then call `minify`. This causes all the additional files to be merged in the initial file and then minified, thus generating a single merged and minified file.

Grunt

According to its official website, Grunt is a JavaScript task runner. It automates certain repetitive tasks so that you don't have to work

repeatedly. It is an awesome tool and is widely used among web programmers.

Installing Grunt is very easy. Here, we will install it on MAC OS X, and the same method is used for most Linux systems, such as Debian and Ubuntu.

Note

Grunt requires Node.js and npm. Installing and configuring Node.js and npm is out of the scope of this book, so for this book, we will assume that these tools are installed on your machine or that you can search for them and figure out how to install them.

If Node.js and npm are installed on your machine, just fire up the following command in your terminal:

```
sudo npm install -g grunt
```

This will install Grunt CLI. If everything goes fine, then the following command will show you the version the of Grunt CLI:

```
grunt -version
```

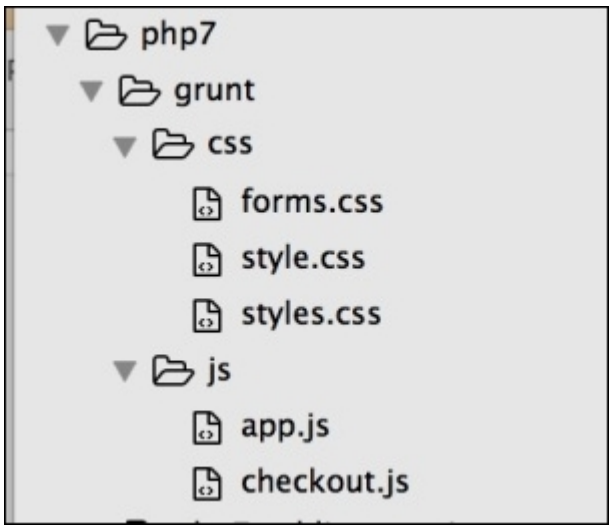
The output of the preceding command is `grunt-cli v0.1.13;` as of writing this book, this version is available.

Grunt provides you with a command-line, which enables you to run a Grunt command. A Grunt project requires two files in your project file tree. One is `package.json`, which is used by `npm` and lists Grunt and the Grunt plugins that the project needs as DevDependencies.

The second file is the `GruntFile`, which is stored as `GruntFile.js` or `GruntFile.coffee` and is used to configure and define Grunt tasks

and load Grunt plugins.

Now, we will use the same preceding project, but our folder structure will be as follows:



Now, open the terminal in your project root and issue the following command:

```
sudo npm init
```

This will generate the `package.json` file by asking a few questions. Now, open the `package.json` file and modify it so that the contents of the final `package.json` files look similar to the following:

```
{
  "name" : "grunt"  //Name of the project
  "version : "1.0.0" //Version of the project
  "description" : "Minify and Merge JS and CSS
file",
  "main" : "index.js",
  "DevDependencies" : {
    "grunt" : "0.4.1", //Version of Grunt

    //Concat plugin version used to merge css and
js files
    "grunt-contrib-concat" : "0.1.3"

    //CSS minifying plugin
```

```
    "grunt-contrib-cssmin" : "0.6.1",

    //Uglify plugin used to minify JS files.
    "grunt-contrib-uglify" : "0.2.0"

   },
"author" : "Altaf Hussain",
"license" : ""
}
```

I added comments to different parts of the `package.json` file so that it is easy to understand. Note that for the final file, we will remove the comments from this file.

It can be seen that in the `DevDependencies` section, we added three Grunt plugins used for different tasks.

The next step is to add `GruntFile`. Let's create a file called `GruntFile.js` in our project root similar to the `package.json` file. Place the following contents in `GruntFile`:

```
module.exports = function(grunt) {
   /*Load the package.json file*/
   pkg: grunt.file.readJSON('package.json'),
  /*Define Tasks*/
  grunt.initConfig({
    concat: {
      css: {
        src: [
        'css/*' //Load all files in CSS folder
],
         dest: 'dest/combined.css' //Destination
of the final combined file.

      }, //End of CSS
js: {
      src: [
     'js/*' //Load all files in js folder
],
      dest: 'dest/combined.js' //Destination of
the final combined file.

    }, //End of js
```

```
}, //End of concat
cssmin:  {
  css: {
    src : 'dest/combined.css',
    dest : 'dest/combined.min.css'
}
},//End of cssmin
uglify: {
  js: {
    files: {
      'dest/combined.min.js' :
['dest/combined.js'] // destination Path : [src
path]
    }
  }
} //End of uglify

}); //End of initConfig

grunt.loadNpmTasks('grunt-contrib-concat');
grunt.loadNpmTasks('grunt-contrib-uglify');
grunt.loadNpmTasks('grunt-contrib-cssmin');
grunt.registerTask('default', ['concat:css',
'concat:js', 'cssmin:css', 'uglify:js']);

}; //End of module.exports
```

The preceding code is simple and self-explanatory, and the comments are added whenever needed. At the top, we loaded our `package.json` file, and after this, we defined different tasks along with their src and destination files. Remember that every task's src and destination syntax is different, and it depends on the plugin. After `initConfig` block, we loaded different plugins and npm tasks and then registered them with GRUNT.

Now, let's run our tasks.

First, let's combine CSS and JavaScript files and store them in their respective destinations defined in our tasks list in GruntFile via the following command:

```
grunt concat
```

After running the preceding command in your terminal, if you see a message such as `Done, without errors`, then the task is completed successfully.

In the same way, let's minify our css file using the following command:

```
grunt cssmin
```

Then, we will minify our JavaScript file using the following command:

```
grunt uglify
```

Now, it may seem like a lot of work to use Grunt, but it provides some other features that can make a developer's life easy. For example, what if you need to change your JavaScript and CSS files? Should you run all the preceding commands again? No, Grunt provides a watch plugin, which activates and executes all the files in the destination paths in the tasks, and if any changes occur, it runs the tasks automatically.

For a more detailed learning, take a look at Grunt's official website at http://gruntjs.com/.

Full page caching

In full page caching, the complete page of the website is stored in a cache, and for the next requests, this cached page is served. Full page cache is more effective if your website content does not change too often; for example, on a blog with simple posts, new posts are added on a weekly basis. In this case, the cache can be cleared after new posts are added.

What if you have a website that has pages with dynamic parts, such as an e-commerce website? In this case, a complete page caching will create problems because the page is always different for each request; as a user is logged in, he/she may add products to the shopping cart and so on. In this case, using full page caching may not be that easy.

Most popular platforms provide either built-in support for full page cache or through plugins and modules. In this case, the plugin or module takes care of the dynamic blocks of the page for each request.

Varnish

Varnish, as mentioned on its official website, makes your website fly; and this is true! Varnish is an open source web application accelerator that runs in front of your web server software. It has to be configured on port 80 so that each request comes to it.

Now, the Varnish configuration file (called VCL files with the `.vcl` extenstion) has a definition for backends. A backend is the web server (Apache or NGINX) configured on another port (let's say 8080). Multiple backends can be defined, and Varnish will take care of the load balancing too.

When a request comes to Varnish, it checks whether the data for this request in available at its cache or not. If it finds the data in its cache, this cached data is returned to the request, and no request is sent to the web server or backend. If Varnish does not find any data in its cache, it sends a request to the web server and requests the data. When it receives data from the web server, it first caches this data and then sends it back to the request.

As it is clear in the preceding discussion, if Varnish finds the data in the cache, there is no need for a request to the web server and, therefore, for processing in there, and the response is sent back very fast.

Varnish also provides features such as load balancing and health checks. Also, Varnish has no support for SSL and cookies. If Varnish receives cookies from the web server or backend, this page is not cached. There are different ways to overcome these issues easily.

We've done enough theory; now, let's install Varnish on a Debian/Ubuntu server via the following steps:

1. First, add the Varnish repositories to the `sources.list` file. Place the following line in the file:

```
    deb https://repo.varnish-cache.org/debian/
Jessie varnish-4.1
```

2. After this, issue the following command to update the repositories:

```
sudo apt-get update
```

3. Now, issue the following command:

```
sudo apt-get install varnish
```

4. This will download and install Varnish. Now, the first thing to do is configure Varnish to listen at port 80 and make your web server listen at another port, such as 8080. We will configure it here with NGINX.
5. Now, open the Varnish configuration file location at `/etc/default/varnish` and change it so that it looks similar to the following code:

```
DAEMON_OPS="-a :80 \
  -T localhost:6082 \
  -f /etc/varnish/default.vcl \
  -S /etc/varnish/secret \
  -s malloc,256m"
```

6. Save the file and restart Varnish by issuing the following command in the terminal:

```
sudo service varnish restart
```

7. Now our Varnish runs on port `80`. Let's make NGINX run on port `8080`. Edit the NGINX `vhost` file for the application and change the listen port from `80` to `8080`, as follows:

```
listen 8080;
```

8. Now, restart NGINX by issuing the following command in the terminal:

```
sudo service nginx restart
```

9. The next step is to configure the Varnish VCL file and add a backend that will communicate with our backend on port `8080`. Edit the Varnish VCL file located at `/etc/varnish/default.vcl`, as follows:

```
backend default {
  .host = "127.0.0.1";
  .port = "8080";
}
```

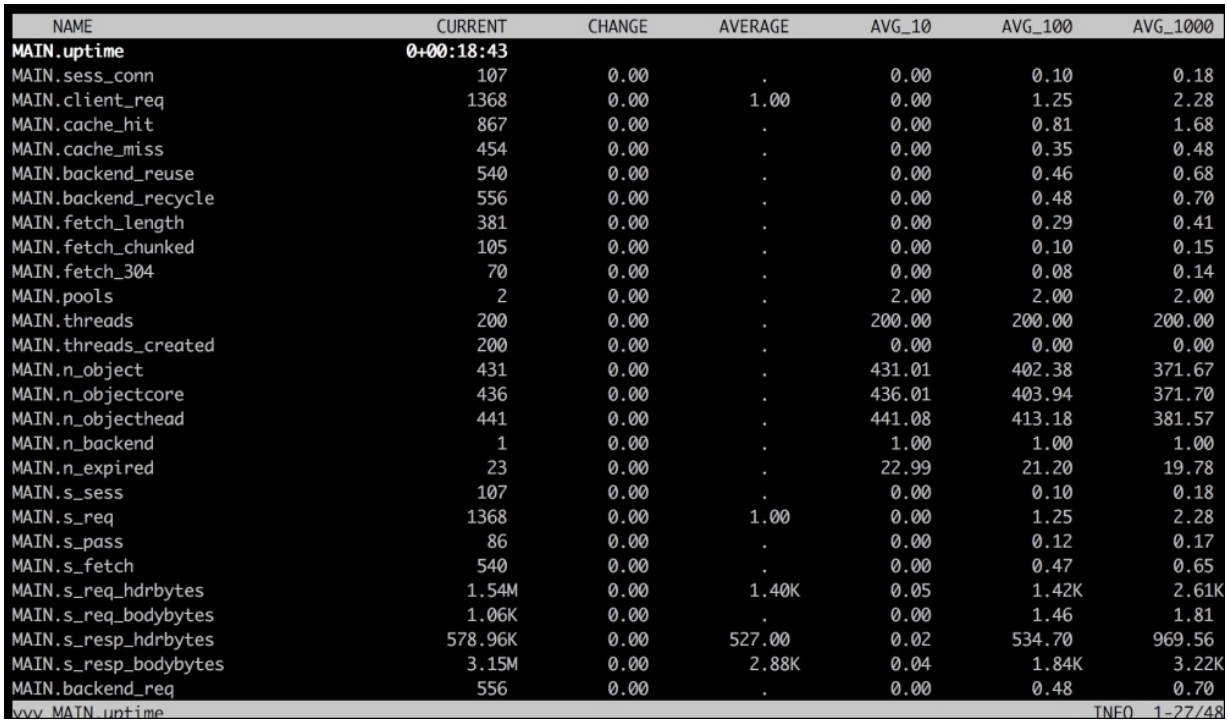
In the preceding configuration, our backend host is located at the same server on which Varnish runs, so we entered the local IP. We can also enter a localhost in this case. However, if our backend runs on a remote host or another server, the IP of this server should be entered.

Now, we are done with Varnish and web server configuration. Restart both Varnish and NGINX. Open your browser and enter the IP or hostname of the server. The first response may seem slow, which is because Varnish is fetching data from the backend and then caching it, but other subsequent responses will be extremely fast, as Varnish cached them and is now sending back the cached data without communicating with the backend.

Varnish provides a tool in which we can easily monitor the Varnish cache status. It is a real-time tool and updates its contents in real time. It is called varnishstat. To start varnishstat, just issue the following command in the terminal:

```
varnishstat
```

The preceding command will display a session similar to the following screenshot:



NAME	CURRENT	CHANGE	AVERAGE	AVG_10	AVG_100	AVG_1000
MAIN.uptime	**0+00:18:43**					
MAIN.sess_conn	107	0.00	.	0.00	0.10	0.18
MAIN.client_req	1368	0.00	1.00	0.00	1.25	2.28
MAIN.cache_hit	867	0.00	.	0.00	0.81	1.68
MAIN.cache_miss	454	0.00	.	0.00	0.35	0.48
MAIN.backend_reuse	540	0.00	.	0.00	0.46	0.68
MAIN.backend_recycle	556	0.00	.	0.00	0.48	0.70
MAIN.fetch_length	381	0.00	.	0.00	0.29	0.41
MAIN.fetch_chunked	105	0.00	.	0.00	0.10	0.15
MAIN.fetch_304	70	0.00	.	0.00	0.08	0.14
MAIN.pools	2	0.00	.	2.00	2.00	2.00
MAIN.threads	200	0.00	.	200.00	200.00	200.00
MAIN.threads_created	200	0.00	.	0.00	0.00	0.00
MAIN.n_object	431	0.00	.	431.01	402.38	371.67
MAIN.n_objectcore	436	0.00	.	436.01	403.94	371.70
MAIN.n_objecthead	441	0.00	.	441.08	413.18	381.57
MAIN.n_backend	1	0.00	.	1.00	1.00	1.00
MAIN.n_expired	23	0.00	.	22.99	21.20	19.78
MAIN.s_sess	107	0.00	.	0.00	0.10	0.18
MAIN.s_req	1368	0.00	1.00	0.00	1.25	2.28
MAIN.s_pass	86	0.00	.	0.00	0.12	0.17
MAIN.s_fetch	540	0.00	.	0.00	0.47	0.65
MAIN.s_req_hdrbytes	1.54M	0.00	1.40K	0.05	1.42K	2.61K
MAIN.s_req_bodybytes	1.06K	0.00	.	0.00	1.46	1.81
MAIN.s_resp_hdrbytes	578.96K	0.00	527.00	0.02	534.70	969.56
MAIN.s_resp_bodybytes	3.15M	0.00	2.88K	0.04	1.84K	3.22K
MAIN.backend_req	556	0.00	.	0.00	0.48	0.70
vvv MAIN.uptime						INFO 1-27/48

As can be seen in the preceding screenshot, it displays very useful information, such as the running time and the number of requests made at the beginning, cache hits, cache misses, all backends, backend reusages, and so on. We can use this information to tune Varnish for its best performance.

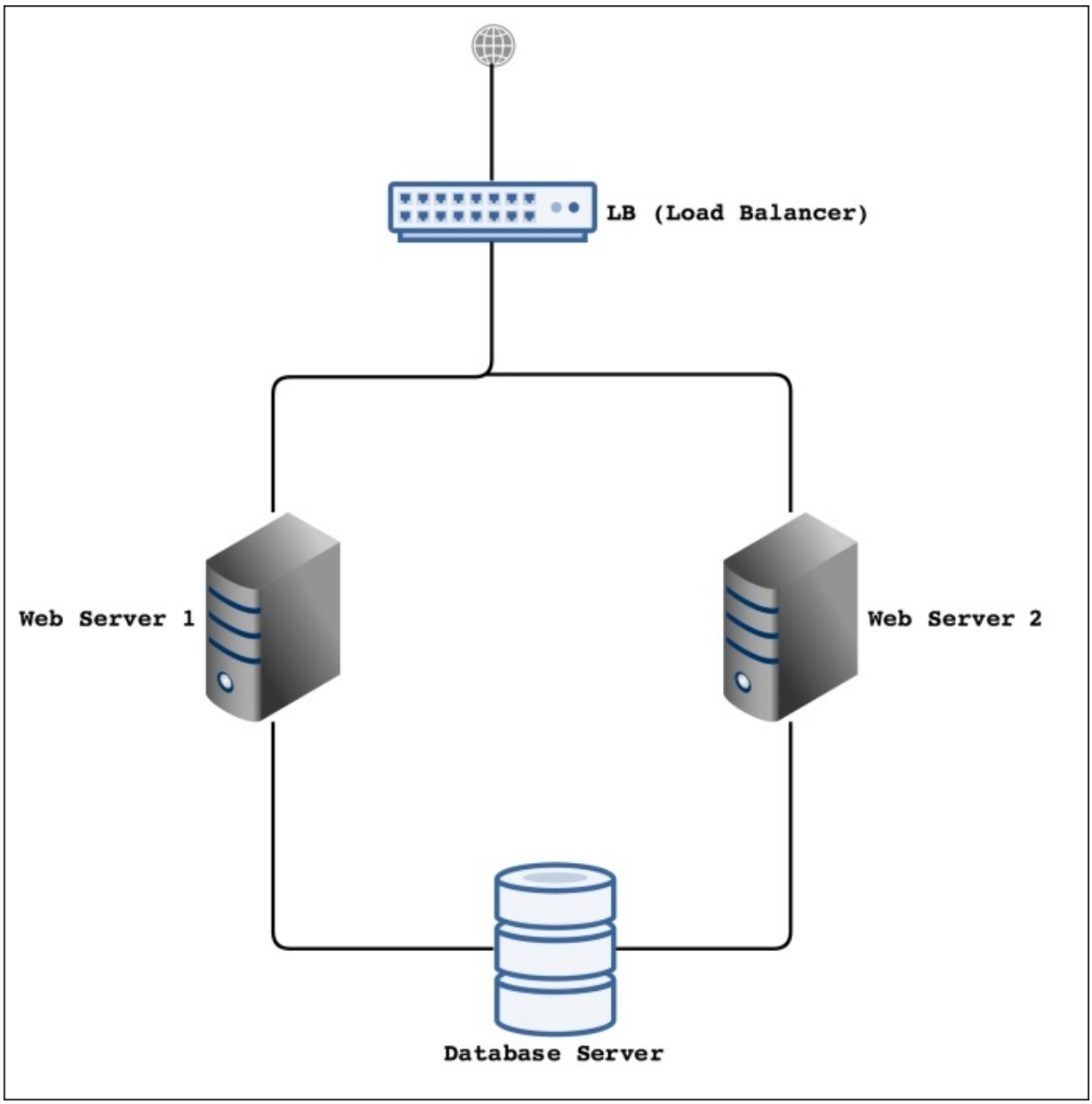
Note

A complete Varnish configuration is out of the scope of this book, but a good documentation can be found on the Varnish official website at https://www.varnish-cache.org.

The infrastructure

We discussed too many topics on increasing the performance of our application. Now, let's discuss the scalability and availability of our application. With time, the traffic on our application can increase to thousands of users at a time. If our application runs on a single server, the performance will be hugely effected. Also, it is not a good idea to keep the application running at a single point because in case this server goes down, our complete application will be down.

To make our application more scalable and better in availability, we can use an infrastructure setup in which we can host our application on multiple servers. Also, we can host different parts of the application on different servers. To better understand, take a look at the following diagram:



This is a very basic design for the infrastructure. Let's talk about its different parts and what operations will be performed by each part and server.

Note

It is possible that only the Load Balancer (LB) will be connected to the public Internet, and the rest of the parts can be connected to each through a private network in a Rack. If a Rack is

available, this will be very good because all the communication between all the servers will be on a private network and therefore secure.

Web servers

In the preceding diagram, we have two web servers. There can be as many web servers as needed, and they can be easily connected to LB. The web servers will host our actual application, and the application will run on NGINX or Apache and PHP 7. All the performance tunings we will discuss in this chapter can be used on these web servers. Also, it is not necessary that these servers should be listening at port 80. It is good that our web server should listen at another port to avoid any public access using browsers.

The database server

The database server is mainly used for the database where the MySQL or Percona Server can be installed. However, one of the problems in the infrastructure setup is to store session data in a single place. For this purpose, we can also install the Redis server on the database server, which will handle our application's session data.

The preceding infrastructure design is not a final or perfect design. It is just to give the idea of a multiserver application hosting. It has room for a lot of improvement, such as adding another local balancer, more web servers, and servers for the database cluster.

Load balancer (LB)

The first part is the **load balancer** (**LB**). The purpose of the load balancer is to divide the traffic among the web servers according to the load on each web server.

For the load balancer, we can use HAProxy, which is widely used for this purpose. Also, HAProxy checks the health of each web server, and if a web server is down, it automatically redirects the traffic of this down web server to other available web servers. For this purpose, only LB will be listening at port 80.

We don't want to place a load on our available web servers (in our case, two web servers) of encrypting and decrypting the SSL communication, so we will use the HAProxy server to terminate SSL there. When our LB receives a request with SSL, it will terminate SSL and send a normal request to one of the web servers. When it receives a response, HAProxy will encrypt the response and send it back to the client. This way, instead of using both the servers for SSL encryption/decryption, only a single LB server will be used for this purpose.

> **Note**
>
> Varnish can be also used as a load balancer, but this is not a good idea because the whole purpose of Varnish is HTTP caching.

HAProxy load balancing

In the preceding infrastructure, we placed a load balancer in front of our web servers, which balance load on each server, check the health of each server, and terminate SSL. We will install HAProxy and configure it to achieve all the configurations mentioned before.

HAProxy installation

We will install HAProxy on Debian/Ubuntu. As of writing this book, HAProxy 1.6 is the latest stable version available. Perform the following steps to install HAProxy:

1. First, update the system cache by issuing the following command in the terminal:

```
sudo apt-get update
```

2. Next, install HAProxy by entering the following command in the terminal:

```
sudo apt-get install haproxy
```

   This will install HAProxy on the system.

3. Now, confirm the HAProxy installation by issuing the following command in the terminal:

```
haproxy -v
```



If the output is as in the preceding screenshot, then congratulations! HAProxy is installed successfully.

HAProxy load balancing

Now, it's time to use HAProxy. For this purpose, we have the following three servers:

- The first is a load balancer server on which HAProxy is installed. We will call it LB. For this book's purpose, the IP of the LB server is 10.211.55.1. This server will listen at port 80, and all HTTP requests will come to this server. This server also acts as a frontend server as all the requests to our application will come to this server.

- The second is a web server, which we will call Web1. NGINX, PHP 7, MySQL, or Percona Server are installed on it. The IP of this server is 10.211.55.2. This server will either listen at port 80 or any other port. We will keep it to listen at port 8080.
- The third is a second web server, which we will call Web2, with the IP 10.211.55.3. This has the same setup as of the Web1 server and will listen at port 8080.

The Web1 and Web2 servers are also called backend servers. First, let's configure the LB or frontend server to listen at port 80.

Open the `haproxy.cfg` file located at `/etc/haproxy/` and add the following lines at the end of the file:

```
frontend http
  bind *:80
  mode http
  default_backend web-backends
```

In the preceding code, we set HAProxy to listen at the HTTP port 80 on any IP address, either the local loopback IP 127.0.0.1 or the public IP. Then, we set the default backend.

Now, we will add two backend servers. In the same file, at the end, place the following code:

```
backend web-backend
  mode http
  balance roundrobin
  option forwardfor
  server web1 10.211.55.2:8080 check
  server web2 10.211.55.3:8080 check
```

In the preceding configuration, we added two servers into the web backend. The reference name for the backend is `web-backend`, which is used in the frontend configuration too. As we know, both our web servers listen at port 8080, so we mentioned that it is the definition of each web server. Also, we used `check` at the end of the definition of each web server, which tells HAProxy to check the server's health.

Now, restart HAProxy by issuing the following command in the terminal:

```
sudo service haproxy restart
```

Note

To start HAProxy, we can use the `sudo service haproxy start` command. To stop HAProxy, we can use the `sudo service haproxy stop` command.

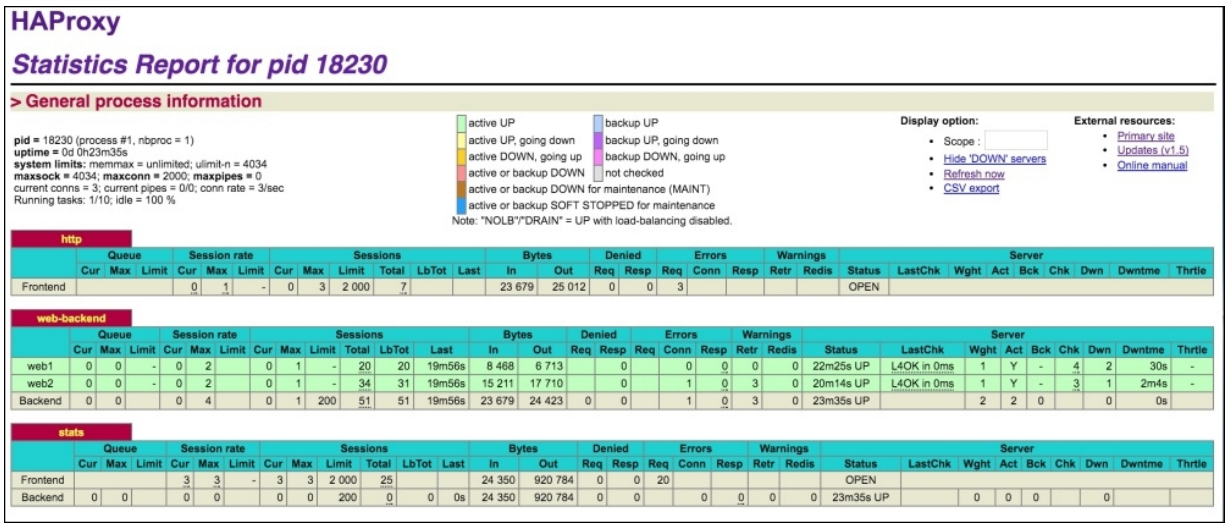
Now, enter the IP or hostname of the LB server in the browser, and our web application page will be displayed either from Web1 or Web2.

Now, disable any of the web servers and then reload the page again. The application will still work fine, because HAProxy automatically detected that one of web servers is down and redirected the traffic to the second web server.

HAProxy also provides a stats page, which is browser-based. It provides complete monitoring information about the LB and all the backends. To enable stats, open `haprox.cfg`, and place the following code at the end of the file:

```
listen stats *:1434
  stats enable
  stats uri /haproxy-stats
  stats auth phpuser:packtPassword
```

The stats are enabled at port `1434`, which can be set to any port. The URL of the page is `stats uri`. It can be set to any URL. The `auth` section is for basic HTTP authentication. Save the file and restart HAProxy. Now, open the browser and enter the URL, such as `10.211.55.1:1434/haproxy-stats`. The stats page will be displayed as follows:



HAProxy

Statistics Report for pid 18230

> General process information

pid = 18230 (process #1, nbproc = 1)
uptime = 0d 0h23m35s
system limits: memmax = unlimited; ulimit-n = 4034
maxsock = 4034; maxconn = 2000; maxpipes = 0
current conns = 3; current pipes = 0/0; conn rate = 3/sec
Running tasks: 1/10; idle = 100 %

active UP
active UP, going down
active DOWN, going up
active or backup DOWN
active or backup DOWN for maintenance (MAINT)
active or backup SOFT STOPPED for maintenance
backup UP
backup UP, going down
backup DOWN, going up
not checked
Note: "NOLB"/"DRAIN" = UP with load-balancing disabled.

Display option:
- Scope :
- Hide 'DOWN' servers
- Refresh now
- CSV export

External resources:
- Primary site
- Updates (v1.5)
- Online manual

http

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server									
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
Frontend				0	1	-	0	3	2 000	7			23 679	25 012	0	0	3					OPEN								

web-backend

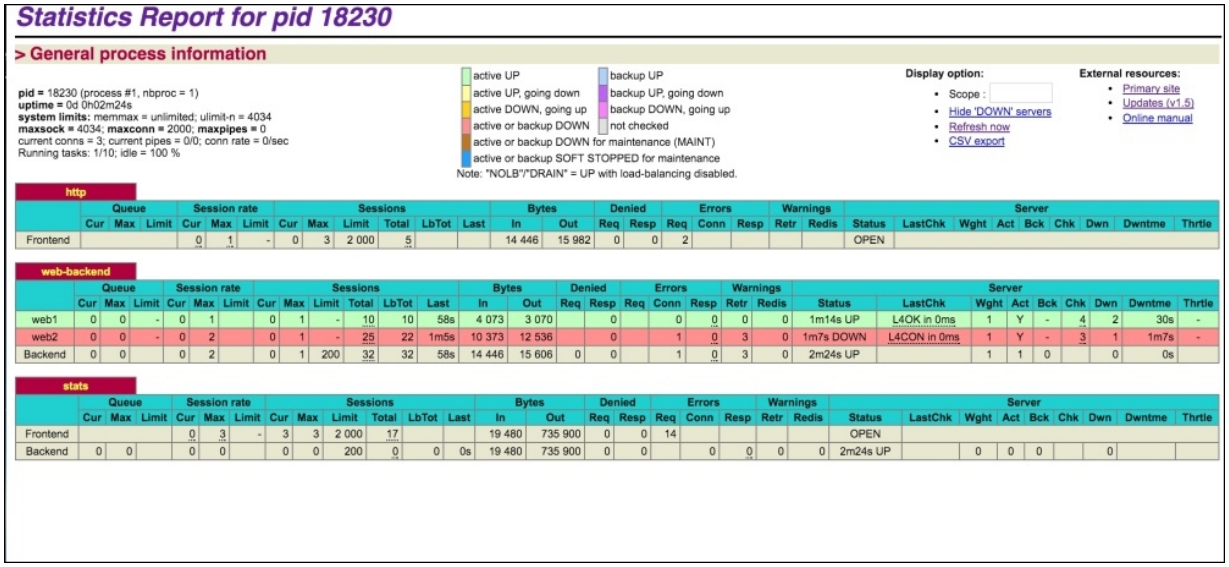
	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server									
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
web1	0	0	-	0	2		0	1	-	20	20	19m56s	8 468	6 713		0		0	0	0	0	22m25s UP	L4OK in 0ms	1	Y	-	4	2	30s	-
web2	0	0	-	0	2		0	1	-	34	31	19m56s	15 211	17 710		0		1	0	3	0	20m14s UP	L4OK in 0ms	1	Y	-	3	1	2m4s	-
Backend	0	0		0	4		0	1	200	51	51	19m56s	23 679	24 423	0	0		1	0	3	0	23m35s UP		2	2	0		0	0s	

stats

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server									
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
Frontend				3	3	-	3	3	2 000	25			24 350	920 784	0	0	20					OPEN								
Backend	0	0		0	0		0	0	200	0	0	0s	24 350	920 784	0	0		0	0	0	0	23m35s UP		0	0	0		0		

In the preceding screenshot, each backend web server can be seen, including frontend information.

Also, if a web server is down, HAProxy stats will highlight the row for this web server, as can be seen in the following screenshot:



Statistics Report for pid 18230

> General process information

pid = 18230 (process #1, nbproc = 1)
uptime = 0d 0h02m24s
system limits: memmax = unlimited; ulimit-n = 4034
maxsock = 4034; maxconn = 2000; maxpipes = 0
current conns = 3; current pipes = 0/0; conn rate = 0/sec
Running tasks: 1/10; idle = 100 %

active UP
active UP, going down
active DOWN, going up
active or backup DOWN
active or backup DOWN for maintenance (MAINT)
active or backup SOFT STOPPED for maintenance
backup UP
backup UP, going down
backup DOWN, going up
not checked
Note: "NOLB"/"DRAIN" = UP with load-balancing disabled.

Display option:
- Scope :
- Hide 'DOWN' servers
- Refresh now
- CSV export

External resources:
- Primary site
- Updates (v1.5)
- Online manual

http

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server									
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
Frontend				0	1	-	0	3	2 000	5			14 446	15 982	0	0	2					OPEN								

web-backend

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server									
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
web1	0	0	-	0	1		0	1	-	10	10	58s	4 073	3 070		0		0	0	0	0	1m14s UP	L4OK in 0ms	1	Y	-	4	2	30s	-
web2	0	0	-	0	2		0	1	-	25	22	1m5s	10 373	12 536		0		1	0	3	0	1m7s DOWN	L4CON in 0ms	1	Y	-	3	1	1m7s	-
Backend	0	0		0	2		0	1	200	32	32	58s	14 446	15 606	0	0		1	0	3	0	2m24s UP		1	1	0		0	0s	

stats

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server									
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
Frontend				0	3	-	3	3	2 000	17			19 480	735 900	0	0	14					OPEN								
Backend	0	0		0	0		0	0	200	0	0	0s	19 480	735 900	0	0		0	0	0	0	2m24s UP		0	0	0		0		

For our test, we stopped NGINX at our Web2 server and refreshed the stats page, and the Web2 server row in the backend section was highlighted.

To terminate SSL using HAProxy, it is pretty simple. To terminate SSL using HAProxy, we will just add the SSL port 443 binding along

with the SSL certificate file location. Open the `haproxy.cfg` file, edit the frontend block, and add the highlighted code in it, as in the following block:

```
frontend http
bind *:80
bind *:443 ssl crt /etc/ssl/www.domain.crt
  mode http
  default_backend web-backends
```

Now, HAProxy also listens at 443, and when an SSL request is sent to it, it processes it there and terminates it so that no HTTPS requests are sent to backend servers. This way, the load of SSL encryption/decryption is removed from the web servers and is managed by the HAProxy server only. As SSL is terminated at the HAProxy server, there is no need for web servers to listen at port 443, as regular requests from HAProxy server are sent to the backend.

Summary

In this chapter, we discussed several topics starting from NGINX and Apache to Varnish. We discussed how we can optimize our web server's software settings for the best performance. Also, we discussed CDNs and how to use them in our customer applications. We discussed two ways to optimize JavaScript and CSS files for the best performance. We briefly discussed full page cache and Varnish installation and configuration. At the end, we discussed multiserver hosting or infrastructure setup for our application to be scalable and the best in availability.

In next chapter, we will look into the ways of increasing the performance of our database. We will discuss several topics, including the Percona Server, different storage engines for the database, query caching, Redis, and Memcached.

Chapter 4. Improving Database Performance

Databases play a key role in dynamic websites. All incoming and outgoing data is stored in a database. So, if the database for a PHP application is not well designed and optimized, it will effect the application's performance tremendously. In this chapter, we will look into the ways of optimizing our PHP application's database. The following topics will be covered in this chapter:

- MySQL
- Query caching
- The MyISAM and InnoDB storage engines
- The Percona DB and Percona XtraDB storage engines
- MySQL performance monitoring tools
- Redis
- Memcached

The MySQL database

MySQL is the most commonly used **Relational Database Management System** (**RDMS**) for the Web. It is open source and has a free community version. It provides all those features that can be provided by an enterprise-level database.

The default settings provided with the MySQL installation may not be so good for performance, and there are always ways to fine-tune these settings to get an improved performance. Also, remember that your database design plays a big role in performance. A poorly designed database will have an effect on the overall performance.

In this section, we will discuss how to improve the MySQL database's performance.

Note

We will modify the MySQL configuration's `my.cnf` file. This file is located in different places in different operating systems. Also, if you are using XAMPP, WAMP, or any other cross-platform web server solution stack package on Windows, this file will be located in the respective folder. Whenever `my.cnf` is mentioned, it is assumed that the file is open no matter which OS is used.

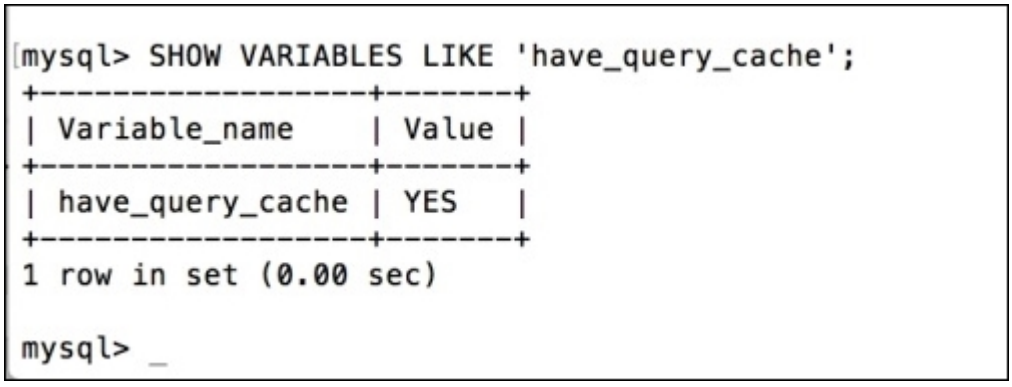
Query caching

Query caching is an important performance feature of MySQL. It caches `SELECT` queries along with the resulting dataset. When an identical `SELECT` query occurs, MySQL fetches the data from memory so that the query is executed faster and thus reduces the load on the database.

To check whether query cache is enabled on a MySQL server or not, issue the following command in your MySQL command line:

```
SHOW VARIABLES LIKE 'have_query_cache';
```

The preceding command will display the following output:



The previous result set shows that query cache is enabled. If query cache is disabled, the value will be `NO`.

To enable query caching, open up the `my.cnf` file and add the following lines. If these lines are there and are commented, just uncomment them:

```
query_cache_type = 1
query_cache_size = 128MB
query_cache_limit = 1MB
```

Save the `my.cnf` file and restart the MySQL server. Let's discuss what the preceding three configurations mean:

- `query_cache_type`: This plays a little confusing role.
    - If `query_cache_type` is set to `1` and `query_cache_size` is 0, then no memory is allocated, and query cache is disabled.

      If `query_cache_size` is greater than 0, then query cache is enabled, memory is allocated, and all queries that do not exceed the `query_cache_limit` value or use the `SQL_NO_CACHE` option are cached.
    - If the `query_cache_type` value is 0 and `query_cache_size` is `0`, then no memory is allocated, and cache is disabled.

      If `query_cache_size` is greater than 0, then memory is allocated, but nothing is cached—that is, cache is disabled.
- `query_cache_size`: `query_cache_size`: This indicates how much memory will be allocated. Some think that the more memory is used, the better it will be, but this is just a misunderstanding. It all depends on the database size, query types and ratios between read and writes, hardware, database traffic, and other factors. A good value for `query_cache_size` is between 100 MB and 200 MB; then, you can monitor the performance and other variables on which query cache depends, as mentioned, and adjust the size. We have used 128MB for a medium traffic Magento website and it is working perfectly. Set this value to `0` to disable query cache.

- `query_cache_limit`: This defines the maximum size of a query dataset to be cached. If a query dataset's size is larger than this value, it isn't cached. The value of this configuration can be guessed by finding out the largest `SELECT` query and the size of its returned dataset.

Storage engines

Storage engines (or table types) are a part of core MySQL and are responsible for handling operations on tables. MySQL provides several storage engines, and the two most widely used are MyISAM and InnoDB. Both these storage engines have their own pros and cons, but InnoDB is always prioritized. MySQL started using InnoDB as the default storage engine, starting from 5.5.

> **Note**
>
> MySQL provides some other storage engines that have their own purposes. During the database design process, which table should use which storage engine can be decided. A complete list of storage engines for MySQL 5.6 can be found at http://dev.mysql.com/doc/refman/5.6/en/storage-engines.html.

A storage engine can be set at database level, which is then used as the default storage engine for each newly created table. Note that the storage engine is the table's base, and different tables can have different storage engines in a single database. What if we have a table already created and want to change its storage engine? It is easy. Let's say that our table name is `pkt_users`, its storage engine is MyISAM, and we want to change it to InnoDB; we will use the following MySQL command:

```
ALTER TABLE pkt_users ENGINE=INNODB;
```

This will change the storage engine value of the table to `INNODB`.

Now, let's discuss the difference between the two most widely used storage engines: MyISAM and InnoDB.

The MyISAM storage engine

A brief list of features that are or are not supported by MyISAM is as follows:

- MyISAM is designed for speed, which plays best with the `SELECT` statement.
- If a table is more static—that is, the data in this table is less frequently updated/deleted and mostly only fetched—then MyISAM is the best option for this table.
- MyISAM supports table-level locking. If a specific operation needs to be performed on the data in a table, then the complete table can be locked. During this lock, no operations can be performed on this table. This can cause performance degradation if the table is more dynamic—that is, if the data is frequently changed in this table.
- MyISAM does not have support for foreign keys.
- MyISAM supports full-text search.
- MyISAM does not support transactions. So, there is no support for `COMMIT` and `ROLLBACK`. If a query on a table is executed, it is executed, and there is no coming back.
- Data compression, replication, query caching, and data encryption is supported.
- The cluster database is not supported.

The InnoDB storage engine

A brief list of features that are or are not supported by InnoDB is as follows:

- InnoDB is designed for high reliability and high performance when processing a high volume of data.
- InnoDB supports row-level locking. It is a good feature and is great for performance. Instead of locking the complete table as with MyISAM, it locks only the specific row for the `SELECT`, `DELETE`, or `UPDATE` operations, and during these operations, other data in this table can be manipulated.
- InnoDB supports foreign keys and forcing foreign keys constraints.

- Transactions are supported. COMMIT and ROLLBACK are possible, so data can be recovered from a specific transaction.
- Data compression, replication, query caching, and data encryption is supported.
- InnoDB can be used in a cluster environment, but it does not have full support. However, InnoDB tables can be converted to the NDB storage engine, which is used in the MySQL cluster by changing the table engine to NDB.

In the following sections, we will discuss some more performance features that are related to InnoDB. Values for the following configuration are set in the `my.cnf` file.

innodb_buffer_pool_size

This setting defines how much memory should be used for InnoDB data and the indices loaded into memory. For a dedicated MySQL server, the recommended value is 50-80% of the installed memory on the server. If this value is set too high, there will be no memory left for the operating system and other subsystems of MySQL, such as transaction logs. So, let's open our `my.cnf` file, search for `innodb_buffer_pool_size`, and set the value between the recommended value (that is, 50-80%) of our RAM.

innodb_buffer_pool_instances

This feature is not that widely used. It enables multiple buffer pool instances to work together to reduce the chances of memory contentions on a 64-bit system and with a large value for `innodb_buffer_pool_size`.

There are different choices on which the value for `innodb_buffer_pool_instances` are calculated. One way is to use one instance per GB of `innodb_buffer_pool_size`. So, if the value of `innodb_bufer_pool_size` is 16 GB, we will set `innodb_buffer_pool_instances` to 16.

innodb_log_file_size

The `innodb_log_file_size` is the the size of the log file that stores every query information executed. For a dedicated server, a value up to 4 GB is safe, but the time taken for crash recovery may increase if the log file's size is too large. So, in best practice, it is kept in between 1 and 4 GB.

The Percona Server - a fork of MySQL

According to the Percona website, Percona is a free, fully compatible, enhanced, open source, and drop-in replacement for MySQL that provides superior performance, scalability, and instrumentation.

Percona is a fork of MySQL with enhanced features for performance. All the features available in MySQL are available in Percona. Percona uses an enhanced storage engine called XtraDB. According to the Percona website, it is an enhanced version of the InnoDB storage engine for MySQL that has more features, faster performance, and better scalability on modern hardware. Percona XtraDB uses memory more efficiently in high-load environments.

As mentioned earlier, XtraDB is a fork of InnoDB, so all the features available in InnoDB are available in XtraDB.

Installing the Percona Server

Percona is only available for Linux systems. It is not available for Windows as of now. In this book, we will install Percona Server on Debian 8. The process is same for both Ubuntu and Debian.

Note

To install the Percona Server on other Linux flavors, check out the Percona installation manual at https://www.percona.com/doc/percona-server/5.5/installation.html. As of now, they provide instructions for Debian, Ubuntu, CentOS, and RHEL. They also provide instructions to install the Percona Server from sources and Git.

Now, let's install the Percona Server through the following steps:

1. Open your sources list file using the following command in your terminal:

```
sudo nano /etc/apt/sources.list
```

   If prompted for a password, enter your Debian password. The file will be opened.

2. Now, place the following repository information at the end of the `sources.list` file:

```
deb http://repo.percona.com/apt jessie main
deb-src http://repo.percona.com/apt jessie
main
```

3. Save the file by pressing *CTRL* + *O* and close the file by pressing *CTRL* + *X*.
4. Update your system using the following command in the terminal:

```
sudo apt-get update
```

5. Start the installation by issuing the following command in the terminal:

```
sudo apt-get install percona-server-server-5.5
```

6. The installation will be started. The process is the same as the MySQL server installation. During the installation, the root password for the Percona Server will be asked; you just need to enter it. When the installation is complete, you will be ready to use the Percona Server in the same way as MySQL.
7. Configure the Percona Server and optimize it as discussed in the earlier sections.

MySQL performance monitoring tools

There is always a need to monitor the performance of database servers. For this purpose, there are many tools available that make it easy to monitor MySQL servers and performance. Most of them are open source and free, and some provide a GUI. The command-line tools are more powerful and the best to use, though it takes a little time to understand and get used to them. We will discuss a few here.

phpMyAdmin

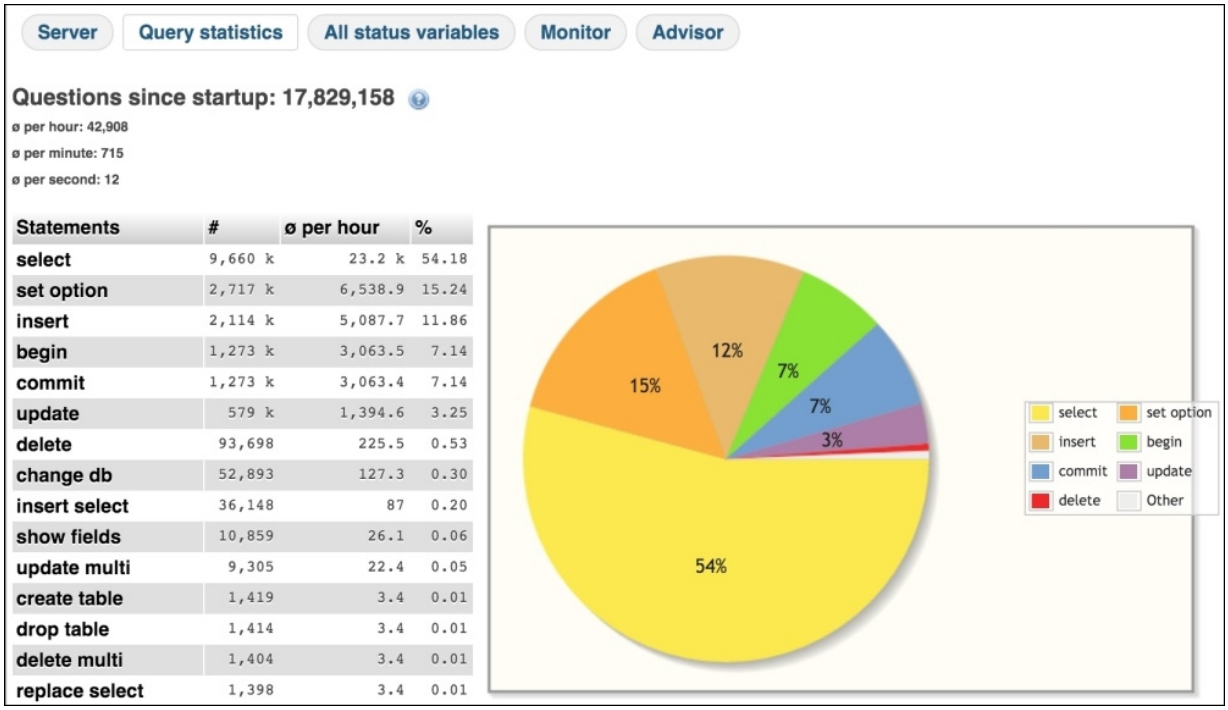
This is the most famous, web-based, open source, and free tool available to manage MySQL databases. Despite managing a MySQL server, it also provides some good tools to monitor a MySQL server. If we log in to phpMyAdmin and then click on the **Status** tab at the top, we will see the following screen:



The **Server** tab shows us basic data about the MySQL server, such as when it started, how much traffic is handled from the last start, information about connections, and so on.

The next is **Query Statistics**. This section provides full stats about all of the queries executed. It also provides a pie chart, which visualizes the percentage of each query type, as shown in the following screenshot.

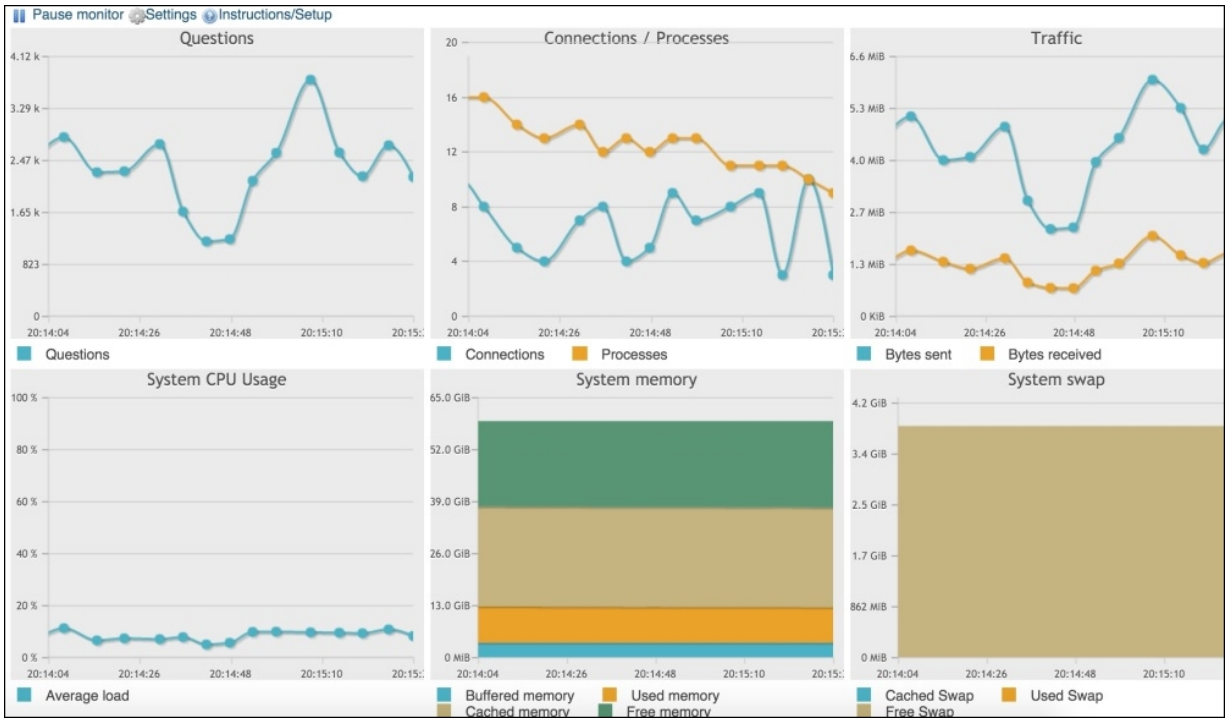
If we carefully look at the chart, we can see that we have 54% of the `SELECT` queries running. If we use some kind of cache, such as Memcached or Redis, these `SELECT` queries should not be this high. So, this graph and statistics information provides us with a mean to analyze our cache systems.



The next option is **All Status Variables**, which lists all of the MySQL variables and their current values. In this list, one can easily find out how MySQL is configured. In the following screenshot, our query cache variables and their values are shown:

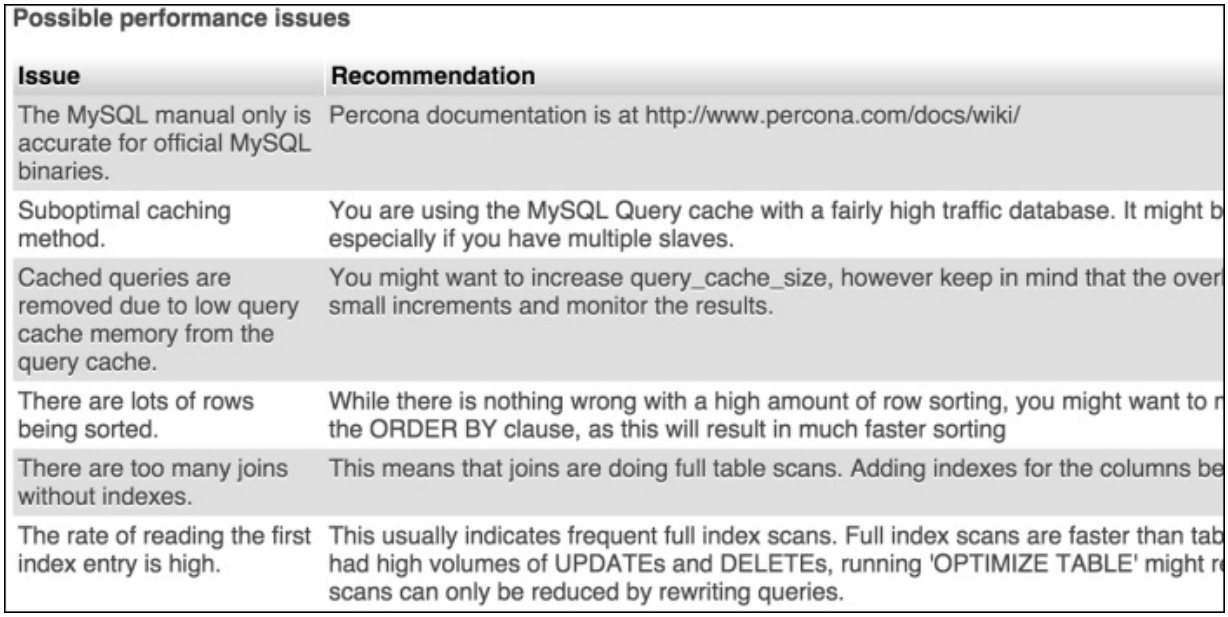
Qcache free blocks	6.1 k	The number of free memc issuing a FLUSH QUERY
Qcache free memory	17 M	The amount of free memc
Qcache hits	261.1 M	The number of cache hits
Qcache inserts	9.4 M	The number of queries ad
Qcache lowmem prunes	5.6 M	The number of queries tha help you tune the query ca from the cache.
Qcache not cached	252.3 k	The number of non-cache
Qcache queries in cache	24 k	The number of queries reg
Qcache total blocks	60.2 k	The total number of block
Queries	280.3 M	
Questions	280.3 M	

The next option that phpMyAdmin provides is **Monitor**. This is a very powerful tool that displays the server resources and their usages in real time in a graphical way.



As shown in the preceding screenshot, we can see **Questions**, **Connections/Processes**, **System CPU Usage**, **Traffic**, **System Memory**, and **System swap** in a nice graphical interface.

The last important section is **Advisor**. This gives us advice regarding the settings for performance. It gives you as many details as possible so that the MySQL server can be tuned for performance. A small section from the advisor section is shown in the following screenshot:



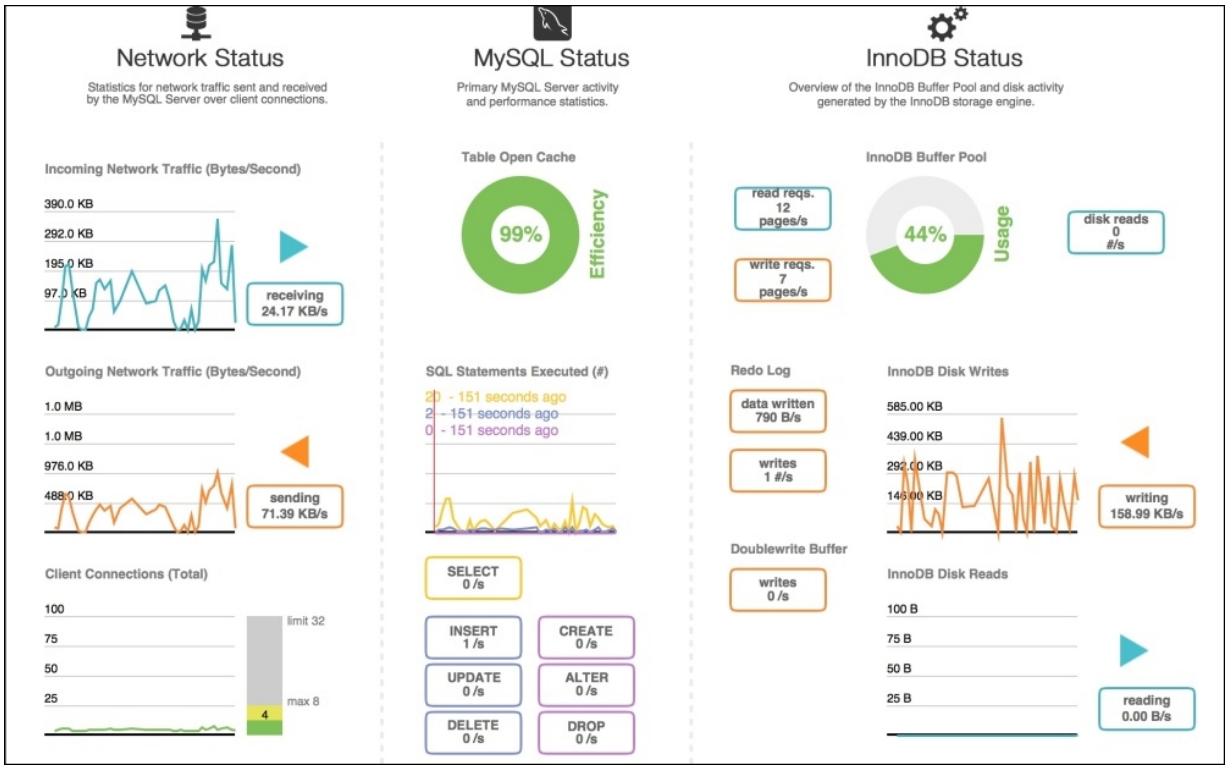
Possible performance issues

Issue	Recommendation
The MySQL manual only is accurate for official MySQL binaries.	Percona documentation is at http://www.percona.com/docs/wiki/
Suboptimal caching method.	You are using the MySQL Query cache with a fairly high traffic database. It might b especially if you have multiple slaves.
Cached queries are removed due to low query cache memory from the query cache.	You might want to increase query_cache_size, however keep in mind that the over small increments and monitor the results.
There are lots of rows being sorted.	While there is nothing wrong with a high amount of row sorting, you might want to r the ORDER BY clause, as this will result in much faster sorting
There are too many joins without indexes.	This means that joins are doing full table scans. Adding indexes for the columns be
The rate of reading the first index entry is high.	This usually indicates frequent full index scans. Full index scans are faster than tab had high volumes of UPDATEs and DELETEs, running 'OPTIMIZE TABLE' might r scans can only be reduced by rewriting queries.

If all these advices are applied, some performance can be gained.

The MySQL workbench

This is a desktop application from MySQL and is fully equipped with tools to manage and monitor the MySQL server. It provides us with a dashboard for performance in which all the data related to the server can be seen in a beautiful and graphical way, as shown in the screenshot that follows:



Percona Toolkit

All the tools mentioned before are good and provide some visual information about our database server. However, they are not good enough to show us some more useful information or provide more features that can make our lives easy. For this purpose, another command-line toolkit is available, which is called Percona Toolkit.

Percona Toolkit is a set of more than 30 command-line tools, which includes those used to do an analysis of slow queries, archive, optimize indices and many more.

Note

Percona Toolkit is free and open source and is available under GPL. Most of its tools run on Linux/Unix-based systems, but some can run on Windows too. An installation guide can be found at https://www.percona.com/doc/percona-

toolkit/2.2/installation.html. A complete set of tools can be found at https://www.percona.com/doc/percona-toolkit/2.2/index.html.

Now, let's discuss a few tools in the subsections to follow.

pt-query-digest

This tool analyzes queries from slow, general, and binary log files. It generates a sophisticated report about the queries. Let's run this tool for slow queries using the following command:

```
Pt-query-digest /var/log/mysql/mysql-slow.log
```

After entering the preceding command in the terminal, we will see a long report. Here, we will discuss a short part of the report, as shown in the following screenshot:

```
# Profile
# Rank Query ID           Response time    Calls R/Call  V/M   Item
# ==== ================== ================ ===== ======= ===== =======
#    1 0xCEB312D0FA1C37CE 10683.7659 11.9%   165 64.7501 70.89 SELECT
#    2 0xA836779D3D007C7C 10323.4901 11.5%   142 72.7006 92.79 SELECT
#    3 0x0C7EA293C3196265  9889.3175 11.0%   147 67.2743 14... SELECT
#    4 0xD70662D2ECA11099  8618.3030  9.6%   147 58.6279 14... SELECT
#    5 0xEE6E30152233C978  5687.1127  6.3%   134 42.4411 38.55 SELECT
#    6 0x7C28ED0781B7DC05  5220.0295  5.8%   129 40.4653 41.13 SELECT
#    7 0x813031B8BBC3B329  4895.0396  5.4%  2926  1.6729  0.19 COMMIT
#    8 0x71CE1B0B17DCC70F  3179.1896  3.5%   151 21.0542 19.94 SELECT
#    9 0x1E3F38720A7F9E01  2805.9224  3.1%   126 22.2692 21.57 SELECT
#   10 0x86485AAB2E3523AB  1951.5777  2.2%   374  5.2181 18.41 INSERT
#   11 0x5F9B42BF4A256B2E  1806.6999  2.0%    69 26.1841 10.16 INSERT
#   12 0x2CE0A1392B331E05  1713.9884  1.9%   558  3.0717  4.03 SELECT
```

In the preceding screenshot, slow queries are listed with the slowest at the top. The first query, which is a `SELECT` query, takes the most time, which is about 12% of the total time. The second query, which is also a `SELECT` query, takes 11.5% of the total time. From this report, we can see which queries are slow so that we can optimize them for the best performance.

Also, pt-query-digest displays information for each query, as shown in the following screenshot. In the screenshot, data about the first query is mentioned, including the total timing; percentage (pct) of time; min, max, and average time; bytes sent; and some other parameters:

```
# Query 1: 0.00 QPS, 0.00x concurrency, ID 0xCEB312D0FA1C37CE at byte 7534347
# This item is included in the report because it matches --limit.
# Scores: V/M = 70.89
# Time range: 2014-12-21 11:14:08 to 2015-11-14 11:24:17
# Attribute    pct   total     min     max     avg     95%  stddev  median
# ============ === ======= ======= ======= ======= ======= ======= =======
# Count          0     165
# Exec time     11  10684s      5s    647s     65s    159s     68s     45s
# Lock time      0    16ms       0   184us    94us   144us    23us    84us
# Rows sent     10  47.61M       0 368.14k 295.45k 362.29k  66.84k 298.06k
# Rows examine   5  47.61M       0 368.14k 295.45k 362.29k  66.84k 298.06k
# Rows affecte   0       0       0       0       0       0       0       0
# Bytes sent    13   1.20G   2.77M  31.57M   7.46M   8.03M   5.35M   7.29M
# Merge passes   0       0       0       0       0       0       0       0
# Tmp tables     0       0       0       0       0       0       0       0
# Tmp disk tbl   0       0       0       0       0       0       0       0
# Tmp tbl size   0       0       0       0       0       0       0       0
# Query size     0   8.70k      54      54      54      54       0      54
# InnoDB:
```

pt-duplicate-key-checker

This tool finds duplicate indices and duplicate foreign keys either in a set of specified tables or in a complete database. Let's execute this tool again in a large database using the following command in the terminal:

```
Pt-duplicate-key-checker –user packt –password
dbPassword –database packt_pub
```

When executed, the following output is printed:

```
# ########################################################################
# [redacted]_live.widget_instance_page_layout
# ########################################################################

# IDX_WIDGET_INSTANCE_PAGE_LAYOUT_LAYOUT_UPDATE_ID is a left-prefix of UNQ_WIDGET_INSTANCE_PAGE_LAYOUT_LAYOUT_UPDATE_ID_PAGE_ID
# Key definitions:
#   KEY `IDX_WIDGET_INSTANCE_PAGE_LAYOUT_LAYOUT_UPDATE_ID` (`layout_update_id`),
#   UNIQUE KEY `UNQ_WIDGET_INSTANCE_PAGE_LAYOUT_LAYOUT_UPDATE_ID_PAGE_ID` (`layout_update_id`,`page_id`),
# Column types:
#         `layout_update_id` int(10) unsigned not null default '0' comment 'layout update id'
#         `page_id` int(10) unsigned not null default '0' comment 'page id'
# To remove this duplicate index, execute:
ALTER TABLE `[redacted]_live`.`widget_instance_page_layout` DROP INDEX `IDX_WIDGET_INSTANCE_PAGE_LAYOUT_LAYOUT_UPDATE_ID`;

# ########################################################################
# Summary of indexes
# ########################################################################

# Size Duplicate Indexes   361243847
# Total Duplicate Indexes  84
# Total Indexes            1719
```

At the end of the report, a summary of the indices is displayed, which is self-explanatory. Also, this tool prints out an `ALTER` query for each duplicate index that can be executed as a MySQL query to fix the index, as follows:

```
Pt-variable-advisor
```

This tool displays MySQL config information and advice for each query. This is a good tool that can help us set up MySQL configurations properly. We can execute this tool by running the following command:

```
Pt-variable-advisor –user packt –password
DbPassword localhost
```

After execution, the following output will be displayed:

```
# NOTE connect_timeout: A large value of this setting can create a denial of service vulnerability.

# WARN delay_key_write: MyISAM index blocks are never flushed until necessary.

# WARN innodb_additional_mem_pool_size: This variable generally doesn't need to be larger than 20MB.

# WARN innodb_fast_shutdown: InnoDB's shutdown behavior is not the default.

# WARN innodb_flush_log_at_trx_commit=1: InnoDB is not configured in strictly ACID mode.

# WARN innodb_log_buffer_size: The InnoDB log buffer size generally should not be set larger than 16MB.

# NOTE log_warnings=2: Log_warnings must be set greater than 1 to log unusual events such as aborted connections.

# NOTE max_binlog_size: The max_binlog_size is smaller than the default of 1GB.
```

There are many other tools provided by Percona Toolkit that are out of the scope of this book. However, the documentation at

https://www.percona.com/doc/percona-toolkit/2.2/index.html is very helpful and easy to understand. It provides complete details for each tool, including its description and risks, how to execute it, and other options if there are any. This documentation is worth reading if you wish to understand any tool in Percona Toolkit.

Percona XtraDB Cluster (PXC)

Percona XtraDB Cluster provides a high-performance cluster environment that can help easily configure and manage a database on multiple servers. It enables databases to communicate with each other using the binary logs. The cluster environment helps divide the load among different database servers and provides safety from failure in case a server is down.

To set up the cluster, we need the following servers:

- One server with IP 10.211.55.1, which we will call Node1
- A second server with IP 10.211.55.2, which we will call Node2
- And a third server with IP 10.211.55.3, which we will call Node3

As we already have the Percona repository in our sources, let's start by installing and configuring Percona XtraDB Cluster, also called PXC. Perform the following steps:

1. First, install Percona XtraDB Cluster on Node1 by issuing the following command in the terminal:

   ```
   apt-get install percona-xtradb-cluster-56
   ```

   The installation will start similarly to a normal Percona Server installation. During the installation, the password for a root user will be also asked.

2. When the installation is complete, we need to create a new user that has replication privileges. Issue the following commands in the MySQL terminal after logging in to it:

   ```
   CREATE USER 'sstpackt'@'localhost' IDENTIFIED
   BY 'sstuserpassword';
   GRANT RELOAD, LOCK TABLES, REPLICATION CLIENT
   ON *.* TO 'sstpackt'@'localhost';
   FLUSH PRIVILEGES;
   ```

The first query creates a user with the username `sstpackt` and password `sstuserpassword`. The username and password can be anything, but a good and strong password is recommended. The second query sets proper privileges to our new user, including locking tables and replication. The third query refreshes the privileges.

3. Now, open the MySQL configuration file located at `/etc/mysql/my.cnf`. Then, place the following configuration in the `mysqld` block:

```
#Add the galera library
wsrep_provider=/usr/lib/libgalera_smm.so

#Add cluster nodes addresses
wsrep_cluster_address=gcomm://10.211.55.1,10.2
11.55.2,10.211.55.3

#The binlog format should be ROW. It is
required for galera to work properly
binlog_format=ROW

#default storage engine for mysql will be
InnoDB
default_storage_engine=InnoDB

#The InnoDB auto increment lock mode should be
2, and it is required for galera
innodb_autoinc_lock_mode=2

#Node 1 address
wsrep_node_address=10.211.55.1

#SST method
wsrep_sst_method=xtrabackup

#Authentication for SST method. Use the same
user name and password created in above step 2
wsrep_sst_auth="sstpackt:sstuserpassword"

#Give the cluster a name
wsrep_cluster_name=packt_cluster
```

Save the file after adding the preceding configuration.

4. Now, start the first node by issuing the following command:

```
/etc/init.d/mysql bootstrap-pxc
```

   This will bootstrap the first node. Bootstrapping means getting the initial cluster up and running and defining which node has the correct information and which one all the other nodes should sync to. As Node1 is our initial cluster node and we created a new user here, we have to only bootstrap Node1.

> ## Note
>
> **SST** stands for **State Snapshot Transfer**. It is responsible for copying full data from one node to another. It is only used when a new node is added to the cluster and this node has to get complete initial data from an existing node. Three SST methods are available in `Percona XtraDB Cluster`, `mysqldump`, `rsync`, and `xtrabackup`.

5. Log in to the MySQL terminal on the first node and issue the following command:

```
SHOW STATUS LIKE '%wsrep%';
```

   A very long list will be displayed. A few of them are shown in the following screenshot:

```
mysql> show status like '%wsrep_cluster%';
+--------------------------+--------------------------------------+
| Variable_name            | Value                                |
+--------------------------+--------------------------------------+
| wsrep_cluster_conf_id    | 1                                    |
| wsrep_cluster_size       | 1                                    |
| wsrep_cluster_state_uuid | 65925905-f650-11e5-b9a4-b3804a801699 |
| wsrep_cluster_status     | Primary                              |
+--------------------------+--------------------------------------+
4 rows in set (0.00 sec)

mysql>
```

6. Now, repeat Step 1 and Step 3 for all nodes. The only configuration that needs to be changed for each node is `wsrep_node_address`, which should be the IP address of the node. Edit the `my.cnf` configuration file for all the nodes and place the node address in `wsrep_node_address`.
7. Start the two new nodes by issuing the following command in the terminal:

```
/etc/init.d/mysql start
```

   Now each node can be verified by repeating step 7.

To verify whether the cluster is working fine, create a database in one node and add some tables and data into the tables. After this, check other nodes for the newly created database, tables, and the data entered in each table. We will have all this data synced to each node.

Redis – the key-value cache store

Redis is an open source, in-memory key-value data store that is widely used for database caching. According to the Redis website (www.Redis.io), Redis supports data structures such as strings, hashes, lists, sets, and sorted lists. Also, Redis supports replication and transactions.

> **Note**
>
> Redis installation instructions can be found at http://redis.io/topics/quickstart.

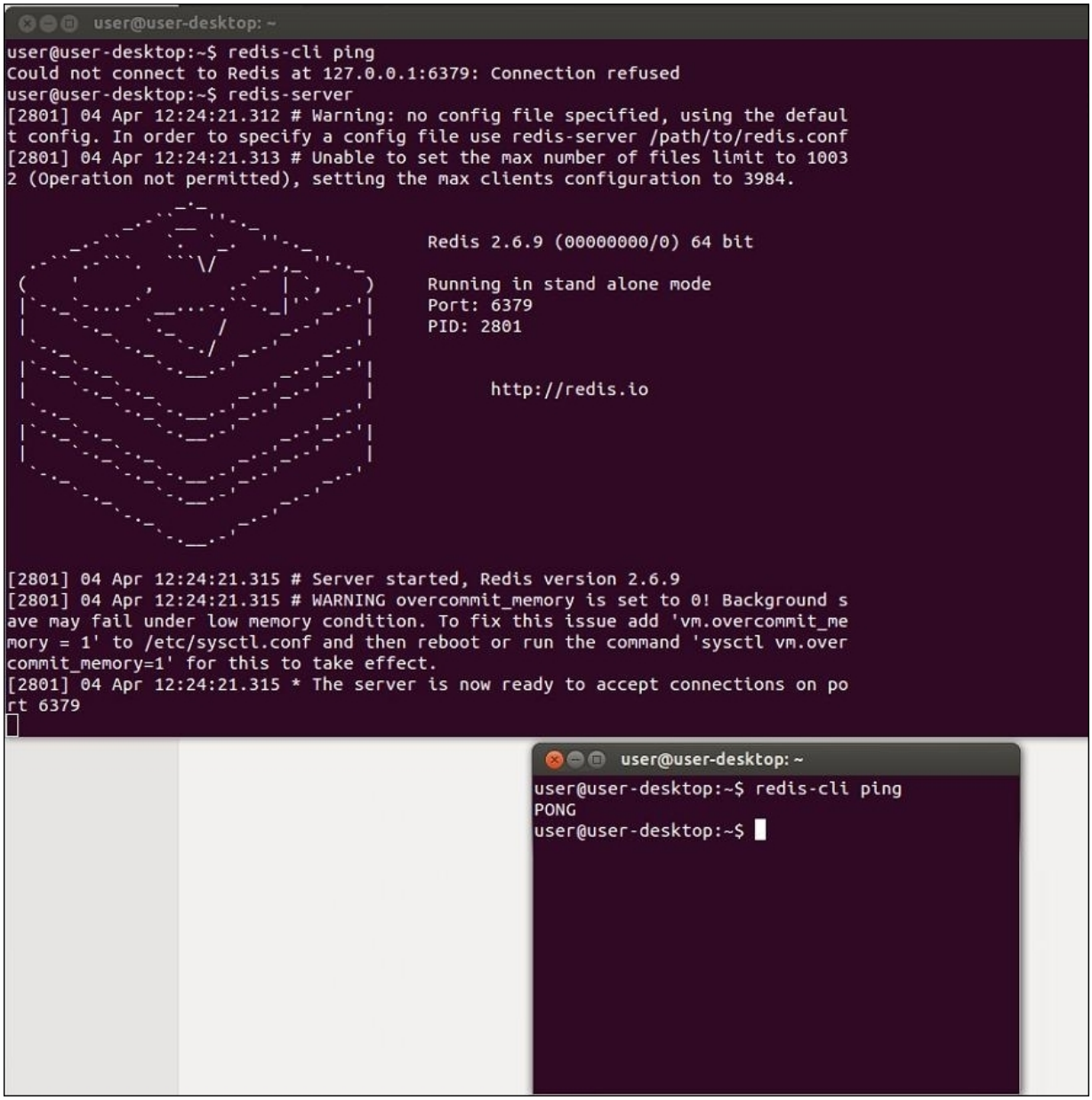
To check whether Redis is working fine on your server or not, start the Redis server instance by running the following command in the terminal:

```
redis server
```

Then issue the following command in a different terminal window:

```
redis-cli ping
```

If the output of the preceding command is as follows, the Redis server is ready to be run:



Redis provides a command line, which provides some useful commands. There are two ways to execute commands on the Redis server. You can either use the previous method or just type `redis-cli` and hit *Enter*; we will be presented with the Redis command line, where we can then just type the Redis commands that will be executed.

By default, Redis uses the IP 127.0.0.1 and port 6379. Remote connections are not allowed, though remote connections can be

enabled. Redis stores data that is already created in the database. Database names are integer numbers, such as 0, 1, 2, and so on.

We won't go in much detail about Redis here, but we will discuss a few commands that are worth noting. Note that all these commands can be either executed in the previous way, or we can just enter the `redis-cli` command window and type the commands without typing `redis-cli`. Also, the following commands can be executed directly in PHP, which makes it possible to clear out the cache directly from our PHP application:

- `SELECT`: This command changes the current database. By default, redis-cli will be opened at database 0. So, if we want to go to database 1, we will run the following command:

```
SELECT 1
```

- `FLUSHDB`: This command flushes the current database. All keys or data from the current database will be deleted.
- `FLUSHALL`: This command flushes all the databases, no matter which database it is executed in.
- `KEYS`: This command lists all the keys in the current database matching a pattern. The following command lists all the keys in the current database.

```
KEYS *
```

Now, it's time for some action in PHP with Redis.

Note

As of writing this topic, PHP 7 does not have built-in support for Redis yet. For this book's purpose, we compiled the PHPRedis module for PHP 7, and it works very nicely. The module can be found at https://github.com/phpredis/phpredis.

Connecting with the Redis server

As mentioned before, by default, the Redis server runs on the IP 127.0.0.1 and port 6379. So, to make a connection, we will use these details. Take a look at the following code:

```
$redisObject = new Redis();
if( !$redisObject->connect('127.0.0.1', 6379))
  die("Can't connect to Redis Server");
```

In the first line, we instantiated a Redis object by the name of `redisObject`, which is then used in the second line to connect to the Redis server. The host is the local IP address 127.0.0.1, and the port is 6379. The `connect()` method returns `TRUE` if the connection is successful; otherwise, it returns `FALSE`.

Storing and fetching data from the Redis server

Now, we are connected to our Redis server. Let's save some data in the Redis database. For our example, we want to store some string data in the Redis database. The code is as follows:

```
//Use same code as above for connection.
//Save Data in to Redis database.
$rdisObject->set('packt_title', 'Packt
Publishing');

//Lets get our data from database
echo $redisObject->get('packt_title');
```

The `set` method stores data into the current Redis database and takes two arguments: a key and a value. A key can be any unique name, and a value is what we need to store. So, our key is `packt_title`, and the value is `Packt Publishing`. The default database is always set to 0 (zero) unless explicitly set otherwise. So,

the preceding `set` method will save our data to database 0 with the `packt_title` key.

Now, the `get` method is used to fetch data from the current database. It takes the key as the argument. So, the output of the preceding code will be our saved string data `Packt Publishing`.

Now, what about arrays or a set of data coming from the database? We can store them in several ways in Redis. Let's first try the normal strings way, as shown here:

```
//Use same connection code as above.

/* This $array can come from anywhere, either it
is coming from database or user entered form data
or an array defined in code */

$array = ['PHP 5.4', PHP 5.5, 'PHP 5.6', PHP 7.0];

//Json encode the array
$encoded = json_encode($array);

//Select redis database 1
$redisObj->select(1);

//store it in redis database 1
$redisObject->set('my_array', $encoded);

//Now lets fetch it
$data = $redisObject->get('my_array');

//Decode it to array
$decoded = json_decode($data, true);

print_r($decoded);
```

The output of the preceding code will be the same array. For testing purposes, we can comment out the `set` method and check whether the `get` method fetches the data or not. Remember that in the preceding code, we stored the array as a `json` string, then fetched it as a `json` string, and decoded it to the array. This is because we

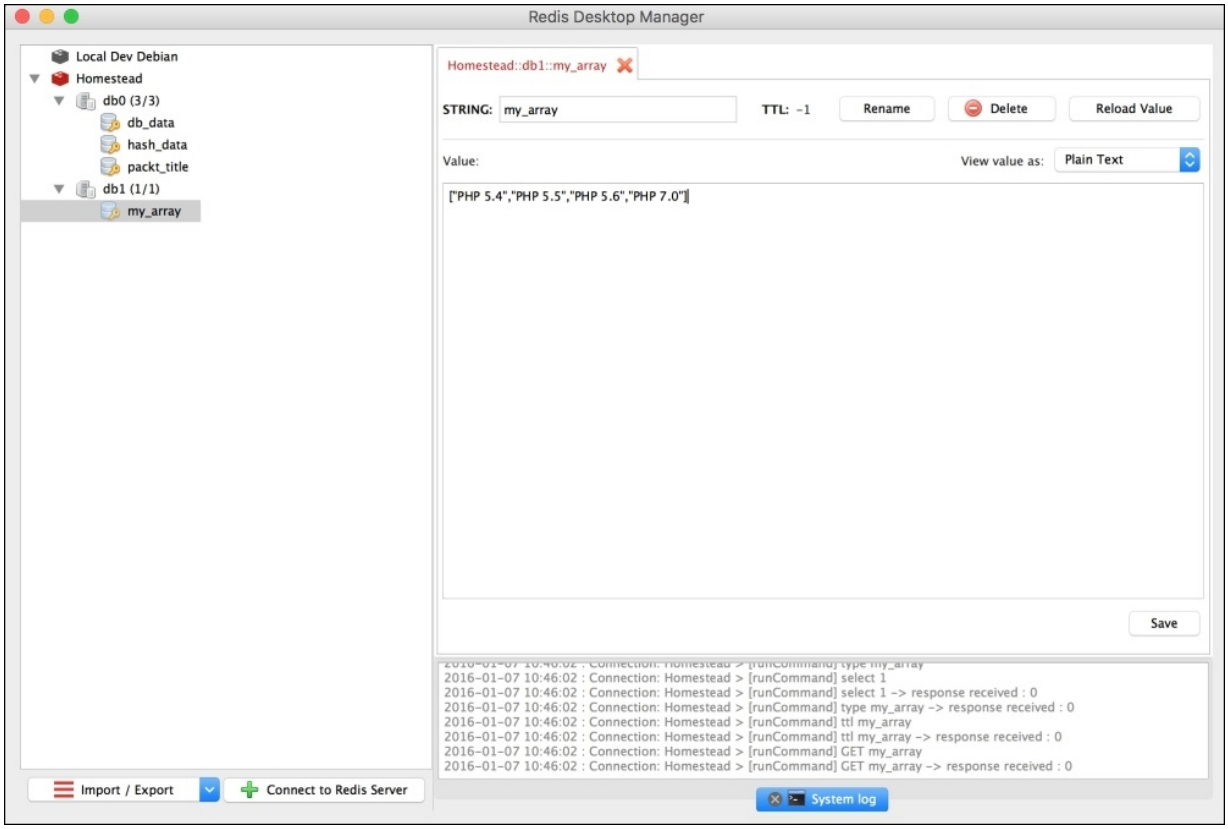
used the methods that are available for the string datatype, and it is not possible to store arrays in the string datatype.

Also, we used the `select` method to select another database and use it instead of 0. This data will be stored in database 1 and can't be fetched if we are at database 0.

> ## Note
>
> A complete discussion of Redis is out of the scope of this book. So, we have provided an introduction. Note that if you use any framework, you have built-in libraries available for Redis that are easy to use, and any datatype can be used easily.

Redis management tools

Redis management tools provide an easy way to manage Redis databases. These tools provide features so that every key can be checked and a cache can be cleared easily. One default tool comes with Redis, called Redis-cli, and we discussed it earlier. Now, let's discuss a visual tool that is great and easy to use, called **Redis Desktop Manage** (**RDM**). A screenshot of the main window of RDM looks like the following screenshot:



RDM provides the following features:

- It connects to remote multiple Redis servers
- It displays data in a specific key in different formats
- It adds new keys to a selected database
- It adds more data to a selected key
- It edits/deletes keys and their names
- It supports SSH and SSL and is cloud ready

There are some other tools that can be used, but RDM and Redis-cli are the best and easiest to use.

Memcached key-value cache store

According to the Memcached official website, it's a free, open source, high performance, and distributed memory object caching system. Memcached is an in-memory key-value store that can store datasets from a database or API calls.

Similarly to Redis, Memcached also helps a lot in speeding up a website. It stores the data (strings or objects) in the memory. This allows us to reduce the communication with outside resources, such as databases and or APIs.

> **Note**
>
> We are assuming that Memcached is installed on the server. Also, the PHP extension for PHP 7 is also assumed to be installed.

Now, let's play a little with Memcachd in PHP. Take a look at the following code:

```
//Instantiate Memcached Object
$memCached = new Memcached();

//Add server
$memCached->addServer('127.0.0.1', 11211);

//Lets get some data
$data = $memCached->get('packt_title');

//Check if data is available
if($data)
{
  echo $data;
}
```

```
else
{
  /*No data is found. Fetch your data from any
where and add to   memcached */

  $memCached->set('packt_title', 'Packt
Publishing');

}
```

The preceding code is a very simple example of using Memcached. The comments are written with each line of code and are self-explanatory. After instantiating a Memcached object, we have to add a Memcached server. By default, the Memcached server server runs on the localhost IP, which is 127.0.0.1, and on the port 11211. After this, we checked for some data using a key, and if it is available, we can process it (in this case, we displayed it. It can be returned, or whatever processing is required can be carried out.). If the data is not available, we can just add it. Please note that the data can come from a remote server API or from the database.

Note

We have just provided an introduction to Memcached and how it can help us store data and improve performance. A complete discussion is not possible in this title. A good book on Memcached is *Getting Started with Memcached* by Packt Publishing.

Summary

In this chapter, we covered MySQL and the Percona Server. Also, we discussed in detail query caching and other MySQL configuration options for performance in detail. We mentioned different storage engines, such as MyISAM, InnoDB, and Percona XtraDB. We also configured Percona XtraDB Cluster on three nodes. We discussed different monitoring tools, such as PhpMyAdmin monitoring tools, MySQL workbench performance monitoring, and Percona Toolkit. We also discussed Redis and Memcached caching for PHP and MySQL.

In the next chapter, we will discuss benchmarking and different tools. We will use XDebug, Apache JMeter, ApacheBench, and Siege to benchmark different open source systems, such as WordPress, Magento, Drupal, and different versions of PHP, and compare their performance with PHP 7.

Chapter 5. Debugging and Profiling

During development, every developer faces problems, and it becomes unclear what is really going on here and why the problem is generated. Most the time, these issues can be logical or with the data. It is always hard to find such issues. Debugging is a process to find such issues and problems and fix them. Similarly, we often need to know how many resources a script consumes, including memory consumption, CPU, and how much time it takes to execute.

In this chapter, we will cover the following topics:

- Xdebug
- Debugging with Sublime Text 3
- Debugging with Eclipse
- Profiling with Xdebug
- PHP DebugBar

Xdebug

Xdebug is an extension for PHP that provides both debugging and profiling information for PHP scripts. Xdebug displays a full-stake trace information for errors, including function names, line numbers, and filenames. Also, it provides the ability to debug scripts interactively using different IDEs, such as Sublime Text, Eclipse, PHP Storm, and Zend Studio.

To check whether Xdebug is installed and enabled on our PHP installation, we need to check the phpinfo() details. On the phpinfo details page, search for Xdebug, and you should see details similar to the following screenshot:



This means that our PHP installation has Xdebug installed. Now, we need to configure Xdebug. Either the Xdebug configuration will be in the `php.ini` file, or it will have its separate `.ini` file. At our installation, we will have a separate `20-xdebug.ini` file placed at the `/etc/php/7.0/fpm/conf.d/` path.

Note

For the purpose of this book, we will use the Homestead Vagrant box from Laravel. It provides complete tools on the Ubuntu 14.04 LTS installation, including PHP7 with Xdebug, NGINX, and MySQL. For the purpose of development, this Vagrant box is a perfect solution. More information can be found at https://laravel.com/docs/5.1/homestead.

Now, open the `20-xdebug.ini` file and place the following configuration in it:

```
zend_extension = xdebug.so
xdebug.remote_enable = on
xdebug.remote_connect_back = on
xdebug.idekey = "vagrant"
```

The preceding are the minimum configurations we should use that enable remote debugging and set an IDE key. Now, restart PHP by issuing the following command in the terminal:

```
sudo service php-fpm7.0 restart
```

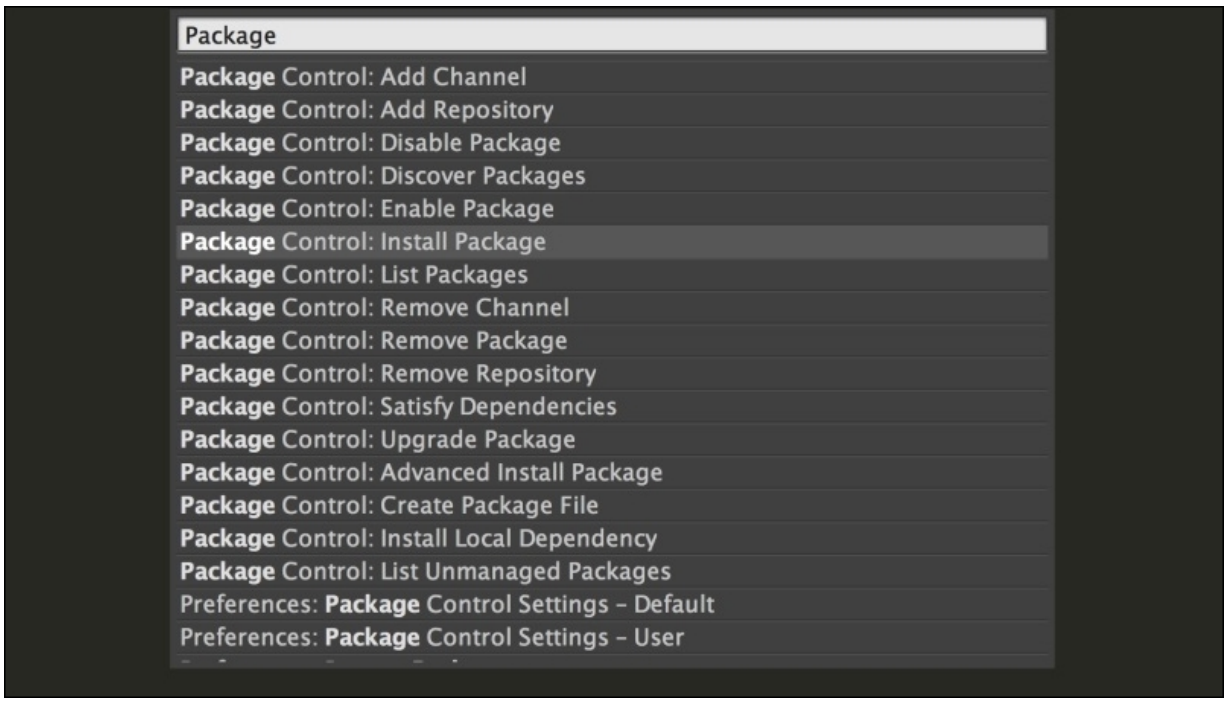
Now we are ready to debug some code.

Debugging with Sublime Text

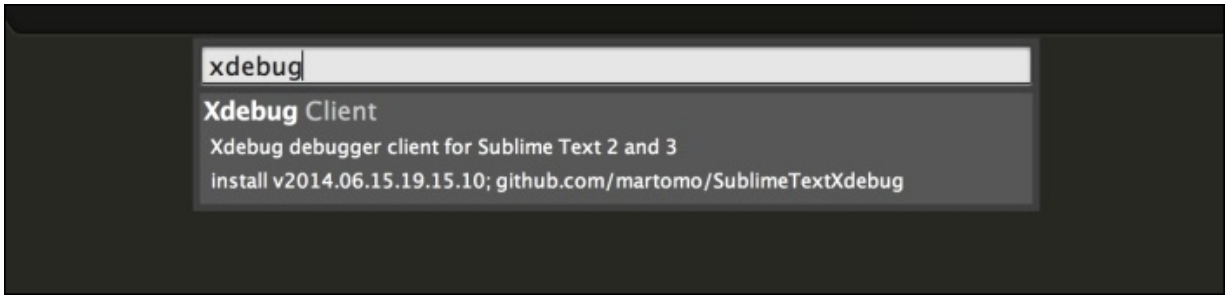
The Sublime Text editor has a plugin that can be used to debug PHP code with Xdebug. First, let's install the `xdebug` package for Sublime Text.

> ## Note
>
> For this topic, we will use Sublime Text 3, which is still in beta. It is your own choice to use version 2 or 3.

First, go to **Tools** | **Command Pallet**. A popup similar to the following will be displayed:



Select **Package Control: Install Package**, and a popup similar to the following screenshot will be displayed:



Type in `xdebug`, and the **Xdebug Client** package will be displayed. Click on it and wait for a while until it is installed.

Now, create a project in Sublime Text and save it. Open the Sublime Text project file and insert the following code in it:

```
{
  "folders":
  [
    {
    "follow_symlinks": true,
    "path": "."
  }
],

"settings": {
  "xdebug": {
    "path_mapping": {
    "full_path_on_remote_host" :
"full_path_on_local_host"
    },
    "url" : http://url-of-application.com/,
    "super_globals" : true,
    "close_on_stop" : true,
    }
  }
}
```

The highlighted code is important, and it has to be entered for Xdebug. Path mapping is the most important part. It should have a full path to the root of the application on the remote host and a full path to the root of the application on the localhost.

Now, let's start debugging. Create a file at the project's root, name it `index.php`, and place the following code in it:

```
$a = [1,2,3,4,5];
$b = [4,5,6,7,8];

$c = array_merge($a, $b);
```

Now, right-click on a line in the editor and select **Xdebug**. Then, click on **Add/Remove Breakpoint**. Let's add a few breakpoints as shown in the following screenshot:

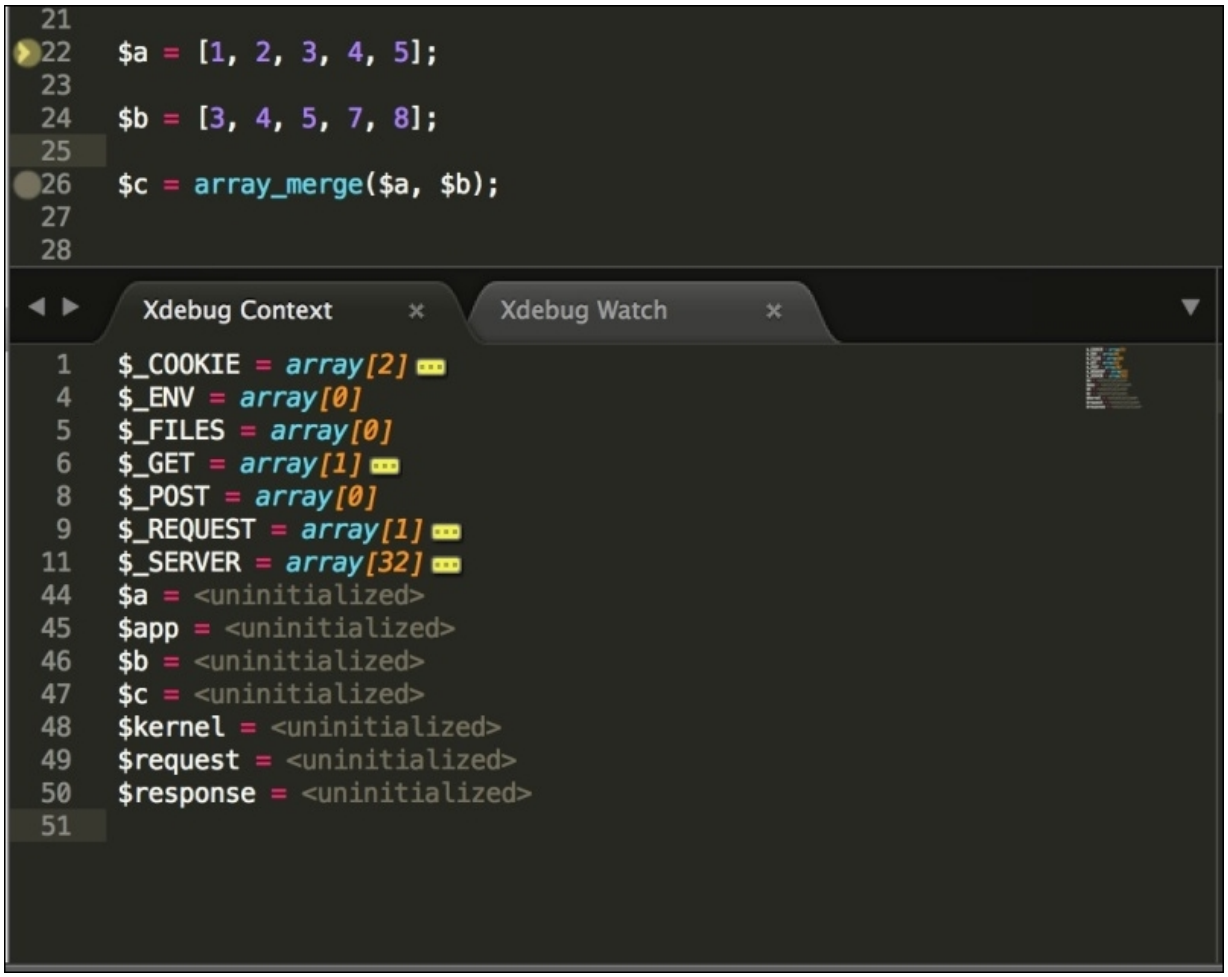


When a breakpoint is added to a line, a filled circle will be displayed on the left-hand side near the line number, as can be seen in the preceding screenshot.

Now we are ready to debug our PHP code. Navigate to **Tools** | **Xdebug** | **Start Debugging (Launch in Browser)**. A browser window will open the application along with a Sublime Text debug session parameter. The browser windows will be in the loading state because as soon as the first breakpoint is reached, the execution stops. The browser window will be similar to the following:



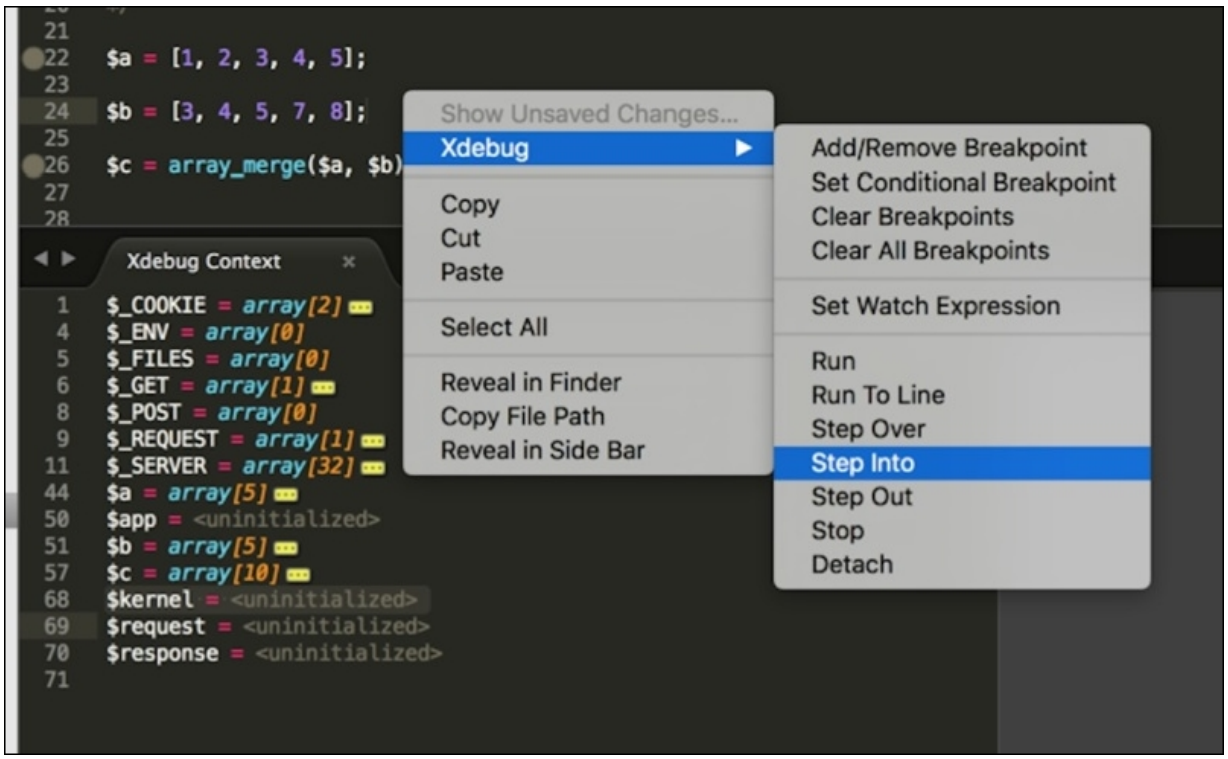
Some new small windows will also open in the Sublime Text editor that will display debugging information along with all the variables available, as in the following screenshot:



In the preceding screenshot, our `$a`, `$b`, and `$c` arrays are uninitialized because the execution cursor is at Line 22, and it has stopped there. Also, all server variables, cookies, environment variables, request data, and POST and GET data can be seen here. This way, we can debug all kind of variables, arrays, and objects and check what data each variable, object, or array holds at a certain point. This gives us the possibility to find out the errors that are very hard to detect without debugging.

Now, let's move the execution cursor ahead. Right-click in the editor code section and go to **Xdebug** | **Step Into**. The cursor will move

ahead, and the variables data may change according to the next line. This can be noted in the following screenshot:

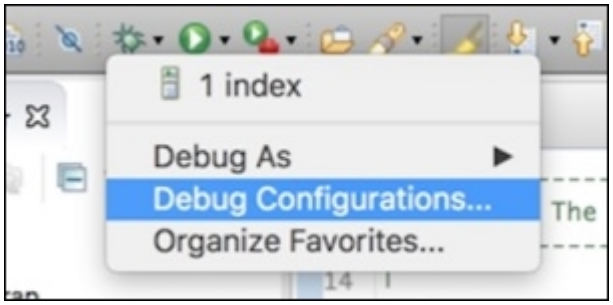


Debugging can be stopped by clicking on **Tools** | **Xdebug** | **Stop Debugging**.

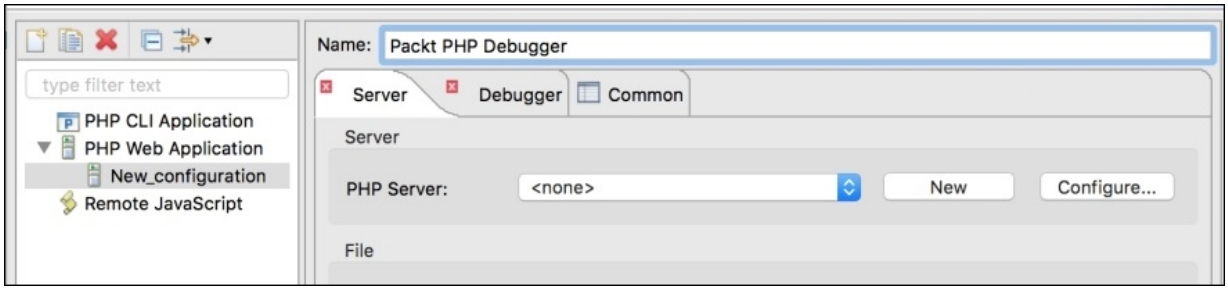
Debugging with Eclipse

Eclipse is the most free and powerful IDE widely used. It supports almost all major programming languages, including PHP. We will discuss how to configure Eclipse to use Xdebug to debug.

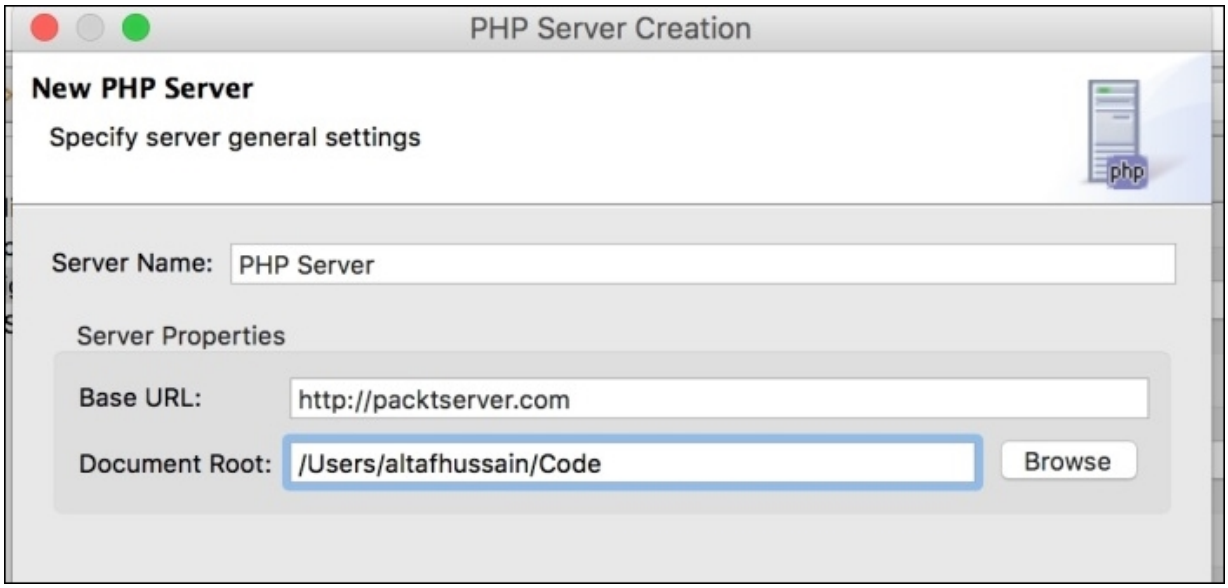
First, open the project in Eclipse. Then, click on the down arrow to the right of the small bug icon in the tool bar, as shown in the following screenshot:



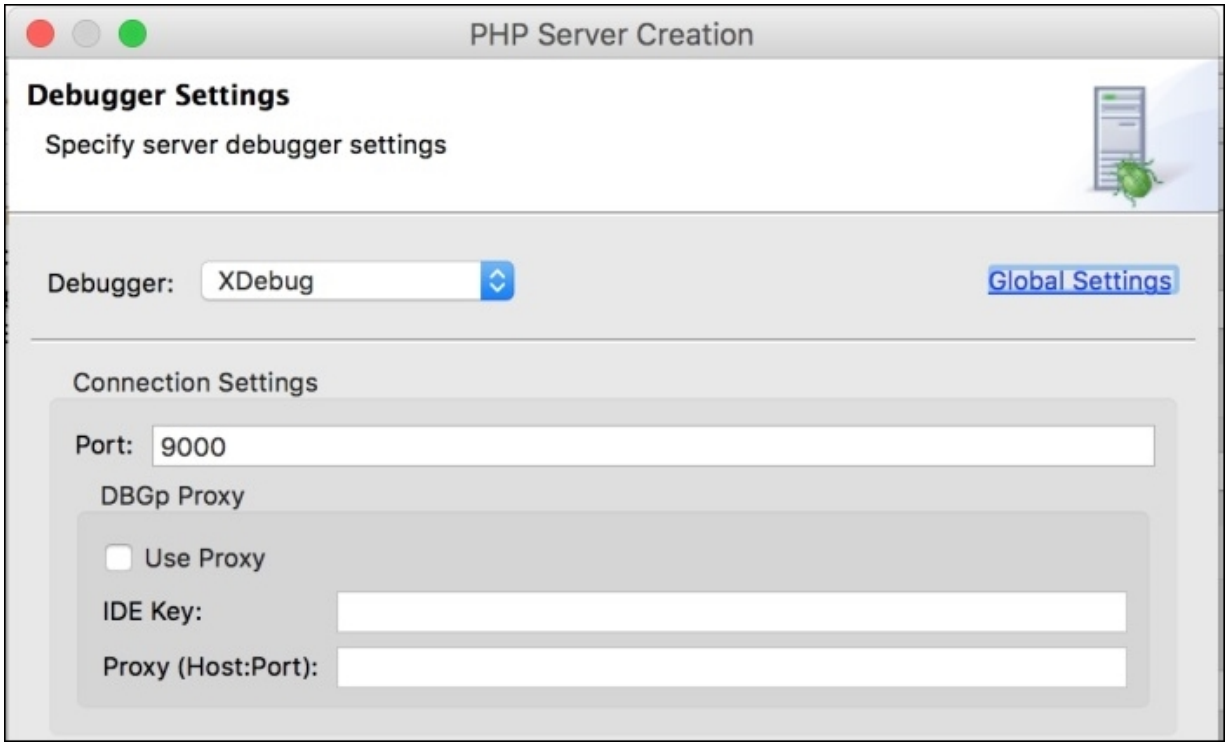
After this, click on the **Debug Configuration** menu, and the following windows will open:



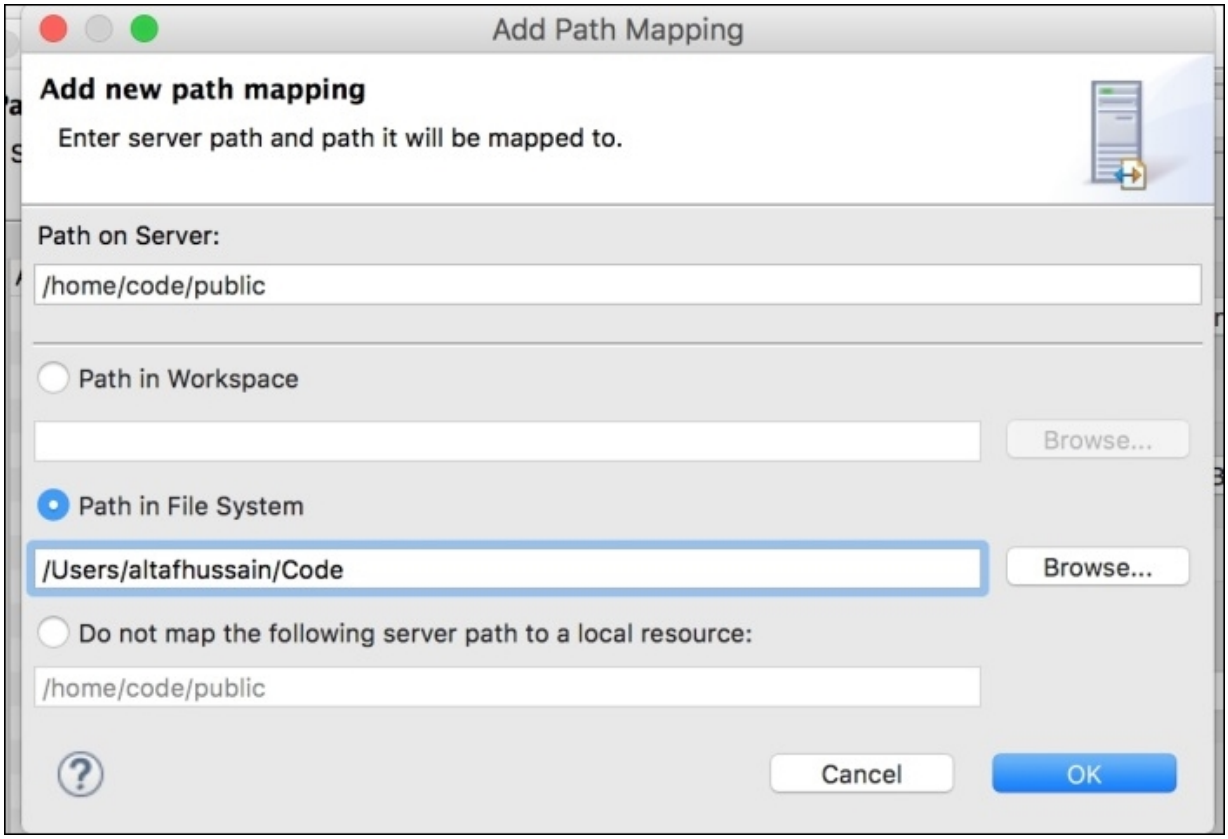
Select **PHP Web Application** on left panel and then click on the **Add New** icon in the top-left corner. This will add a new configuration, as shown in the preceding screenshot. Give the configuration a name. Now, we need to add a PHP server to our configuration. Click on the **New** button on the right-hand side panel, and the following window will open:



We will enter the server name as `PHP Server`. The server name can be anything as long as it is user-friendly and can be recognized for later use. In the **Base URL** field, enter the complete URL of the application. **Document Root** should be the local path of the root of the application. After entering all the valid data, click on the **Next** button, and we will see the following window:

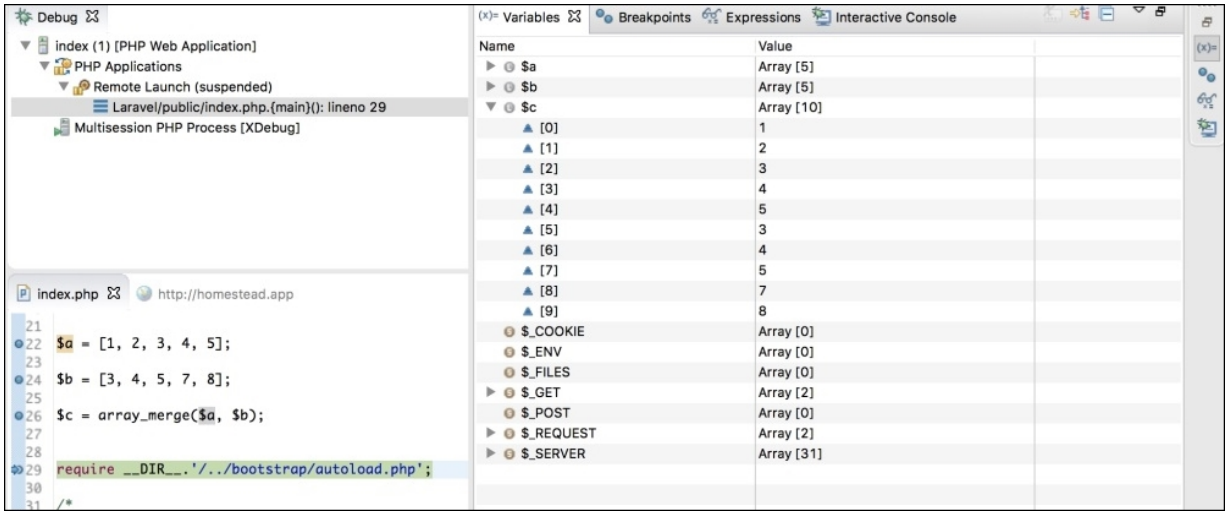


Select **XDebug** in the **Debugger** drop-down list and leave rest of the fields as they are. Click on the **Next** button, and we will have the path mapping window. It is very important to map the correct local path to the correct remote path. Click on the **Add** button, and we will have the following window:



Enter the full path to the document root of the application on the remote server. Then, select **Path in File System** and enter the local path of the application's document root. Click on **OK** and then click on the **Finish** button in the path mapping window. Then, click on **Finish** in the next window to complete adding a PHP server.

Now, our configuration is ready. First, we will add some breakpoints to our PHP file by clicking on the line number bar and a small blue dot will appear there, as shown in the following screenshot. Now, click on the small bug icon on the tool bar, select **Debug As**, and then click on **PHP Web Application**. The debug process will start, and a window will be opened in the browser. It will be in the loading state, same as we saw in Sublime Text debugging. Also, the Debug perspective will be opened in Eclipse, as shown here:



When we click on the small (**X**)= icon in the right-hand side bar, we will see all the variables there. Also, it is possible to edit any variable data, even the element values of any array, object properties, and cookie data. The modified data will be retained for the current debug session.

To step into the next line, we will just press *F5*, and the execution cursor will be moved to the next line. To step out to the next breakpoint, we will press *F6*.

Profiling with Xdebug

Profiling gives us information about the cost of each script or task executed in an application. It helps to provide information about how much time a task takes, and hence we can optimize our code to consume less time.

Xdebug has a profiler that is disabled by default. To enable the profiler, open the configuration file and place the following two lines in it:

```
xdebug.profiler_enable=on
xdebug.profiler_output_dir=/var/xdebug/profiler/
```

The first line enables the profiler. The second line, where we defined the output directory for the profiler file, is important. In this directory, Xdebug will store the output file when the profiler is executed. The output file is stored with a name, such as `cachegrind.out.id`. This file contains all the profile data in a simple text format.

Now, we are set to profile a simple installation of the Laravel application home page. The installation is a fresh and clean one. Now, let's open the application in a browser and append `?XDEBUG_PROFILE=on` at the end, as shown here:

```
http://application_url.com?XDEBUG_PROFILE=on
```

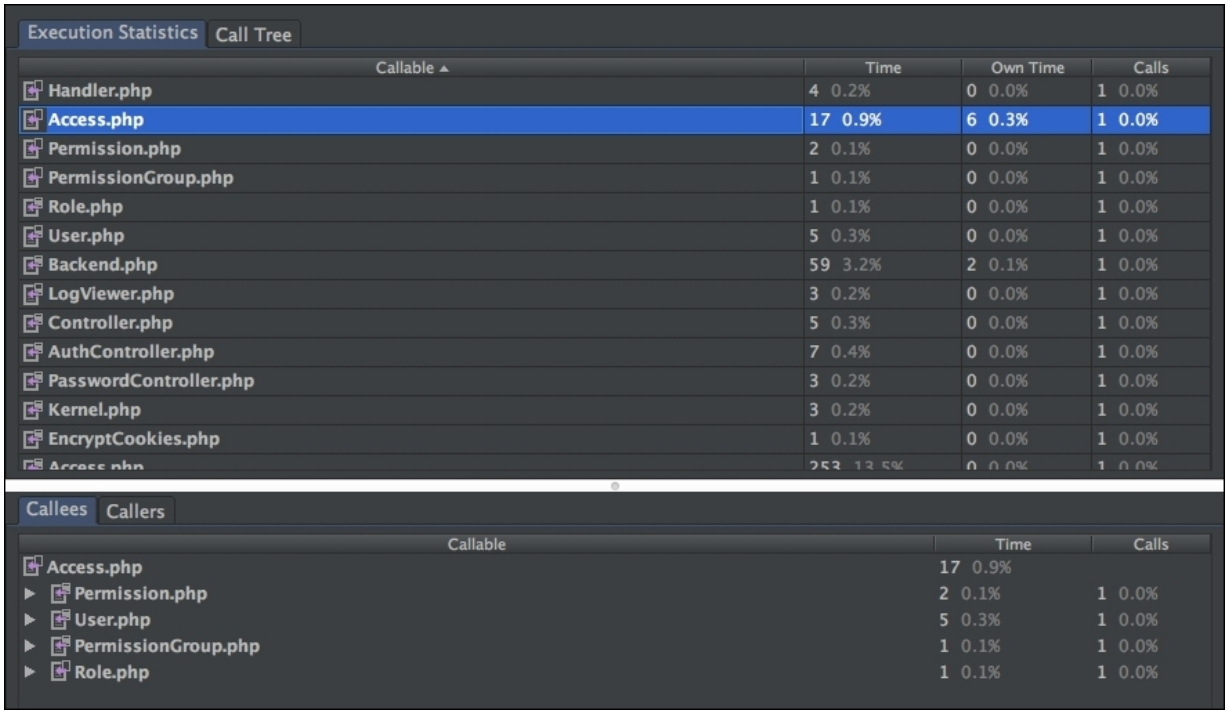
After this page is loaded, a `cachegrind` file will be generated at the specified location. Now, when we open the file in a text editor, we will just see some text data.

Note

The `cachegrind` file can be opened with different tools. One of the tools for Windows is WinCacheGrind. For Mac, we have qcachegrind. Any of these applications will view the file data in

such a way that we will see all the data in an interactive form that can be easily analyzed. Also, PHP Storm has a nice analyzer for cachegrind. For this topic, we used PHP Storm IDE.

After opening the file in PHP Storm, we will get a window similar to the following screenshot:



As shown in the preceding screenshot, we have execution statistics in the upper pane that shows the time (in ms) taken by each called script individually along with the number of times it is called. In the lower pane, we have the callees that called this script.

We can analyze which script takes more time, and we can optimize this script to reduce its execution time. Also, we can find out whether, at a certain point, we need to call a specific script or not. If not, then we can remove this call.

PHP DebugBar

PHP DebugBar is another awesome tool that displays a nice and full information bar at the bottom of the page. It can display custom messages added for the purposes of debugging and full request information including `$_COOKIE`, `$_SERVER`, `$_POST`, and `$_GET` arrays along with the data if any of them have. Besides that, PHP DebugBar displays details about exceptions if there are any, database queries executed, and their details. Also it displays the memory taken by the script and the time the page is loaded in.

According to the PHP Debug website, DebugBar integrates easily in any application project and displays debugging and profiling data from any part of the application.

Its installation is easy. You can either download the complete source code, place it somewhere in your application, and set up the autoloader to load all the classes, or use composer to install it. We will use composer as it is the easy and clean way to install it.

> ## Note
>
> Composer is a nice tool for PHP to manage the dependencies of a project. It is written in PHP and is freely available from https://getcomposer.org/. We assume that composer is installed on your machine.

In your project's `composer.json` file, place the following code in the required section:

```
"maximebf/debugbar" : ">=1.10.0"
```

Save the file and then issue the following command:

```
composer update
```

The Composer will start updating the dependencies and install composer. Also, it will generate the autoloader file and/or the other dependencies required for DebugBar.

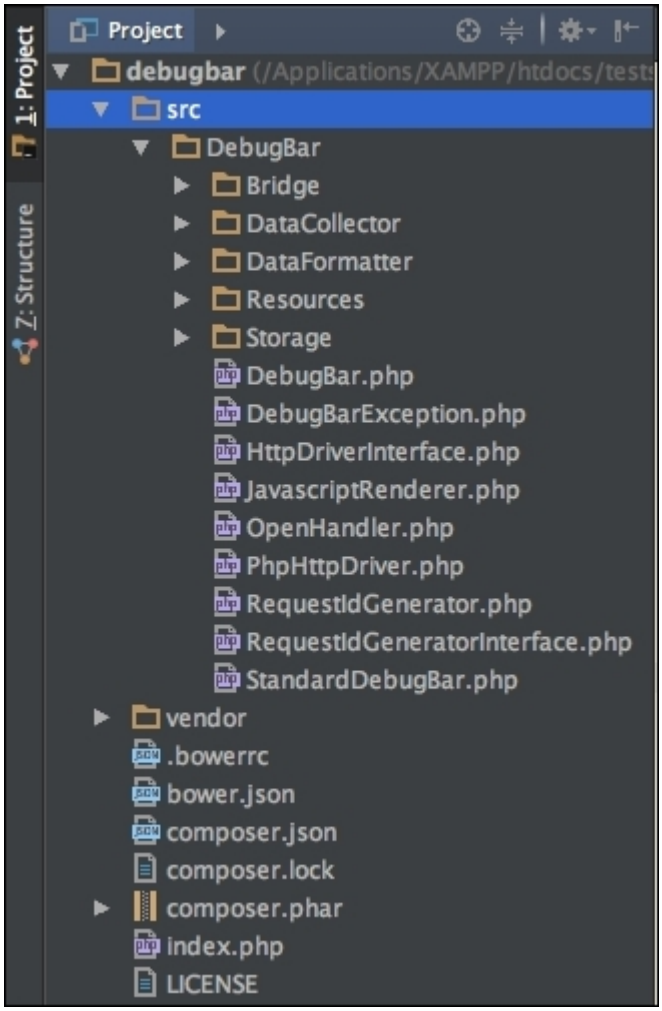
Note

The preceding composer command will only work if composer is installed globally on the system. If it is not, we have to use the following command:

```
php composer.phar update
```

The preceding command should be executed in the folder where `composer.phar` is placed.

After it is installed, the project tree for the DebugBar can be as follows:



The directories' structure may be a little bit different, but normally, it will be as we previously noted. The `src` directory has the complete source code for DebugBar. The `vendor` directory has some third-party modules or PHP tools that may or may not be required. Also, note that the `vendor` folder has the autoloader to autoload all the classes.

Let's check our installation now to see whether it is working or not. Create a new file in your project root and name it `index.php`. After this, place the following code in it:

```
<?php
require "vendor/autoloader.php";
use Debugbar\StandardDebugBar;
$debugger = new StandardDebugBar();
```

```
$debugbarRenderer = $debugbar-
>getJavascriptRenderer();

//Add some messages
$debugbar['messages']->addMessage('PHP 7 by
Packt');
$debugbar['messages']->addMessage('Written by
Altaf Hussain');

?>

<html>
  <head>
    <?php echo $debugbarRenderer->renderHead(); ?>
  </head>
  <title>Welcome to Debug Bar</title>
  <body>
    <h1>Welcome to Debug Bar</h1>

  <!-- display debug bar here -->
  <?php echo $debugbarRenderer->render();  ?>

  </body>
</html>
```

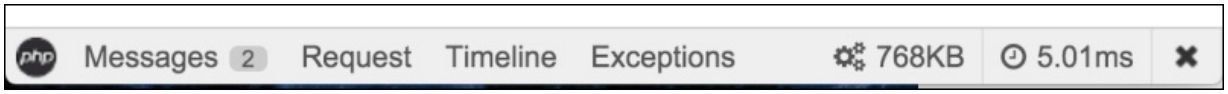
In the preceding code, we first included our autoloader, which is generated by composer for us to autoload all the classes. Then, we used the `DebugBar\StandardDebugbar` namespace. After this, we instantiated two objects: `StandardDebugBar` and `getJavascriptRenderer`. The `StandardDebugBar` object is an array of objects that has objects for different collectors, such as message collectors and others. The `getJavascriptRenderer` object is responsible for placing the required JavaScript and CSS code at the header and displaying the bar at the bottom of the page.

We used the `$debugbar` object to add messages to the message collector. Collectors are responsible for collecting data from different sources, such as databases, HTTP requests, messages, and others.

In the head section of the HTML code, we used the `renderHead` method of `$debugbarRenderer` to place the required JavaScript and

CSS code. After this, just before the end of the `<body>` block, we used the `render` method of the same object to display the debug bar.

Now, load the application in the browser, and if you notice a bar at the bottom of the browser as in the following screenshot, then congrats! DebugBar is properly installed and is working fine.



On the right-hand side, we have the memory consumed by our application and the time it is loaded in.

If we click on the **Messages** tab, we will see the messages we added, as shown in the following screenshot:



DebugBar provides data collectors, which are used to collect data from different sources. These are called *base collectors*, and some of the data collectors are as follows:

- The message collector collects log messages, as shown in the preceding example
- The TimeData collector collects the total execution time as well as the execution time for a specific operation
- The exceptions collector displays all the exceptions that have occurred
- The PDO collector logs SQL queries
- The RequestData collector collects data of PHP global variables, such as `$_SERVER`, `$_POST`, `$_GET`, and others
- The config collector is used to display any key-value pairs of arrays

Also, there are some collectors that provide the ability to collect data from third-party frameworks such as Twig, Swift Mailer, Doctrine, and others. These collectors are called bridge collectors. PHP DebugBar can be easily integrated into famous PHP frameworks such as Laravel and Zend Framework 2 too.

Note

A complete discussion of PHP DebugBar is not possible in this book. Therefore, only a simple introduction is provided here. PHP DebugBar has a nice documentation that provides complete details with examples. The documentation can be found at http://phpdebugbar.com/docs/readme.html.

Summary

In this chapter, we discussed different tools to debug a PHP application. We used Xdebug, Sublime Text 3, and Eclipse to debug our applications. Then, we used the Xdebug profiler to profile an application to find out the execution statistics. Finally, we discussed PHP DebugBar to debug an application.

In the next chapter, we will discuss load testing tools, which we can use to place load or virtual visitors on our application in order to load test it, and find out how much load our application can bear, and how it affects the performance.

Chapter 6. Stress/Load Testing PHP Applications

After an application is developed, tested, debugged and then profiled, it is time to bring it to production. However, before going to production, it is best practice to stress/load test the application. This test will give us an approximate result of how many requests at a certain time can be handled by our server running the application. Using these results, we can optimize the application, web server, database, and our caching tools to get a better result and process more requests.

In this chapter, we will load test different open source tools on both PHP 5.6 and PHP 7 and compare these applications' performance for both versions of PHP.

We will cover the following topics:

- Apache JMeter
- ApacheBench (ab)
- Seige
- Load testing Magento 2 on PHP 5.6 and PHP 7
- Load testing WordPress on PHP 5.6 and PHP 7
- Load testing Drupal 8 on PHP 5.6 and PHP 7

Apache JMeter

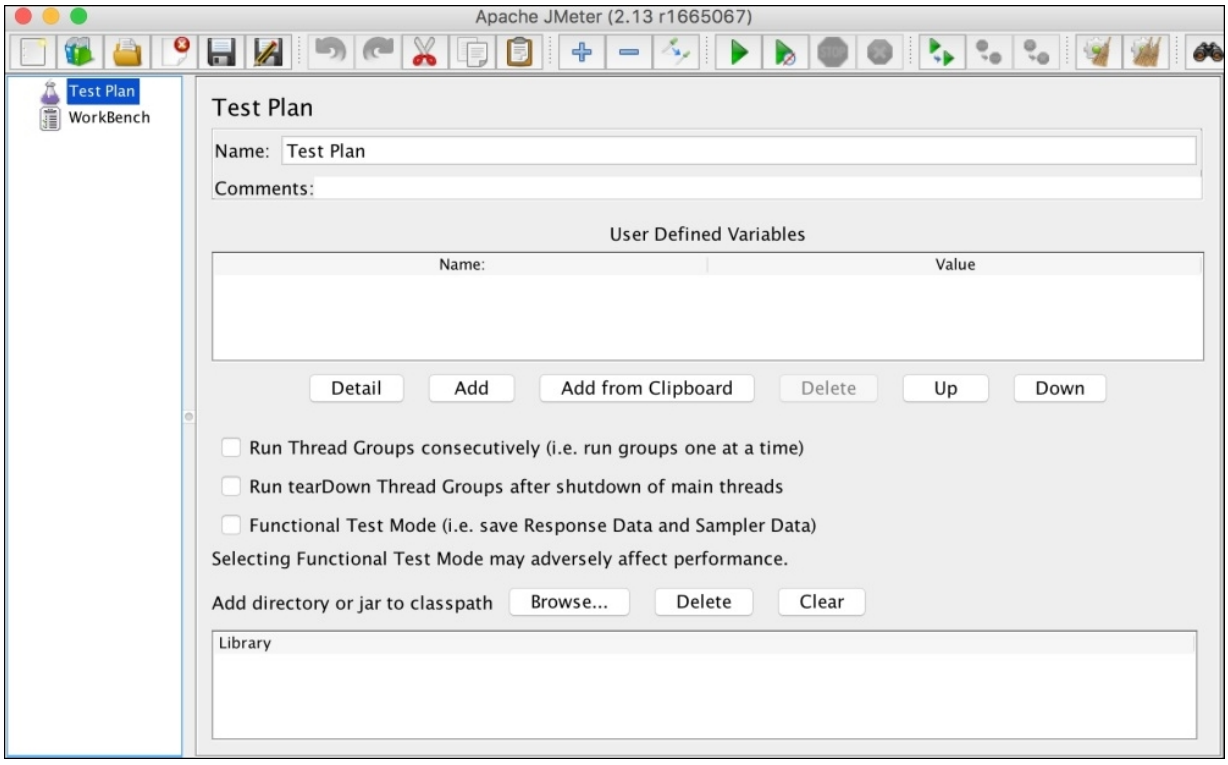
Apache JMeter is a graphical and open source tool used to load test a server's performance. JMeter is completely written in Java, so it is compatible with all operating systems that have Java installed. JMeter has a complete set of extensive tools for every kind of load testing, from static content to dynamic resources and web services.

Its installation is simple. We need to download it from the JMeter website and then just run the application. As mentioned before, it will require Java to be installed on the machine.

Note

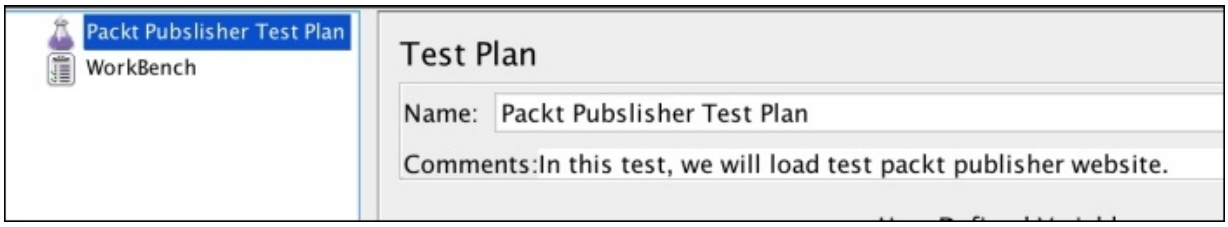
JMeter can test FTP servers, mail servers, database servers, queries, and more. In this book, we can't cover all these topics, so we will only load test web servers. Apache JMeter's list of features can be found at http://jmeter.apache.org/.

When we run the application at first, we will see the following window:



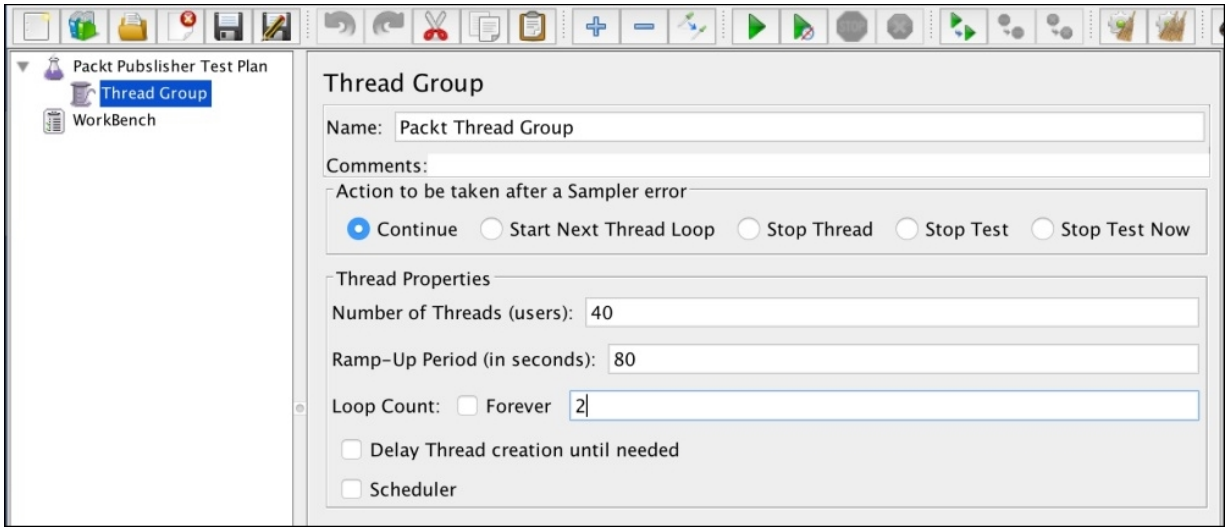
To run any kind of test, you need to first create a test plan. A test plan has all the components required to execute this test. By default, JMeter has a test plan called Test Plan. Let's name it to our own

plan, `Packt Publisher Test Plan`, as shown in the following screenshot:



Now, save the test plan, and JMeter we will create a `.jmx` file. Save it in an appropriate place.

The next step is to add a thread group. *A thread group defines some basic properties for the test plan, which can be common among all types of tests*. To add a thread group, right-click on the plan in the left panel, then navigate to **Add** | **Threads (Users)** | **Thread Group**. The following window will be displayed:

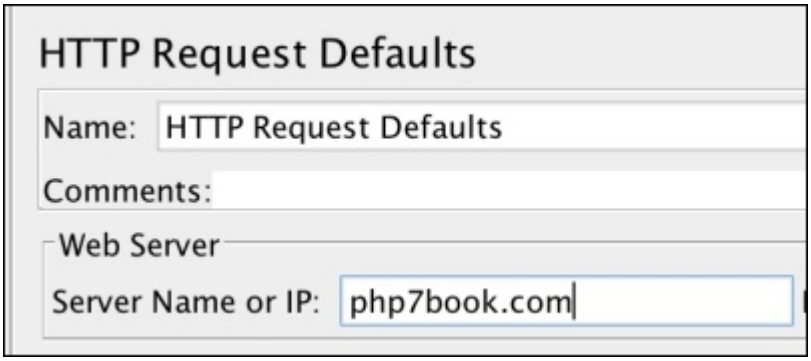


The thread group has the following important properties:

- **Number of Threads**: This is the number of virtual users.
- **The Ramp-Up period**: This tells JMeter how long it should take to ramp up to the full capacity of the number of threads. For example, in the preceding screenshot, we have 40 threads and 80 seconds of ramp-up time; here, J Meter will take 80 seconds

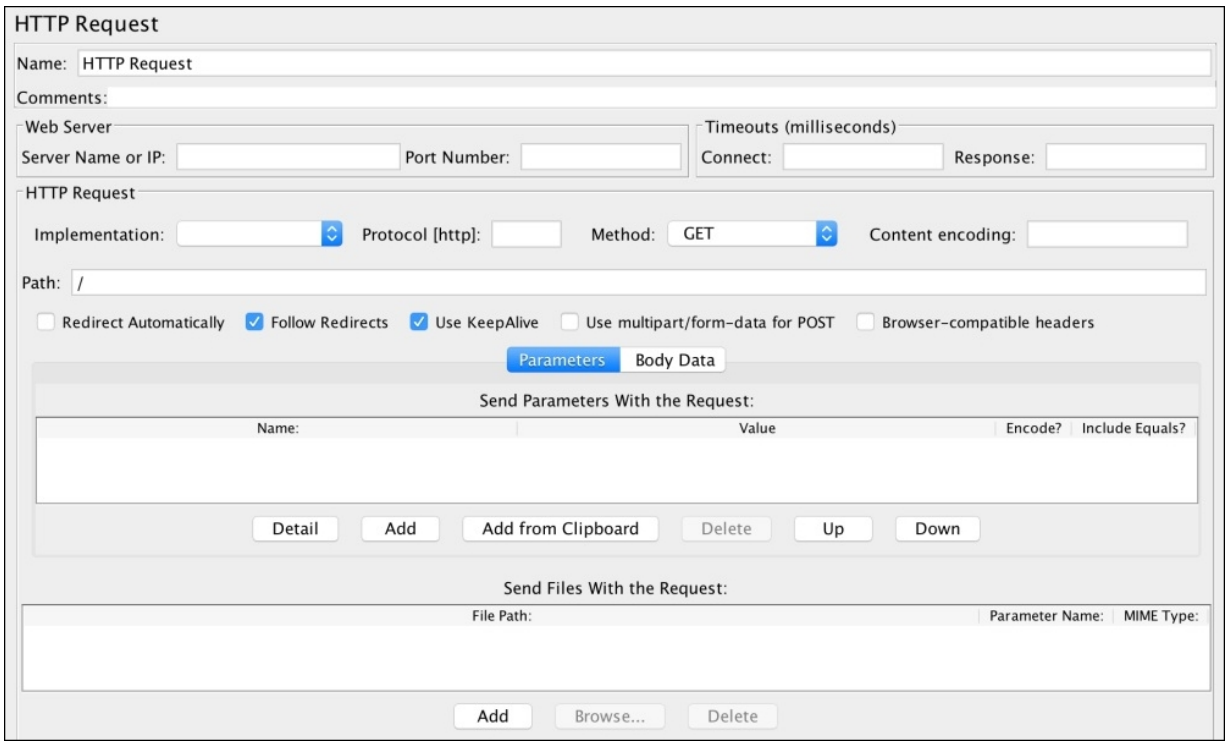
to completely fire up 40 threads, and it will take 2 seconds for each of the thread to start.
- **Loop Count**: This tells JMeter how much time it should take to run this thread group.
- **Scheduler**: This is used to schedule the execution of the thread group for a later time.

Now, we will need to add the HTTP request defaults. Right-click on **Packt Thread Group** and then go to **Add** | **Config Element** | **HTTP Request Defaults**. A window similar to the following will appear:



In the preceding window, we have to just enter the URL of the application or the IP address. If the web server uses cookies, we can add HTTP Cookie Manager too, in which we can add user-defined cookies with all the data, such as the name, value, domain, path, and so on.

Next, we will add an HTTP request by right-clicking and navigating to **Packt Thread Group** | **Add** | **Sampler** | **HTTP Request**, and the following window will appear:

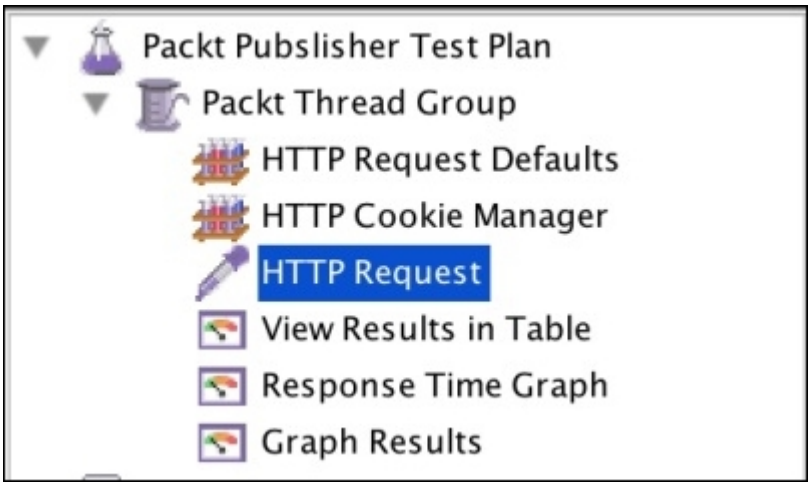


The important field here is **Path**. We want to run the test only against the home page, so for this HTTP request, we will just add a slash (`/`) in the **Path** field. If we want to test another path, such as "Contact us", we will need to add another HTTP request sampler, as in the preceding screenshot. Then, in the path, we will add `path/contact-us`.

The HTTP Request sampler can be used to test forms too, where POST requests can be sent to the URL by selecting the POST method in the **Method** field. Also, file upload can be simulated.

The next step is to add some listeners. *Listeners provide some powerful views to display results*. The results can be displayed in a table view and different kinds of graphs can be saved in a file. For this thread group, we will add three listeners: View Results in Table, Response Time Graph, and Graph Results. Each listener view displays a different kind of data. Add all the preceding listeners by right-clicking on **Packt Thread Group** and then navigating to **Add** | **Listeners**. We will have a complete list of all the available listeners. Add all the three listeners one by one. Our final **Packt Publisher**

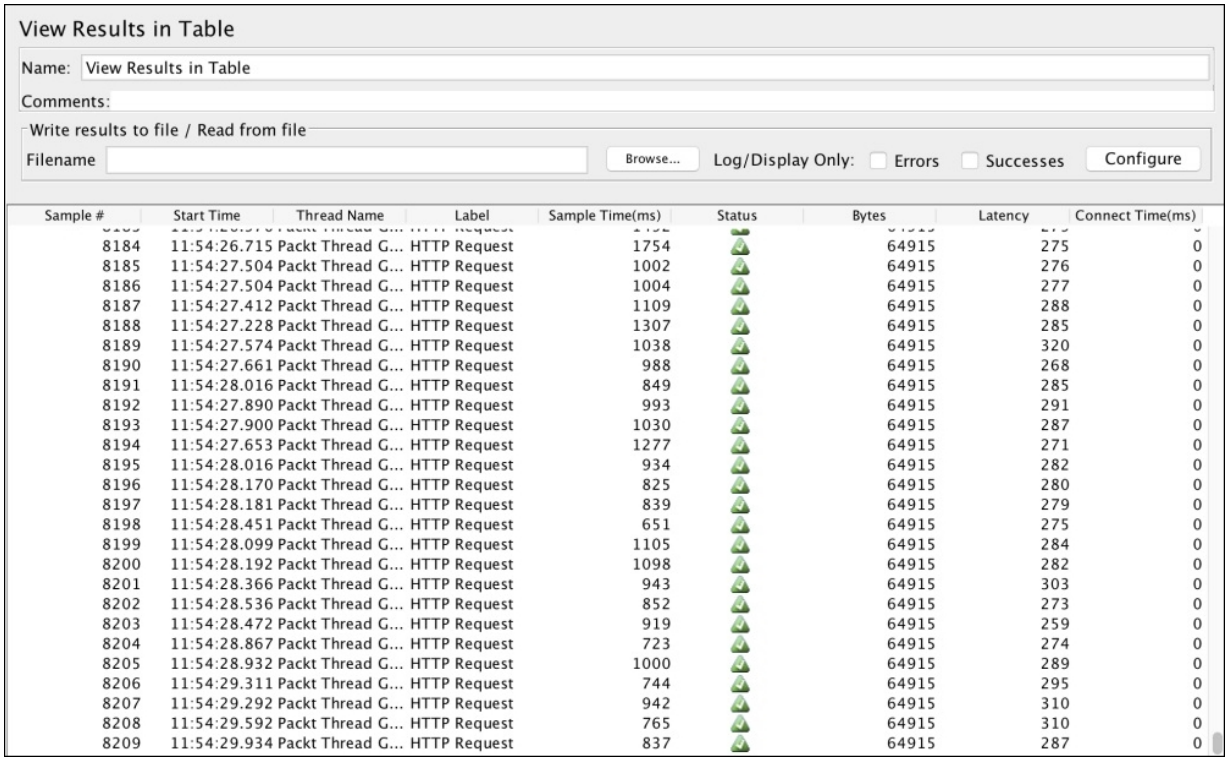
Test Plan panel on the left-hand side of JMeter will look similar to the following:



Now, we are ready to run our test plan by clicking on the **Start** button in the upper tool bar, as shown in the following screenshot:



As soon as we click on the **Start** button (the green arrow pointing to the right-hand side), JMeter will start our test plan. Now, if we click on the **View Results in Table** listener on the left panel, we will see data for each request in a table, as shown in the following screenshot:

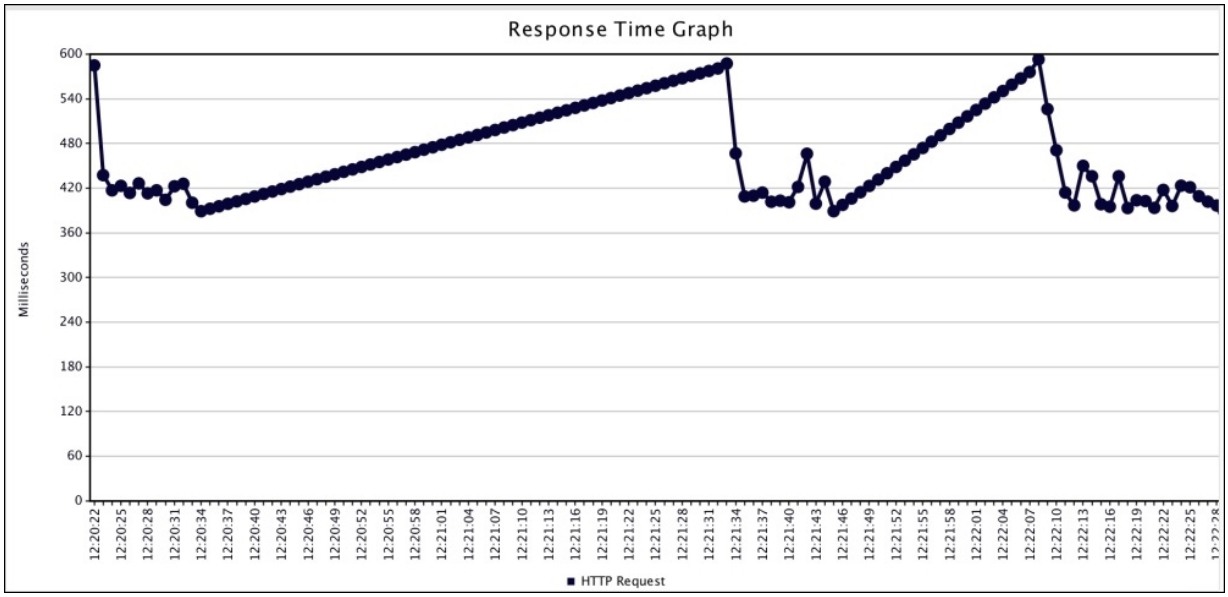


Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Latency	Connect Time(ms)
8184	11:54:26.715	Packt Thread G...	HTTP Request	1754		64915	275	0
8185	11:54:27.504	Packt Thread G...	HTTP Request	1002		64915	276	0
8186	11:54:27.504	Packt Thread G...	HTTP Request	1004		64915	277	0
8187	11:54:27.412	Packt Thread G...	HTTP Request	1109		64915	288	0
8188	11:54:27.228	Packt Thread G...	HTTP Request	1307		64915	285	0
8189	11:54:27.574	Packt Thread G...	HTTP Request	1038		64915	320	0
8190	11:54:27.661	Packt Thread G...	HTTP Request	988		64915	268	0
8191	11:54:28.016	Packt Thread G...	HTTP Request	849		64915	285	0
8192	11:54:27.890	Packt Thread G...	HTTP Request	993		64915	291	0
8193	11:54:27.900	Packt Thread G...	HTTP Request	1030		64915	287	0
8194	11:54:27.653	Packt Thread G...	HTTP Request	1277		64915	271	0
8195	11:54:28.016	Packt Thread G...	HTTP Request	934		64915	282	0
8196	11:54:28.170	Packt Thread G...	HTTP Request	825		64915	280	0
8197	11:54:28.181	Packt Thread G...	HTTP Request	839		64915	279	0
8198	11:54:28.451	Packt Thread G...	HTTP Request	651		64915	275	0
8199	11:54:28.099	Packt Thread G...	HTTP Request	1105		64915	284	0
8200	11:54:28.192	Packt Thread G...	HTTP Request	1098		64915	282	0
8201	11:54:28.366	Packt Thread G...	HTTP Request	943		64915	303	0
8202	11:54:28.536	Packt Thread G...	HTTP Request	852		64915	273	0
8203	11:54:28.472	Packt Thread G...	HTTP Request	919		64915	259	0
8204	11:54:28.867	Packt Thread G...	HTTP Request	723		64915	274	0
8205	11:54:28.932	Packt Thread G...	HTTP Request	1000		64915	289	0
8206	11:54:29.311	Packt Thread G...	HTTP Request	744		64915	295	0
8207	11:54:29.292	Packt Thread G...	HTTP Request	942		64915	310	0
8208	11:54:29.592	Packt Thread G...	HTTP Request	765		64915	310	0
8209	11:54:29.934	Packt Thread G...	HTTP Request	837		64915	287	0

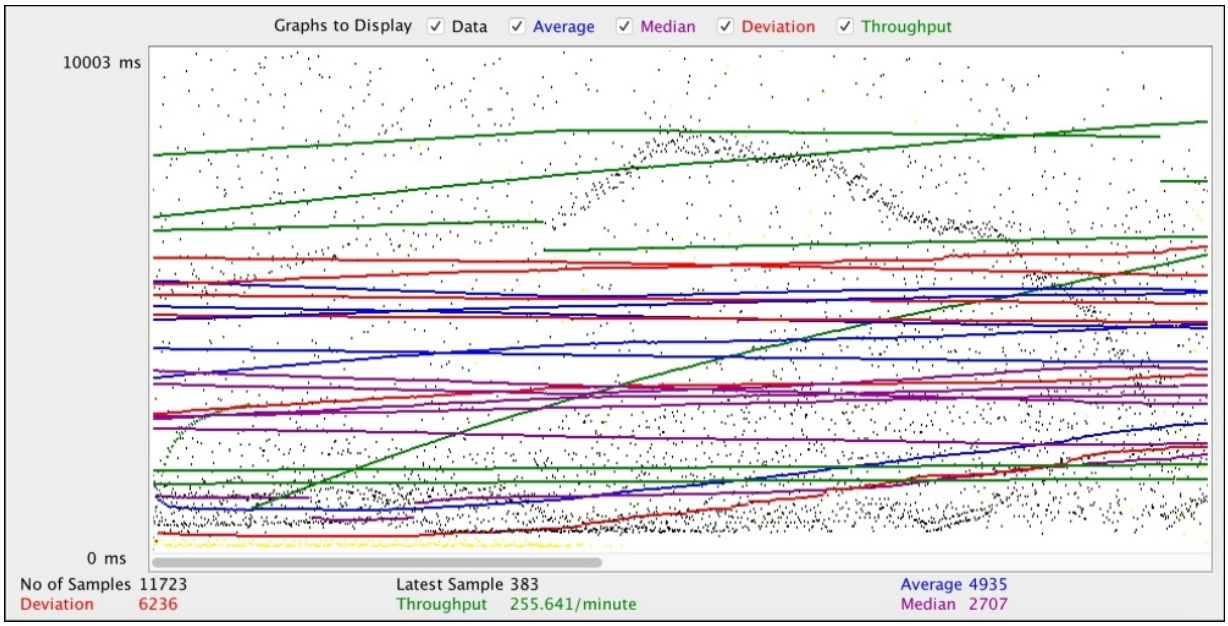
The preceding screenshot shows some interesting data, such as sample time, status, bytes, and latency.

Sample time is the number of milliseconds in which the server served the complete request. **Status** is the status of the request. It can be either a success, warning, or error. **Bytes** is the number of bytes received for the request. **Latency** is the number of milliseconds in which JMeter received the initial response from the server.

Now, if we click on **Response Time Graph**, we will see a visual graph for the response time, which is similar to the one that follows:



Now, if we click on **Graph Results**, we will see the response time data along with graphs for average, median, deviation, and throughput graphs, as shown in the following graph:



Apache JMeter provides very powerful tools to load test our web servers by simulating users. It can provide us with data regarding the amount of load that makes our web server's response slow, and using this data, we can optimize our web server and application.

ApacheBench (ab)

ApacheBench (ab) is also provided by Apache and is a command-line tool. It is a lovely tool for command line lovers. This tool is normally installed on most Linux flavors by default. Also, it is installed with Apache, so if you have Apache installed, you will probably have ab installed too.

The basic syntax for an ab command is as follows:

```
ab -n <Number_Requests> -c <Concurrency>
<Address>:<Port><Path>
```

Let's discuss what each part of the preceding command means:

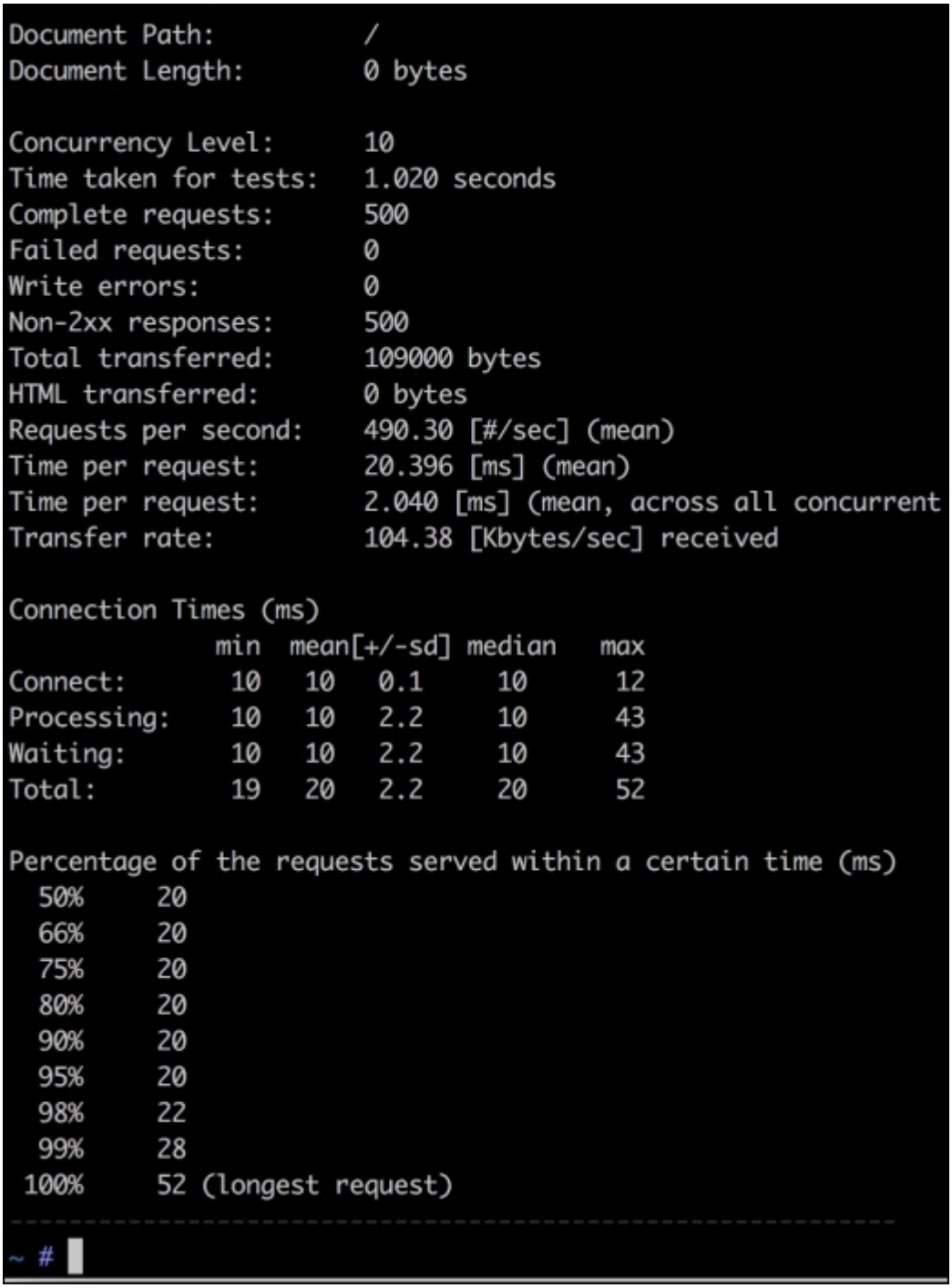
- `n`: This is the number of requests for test.
- `c`: This is concurrency, which is the number of simultaneous requests at a time.
- `Address`: This is either the application URL or IP address of the web server.
- `Port`: This is the port number at which the application is running.
- `Path`: This is the web path of the application that we can use to test. A slash (`/`) is used for the home page.

Now, let's conduct a test using the ab tool by issuing the following command:

```
ab -n 500 -c 10 packtpub.com/
```

As the default port for the web server is 80, it is not required to mention it. Note the slash at the end; this is required to place it there because it is the path's part.

After executing the preceding command, we will have an output that looks similar to the following:



We can see some useful information here, including the number of requests per second, which is **490.3**; the total time taken for the test, which is **1.020 seconds**; the shortest request, which is **20 ms**; and the longest request, which is **52 ms**.

The server load limit can be found by increasing the number of requests and concurrency level and checking the web server's performance.

Siege

Siege is another command-line open source tool to test load and performance. Siege is an HTTP/FTP load tester and benchmarking utility. It is designed for developers and administrators to measure the performance of their applications under load. It can send a configurable number of simultaneous requests to a server and those requests that place the server under a siege.

Its installation is simple and easy. For Linux and Mac OS X, first download Siege by issuing the following command in the terminal:

```
wget http://download.joedog.org/siege/siege-
3.1.4.tar.gz
```

It will download the Siege TAR compressed file. Now, uncompress it by issuing the following command:

```
tar -xvf siege-3.1.4.tar.gz
```

Now, all the files will be in the `siege-3.1.4` folder. Build and install it by issuing the following commands one by one in the terminal:

```
cd siege-3.1.4
./configure
make
make install
```

Now, Siege is installed. To confirm this, issue the following command to check the Siege version:

```
siege -V
```

If it displays the version with some other information, then Siege is installed successfully.

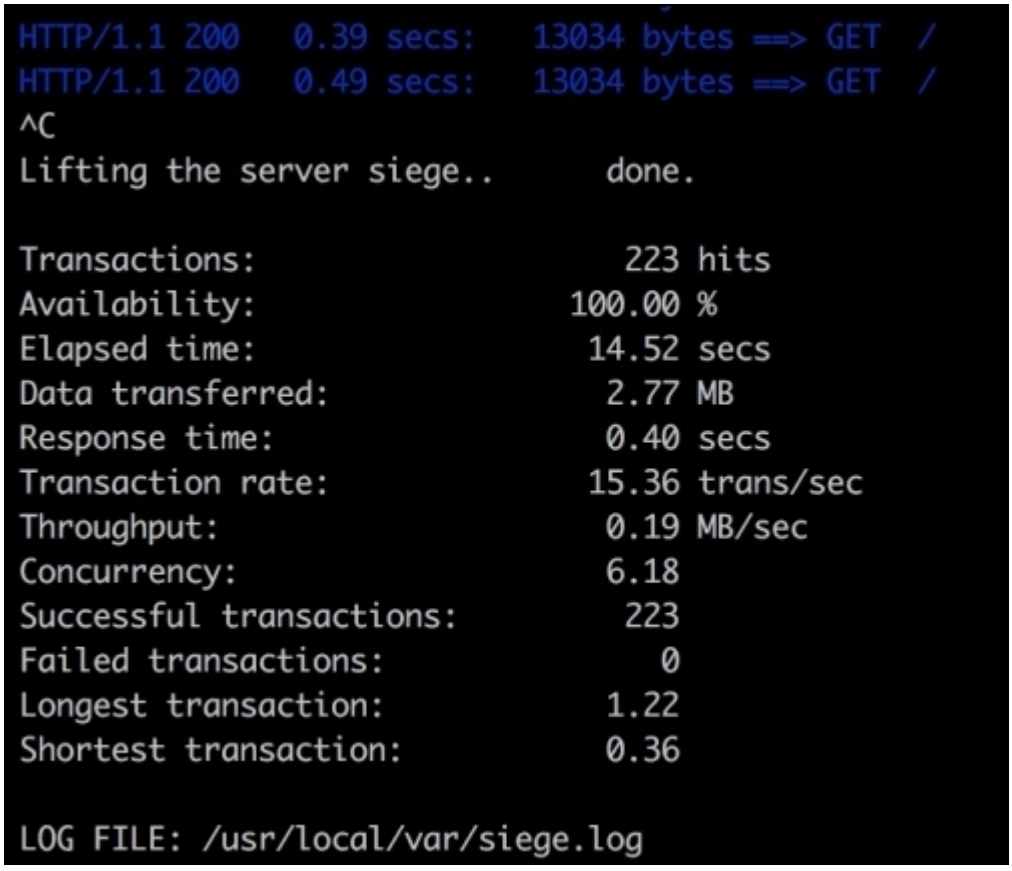
Note

As of writing this book, the current Siege stable version is 3.1.4. Also, Siege does not support Windows natively, and, of course, Windows servers can be tested and benchmarked using Siege.

Now, let's have a load test. A basic load test can be executed by running the following command:

```
siege some_url_or_ip
```

Siege will then start the test. We have to enter the application URL or server IP that we want to load test. To stop the test, press *Ctrl* + *C*, and we will have an output similar to the following:



In the preceding screenshot we can see **Transactions**, **Response time**, and **Transaction rate** along with **Longest transaction** and **Shortest transaction**.

By default, Siege creates 15 concurrent users. This can be changed by using the `-c` option, which is done by making the following alteration in the command:

```
siege url_or_ip -c 100
```

However, Siege has a limitation for the concurrent users, which may be different for each OS. This can be set in the Siege configuration file. To find out the `config` file location and concurrent user limit, issue the following command in terminal:

```
siege -C
```

A list of the configuration options will be displayed. Also the resource file or `config` file location will be displayed. Open that file and find the config concurrent and set its value to an appropriate required value.

Another important feature of Siege is that a file that has all the URLs that need to be tested can be used. The file should have a single URL in each line. The `-f` flag is used with Siege as follows:

```
siege -f /path/to/url/file.txt -c 120
```

Siege will load the file and start load testing each URL.

Another interesting feature of Siege is the internet mode, which can be entered using the `-i` flag in the following command:

```
siege -if path_to_urls_file -c 120
```

In the internet mode, each URL is hit randomly and mimics a real-life situation, in which it can't be predicted which URL will be hit.

Note

Siege has lots of useful flags and features. A detailed list can be found in the official documentation at https://www.joedog.org/siege-manual/.

Load testing real-world applications

We studied three tools in this chapter to load test. Now, it is time to load test some real-world applications. In this section, we will test Magento 2, Drupal 8, and WordPress 4. All these open source tools will have their default data.

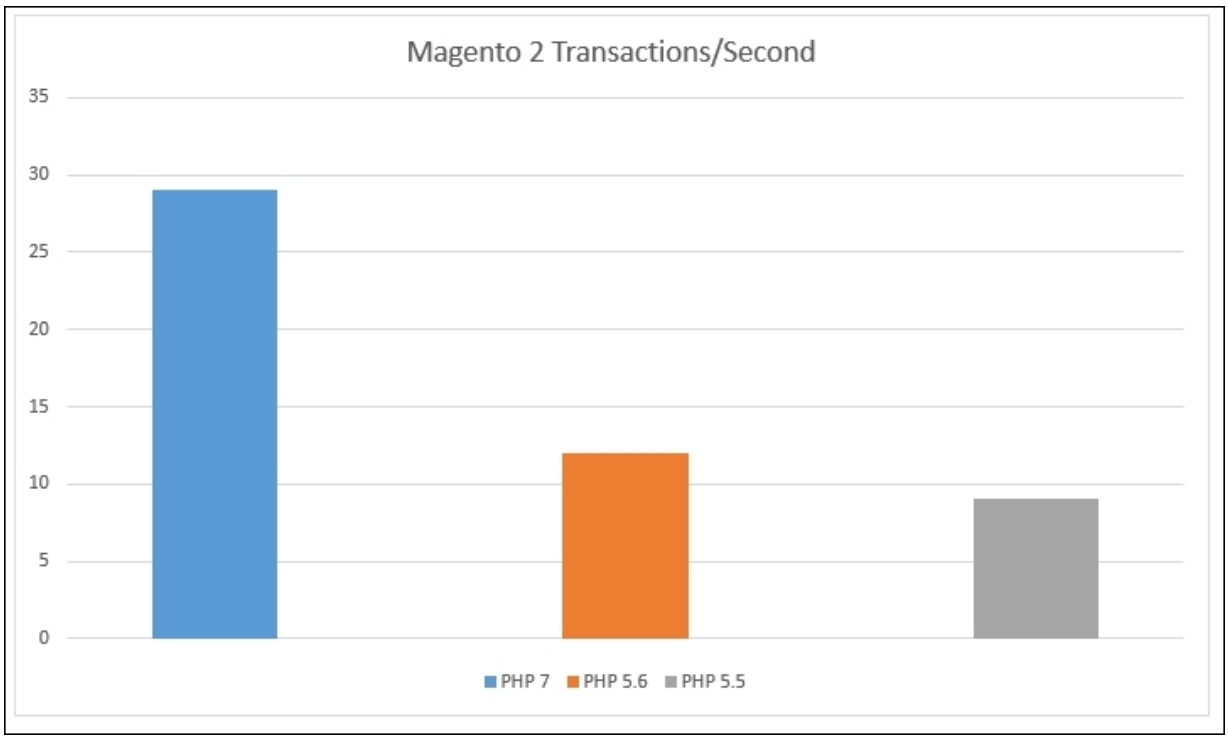
We have three VPS configured with NGINX as the web server. One VPS has PHP 5.5-FPM, the second has PHP 5.6-FPM, and the third has PHP 7-FPM installed. The hardware specs for all the three VPS are same, and all applications we will test will have the same data and the same versions.

This way, we will benchmark these applications with PHP 5.5, PHP 5.6, and PHP 7 and take a look at how fast these applications can run on different versions of PHP.

> **Note**
>
> In this topic, we won't cover configuring the servers with NGINX, PHP, and the databases. We will assume that the VPS are configured and that Magento 2, Drupal 8, and WordPress 4 are installed on them.

Magento 2

Magento 2 is installed on all VPS, and all the caches are enabled for Magento. PHP OPcache is also enabled. After running the tests, we got an average result for all the three Magento 2 installations, as shown in the following graphs:



In the preceding chart, the vertical line, or Y-axis, shows the transactions per second. As can be seen in the charts, Magento 2 on PHP 7 has 29 transactions per second, while the same Magento 2 installation on the same hardware with PHP 5.6 has 12 transactions per second. Also, on PHP 5.5, the same Magento installation has 9 transactions per second. So, in this case, Magento runs about 241% faster on PHP 7 than PHP 5.6 and about 320% faster than in PHP 5.5. This is a very huge improvement of PHP 7 on both PHP 5.6 and PHP 5.5.

WordPress 4

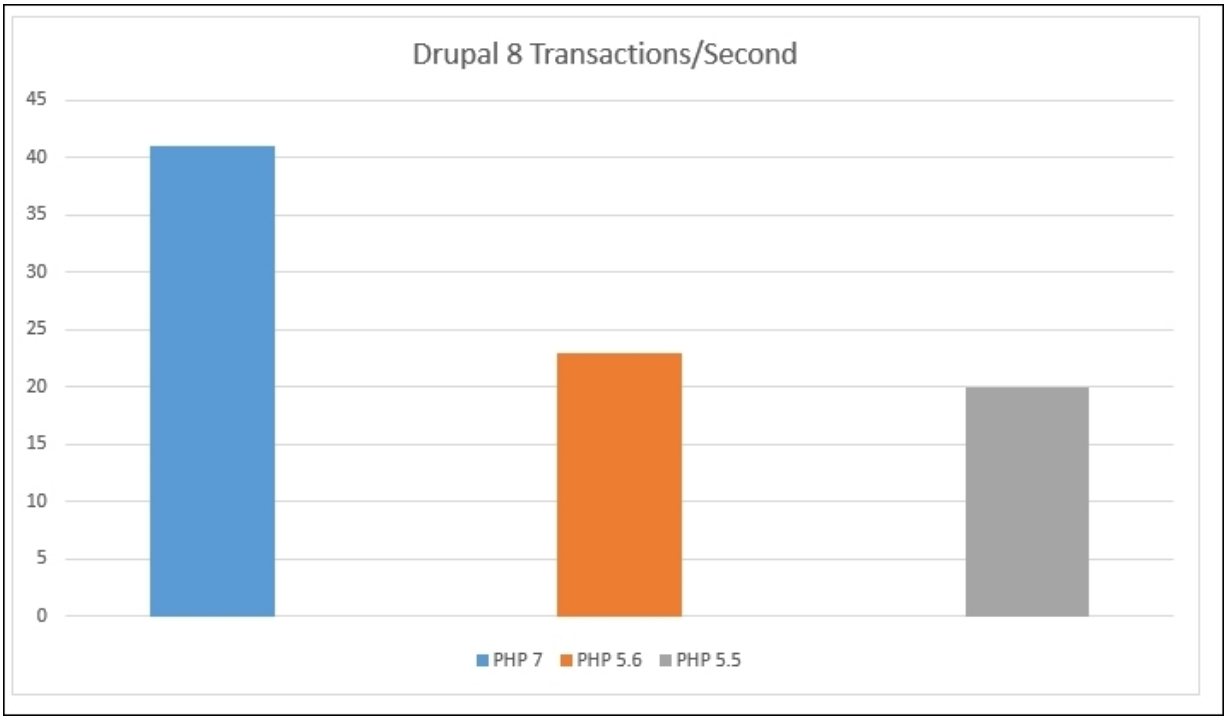
WordPress is installed on all of the three VPS. Unfortunately, there is no default cache embedded into WordPress, and we will not install any third-party modules, so no cache is used. The results are still good, as can be seen in the following graphs. PHP OPcache is enabled.



As can be seen in the preceding graph, WordPress runs 135% faster in PHP 7 than in PHP 5.6 and 182% faster than in PHP 5.5.

Drupal 8

We used the same VPS for PHP 5.5, PHP 5.6, and PHP 7. The default Drupal 8 cache is enabled. After load testing the default home of Drupal 8, we got the following results:

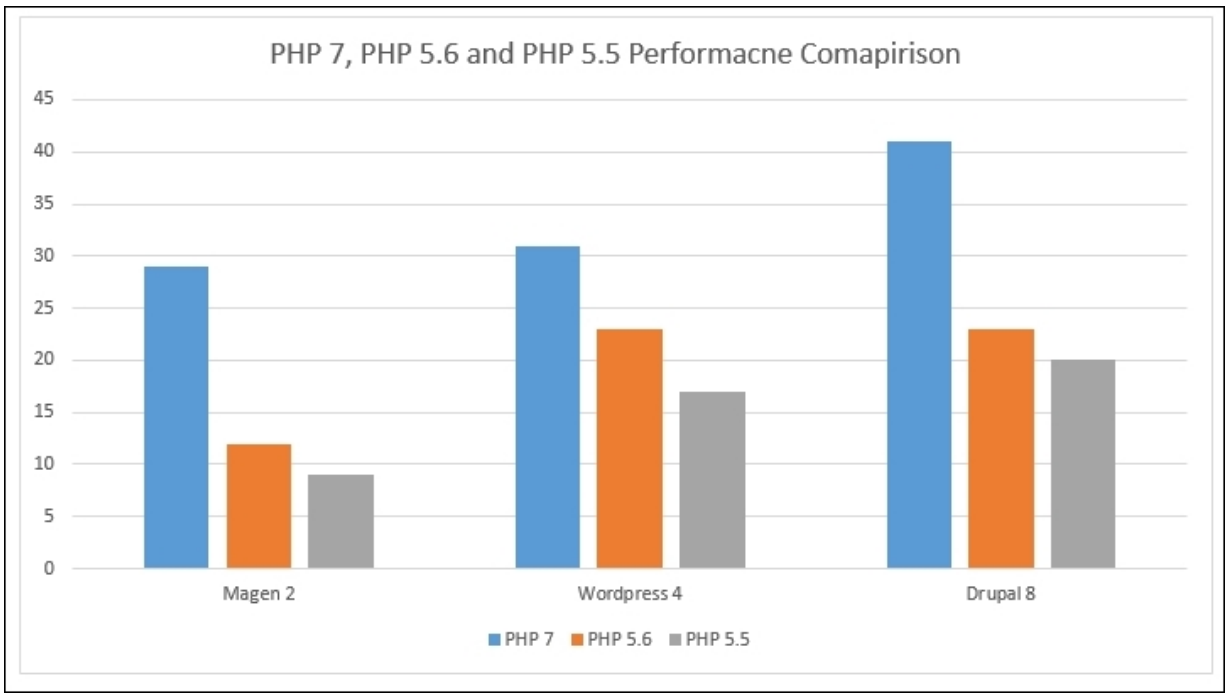


The preceding graph shows that Drupal 8 runs 178% faster in PHP 7 than in PHP 5.6 and 205% faster than in PHP 5.5.

Note

In the preceding graphs, all these values are approximate values. If a low-power hardware is used, then smaller values will be generated. If we use a more powerful multiprocessor-dedicated server with the web server and database optimizations, we will get higher values. The point to consider is that we will always get better performance for PHP 7 than PHP 5.6.

A combined graph is shown here, which displays the performance improvements for different applications in PHP 7 over PHP 5.5 and PHP 5.6:


PHP 7, PHP 5.6 and PHP 5.5 Performacne Comapirison
45
40
35
30
25
20
15
10
5
0
Magen 2
Wordpress 4
Drupal 8
PHP 7
PHP 5.6
PHP 5.5

Summary

In this chapter, we discussed a few load testing and benchmarking tools, such as JMeter, ApacheBench (ab), and Siege. We used each tool to load test, and discussed the output and what it means. Finally, we load tested three famous open source applications, Magento 2, WordPress 4, and Drupal 8, and created graphs for each application's transactions per second in both PHP 7 and PHP 5.6.

In the next chapter, we will discuss best practices for PHP development. These practices are not limited only to PHP and can be used for any programming language.

Chapter 7. Best Practices in PHP Programming

So far, we discussed performance-related topics. Now, in this chapter, we will study best practices in PHP applications' development and deployment. This is a vast topic, but we will cover it briefly. PHP provides all levels of programmers with the ability to write quality code easily and quickly. However, when the application advances to a more complex nature, we forget to follow the best practices. To produce a high performance PHP application, it is necessary to keep in mind the performance at every line of the code.

We will cover the following topics:

- Coding styles
- Design patterns
- Service-oriented architecture (SOA)
- Test-driven development (TDD) and PHPUnit testing
- PHP frameworks
- Version control systems and Git
- Deployment

Coding styles

There are too many coding styles out there, such as PSR-0, PSR-1, PSR-2, PSR-3, and so on. Programmers can use different standards as they want, but it is necessary to follow a standard that is already used in the libraries or a framework in use to make the code more readable. For example, Laravel uses the PSR-1 and PSR-4 coding standards, so if we are developing in Laravel, we should follow these coding standards. Some PHP frameworks, such as Yii 2 and **Zend Framework 2**, follow the PSR-2 coding standards. However, none of these frameworks stick to a single standard; most of them follow a mixed standard according to their requirements.

The important point is to follow the standard that is used in the libraries used in the application. An organization can also use its own coding standards for internal purposes. It is not a requirement for coding; it is a requirement for readability and producing quality code that others can understand.

PHP Framework Interop Group (**PHP-FIG**) is a group whose members defined coding standards for PHP. Full details about PSR standards can be found on their website at http://www.php-fig.org/.

Instead of discussing a specific coding standard, let's discuss best practices in coding styles for PHP:

- The first letter of each word in the class name must be capital. The opening brace should be on the line after the class declaration, and the closing brace should be on the line after the class end line. Here's an example:

```
class Foo
{
  …
  …
  …
}
```

- Class methods and function names should follow the camel case naming convention. The starting braces should be on the next line of the class declaration, and the end brace should be on the line at the end of the function definition. There should be no spaces between the method name and the parenthesis. Also, there should be no space between the first argument, the opening parenthesis, the last argument, and the closing parenthesis. Also, there should be no space between an argument and the comma at the end of this argument, but there should be a space between a comma and the next argument. Here's an example:

```
public function phpBook($arg1, $arg2, $arg3)
{
  …
  …
```

```
  …
}
```

- If there is a namespace declaration, there must be a single empty line after its declaration. If there are use declarations, all of them must go after that namespace's declarations. There must be one use declaration per line, and there must be a space after the use block. Also, the `extends` and `implements` keywords must be on the same line as the class declaration. Here's an example:

```
namespace Packt\Videos;

use Packt\Books;
use Packt\Presentations;

class PacktClass extends VideosClass
implements BaseClass
{
  …
  …
  …
}
```

- Visibility must be declared for all properties, and the properties must be in camel case. Also, properties must not be prepended with an underscore for private or protected visibilities. Take a look at the following example:

```
class PacktClass
{
  public $books;
  private $electronicBooks;
  …
  …
  …
}
```

- If there is an `abstract` keyword, it must come before the `class` keyword for classes, and the `final` keyword must come before the method's visibility in the case of methods. On the other hand, the `static` keyword must come after the method visibility. Take a look at this example:

```
abstract class PacktClass
{
  final public static function favoriteBooks()
  {
    …
    …
    …
  }
}
```

- All PHP keywords must be used in lowercase, including the `true` and `false` keywords. Constants must be declared and used in capital case.
- For all control structures, there must be a space after the control structure keyword. If there is an expression for this control structure, there must be no space between the parenthesis holding this expression and the block of code that follows. There must be a space after the parenthesis and the starting brace. The starting brace must be on the same line as the control structure. The closing brace must be on the line after the end of the body. Refer to the following code for a better understanding:

```
if ($book == "PHP 7") {
  …
  …
  …
} else {
  …
  …
  …
}
```

- In the case of loops, the spaces must be as in the following examples:

```
for ($h = 0; $h < 10; $h++) {
  …
  …
  …
}

foreach ($books as $key => $value) {
  …
  …
```

```
  …
}

while ($book) {
  …
  …
  …
}
```

For the purpose of this book, I did not follow the rule of the opening brace being on the same line as the control structure declaration and always used it on the next line of the declaration. I did not find it clearer; it is a personal choice, and anybody can follow the standards mentioned here.

Standards are good to follow as they make the code more readable and professional. However, never try to invent your own new standards; always follow those that are already invented and followed by the community.

Test-driven development (TDD)

Test-driven development is the process of testing every aspect of the application during development. Either the tests are defined before development and then development is made to pass these tests, or the classes and libraries are built and then tested. Testing the application is very important, and launching an application without tests is like jumping from a 30-floor-high building without a parachute.

PHP does not provide any built-in features to test, but there are other test frameworks that can be used for this purpose. One of most widely used frameworks or libraries is PHPUnit. It is a very powerful tool and provides lots of features. Now, let's have a look at it.

The installation of PHPUnit is easy. Just download it and place it in your project root so that it can be accessed from the command line.

> ## Note
>
> PHPUnit installation and basic details, including features and examples, can be found at https://phpunit.de/.

Let's have a simple example. We have a `Book` class, as follows:

```
class Book
{
  public $title;
  public function __construct($title)
  {
    $this->title = $title;
}

  public function getBook()
  {
    return $this->title;
```

```
    }
}
```

This is an example of a simple class that initializes the `title` property when the class is instantiated. When the `getBook` method is called, it returns the title of the book.

Now, we want to make a test in which we will check whether the `getBook` method returns `PHP 7` as a title. So, perform the following steps to create the test:

1. Create a `tests` directory at your project's root. Create a `BookTest.php` file in the `tests` directory.
2. Now, place the following code in the `BookTest.php` file:

```
include (__DIR__.'/../Book.php');

class BookTest extends
PHPUnit_Framework_TestCase
{
  public function testBookClass()
  {
    $expected = 'PHP 7';
    $book = new Book('PHP 7');
    $actual = $book->getBook();
    $this->assertEquals($expected, $book);
  }
}
```

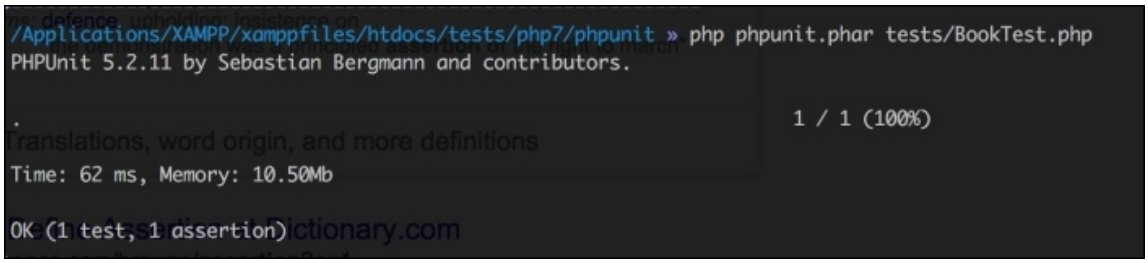
3. Now, we have written our first test. Note that we named our class `BookTest`, which extends the `PHPUnit_Framework_TestCase` class. We can name our test class whatever we want. However, the name should be easily recognizable so that we know this is written for the class that needs to be tested.
4. Then, we added a method named `testBookClass`. We are also free to select whatever name we want to give to this method, but it should start with the word `test`. If not, PHPUnit will not execute the method and will issue a warning—in our case, for the preceding test class—that no tests were found.

In the `testBookClass` method, we created an object of the `Book` class and passed `PHP 7` as our title. Then, we fetched the title using the `getBook` method of the `Book` class. The important part is the last line of the `testBookClass` method, which performs the assertion and checks whether the data returned from `getBook` is the desired data or not.

5. Now, we are ready to run our first test. Open the command line or terminal in the root of the project and issue the following command:

```
php phpunit.phar tests/BookTest.php
```

When the command is executed, we will have an output similar to the following screenshot:

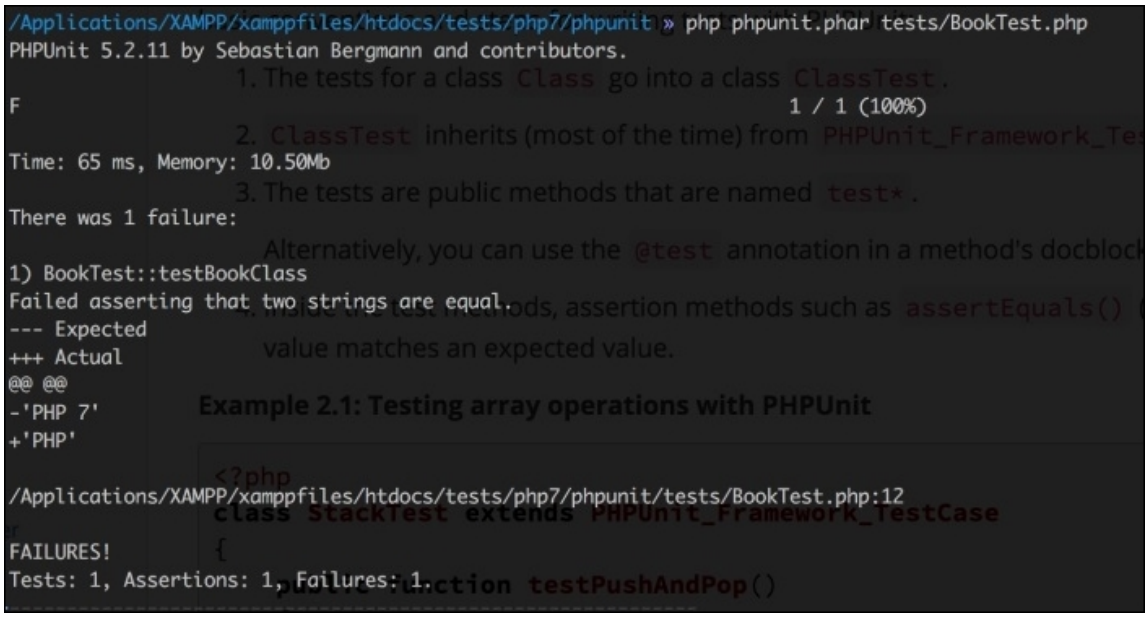


Our test is executed successfully as it met the criteria defined in our test.

6. Now, let's change our class a little bit and pass `PHP` to the `Book` class, as shown in the following code:

```
public function testBookClass()
{
  $book = new Book('PHP');
  $title = $book->getBook();
  $this->assertEquals('PHP 7', $book);
}
```

7. Now, we are looking for PHP 7, and our `Book` class returns `PHP`, so it does not pass our test. After executing this test, we will have a failure, as shown in the following screenshot:



As seen in the preceding screenshot, we expected `PHP 7`, and we got an actual result of `PHP 7`. The `-` sign shows the expected value, and the `+` sign shows the actual value.

Note

In the previous topic, we discussed how we can perform tests on our libraries. We only discussed a simple basic test. PHPUnit is not limited to these simple tests, but covering PHPUnit completely is out of the scope of this book. A very nice book on PHPUnit is *PHPUnit Essentials*, published by Packt Publishing.

Design patterns

A design pattern solves a specific problem. It is not a tool; it is just a description or template that describes how to solve a specific problem. Design patterns are important, and they play a good role in writing clean and clear code.

One of the most widely used design patterns in the PHP community is the **Model View Controller** (**MVC**) pattern. Most PHP frameworks are built upon this pattern. MVC advises you to keep the business logic and data operations (that is, the model) separate from the presentation (the view). Controllers just play the role of a middleman between models and views and make the communication between them possible. There is no direct communication between models and views. If a view needs any kind of data, it sends a request to the controller. The controller knows how to operate on this request and, if needed, make a call to the model to perform any operation on the data (fetch, insert, validate, delete, and so on). Then at last, the controller sends a response to the view.

In best practices, fat models and skinny controllers are used. This means that controllers are only used to take a specific action on a request and nothing else. Even in some modern frameworks, the validation is moved out of the controllers and is performed at the model level. These models perform all the operations on the data. In modern frameworks, models are considered as a layer, which can have multiple parts, such as the business logic, **Create Read Update Delete** (**CRUD**) database operations, data mapper pattern and services, and so on. So, a full load of models and controllers is just sitting there and enjoying the lazy work load.

Another widely used design pattern is the factory design pattern. This pattern simply creates objects that are needed to be used. Another good pattern is the observer pattern, in which an object calls different observers on a specific event or task on it. This is mainly used for event handling. Yet another widely used pattern is the

singleton pattern, which is used when there is a requirement that only a single object of a class be used throughout the application's execution. *A singleton object can't be serialized and cloned*.

Service-oriented architecture (SOA)

In service-oriented architecture, the application's components provide services to each other on a defined protocol. Each component is loosely coupled with each other, and the only way of communication between them is through the services they provide.

In PHP, Symfony provides the best way to have SOA as it is mainly an HTTP-centric framework. Symfony is the most mature, well-tested collection of libraries that are widely used by other PHP frameworks, such as Zend Framework, Yii, Laravel, and others.

Let's consider a scenario where we have a backend and a frontend for a website and a mobile application. Normally, in most applications, the backend and frontend run on the same code base and on a single access point, and an API or web service is built for mobile applications to communicate with this backend. It is good, but we need great. So, for high performance and scalable applications, the separate components run independently of each other. If they need to communicate with each other, they communicate through the web services.

Web services are the central communication point between the frontend and backend and between the backend and mobile applications. The backend is the main hub of data and any other business logic. It can be standalone and built using any programming language, such as PHP. The frontend can be built using normal HTML/CSS, AngularJS, Node.js, jQuery, or any other technology for the frontend. Similarly, mobile apps can be native or built on cross-platform technologies. The backend doesn't care what the frontend and mobile apps are built on.

Being object-oriented and reusable always

This may seem difficult for a small, single-page application in which only a few things are happening, but this is not the case. The classes are easy to handle, and the code is always clear. Also, the classes separate the application logic from the views. This make things more logical. In the earlier days when structure code was used and a bunch of functions had to be created either in the view files or in a separate file, this would have been too easy. However, when applications got more complex, it got more difficult to handle.

Always try to create loosely coupled classes to make them more reusable in other applications. Also, always perform a single task in each method of the class.

PHP frameworks

We all know about frameworks, and they are not essential to a programmer's life. There are lots of frameworks, and each framework has its own superiority over other frameworks in some features. All frameworks are good, but what make a framework not suitable for an application are the application's requirements.

Let's say that we want to build an enterprise-level CRM application, which framework will suit us best? This is the most important, confusing, and time-wasting question. First, we need to know the complete requirements for the CRM application, usage capacity, features, data security, and performance.

Version control system (VCS) and Git

Version controller system provides the flexibility to properly maintain code, changes, and versions of the application. Using VCS, a complete team can work together on an application, and they can pull other team members' changes and their own changes to the system without any big troubles. In case of a disaster, VCS provides the ability to fall back to an old, more stable version of the application.

Oh wait! Are we talking about VCS? Did we mention Git? Nope! So, let's start with Git.

Git is a powerful tool. It monitors changes in each file in a branch, and when pushed to a remote branch, only the changed files are uploaded. Git keeps a history of the file changes and provides you with the ability to compare the changed files.

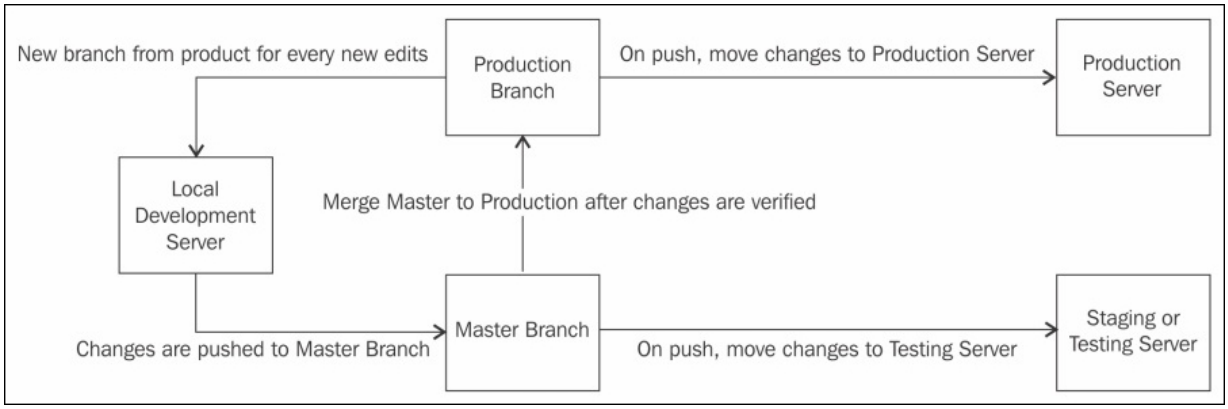
> **Note**
>
> A very informative and good book on Git is *Git Essentials* published by Packt Publishing. Also, an official and free book about Git can be found at https://git-scm.com/book/en/v2.

Deployment and Continuous Integration (CI)

FTP is obsolete. It is not feasible for today, it makes things slow, and a normal FTP connection is insecure. It is hard for a team to deploy their changes using FTP because it creates huge conflicts in their code and this may cause problems, while uploading changes and can override each other's changes.

Using a Git versioning system, such as GitHub, GitLab, and Bitbucket, we can make our deployment automatic. Different developers use different setups for automatic deployments, and it all depends on their own choice and ease. The general rules of using automatic deployments are to make them easy for a team and to not use FTP.

The following is a general flowchart for a deployment setup:



As shown in the preceding flowchart, we have two servers: the staging or testing the server and production server. On the staging server, we have an exact copy of the website to test new features and others, and the production server has our live website.

Now, we have a repository that has two main branches: the master branch and the production branch. The master branch is used for development and testing purposes, and the production branch is

used for final production features. Note that the production branch should only accept merging, and it should not accept commits so that the production environment is completely safe.

Now, let's say that we want to add a customer registration feature to our application. We will perform the following steps:

1. The first and most important thing to do is to create a new branch from the production branch head. Let's name this branch `customer-registration`.
2. Now, add all the new features to this `customer-registration` branch and while verifying on the local development server, merge this branch to the local master branch.
3. After merging the new branch to the local master branch, push the master branch to remote master branch. A successful push will cause the new features to be moved to the staging server.
4. Now, test all the new features on the staging server.
5. When everything works fine, merge the remote master branch with the remote production branch. This will cause all the changes to be moved to the production branch, and this merge will cause all the new changes to be moved to the production server.
6. An ideal setup similar to the preceding one makes deployment very easy, and a complete team can work on the application regardless of the geographical location. In case any issue occurs during the deployment, one can be easily fall back to the old version of the production branch.

Continuous Integration (**CI**) is a technique in which all the members of a team have to integrate their code into a shared repository, and then each check by the team member is verified by automatic builds to catch errors and problems in the early stages.

There are several tools that are used for CI for PHP; some of these are PHPCI, Jenkins, Travis CI, and others.

Summary

In this chapter, we discussed a few best practices, including coding standards and styles, PHP frameworks, design patterns, Git, and deployment. Also, we discussed the PHPUnit framework to test classes and libraries against tests. Also, we discussed Service-oriented design, which plays a major role in creating APIs for applications.

In this book, we studied setting up development environments, including Linux servers, specifically Debian and Ubuntu, and we also discussed Vagrant. The new features of PHP are also listed with sample codes. You read in detail about the tools that we can use to improve the performance of an application and a database. Also, we discussed debugging and stress or load testing our applications and some best practices of writing quality code.

We mostly summarized the tools and techniques with simple examples to introduce the reader to these tools and techniques. There is a good chance that each tool and technique has its own book written for a more advanced usage. We recommend you follow up on these tools and techniques and conduct more research for their advance usage. Good luck Php-ing!

Appendix A. Tools to Make Life Easy

We covered many things in this book, starting with new features in PHP 7 and ending with the best techniques in programming. In each chapter, we used and talked about some tools, but due to the finite length of chapters and the book, we did not go too much in detail for these tools. In this appendix, we will discuss three of these tools in much more detail. The tools we will to discuss are as follows:

- Composer
- Git
- Grunt watch

So, let's start.

Composer – A dependency manager for PHP

Composer is a dependency management tool for PHP that enables us to define dependencies for a PHP application, and Composer installs/updates them. Composer is completely written in PHP and is an application in the PHP Archive (PHAR) format.

> ### Note
>
> Composer downloads dependencies from https://packagist.org/. Any dependency for an application can be installed through Composer as long as it is available on Packagist. Also, complete applications can be installed through Composer if they are available at Packagist.

Composer installation

Composer is a command line tool and can be installed globally in the operating system, or the `composer.phar` file can be placed in the root of the application and then executed from the command line. For Windows, an executable setup file is provided, which can be used to install Composer globally. For this book, we will follow the instructions for Debian/Ubuntu globally. Perform the following steps:

1. Issue the following command to download the Composer installer. The file name is `installer` and can only be executed with PHP once installed via the following code:

```
Wget https://getcomposer.org/installer
```

2. Issue the following command to install it globally on Debian or Ubuntu:

```
Php install --install-dir=/usr/local/bin --
filename=composer
```

   This command will download Composer and will install it in the `/usr/local/bin` directory with the file name `composer`. Now, we will be able to run Composer globally.

3. Verify the Composer installation by issuing the following command in the terminal:

```
Composer --version
```

   If the Composer version is displayed, then Composer is successfully installed globally.

> ## Note
>
> If Composer is installed locally to an application, then we will have a `composer.phar` file. The commands are the same, but

all the commands should be executed with PHP. For example, `php composer.phar --version` will display the Composer version.

Now, Composer is installed successfully and is working; it's time to use it.

Using Composer

To use Composer in our project, we will need a `composer.json` file. This file contains all the dependencies required for the project and some other metadata. Composer uses this file to install and update different libraries.

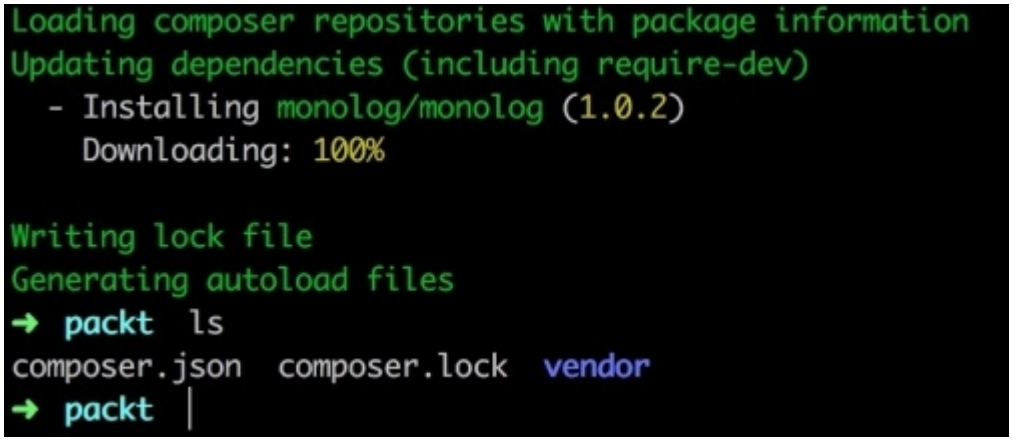
Let's assume that our application needs to log different information in different ways. For this, we can use the `monolog` library. First, we will create a `composer.json` file in the root of our application and add the following code to it:

```
{
  "require": {
    "monolog/monolog": "1.0.*"
  }
}
```

After saving the file, execute the following command to install the dependencies of the application:

```
Composer install
```

This command will download the dependencies and place them in the `vendor` directory, as can be seen in the following screenshot:



As can be seen in the preceding screenshot, monolog version 1.0.2 is downloaded, and a `vendor` directory is created. The `monolog` library is placed in this directory. Also, if a package has to autoload information, then Composer places the library in the Composer autoloader, which is also placed in the `vendor` directory. So, any new libraries or dependencies will be autoloaded automatically during the application's execution.

Also a new file can be seen, which is `composer.lock`. When Composer downloads and installs any dependencies, the exact version and other information is written to this file to lock the application to this specific version of dependencies. This ensures that all the team members or whoever wants to set up the application will use the exact same version of the dependencies, and thus, it will reduce the chances of using different versions of dependencies.

Nowadays, Composer is widely used for package management. Big open source projects such as Magento, Zend Framework, Laravel, Yii, and many others are easily available for installation through Composer. We will install some of these in the next appendix using Composer.

Git – A version control system

Git is the most widely used version control system. According to the Git official website, it is a distributed version control system capable of handling everything from small- to large-sized projects with speed and efficiency.

Git installation

Git is available for all major operating systems. For Windows, an executable setup file is provided that can be used to install Git and use it in the command line. On OS X, Git comes already installed, but if it is not found, it can be downloaded from their official website. To install Git on Debian/Ubuntu, just issue the following command in the terminal:

```
sudo apt-get install git
```

After installation, issue the following command to check whether it is properly installed:

```
git –version
```

Then, we will see the current installed version of Git.

Using Git

For a better understanding of Git, we will start with a test project. Our test project name is `packt-git`. For this project, we also created a GitHub repository named `packt-git`, where will push our project files.

First, we will initialize Git in our project by issuing the following command:

```
git init
```

The preceding command will initialize an empty Git repository in our project root directory, and the head will be kept on the master branch, which is the default branch for every Git repository. It will create a hidden `.git` directory that will contain all the information about the repository. Next, we will add a remote repository that we will create on GitHub. I created a test repository at GitHub that has the URL https://github.com/altafhussain10/packt-git.git.

Now, issue the following command to add the GitHub repository to our empty repository:

```
git remote add origin
https://github.com/altafhussain10/packt-git.git
```

Now, create a `README.md` file at your project root and add some content to it. The `README.md` file is used to display the repository information and other details about the repository at Git. This file is also used to display instructions regarding how to use the repository and/or the project for which this repository is created.

Now, issue the following command to see the status of our Git repository:

```
git status
```

This command will display the status of the repository, as can be seen in the following screenshot:

```
~/packt-git(branch:master*) » git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        README.md

nothing added to commit but untracked files present (use "git add" to track)
```

As can be seen in the preceding screenshot, we have an untracked file in our repository that is not committed yet. First, we will add the files to be tracked by issuing the following command in the terminal:

```
git add README.md
```

The `git add` command updates the index using the current contents found in the working tree. This command adds all the changes made to the path. There are some options that can be used to add some specific changes. The previous command we used will only add the `README.md` file to the track in the repository. So, if we want to track all the files, then we will use the following command:

```
git add
```

This will start tracking all the files in the current working directory or at the root of the current branch. Now, if we want to track some specific files, such as all files with the `.php` extension, then we can use it as follows:

```
git add '*.php'
```

This will add all the files with the `.php` extension to track.

Next, we will commit changes or additions to our repository using the following command:

```
git commit -m "Initial Commit"
```

The `git commit` command commits all the changes to the local repository. The `-m` flag specifies any log message to `commit`. Remember that the changes are only committed to the local repository.

Now, we will push the changes to our remote repository using the following command:

```
git push -u origin master
```

The preceding command will push all the changes from the local repository to the remote repository or origin. The `-u` flag is used to set the upstream, and it links our local repo to our remote central repo. As we pushed our changes for the first time, we have to use the `-u` option. After this, we can just use the following command:

```
git push
```

This will push all the changes to the main repository of the current branch at which we are.

Creating new branches and merging

New branches are always required during development. If any kind of changes are required, it is good to create a new branch for these changes. Then, make all the changes on this branch and finally commit, merge, and push them to the remote origin.

To better understand this, let's suppose we want to fix an issue in the login page. The issue is about validation errors. We will name our new branch `login_validation_errors_fix`. It is good practice to give a more understandable name to branches. Also, we would like to create this new branch from the master branch head. This means that we want the new branch to inherit all the data from the master

branch. So, if we are not at the master branch, we have to use the following command to switch to the master branch:

```
git checkout master
```

The preceding command will switch us to the master branch no matter which branch we are at. To create the branch, issue the following command in the terminal:

```
git branch login_validation_errors_fix
```

Now, our new branch is created from the master branch head, so all the changes should be made to this new branch. After all the changes and fixes are done, we have to commit the changes to the local and remote repositories. Note that we did not create the new branch in our remote repository. Now, let's commit the changes using the following command:

```
git commit -a -m "Login validation errors fix"
```

Note that we did not use `git add` to add the changes or new additions. To automatically commit our changes, we used the `-a` option in `commit`, which will add all the files automatically. If `git add` is used, then there is no need to use the `-a` option in `commit`. Now, our changes are committed to the local repository. We will need to push the changes to the remote origin. Issue the following command in the terminal:

```
git push -u origin login_validation_errors_fix
```

The preceding command will create a new branch at the remote repository, set the tracking of the same local branch to the remote branch, and push all the changes to it.

Now, we want to merge the changes with our master branch. First, we need to switch to our master branch using the following

command:

```
git checkout master
```

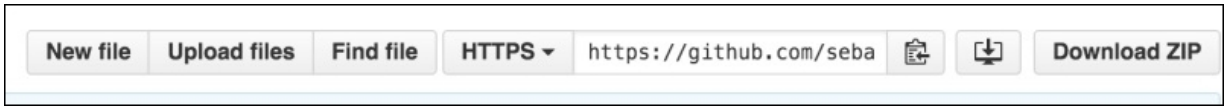
Next, we will issue the following commands to merge our new branch `login_validation_errors_fix` with the master branch:

```
git checkout master
git merge login_validation_errors_fix
git push
```

It is important to switch to the branch to which we want to merge our new branch. After this, we need to use the `git merge branch_to_merge` syntax to merge this branch with the current branch. Finally, we can just push to the remote origin. Now, if we take a look at our remote repository, we will see the new branch and also the changes in our master branch.

Cloning a repository

Sometimes, we need to work on a project that is hosted on a repository. For this, we will first clone this repository, which will download the complete repository to our local system, and then create a local repository for this remote repository. The rest of the working is the same as we discussed before. To clone a repository, we should first know the remote repository web address. Let's say that we want to clone the `PHPUnit` repository. If we go to the GitHub repository for PHPUnit, we will see the web address of the repository at the upper right-hand side, as shown in the screenshot that follows:



The URL just after the **HTTPS** button is the web address for this repository. Copy this URL and use the following command to clone this repository:

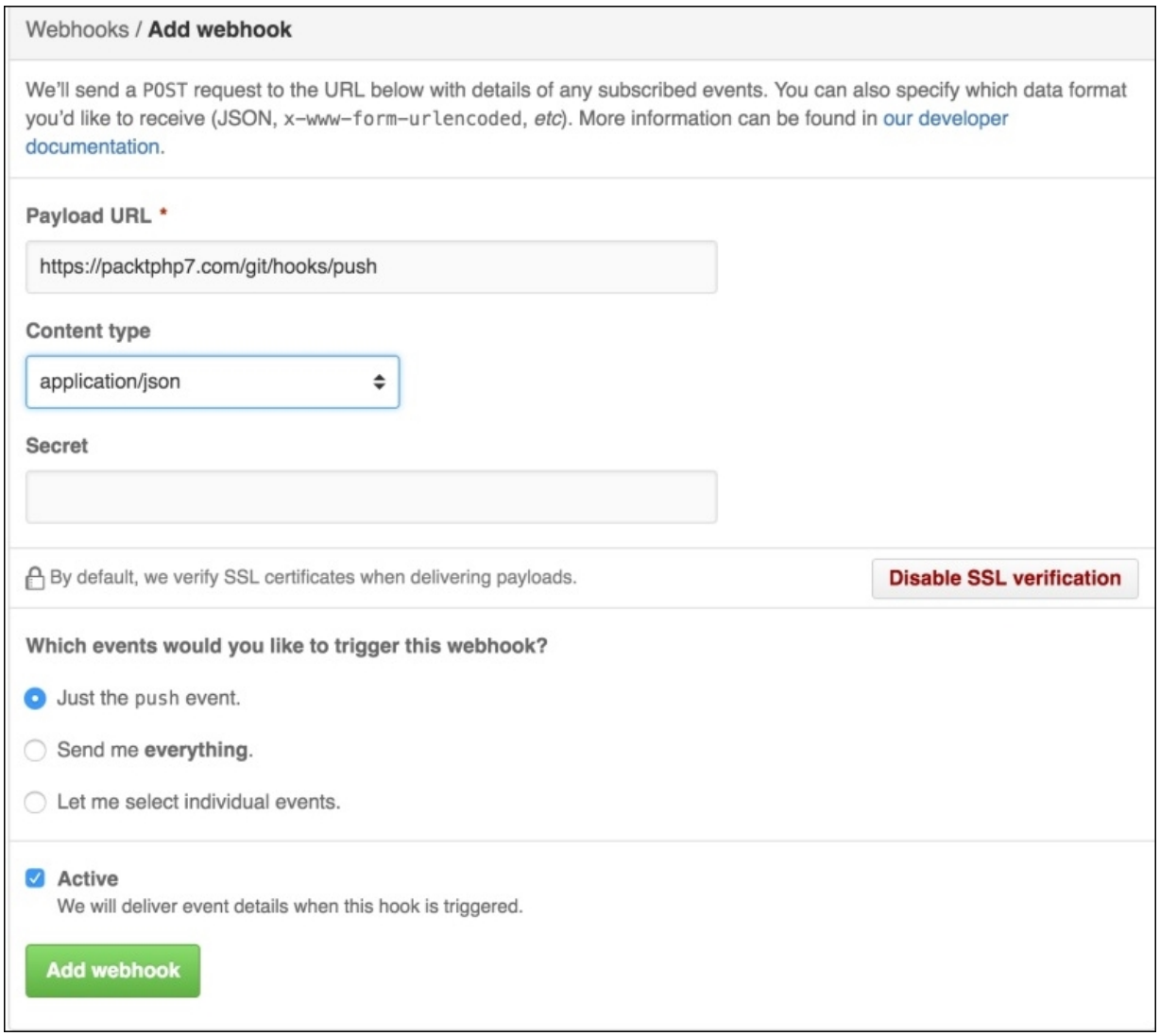
```
git clone
https://github.com/sebastianbergmann/phpunit.git
```

This will start downloading the repository. After it is completed, we will have a `PHPUnit` folder that will have the repository and all its files. Now, all the operations mentioned in the preceding topics can be performed.

Webhooks

One of the most powerful features of Git is webhooks. Webhooks are events that are fired when a specific action occurs on the repository. If an event or hook for the `Push` request is made, then this hook will be fired every time a push is made to this repository.

To add a webhook to a repository, click on the **Settings** link for the repository in the upper right-hand side. In the new page, on the left-hand side, we will have a **Webhooks and Services** link. Click on it, and we will see a page similar to the following one:



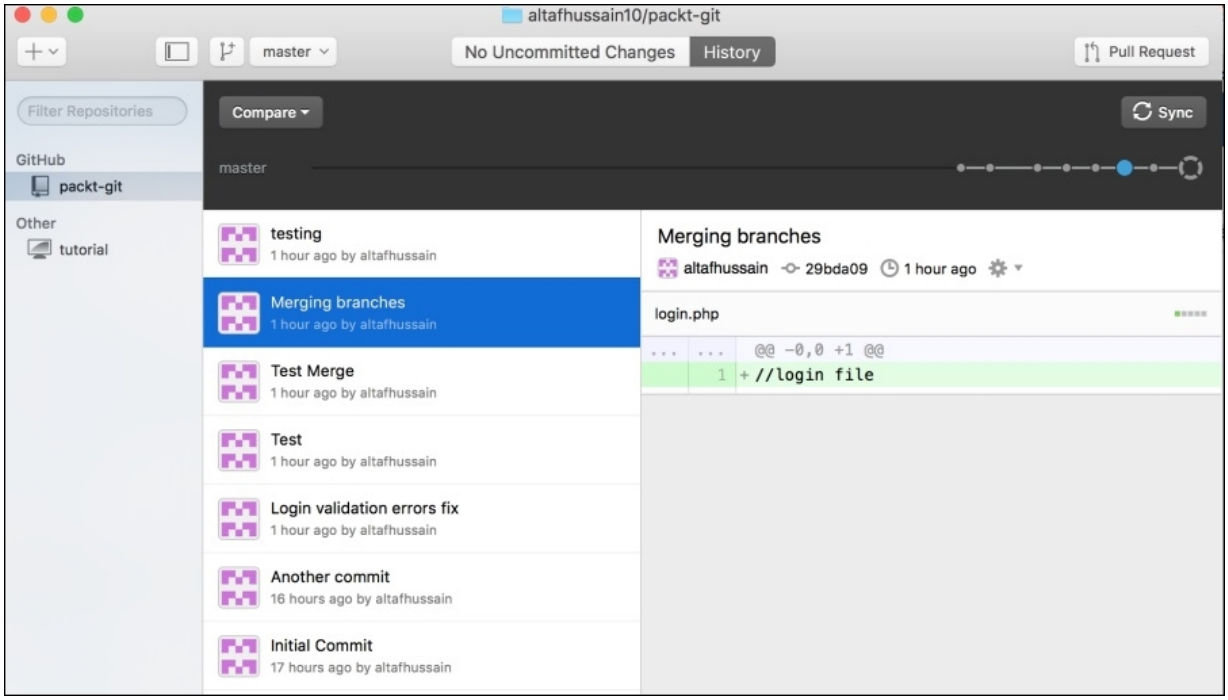
As can be seen in the preceding screenshot, we have to enter a payload URL, which will be called every time our selected event is fired. In **Content type**, we will select the data format in which the payload will be sent to our URL. In the events section, we can select whether we want only push events or all the events; we can select multiple events for which we want this hook to be fired. After saving this hook, it will be fired every time the selected event occurs.

Webhooks are mostly used for deployment. When the changes are pushed and if there is a webhook for the push event, the specific URL is called. Then, this URL executes some command to download the changes and processes them on the local server and places

them at the appropriate place. Also, webhooks are used for continues integration and to deploy to cloud services.

Desktop tools to manage repositories

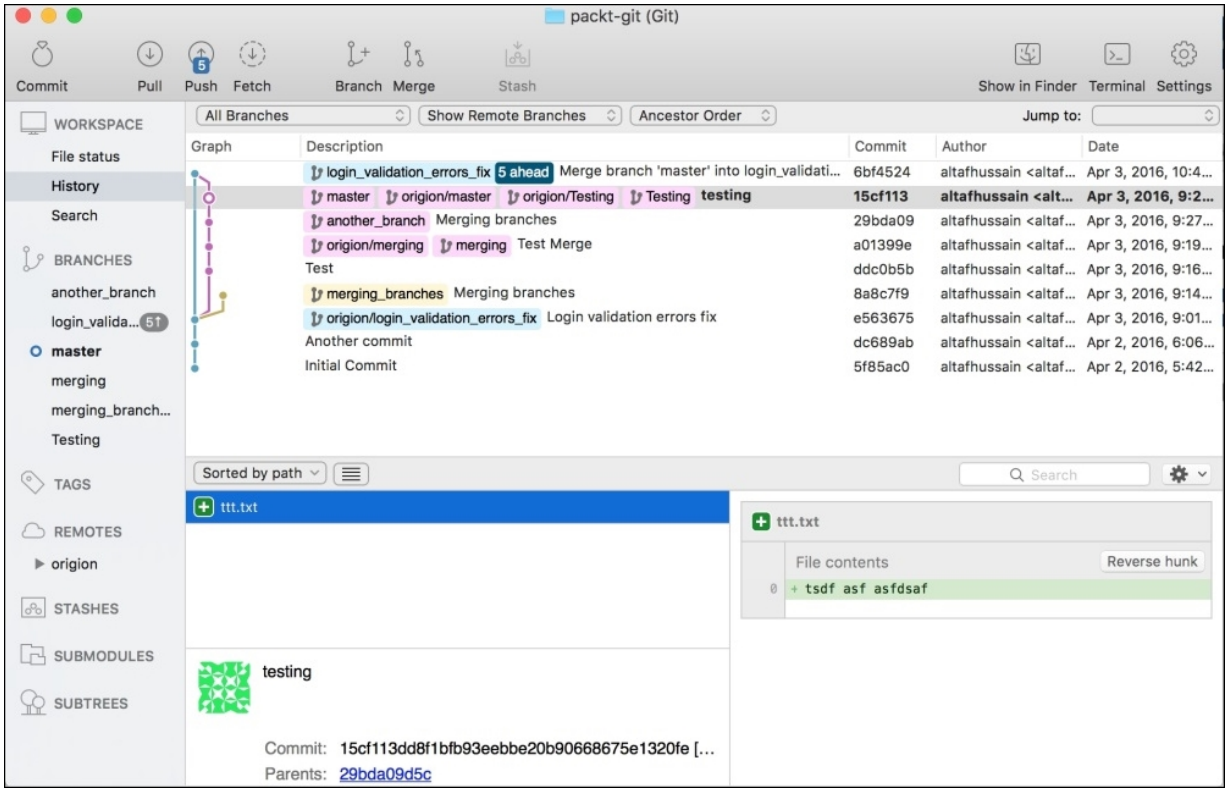
There are several tools that can be used to manage Git repositories. GitHub provides its own tool called GitHub Desktop that can be used to manage GitHub repositories. This can be used to create new repositories, see the history, and push, pull, and clone repositories. It provides every feature that we can use in the command line. The screenshot that follows shows our test `packt-git` repository:



Note

GitHub Desktop can be downloaded from https://desktop.github.com/ and is available for Mac and Windows only. Also, GitHub Desktop can be only used with GitHub unless some hacks are used to make it work with other repositories, such as GitLab or Bitbucket.

Another powerful tool is SourceTree. SourceTree can be used with GitHub, GitLab, and Bitbucket easily. It provides complete features to manage repositories, pull, push, commit, merge, and other actions. SourceTree provides a very powerful and beautiful graph tool for the branches and commits. The following is a screenshot for SourceTree that is used to connect with our `packt-git` test repository:



Besides the previous two nice tools, every development IDE provides version control systems with full support and also provides features such as different colors for modified and newly added files.

Note

Git is a powerful tool; it can't be covered in this appendix. There are several books available, but Git Book is a very good place to start. This can be downloaded in different formats from https://git-scm.com/book/en/v2 or can be read online.

Grunt watch

We studied Grunt in Chapter 3, *Improving PHP 7 Application Performance*. We only used it to merge CSS and JavaScript files and minify them. However, Grunt is not used only for this purpose. It is a JavaScript task runner, which can run tasks either by watching specific files for changes or by manually running tasks. We studied how we can run tasks manually, so now we will study how to use grunt watch to run specific tasks when some changes are made.

Grunt watch is useful and saves a lot of time because it runs the specific tasks automatically instead of running the tasks manually every time we change something.

Let's recall our examples from Chapter 3, *Improving PHP 7 Application Performance*. We used Grunt to combine and compress CSS and JavaScript files. For this purpose, we created four tasks. One task was combining all CSS files, the second task was combining all JavaScript files, the third task was compressing the CSS files, and the fourth task was compressing all JavaScript files. It will be very time consuming if we run all these tasks manually every time we make some changes. Grunt provides a feature called watch that watches different destinations for file changes, and if any change occurs, it executes the tasks that are defined in the watch.

First, check whether the `grunt watch` module is installed or not. Check the `node_modules` directory and see whether there is another directory with the name `grunt-contrib-watch`. If this directory is there, then watch is already installed. If the directory is not there, then just issue the following command in the terminal at the project root directory where `GruntFile.js` is located:

```
npm install grunt-contrib-watch
```

The preceding command will install Grunt watch and the `grunt-contrib-watch` directory will be available with the `watch` module.

Now, we will modify this `GruntFile.js` file to add the `watch` module, which will monitor all the files in our defined directories, and if any changes occur, it will run these tasks automatically. This will save a lot of time in manually executing these tasks again and again. Look at the following code; the highlighted code is the modified section:

```
module.exports = function(grunt) {
  /*Load the package.json file*/
  pkg: grunt.file.readJSON('package.json'),
  /*Define Tasks*/
  grunt.initConfig({
    concat: {
      css: {
      src: [
        'css/*' //Load all files in CSS folder
],
      dest: 'dest/combined.css' //Destination of
the final combined file.

      },//End of CSS
js: {
      src: [
        'js/*' //Load all files in js folder
],
       dest: 'dest/combined.js' //Destination of
the final combined file.

      }, //End of js

}, //End of concat
cssmin:  {
  css: {
    src : 'dest/combined.css',
    dest : 'dest/combined.min.css'
}
}, //End of cssmin
uglify: {
  js: {
        files: {
        'dest/combined.min.js' :
['dest/combined.js']//destination Path : [src
path]
}
}
}, //End of uglify
```

```
//The watch starts here
watch: {
  mywatch: {
    files: ['css/*', 'js/*', 'dist/*'],
    tasks: ['concat', 'cssmin', 'uglify']
  },
},
}); //End of initConfig

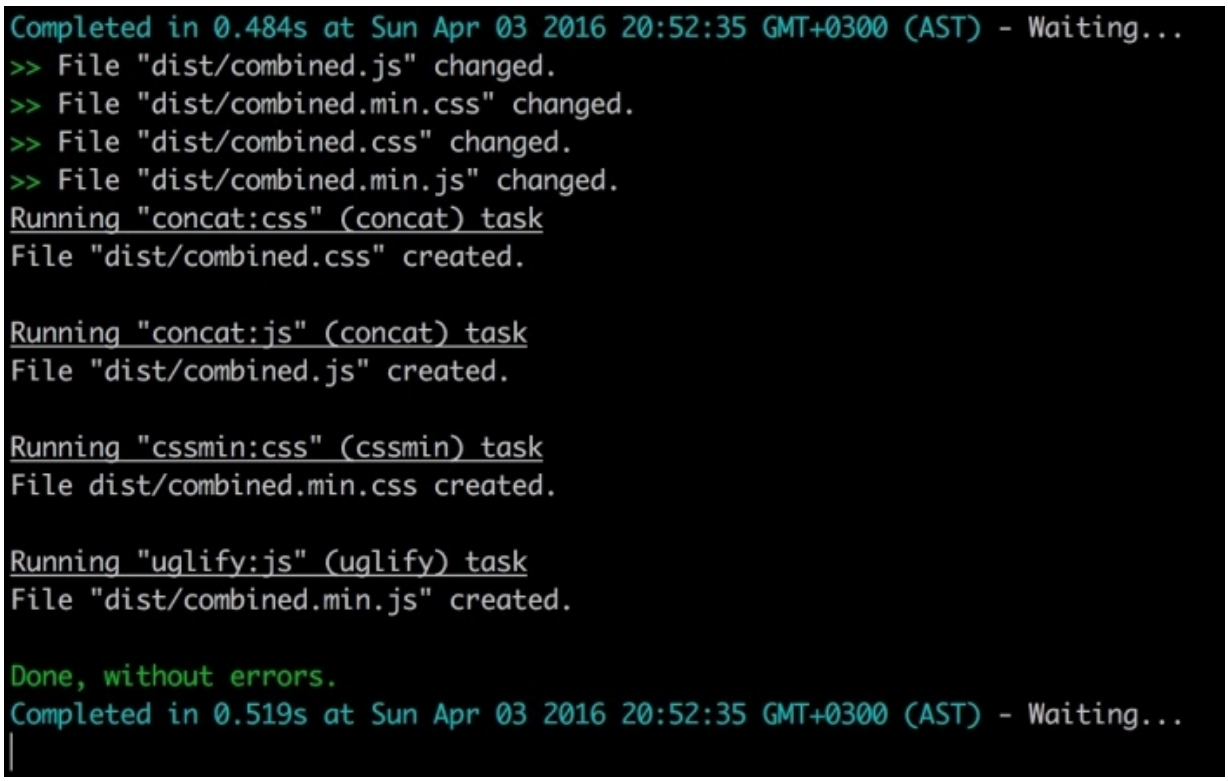
grunt.loadNpmTasks('grunt-contrib-watch');
//Include watch module
grunt.loadNpmTasks('grunt-contrib-concat');
grunt.loadNpmTasks('grunt-contrib-uglify');
grunt.loadNpmTasks('grunt-contrib-cssmin');
grunt.registerTask('default', ['concat:css',
'concat:js', 'cssmin:css', 'uglify:js']);
}; //End of module.exports
```

In preceding highlighted code, we added a `watch` block. The `mywatch` title can be any name. The `files` block is required, and it takes an array of the source paths. The Grunt watch watches for changes in these destinations and executes the tasks that are defined in the tasks block. Also, the tasks that are mentioned in the `tasks` block are already created in `GruntFile.js`. Also, we have to load the `watch` module using `grunt.loadNpmTasks`.

Now, open the terminal at the root of the project where `GruntFile.js` is located and run the following command:

```
grunt watch
```

Grunt will start watching the source files for changes. Now, modify any file in the paths defined in the `files` block in `GruntFile.js` and save the file. As soon as the file is saved, the tasks will be executed and the output for the tasks will be displayed in the terminal. A sample output can be seen in the following screenshot:



It is possible to watch as many tasks as required in the `watch` block, but these tasks should be present in `GruntFile.js`.

Summary

In this appendix, we discussed Composer and how to use it to install and update packages. Also, we discussed Git in detail, including pushing, pulling, committing, creating branches, and merging different branches. Also, we discussed Git hooks. Lastly, we discussed Grunt watch and created a watch that executed four tasks whenever any changes occurred in the files paths defined in `GruntFile.js`.

Appendix B. MVC and Frameworks

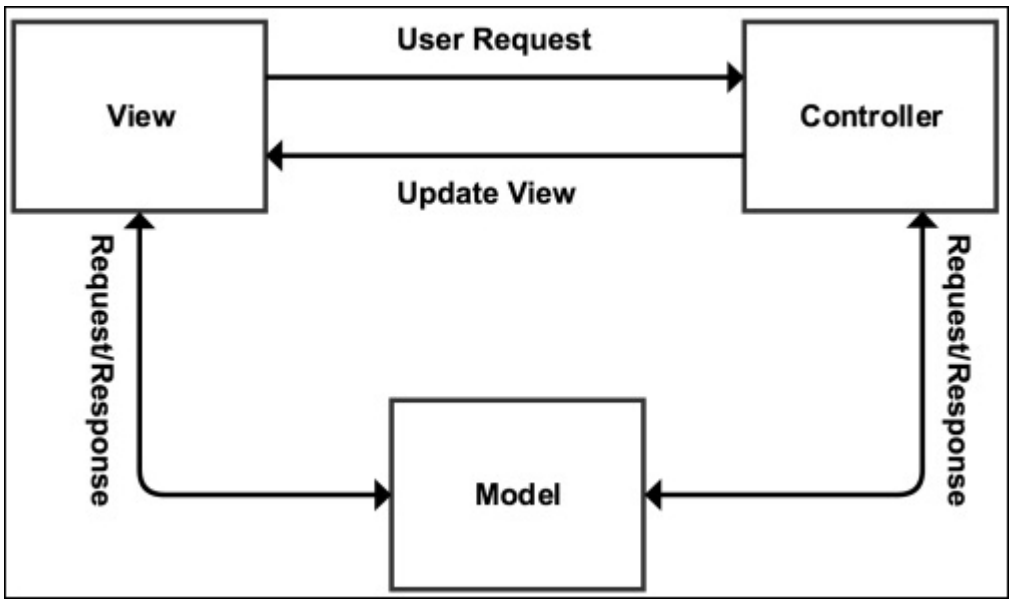
We covered the names of some of the frameworks in different chapters, but we did not discuss them. In today's world, we don't invent the wheel again; we build upon the tools that are already built, tested, and widely used. So, as best practice, if there is nothing available to fulfill the requirements, we can build it using a framework that suits the requirements best.

We will cover the following topics:

- The MVC design pattern
- Laravel
- Lumen
- Apigility

The MVC design pattern

Model View Controller (**MVC**) is a design pattern widely used in different programming languages. Most PHP frameworks use this design pattern. This pattern divides the application into three layers: Model, View, and Controller. Each one of these has separate tasks, and they are all interconnected. There are different visual representations for MVC, but an overall and simple representation can be seen in the following diagram:



Now, let's discuss each part of the MVC design pattern.

Model

The model layer is the backbone of the application and handles the data logic. Mostly, it is considered that model is responsible for CRUD operations on a database, which may or may not be true. As we mentioned previously, model is responsible for the data logic, which means that data validation operations can also be performed here. In simple words, models provide an abstraction for the data. The remaining application layers don't know or care how and from where the data comes or how an operation is performed on data. It is the model's responsibility to take care of all data logic.

In today's complex framework structures, the overall MVC structure is changed, and not only do models handle data operations, but also, every other application logic is handled by models. The method followed is fat models and slim controllers, which means keep all the application logic in models and the controllers as clean as possible.

Views

Views are what is visible to end users. All data related to this user and public is displayed in the views, so views can be called the visual representation of the models. Views need data to display. It asks for some specific data or action from the controller. Views do not know or want to know from where the controller gets this data; it just asks the controller to get it. Controller knows who to ask for this specific data and communicates with the specific model. It means that views do not have any direct link to models. However, in the earlier diagram, we linked model to view directly. This is because in the advanced systems nowadays, views can directly take data from models. For example, Magento controllers can't send data back to views. For the data (that is, to get data directly from the database) and/or to communicate with models, views communicate with blocks and helper classes. In modern practices, views can be connected to models directly.

Controllers

Controllers respond to actions performed by a user in the views and respond to the view. For example, a user fills a form and submits it. Here, the controller comes in the middle and starts taking action on the submission of the form. Now, the controller will first check whether the user is allowed to make this request or not. Then, the controller will take the appropriate action, such as communicating with the model or any other operation. In a simple analogy, the controller is the middle man between views and models. As we mentioned before in the models section, controllers should be slim. So, mostly, controllers are only used to handle the requests and communicate with models and views. All kinds of data operations are performed in models.

The MVC design pattern's sole job is to separate the responsibilities of different parts in an application. So, models are used to manage the application data. Controllers are used to take actions on user inputs, and views are responsible for the visual representation of data. As we mentioned before, MVC separates the responsibilities of each part, so it does not matter whether it accesses the model from

controllers or views; the only thing that matters is that views and controllers should not be used to perform operations on data, as it is the model's responsibility, and controllers should not be used to view any kind of data by the end user as this is the view's responsibility.

Laravel

Laravel is one of the most popular PHP frameworks, and according to the Laravel official website, it is a framework for Web Artisans. Laravel is beautiful, powerful, and has tons of features that can enable developers to write efficient and quality code. The Laravel official documentation is well written and very easy to understand. So, let's play a little with Laravel.

Installation

Installation is very easy and simple. Let's use Composer to install Laravel. We discussed Composer in Appendix A. Issue the following command in the terminal to install and create a project in Laravel:

```
composer create-project --prefer-dist
laravel/laravel packt
```

If Composer is not installed globally on the system, place `composer.phar` in a directory where Laravel should be installed and issue the following command in the terminal at the root of this directory:

```
php composer.phar create-project --prefer-dist
laravel/laravel packt
```

Now, Laravel will be downloaded, and a new project with the name `packt` will be created. Also, Composer will download and install all the dependencies for the project.

Open the browser and head to the project's URL, and we will be welcomed with a nice simple page saying **Laravel 5**.

Note

As of the writing of this book, Laravel 5.2.29 is the latest version available. However, if Composer is used, then every time the `composer update` command is used, Laravel and all other components will be automatically updated.

Features

Laravel provides tons of features, and we will only discuss a few here.

Routing

Laravel provides powerful routing. Routes can be grouped, and prefixes, namespaces, and middleware can be defined for route groups. Also, Laravel supports all HTTP methods, including `POST`, `GET`, `DELETE`, `PUT`, `OPTIONS`, and `PATCH`. All the routes are defined in the `routes.php` file in the application's `app` folder. Take a look at the following example:

```
Route::group(['prefix' => 'customer', 'namespace'
=> 'Customer', 'middleware' => 'web'], function()
{
    Route::get('/', 'CustomerController@index');
    Route::post('save',
'CustomerController@save');
    Route::delete('delete/{id}',
'CustomerController@delete');
});
```

In the preceding snippet, we created a new routes group. This will be only used when the URL has a prefixed customer. For example, if a URL is similar to `domain.com/customer`, this group will be used. We also used a customer namespace. Namespacing allows us to use standard PHP namespaces and divide our files in subfolders. In the preceding example, all customer controllers can be placed in the Customer subfolder in the `Controllers` directory, and the controller will be created as follows:

```
namespace App\Http\Controllers\Customer

use App\Http\{
Controllers\Controller,
Requests,
};
use Illuminate\Http\Request;

Class CustomerController extends Controller
{
  …
  …
}
```

So, namespacing a route group enables us to place our controller files in subfolders, which are easy to manage. Also, we used the web middleware. Middleware provides a way to filter the request before entering the application, which enables us to use it to check whether a user is logged in or not, the CSRF protection, or whether there are any other actions that can be performed in a middleware and need to be performed before the request is sent to application. Laravel comes with a few middleware, including `web`, `api`, `auth`, and so on.

If a route is defined as `GET`, no `POST` request can be sent to this route. It is very convenient, which enables us to not worry about the request method filtering. However, HTML forms do not support the HTTP methods like `DELETE`, `PATCH`, and `PUT`. For this, Laravel provides method spoofing, in which a hidden form field with `name _method` and the value of the HTTP method is used to make this request possible. For example, in our routes group, to make the request possible to delete a route, we need a form similar to the following:

```
<form action="/customer/delete" method="post">
  {{ method_field('DELETE') }}
  {{ csrf_field() }}
</form>
```

When the preceding form is submitted, it will work, and the delete route will be used. Also, we created a CSRF hidden field, which is used for CSRF protection.

> **Note**
>
> Laravel routing is very interesting, and it is a big topic. More in-depth detail can be found at https://laravel.com/docs/5.2/routing.

Eloquent ORM

Eloquent ORM provides active records to interact with the database. To use Eloquent ORM, we have to just extend our models from the Eloquent model. Let's have a look at a simple user model, as follows:

```
namespace App;

use Illuminate\Database\Eloquent\Model;

class user extends Model
{
  //protected $table = 'customer';
  //protected $primaryKey = 'id_customer';
  …
  …
}
```

That's it; we have a model that can handle all the CRUD operations now. Note that we commented the `$table property` and did the same for `$primaryKey`. This is because Laravel uses a plural name of the class to look for the table unless the table is defined with the protected `$table property`. In our case, Laravel will look for table name users and use it. However, if we want to use a table named `customers`, we can just uncomment the line, as follows:

```
protected $table = 'customers';
```

Similarly, Laravel thinks that a table will have a primary key with the column name `id`. However, if another column is needed, we can

override the default primary key, as follows:

```
protected $primaryKey = 'id_customer';
```

Eloquent models also make it easy for timestamps. By default, if the table has the `created_at` and `updated_at` fields, then these two dates will be generated automatically and saved. If no timestamps are required, these can be disabled, as follows:

```
protected $timestamps = false;
```

Saving data to the table is easy. The table columns are used as properties of the models, so if our `customer` table has columns such as `name`, `email`, `phone`, and so on, we can set them as follows in our `customer` controller, mentioned in the routing section:

```
namespace App\Http\Controllers\Customer

use App\Http\{
Controllers\Controller,
Requests,
};
use Illuminate\Http\Request;
use App\Customer

Class CustomerController extends Controller
{
  public function save(Request $request)
  {
    $customer = new Customer();
    $customer->name = $request->name;
    $customer->email = $request->email;
    $customer->phone = $request->phone;

    $customer->save();

  }
}
```

In the preceding example, we added the `save` action to our controller. Now, if a `POST` or `GET` request is made along the form data, Laravel

assigns all the form-submitted data to a Request object as properties with the same names as that of the form fields. Then, using this request object, we can access all the data submitted by the form either using `POST` or `GET`. After assigning all the data to model properties (the same names as those of table columns), we can just call the save method. Now, our model does not have any save method, but its parent class, which is the Eloquent model, has this method defined. However, we can override this `save` method in our `model` class in case we need some other features in this method.

Fetching data from the Eloquent model is also easy. Let's try an example. Add a new action to the `customer` controller, as follows:

```
public function index()
{
  $customers = Customer::all();
}
```

We used the `all()` static method in the model, which is basically defined in the Eloquent model, which, in turn, fetches all the data in our `customers` table. Now, if we want to get a single customer by the primary key, we can use the `find($id)` method, as follows:

```
$customer = Customer::find(3);
```

This will fetch the customer with the ID `3`.

Updating is simple, and the same `save()` method is used, as shown here:

```
$customer = Customer::find(3);
$customer->name = 'Altaf Hussain';

$customer->save();
```

This will update the customer with the ID `3`. First, we loaded the `customer`, then we assigned new data to its properties, and then we

called the same `save()` method. Deleting the model is simple and easy and can be done as follows:

```
$customer = Customer::find(3);
$customer->delete();
```

We first loaded the customer with the ID `3`, and then we called the `delete` method, which will delete the customer with the ID `3`.

> **Note**
>
> Laravel's Eloquent models are very powerful and provide lots of features. These are well explained in the documentation at https://laravel.com/docs/5.2/eloquent. The Laravel database section is also worth reading and can be found at https://laravel.com/docs/5.2/database.

Artisan CLI

Artisan is the command-line interface provided with Laravel, and it has some nice commands that can be used for quicker operations. It has lots of commands, and a full list can be seen using the following command:

```
php artisan list
```

This will list all the options and commands available.

> **Note**
>
> The `php artisan` command should be run in the same directory in which the `artisan` file is located. It is placed at the root of the project.

Some of the basic commands are as follows:

- `make:controller`: This command creates a new controller in the `Controllers` folder. The command can be used as follows:

```
php artisan make:controller MyController
```

  If a namespaced controller is required, as it happened before with the `Customer` namespace, it can be done as follows:

```
php artisan make:controller
Customer/CustomerController
```

  This command will create `CustomerController` in the `Customer` folder. If the `Customer` folder is not available, it will create the folder as well.

- `make:model`: This creates a new model in the `app` folder. The syntax is the same as the `make:controller` command, as follows:

```
php artisan make:model Customer
```

  For the namespaced models, it can be used as follows:

```
php artisan make:model Customer/Customer
```

  This will create the `Customer` model in the `Customer` folder and use the `Customer` namespace for it.

- `make:event`: This creates a new `event` class in the `Events` folder. It can be used as follows:

```
php artisan make:event MyEvent
```

- `make:listener`: This command creates a new listener for an event. This can be used as follows:

```
php artisan make:listener MyListener --event
MyEvent
```

The preceding command will create a new listener for our `MyEvent` event. We have to always mention the event for which we need to create a listener using the `--event` option.

- `make:migration`: This command creates a new migration in the database/migrations folder.
- `php artisan migrate`: This runs all the available migrations that are not executed.
- `php artisan optimize`: This command optimizes the framework for better performance.
- `php artisan down`: This puts the application in maintenance mode.
- `php artisan up`: This command brings the application back live from the maintenance mode.
- `php artisan cache:clear`: This command clears the application cache.
- `php artisan db:seed`: This command seeds the database with records.
- `php artisan view:clear`: This clears all the compiled view files.

> **Note**
>
> More detail about the Artisan console or Artisan CLI can be found in the documentation at https://laravel.com/docs/5.2/homestead.

Migrations

Migrations is another powerful feature in Laravel. In migrations, we define the database schemas—whether it creates tables, removes tables, or adds/updates columns in the tables. Migrations are very

convenient in deployment and act as version control for the database. Let's create a migration for our customer table that is not available in the database yet. To create a migration, issue the following command in the terminal:

```
php artisan make:migration create_custmer_table
```

A new file in the `database/migrations` folder will be created with the filename `create_customer_table` prefixed with the current date and a unique ID. The class is created as `CreateCustomerTable`. This is a class as follows:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateCustomerTable extends Migrations
{
  //Run the migrations

  public function up()
  {
    //schemas defined here
  }

  public function down()
  {
    //Reverse migrations
  }
}
```

The class will have two public methods: `up()` and `down()`. The `up()` method should have all the new schemas for the table(s). The `down()` method is responsible for reversing the executed migration. Now, lets add the `customers` table schema to the `up()` method, as follows:

```
public function up()
{
  Schema::create('customers', function (Blueprint
$table)
  {
    $table->increments('id', 11);
```

```
            $table->string('name', 250)
            $table->string('email', 50);
            $table->string('phone', 20);
            $table->timestamps();
        });
    }
    public function down()
    {
        Schema::drop('customers');
    }
```

In the `up()` method, we defined the schema and table name. Columns for the table are individually defined, including the column size. The `increments()` method defines the autoincrement column, which, in our case, is the `id` column. Next, we created three string columns for `name`, `email`, and `phone`. Then, we used the `timestamps()` method, which creates the `created_at` and `updated_at` timestamp columns. In the `down()` method, we just used the `drop()` method of the `Schema` class to drop out the `customers` table. Now, we need to run our migrations using the following command:

```
php artisan migrate
```

The preceding command will not only run our migration but will also run all those migrations that are not executed yet. When a migration is executed, Laravel stores the migration name in a table called `migrations`, from where Laravel decides which migrations it has to execute and which to skip.

Now, if we need to roll back the latest executed migration, we can use the following command:

```
php artisan migrate:rollback
```

This will roll back to the last batch of migrations. To roll back all the migrations of the application, we can use the reset command, as follows:

```
php artisan migrate:reset
```

This will roll back the complete application migrations.

Migrations make it easy for deployment because we won't need to upload the database schemas every time we create some new changes in the tables or database. We will just create the migrations and upload all the files, and after this, we will just execute the migration command, and all the schemas will be updated.

Blade templates

Laravel comes with its own template language called Blade. Also, Blade template files support plain PHP code. Blade template files are compiled to plain PHP files and are cached until they are changed. Blade also supports layouts. For example, the following is our master page layout in Blade, placed in the `resources/views/layout` folder with the name `master.blade.php`. Take a look at the following code:

```
<!DOCTYPE html>
<html>
  <head>
    <title>@yield('title')</title>
  </head>
  <body>
    @section('sidebar')
      Our main sidebar
      @show

      <div class="contents">
        @yield('content')
      </div>
  </body>
</html>
```

In the preceding example, we had a section for the sidebar that defines a `content` section. Also, we had `@yield`, which displays the contents of a section. Now, if we want to use this layout, we will need to extend it in the child template files. Let's create the `customers.blade.php` file in the `resources/views/` folder and place the following code in it:

```
@extend('layouts.master')
  @section('title', 'All Customers')
  @section('sidebar')
  This will be our side bar contents
  @endsection
  @section('contents')
    These will be our main contents of the page
  @endsection
```

As can be seen in the preceding code, we extended the `master` layout and then placed contents in every section of the `master` layout. Also, it is possible to include different templates in another template. For example, let's have two files, `sidebar.blade.php` and `menu.blade.php`, in the `resources/views/includes` folder. Then, we can include these files in any template, as follows:

```
@include(includes.menu)
@include(includes.sidebar)
```

We used `@include` to include a template. The dot (`.`) indicates a folder separation. We can easily send data to Blade templates or views from our controllers or routers. We have to just pass the data as an array to a view, as follows:

```
return view('customers', ['count => 5]);
```

Now, `count` is available in our `customers` view file and can be accessed as follows:

```
Total Number of Customers: {{ count }}
```

Yes, Blade uses double curly braces to echo a variable. For control structures and loops, let's have another example. Let's send data to the `customers` view, as follows:

```
return view('customers', ['customers' =>
$allCustomers]);
```

Now, our `customers` view file will be similar to the following if we want to display all the `customers` data:

```
…
…
@if (count($customers) > 0)
{{ count($customers) }} found. <br />
@foreach ($customers as $customer)
{{ $customer->name }} {{ $customer->email }} {{
$customer->phone }} <br>
@endforeach

@else
Now customers found.
@endif;
…
…
```

All the preceding syntax looks familiar as it is almost the same as plain PHP. However, to display a variable, we have to use double curly braces `{{}}`.

> **Note**
>
> A nice and easy-to-read documentation for Blade templates can be found at https://laravel.com/docs/5.2/blade.

Other features

We only discussed a few basic features in the previous section. Laravel has tons of other features, such as Authentication and Authorization, which provide an easy way to authenticate and authorize users. Also, Laravel provides a powerful caching system, which supports file-based cache, the Memcached, and Redis cache. Laravel also provides events and listeners for these events, which is very convenient when we want to perform a specific action and when a specific event occurs. Laravel supports localization, which enables

us to use localized contents and multiple languages. Laravel also supports task scheduling and queues, in which we schedule some tasks to run at a specific time and queue some tasks to be run when their turn arrives.

Lumen

Lumen is a micro-framework provided by Laravel. Lumen is mainly intended to create stateless APIs and has a minimal set of features of Laravel. Also, Lumen is compatible with Laravel, which means that if we just copy our Lumen application to Laravel, it will work fine. The installation is simple. Just use the following Composer command to create a Lumen project, and it will download all the dependencies, including Lumen:

```
composer create-project --prefer-dist
laravel/lumen api
```

The preceding command will download Lumen and then create our API application. After this, rename `.env.example` as `.env`. Also, create a 32-characters-long app key and place it in the `.env` file. Now, the basic application is ready to use and create APIs.

> **Note**
>
> Lumen is almost the same as Laravel, but some Laravel features are not included by default. More details can be found at https://lumen.laravel.com/docs/5.2.

Apigility

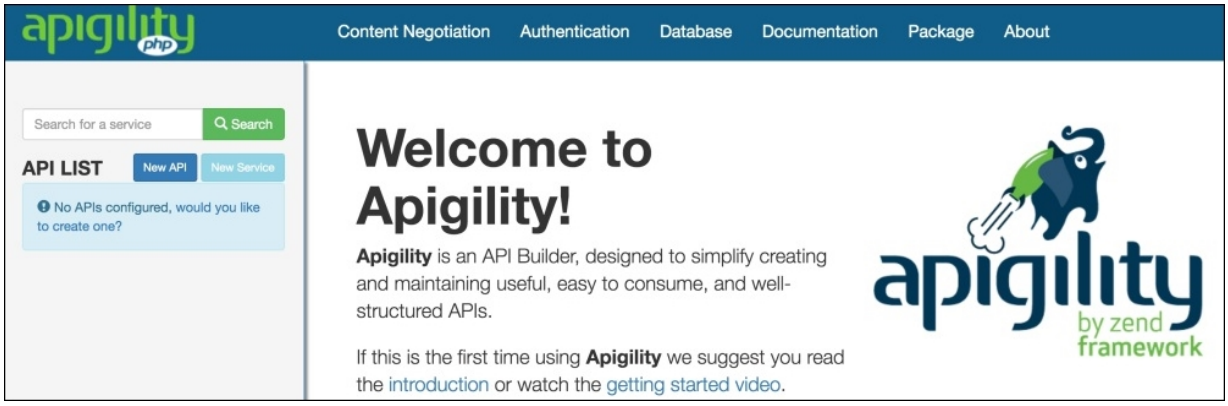
Apigility is built and developed by Zend in Zend Framework 2. Apigility provides an easy to use GUI to create and manage APIs. It is very easy to use and is capable of creating complex APIs. Let's start by installing Apigility using Composer. Issue the following command in the terminal:

```
composer create-project -sdev zfcampus/zf-apigility-skeleton packt
```

The preceding command will download Apigility and its dependencies, including Zend Framework 2, and will set up our project named `packt`. Now, issue the following command to enable the development mode so that we can have access to the GUI:

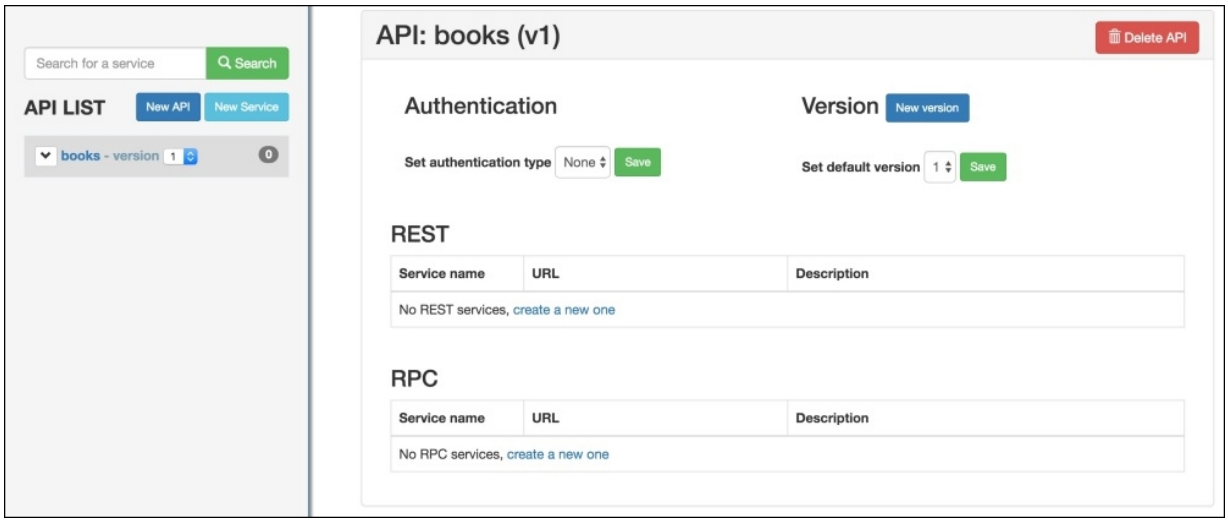
```
php public/index.php development enable
```

Now, open the URL as yourdomain.com/packt/public, and we will see a beautiful GUI, as shown in the following screenshot:

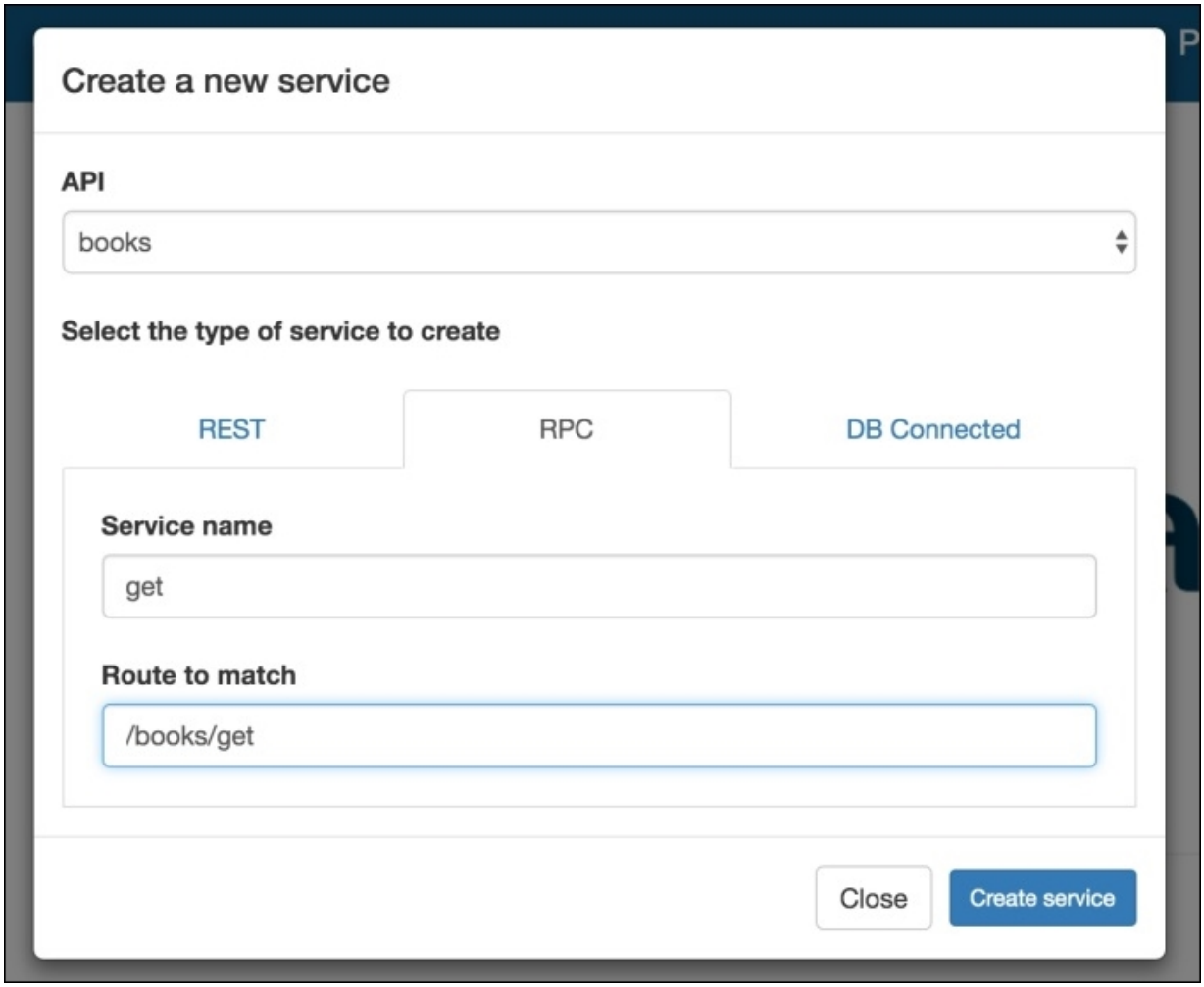


Now, let's create our first API. We will call this API "`books`", which will return a list of books. Click on the **New API** button, as shown in the preceding picture, and a popup will be displayed. In the text box, enter `books` as the API name and click on `Create` button; the new API

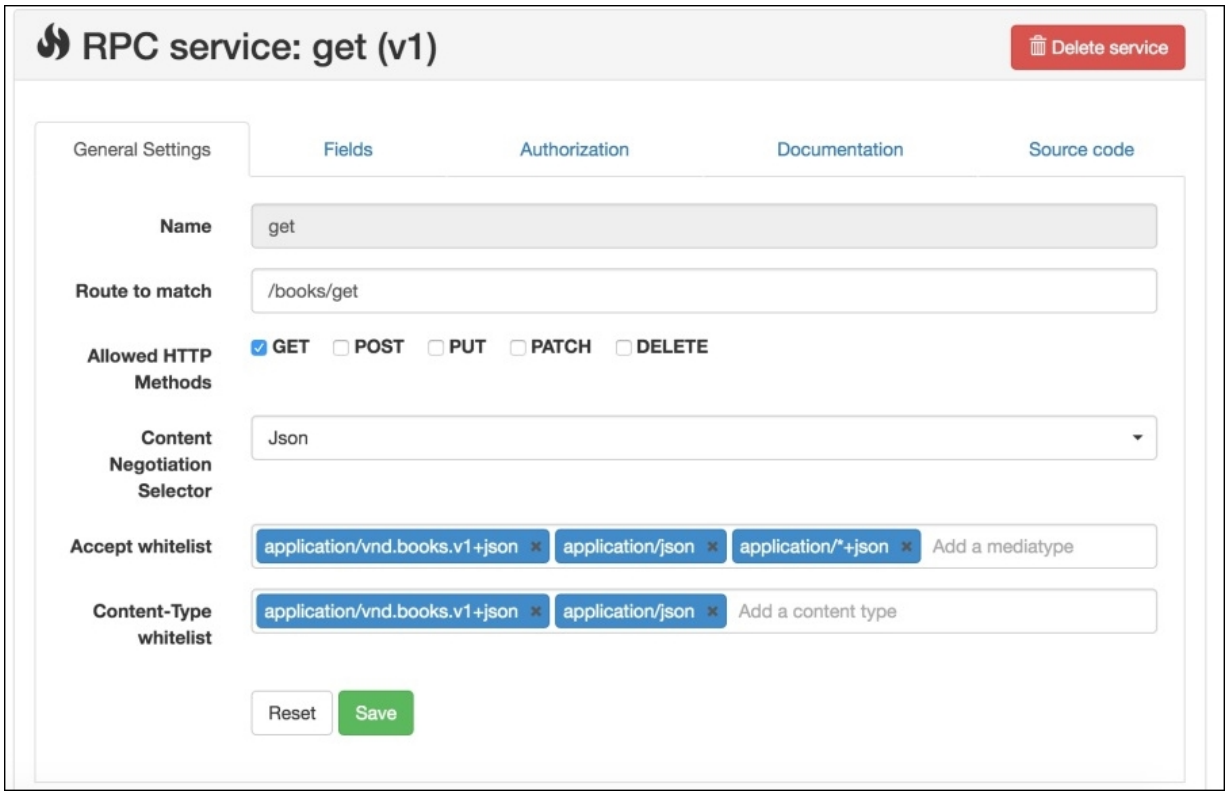
will be created. When the API is created, we will be presented with the following screen:



Apigility provides easy ways to set other properties for the API, such as versioning and authentication. Now, let's create an RPC service by clicking on the **New Service** button in the left sidebar. Also, we can click on the **Create a new one** link in the **RPC** section in the preceding screenshot. We will be presented with the following screen:

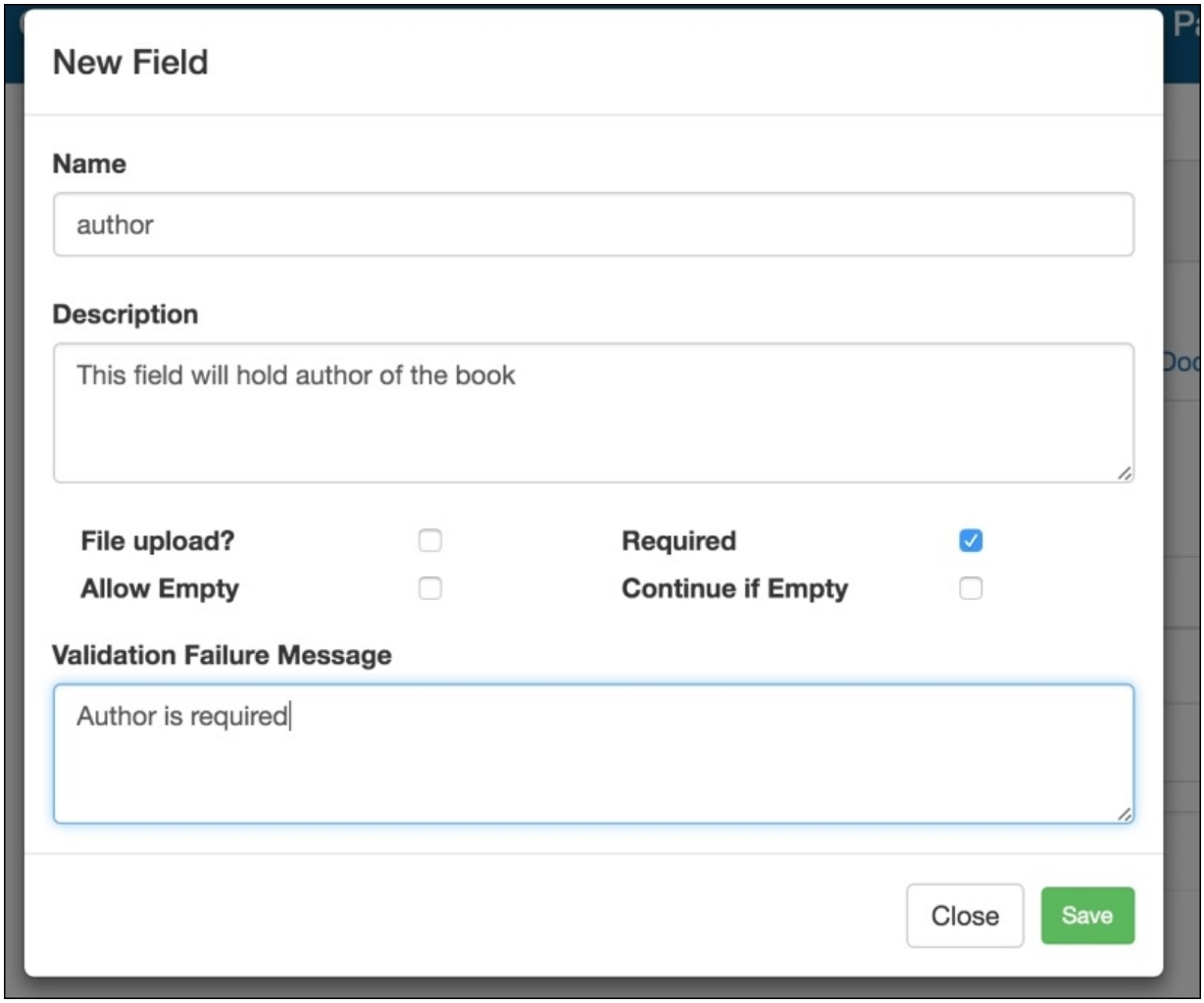


As shown in the preceding screenshot, we created an RPC service named `get` in the `books` API. The route URI entered is `/books/get`, which will be used to call this RPC service. When we click on the `Create service` button, the API creation successful message will be displayed, and also, the following screen will be displayed:

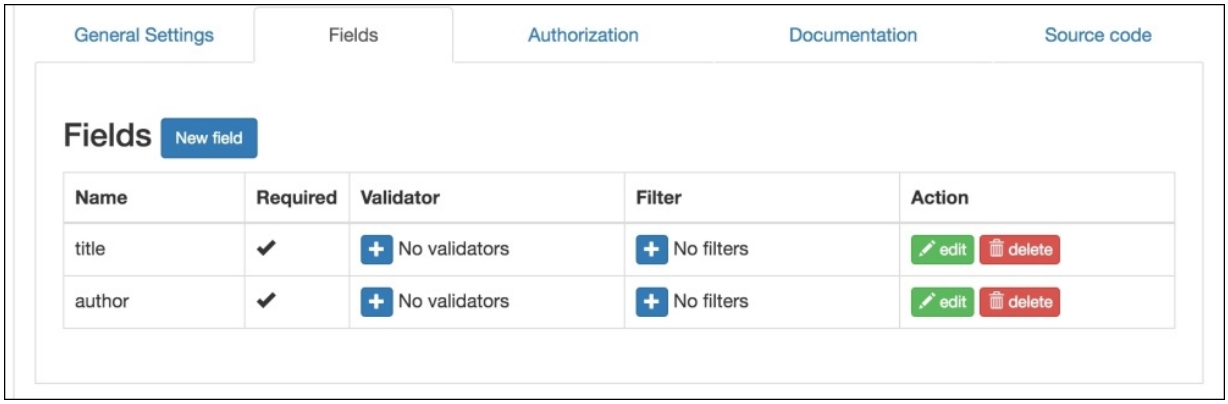


As can be seen in the preceding screenshot, the allowed HTTP method for this service is only **GET**. Let's keep this as it is, but we can select all or any of them. Also, we want to keep **Content Negotiation Selector** as `Json`, and our service will accept/receive all the content in the JSON format. Also, we can select different media types and content types.

Next, we should add some fields to our service that will be used. Click on the **Fields** tab, and we will see the **Fields** screen. Click on the **New Field** button, and we will be presented with the following popup:



As can be seen in the preceding screenshot, we can set all the properties for a field, such as the **Name**, **Description**, whether it is required or not, and some other settings, including an error message if the validation fails. After we created two fields, **title** and **author**, we will have a screen similar to the following:

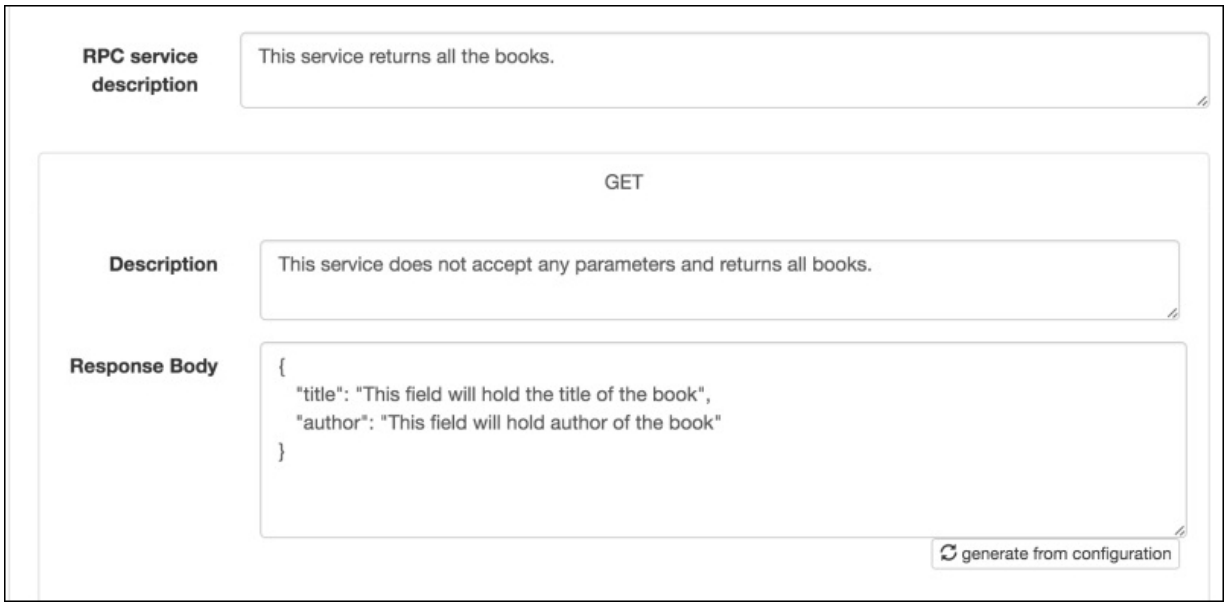


As can be seen in the preceding screen, we can add validators and filters to each individual field too.

Note

As this is just an introductory topic for Apigility, we will not cover validators and filters and some other topics in this book.

The next topic is documentation. When we click on the **Documentation** tab, we will see the following screen:



Here, we will document our service, add some description, and also can generate the response body for documentation purposes. This is very important as it will enable others to better understand our APIs and services.

Now, we need to get the all the books from somewhere. It can be either from the database or from another service or any other source. However, for now, we will just use an array of books for test purposes. If we click on the **Source** tab, we will find that our code for the service is placed at `module/books/src/books/V1/Rpc/Get/GetController.php`. Apigility

created a module for our API `books` and then placed all the source code in this module in different folders according to the version of our API, which is V1 by default. We can add more versions, such as V2 and V3, to our APIs. Now, if we open the `GetController` file, we will find some code and an action called `getAction` according to our route URI. The code is as follows, and the highlighted code is the one we added:

```
namespace books\V1\Rpc\Get;

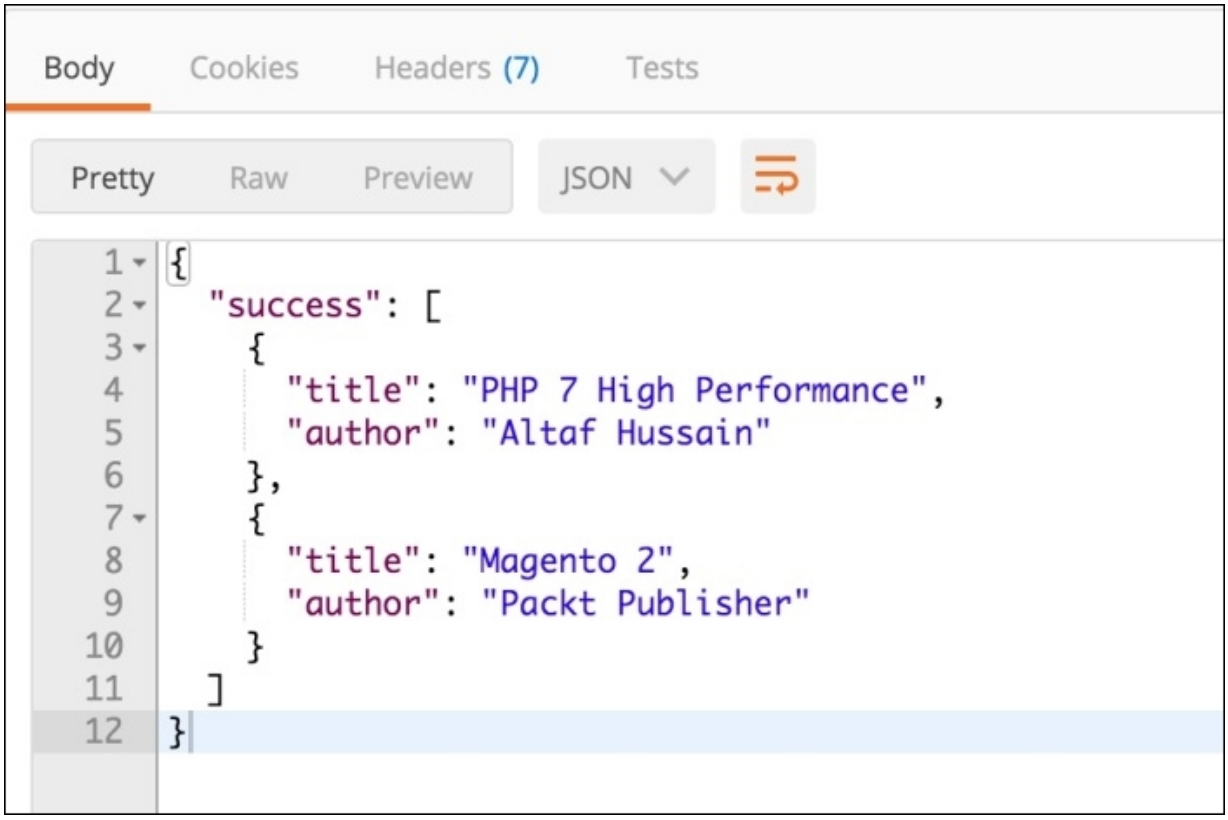
use Zend\Mvc\Controller\AbstractActionController;
use ZF\ContentNegotiation\ViewModel;


class GetController extends
AbstractActionController
{
  public function getAction()
  {
    $books = [ 'success' => [
    [
      'title' => 'PHP 7 High Performance',
      'author' => 'Altaf Hussain'
    ],
    [
      'title' => 'Magento 2',
      'author' => 'Packt Publisher'
    ],
    ]
    ];

    return new ViewModel($books);
  }
}
```

In the preceding code, we used `ContentNegotiation\ViewModel`, which is responsible for responding with the data in the format that we selected in the service setup, which is JSON in our case. Then, we created a simple `$books` array with the fieldnames we created for the service and assigned our values to them. Then, we returned them using the `ViewModel` object, which handles the response data conversion to JSON.

Now, let's test our API. As our service can accept `GET` requests, we will just type our URL in the browser with the `books/get` URI, and we will see the JSON response. It is best to check the API with tools such as RestClient or Postman for Google Chrome, which provides an easy-to-use interface to make different types of requests to APIs. We tested it with Postman and got the response shown in the following screenshot:



Also note that we set our service to accept only `GET` requests. So, if we send a request other than `GET`, we will get an `HTTP Status code 405 methods not allowed` error.

Apigility is very powerful and provides tons of features, such as RESTFul APIs, HTTP authentication, database connected services with easy-to-create DB connectors, and a selection of tables for a service. While using Apigility, we need not worry about the API, service structure security, and other things, as Apigility does this for us. We need to only concentrate on the business logic of the APIs and services.

Note

Apigility can't be covered completely in this Appendix. Apigility has lots of features that can be covered in a complete book. Apigility's official documentation at https://apigility.org/documentation is a good place to get started and read more about this.

Summary

In this Appendix, we discussed the basics of the MVC design pattern. We also discussed the Laravel framework and some of its good features. We introduced you to the Laravel-based micro-framework, Lumen. At the end, we had a small introduction to Apigility and created a test API and web service.

In IT, things get obsolete in a very short time span. It is always required to study upgraded tools and find new ways and techniques for the best approaches in programming. Therefore, one should not stop after completing this book and start studying new topics and also the topics that are not covered completely in this book. Until this point, you will have the knowledge that you can use to set up high-performance environments for high-performance applications. We wish you good luck and success in PHP-ing!

Part 3. Module 3

Modernizing Legacy Applications in PHP

Get your code under control in a series of small, specific steps

Chapter 1. Legacy Applications

In its simplest definition, a legacy application is any application that you, as a developer, inherit from someone else. It was written before you arrived, and you had little or no decision-making authority in how it was built.

However, there is a lot more weight to the word legacy among developers. It carries with it connotations of poorly organized, difficult to maintain and improve, hard to understand, untested or untestable, and a series of similar negatives. The application works as a product in that it provides revenue, but as a program, it is brittle and sensitive to change.

Because this is a book specifically about PHP-based legacy applications, I am going to offer some PHP-specific characteristics that I have seen in the field. For our purposes, a legacy application in PHP is one that matches two or more of the following descriptions:

- It uses page scripts placed directly in the document root of the web server.
- It has special index files in some directories to prevent access to those directories.
- It has special logic at the top of some files to `die()` or `exit()` if a certain value is not set.
- Its architecture is include-oriented instead of class-oriented or object-oriented.
- It has relatively few classes.
- Any class structure that exists is disorganized, disjointed, and otherwise inconsistent.
- It relies more heavily on functions than on class methods.
- Its page scripts, classes, and functions combine the concerns of model, view, and controller into the same scope.
- It shows evidence of one or more incomplete attempts at a rewrite, sometimes as a failed framework integration.
- It has no automated test suite for the developers to run.

These characteristics are probably familiar to anyone who has had to deal with a very old PHP application. They describe what I call a typical PHP application.

The typical PHP application

Most PHP developers are not formally trained as programmers, or are almost entirely self-taught. They often come to the language from other, usually non-technical, professions. Somehow or another, they are tasked with the duty of creating webpages because they are seen as the most technically-savvy person in their organization. Since PHP is such a forgiving language and grants a lot of power without a lot of discipline, it is very easy to produce working web pages and even applications without a lot of training.

These and other factors strongly influence the underlying foundation of the typical PHP application. They are usually not written in a popular full-stack framework or even a micro-framework. Instead, they are often a series of page scripts, placed directly in the web server document root, to which clients can browse directly. Any functionality that needs to be reused has been collected into a series of `include` files. There are `include` files for common configurations and settings, headers and footers, common forms and content, function definitions, navigation, and so on.

This reliance on `include` files in the typical PHP application is what makes me call them include-oriented architectures. The legacy application uses `include` calls everywhere to couple the pieces of the program into a single whole. This is in contrast to a class-oriented architecture, where even if the application does not adhere to good object-oriented programming principles, at least the behaviors are bundled into classes.

File Structure

The typical include-oriented PHP application generally looks something like this:

```
/path/to/docroot/
bin/                          # command-line tools
cache/                      # cache files
common/                   # commonly-used include
files
classes/                    # custom classes
Image.php                 #
Template.php            #
functions/                  # custom functions
db.php                      #
log.php                     #
cache.php                 #
setup.php                  # configuration and setup
css/                        # stylesheets
img/                       # images
index.php                # home page script
js/                         # JavaScript
lib/                       # third-party libraries
log/                      # log files
page1.php              # other page scripts
page2.php              #
page3.php              #
sql/                       # schema migrations
sub/                     # sub-page scripts
index.php               #
subpage1.php #
subpage2.php #
theme/                   # site theme files
header.php            # a header template
footer.php              # a footer template
nav.php                 # a navigation template ~~
```

The structure shown is a simplified example. There are many possible variations. In some legacy applications, I have seen literally hundreds of main-level page scripts and dozens of subdirectories with their own unique hierarchies for additional pages. The key is that the legacy application is usually in the document root, has page scripts that users browse to directly, and uses `include` files to manage most program behavior instead of classes and objects.

Page Scripts

Legacy applications will use individual page scripts as the access point for public behavior. Each page script is responsible for setting up the global environment, performing the requested logic, and then delivering output to the client.

Appendix A, *Typical Legacy Page Script* contains a sanitized, anonymized version of a typical legacy page script from a real application. I have taken the liberty of making the indentation consistent (originally, the indents were somewhat random) and wrapping it at 60 characters so it fits better on e-reader screens. Go take a look at it now, but be careful. I won't be held liable if you go blind or experience post-traumatic stress as a result! As we examine it, we find all manner of issues that make maintenance and improvement difficult:

- The `include` statements to execute setup and presentation logic
- inline function definitions
- global variables
- model, view, and controller logic all combined in a single script
- trusting user input
- possible SQL injection vulnerabilities
- possible cross-site scripting vulnerabilities
- unquoted array keys generating notices
- The `if` blocks not wrapped in braces (adding a line in the block later will not actually be part of the block)
- copy-and-paste repetition

The Appendix A, *Typical Legacy Page Script* example is relatively tame as far as legacy page scripts go. I have seen other scripts where JavaScript and CSS code have been mixed in, along with remote-file inclusions and all sorts of security flaws. It is also only (`!`) about 400 lines long. I have seen page scripts that are thousands of lines long which generate several different page variations, all wrapped into a single `switch` statement with a dozen or more `case` conditions.

Rewrite or Refactor?

Many developers, when presented with a typical PHP application, are able to live with it for only so long before they want to scrap it and rewrite it from scratch. Nuke it from orbit; it's the only way to be sure! is the rallying cry of these enthusiastic and energetic programmers. Other developers, their enthusiasm drained by their death march experience, feel cautious and wary at such a suggestion. They are fully aware that the codebase is bad, but the devil (or in our case, code) they know is better than the devil they don't.

The Pros and Cons of Rewriting

A complete rewrite is a very tempting idea. Developers championing a rewrite feel like they will be able to do all the right things the first time through. They will be able to write unit tests, enforce best practices, separate concerns according to modern pattern definitions, and use the latest framework or even write their own framework (since they know best what their own needs are). Because the existing application can serve as a reference implementation, they feel confident that there will be little or no trial-and-error work in rewriting the application. The needed behaviors already exist; all the developers need to do is copy them to the new system. The behaviors that are difficult or impossible to implement in the existing system can be added on from the start as part of the rewrite.

As tempting as a rewrite sounds, it is fraught with many dangers. Joel Spolsky had this to say regarding the old Netscape Navigator web browser rewrite in 2000:

> *Netscape made the single worst strategic mistake that any software company can make by deciding to rewrite their code from scratch. Lou Montulli, one of the 5 programming superstars*

> *who did the original version of Navigator, emailed me to say, I agree completely, it's one of the major reasons I resigned from Netscape. This one decision cost Netscape 3 years. That's three years in which the company couldn't add new features, couldn't respond to the competitive threads from Internet Explorer, and had to sit on their hands while Microsoft completely ate their lunch.*
>
> *--Joel Spolsky, Netscape Goes Bonkers*

Netscape went out of business as a result.

Josh Kerr relates a similar story regarding TextMate:

> *Macromates, an indie company who had a very successful text editor called Textmate, decided to rewrite the code base for Textmate 2. It took them 6 years to get a beta release out the door which is an eternity in today's time and they lost a lot of market share. When they did release a beta, it was too late and 6 months later they folded the project and pushed it on to Github as an open source project.*
>
> *--Josh Kerr, TextMate 2 And Why You Shouldn't Rewrite Your Code*

Fred Brooks calls the urge to do a complete rewrite the second-system effect. He wrote about this in 1975:

> *The second is the most dangerous system a man ever designs. ... The general tendency is to over-design the second system, using all the ideas and frills that were cautiously sidetracked on the first one. ... The second-system effect has ... a tendency to refine techniques whose very existence has been made obsolete by changes in basic system assumptions. ... How does the project*

> *manager avoid the second-system effect? By insisting on a senior architect who has at least two systems under his belt.*
>
> --*Fred Brooks, The Mythical Man-Month*, pp. 53-58.

Developers were the same forty years ago as they are today. I expect them to be the same over the next forty years as well; human beings remain human beings. Overconfidence, insufficient pessimism, ignorance of history, and the desire to be one's own customer all lead developers easily into rationalizations that this time will be different when they attempt a rewrite.

Why Don't Rewrites Work?

There are lots of reasons why a rewrite rarely works, but I will concentrate on only one general reason here: the intersection of resources, knowledge, communication, and productivity. (Be sure to read *The Mythical Man-Month* (pp. 13-26) for a great description of the problems associated with thinking of resources and scheduling as interchangeable elements.)

As with all things, we have only limited resources to bring to bear against the rewrite project. There are only a certain number of developers in the organization. These are the developers who will have to do *both* maintenance on the existing program *and* write the completely new version of the program. Any developers working on the one project will not be able to work on the other.

The Context-switching problem

One idea is to have the existing developers spend part of their time on the old application and part of their time on the new one. However, moving a developer between the two projects will not be an even split of productivity. Because of the cognitive load of context-switching, the developer will be less than half as productive on each.

The Knowledge problem

To avoid the productivity losses from switching developers between maintenance and the rewrite, the organization may try to hire more developers. Some can then be dedicated to the old project and others to the new project. Unfortunately, this approach reveals what F. A. Hayek calls the knowledge problem. Originally applied to the realm of economics, the knowledge problem applies equally as well to programming.

If we put the new developers on the rewrite project, they won't know enough about the existing system, the existing problems, the business goals, and perhaps not even the best practices for doing the rewrite to be effective. They will have to be trained on these things, most likely by the existing developers. This means the existing developers, who have been relegated to maintaining the existing program, will have to spend a lot of time communicating knowledge to the new hires. The amount of time involved is non-trivial, and the communication of this knowledge will have to continue until the new developers are as well-versed as the existing developers. This means that the linear increase in resources results in a less-than-linear increase in productivity: a 100% increase in the number of programmers will result in a less than 50% increase in output, sometimes much less (cf. *The Miserable Mathematics of the Man-Month* – http://paul-m-jones.com/archives/1591).

Alternatively, we could put the existing developers on the rewrite project, and the new hires on maintenance of the existing program. This too reveals a knowledge problem because the new developers are completely unfamiliar with the system. Where will they get the knowledge they need to do their work? From the existing developers, of course, who will still need to spend valuable time communicating their knowledge to the new hires. Once again, we see that the linear increase in developers leads to a less-than-linear increase in productivity.

The Schedule Problem

To deal with the knowledge problem and the related communication costs, some may feel the best way to handle the project would be to dedicate all the existing developers on the rewrite, and delay maintenance and upgrades on the existing system until the rewrite is done. This is a great temptation because the developers will be all too eager to salve their own pains and become their own customers - becoming excited about what features they want to have and what fixes they want to make. These desires will lead them to overestimate their own ability to perform a full rewrite and underestimate the amount of time needed to complete it. The managers, for their part, will accept the optimism of the developers, perhaps adding some buffer in the schedule for good measure.

The overconfidence and optimism of the developers will morph into frustration and pain when they realize the task is actually much greater and more overwhelming than they first thought. The rewrite will go on much longer than anticipated, not by a little, but by an order of magnitude or more. For the duration of the rewrite, the existing program will languish - buggy and missing features - disappointing existing customers and failing to attract new ones. The rewrite project will, at the end, become a panicked death march to get it done at all costs, and the result will be a codebase that is just as bad as the first one, only in different ways. It will be merely a copy of the first system, because schedule pressures will have dictated that new features be delayed until after an initial release is achieved.

Iterative Refactoring

Given the risks associated with a complete rewrite, I recommend refactoring instead. Refactoring means that the *quality* of the program is improved in small steps, without changing the *functionality* of the program. A single, relatively small change is introduced across the entire system. The system is then tested to make sure it still works properly, and finally, the system is put into production. A second small change builds on the previous one, and so on. Over a period of time, the system becomes markedly easier to maintain and improve.

A refactoring approach is decidedly less appealing than a complete rewrite. It defies the core sensibilities of most developers. The developers have to continue working with the system as it is, warts and all, for long periods of time. They do not get to switch over to the latest, hottest framework. They do not get to become their own customers and indulge their desires to do things right the first time. Being a longer-term strategy, the refactoring approach does not appeal to a culture that values rapid development of new applications over patching existing ones. Developers usually prefer to start their own new projects, not maintain older projects developed by others.

However, as a risk-reducing strategy, using an iterative refactoring approach is undeniably superior to a rewrite. The individual refactorings themselves are small compared to any similar portion of a rewrite project. They can be applied in much shorter periods of time than a comparable feature would be in a rewrite, and they leave the existing codebase in a working state at the end of each iteration. At no point does the existing application stop operating or progressing. The iterative refactorings can be integrated into a larger process with scheduling that allows for cycles of bug fixes, feature additions, and refactorings to improve the next cycle.

Finally, the goal of any single refactoring step is not perfection. The goal in each step is merely improvement. We are not trying to realize an impossible goal over a long period of time. We are taking small steps toward easily-visualized goals that can be accomplished in short timeframes. Each small refactoring win will both improve morale and drive enthusiasm for the next refactoring step. Over time, these many small wins accumulate into a single big win: a fully-modernized codebase that has never stopped generating revenue for the business.

Legacy Frameworks

Until now, we have been discussing legacy applications as page-based, include-oriented systems. However, there is also a large base of legacy code out there using public frameworks.

Framework-based Legacy Applications

Each different public framework in PHP land is its own unique hell. Applications written in *CakePHP* (http://cakephp.org/) suffer from different legacy issues than those written in CodeIgniter, Solar, Symfony 1, Zend Framework 1, and so on. Each of these different frameworks, and their varying work-alikes, encourage different kinds of tight-coupling in applications. Thus, the specific steps needed to refactor applications built using one of these frameworks are very different from the steps needed for a different framework.

As such, various parts of this book may be useful as a guide to refactoring different parts of a legacy application based on a public framework, but as a whole, the book is not targeted at refactoring applications based on these public frameworks.

In-house, private, or otherwise non-public frameworks under the direct control of their own architects within the organization likely to benefit from the refactorings included in this book.

Refactoring to a Framework

I sometimes hear about how developers wisely wish to avoid a complete rewrite and instead want to refactor or migrate to a public framework. This sounds like the best of both worlds, combining an iterative approach with the developers' desire to use the hottest new technology.

My experience with legacy PHP applications has been that they are almost as resistant to framework integration as they are to unit testing. If the application was already in a state where its logic could be ported to a framework, there would be little need to port it in the first place.

However, by the time we have completed the refactorings in this book, the application is very likely to be in a state that will be much more amenable to a public framework migration. Whether the developers will still want to do so is another matter.

Review and next steps

At this point, we have realized that a rewrite, while appealing, is a dangerous approach. An iterative refactoring approach sounds a lot more like actual work, but has the benefit of being achievable and realistic.

The next step is to prepare ourselves for the refactoring approach by getting some prerequisites out of the way. After that, we will proceed toward modernizing our legacy application in a series of relatively small steps, one step per chapter with each step broken down into an easy-to-follow process with answers to common questions.

Let's get started!

Chapter 2. Prerequisites

Before we begin modernizing our application, we need to make sure we have the necessary prerequisites in place to do the work of refactoring. These are as following:

- A revision control system
- A PHP version of 5.0 or higher
- An editor or IDE with multi-file search-and-replace
- A style guide of some sort
- A test suite

Revision control

Revision control (also known as source control or version control) allows us to keep track of the prerequisites:revision control" changes we make to our codebase. We can make a change, then commit it to source control, make more changes and commit them, and push our changes to other developers on the team. If we discover an error, we can revert to an earlier version of the codebase to a point where the error does not exist and start over.

If you are not using a source control tool like Git, Mercurial, Subversion, or some other revision control system, then that's the very first thing you need to put in place. Using source control will be a great benefit to you, even if you don't modernize your PHP application at all.

I prefer Mercurial in many ways, but I recognize that Git is more widely used, and as such I must recommend Git for new users of source control systems.

While it is beyond the scope of this book to discuss how to set up and use a source control system, there are some good Git books and Mercurial books available for free.

PHP version

In order to prerequisites:PHP 5.0" apply the refactorings listed in this book, we need at least PHP 5.0 installed. Yes, I know that PHP 5.0 is ancient, but we are talking about legacy applications here. It is entirely possible that the business owners have not upgraded their PHP versions in years. PHP 5.0 is the bare minimum, because that was when class autoloading became available, and we depend on autoloading as one of our very first improvements. (If for whatever reason we are stuck on PHP 4.x, then this book will be of little use.)

If we can get away with it, we should upgrade to the very latest version of PHP. I recommend using the most-recent version of PHP available to your chosen operating system. At the time of the latest update to this book, the most recent versions were PHP 5.6.11, 5.5.27, and 5.4.43.

Doing an upgrade from an older PHP version might itself entail modifying the application, as there are changes between minor versions in PHP. Approach this with care and attention to detail: check the release notes for the release and all intervening releases, look over the codebase, identify problems, make fixes, spot check locally, commit, push, and notify QA.

Editor/IDE

Throughout prerequisites: editor/IDE this book, we will be doing a lot of searching and modifying across the legacy codebase. We will need to have a text editor or IDE that allows us to find and replace text in multiple files at once. These include:

- Emacs
- PHPStorm
- SublimeText
- TextMate
- Vim

- Zend Studio

There are likely to be others as well.

Alternatively, if our CLI-fu is strong, we may wish to use grep and `sed` at the command line across multiple files at once.

Style Guide

Using a consistent prerequisites:style guide" coding style throughout the codebase is an important consideration. Most legacy codebases that I have seen are a mishmash of styles preferred by the various authors over time. One example of this kind of mishmash is the mixing of tabs and spaces for indenting code blocks: the developers early in the project used 2 spaces for indents, developers in the middle of the project used tabs, and recent developers used 4 spaces. This had the effect of putting some child blocks completely out prerequisites:style guide" of line with their parent blocks, either too much indented or not enough, making it difficult to scan for the beginning or end of a block.

We all long for a consistent, familiar coding style. There are few urges stronger than the urge to reformat an unfamiliar or undesired coding style to one that is more preferable. But modifying the existing style, no matter how ugly or inconsistent it is, can give rise to subtle bugs and behavioral changes from something as simple as adding or removing braces in a conditional. Then again, we want the code to be consistent and familiar so that we can read it with a minimum of cognitive friction.

It is tough to give good advice here. I suggest that the only reason to modify the existing style is when it is inconsistent within an individual file. If it is ugly or unfamiliar but otherwise consistent throughout the codebase, reformatting is likely to cause more problems than it solves.

If you decide to reformat, do so only as you move bits of code from one file to another, or as you move files from one location to another. This combines the large change of extraction-and-relocation with the more subtle change of style modification, and makes it possible to test those changes in a single pass.

Finally, you may want to convert to a completely new style, even though the existing one is consistent throughout the codebase. Resist that urge. If your desire to reformat in toto is overwhelming and cannot be ignored, use a publicly documented non-project-specific coding style instead of trying to create or apply your own personal or project-specific style. The code in this book uses the PSR-1 and PSR-2 style recommendations as a reflection of that advice.

Test suite

As this is a bookprerequisites:test suite" about legacy applications, it would be the height of optimism to expect that the codebase has a suite of unit tests. Most legacy applications, especially include-oriented, page-based applications, are highly resistant to unit tests. There are no units to test, only a spaghetti mess of tightly coupled functionality.

And yet it is possible to test a legacy application. The key here is not to test what the system units ought to do, but what the system as a whole already does. The criteria for a successful test is that the system generates the same output after a change as it did before that change. This kind of test is called a *characterization test*.

It is not in the scope of this book to discuss how to write a characterization test suite. There are some good tools out there already for writing these kinds of tests, such as Selenium and Codeception. Having tests of this sort before we go about refactoring the codebase is invaluable. We will be able to run the tests after each change to make sure the application still operates properly.

I will not pretend prerequisites:test suite" that we are likely to spend the time writing these kinds of tests. If we were interested in testing to begin with, we would have a test suite of some sort already. The issue here is a very human one, not of doing the right thing for its own sake or even of rational expectations but of incentives based on rewards. The reward for writing tests is a longer-term one, whereas making an improvement to the codebase right now feels immediately rewarding, even if we have to suffer with manual checking of the application output.

If you have the time, the self-discipline, and the resources, the best option is to create a series of characterization tests for the parts of the application you know you will be refactoring. It is the most responsible and most professional approach. As a second-best

option, if you have a QA team that already has a series of application-wide tests in place, you can delegate the testing process to them since they are doing it anyway. Perhaps they will show you how to run the test suite locally as you make changes to the codebase. Finally, as the least-professional but most-likely option, you will have to pseudo-test or spot check the application by hand when you make changes. This is probably what you are used to doing anyway. As your codebase improves, the reward for improving your own practices will become more evident; as with refactoring in general, the goal is to make things better than they were before in small increments, not to insist on immediate perfection.

Review and next steps

At this point we should have all our prerequisites in place, especially our revision control system and a modern version of PHP. Now we can begin with our very first step in refactoring: adding an autoloader to the codebase.

Chapter 3. Implement an Autoloader

In this step, we will set up automatic class loading. After this, when we need a class file, we will not need an `include` or `require` statement to load it for us. You should review the PHP documentation on autoloaders before continuing – http://www.php.net/manual/en/language.oop5.autoload.php.

PSR-0

There are many different autoloader recommendations in PHP land. The one we will be using to modernize our legacy application is based on something called `PSR-0`.

PSR-0 is a PHP Framework Interoperability Group recommendation for structuring your class files. The recommendation rises out of a long history of projects using the "class-to-file" naming convention from the days of PHP 4. Originating with Horde and PEAR, the convention was adopted by early PHP 5 projects such as Solar and Zend Framework, and later by projects such as Symfony2.

We use PSR-0 instead of the newer PSR-4 recommendation because we are dealing with legacy code, code that was probably developed before PHP 5.3 namespaces came into being. Code written before PHP 5.3 did not have access to namespace separators, so authors following the class-to-file naming convention would typically use underscores in class names as a pseudo-namespace separator. PSR-0 makes an allowance for older non-PHP-5.3 pseudo-namespaces, making it more suitable for our legacy needs, whereas PSR-4 does not.

Under PSR-0, the class name maps directly to a file system sub-path. Given a fully-qualified class name, any PHP 5.3 namespace

separators are converted to directory separators, and underscores in the class portion of the name are also converted to directory separators. (Underscores in the namespace portion proper are *not* converted to directory separators.) The result is prefixed with a base directory location, and suffixed with `.php`, to create a file path where the class file may be found. For example, the fully-qualified class name `\Foo\Bar\Baz_Dib` would be found in a sub-path named `Foo/Bar/Baz/Dib.php` on a UNIX-style file system.

A Single Location for Classes

Before we implement a PSR-0 autoloader, we need to pick a directory location in the codebase to hold every class that will ever be used in the codebase. Some projects already have such a location; it may be called `includes`, `classes`, `src`, `lib` or something similar.

If a location like that already exists, examine it carefully. Does it have *only* class files in it, or is it a combination of class files and other kinds of files? If it has anything besides class files in it, or if no such location exists, create a new directory location and call it classes (or some other properly descriptive name).

This directory will be the central location for all classes used throughout the project. Later, we will begin moving classes from their scattered locations in the project to this central location.

Add Autoloader Code

Once we have a central directory location for our class files, we need to set up an autoloader to look in that location for classes. We can create the autoloader as a static method, an instance method, an anonymous function, or a regular global function. (Which one we use is not as important as actually doing the autoloading.) Then we will register it with `spl_autoload_register()` early in our bootstrap or setup code, before any classes are called.

As a Global Function

Perhaps the most straightforward way to implement our new autoloader code is as a global function. Below, we find the autoloader code to use; the function name is prefixed with `mlaphp_` to make sure it does not conflict with any existing function names.

setup.php

```
1 <?php
2 // ... setup code ...
3
4 // define an autoloader function in the global
namespace
5 function mlaphp_autoloader($class)
6 {
7 // strip off any leading namespace separator
from PHP 5.3
8 $class = ltrim($class, '\\');
9
10 // the eventual file path
11 $subpath = '';
12
13 // is there a PHP 5.3 namespace separator?
14 $pos = strrpos($class, '\\');
15 if ($pos !== false) {
16 // convert namespace separators to directory
separators
17 $ns = substr($class, 0, $pos);
18 $subpath = str_replace('\\',
DIRECTORY_SEPARATOR, $ns)
19 . DIRECTORY_SEPARATOR;
20 // remove the namespace portion from the final
class name portion
21 $class = substr($class, $pos + 1);
22 }
23
24 // convert underscores in the class name to
directory separators
25 $subpath .= str_replace('_',
DIRECTORY_SEPARATOR, $class);
26
27 // the path to our central class directory
location
28 $dir = '/path/to/app/classes';
29
30 // prefix with the central directory location
and suffix with .php,
31 // then require it.
32 $file = $dir . DIRECTORY_SEPARATOR . $subpath .
'.php';
33 require $file;
34 }
35
36 // register it with SPL
```

```
37 spl_autoload_register('mlaphp_autoloader');
38 ?>
```

Note that the `$dir` variable represents an absolute directory as the base path for our central class directory. As an alternative on PHP 5.3 and later, it is perfectly acceptable to use the `__DIR__` constant in that variable so the absolute path is no longer hard-coded, but is instead relative to the file where the function is located. For example:

```
1 <?php
2 // go "up one directory" for the central classes
location
3 $dir = dirname(__DIR__) . '/classes';
4 ?>
```

If you are stuck on PHP 5.2 for some reason, the `__DIR__` constant is not available. You can replace `dirname(__DIR__)` with `dirname(dirname(__FILE__))` in that case.

As a Closure

If we are using PHP 5.3, we can create the autoloader code as a closure and register it with SPL in a single step:

setup.php

```
1 <?php
2 // ... setup code ...
3
4 // register an autoloader as an anonymous
function
5 spl_autoload_register(function ($class) {
6 // ... the same code as in the global function
...
7 });
8
9 // ... other setup code ...
10 ?>
```

As a Static or Instance method

This is my preferred way of setting up an autoloader. Instead of using a function, we create the autoloader code in a class as an instance method or a static method. I recommend instance methods over static ones, but your situation will dictate which is more appropriate.

First, we create our autoloader class file in our central class directory location. If we are using PHP 5.3 or later, we should use a proper namespace; otherwise, we use underscores as pseudo-namespace separators.

The following is a PHP 5.3 example. Under versions earlier than PHP 5.3, we would omit the `namespace` declaration and name the class `Mlaphp_Autoloader`. Either way, the file should be in the sub-path `Mlaphp/Autoloader.php`:

```
/path/to/app/classes/Mlaphp/Autoloader.php
1 <?php
2 namespace Mlaphp;
3
4 class Autoloader
5 {
6 // an instance method alternative
7 public function load($class)
8 {
9 // ... the same code as in the global function
...
10 }
11
12 // a static method alternative
13 static public function loadStatic($class)
14 {
15 // ... the same code as in the global function
...
16 }
17 }
18 ?>
```

Then, in the setup or bootstrap file, `require_once` the class file, instantiate it if needed, and register the method with SPL. Note that we use the array-callable format here, where the first array element

is either a class name or an object instance, and the second element is the method to call:

setup.php

```
1 <?php
2 // ... setup code ...
3
4 // require the autoloader class file
5 require_once
'/path/to/app/classes/Mlaphp/Autoloader.php';
6
7 // STATIC OPTION: register a static method with
SPL
8 spl_autoload_register(array('Mlaphp\Autoloader',
'loadStatic'));
9
10 // INSTANCE OPTION: create the instance and
register the method with SPL
11 $autoloader = new \Mlaphp\Autoloader();
12 spl_autoload_register(array($autoloader,
'load'));
13
14 // ... other setup code ...
15 ?>
```

Please pick either an instance method or a static method, not both. The one is not a fallback for the other.

Using The __autoload() Function

If we are stuck on PHP 5.0 for some reason, we can use the `__autoload()` function in place of the SPL autoloader registry. There are drawbacks to doing things this way, but under PHP 5.0 it is our only alternative. We do not need to register it with SPL (in fact, we cannot, since SPL was not introduced until PHP 5.1). We will not be able to mix-and-match other autoloaders in this implementation; only one `__autoload()` function is allowed. If an `__autoload()` function is already defined, we will need to merge this code with any code already existing in the function:

setup.php

```
1 <?php
2 // ... setup code ...
3
4 // define an __autoload() function
5 function __autoload($class)
6 {
7 // ... the global function code ...
8 }
9
10 // ... other setup code ...
11 ?>
```

I strongly recommend against using this kind of implementation in PHP 5.1 and later.

Autoloader Priority

Regardless of how we implement our autoloader code, we need it to be available *before* any classes get called in the codebase. It cannot hurt to register the autoloader as one of the very first bits of logic in our codebase, probably in a setup or bootstrap script.

Common Questions

What If I Already Have An Autoloader?

Some legacy applications may already have a custom autoloader in place. If this is our situation, we have some options:

1. **Use the existing autoloader as-is**: This is our best option if there is already a central directory location for the application class files.
2. **Modify the existing autoloader to add PSR-0 behavior**: This is a good option if the autoloader does not conform to PSR-0 recommendations.
3. Register the PSR-0 autoloader described in this chapter with SPL in addition to the existing autoloader. This is another good option when the existing autoloader does not conform to PSR-0 recommendations.

Other legacy codebases may have a third-party autoloader in place, such as Composer. If Composer is present, we can obtain its autoloader instance and add our central class directory location for autoloading like so:

```
1 <?php
2 // get the registered Composer autoloader
instance from the vendor/
3 // subdirectory
4 $loader = require
'/path/to/app/vendor/autoload.php';
5
6 // add our central class directory location; do
not use a class prefix as
7 // we may have more than one top-level namespace
in the central location
8 $loader->add('', '/path/to/app/classes');
9 ?>
```

With that, we can co-opt Composer for our own purposes, making our own autoloader code unnecessary.

What are the Performance Implications Of Autoloading?

There is some reason to think that using autoloaders may cause a slight performance drag compared to using `include`, but the evidence is mixed and situation-dependent. If it is true that autoloading is comparatively slower, how big of a performance hit can be expected?

I assert that, when modernizing a legacy application, it is probably not an important consideration. Any performance drag incurred from autoloading is minuscule compared to the other possible performance issues in your legacy application, such as the database interactions.

In most legacy applications, or even in most modern ones, attempting to optimize performance on autoloading is a case of attempting to optimize on the wrong resource. There are other resources that are likely to be worse offenders, just ones that we don't see or don't think of.

If autoloading is the single worst performance bottleneck in your legacy application, then you are in fantastic shape. (In that case, you should return this book for a refund, and then tell me if you are hiring, because I want to work for you.)

How Do Class Names Map To File Names?

The PSR-0 rules can be confusing. Here are some class-to-file mapping examples to illustrate its expectations:

```
Foo             => Foo.php
Foo_Bar         => Foo/Bar.php
Foo             => Foo/Bar.php
Foo_Bar\Bar     => Foo_Bar/Baz.php
Foo\Bar\Baz     => Foo/Bar/Baz.php # ???
Foo\Baz_Bar     => Foo/Bar/Baz.php # ???
Foo_Bar_Baz     => Foo/Bar/Baz.php # ???
```

We can see that there is some unexpected behavior in the last three examples. This is born of the transitional nature of PSR-0: `Foo\Bar\Baz, Foo\Bar_Baz`, and `Foo_Bar_Baz` all map to the same file. Why is this?

Recall that pre-PHP-5.3 codebases did not have namespaces, and so used underscores as a pseudo-namespace separator. PHP 5.3 introduced a real namespace separator. The PSR-0 standard had to accommodate both cases simultaneously, so it honors underscores in the relative class name (i.e., the final part of the fully-qualified name) as directory separators, but underscores in the namespace part are left alone.

The lesson here is that if you are on PHP 5.3, you should never use underscores in your relative class names (although underscores in the namespace are fine). If you are on a version before PHP 5.3, you have no choice but to use only underscores, as there is only the class name and no actual namespace portion; interpret underscores as namespace separators in that case.

Review and next steps

At this point we have not modified our legacy application very much. We have added and registered some autoloader code, but it is not actually being called yet.

No matter. Having an autoloader is critical to the next step in modernizing our legacy application. Using an autoloader will allow us to start removing `include` statements that only load classes and functions. The remaining `include` statements will be logical-flow includes, showing us which parts of the system are logic and which are definition-only. This is the beginning of our transition from an include-oriented architecture toward a class-oriented architecture.

Chapter 4. Consolidate Classes and Functions

Now that we have an autoloader in place, we can begin to remove all the `include` calls that only load up class and function definitions. When we are done, the only remaining `include` calls will be those that are executing logic. This will make it easier to see which `include` calls are forming the logic paths in our legacy application, and which are merely providing definitions.

We will start with a scenario where the codebase is structured relatively well. Afterwards, we will answer some questions related to layouts that are not so amenable to revision.

> **Note**
>
> For the purposes of this chapter, we will use the term `include` to cover not just `include` but also `require`, `include_once`, and `require_once`.

Consolidate Class Files

First, we will consolidate all the application classes to our central directory location as determined in the previous chapter. Doing so will put them where our autoloader can find them. Here is the general process we will follow:

1. Find an `include` statement that pulls in a class definition file.
2. Move that class definition file to our central class directory location, making sure that it is placed in a sub-path matching the PSR-0 rules.

3. In the original file *and in all other files in the codebase* where an `include` pulls in that class definition, remove that `include` statement.
4. Spot check to make sure that all the files now autoload that class by browsing to them or otherwise running them.
5. Commit, push, and notify QA.
6. Repeat until there are no more `include` calls that pull in class definitions.

For our examples, we will assume we have a legacy application with this partial file system layout:

```
/path/to/app/

classes/           # our central class directory
location
Mlaphp/
Autoloader.php     # A hypothetical autoloader
class
foo/ bar/ baz.php  # a page script
includes/          # a common "includes" directory
setup.php          # setup code
index.php          # a page script
lib/               # a directory with some classes
in it
sub/ Auth.php      # class Auth { ... }
Role.php           # class Role { ... }
User.php           # class User { ... }
```

Your own legacy application may not match this exactly, but you get the idea.

Find a candidate include

We begin by picking a file, any file, then we examine it for `include` calls. The code therein might look like this:

```
1 <?php
2 require 'includes/setup.php';
3 require_once 'lib/sub/User.php';
```

```
4
5 // ...
6 $user = new User();
7 // ...
8 ?>
```

We can see that there is a new `User` class being instantiated. On inspecting the `lib/sub/User.php` file, we can see it is the only class defined therein.

Move the class file

Having identified an `include` statement that loads a class definition, we now move that class definition file to the central class directory location so that our autoloader function can find it. The resulting file system layout now looks like this (note that `User.php` is now in `classes/`):

```
/path/to/app/
classes/                # our central class
directory location
Mlaphp/ Autoloader.php  # A hypothetical
autoloader class
User.php                # class User { ... }
foo/ bar/ baz.php       # a page script
includes/               # a common "includes"
directory
setup.php               # setup code
db_functions.php        # a function definition
file
index.php               # a page script
lib/                    # a directory with some
classes in it
sub/
Auth.php                # class Auth { ... }
Role.php                # class Role { ... } ~~
```

Remove the related include calls

Now the problem is that our original file is trying to `include` the class file from its old location, a location that no longer exists. We need to

remove that call from the code:

```
index.php
1 <?php
2 require 'includes/setup.php';
3
4 // ...
5 // the User class is now autoloaded
6 $user = new User();
7 // ...
8 ?>
```

However, there are likely to be other places where the code attempts to load the now-missing `lib/sub/User.php` file.

This is where a project-wide search facility comes in handy. We have different options here, depending on your editor/IDE of choice and operating system.

- In GUI editors like TextMate, SublimeText, and PHPStorm, there is usually a **Find in Project** menu item that we can use to search for a string or regular expression across all the application files at once.
- In other editors like Emacs and Vim, there is generally a key-binding that will search all the files in a particular directory and its subdirectories for a string or regular expression.
- Finally, if you are of the old school, you can use `grep` at the command line to search all the files in a particular directory and its subdirectories.

The point is to find all the `include` calls that refer to `lib/sub/User.php`. Because the `include` calls can be formed in different ways, we need to use a regular expression like this to search for the `include` calls:

```
^[ \t]*
(include|include_once|require|require_once).*User\
.php
```

If you are not familiar with regular expressions, here is a breakdown of what we are looking for:

```
^               Starting at the beginning of each
line,
[ \t]*          followed by zero or more spaces
and/or tabs,
(include|...)   followed by any of these words,
.*             followed by any characters at all,
User\.php      followed by User.php, and we don't
care what comes after.
```

(Regular expressions use `.` to mean `any character` so we have to specify `User\.php` to indicate we mean a literal dot, not any character.)

If we use a regular expression search to find those strings in the legacy codebase, we will be presented with a list of all matching lines and their corresponding files. Unfortunately, it is up to us to examine each line to see if it really is a reference to the `lib/sub/User.php` file. For example, this line might turn up in the search results:

```
include_once("/usr/local/php/lib/User.php");
```

However, clearly it is not the `User.php` file we are looking for.

Note

We could be more strict with our regular expression so that we search specifically for `lib/sub/User.php` but that is more likely to miss some `include` calls, especially those in files under the `lib/` or `sub/` directories. For example, an `include` in a file in `sub/` could look like this:

```
include 'User.php';
```

As such, it's better to be a little loose with the search to get every possible candidate, then work through the results manually.

Examine each search result line, and if it is an `include` that pulls in the `User` class, remove it and save the file. Keep a list of each modified file, as we will need to test them later.

At the end of this, we will have removed all the `include` calls for that class throughout the codebase.

Spot check the codebase

After removing the `include` statements for the given class, we now need to make sure the application works. Unfortunately, because we have no testing process in place, this means we need to pseudo-test or spot check by browsing to or otherwise invoking the modified files. In practice this is generally not difficult, but it is tedious.

When we spot check we are looking specifically for *file not found* and *class not defined* errors. These mean, respectively, that a file tried to `include` the missing class file, or that the autoloader failed to find the class file.

To do the testing we need to set PHP error reporting so that it either shows us the errors directly, or logs the errors to a file that we examine while testing the codebase. In addition, the error reporting level needs to be sufficiently strict that we actually see the errors. In general, `error_reporting(E_ALL)` is what we want, but because this is a legacy codebase, it may show more errors than we can bear (especially *variable not defined* notices). As such, it may be more productive to set `error_reporting(E_WARNING)`. The error reporting values can be set either in a setup or bootstrap file, or in the correct `php.ini` file.

Commit, Push, Notify QA

After the testing is complete and all errors have been fixed, commit the code to source control and (if needed) push it to the central code repository. If you have a QA team, now would be the time to notify them that a new testing round is needed, and provide them the list of files to test.

Do ... While

That is the process to convert a single class from `include` to autoloading. Go back through the codebase and find the next `include` that pulls in a class file and begin the process again. Continue doing so until all classes have been consolidated into the central class directory location and their relevant `include` lines have been removed. Yes, this is a tedious, tiresome, and time-consuming process, but it is a necessary step towards modernizing our legacy codebase.

Consolidate functions into class files

Not all legacy applications use a large set of classes. Often, instead of classes, there is a significant number of user-defined functions for core logic.

Using functions is not a problem in itself, but it does mean that we need to `include` the files where the functions are defined. But autoloading only works for classes. It would be good to find a way to automatically load the function files as well as the class files. That would help us remove even more `include` calls.

The solution here is to move the functions into class files, and call the functions as static methods on those classes. That way, the autoloader can load up the class file for us, and then we can call the methods in that class.

This procedure is more complex than when we consolidated class files. Here is the general process we will follow:

1. Find an `include` statement that pulls in a function definition file.
2. Convert that function definition file into a class file of static methods; we need to pick a unique name for the class, and we may need to rename the functions to more suitable method names.
3. In the original file *and in all other files in the codebase* where any functions from that file are used, change calls to those functions into static method calls.
4. Spot check to see if the new static method calls work by browsing to or otherwise invoking the affected files.
5. Move the class file to the central class directory location.
6. In the original file *and in all other files in the codebase* where an `include` pulls in that class definition, remove the relevant `include` statement.

7. Spot check again to make sure that all the files now autoload that class by browsing to them or otherwise running them.
8. Commit, push, and notify QA.
9. Repeat until there are no more `include` calls that pull in function definition files.

Find a candidate include

We pick a file, any file, and look through it for `include` calls. The code in our chosen file might look like this:

```
1 <?php
2 require 'includes/setup.php';
3 require_once 'includes/db_functions.php';
4
5 // ...
6 $result = db_query('SELECT * FROM table_name');
7 // ...
8 ?>
```

We can see that there is a `db_query()` function being used, and on inspecting the `includes/db_functions.php` file, we can see that function along with several others defined therein.

Convert the function file to a class file

Let's say that the `db_functions.php` file looks something like this:

includes/db_functions.php

```
1 <?php
2 function db_query($query_string)
3 {
4 // ... code to perform a query ...
5 }
6
7 function db_get_row($query_string)
8 {
9 // ... code to get the first result row
10 }
11
```

```
12 function db_get_col($query_string)
13 {
14 // ... code to get the first column of results
...
15 }
16 ?>
```

To convert this function file to a class file, we need to pick a unique name for the class we're about to create. It seems pretty clear in this case, both from the file name and from the function names, that these are all database-related calls. As such, we'll call this class Db.

Now that we have a name, we'll create the class. The functions will become static methods in the class. We are not going to move the file just yet; leave it in place with its current file name.

Then we make our changes to convert the file to a class definition. If we change function names, we need to keep a list of old and the new names for later use. After the changes, it will look something like the following (note the changed method names):

includes/db_functions.php

```
1 <?php
2 class Db
3 {
4 public static function query($query_string)
5 {
6 // ... code to perform a query ...
7 }
8
9 public static function getRow($query_string)
10 {
11 // ... code to get the first result row
12 }
13
14 public static function getCol($query_string)
15 {
16 // ... code to get the first column of results
...
17 }
18 }
19 ?>
```

The changes are very moderate: we wrapped the functions in a unique class name, marked them as `public static`, and made minor changes to the function names. We made no changes at all to the function signatures or code in the functions themselves.

Change function calls to static method calls

We have converted the contents of `db_functions.php` from function definitions to a class definition. If we try to run the application now, it will fail with "undefined function" errors. So, the next step is to find all of the relevant function calls throughout the application and rename them to static method calls on our new class.

There is no easy way to do this. This is another case where project-wide search-and-replace becomes very handy. Using our preferred project-wide search tool, search for the `old` function call, and replace it with the `new` static method call. For example, using a regular expression, we might do this:

Search for:

```
db_query\s*\(
```

Replace with:

```
Db::query(
```

The regular expression indicates the opening parenthesis, not the closing one, as we don't need to look for parameters in the function call. This helps to distinguish from function names that might be prefixed with the function name we're searching for, such as `db_query_raw()`. The regular expression also allows for optional whitespace between the function name and the opening parenthesis, since some style guides recommend such spacing.

Perform this search-and-replace for each of the `old` function names in the old function file, converting each to the `new` static method call

in the new class file.

Spot check the static method calls

When we are finished renaming the old function names to the new static method calls, we need to run through the codebase to make sure everything works. Again, there is no easy way to do this. You may need to go so far as browsing to, or otherwise invoking, each file that was changed in this process.

Move the class file

At this point we have replaced the contents of the function definition file with a class definition, and "testing" has showed that the new static method calls work as expected. Now we need to move the file to our central class directory location and name it properly.

Currently, our class definition is in the `includes/db_functions.php` file. The class in that file is named `Db`, so move the file to its new autoloadable location as `classes/Db.php`. Afterwards, the file system will look something like this:

```
/path/to/app/
classes/            # our central class directory
location
Db.php              # class Db { ... }
Mlaphp/
Autoloader.php      # A hypothetical autoloader
class
User.php            # class User { ... }
foo/
bar/
baz.php             # a page script
includes/           # a common "includes" directory
setup.php           # setup code
index.php           # a page script
lib/                # a directory with some classes
in it
sub/
Auth.php            # class Auth { ... }
Role.php            # class Role { ... }
```

Do ... While

Finally, we follow the same ending process as we did when moving class files:

- Remove the related `include` calls for the function definition file throughout the codebase
- Spot check the codebase
- Commit, push, notify QA

Now repeat it for every function definition file we find in the codebase.

Common Questions

Should we remove the autoloader include call?

If we placed our autoloader code in a class as a static or instance method, our search for `include` calls will reveal the inclusion of that class file. If you remove that `include` call, autoloading will fail, because the class file will not have been loaded. This is a chicken-and-egg problem. The solution is to leave the autoloader `include` in place as part of our bootstrapping or setup code. If we are fully diligent about removing `include` calls, that is likely to be the only `include` remaining in the codebase.

How should we pick files for candidate include calls?

There are several ways to go about this. We could do the following:

- We can manually traverse the entire codebase and work file-by-file.
- We can generate a list of class and function definition files, and then generate a list of files that `include` those files.
- We can search for every `include` call and look at the related file to see if it has class or function definitions.

What if an include defines more than one class?

Sometime a class definition file may have more than one class definition in it. This can mess with the autoloading process. If a file

named `Foo.php` defines both `Foo` and `Bar` classes, then the `Bar` class will never be autoloaded, because the file name is wrong.

The solution is to split the single file into multiple files. That is, create one file per class, and name each file for the class it contains per the PSR-0 naming and autoloading expectations.

What if the one-class-per-file rule is disagreeable?

I sometimes hear complaints about how the one-class-per-file rules is somehow wasteful or otherwise not aesthetically pleasing when examining the file system. Isn't it a drag on performance to load that many files? What if some classes are only needed along with some other class, such as an `Exception` that is only used in one place? I have some responses here:

- There is, of course, a performance reduction in loading two files instead of one. The question is *how much* of a reduction, and *compared to what*? I assert that, compared to the other more likely performance issues in our legacy application, the drag from loading multiple files is a rounding error. It is more likely that we have other, far greater performance concerns. If it really is a problem, using a bytecode cache like APC will reduce or completely remove these comparatively small performance hits.
- Consistency, consistency, consistency. If some of the time a class file has only one class in it, and at other times a class file has more than one class in it, that inconsistency will later become a source of cognitive friction for everyone on the project. One of the main themes through legacy applications is that of inconsistency; let us reduce that inconsistency as much as we can by adhering to the one-class-per-file rule.

If we feel that some classes naturally belong together, it is perfectly acceptable to place the subordinate or child classes in a subdirectory beneath the master or parent class. The subdirectory should be

named for that higher class or namespace, per the PSR-0 naming rules.

For example, if we have a series of `Exception` classes related to a `Foo` class:

```
Foo.php                        # class Foo { ... }
Foo/
NotFoundException.php          # class
Foo_NotFoundException { ... }
MalformedDataException.php     # class
Foo_MalformedDataException { ... }
```

Renaming classes in this way will change the related class names throughout the codebase where they are instantiated or otherwise referenced.

What if a Class or Function is defined inline?

I have seen cases where a page script has one or more classes or functions defined inside it, generally when the classes or functions are used only by that particular page script.

In these cases, remove the class definitions from the script and place them in their own files in the central class directory location. Be sure to name the files for their class names per the PSR-0 autoloader rules. Similarly, move the function definitions to their own related class file as static methods, and rename the function calls to static method calls.

What if a definition file also executes logic?

I have also seen the opposite case, where a class file has some logic that gets executed as a result of the file being loaded. For

example, a class definition file might look like this:

```
/path/to/foo.php
1 <?php
2 echo "Doing something here ...";
3 log_to_file('a log entry');
4 db_query('UPDATE table_name SET incrementor =
incrementor + 1');
5
6 class Foo
7 {
8 // the class
9 }
10 ?>
```

In the above case, the logic before the class definition will be executed when the file is loaded, even if the class is never instantiated or otherwise called.

This is a much tougher situation to deal with than when classes are defined inline with a page script. The class should be loadable without side effects, and the other logic should be executable without having to load the class.

In general, the easiest way to deal with this is to modify our relocation process. Cut the class definition from the original file and place it in its own file in the central class directory location. Leave the original file with its executable code in place, and leave all the related `include` calls in place as well. This allows us to pull out the class definition so it can be autoloaded, but scripts that `include` the original file still get the executable behavior.

For example, given the above combined executable code and class definition, we could end up with these two files:

```
/path/to/foo.php
1 <?php
2 echo "Doing something here ...";
3 log_to_file('a log entry');
4 db_query('UPDATE table_name SET incrementor =
```

```
incrementor + 1');
5 ?>
```

/path/to/app/classes/Foo.php

```
1 <?php
2 class Foo
3 {
4 // the class
5 }
6 ?>
```

This is messy, but it preserves the existing application behavior while allowing for autoloading.

What if two classes have the same name?

When we start moving classes around, we may discover that `application flow A` uses a `Foo` class, and that `application flow B` also uses a `Foo` class, but the two classes of the same name are actually different classes defined in different files. They never conflict with each other because the two different application flows never intersect.

In this case, we have to rename one or both of the classes when we move them to our central class directory location. For example, call one of them `FooOne` and the other `FooTwo`, or pick better descriptive names of your own. Place them each in separate class files named for their class names, per the PSR-0 autoloading rules, and rename all references to these classes throughout the codebase.

What about third-party libraries?

When we consolidate our classes and functions, we may find some third-party libraries in the legacy application. We don't want to move or rename the classes and functions in a third-party library, because that would make it too difficult to upgrade the library later. We would have to remember what classes were moved where and which functions were renamed to what.

With any luck, the third-party library uses autoloading of some sort already. If it comes with its own autoloader, we can add that autoloader to the SPL autoloader registry stack in our setup or bootstrap code. If its autoloading is managed by another autoloader system, such as that found in Composer, we can add *that* autoloader to the SPL autoloader registry stack, again in our setup or bootstrap code.

If the third-party library does not use autoloading, and depends on `include` calls both in its own code and in the legacy application, we are in a bit of a bind. We don't want to modify the code in the library, but at the same time we want to remove `include` calls from the legacy application. The two solutions here are *least-worst* options:

- Modify our application's main autoloader to allow for one or more third party libraries
- Write an additional autoloader for the third-party library and add it to the SPL autoloader registry stack.

Both of these options are beyond the scope of this book. You will need to examine the library in question, determine its class naming scheme, and come up with appropriate autoloader code on your own.

Finally, in terms of how to organize third-party libraries in the legacy application, it might be wise to consolidate them all to their own central location in the codebase. For example, this might be under a directory called `3rdparty/` or `external_libs/`. If we move a library, we should move the entire package, not just its class files, so we can upgrade it properly later. This will also allow us to exclude the central third-party directory from our search for `include` calls so that we don't get extra search results from files that we don't want to modify.

What about system-wide libraries?

System-wide library collections, like those provided by Horde and PEAR, are a special case of third-party libraries. They are generally

located on the server file system *outside* of the legacy application so they can be available to all applications running on that server. The `include` statements related to these system-wide libraries generally depend on the `include_path` settings, or else are referenced by absolute path.

These present a special problem when trying to eliminate `include` calls that only pull in class and function definitions. If we are lucky enough to be using PEAR-installed libraries, we can modify our existing autoloader to look in two directories instead of one. This is because the PSR-0 naming conventions rise out of the Horde/PEAR conventions. The trailing autoloader code changes from this:

```
1 <?php
2 // convert underscores in the class name to
directory separators
3 $subpath .= str_replace('_',
DIRECTORY_SEPARATOR, $class);
4
5 // the path to our central class directory
location
6 $dir = '/path/to/app/classes'
7
8 // prefix with the central directory location
and suffix with .php,
9 // then require it.
10 require $dir . DIRECTORY_SEPARATOR . $subpath .
'.php';
11 ?>
```

To this:

```
1 <?php
2 // convert underscores in the class name to
directory separators
3 $subpath .= str_replace('_',
DIRECTORY_SEPARATOR, $class);
4
5 // the paths to our central class directory
location and to PEAR
6 $dirs = array('/path/to/app/classes',
'/usr/local/pear/php');
```

```
7 foreach ($dirs as $dir) {
8 $file = $dir . DIRECTORY_SEPARATOR . $subpath .
'.php';
9 if (file_exists($file)) {
10 require $file;
11 }
12 }
13 ?>
```

For functions, can we use instance methods instead of static methods?

When we consolidated user-defined global functions into classes, we redefined them as static methods. This left their global scope unchanged. If we feel particularly diligent, we can change them from static to instance methods. This involves more work, but in the end it can make testing easier and is a cleaner technical approach. Given our earlier `Db` example, using instance instead of static methods would look like this:

classes/Db.php

```
1 <?php
2 class Db
3 {
4 public function query($query_string)
5 {
6 // ... code to perform a query ...
7 }
8
9 public function getRow($query_string)
10 {
11 // ... code to get the first result row
12 }
13
14 public function getCol($query_string)
15 {
16 // ... code to get the first column of results
...
17 }
18 }
19 ?>
```

The only added step when using instance methods instead of static ones is that we need to instantiate the class before calling its methods. That is, instead of this:

```
1 <?php
2 Db::query(...);
3 ?>
```

We would do this:

```
1 <?php
2 $db = new Db();
3 $db->query(...);
4 ?>
```

Even though it is more work in the beginning, I recommend instance methods over static ones. Among other things, it gives us a constructor method that can be called on instantiation, and it makes testing easier in many cases.

If you like, you may wish to start by converting to static methods, and then later convert the static methods to instance methods, along with all the related method calls. However, your schedule and preferences will dictate which approach you choose.

Can we automate this process?

As I have noted before, this is a tedious, tiresome, and time-consuming process. Depending on the size of the codebase, it may take days or weeks of effort to fully consolidate the classes and functions for autoloading. It would be great if there was some way to automate the process to make it both faster and more reliable.

Unfortunately, I have not yet discovered any tools that make this process easier. As far as I can tell, this kind of refactoring is still best done by hand with strong attention to detail. Having obsessive tendencies and long periods of uninterrupted concentration on this task are likely to be of benefit here.

Review and next steps

At this point, we have made a big step forward in modernizing our legacy application. We have begun converting from an *include-oriented* architecture to a *class-oriented* one. Even if we later discover a class or function that we missed, that's OK; we can follow the above process as many times as needed until all definitions have been moved to the central location.

We may still have lots of `include` statements in the application, but those that remain are related to the application flow, and not to pulling in class and function definitions. Any `include` calls that remain are executing logic. We can now see the flow of the application much better.

We have put in place a structure for new functionality. Any time we need to add a new behavior, we can place it in a new class, and that class will be autoloaded whenever we need it. We can stop writing new stand-alone functions; instead, we will write new methods on classes. These new methods will be much more amenable to unit tests.

However, the *existing* classes that we have consolidated for autoloading are likely to have globals and other dependencies in them. This makes them tightly bound to each other and difficult to write tests for. With that in mind, the next step is to examine the dependencies in our existing classes, and attempt to break those dependencies to improve the maintainability of our application.

Chapter 5. Replace global With Dependency Injection

At this point, all of our classes and functions have been consolidated to a central location, and all related `include` statements have been removed. We would prefer to start writing tests for our classes, but it is very likely that we have a lot of `global` variables embedded in them. These can cause a lot of trouble via action at a distance where modifying a `global` in one place changes its value in another place. The next step, then, is to remove all uses of the `global` keyword from our classes, and inject the necessary dependencies instead.

> ## Note
>
> **What Is Dependency Injection?**
>
> Dependency injection means that we push our dependencies into a class from the outside, instead of pulling them into a class while inside the class. (U sing `global` pulls a variable into the current scope from the global scope, so it is the opposite of injection.) Dependency injection turns out to be very straightforward as a concept, but is sometimes difficult to adhere to as a discipline.

Global Dependencies

To start with a naive example, let's say an `Example` class needs a database connection. Here we create the connection inside a class method:

classes/Example.php

```
1 <?php
```

```
2 class Example
3 {
4 public function fetch()
5 {
6 $db = new Db('hostname', 'username',
'password');
7 return $db->query(...);
8 }
9 }
10 ?>
```

We are creating the `Db` dependency inside the method that needs it. There are several problems with this. Some of them are:

- Every time we call this method, we create a new database connection, which may strain our resources.
- If we ever need to change the connection parameters, we need to modify them in every place we create a connection.
- It is difficult to see from the outside of this class what its dependencies are.

After writing code like this, many developers discover the `global` keyword, and realize they can create the connection once in a setup file, then pull it in from the global scope:

setup.php

```
1 <?php
2 // some setup code, then:
3 $db = new Db('hostname', 'username',
'password');
4 ?>
```

classes/Example.php

```
1 <?php
2 class Example
3 {
4 public function fetch()
5 {
6 global $db;
7 return $db->query(...);
8 }
9 }
10 ?>
```

Even though we are still pulling in the dependency, this technique solves the problem of multiple database connections using up limited resources, since the same database connection is reused across the codebase. The technique also makes it possible to change our connection parameters in a single location, the `setup.php` file, instead of several locations. However, one problem remains, and one is added:

- We still cannot see from the outside of the class what its dependencies are.
- If the `$db` variable is ever changed by any of the calling code, that change is reflected throughout the codebase, leading to debugging trouble.

The last point is a killer. If a method ever sets `$db = 'busted';` then the `$db` value is now a string, and not a database connection object, throughout the entire codebase. Likewise, if the `$db` object is modified, then it is modified for the entire codebase. This can lead to very difficult debugging sessions.

The replacement process

Thus, we want to remove all `global` calls from the codebase to make it easier to troubleshoot, and to reveal the dependencies in our classes. Here is the general process we will use to replace `global` calls with dependency injection:

1. Find a `global` variable in one of our classes.
2. Move all `global` variables in that class to the constructor and retain their values as properties, and use the properties instead of the globals.
3. Spot check that the class still works.
4. Convert the `global` calls in the constructor to constructor parameters.
5. Convert all instantiations of the class to pass the dependencies.
6. Spot check, commit, push, and notify QA.

7. Repeat with the next `global` call in our class files, until none remain.

> **Note**
>
> In this process, we work *one class at a time* and not *one variable at a time*. The former is much less time-consuming and more unit-oriented than the latter.

Find a global variable

This is easy with a project-wide search function. We search for `global` within the central class directory location, and get back a list of class files with that keyword in them.

Convert global variables to properties

Let's say that our search revealed an `Example` class with code something like the following:

```
classes/Example.php
1 <?php
2 class Example
3 {
4 public function fetch()
5 {
6 global $db;
7 return $db->query(...);
8 }
9 }
10 ?>
```

We now move the global variable to a property that gets set in the constructor, and convert the `fetch()` method to use the property:

classes/Example.php

```
1 <?php
2 class Example
3 {
4 protected $db;
5
6 public function __construct()
7 {
8 global $db;
9 $this->db = $db;
10 }
11
12 public function fetch()
13 {
14 return $this->db->query(...);
15 }
16 }
17 ?>
```

> **Tip**
>
> If there are multiple `global` calls inside the same class, we should convert all of them to properties in that class. We want to work *one class at a time* as this makes later parts of this process easier.

Spot check the class

Now that we have converted `global` calls to properties in this one class, we need to test the application to make sure it still works. However, since there is no formal testing system in place yet, we pseudo-test or spot check by browsing to or otherwise invoking files that use the modified class.

If we like, we can make an interim commit here once we are sure the application still works. We will not push to the central repository or notify QA just yet; all we want is a point to which we can roll back if later changes need to be undone.

Convert global properties to constructor parameters

Once we ascertain that the class works with the properties in place, we need to convert the `global` calls in the constructor to use passed parameters instead. Given our `Example` class above, the converted version might look like this:

classes/Example.php

```
1 <?php
2 class Example
3 {
4 protected $db;
5
6 public function __construct(Db $db)
7 {
8 $this->db = $db;
9 }
10
11 public function fetch()
12 {
13 return $this->db->query(...);
14 }
15 }
16 ?>
```

All we have done here is remove the `global` call, and added a constructor parameter. We need to do this for every `global` in the constructor.

Since the `global` is for a particular class of object, we typehint the parameter to that class (in this case `Db`). If possible, we should typehint to an interface instead, so if the `Db` object implements a *DbInterface*, we should typehint to *DbInterface*. This will help with testing and later refactoring. We may also typehint to `array` or `callable` as appropriate. Not all `global` calls are for typed values, so not all parameters will need typehints (e.g., when the parameter is expected to be a string).

Convert instantiations to use parameters

After converting `global` variables to constructor parameters, we will find that every instantiation of the class throughout the legacy application is now broken. This is because the constructor signature has changed. With that in mind, we now need to search *the entire codebase* (not just the classes) for instantiations of the class, and change the instantiations to the new signature.

To search for instantiations, we use our project-wide search facility to find uses of the `new` keyword with our class name using a regular expression:

```
new\s+Example\W
```

The expression searches for the `new` keyword, followed by at least one character of whitespace, followed by a terminating non-word character (such as a parenthesis, space, or semicolon).

Note

Formatting Issues

Legacy codebases are notorious for having messed-up formatting, which means this expression is imperfect in some situations. The expression as given here may not find instantiations where, for example, the `new` keyword is on one line, and the class name is the very next thing but is on the next line, not the same line.

Class Aliases With use

In PHP 5.3 and later, classes may be aliased to another class name with a use statement, like so:

```
1 <?php
2 use Example as Foobar;
3 // ...
4 $foo = new Foobar;
5 ?>
```

In this case, we need to do two searches: one for use `\s+Example\s+as` to discover the various aliases, and a second search for the new keyword with the alias.

As we discover instantiations of the class in the codebase, we modify them to pass the parameters as needed. If, for example, a page script looks like this:

page_script.php

```
1 <?php
2 // a setup file that creates a $db variable
3 require 'includes/setup.php';
4 // ...
5 $example = new Example;
6 ?>
```

We need to add the parameter to the instantiation:

page_script.php

```
1 <?php
2 // a setup file that creates a $db variable
3 require 'includes/setup.php';
4 // ...
5 $example = new Example($db);
6 ?>
```

The new instantiations need to match the new constructor signature, so if the constructor takes more than one parameter, we need to pass all of the parameters.

Spot check, Commit, Push, Notify QA

We have reached the end of the conversion process for this class. We need to spot check the converted instantiations now, but (as always) this is not an automated process, so we need to run or otherwise invoke the files with the changed code. If there are problems, go back and fix them.

Once we have done so, and are sure there are no errors, we can commit the changed code, push it to our central repository, and notify QA that it needs to run its test suite over the legacy application.

Do ... While

That is the process to convert a single class from using `global` calls to using dependency injection. Go back through the class files and find the next class with a `global` call and begin the process again. Continue to do so until there are no more `global` calls in the classes.

Common Questions

What if we find a global in a static method?

Sometimes we will find that a static class method uses a `global` variable like so:

```
1 <?php
2 class Foo
3 {
4 static public function doSomething($baz)
5 {
6 global $bar;
7 // ... do something with $bar ...
8 }
9 }
10 ?>
```

This is a problem because there is no constructor to which we can move the `global` variable as a property. There are two options here.

The first option is to pass all the needed globals as parameters on the static method itself, thereby changing the signature of the method:

```
1 <?php
2 class Foo
3 {
4 static public function doSomething($bar, $baz)
5 {
6 // ... do something with $bar ...
7 }
8 }
9 ?>
```

We would then search the codebase for all uses of `Foo::doSomething(` and pass the `$bar` value each time. For that

reason, I suggest adding the new parameters to the *beginning* of the signature, rather than to the end, because it makes search-and-replace much easier. For example:

Search for:

```
Foo::doSomething\(
```

Replace with:

```
Foo::doSomething\($bar,
```

The second option is to change the class so that it must be instantiated, and make all the methods instance methods. The class, after conversion, might look like this:

```
1 <?php
2 class Foo
3 {
4 protected $bar;
5
6 public function __construct($bar)
7 {
8 $this->bar = $bar;
9 }
10
11 public function doSomething($baz)
12 {
13 // ... do something with $this->bar ...
14 }
15 }
16 ?>
```

After that, we would need to:

1. Search the codebase for all `Foo::` static calls;
2. Create instances of `Foo` with its `$bar` dependency (e.g., `$foo = new Foo($bar);`) before those static calls are made, and
3. Replace calls of `Foo::doSomething()` with `$foo->doSomething()`.

Is there an alternative conversion process?

The process described above is a class-by-class process, where we first move the globals in a single class to the constructor, then change from global properties to instance properties in that class, and finally change instantiations of that class.

Alternatively, we might choose a modified process:

1. Change all global variables to properties in all classes, then test/commit/push/QA.
2. Change all global properties to constructor parameters *in all classes*, and change instantiations *of all classes*, then test/commit/push/QA.

This may be a reasonable alternative for smaller codebases, but it comes with some problems, such as:

1. The search for `global` calls becomes a little more difficult while converting globals to properties, because we will see the `global` keyword in both the converted and unconverted classes.
2. The commits for each major step will be much larger and harder to read.

For these reasons and others, I think it's better to stay with the process as described. It works with large and small codebases, and keeps incremental changes in smaller easier-to-read portions.

What about class names in variables?

Sometimes we will find that classes are instantiated based on variable values. For example, this creates an object based on the value of the `$class` variable:

```
page_script.php
1 <?php
2 // $type is defined earlier in the file, and
then:
3 $class = $type . '_Record';
4 $record = new $class;
5 ?>
```

If `$type` is `Blog`, then the `$record` object will be of the class `Blog_Record`.

This kind of thing is very difficult to discover when searching for class instantiations to convert to using constructor parameters. I'm afraid I have no good advice for automatically finding these kinds of instantiations. The best we can do is to search for `new\s+\$` without any class name, and modify the calls individually by hand.

What about superglobals?

Superglobals represent a challenging special case when removing global variables. They are automatically global within every scope, so they carry all the drawbacks of globals. We won't find them with a search for the `global` keyword (although we can search for them by name). Because they truly are global, we need to remove them from our classes just as much as we need to remove the `global` keyword.

We could pass a copy of each superglobal into the class when we need it. In cases where we need only one this might be fine, but frequently we need two or three or more superglobals. In addition, passing a copy of `$_SESSION` will not work as expected; PHP uses the actual superglobal of `$_SESSION` for writing session data, so changes to the copy will not be honored.

As a solution, we can use a `Request` data structure class. The `Request` encapsulates a copy of each of the non-`$_SESSION` superglobals. At the same time, the `Request` maintains a reference to `$_SESSION` so that changes to the object property are honored by the real `$_SESSION` superglobal.

Note

Note that the `Request` is not an HTTP request object per se. It is merely a representation of the request environment for PHP, including server, environment, and session values, many of which are not found in HTTP messages.

For example, say we have a class that uses `$_POST`, `$_SERVER`, and `$_SESSION`:

```
1 <?php
2 class PostTracker
3 {
4 public function incrementPostCount()
5 {
6 if ($_SERVER['REQUEST_METHOD'] != 'POST') {
7 return;
8 }
9
10 if (isset($_POST['increment_count'])) {
11 $_SESSION['post_count'] ++;
12 }
13 }
14 }
15 ?>
```

To replace these calls, we first create a shared `Request` object in our setup code.

includes/setup.php

```
1 <?php
2 // ...
3 $request = new \Mlaphp\Request($GLOBALS);
4 // ...
5 ?>
```

We can then decouple from the superglobals by injecting that shared `Request` object in to any class that needs it, and use the `Request` properties instead of the superglobals:

```
1 <?php
2 use Mlaphp\Request;
3
4 class PostTracker
5 {
6 public function __construct(Request $request)
7 {
8 $this->request = $request;
9 }
10
11 public function incrementPostCount()
12 {
13 if ($this->request->server['REQUEST_METHOD'] !=
'POST') {
14 return;
15 }
16
17 if (isset($this->request-
>post['increment_count'])) {
18 $this->request->session['post_count'] ++;
19 }
20 }
21 }
22 ?>
```

Tip

If it is important to maintain changes to the superglobal values across scopes, be sure to use the same `Request` object throughout the application. Modifications to the values in one `Request` object will not be reflected in a different `Request` object, except for `$session` values (because they are all references to `$_SESSION`).

What about $GLOBALS?

There is one more superglobal that PHP provides: `$GLOBALS`. Using this superglobal inside our classes and methods should be treated as a use of the `global` keyword. For example, `$GLOBALS['foo']` is the

equivalent of `global $foo`. We should remove it from our classes in just the same way as we do with uses of `global`.

Review and next steps

At this point, we have removed all `global` calls in our classes, as well as all uses of superglobals. This is another big improvement in the quality of our codebase. We know that variables can be modified locally and not affect other parts of the codebase.

However, our classes may still have hidden dependencies in them. In order to make our classes more testable, we need to discover and reveal those dependencies. That is the subject of the next chapter.

Chapter 6. Replace new with Dependency Injection

Even though we have removed all `global` calls in our classes, they are likely to retain other hidden dependencies. In particular, we are probably creating new object instances in inappropriate locations, tightly coupling the classes together. These things make it much harder to write tests and to see what the internal dependencies are.

Embedded instantiation

After converting the `global` calls in a hypothetical *ItemsGateway* class, we might have something like this:

classes/ItemsGateway.php

```
1 <?php
2 class ItemsGateway
3 {
4 protected $db_host;
5 protected $db_user;
6 protected $db_pass;
7 protected $db;
8
9 public function __construct($db_host, $db_user,
$db_pass)
10 {
11 $this->db_host = $db_host;
12 $this->db_user = $db_user;
13 $this->db_pass = $db_pass;
14 $this->db = new Db($this->db_host, $this-
>db_user, $this->db_pass);
15 }
16
17 public function selectAll()
18 {
19 $rows = $this->db->query("SELECT * FROM items
ORDER BY id");
20 $item_collection = array();
21 foreach ($rows as $row) {
```

```
22 $item_collection[] = new Item($row);
23 }
24 return $item_collection;
25 }
26 }
27 ?>
```

There are two dependency injection issues here:

1. First, the class probably got converted from a function that used `global $db_host`, `$db_user`, `$db_pass` and then constructed a `Db` object internally. Our initial pass at removing `global` calls got rid of the globals, but it left this `Db` dependency in place. This is what we will call a one-time creation dependency.
2. Second, the `selectAll()` method creates new `Item` objects, and as such is dependent on the `Item` class. We cannot see this dependency from the outside of the class. This is what we will call a repeated creation dependency.

Note

As far as I know, the terms one-time creation dependency and repeated creation dependency are not industry-standard terms. They are for the purposes of this book only. Please inform the author if you are aware of similar concepts that have industry-standard terms.

The point of dependency injection is to push the dependencies in from the outside, thereby revealing the dependencies in our classes. Using a `new` keyword inside a class is in opposition to that idea, so we need to work through the codebase to remove that keyword from our non-`Factory` classes.

Note

What is a Factory Object?

One of the keys to dependency injection is that an object may either create other objects, *or* it may operate on other objects, *but not both*. Any time we need to create an object inside another object, we let that work be done by something called a *Factory* with a `newInstance()` method and inject that *Factory* into the object that needs to do creation. The `new` keyword is restricted to being used inside *Factory* objects. This allows us to switch out *Factory* objects any time we need to create different kinds of objects.

The replacement process

The next step, then, is to remove all use of `new` keyword from our non-*Factory* classes, and inject the necessary dependencies instead. We will also use *Factory* objects as needed to deal with repeated creation dependencies. This is the general process we will follow:

1. Find a class with the `new` keyword in it. If the class is already a `Factory`, we can ignore it and move on.
2. For each one-time creation in the class:

    - Extract each instantiation to a constructor parameter.
    - Assign the constructor parameter to a property.
    - Remove any constructor parameters and class properties that are used only for the `new` call.

3. For each repeated creation in the class:

    - Extract each block of creation code to a new `Factory` class.
    - Create a constructor parameter for each `Factory` and assign it to a property.
    - Modify the previous creation logic in the class to use the *Factory*.

4. Change all instantiation calls for the modified class throughout the project so that the necessary dependency objects are passed to the constructor.
5. Spot check, commit, push, and notify QA.
6. Repeat with the next `new` call that is not inside a *Factory* object.

Find a new keyword

As in other steps, we begin this one by using our project-wide search facility to look for the `new` keyword in our class files using the following regular expression:

Search for:

```
new\s+
```

We have two kinds of creation to look for: one-time and repeated. How can we tell the difference? In general:

- If the instantiation is assigned to a property, and is never changed, it is most likely a one-time creation. Generally, we see this in constructors.
- If the instantiation occurs in a non-constructor method, it is most likely a repeated creation, because it occurs each time the method is called.

Extract One-Time creation to dependency injection

Let's say we find the *ItemsGateway* class listed above when we search for the `new` keyword, and encounter the constructor:

classes/ItemsGateway.php

```
1 <?php
2 class ItemsGateway
3 {
4 protected $db_host;
5 protected $db_user;
6 protected $db_pass;
7 protected $db;
8
9 public function __construct($db_host, $db_user,
$db_pass)
10 {
11 $this->db_host = $db_host;
12 $this->db_user = $db_user;
13 $this->db_pass = $db_pass;
14 $this->db = new Db($this->db_host, $this-
>db_user, $this->db_pass);
15 }
16 // ...
```

```
17 }
18 ?>
```

On examining the class, we find that `$this->db` is assigned once as a property. This appears to be a one-time creation. In addition, it appears that at least some of the existing constructor parameters are used only for the `Db` instantiation.

We proceed to remove the instantiation call entirely, along with the properties used only for the instantiation call, and replace the constructor parameters with a single Db parameter:

```
classes/ItemsGateway.php
1 <?php
2 class ItemsGateway
3 {
4 protected $db;
5
6 public function __construct(Db $db)
7 {
8 $this->db = $db;
9 }
10
11 // ...
12 }
13 ?>
```

Extract repeated creation to factory

If we find a repeated creation instead of a one-time creation, we have a different task to accomplish. Let's return to the *ItemsGateway* class, but this time we'll look at the `selectAll()` method.

```
classes/ItemsGateway.php
1 <?php
2 class ItemsGateway
3 {
4 protected $db;
5
6 public function __construct(Db $db)
7 {
8 $this->db = $db;
```

```
9 }
10
11 public function selectAll()
12 {
13 $rows = $this->db->query("SELECT * FROM items
ORDER BY id");
14 $item_collection = array();
15 foreach ($rows as $row) {
16 $item_collection[] = new Item($row);
17 }
18 return $item_collection;
19 }
20 }
21 ?>
```

We can see here that the `new` keyword occurs in a loop inside a method. This is clearly a case of repeated creation.

First, we extract the creation code to its own new class. Because the code creates an `Item` object, we will call the class *ItemFactory*. In it, we will create a method for returning new instances of `Item` objects:

```
classes/ItemFactory.php
1 <?php
2 class ItemFactory
3 {
4 public function newInstance(array $item_data)
5 {
6 return new Item($item_data);
7 }
8 }
9 ?>
```

Note

The only purpose of a *Factory* is to create new objects. It should not have any other functionality. It will be tempting to place other behavior in a `Factory` in a bid to centralize common logic. Resist this temptation!

Now that we have extracted the creation code to a separate class, we will modify the *ItemsGateway* to take an *ItemFactory* parameter, retain it in a property, and use the *ItemFactory* to create *Item* objects.

classes/ItemsGateway.php

```
1 <?php
2 class ItemsGateway
3 {
4 protected $db;
5
6 protected $item_factory;
7
8 public function __construct(Db $db, ItemFactory
$item_factory)
9 {
10 $this->db = $db;
11 $this->item_factory = $item_factory;
12 }
13
14 public function selectAll()
15 {
16 $rows = $this->db->query("SELECT * FROM items
ORDER BY id");
17 $item_collection = array();
18 foreach ($rows as $row) {
19 $item_collection[] = $this->item_factory-
>newInstance($row);
20 }
21 return $item_collection;
22 }
23 }
24 ?>
```

Change instantiation calls

Because we have changed the constructor signature, all the existing instantiations of *ItemsGateway* are now broken. We need to find all the places in the code where the *ItemsGateway* class is instantiated, and change the instantiations to pass a properly-constructed `Db` object and an *ItemFactory*.

To do so, we use our project-wide search facility to search using a regular expression for our changed class name:

Search for:

```
new\s+ItemsGateway\(
```

Doing so will give us a list of all instantiations in the project. We need to review each result and change it by hand to instantiate the dependencies and pass them to the *ItemsGateway*.

For example, if a page script from the search results looks like this:

page_script.php

```
1 <?php
2 // $db_host, $db_user, and $db_pass are defined
in the setup file
3 require 'includes/setup.php';
4
5 // ...
6
7 // create a gateway
8 $items_gateway = new ItemsGateway($db_host,
$db_user, $db_pass);
9
10 // ...
11 ?>
```

We need to change it to something more like this:

```
page_script.php
1 <?php
2 // $db_host, $db_user, and $db_pass are defined
in the setup file
3 require 'includes/setup.php';
4
5 // ...
6
7 // create a gateway with its dependencies
8 $db = new Db($db_host, $db_user, $db_pass);
9 $item_factory = new ItemFactory;
10 $items_gateway = new ItemsGateway($db,
$item_factory);
11
12 // ...
13 ?>
```

Do this for each instantiation of the changed class.

Spot Check, Commit, Push, Notify QA

Now that we have changed the class and the instantiations of the class throughout the codebase, we need to make sure our legacy application works. Again, we have no formal testing process in place, so we need to run or otherwise invoke the parts of the application that use the changed class and look for errors.

Once we feel sure that the application still operates properly, we commit the code, push it to our central repository, and notify QA that we are ready for them to test our new additions.

Do ... While

Search for the next `new` keyword in a class, and start the process all over again. When we find that `new` keywords exist only in *Factory* classes, our job is complete.

Common Questions

What About Exceptions and SPL Classes?

In this chapter, we concentrate on removing all use of the `new` keyword, except inside *Factory* objects. I believe there are two reasonable exceptions to this rule: *Exception* classes themselves, and certain built-in PHP classes, such as the SPL classes.

It would be perfectly consistent with the process described in this chapter to create an `ExceptionFactory` class, inject it into objects that throw exceptions, and then use the `ExceptionFactory` to create the `Exception` objects to be thrown. This strikes even me as going a bit too far. I think that `Exception` objects are a reasonable exception to the rule of no `new` outside `Factory` objects.

Similarly, I think built-in PHP classes are also frequently an exception to the rule. While it would be nice to have, say, an *ArrayObjectFactory* or an *ArrayIteratorFactory* to create *ArrayObject* and *ArrayIterator* classes that are provided by SPL itself, it may be a little too much. Creating these kinds of objects directly inside the objects that use them is usually all right.

However, we need to be careful. Creating a complex or otherwise powerful object like a `PDO` connection directly inside the class that needs it is probably overstepping our bounds. It's tough to describe a good rule of thumb here; when in doubt, err on the side of dependency injection.

What about Intermediary Dependencies?

Sometime we will discover classes that have dependencies, and the dependencies themselves have dependencies. These intermediary dependencies are passed to the outside class, which carries them along only so that the internal objects can be instantiated with them.

For example, say we have a `Service` class that needs an *ItemsGateway*, which itself needs a `Db` connection. Before removing `global` variables, the `Service` class might have looked like this:

classes/Service.php

```
1 <?php
2 class Service
3 {
4 public function doThis()
5 {
6 // ...
7 $db = global $db;
8 $items_gateway = new ItemsGateway($db);
9 $items = $items_gateway->selectAll();
10 // ...
11 }
12
13 public function doThat()
14 {
15 // ...
16 $db = global $db;
17 $items_gateway = new ItemsGateway($db);
18 $items = $items_gateway->selectAll();
19 // ...
20 }
21 }
22 ?>
```

After removing `global` variables, we are left with a `new` keyword, but we still need the `Db` object as a dependency for *ItemsGateway*:

classes/Service.php

```
1 <?php
2 class Service
3 {
4 protected $db;
5
6 public function __construct(Db $db)
7 {
8 $this->db = $db;
9 }
10
11 public function doThis()
12 {
13 // ...
14 $items_gateway = new ItemsGateway($this->db);
15 $items = $items_gateway->selectAll();
16 // ...
17 }
```

```
18
19 public function doThat()
20 {
21 // ...
22 $items_gateway = new ItemsGateway($this->db);
23 $items = $items_gateway->selectAll();
24 // ...
25 }
26 }
27 ?>
```

How do we successfully remove the `new` keyword here? The *ItemsGateway* needs a `Db` connection. The `Db` connection is never used by the `Service` directly; it is used only for building the *ItemsGateway*.

The solution in cases like this is to inject a fully-constructed *ItemsGateway*. First, we modify the `Service` class to receive its real dependency, the *ItemsGateway*:

classes/Service.php

```
1 <?php
2 class Service
3 {
4 protected $items_gateway;
5
6 public function __construct(ItemsGateway
$items_gateway)
7 {
8 $this->items_gateway = $items_gateway;
9 }
10
11 public function doThis()
12 {
13 // ...
14 $items = $this->items_gateway->selectAll();
15 // ...
16 }
17
18 public function doThat()
19 {
20 // ...
21 $items = $this->items_gateway->selectAll();
22 // ...
```

```
23 }
24 }
25 ?>
```

Second, throughout the entire legacy application, we change all instantiations of the *Service* to pass an *ItemsGateway*.

For example, a page script might have done this when using `global` variables everywhere:

page_script.php (globals)

```
1 <?php
2 // defines the $db connection
3 require 'includes/setup.php';
4
5 // creates the service with globals
6 $service = new Service;
7 ?>
```

And then we changed it to inject the intermediary dependency after removing globals:

page_script.php (intermediary dependency)

```
1 <?php
2 // defines the $db connection
3 require 'includes/setup.php';
4
5 // inject the Db object for the internal
ItemsGateway creation
6 $service = new Service($db);
7 ?>
```

But we should finally change it to inject the real dependency:

page_script.php (real dependency)

```
1 <?php
2 // defines the $db connection
3 require 'includes/setup.php';
4
5 // create the gateway dependency and then the
service
6 $items_gateway = new ItemsGateway($db);
```

```
7 $service = new Service($items_gateway);
8 ?>
```

Isn't this a lot of code?

I sometimes hear the complaint that using dependency injection means a lot of extra code to do the same thing as before.

It's true. Having a call like this, where the class manages its own dependencies internally.

Without dependency injection:

```
1 <?php
2 $items_gateway = new ItemsGateway;
3 ?>
```

This is obviously less code than using dependency injection by creating the dependencies and using `Factory` objects.

With dependency injection:

```
1 <?php
2 $db = new Db($db_host, $db_user, $db_pass);
3 $item_factory = new ItemFactory;
4 $items_gateway = new ItemsGateway($db,
$item_factory);
5 ?>
```

The real issue here, though, is not more code. The issues are more testable,more clear, and more decoupled.

In looking at the first example, how can we tell what *ItemsGateway* needs to operate? What other parts of the system will it affect? It's very difficult to tell without examining the entire class and looking for `global` and `new` keywords.

In looking at the second example, it is very easy to tell what the class needs to operate, what we can expect it to create, and what parts of

the system it interacts with. These things additionally make it easier to test the class later.

Should a factory create collections?

In the examples above, our `Factory` class only creates a single `newInstance()` of an object. If we regularly create collections of objects, it may be reasonable to add a `newCollection()` method to our `Factory`. For example, given our *ItemFactory* above, we may do something like the following:

classes/ItemFactory.php

```
1 <?php
2 class ItemFactory
3 {
4 public function newInstance(array $item_data)
5 {
6 return new Item($item_data);
7 }
8
9 public function newCollection(array $items_data)
10 {
11 $collection = array();
12 foreach ($items_data as $item_data) {
13 $collection[] = $this->newInstance($item_data);
14 }
15 return $collection;
16 }
17 }
18 ?>
```

We may go so far as to create an `ItemCollection` class for the collection instead of using an array. If so, it would be reasonable to use a `new` keyword inside our `ItemFactory` to create the `ItemCollection` instance. (The `ItemCollection` class is omitted here).

classes/ItemFactory.php

```
1 <?php
2 class ItemFactory
3 {
4 public function newInstance(array $item_data)
5 {
```

```
6 return new Item($item_data);
7 }
8
9 public function newCollection(array $item_rows)
10 {
11 $collection = new ItemCollection;
12 foreach ($item_rows as $item_data) {
13 $item = $this->newInstance($item_data);
14 $collection->append($item);
15 }
16 return $collection;
17 }
18 }
19 ?>
```

Indeed, we may wish to have a separate *ItemCollectionFactory*, using an injected *ItemFactory* to create Item objects, with its own `newInstance()` method to return a new *ItemCollection*.

There are many variations on the proper use of `Factory` objects. The key is to keep object creation (and related operations) separate from object manipulation.

Can we automate all these Injections?

All the dependency injection we have been doing so far has been manual injection, where we create the dependencies ourselves and then inject them as we create the objects we need. This can be a tedious process. Who wants to create a `Db` object over and over again just so it can be injected into a variety of `Gateway` classes? Isn't there some way to automate that?

Yes, there is. It is called a `Container`. A `Container` may go by various synonyms indicating how it is to be used. A Dependency Injection `Container` is intended to be used always-and-only outside the non-`Factory` classes, whereas an identical `Container` implementation going by the name `Service Locator` is intended to be used `inside` non-`Factory` objects.

Using a `Container` brings distinct advantages:

- We can create shared services that are instantiated only when they are called. For example, the `Container` can house a `Db` instance that only gets created when we ask the `Container` for a database connection; the connection is created once and then reused over and over again.
- We can put complex object creation inside the `Container`, where objects that need multiple services for their constructor parameters can retrieve those services from the `Container` inside their own creation logic.

But using a `Container` has disadvantages as well:

- We have to drastically change how we think about our object creation, and where those objects live in the application. In the end this is a good thing, but it can be trouble in the interim.
- A `Container` used as a Service Locator replaces our `global` variables with a fancy new toy that has many of the same problems as `global`. The `Container` hides dependencies because it is called only from inside the class that needs dependencies.

At this stage of modernizing our legacy application it can be very tempting to start using a `Container` to automate dependency injection for use. I suggest that we do not add one just now, because so much of our legacy application remains to be modernized. We will add one eventually, but it will be as the very last step of our modernization process.

Review and next steps

We have now made great strides in modernizing our application. Removing `global` and `new` keywords in favor of dependency injections has already improved the quality of the codebase and made tracking down bugs a lot easier, if only because modifying a variable over here no longer causes a variable over there to be affected at a distance. Our page scripts might be somewhat longer, because we have to create the dependencies, but we can now see exactly what parts of the system we are interacting with much more clearly.

Our next step is to examine our newly refactored classes and start writing tests for them. That way, when we start making changes to the classes, we will know if we broke any previously existing behavior.

Chapter 7. Write Tests

At this point, our legacy application has been partially modernized so that we have all existing classes in a central location. Those classes now enjoy freedom from `global` and `new` using dependency injection. The proper thing to do now is to write tests for these classes so that, if we ever need to change them, we know that their pre-existing behavior remains intact.

Fighting test resistance

We are probably not eager to spend time writing tests right now. We don't want to lose the forward momentum we are feeling. Just as we believe we are making some real progress, stopping to write tests feels like make-work. It takes away from the joy of making yet another series of improvements to the awful codebase that we have been suffering under for so long.

The resistance to writing tests is understandable. I myself was only a slow convert to automated testing. If one is not used to it, the act of writing tests feels alien, unfamiliar, challenging, and unproductive. It is very easy to say, I can see the code is working, because the application is working.

And yet if we do not write tests, we are condemning ourselves to continued rounds of suffering later. We are enabling a particular kind of awfulness in our legacy application: that feeling of dread we feel when we change one part of the application, not knowing what other parts of the application are going to break as a result.

So while it may be true that writing tests sucks, it is also true that "having written tests" is awesome. It is awesome because, as we make changes to our classes, we can run an automated test suite, and it will tell us immediately if anything has broken after a change.

The way of Testivus

Even if we are already familiar with writing tests, all of the dogma that surrounds testing can be intimidating:

- Do not interact with the file system; build a virtual file system instead.
- Do not interact with the database; build a set of data fixtures instead.
- Rewrite your classes to use interfaces instead of concrete classes, and write test doubles for all the dependencies.

These are the kinds of dogmatic commands that make testing seem like an insurmountable challenge. Surely we can build our tests later when everything else is done! The problem is that there will never be a point when everything else is done, and so the tests will never come into being.

As an antidote to the dogma of testing commandments, I suggest following *The Way Of Testivus* (http://www.artima.com/weblogs/viewpost.jsp?thread=203994) instead. The core message of *The Way of Testivus* is More testing karma, less testing dogma.

These are the major points we need to take from *Testivus* in regard to modernizing our legacy application:

- The best time to test is when the code is fresh.
- Write the test that needs to be written.
- An imperfect test today is better than a perfect test someday.
- Write the test you can today.

The code in the classes is stale. That code is part of a legacy application, after all. But now that we have spent so much time reorganizing the classes and removing their `global` and `new` keywords in favor of dependency injection, the code in those classes

has been made fresh again in our minds. Now is the time to write the tests for these classes, while their operation is still in recent memory.

We should not get hung up on writing proper unit tests that adhere to every commandment of testing dogma. Instead, we should write the best test we can, even if the test is imperfect:

- If we can write a characterization test that only checks the output as-it-is, then we should do so.
- If we can write a functional or integration test that interacts with the database, network, or file system, then we should do so.
- If we can write a loose unit test that combines concrete classes, then we should do so.
- If we can write a strict unit test that uses test doubles to fully isolate the tested class, then we should do so.

An imperfect test can be perfected as we become more adept at testing. A test that does not exist cannot be perfected at all.

We will write the tests we can, while we can, as soon as we can. Waiting to write tests will only increase the inertia against writing tests. The code will grow more stale in our mind, making the tests harder to write. Writing tests today will give us a sense of accomplishment and increase our inertia in favor of writing tests.

Setting up a test suite

It is not within the scope of this book to fully explain the technique and methodology of writing tests. Instead, we will review a very brief summary of the process involved in setting up the automated test suite and writing a simple test. For a more thorough treatment of testing in PHP, refer to *The Grumpy Programmer's PHPUnit Cookbook* (http://grumpy-phpunit.com/)by *Chris Hartjes*.

Install PHPUnit

There are many different testing systems in PHP land, but the most commonly used one is PHPUnit. We need to install PHPUnit on our development and testing servers in order to write and execute our tests. The full installation instructions are at the PHPUnit site.

One easy way to install PHPUnit is via Composer:

```
$ composer global require phpunit/phpunit=~4.5
```

Another way is to install a `.phar` of PHPUnit directly:

```
$ wget https://phar.phpunit.de/phpunit.phar
$ chmod +x phpunit.phar
$ sudo mv phpunit.phar /usr/local/bin/phpunit
```

Create a tests/ directory

Once we have PHPUnit installed, we need to create a `tests/` directory in our legacy application. The name and location do not matter so much as the fact that the purpose and place are obvious. The most obvious place is probably at the root of the legacy application, though it should not be directly accessible by browsers.

Within the `tests/` directory, we need to create a subdirectory named for our central class directory location. If all our application classes are in a directory named `classes/`, then we should have a `tests/classes/` directory. The idea is for our testing structure to mimic the structure of our application classes.

In addition to the `tests/classes/` subdirectory, the `tests/` directory should contain two files. The first is a `bootstrap.php` file that PHPUnit will execute when it runs. Its purpose is to help set up the execution environment for the tests. By default, PHPUnit will not use the application autoloader code, so creating and registering the autoloader is a classic use for the `bootstrap.php` file. Here is an example using the autoloader from an earlier chapter:

tests/bootstrap.php

```
1 <?php
2 require "../classes/Mlaphp/Autoloader.php";
3 $loader = new \Mlaphp\Autoloader;
4 spl_autoload_register(array($loader, 'load'));
5 ?>
```

Also in the `tests/` directory, we need to create a `phpunit.xml` file. This tells PHPUnit how to bootstrap itself and where the tests are located:

tests/phpunit.xml

```
1 <phpunit bootstrap="./bootstrap.php">
2 <testsuites>
3 <testsuite>
4 <directory>./classes</directory>
5 </testsuite>
6 </testsuites>
7 </phpunit>
```

After creating the `tests/` directory and its contents, our legacy application directory structure should look something like this:

/path/to/app/

```
classes/          # our central class directory
location
Auth.php          # class Auth { ... }
Db.php            # class Db { ... }
Mlaphp/
Autoloader.php    # A hypothetical autoloader class
Role.php          # class Role { ... }
User.php          # class User { ... }
foo/
bar/
baz.php           # a page script
includes/         # a common "includes" directory
setup.php         # setup code
index.php         # a page script
tests/            # tests directory
bootstrap.php     # phpunit bootstrap code
classes/          # test cases
phpunit.xml       # phpunit setup file
```

Pick a class to test

Now that we have a `tests/` directory in place, we can actually write a test for one of our application classes. The easiest way to get started is to pick a class that has no dependencies. We should be familiar enough with the codebase at this point that we know which classes have dependencies and which do not. If we cannot find a class that has no dependencies, we should pick the one with the fewest dependencies, or with the least-complex dependencies.

What we want to do here is *start small* and get some early wins. Each win will give us the drive and motivation to continue with larger, more complex tests. These smaller wins will accumulate into a final big win: a fully-tested set of classes.

Write a test case

Let's say we have picked a class named `Foo` that has no dependencies, and that it has a method called `doSomething()`. We are now going to write a test for this class method.

First, we create a skeleton test file in our `tests/classes/` directory. Its location should mimic the location of the class being tested. We add `Test` to the end of the class name, and extend `PHPUnitFramework_TestCase_` so that we have access to the various `assert*()` methods in the testing class:

tests/classes/FooTest.php

```
1 <?php
2 class FooTest extends
\PHPUnit_Framework_TestCase
3 {
4 }
5 ?>
```

If we try to run our tests now with `phpunit`, the test will fail because it has no test methods in it:

```
tests $ phpunit
PHPUnit 3.7.30 by Sebastian Bergmann.
Configuration read from tests/phpunit.xml
F
Time: 45 ms, Memory: 2.75Mb
There was 1 failure:
1) Warning
No tests found in class "FooTest".
FAILURES!
Tests: 1, Assertions: 0, Failures: 1.
tests $
```

Believe it or not, this is just fine! As *The Way Of Testivus* tells us, we rejoice when our tests pass, and we rejoice when they fail. The failure here tells us that PHPUnit has successfully found our test class, but did not find any tests in that class. That tells us what to do next.

The next step is to add a test method for a public method of the class being tested. All test methods begin with the word `test`, so we will test the `doSomething()` method using a method named `testDoSomething()`. In it, we will create an instance of the `_Foo_` class, invoke its public `doSomething()` method, and assert that its actual output is the same as what we expect it to be:

tests/classes/FooTest.php

```
1 <?php
2 class FooTest extends
\PHPUnit_Framework_TestCase
3 {
4 public function testDoSomething()
5 {
6 $foo = new Foo;
7 $expect = 'Did the thing!';
8 $actual = $foo->doSomething();
9 $this->assertSame($expect, $actual);
10 }
11 }
12 ?>
```

Now we can run our test suite again with `phpunit`. As long as the `doSomething()` method returns the string `Did the thing!` then our test

will pass.

```
tests $ phpunit
PHPUnit 3.7.30 by Sebastian Bergmann.
Configuration read from tests/phpunit.xml
.
Time: 30 ms, Memory: 2.75Mb
OK (1 test, 1 assertion)
tests $
```

We rejoice because our tests have passed!

If `doSomething()` ever returns anything different, then the test will fail. This means that if we change `doSomething()` in the course of our following work, we will know that its behavior has changed. We will rejoice in its failure, knowing that we have caught a bug before it went into production, and then fix the code until all the tests pass.

Do ... While

After we write a passing test, we commit it to revision control and push it to our central repository. We continue writing tests for each public method in the application class, committing and pushing as we go. When all the public methods in the application class have passing tests, we pick another class to test, and begin again with a new test class.

Common Questions

Can we skip this step and do it later?

No.

Come On, Really, Can We Do This Later?

Look, I get it. I really do. Testing feels unrewarding at this point in our modernization process. If the entire rest of the chapter has not

convinced you of the benefit of tests, then there's not much else I can say to convince you now. If you want to skip this step, you're going to skip it no matter what advice you read here.

So let us assume that our reasons for avoiding tests at this point are perfectly sensible and well-adjusted to our particular context. With that in mind, let's take a look at some things we can do to get those tests done, if not now then over the course of the project. Go on to the next chapter (unadvisedly!) but then commit to one or more of the following options:

1. Complete at least one new test class per day.
2. Each time we use a method in the codebase, check to see if there is a test for it. If there is not, write one before using the method.
3. As we fix a bug or build a feature, create a list of methods used over the course of the task, then write tests for those methods when the task is complete.
4. When we add a new class method, write a corresponding test for it.
5. Delegate the writing of tests to another developer, perhaps a junior developer. Then we can have the "fun" of modernizing, and the junior developer can have the perceived boring work of writing tests, but beware ... pretty soon, the junior developer will know more about the codebase than we do.

These options allow us to build up a test suite and still feel like we are making progress elsewhere. Creating an automated test suite is a non-negotiable aspect of modernizing our legacy application. Write tests now, or write them as we go, but write them, sooner rather than later.

What about hard-to-test classes?

Even with dependency injection in place, some of the classes in the legacy application are going to be hard to write tests for. There are lots of ways in which the classes can be hard to test and I cannot do

the solutions justice in this book. Instead, please refer to these works:

- *Working Effectively With Legacy Code* by *Michael Feathers*. The examples throughout are in Java, but the situations are similar to those in PHP. Feathers shows how to break dependencies, introduce seams, and otherwise improve the testability of legacy classes.
- Refactoring by Fowler et al. This one also uses Java in the examples, but thanks to Adam Culp, we have the same examples converted to PHP. As with Fowler's Patterns of Enterprise Application Architecture, the Refactoring book will give you a vocabulary to describe things you probably already know how to do, as well as introduce you to new techniques.

The information in these publications will help us improve the quality of our classes without changing the behavior of the classes.

What about our earlier characterization tests?

The tests we write as a result of this chapter are probably not a replacement for any existing characterization tests from the chapter on Prerequisites. Having the two sets of tests is likely to be a blessing, not a curse. At some point, the characterization tests may end up being converted to acceptance tests for use by the QA team. Until then, run both sets of tests from time to time.

Should we test private and protected methods?

Probably not. There are dogmatic reasons for this that I will not go into here, but the short version is this: tests that inspect too deeply the internal workings of a class become difficult to work with.

Instead, we should test only the public methods on our classes. Any behavior exposed by these methods is likely to be the only behavior we care about. There are some exceptions to this rule, but at this stage in our testing career, the exceptions are less important than the rule.

Can we change a test after we write it?

The time will come when we need to change the existing behavior of an application class method. In these cases, it is all right to change the related test to suit the new behavior. However, when we do so, we must be sure to run the entire test suite, not just the tests for that application class. Running the entire test suite will help us make sure that the change does not break behavior in other classes.

Do we need to test Third-party libraries?

If our legacy application uses third-party libraries, they may already come with tests. We should run these from time to time.

If the third-party libraries do not come with tests, we may choose to write some, depending on our priorities. If we are dependent on the library behaving the same way between upgrades, it would be wise to write some tests of our own to make sure that the expected behaviors remain in place.

Building tests for a third-party library may be difficult if it is not written in an easily-testable way. If the library is free software or open source, perhaps this is an opportunity to contribute back to the project. However, our main priority is probably our own legacy application, not third-party libraries.

What about code coverage?

Code coverage is a report given by PHPUnit to tell us how many lines of code we have tested. (Strictly speaking, it tells us the number of statements that have been tested).

A particular case may only test part of a class, or part of a method, and leave some of the code untested. The parts that are tested are called the covered portions of the code and the parts that are not tested are uncovered.

It is mostly the uncovered parts of the codebase that we need to worry about. If anything in the uncovered code changes, the tests won't detect it, and so we may be open to bugs and other regressions.

If we can, we should discover the code coverage of our tests early and often. These coverage reports will help us determine what needs to be tested next, and which parts of the codebase need to be refactored so they are easier to test.

More code coverage is better. However, reaching 100% line coverage is probably not feasible (and, indeed, is not the final goal, that being 100% condition/decision coverage among other things). If we can reach 100% coverage, though, we should.

For more on this topic, please review the PHPUnit documentation on code coverage at https://phpunit.de/manual/3.7/en/code-coverage-analysis.html.

Review and next steps

When we have completed writing our tests, as briefly outlined in this chapter, we will have created a great trap for future bugs. Each time we run our tests, any changes to the expected behavior will stand out as failures for us to correct. This ensures that as we continue refactoring, we will be doing more good than harm to our legacy codebase as a whole.

In addition, because we now have a working test suite, we can add tests for any new behavior extracted from our legacy codebase into our application classes. Each time we create a new application class method, we will also create a passing test for that method. Each time we modify an application class method, we will run the test suite so that we can find bugs and breaks before they make it into production. We will rejoice when our tests pass, and we will rejoice when they fail; each outcome is a positive sign when it comes to modernizing our legacy application.

With that, we can continue our modernization process. The next step is to extract our data retrieval and persistence behaviors from the page scripts and into a series of classes. Generally, this means moving all our SQL calls to a separate layer.

Chapter 8. Extract SQL statements to Gateways

Now that we have moved all our class-oriented functionality to a central directory location (and have a reasonable test suite for those classes) we can begin extracting more logic from our page scripts and place that logic into classes. This will have two benefits: first, we will be able to keep the various concerns of the application separated; second, we will be able to test the extracted logic so that any breaks will be easy to notice before we deploy into production.

The first of these extractions will be to move all SQL-oriented code to its own set of classes. For our purposes, SQL is a stand-in for any system of reading from and writing to a data store. This may be a no-SQL system, a CSV file, a remote resource, or anything else. We will concentrate on SQL-oriented data stores in this chapter because they are so common throughout legacy applications, but the principles apply to any form of data storage.

Embedded SQL Statements

Right now, our page scripts (and probably some of our classes) interact with a database directly with embedded SQL statements. For example, a page script might have some logic like this:

page_script.php

```
1 <?php
2 $db = new Db($db_host, $db_user, $db_pass);
3 $post_id = $_GET['post_id'];
4 $stm = "SELECT * FROM comments WHERE post_id =
$post_id";
5 $rows = $db->query($stm);
6 foreach ($rows as $row) {
7 // output each row
8 }
9 ?>
```

The problems with using embedded SQL strings are numerous. Among other things, we want to:

- Test the SQL interactions in isolation from the rest of the code
- Reduce the number of repeated SQL strings throughout the codebase
- Collect related SQL commands for generalization and reuse
- Isolate and remove security flaws such as SQL injection

These problems and more lead us to the conclusion that we need to extract all our SQL-related code to an SQL layer, and replace the embedded SQL logic with calls to our SQL-related class methods. We will do this by creating a series of `Gateway` classes. The only thing these `Gateway` classes will do is get data from, and send data back to, our data sources.

The `Gateway` classes in this chapter are technically more like table data gateways. However, you may choose to set up any kind of `Gateway` that is appropriate for your data source.

The extraction process

In general, this is the process we will follow:

1. Search the entire codebase for SQL statements.
2. For each statement not already in a `Gateway`, move the statement and relevant logic to a related `Gateway` class method.
3. Write a test for the new `Gateway` method.
4. Replace the statement and relevant logic in the original file with a call to the `Gateway` class method.
5. Test, commit, push, and notify QA.
6. Repeat with the next SQL statement that is outside a `Gateway` class.

Search for SQL statements

As in previous chapters, we employ a project-wide search function here. Use a regular expression like the following to identify where SQL statement keywords are located in the codebase:

Search for:

```
(SELECT|INSERT|UPDATE|DELETE)
```

We may find that our codebase uses other SQL commands as well. If so, we should include them in the search expression.

It will be easier on us if the codebase consistently uses only one case, whether upper or lower, for SQL keywords. However, this is not always the convention in legacy code. If our codebase is inconsistent regarding the case for SQL keywords, and our project-wide search facility has a case-insensitive option, we should use that option in this search. Otherwise, we need to expand the search terms to include lower-case (and perhaps mixed-case) variations of the SQL keywords.

Finally, the search results are likely to include false positives. For example, narrative text such as "Select one of the following options" will be in the result list. We will need to examine the results individually to determine if they are SQL statements or only narrative text.

Move SQL to a Gateway class

The task of extracting the SQL to a `Gateway` is detail-oriented and context-specific. The structure of the legacy codebase itself will determine one or more correct approaches to this task.

To begin with, extracting a plain old SQL statement like the following seems straightforward:

```
1 <?php
2 $stm = "SELECT * FROM comments WHERE post_id =
$post_id";
```

```
3 $rows = $db->query($stm);
4 ?>
```

But it turns out we need to make a lot of decisions even with this simple example:

- What should we name the `Gateway` class and method?
- How should we deal with parameters to the query?
- How do we avoid security flaws?
- What is the proper return value?

Namespace and Class names

To determine our namespace and class names, the first thing we need to decide is whether to organize by layer or by entity.

- If we organize by implementation layers, the top-level namespaces for our class might be `Gateway` or `DataSource\Gateway`. This naming arrangement structures the classes by their operational purpose in the codebase.
- If we organize by domain entities, the top-level namespaces would be something like `Comments`, or even `Domain\Comments`. This naming arrangement structures the classes by their purpose within the business logic domain.

The legacy codebase is likely to dictate which way to go. If there is code already organized by one or the other, then it might be better to continue in the established structure rather than redoing the existing work. We want to avoid setting up conflicting or inconsistent organizational structures in the codebase.

Of the two, I recommend organizing by domain entities. I find it more sensible to collect functionality related to specific domain entity types within their relevant namespaces, than to spread operational implementations across several namespaces. We can also further segregate the implementation pieces within the specific domain feature in a way that is not easily done when organizing by layer.

To reflect my domain entity bias, the examples in the rest of this chapter will be structured along domain lines, not implementation layers.

Once we have an organizing principle for our `Gateway` classes, we can easily find good class names. For example, our comments-related `Gateway` in PHP 5.3 and later might be named `Domain\Comments\CommentsGateway`. If we are using PHP 5.2 or earlier, we will need to avoid namespaces proper and use underscores in the class name instead; e.g., `Domain_Comments_CommentsGateway`.

Method names

Choosing an appropriate method name, however, is a little more difficult. Once again, we should look to the existing legacy codebase for a convention. The common idiom may be to `get()` the data, `find()` the data, `fetch()` the data, `select()` the data, or something else entirely.

We should stick with any existing naming convention as much as possible. While the method name itself does not matter, consistency-of-naming does. Consistency will make it easier for us to look at calls to the `Gateway` object and understand what is happening without having to read the underlying method code, and to search the codebase for data-access calls.

If our legacy codebase does not reveal a consistent pattern, it is up to us to choose a consistent naming convention for the new `Gateway` methods. Because the `Gateway` classes are supposed to be a simple layer to wrap SQL calls, the examples in this chapter will use method names like `select`, `insert`, etc. to identify the behavior being wrapped.

Finally, the method name should probably indicate the kind of `select()` being performed. Are we selecting one record or or all of them? Are we selecting by specific criteria? Are there other

considerations on the query? These and other questions will give us hints on how to name the `Gateway` methods.

An initial Gateway class method

When extracting logic to a class method, we should be careful to follow all the lessons we learned about dependency injection in prior chapters. Among other things, this means: no use of globals, replacing superglobals with a `Request` object, no use of the `new` keyword outside `Factory` classes, and (of course) injecting objects via the constructor as needed.

Given the above naming principles, and the original `SELECT` statement to retrieve comment rows, we may build a `Gateway` that looks something like this:

classes/Domain/Comments/CommentsGateway.php

```
1 <?php
2 namespace Domain\Comments;
3
4 class CommentsGateway
5 {
6 protected $db;
7
8 public function __construct(Db $db)
9 {
10 $this->db = $db;
11 }
12
13 public function selectAllByPostId($post_id)
14 {
15 $stm = "SELECT * FROM comments WHERE post_id =
{$post_id}";
16 return $this->db->query($stm);
17 }
18 }
19 ?>
```

This is practically an exact copy of the logic from the original page script. It leaves at least one major issue in place, though: it uses the

input parameter directly in the query. This leaves us open to SQL injection attacks.

> ## Note
>
> **What Is SQL Injection**
>
> The classic XKCD comic regarding Little Bobby Tables should help to illustrate the problem. The input parameter being used directly in a database query is maliciously formed to change the query so that it damages or otherwise exploits the database.

Defeating SQL Injection

When we create our `Gateway` methods, we should never assume the parameter values are safe. It doesn't matter if we expect the parameters to be hard-coded to constant values on every call, or otherwise guaranteed to be safe. At some point, someone is going to change a bit of code that calls the `Gateway` method and we will have a security issue. Instead, we need to treat every parameter value as unsafe, and deal with it accordingly.

Thus, to defeat SQL injection attempts, we should do one of three things with every query in our `Gateway` methods (indeed, in any SQL statement anywhere in the codebase):

1. The best solution is to use prepared statements and parameter binding instead of query string interpolation.
2. The second-best solution is to use the database layer's "quote-and-escape" mechanism on each parameter before interpolating it into the query string.
3. The third-best solution is to escape each of the input parameters before interpolating them into the query string.

> ## Tip

> Alternatively, we can avoid the problem of strings entirely by casting expected numeric values to `int` or `float`.

Let's examine the third-best solution first, since it is more likely to already be present in our legacy codebase. We use the database's `escape` functionality to escape each parameter before we use it in a query string, and quote it appropriately for the database. Thus, we can rewrite the `selectAllByPostId()` method like so, assuming a MySQL database:

```
<?php
2 public function selectAllByPostId($post_id)
3 {
4 $post_id = "'" . $this->db->escape($post_id) .
"'";
5 $stm = "SELECT * FROM comments WHERE post_id =
{$post_id}";
6 return $this->db->query($stm);
7 }
8 ?>
```

Escaping values for interpolation into strings is a third-best solution for several reasons. The main one is that the escaping logic is sometimes insufficient. Something like `mysql_escape_string()` function is simply not good enough for our purposes here. Even the `mysql_real_escape_string()` method has a flaw that will allow attackers to succeed at SQL injection attempts depending on the current character set. However, this may be the only option available to the underlying database driver.

The second-best solution is a variation of escaping called quote-and-escape. This feature, available only via the `PDO::quote()` method, is safer than escaping because it additionally wraps the value in quotes and deals with the proper character set automatically. This avoids the character set mismatch issues inherent to just escaping and adding quotes on our own.

A rewritten `selectAllByPostId()` method might look like this with a `Db` object that exposes the `PDO::quote()` method:

```
<?php
2 public function selectAllByPostId($post_id)
3 {
4 $post_id = $this->db->quote($post_id);
5 $stm = "SELECT * FROM comments WHERE post_id =
{$post_id}";
6 return $this->db->query($stm);
7 }
8 ?>
```

This is a safe approach when we remember to use it. The issue here, of course, is that if we add a parameter to the method, we may forget to quote it, and then we are vulnerable to SQL injection again.

Finally, the best solution: prepared statements and parameter binding. These are only available via PDO (which works with almost all databases) and the `mysqli` extension. Each has its own variations on how to handle statement preparation. We will use `PDO` style examples here.

Instead of interpolating values into query strings, we use named placeholders to indicate where parameters should be placed in a query string. We then tell `PDO` to prepare the string as a `PDOStatement` object, and bind values to the named placeholders when we execute the query through that prepared statement. `PDO` automatically uses safe representations of the parameter values, making us secure against SQL injection attacks.

Here is an example of what a rewrite would look like using a `Db` object that exposes the `PDO` statement preparation logic and execution:

```
1 <?php
2 public function selectAllByPostId($post_id)
3 {
4 $stm = "SELECT * FROM comments WHERE post_id =
:post_id";
```

```
5 $bind = array('post_id' => $post_id);
6
7 $sth = $this->db->prepare($stm);
8 $sth->execute($bind);
9 return $sth->fetchAll(PDO::FETCH_ASSOC);
10 }
11 ?>
```

The great benefit here is that we never use the parameter variables in the query string. We always and only use named placeholders, and bind the placeholders to the parameter values into the prepared statement. This idiom makes it clear when we are improperly using interpolated variables, and `PDO` automatically complains if there are extra or missing bound values, so the chances of accidentally making an insecure change are greatly reduced.

Write a test

Now it is time to write a test for our new class method. The test we write at this point is going to be less perfect than we would like because we need to interact with the database. However, an imperfect test is better than no test. As *The Way Of Testivus* tells us, we write the test we can at the time we can.

The test for our new `Gateway` method might look something like this:

tests/classes/Domain/Comments/CommentsGatewayTest.php

```
1 <?php
2 namespace Domain\Comments;
3
4 use Db;
5
6 class CommentsGatewayTest
7 {
8 protected $db;
9
10 protected $gateway;
11
12 public function setUp()
13 {
14 $this->db = new Db('test_host', 'test_user',
```

```
'test_pass');
15 $this->gateway = new CommentsGateway($this-
>db);
16 }
17
18 public function testSelectAllByPostId()
19 {
20 // a range of known IDs in the comments table
21 $post_id = mt_rand(1,100);
22
23 // get the comment rows
24 $rows = $this->gateway-
>selectAllByPostId($post_id);
25
26 // make sure they all match the post_id
27 foreach ($rows as $row) {
28 $this->assertEquals($post_id, $row['post_id']);
29 }
30 }
31 }
32 ?>
```

Now we run our test suite to see if the test passes. If it does, we rejoice and move on! If it does not, we continue to refine the `Gateway` method and the related test until both are working properly.

Tip

Perfecting Our Tests

As noted earlier, this is very much an imperfect test. Among other things, it depends on a working database connection, and on having data seeded into the database in the first place. By depending on the database, we are dependent on it being in a correct state. If the database does not have the correct data in it, then the test will fail. The failure will come not from the code, which is what we are testing, but from the database, which is mostly beyond our control. One opportunity for improving the test is to change the `Gateway` class to depend on a `DbInterface` instead of a concrete `Db` class. We would then create a `FakeDb` class for testing purposes that implements the `DbInterface`, and

inject a `FakeDb` instance into the `Gateway` instead of a real `Db` instance. Doing so would give us greater insight into the correctness of the SQL query string along with greater control over the the data being returned to the `Gateway`. Above all, it would decouple the test from its dependency on a working database. For now, in the interest of expediency, we will use the imperfect test.

Replace the original code

Now that we have a working and tested `Gateway` method, we replace the original code with a call to the `Gateway`. Whereas the old code looked like this:

page_script.php (before)

```
1 <?php
2 $db = new Db($db_host, $db_user, $db_pass);
3 $post_id = $_GET['post_id'];
4 $stm = "SELECT * FROM comments WHERE post_id =
$post_id";
5 $rows = $db->query($stm);
6 foreach ($rows as $row) {
7 // output each row
8 }
9 ?>
```

The new version would look like this:

page_script.php (after)

```
1 <?php
2 $db = new Db($db_host, $db_user, $db_pass);
3 $comments_gateway = new CommentsGateway($db);
4 $rows = $comments_gateway-
>selectAllByPostId($_GET['post_id']);
5 foreach ($rows as $row) {\
6 // output each row
7 }
8 ?>
```

Note that we have done little to modify the operational logic. For example, we have not added error checking that was not previously present. The furthest extent of our modification has been to secure the query against SQL injection via prepared statements.

Test, Commit, Push, Notify QA

As in previous chapters, we now need to spot check the legacy application. Although we have a unit test for the new `Gateway` method, we still need to spot check the part of the application we have modified. If we prepared a characterization test earlier that covers this part of our legacy application, we can run that now. Otherwise, we can do this by browsing to or otherwise invoking the changed part of the application.

Once we are satisfied that we have successfully replaced the embedded SQL with a call to our new `Gateway` method, we commit the changes, including our tests, to revision control. We then push to the central repository and notify the QA team of our changes.

Do ... While

With that complete, we search the codebase again for uses of SQL keywords to indicate embedded query strings. Where they exist outside of a `Gateway` class, we continue extracting the queries to an appropriate `Gateway`. Once all the SQL statements have been moved to `Gateway` classes, we're done.

Common Questions

What about INSERT, UPDATE, and DELETE Statements?

So far we have looked exclusively at `SELECT` statements since they are likely to be the most common case in our legacy codebase. However, there will be a significant number of `INSERT`, `UPDATE`, `DELETE`,

and perhaps other statements as well. These are treated essentially the same as the `SELECT` for purposes of extraction to a `Gateway`, but there are some minor differences.

In particular, `INSERT` and `UPDATE` statements may contain a large number of parameters indicating the column values to be inserted or updated. Adding too many parameters to the extracted `Gateway` method signature will make it difficult to work with.

In these cases, we can use a data array to indicate the column names and their corresponding values. We need to make sure that we are inserting or updating only the correct columns, though.

For example, say we start with the following code in a page script to save a new comment with the name of the commenter, the comment body, the IP address of the commenter, and the post ID to which the comment is attached:

page_script.php

```
1 <?php
2 $db = new Db($db_host, $db_user, $db_pass);
3
4 $name = $db->escape($_POST['name']);
5 $body = $db->escape($_POST['body']);
6 $post_id = (int) $_POST['id'];
7 $ip = $db->escape($_SERVER['REMOTE_ADDR']);
8
9 $stm = "INSERT INTO comments (post_id, name,
body, ip) "
10 .= "VALUES ($post_id, '{$name}', '{$body}',
'{$ip}'";
11
12 $db->query($stm);
13 $comment_id = $db->lastInsertId();
14 ?>
```

When we extract this to a method in our `CommentsGateway`, we could have a parameter for each column value being inserted. In this case there are only four columns, but if there were a dozen, the method signature would be harder to deal with.

As an alternative to one parameter per column, we could pass an array of data as a single parameter and then work with that inside the method. This example of using a data array includes a prepared statement with placeholders to defeat SQL injection attacks:

```
1 <?php
2 public function insert(array $bind)
3 {
4 $stm = "INSERT INTO comments (post_id, name,
body, ip) "
5 .= "VALUES (:post_id, :name, :body, :ip)";
6 $this->db->query($stm, $bind);
7 return $this->db->lastInsertId();
8 }
9 ?>
```

Once we have a method like this in the `CommentsGateway`, we can modify the original code to read more like the following:

page_script.php

```
1 <?php
2 $db = new Db($db_host, $db_user, $db_pass);
3 $comments_gateway = new CommentsGateway($db);
4
5 $input = array(
6 'name' => $_POST['name'],
7 'body' => $_POST['body'],
8 'post_id' => $_POST['id'],
9 'ip' => $_SERVER['REMOTE_ADDR'],
10 );
11
12 $comment_id = $comments_gateway-
>insert($input);
13 ?>
```

What about Repetitive SQL strings?

One thing we will probably encounter during this process is a large amount of repetition, or repetition with variation, in the query strings throughout our legacy application.

For example, we may find a comments-related query elsewhere in our legacy application that looks like this:

```
1 <?php
2 $stm = "SELECT * FROM comments WHERE post_id =
$post_id LIMIT 10";
3 ?>
```

The query string is identical to the example code from the beginning of this chapter, except it has a `LIMIT` clause attached. Should we create an entirely new method for this query, or do we modify an existing method?

This is the sort of thing that requires professional judgment and familiarity with the codebase. In this case, modification seems to be reasonable, but in other situations the difference might be great enough to warrant an entirely new method.

If we choose to modify an existing method in the `CommentsGateway`, we might rewrite `selectAllByPostId()` to include an optional `LIMIT`:

```
1 <?php
2 public function selectAllByPostId($post_id,
$limit = null)
3 {
4 $stm = "SELECT * FROM comments WHERE post_id =
:post_id";
5 if ($limit) {
6 $stm .= " LIMIT " . (int) $limit;
7 }
8 $bind = array('post_id' => $post_id);
9 return $this->db->query($stm, $bind);
10 }
11 ?>
```

Now that we have modified the application class, we need to run our existing tests. If they fail, we rejoice! We have discovered that our change was flawed, and the tests have prevented that bug from making it into production. If they pass, we rejoice, because things are still working the way they used to before the change.

Finally, after the existing tests pass, we modify the `CommentsGatewayTest` so that it checks to see if the new `LIMIT` functionality works properly. This test continues to be imperfect, but it gets the point across:

```
tests/classes/Domain/Comments/CommentsGatewayTest.
php
1 <?php
2 public function testSelectAllByPostId()
3 {
4 // a range of known IDs in the comments table
5 $post_id = mt_rand(1,100);
6
7 // get the comment rows
8 $rows = $this->gateway-
>selectAllByPostId($post_id);
9
10 // make sure they all match the post_id
11 foreach ($rows as $row) {
12 $this->assertEquals($post_id, $row['post_id']);
13 }
14
15 // test with a limit
16 $limit = 10;
17 $rows = $this->gateway-
>selectAllByPostId($post_id, $limit);
18 $this->assertTrue(count($rows) <= $limit);
19 }
20 }
21 ?>
```

We run the tests yet again to make sure our new `LIMIT` functionality works, and refine the code and test until it passes.

We then proceed to replace the original embedded SQL code with a call to the `Gateway`, spot check, commit, and so on.

Note

We need to be cautious here. After seeing one variation of a query, we will be able to imagine many other possible variations

of that query. The resulting temptation will be to preemptively modify our `Gateway` methods to account for the imagined variations before we actually encounter them. Unless we have actually seen a particular variation in the legacy codebase, we should restrain ourselves from writing code for that variation. We do not want to get too far ahead of what the codebase actually requires right now. The goal is to improve in small steps on a visible path, not to make giant leaps into a fog of imagination.

What about complex query strings?

The examples so far have been relatively simple query strings. These simple examples help to keep the process clear. However, we are likely to see very complicated queries in our legacy codebase. These may include queries built up in pieces, using several conditional statements to modify how the pieces are built, with many different parameters being used in the query. Here is one example of a complex query taken from Appendix A, *Typical Legacy Page Script*:

```
1 <?php
2 // ...
3 define("SEARCHNUM", 10);
4 // ...
5 $page = ($page) ? $page : 0;
6
7 if (!empty($p) && $p!="all" && $p!="none") {
8 $where = "`foo` LIKE '%$p%'";
9 } else {
10 $where = "1";
11 }
12
13 if ($p=="hand") {
14 $where = "`foo` LIKE '%type1%'"
15 . " OR `foo` LIKE '%type2%'"
16 . " OR `foo` LIKE '%type3%'";
17 }
18
19 $where .= " AND `bar`='1'";
20 if ($s) {
```

```
21 $s = str_replace(" ", "%", $s);
22 $s = str_replace("'", "", $s);
23 $s = str_replace(";", "", $s);
24 $where .= " AND (`baz` LIKE '%$s%')";
25 $orderby = "ORDER BY `baz` ASC";
26 } elseif ($letter!="none" && $letter) {
27 $where .= " AND (`baz` LIKE '$letter%'"
28 . " OR `baz` LIKE 'The $letter%')";
29 $orderby = "ORDER BY `baz` ASC";
30 } else {
31 $orderby = "ORDER BY `item_date` DESC";
32 }
33 $query = mysql_query(
34 "SELECT * FROM `items` WHERE $where $orderby
35 LIMIT $page,".SEARCHNUM;
36 );
37 ?>
```

For complex arrangements of this sort, we will need to pay great attention to detail in extracting the relevant query-building logic to our `Gateway`. The main considerations are to determine which variables are used in the query-building logic, and to set those up as parameters to our new `Gateway` method. We can then move the query-building logic to our `Gateway`.

A first pass at extracting the embedded SQL-related logic to a `Gateway` method follows:

```
1 <?php
2 namespace Domain\Items;
3
4 class ItemsGateway
5 {
6 protected $mysql_link;
7
8 public function __construct($mysql_link)
9 {
10 $this->mysql_link = $mysql_link;
11 }
12
13 public function selectAll(
14 $p = null,
15 $s = null,
16 $letter = null,
```

```
17 $page = 0,
18 $searchnum = 10
19 ) {
20 if (!empty($p) && $p!="all" && $p!="none") {
21 $where = "`foo` LIKE '%$p%'";
22 } else {
23 $where = "1";
24 }
25
26 if ($p=="hand") {
Extract SQL Statements To Gateways 84
27 $where = "`foo` LIKE '%type1%'"
28 . " OR `foo` LIKE '%type2%'"
29 . " OR `foo` LIKE '%type3%'";
30 }
31
32 $where .= " AND `bar`='1'";
33 if ($s) {
34 $s = str_replace(" ", "%", $s);
35 $s = str_replace("'", "", $s);
36 $s = str_replace(";", "", $s);
37 $where .= " AND (`baz` LIKE '%$s%')";
38 $orderby = "ORDER BY `baz` ASC";
39 } elseif ($letter!="none" && $letter) {
40 $where .= " AND (`baz` LIKE '$letter%'"
41 . " OR `baz` LIKE 'The $letter%')";
42 $orderby = "ORDER BY `baz` ASC";
43 } else {
44 $orderby = "ORDER BY `item_date` DESC";
45 }
46
47 $stm = "SELECT *
48 FROM `items`
49 WHERE $where
50 $orderby
51 LIMIT $page, $searchnum";
52
53 return mysql_query($stm, $this->mysql_link);
54 }
55 }
56 ?>
```

Note

Even though we have removed some dependencies (e.g. the implicit global dependency on a `mysql_connect()` link identifier), there are still lots of problems with this first pass. Among other things, it is still vulnerable to SQL injection. We would need to use `mysql_real_escape_string()` on every parameter used in the query, and cast the `LIMIT` values to integers.

Once we complete the extraction and its related test, we change the original code to something like this:

```
1 <?php
2 // ...
3 define("SEARCHNUM", 10);
4 // ...
5 $page = ($page) ? $page : 0;
6 $mysql_link = mysql_connect($db_host, $db_user,
$db_pass);
7 $items_gateway = new
\Domain\Items\ItemsGateway($mysql_link);
8 $query = $items_gateway->selectAll($p, $s,
$letter, $page, SEARCHNUM);
9 ?>
```

What about queries inside non-Gateway classes?

The examples in this chapter show SQL query strings embedded in page scripts. It is just as likely that we will find query strings embedded in non-Gateway classes as well.

In these cases, we follow the same process as we did for page scripts. One added issue is that we will have to pass the `Gateway` dependency to the class. For example, say we have a `Foo` class that uses a `doSomething()` method to retrieve comments:

```
1 <?php
2 class Foo
3 {
4 protected $db;
```

```
5
6 public function __construct(Db $db)
7 {
8 $this->db = $db;
9 }
10
11 public function doSomething($post_id)
12 {
13 $stm = "SELECT * FROM comments WHERE post_id =
$post_id";
14 $rows = $this->db->query($stm);
15 foreach ($rows as $row) {
16 // do something with each row
17 }
18 return $rows;
19 }
20 }
21 ?>
```

We extract the SQL query string and its related logic as we did with the page script. We then modify the `Foo` class to take the `Gateway` as a dependency instead of the `Db` object, and use the `Gateway` as needed:

```
1 <?php
2 use Domain\Comments\CommentsGateway;
3
4 class Foo
5 {
6 protected $comments_gateway;
7
8 public function __construct(CommentsGateway
$comments_gateway)
9 {
10 $this->comments_gateway = $comments_gateway;
11 }
12
13 public function doSomething($post_id)
14 {
15 $rows = $this->comments_gateway-
>selectAllByPostId($post_id);
16 foreach ($rows as $row) {
17 // do something with each row
18 }
19 return $rows;
```

```
20 }
21 }
22 ?>
```

Can we extend from a base Gateway class?

If we have many `Gateway` classes that all have similar functionality, it may be reasonable to collect some of that functionality into an `AbstractGateway`. For example, if they all need the a `Db` connection, and all have similar `select*()` methods, we might do something like the following:

```
classes/AbstractGateway.php
1 <?php
2 abstract class AbstractGateway
3 {
4 protected $table;
5
6 protected $primary_key;
7
8 public function __construct(Db $db)
9 {
10 $this->db = $db;
11 }
12
13 public function
selectOneByPrimaryKey($primary_val)
14 {
15 $stm = "SELECT * FROM {$this->table} "
16 .= "WHERE {$this->primary_key} = :primary_val";
17 $bind = array('primary_val' => $primary_val);
18 return $this->db->query($stm, $bind);
19 }
20 }
21 ?>
```

We can then extend a class from that base `AbstractGateway` and tune the extended properties for a specific table:

```
1 <?php
2 namespace Domain\Items;
3
4 class ItemsGateway extends \AbstractGateway
```

```
5 {
6 protected $table = 'items';
7 protected $primary_key = 'item_id';
8 }
9 ?>
```

The base `selectOneByPrimaryKey()` method can then work with a wide range of `Gateway` classes. We can still add other concrete methods on specific `Gateway` classes as needed.

> **Note**
>
> Be cautious with this approach. We should abstract only the functionality that already exists in the behaviors we have already extracted. Resist the temptation to preemptively create functionality that we have not actually seen in the legacy codebase.

What about multiple queries and complex result structures?

The examples in this chapter have shown single queries against single tables. It is likely that we will encounter logic that uses multiple queries against several different tables and then combines the results into a complex domain entity or collection. Here is an example:

```
1 <?php
2 // build a structure of posts with author and
statistics data,
3 // with all comments on each post.
4 $page = (int) $_GET['page'];
5 $limit = 10;
6 $offset = $page * $limit; // a zero-based paging
system
7 $stm = "SELECT *
8 FROM posts
9 LEFT JOIN authors ON authors.id =
```

```
posts.author_id
10 LEFT JOIN stats ON stats.post_id = posts.id
11 LIMIT {$limit} OFFSET {$offset}"
12 $posts = $db->query($stm);
13
14 foreach ($posts as &$post) {
15 $stm = "SELECT * FROM comments WHERE post_id =
{$post['id']}";
16 $post['comments'] = $db->query($stm);
17 }
18 ?>
```

Note

This example shows a classic N+1 problem where one query is issued for each member of a master set. The first query to get the blog posts will be followed by 10 more queries, one for each blog post, to get the comments. The total number of queries is thus 10, plus one for the initial query. For 50 posts, there would be 51 queries total. This is a typical source of performance drags in legacy applications. For extended discussion of, and solutions to, the N+1 problem, please see *Solving The N+1 Problem in PHP* (https://leanpub.com/sn1php)

The first issue is to identify how we can split up the queries into `Gateway` methods. Some queries will have to go together, while others can be separated. In this case, the first and second queries can be separated into different `Gateway` classes and methods.

The next issue is to determine which `Gateway` class should receive the extracted logic. This can sometimes be hard to figure out when multiple tables are involved, so we have to choose which is the primary subject of the query. The first query above references posts, authors, and stats, but it seems clear from the logic that we are primarily interested in the posts.

As such, we can extract the first query to a `PostsGateway`. We want to modify the query itself as little as possible, so we leave the joins and

such in place:

```
1 <?php
2 namespace Domain\Posts;
3
4 class PostsGateway
5 {
6 protected $db;
7
8 public function __construct(Db $db)
9 {
10 $this->db = $db;
11 }
12
13 public function
selectAllWithAuthorsAndStats($limit = null,
$offset = null)
14 {
15 $limit = (int) $limit;
https://leanpub.com/sn1php
16 $offset = (int) $offset;
17 $stm = "SELECT *
18 FROM posts
19 LEFT JOIN authors ON authors.id =
posts.author_id
20 LEFT JOIN stats ON stats.post_id = posts.id
21 LIMIT {$limit} OFFSET {$offset}"
22 return $this->db->query($stm);
23 }
24 }
25 ?>
```

Once completed, we proceed to write a test for the new functionality based on the first query. We modify the code and test until the test passes.

The second query, the one related to comments, is the same as our earlier example.

After we finish the extractions and their related tests, we can modify the page script to look like the following:

```
1 <?php
2 $db = new Database($db_host, $db_user,
$db_pass);
3 $posts_gateway = new
\Domain\Posts\PostsGateway($db);
4 $comments_gateway = new
\Domain\Comments\CommentsGateway($db);
5
6 // build a structure of posts with author and
statistics data,
7 // with all comments on each post.
8 $page = (int) $_GET['page'];
9 $limit = 10;
10 $offset = $page * $limit; // a zero-based
paging system
11 $posts = $posts_gateway-
>selectAllWithAuthorsAndStats($limit, $offset);
12
13 foreach ($posts as &$post) {
14 $post['comments'] = $comments_gateway-
>selectAllByPostId($post['id']);
15 }
16 ?>
```

What if there is no Database Class?

Many legacy codebases do not have a database access layer. Instead, these legacy applications use the `mysql` extension directly in their page scripts. Calls to `mysql` functions are scattered throughout the codebase and are not gathered into a single class.

If we can upgrade to `PDO`, we should. However, it may not be possible to upgrade away from `mysql` for various reasons. `PDO` does not work quite the same way as `mysql`, and changing from `mysql` idioms to `PDO` ones might be too much to do in a single step. A migration at this point might also make testing more difficult than we would like.

On the other hand, we could move the `mysql` calls as they are into our `Gateway` classes. Doing so seems reasonable at first. However, the `mysql` extension has a bit of global state built into it. Any `mysql` functions that need a link identifer (i.e., a server connection) automatically use the most-recent connection resource when no link

identifier is passed. This is counter to the principles of dependency injection, since we would rather not be dependent on global state if we can help it.

Thus, instead of migrating directly to PDO, and instead of leaving the `msyql` function calls as they are, I suggest we wrap the `mysql` calls in a class that proxies method calls to the `mysql` functions. We can then use the class methods instead of the `mysql` functions. The class itself can contain the link identifier and pass it to each method call. This will give us a database access layer that our `Gateway` objects can use without changing the `mysql` idiomatic usage too greatly.

One operational example implementation of such a wrapper is the `MysqlDatabase` class. When we create an instance of `MysqlDatabase`, it retains the connection information but does not actually connect to the server. It only connects when we call a method that actually needs a server connection. This lazy-loading approach helps us reduce resource usage. In addition, the `MysqlDatabase` class explicitly adds the link identifier argument, which is optional in the relevant `mysql` functions, so that we are never dependent on the implicit global state of the `mysql` extension.

To replace `mysql` function calls with `MysqlDatabase` calls:

1. Search the entire codebase for the `mysql_` prefix on function calls.
2. In each file where there are function calls with the `mysql_` function prefix ...

    - Create or inject an instance of `MysqlDatabase`.
    - Replace each `mysql_` function prefix with the `MysqlDatabase` object variable and a single arrow operator (`->`). If we are sticklers for style, we can can convert the remaining method name portion from `snake_case()` to `camelCase()` as well.

3. Spot check, commit, push, and notify QA.
4. Continue searching for the `mysql_` prefix on function calls until they have all been replaced with `MysqlDatabase` method calls.

For example, say we have a piece of legacy code like this:

Using mysql functions

```
1 <?php
2 mysql_connect($db_host, $db_user, $db_pass);
3 mysql_select_db('my_database');
4 $result = mysql_query('SELECT * FROM table_name
LIMIT 10');
5 while ($row = mysql_fetch_assoc($result)) {
6 // do something with each row
7 }
8 ?>
```

Using the above process, we can convert the code to use the `MysqlDatabase` object instead:

Using the MysqlDatabase class

```
1 <?php
2 $db = new \Mlaphp\MysqlDatabase($db_host,
$db_user, $db_pass);
3 $db->select_db('my_database'); // or $db-
>selectDb('my_database')
4 $result = $db->query('SELECT * FROM table_name
LIMIT 10');
5 while ($row = $db->fetch_assoc($result)) {
6 // do something with each row
7 }
8 ?>
```

That code, in turn, can be extracted to a `Gateway` class using an injected `MysqlDatabase` object.

Note

For our page scripts, it may be best to create a `MysqlDatabase` instance in our existing setup file and use that, instead of creating one separately in each page script. The lazy-connecting nature of the implementation means that if we never make a call to the database, no connection will ever be made, so we don't

need to worry about unnecessary resource usage. The existing legacy codebase will help us determine if this is a reasonable approach.

Once our `Gateway` classes use an injected `MysqlDatabase` object, we can then proceed to planning a migration away from the wrapped `mysql` functions over to `PDO` with its different idioms and usage. Because the database access logic is now wrapped by `Gateway` objects, the migration and testing will be easier than if we had replaced `mysql` calls spread throughout the codebase.

Review and next steps

When we have completed this step, all of our SQL statements will be inside `Gateway` classes, and no longer in our page scripts or other non-Gateway classes. We will also have tests for our `Gateway` classes.

From here on out, any time we need to add new calls to the database, we will do so only in the `Gateway` classes. Any time we need to fetch or save data, we will use the `Gateway` methods instead of writing embedded SQL. This gives us a clear separation of concerns between the database interactions and our future model layer and entity objects.

Now that we have separated our database interactions into their own layer, we are going to inspect all the calls to the `Gateway` objects throughout the legacy application. We will examine how the returned results are manipulated by the page scripts and other classes, and begin extracting the behaviors that define our model layer.

Chapter 9. Extract Domain Logic to Transactions

In the previous chapter, we extracted all our SQL statements to a layer of *Gateway* objects. This encapsulates the interactions between the application and the database.

However, we usually need to apply some amount of business or domain logic to the data coming from and going back to the database. The logic can include things like data validation, adding or modifying values for presentation or calculation purposes, collecting simpler records into more complex ones, using the data to perform related actions, and so on. This domain logic is often embedded into a page script, making that logic difficult to reuse and test.

This chapter describes one way to extract domain behaviors into a separate layer. In many ways, this chapter forms the very core of the book: everything before now has led us to this central concern of the legacy application, and everything that comes after will lead us into layers above and around this core functionality.

Note

Domain or Model?

The domain logic in the legacy application is the model portion of model-view-controller. However, the legacy codebase is unlikely to have separate entity objects that provide a full model of the business domain. Thus, we will be speaking in terms of domain logic and not model logic throughout this chapter. If we are lucky enough to have separate model objects already, so much the better.

Embedded Domain Logic

Although we have extracted SQL statements, the page scripts and classes are probably manipulating the results and performing other actions related to the retrieved data. These manipulations and actions are the core of the domain logic, and it is currently embedded along with other non-domain concerns.

We can see an example of the progression from embedded SQL to using *Gateway* classes by examining the differences between the code in Appendix B, *Code before Gateway* and Appendix C, *Code after Gateway*. The code is too lengthy to present here. What we want to notice is that even after extracting the embedded SQL statements, the code is still doing a lot of work with the incoming and outgoing data before the results are presented to the user.

Having the domain logic embedded in the page script makes it very difficult to test that logic in isolation. We also cannot reuse it easily. If we wanted to search for repetition and duplication in how we work with the domain entities (in this case a series of articles) we would need to review every page script in the entire application.

The solution here is to extract the domain logic to one or more classes so that we can test them independently of any particular page script. We can then instantiate the domain logic classes and use them in any page script we like.

Before we can apply that solution, we need to determine how to structure the target classes for our domain logic.

Domain logic patterns

Martin Fowler's **Patterns of Enterprise Application Architecture** (**PoEAA**) catalogs four domain logic patterns:

- **Transaction Script**: It organizes [domain] logic primarily as a single procedure, making calls directly to the database or through a thin database wrapper. Each transaction will have its own Transaction Script, although common subtasks can be broken into subprocedures."
- **Domain Model**: It creates a web of interconnected objects, where each object represents some meaningful individual, whether as large as a corporation or as small as a single line on an order form.
- **Table Module**: It organizes domain logic with one class per table in the database, and a single instance of a class contains the various procedures that will act on the data, if you have many orders, a Domain Model will have one order object per order while a Table Module will have one object to handle all orders.
- **Service Layer**: It defines an application's boundary and its set of available operations from the perspective of interfacing client layers. It encapsulates the application's business logic, controlling transactions and coordinating responses in the implementation of its operations.

Note

I strongly recommend purchasing PoEAA in hard copy and reading the pattern descriptions and examples in full. The book is an absolute must-have reference for the professional programmer. I find myself consulting it on a weekly basis (sometimes more often) and it always provides clarity and insight.

The choice before us now is this: given the existing structure of our legacy application, which of these patterns best fits the architecture currently in place?

We will dismiss Service Layer at this point, since it implies a level of sophistication that is probably not present in our legacy application.

We will likewise dismiss Domain Model, since it implies a well-designed set of business entity objects that encapsulate behavior. If the legacy application already has one of these patterns implemented, then so much the better. Otherwise, that leaves us with the Table Module and Transaction Script patterns.

When we extracted our SQL statements to `Gateway` classes in the previous chapter, those `Gateway` classes were likely to follow the Table Data Gateway pattern, in particular if they were simple enough to interact with only a single table per `Gateway` class. This makes it seem like the Table Module pattern would be a good fit for our domain logic.

However, it is unlikely that each remaining page script or class with embedded domain logic is interacting with a single table at a time. More frequently, legacy applications have many interactions across many tables in a single class or script. As such, we will begin by using the Transaction Script pattern when we extract our domain logic.

Transaction Script is admittedly a simple pattern to follow. With it, we extract the domain logic from the page script and dump it into a class method mostly intact. We make modifications to the logic only for getting data into and out of the class method so that the original code can still operate properly.

Although we may wish for something more sophisticated than Transaction Script, we have to remember that one of our goals here is to avoid changing the existing logic too dramatically. We are refactoring, not rewriting. What we want right now is to move the code around so that it can be tested and reused appropriately. Thus, a Transaction Script is probably the best way to wrap our legacy domain logic as-it-exists, not as-we-want-it-to-be.

Once we extract the domain logic to its own layer, we will then be able to see that logic more clearly and with less distraction. At that point, if it is truly needed, we can begin to plan a refactoring of the domain layer to something more sophisticated. For example, we may

build a Service Layer that uses Table Modules or a Domain Model to coordinate the various domain interactions. The interface presented to the page scripts by the Service Layer might remain completely unchanged from the Transaction Script interface, although the underlying architecture may have changed completely. But that is a task for another day.

Note

What about Active Record?

Ruby on Rails is famous for using the Active Record pattern, and many PHP developers love that kind of database interaction. It definitely has its strengths. However, Fowler classifies Active Record as a data source architecture pattern, not a domain logic pattern, so we will not be addressing it here.

The Extraction Process

Of the refactoring processes described in this book, extracting domain logic is going to be the most difficult, time consuming, and detail-oriented. This is a very tough thing to do, and it requires a lot of care and attention. The domain logic is the very core of our legacy application, and we need to make sure to pull out just the right parts. This means success is completely dependent on our familiarity and competence with the the legacy application as it exists now.

Luckily, our prior exercises in modernizing our legacy codebase have given us a broad overview of the application as a whole, as well as deep knowledge of the specific parts we have had to extract and refactor. This should endow us with the confidence to complete this task successfully. It is a demanding, but ultimately satisfying, activity.

In general, we proceed as follows:

1. Search the entire codebase for uses of `Gateway` classes that exist outside `Transactions` classes.
2. Where we find `Gateway` usage, examine the logic surrounding the `Gateway` operations to discover which portions of that logic are related to the domain behaviors of the application.
3. Extract the relevant domain logic to one or more `Transactions` classes related to the domain elements, and modify the original code to use the `Transactions` class instead of the embedded domain logic.
4. Spot check to make sure the original code still works properly, and modify the extracted logic as necessary to ensure correct operation.
5. Write tests for the extracted `Transactions` logic, refining them along with the tested code until they pass.
6. When all original tests and new tests pass, commit the code and tests, push to the common repository, and notify QA.
7. Search again for uses of `Gateway` classes, and continue extracting domain logic until `Gateway` usage exists only in

`Transactions`.

Search for uses of Gateway

As in earlier chapters, we use our project-wide search facility to find where we create new instances of `Gateway` classes:

Search for:

```
new .*Gateway
```

The new `Gateway` instance may be used directly in a page script, in which case we have found some candidate code for extracting domain logic. If the `Gateway` instance is injected into a class, we now need to dive into that class to find where the `Gateway` is used. The code surrounding that usage will be our candidate for extracting domain logic.

Discover and Extract Relevant Domain Logic

Tip

When extracting logic to a class method, we should be careful to follow all the lessons we learned about dependency injection in prior chapters. Among other things, this means: no use of globals, replacing superglobals with a `Request` object, no use of the `new` keyword outside `Factory` classes, and (of course) injecting objects via the constructor as needed.

After we have found some candidate code using a `Gateway`, we need to examine the code surrounding `Gateway` usage for these and other operations:

- Normalizing, filtering, sanitizing, and validating of data
- Calculation, modification, creation, and manipulation of data

- Sequential or concurrent operations and actions using the data
- Retention of success/failure/warning/notice messages from those operations and actions
- Retention of values and variables for later inputs and outputs

These and other pieces of logic are likely to be domain-related.

To successfully extract the domain logic to one or more `Transactions` classes and methods, we will have to perform these and other activities:

- Breaking up or reorganizing the extracted domain logic into support methods
- Breaking up or reorganizing the original code to wrap around the new `Transactions` calls
- Retaining, returning, or reporting data needed by the original code
- Adding, changing, or removing variables in the original code related to the extracted domain logic
- Creating and injecting dependencies for the `Transactions` classes and methods

Note

Discovery-and-extraction is best thought of as a learning exercise. Picking apart the legacy application like this is a way of learning how the application is constructed. As such, we should not be afraid to make multiple attempts at extraction. If our first attempt fails, ends up ugly, or gives poor results, we should feel no guilt about scrapping the work and starting over, having learned a little more about what works and what doesn't. For my own part, I often make two or three passes at extracting domain logic before the work is completed to my satisfaction. This is where a revision control system makes our life so much easier; we can work piecemeal, committing only as we are happy with the result, and reverting back to earlier stages if we need to begin again from a clean slate.

Example Extraction

By way of example, recall the code we started with in Appendix B, *Code before Gateways*. Earlier in this chapter we mentioned that we had extracted embedded SQL statements to *ArticlesGateway* classes, ending up with the code in Appendix C, *Code after Gateways*. We now go from that to Appendix D, *Code after Transaction Scripts* where we have extracted the domain logic to an `ArticleTransactions` class.

The extracted domain logic does not appear particularly complicated in its completed form, but actually doing the work turns out to be quite detailed. Review the Appendix C, *Code after Gateways* and compare to the Appendix D, *Code after Transaction Scripts*. Among other things, we should find the following:

- We discovered two separate transactions being performed in the page script: one to submit a new article, and one to update an existing article. In turn, these each needed to operate on the user's credit counts in the database, along with various data normalizing and support operations.
- We extracted the relevant domain logic to an `ArticleTransactions` class and two separate methods, one for creating and one for updating. We named the `ArticleTransactions` methods for the domain logic being performed, not for the implemenation of the underlying technical operations.
- Input filtering has been encapsulated as a support method in the `ArticleTransactions` class for reuse across both of the transaction methods.
- The new `ArticleTransactions` class receives `ArticlesGateway` and `UsersGateway` dependencies to manage the database interactions instead of making direct SQL calls.
- Several variables that were related only to the domain logic have been removed from the page script and placed into the

`Transactions` class as properties.
- The code in the original page script has been greatly reduced. It is now essentially an object creation and injection mechanism, passing user inputs to the domain layer and getting back data to output later.
- Because the domain logic is now encapsulated, the original code can no longer see the `$failure` variable as it gets modified throughout the transaction. That code must now get failure information from the `ArticleTransactions` class for later presentation.

After the extraction, we have a `classes/` directory structure that looks something like the following. This is a result of using a domain-oriented class structure when we extracted SQL to `Gateway` classes:

```
/path/to/app/classes/
1 Domain/
2 Articles/
3 ArticlesGateway.php
4 ArticleTransactions.php
5 Users/
6 UsersGateway.php
```

Note

This need not be our final refactoring. Further modifications of the `ArticleTransactions` are still possible. For example, instead of injecting a `UsersGateway`, it might make sense to extract various domain logic related to users into a `UserTransactions` class and inject that instead. There is still a lot of repetition between the `Transactions` methods. We also need better error checking and condition reporting in the `Transactions` methods. These and other refactorings are secondary, and will be both more noticeable and easier to deal with only after the primary extraction of domain logic.

Spot check the remaining original code

Once we have extracted one or more *Transactions* from the original code, we need to make sure the original code works when using the *Transactions* instead of the embedded domain logic. As before, we do this by running our pre-existing characterization tests. If we do not have characterization tests, we must browse to or otherwise invoke the changed code. If these tests fail, we rejoice! We have discovered that the extraction was flawed, and we have a chance to fix it before we deploy to production. If the "tests" pass, we likewise rejoice, and move on.

Write tests for the extracted transactions

We now know the original code works with the newly extracted *Transactions* logic. However, the new classes and methods need their own set of tests. As with everything else related to extracting domain logic, writing these tests is likely to be detailed and demanding. The logic is probably convoluted, with lots of branches and loops. We should not let this deter us from testing. At the very least, we need to write tests that cover the main cases of the domain logic.

If necessary, we may refactor the extracted logic to separate methods that are themselves more easily testable. Breaking up the extracted logic will make it easier for us to see the flow and find repeated elements of logic. We must remember, though, that our goal is to maintain the existing behavior, not change the behavior presented by the legacy application.

Tip

For insights and techniques on how to make the extracted logic more testable, please see *Refactoring* (http://refactoring.com/)by Martin Fowler et al., as well as *Working Effectively With Legacy Code* (https://www.amazon.com/Working-Effectively-Legacy-Michael-Feathers/dp/01311)by Michael Feathers.

Spot check again, Commit, Push, Notify QA

Finally, because our testing and related refactoring of the extracted *Transactions* logic may have introduced some unexpected changes, we spot check the original code one more time using our characterization tests or by otherwise invoking the relevant code. If these fail, we rejoice! We have found out that our changes were not as good as we thought, and we have a chance to correct the code and tests before they get too far away from us.

When both the original code tests and the extracted *Transactions* tests pass, we rejoice again! We can now commit all of our new work, push it to the central repository, and notify QA that our modernized code is ready for them to review.

Do ... While

We begin the extraction process over again by looking for another *Gateway* usage outside a *Transactions* class. We continue extracting and testing until all *Gateway* calls occur inside *Transactions* classes.

Common Questions

Are we talking about SQL transactions?

The term Transaction Script refers to an architectural pattern, and does not mean the the domain logic must be wrapped in an SQL transaction. It is easy to confuse the two ideas.

Having said that, keeping SQL transactions in mind may help us when extracting domain logic. One useful rule-of-thumb is that pieces of domain logic should be split up according to how well they would fit inside a single SQL transaction. That hypothetical transaction would be committed, or rolled back, as an atomic whole.

This singularity-of-purpose will help us determine where the boundaries of our domain logic lie. We do not actually add SQL transactions, it is just that thinking in those terms can give us some insight as to the boundaries of the domain logic.

What about repeated Domain Logic?

When we extracted SQL statements to `Gateway` classes, we sometimes found queries that were similar but not exactly identical. We had to determine if there was a way to combine them into a single method or not.

In the same way, we may discover that some parts of our legacy domain logic have been copied and pasted in two or more locations. When we find these, we have the same problem as with our `Gateway` classes. Are the pieces of logic similar enough to be combined into a single method, or must they be different methods (or even in completely different `Transactions`)?

The answer here is it depends. In some cases the repeated code will be an obvious copy of logic elsewhere, meaning we can reuse existing `Transactions` methods. If not, we need to extract to a new `Transactions` class or method.

There is also a middle path, where the domain logic as a whole is different, but there are support elements of logic that are identical across different `Transactions`. In these cases, we can refactor the supporting logic as methods on an abstract base `Transactions` class, and then extend new `Transactions` from it. Alternatively, we can extract the logic to a supporting class and inject it into our `Transactions`.

Are printing and echoing part of Domain Logic?

Our `Transactions` classes should not be using `print` or `echo`. The domain logic should only return or retain data.

When we discover output generation in the middle of our domain logic, we should extract that portion so that it lies outside of the domain logic. In general, this means collecting output in the `Transactions` class and then either returning it or making it available by a separate method. Leave output generation for the presentation layer.

Can a transaction be a class instead of a Method?

In the examples, we showed *Transactions* as a collection of methods related to a particular domain entity, such as *ArticleTransactions*. Each part of the domain logic related to that entity was wrapped in a class method.

However, it is also reasonable to break up domain logic into a one-class-per-transaction structure. Indeed, some transactions may be complex enough that they truly require their own separate classes. There is nothing wrong with using a single class to represent a single domain logic transaction.

For example, the earlier *ArticleTransactions* class might be split into an abstract base class with support methods, and two concrete classes for each of the extracted pieces of domain logic. Each of the concrete classes extends the *AbstractArticleTransaction*, like so:

```
classes/
1 Domain/
2 Articles/
3 ArticlesGateway.php
4 Transaction/
5 AbstractArticleTransaction.php
6 SubmitNewArticleTransaction.php
7 UpdateExistingArticleTransaction.php
8 Users/
9 UsersGateway.php
```

If we use a one-class-per-transaction approach, what do we name the main method on the single-transaction class, the one that actually performs the transaction? If there is a common convention for main methods that already exist in our legacy codebase, we should adhere to that convention. Otherwise, we need to pick a single consistent method name. Personally, I enjoy co-opting the `__invoke()` magic method for this purpose, but you may wish to use `exec()` or some other appropriate term to indicate we are executing or otherwise performing the transaction.

What about Domain Logic in Gateway classes?

When we extracted our SQL statements to `Gateway` classes, it is possible that we moved some domain logic into them instead of leaving that logic in its original location. At that earlier point in our refactoring work, it was very easy to confuse domain-level input filtering (which makes sure the data conforms to a domain-specific state) with database-level filtering (which makes sure the data is safe to use with the database).

Now we can more easily tell the difference between the two. If we discover that there is domain-level logic in our `Gateway` classes, we should probably extract it to our `Transactions` classes instead. We need to be sure to update the relevant tests as well.

What about Domain logic embedded in Non-Domain classes?

The examples in this chapter show domain logic embedded in page scripts. It is just as likely that we have domain logic embedded in classes as well. If the class can reasonably be considered part of the domain, and contains only domain-related logic, but is not named for the domain, it may be wise to move the class into the domain namespace.

Otherwise, if the class has any responsibilities other than domain logic, we may proceed to extract the domain logic from it in the same way that we extracted logic from a page script. After the extraction, the original class will then need to have the relevant `Transactions` class injected as a dependency. The original class should then make calls to the `Transactions` as appropriate.

Review and next steps

At this point we have extracted the core of our legacy codebase, the domain logic that sits at the center our application, to its own separate and testable layer. This has been the most demanding of the steps in our modernization process, but it has been very much worth our time. We have not made many modifications or improvements to the domain logic itself. Any changes we have made have been just enough to get data into our new `Transactions` classes and then out again for later use.

In a lot of ways, all we have done is shuffle the logic around so that it is independently addressable. Although the domain logic itself may still have many problems, those problems are now *testable* problems. We can continue adding tests as needed to explore edge cases in our domain logic. If we need to add new domain logic, we can create or modify our `Transactions` classes and methods to encapsulate and test that logic.

The process of extracting domain logic to its own layer leaves us with a great foundation for further iterative refactoring of the domain model. If we choose to pursue it, that refactoring will lead us to a more appropriate architecture for the application domain logic. However, what that architecture will be depends on the application. For more information on developing a good domain model for our application, please read *Domain Driven Design* (https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215) by Eric Evans.

With the extraction of our domain logic to its own layer, we can continue on to the next phase of our modernization process. At this point there are only a few concerns remaining in our original code. Of those concerns, we will next address the presentation layer.

Chapter 10. Extract Presentation Logic to View Files

When it comes to page scripts in legacy applications, it is very common to see business logic intertwined with presentation logic. For example, the page script does some setup work, then includes a header template, makes a call to the database, outputs the results, calculates some values, prints the calculated values, writes the values back to the database, and includes a footer template.

We have made some steps toward decoupling these concerns by extracting a domain layer for our legacy application. However, calls to the domain layer and other business logic within the page scripts are still mixed in with the presentation logic. Among other things, this intermingling of concerns makes it difficult to test the different aspects of our legacy application.

In this chapter, we will separate all of our presentation logic to its own layer so we can test it separately from our business logic.

Embedded presentation logic

For an example of embedded presentation logic, we can take a look at Appendix E, *Code before Collecting*.

Presentation Logic. The code shows a page script that has been refactored to use domain *Transactions*, but it still has some presentation logic entangled within the rest of the code.

> **Note**

What Is The Difference Between Presentation and Business Logic?

For our purposes, presentation logic includes any and all code that generates output sent to the user (such as a browser or mobile client). This includes not only `echo` and `print` but also `header()` and `setcookie()`. Each of these generates some form of output. "Business logic," on the other hand, is everything else.

The key to decoupling the presentation logic from the business logic is to put the code for them into separate scopes. The script should first perform all of the business logic, then pass the results over to the presentation logic. When that is complete, we will be able to test our presentation logic separately from our business logic.

To achieve this separation of scope, we will move toward using a `Response` object in our page scripts. All of our presentation logic will be executed from within a `Response` instance, instead of directly in the page script. Doing so will provide the scope separation that we need to decouple all output generation, including HTTP headers and cookies, from the rest of the page script.

Note

Why A Response Object?

Often, when we think of presentation, we think of a view or a template system that renders content for us. However, these kinds of systems do not usually encapsulate the full set of output that will be sent to the user. We need to output not just HTTP bodies, but HTTP headers as well. In addition, we need to be able to test that the correct headers have been set, and that the content has been generated properly. As such, the `Response` object is a better fit at this point than a view or template system alone. For our `Response` object, we will use the class provided at http://mlaphp.com/code. Note that we will be including files within

a *Response* context, which means that the methods on that object will be available to `include` files running "inside" that object.

The Extraction process

Extracting presentation logic is not as difficult as extracting domain logic. However, it does require careful attention and lots of testing along the way.

In general, the process is as follows:

1. Find a page script that contains presentation logic mixed in with the rest of the code.
2. In that script, rearrange the code to collect all presentation logic into a single block *after* all the other logic in the file, then spot check the rearranged code.
3. Extract the block of presentation logic to a view file to be delivered via a `Response`, and spot check the script again to make sure the script works correctly with the new `Response`.
4. Add proper escaping to the presentation logic and spot check again.
5. Commit the new code, push to the common repository, and notify QA.
6. Begin again with the next page script that contains presentation logic mixed in with other non-presentation code.

Search for Embedded presentation logic

In general, it should be easy for us to find presentation logic in our legacy application. At this point we should be familiar enough with the codebase to have a good idea where the page scripts generate output.

If we need a jump-start, we can use our project-wide search facility to find all occurrences of `echo`, `print`, `printf`, `header`, `setcookie`, and

`setrawcookie`. Some of these may occur in class methods; we will address that at a later point. For now, we will concentrate on page scripts where these calls occur.

Rearrange the Page script and Spot Check

Now that we have a candidate page script, we need to rearrange the code so there is a clear demarcation between the presentation logic and everything else. For our example here, we will use the code in Appendix E, *Code before Collecting*.

First, we go to the bottom of the file and add a `/* PRESENTATION */` comment as the final line. We then go back to the top of the file. Working line-by-line and block-by-block, we move all presentation logic to the end of the file after our `/* PRESENTATION */` comment. When we are done, the part before the `/* PRESENTATION */` comment should consist only of business logic, and the part after should consist only of presentation logic.

Given our starting code in Appendix E, *Code before Collecting*, we should end up with something more like the code in Appendix F, *Code after Collecting*. In particular, note that we have the following:

- Moved variables not used by the business logic, such as `$current_page`, down the presentation block
- Moved the `header.php` include down to the presentation block
- Moved logic and conditions acting only on presentation variables, such as the `if` that sets the `$page_title`, to the presentation block
- Replaced `$_SERVER['PHP_SELF']` with an `$action` variable
- Replaced `$_GET['id']` with an `$id` variable

Note

When creating the presentation block, we should be careful to follow all the lessons we learned from earlier chapters about

dependency injection. Even though the presentation code is a block within a file (not a class) we should treat the block as if it is a class method. Among other things, this means no use of globals, superglobals, or the `new` keyword. This will make things easier on us when we extract the presentation block to a view file later.

Now that we have rearranged the page script so that all presentation logic is collected at the end, we need to spot check to make sure the page script still works properly. As usual, we do this by running our pre-existing characterization tests, if we have any. If not, we must browse to or otherwise invoke the changed code.

If the page does not generate the same output as before, our rearrangement has changed the logic somehow. We need to undo and redo the rearrangement until the page works as it should.

Once our spot check is successful, we may wish to commit our changes so far. If our next set of changes goes badly, we can revert the code to this point as a known working state.

Extract Presentation to View file and Spot Check

Now that we have a working page script with all the presentation logic in a single block, we will extract that entire block to its own file, and then use a `Response` to execute the extracted logic.

Create a views/ Directory

First, we need a place to put view files in our legacy application. While I prefer to keep presentation logic near the business logic, that kind of arrangement will make trouble for us in later modernization steps. As such, we will create a new directory in our legacy application called `views/` and place our view files there. This

directory should be at the same level as our `classes/` and `tests/` directories. For example:

```
/path/to/app/
1 classes/
2 tests/
3 views/
```

Pick a View File name

Now that we have a place to save our view files, we need to pick a file name for the presentation logic we are about to extract. The view file should be named for the page script, in a path under `views/` that matches the page script path. For example, if we are extracting presentation from a page script at `/foo/bar/baz.php`, the target view file should be saved at `/views/foo/bar/baz.php`.

Sometimes it is useful to use an extension other than just `.php` for our view files. I have found it can be helpful to use an extension that indicates the view format. For example, a view that generates HTML may end in `.html.php`, while a view that generates JSON may end in `.json.php`.

Move Presentation Block to View file

Next, we cut the presentation block from the page script, and paste it into our new view file as-is.

Then, in place of the original presentation block in the page script, we create a `Response` object in our page script and point it to our view file with `setView()`. We also set up an empty call to `setVars()` for later, and finally call the `send()` method.

Note

We should *always* use the same variable name for the *Response* object in all of our page scripts. All the examples here will use the name `$response`. This is not because the name `$response` is

special, but because this level of consistency will be very important in a later chapter.

For example:

```
foo/bar/baz.php
1 <?php
2 // ... business logic ...
3
4 /* PRESENTATION */
5 $response = new
\Mlaphp\Response('/path/to/app/views');
6 $response->setView('foo/bar/baz.html.php');
7 $response->setVars(array());
8 $response->send();
9 ?>
```

At this point, we have successfully decoupled the presentation logic from the page script. We can remove the `/* PRESENTATION */` comment. It has served its purpose and is no longer needed.

However, this decoupling fundamentally breaks the presentation logic, because the view file depends on variables from the page script. With that in mind, we begin a spot check-and-modify cycle. We browse to or otherwise invoke the page script and discover that a particular variable is not available to the presentation. We add it to the `setVars()` array, and spot check again. We continue adding variables to the `setVars()` array until the view file has everything it needs, and our spot check runs become completely successful.

Note

For this part of the process, it would be best if we set `error_reporting(E_ALL)`. That way we will get a PHP notice for every uninitialized variable in the presentation logic.

Given our earlier examples in Appendix E, *Code before Collecting* and Appendix F, *Code after Collecting*, we end up at Appendix G, *Code after Response View File*. We can see that the `articles.html.php` view file needed four variables: `$id`, `$failure`, `$input`, and `$action`:

```
1 <?php
2 // ...
3 $response->setVars(array(
4 'id' => $id,
5 'failure' => $article_transactions-
>getFailure(),
6 'input' => $article_transactions->getInput(),
7 'action' => $_SERVER['PHP_SELF'],
8 ));
9 // ...
10 ?>
```

Once we have a working page script, we may wish to commit our work yet again so that we have a known correct state to which we can revert later, if needed.

Add Proper Escaping

Unfortunately, most legacy applications pay little or no attention to output security. One of the most common vulnerabilities is **cross-site scripting** (**XSS**).

Note

What Is XSS?

Cross-site scripting is an attack that is made possible by user input being sent back to the browser unescaped. For example, an attacker can enter maliciously-crafted JavaScript code into a form input or an HTTP header. If that value is then delivered back to the browser without being escaped, the browser will execute that JavaScript code. This has the potential to open the client

browser to further attacks. For more information, please see the *OWASP entry on XSS* (https://www.owasp.org/index.php/Cross-site_Scripting_%28XSS%29).

The defense against XSS is to escape all variables all the time for the context in which they are used. If a variable is used as HTML content, it needs to be escaped as HTML content; if a variable is used in an HTML attribute, it needs to be escaped as such, and so on.

Defending against XSS requires diligence on the part of the developer. If we remember one thing about escaping output, it should be the `htmlspecialchars()` function. Using this function appropriately will save us from most, but not all, XSS exploits.

When using `htmlspecialchars()`, we must be sure to pass a quotes constant and a character set each time. Thus, it is not enough to call `htmlspecialchars($unescaped_text)`. We must call `htmlspecialchars($unescaped_text, ENT_QUOTES, 'UTF-8')`. So, output that looks like this:

unescaped.html.php

```
1 <form action="<?php
2 echo $request->server['PHP_SELF'];
3 ?>" method="POST">
```

This needs to be escaped like this:

escaped.html.php

```
1 <form action="<?php
2 echo htmlspecialchars(
3 $request->server['PHP_SELF'],
4 ENT_QUOTES,
5 'UTF-8'
6 );
7 ?>" method="POST">
```

Any time we send unescaped output, we need to be aware that we are likely opening up a security hole. As such, we must apply escaping to every variable we use for output.

Calling `htmlspecialchars()` repeatedly this way can be cumbersome, so the `Response` class provides an `esc()` method as an alias to `htmlspecialchars()` with reasonable settings:

```
escaped.php
1 <form action="<?php
2 echo $this->esc($request->server['PHP_SELF']);
3 ?>" method="POST">
```

Be aware that escaping via `htmlspecialchars()` is only a starting point. While escaping itself is simple to do, it can be difficult to know the appropriate escaping technique for a particular context.

Unfortunately, it is outside the scope of this book to provide a thorough overview of escaping and other security techniques. For more information, and for a good stand-alone escaping tool, please see the *Zend\Escaper* (https://framework.zend.com/manual/2.2/en/modules/zend.escaper) library.

After we escape all output in the `Response` view file, we can move along to testing.

Write View File Tests

Writing tests for view files presents some unique challenges. Until this chapter, all of our tests have been against classes and class methods. Because our view files are, well, *files*, we need to place them into a slightly different testing structure.

The tests/views/ directory

First, we need to create a `views/` subdirectory in our `tests/` directory. After that, our `tests/` directory should look something like this:

```
/path/to/app/tests/
1 bootstrap.php
2 classes/
3 phpunit.xml
4 views/
```

Next, we need to modify the `phpunit.xml` file so it knows to scan through the new `views/` subdirectory for tests:

```
tests/phpunit.xml
1 <phpunit bootstrap="./bootstrap.php">
2 <testsuites>
3 <testsuite>
4 <directory>./classes</directory>
5 <directory>./views</directory>
6 </testsuite>
7 </testsuites>
8 </phpunit>
```

Writing a View File Test

Now that we have a location for our view file tests, we need to write one.

Although we are testing a file, PHPUnit requires each test to be a class. As a result, we will name our test for the view file being tested, and place it in a subdirectory under `tests/views/` that mimics the original view file location. For example, if we have a view file at `views/foo/bar/baz.html.php`, we would create a test file at `tests/views/foo/bar/BazHtmlTest.php`. Yes, this is a bit ugly, but it will help us keep track of which tests map to which views.

In our test class, we will create a `Response` instance like the one at the end of our page script. We will pass into it the view file path and the needed variables. We will finally require the view, then check the output and headers to see if the view works correctly.

Given our `articles.html.php` file, our initial test might look like this:

tests/views/ArticlesHtmlTest.php

```
1 <?php
2 class ArticlesHtmlTest extends
\PHPUnit_Framework_TestCase
3 {
4 protected $response;
5 protected $output;
6
7 public function setUp()
8 {
9 $this->response = new
\Mlaphp\Response('/path/to/app/views');
10 $this->response->setView('articles.html.php');
11 $this->response->setVars(
12 'id' => '123',
13 'failure' => array(),
14 'action' => '/articles.php',
15 'input' => array(
16 'title' => 'Article Title',
17 'body' => 'The body text of the article.',
18 'max_ratings' => 5,
19 'credits_per_rating' => 1,
20 'notes' => '...',
21 'ready' => 0,
22 ),
23 );
24 $this->output = $this->response->requireView();
25 }
26
27 public function testBasicView()
28 {
29 $expect = '';
30 $this->assertSame($expect, $this->output);
31 }
32 }
33 ?>
```

Note

Why Use requireView() Instead Of send()?

If we use `send()` the `Response` will output the view file results, instead of leaving them in a buffer for us to inspect. Calling `requireView()` invokes the view file but returns the results instead of generating output.

When we run this test, it will fail. We rejoice, because the `$expect` value is empty, but the output should have a lot of content in it. This is the correct behavior. (If the test passes, something is probably wrong.)

Asserting Correctness Of Content

Now we need our test to look at the output to see if it is correct.

The simplest way to do this is to dump the actual `$this->output` string and copy its value to the `$expect` variable. If the output string is relatively short, an `assertSame($expect, $this->output)` to make sure they are identical should be perfectly sufficient for our purposes.

However, if anything changes with any of the other files that our main view file includes, then the test will fail. The failure will occur not because the main view has changed, but because a related view has changed. That is not the kind of failure that helps us.

In the case of large output strings, we can look for an expected substring and make sure it it is present in the actual output. Then when the test fails it will be related to the particular substring for which we are testing, not to the entire output string a a whole.

For example, we can use `strpos()` to see if a particular string is in the output. If the haystack of `$this->output` does not contain the `$expect` needle, `strpos()` will return a boolean `false`. Any other value means the `$needle` is present. (This logic is easier to read if we write our own custom assertion method.)

```
1 <?php
2 public function assertOutputHas($expect)
3 {
```

```
4 if (strpos($this->output, $expect) === false) {
5 $this->fail("Did not find expected output:
$expect");
6 }
7 }
8
9 public function testFormTag()
10 {
11 $expect = '<form method="POST"
action="/articles.php">';
12 $this->assertOutputHas($expect);
13 }
14 ?>
```

This approach has the benefit of being very straightforward, but may not be suitable for complex assertions. We may wish to count the number of times an element occurs, or to assert that the HTML has a particular structure without referencing the contents of that structure, or to check that an element appears in the right place in the output.

For these more-complex content assertions, PHPUnit has an `assertSelectEquals()` assertion, along with other related `assertSelect*()` methods. These work by using CSS selectors to check different parts of the output, but can be difficult to read and understand.

Alternatively, we may prefer to install `Zend\Dom\Query` for finer manipulation of the DOM tree. This library also works by using CSS selectors to pick apart the content. It returns `DOM` nodes and node lists, which makes it very useful for testing the content in a fine-grained manner.

Unfortunately, I cannot give concrete advice on which of these approaches is best for you. I suggest starting with an approach similar to the `assertOutputHas()` method above, and moving along to the `Zend\Dom\Query` approach when it becomes obvious that you need a more powerful system.

After we have written tests that confirm the presentation works as it should, we move on to the last part of the process.

Commit, Push, Notify QA

At this point we should have passing tests for the page script and for the extracted presentation logic. We now commit all our code and tests, push them to the common repository, and notify QA that we are ready for them to look over the new work.

Do ... While

We continue to search for presentation logic mixed in with business logic in our page scripts. When we have extracted all presentation logic to view files via `Response` objects, we are done.

Common Questions

What about Headers and Cookies?

In the above examples we paid attention only to output from `echo` and `print`. However, it is often the case that a page script will also set HTTP headers via `header()`, `setcookie()`, and `setrawcookie()`. These, too, generate output.

Dealing with these output methods can be problematic. Whereas the `Response` class uses `output buffering` to capture `echo` and `print` into return values, there is no similar option for buffering calls to `header()` and related functions. Because the output from these functions is not buffered, we cannot easily test to see what's going on.

This is one place where having a `Response` object really helps us. The class comes with methods that buffer the `header()` and related native PHP functions, but do not call those functions until `send()` time. This allows us to capture the inputs to these calls and test them before they are actually activated.

For example, say we have some code like this in a contrived view file:

```
foo.json.php
1 <?php
2 header('Content-Type: application/json');
3 setcookie('baz', 'dib');
4 setrawcookie('zim', 'gir');
5 echo json_encode($data);
6 ?>
```

Among other things, we cannot test that the headers are what we expect them to be. PHP has already sent them to the client.

When using a view file with a *Response* object, we can prefix the native function calls with `$this->` to call a *Response* method instead of the native PHP function. The *Response* methods buffer the arguments to the native calls instead of making the calls directly. This allows us to inspect the arguments before they are delivered as output.

```
foo.json.php
1 <?php
2 $this->header('Content-Type: application/json');
3 $this->setcookie('baz', 'dib');
4 $this->setrawcookie('zim', 'gir');
5 echo json_encode($data);
6 ?>
```

Note

Because the view file is being executed inside the *Response* instance, it has access to `$this` for the `Response` properties and methods. The `header()`, `setcookie()`, and `setrawcookie()` methods on the `Response` object have the exact same signatures as the native PHP methods, but capture the inputs into a property for later output instead of generating output immediately.

We can now test the `Response` object to check the HTTP body as well as the HTTP headers.

tests/views/FooJsonTest.php

```
1 <?php
2 public function test()
3 {
4 // set up the response object
5 $response = new
\Mlaphp\Response('/path/to/app/views');
6 $response->setView('foo.json.php');
7 $response->setVars('data', array('foo' =>
'bar'));
8
9 // invoke the view file and test its output
10 $expect_body = '{"foo":"bar"}';
11 $actual_body = $response->requireView();
12 $this->assertSame($expect_output,
$actual_output);
13
14 // test the buffered HTTP header calls
15 $expect_headers = array(
16 array('header', 'Content-Type:
application/json'),
17 array('setcookie', 'baz', 'dib'),
18 array('setrawcookie', 'zim', 'gir'),
19 );
20 $actual_headers = $response->getHeaders();
21 $this->assertSame($expect_output,
$actual_output);
22 }
23 ?>
```

Note

The *Response* `getHeaders()` method returns an array of sub-arrays. Each sub-array has an element 0 indicating the native PHP function name to be called, and the remaining elements are arguments to the function. These are the function calls that will be made at `send()` time.

What if we already have a Template system?

Many times, a legacy application will have a view or template system already in place. If so, it may be sufficient to keep using the existing template system instead of introducing a new `Response` class.

If we decide to keep an existing template system, the other steps in this chapter still apply. We need to move all of the template calls to a single location at the end of the page script, disentangling all of the template interactions from the rest of the business logic. We can then display the template at the end of the page script. For example:

foo.php

```
1 <?php
2 // ... business logic ...
3
4 /* PRESENTATION */
5 $template = new Template;
6 $template->assign($this->getVars());
7 $template->display('foo.tpl.php');
8 ?>
```

If we are not sending HTTP headers, this approach is just as testable as using a `Response` object. However, if we mix in calls to `header()` and related functions, our testability will be more limited.

In the interest of future-proofing our legacy code, we may move the template logic to a view file, and interact with a `Response` object in our page script instead. For example:

foo.php

```
1 <?php
2 // ... business logic ...
3
4 /* PRESENTATION */
5 $response = new Response('/path/to/app/views');
6 $response->setView('foo.html.php');
7 $response->setVars(array('foo' => $foo));
8 $response->send();
9 ?>
```

foo.html.php

```
1 <?php
```

```
2 // buffer calls to HTTP headers
3 $this->setcookie('foo', 'bar');
4 $this->setrawcookie('baz', 'dib');
5
6 // set up the template object with Response vars
7 $template = new Template;
8 $template->assign($this->getVars());
9
10 // display the template
11 $template->display('foo.tpl.php');
12 ?>
```

This allows us to keep using the existing template logic and files, while adding testability for HTTP headers via the `Response` object.

For consistency's sake, we should either use the existing template system or wrap all template logic in view files via `Response` objects. We should not use the template system in some page scripts and the `Response` object in others. In later chapters, it will be important that we have a single way of interacting with the presentation layer in our page scripts.

What about Streaming Content?

Most of the time, our presentation is small enough that it can be buffered into memory by PHP until it is ready to send. However, sometimes our legacy application may need to send large amounts of data, such as a file that is tens or hundreds of megabytes.

Reading a large file into memory so that we can output it to the user is usually not a good approach. Instead, we stream the file: we read a small piece of the file and send it to the user, then read the next small piece and send it to the user, and so on until the whole file has been delivered. That way, we never have to keep the entire file in memory.

The examples so far have only dealt with buffering a view into memory and then outputting it all at once, not with streaming. It would be a poor approach for our view file to read the entire

resource into memory and then output it. At the same time, we need to make sure headers are delivered before any streamed content.

The `Response` object has a method to handle this situation. The `Response` method `setLastCall()` allows us to set a user-defined function (a callable) to invoke after requiring the view file and sending the headers. With this, we can pass a class method that will stream the resource out for us.

For example, say we need to stream out a large image file. We can write a class like the following to handle the stream logic:

classes/FileStreamer.php

```
1 <?php
2 class FileStreamer
3 {
4 public function send($file, $dest = STDOUT)
5 {
6 $fh = fopen($file, 'rb');
7 while (! feof($fh)) {
8 $data = fread($fh, 8192);
9 fwrite($dest, $data);
10 }
11 fclose($fh);
12 }
13 }
14 ?>
```

There is much to be desired here, such as error checking and better resource handling, but it accomplishes the purpose for our example.

We can then create an instance of the *FileStreamer* in our page script, and the view file can use it as the callable argument for `setLastCall()`:

foo.php

```
1 <?php
2 // ... business logic ...
3 $file_streamer = new FileStreamer;
4 $image_file = '/path/to/picture.tiff';
5 $content_type = 'image/tiff';
6
```

```
7 /* PRESENTATION */
8 $response = new Response('/path/to/app/views');
9 $response->setView('foo.stream.php');
10 $response->setVars(array(
11 'streamer' => $file_streamer,
12 'file' => $image_file,
13 'type' => $content_type,
14 ));
15 ?>
```

views/foo.stream.php

```
1 <?php
2 $this->header("Content-Type: {$type}");
3 $this->setLastCall(array($streamer, 'send'),
$file);
4 ?>
```

At `send()` time, the `Response` will require the view file, which sets a header and the last call with arguments. The `Response` then sends the headers and the captured output of the view (which in this case is nothing). Finally, it invokes the callable and arguments from `setLastCall()`, which streams out the file.

What if we have lots of Presentation variables?

In the example code from this chapter, we had only a handful of variables to pass to the presentation logic. Unfortunately, it is more likely that there will be 10 or 20 or more variables to pass. This is usually because the presentation is composed of several `include` files, each of which needs its own variables.

These extra variables are usually needed for things like the site header, navigation, and footer portions. Because we have decoupled the business logic from the presentation logic and are executing the presentation logic in a separate scope, we have to pass in all the variables needed for all the `include` files.

Say we have a view file that includes a `header.php` file, like this:

header.php

```
1 <html>
2 <head>
3 <title><?php
4 echo $this->esc($page_title);
5 ?></title>
6 <link rel="stylesheet" href="<?php
7 echo $this->esc($page_style);
8 ?>"></link>
9 </head>
10 <body>
11 <h1><?php echo $this->esc($page_title); ?></h1>
12 <div id="navigation">
13 <ul>
14 <?php foreach ($site_nav as $nav_item) {
Extract Presentation Logic To View Files 117
15 $href = $this->esc($nav_item['href']);
16 $name = $this->esc($nav_item['name']);
17 echo '<li><a href="' . $href
18 . '"/a>' . $name
19 . '</li>' . PHP_EOL;
20 }?>
21 </ul>
22 </div>
23 <!-- end of header.php -->
```

Our page script will have to pass `$page_title`, `$page_style`, and `$site_nav` variables in order for the header to display properly. This is a relatively tame case; there could be many more variables than this.

One solution is to collect commonly-used variables into one or more objects of their own. We can then pass those common-use objects into the `_Response_` for the view file to use. For example, header-specific display variables can be placed in a `HeaderDisplay` class, which can then be passed to the `_Response_`.

```
classes/HeaderDisplay.php
1 <?php
2 class HeaderDisplay
3 {
4 public $page_title;
5 public $page_style;
```

```
6 public $site_nav;
7 }
8 ?>
```

We can then modify the `header.php` file to use the *HeaderDisplay* object, and the page script can pass an instance of *HeaderDisplay* instead of all the separate header-related variables.

> ## Tip
>
> Once we begin collecting related variables into classes, we will begin to see how we can collect presentation logic into methods on those classes, and thereby reduce the amount of logic in our view files. For example, it should not be hard for us to imagine a `getNav()` method on the *HeaderDisplay* class that returns the proper HTML for our navigation widgets.

What about class methods that generate output?

In the example code for this chapter, we concentrated on presentation logic in page scripts. However, it may be the case that domain classes or other support classes use `echo` or `header()` to generate output. Because output generation must be restricted to the presentation layer, we need to find a way to remove these calls without breaking our legacy application. Even classes that are intended for presentation purposes should not generate output on their own.

The solution here is to convert each use of `echo`, `print`, and so on to a `return`. We can then either output the result immediately, or capture the result into a variable and output it later.

For example, say we have a class method that looks like this:

```
1 <?php
2 public function namesAndRoles($list)
3 {
4 echo "<p>Names and roles:</p>";
5 foreach ($list as $item) {
6 echo "<dl>";
7 echo "<dt>Name</dt><dd>{$item['name']}</dd>";
8 echo "<dt>Role</dt><dd>{$item['role']}</dd>";
9 echo "</dl>";
10 }
11 }
12 ?>
```

We can convert it to something like this instead (and remember to add escaping!):

```
1 <?php
2 public function namesAndRoles($list)
3 {
4 $html = "<p>Names and roles:</p>";
5 foreach ($list as $item) {
6 $name = htmlspecialchars($item['name'],
ENT_QUOTES, 'UTF-8');
7 $role = htmlspecialchars($item['role'],
ENT_QUOTES, 'UTF-8');
8 $html .= "<dl>";
9 $html .= "<dt>Name</dt><dd>{$name}</dd>";
10 $html .= "<dt>Role</dt><dd>{$role}</dd>";
11 $html .= "</dl>";
12 }
13 return $html;
14 }
15 ?>
```

What about Business Logic Mixed into the presentation?

When rearranging the page script to separate the business logic from the presentation logic, we may discover that the presentation code makes calls to *Transactions* or other classes or resources. This is a pernicious form of mixing concerns, since the presentation is dependent on the results of these calls.

If the called code is specifically for output, then there's no problem; we can leave the calls in place. But if the called code interacts with an external resource such as a database or a network connection, we have a mixing of concerns that needs to be separated.

The solution is to extract an equivalent set of business logic calls from the presentation logic, capture the results to a variable, and then pass that variable to the presentation.

For a contrived example, the following mixed code makes database calls and then presents them in a single loop:

```
1 <?php
2 /* PRESENTATION */
3 foreach ($post_transactions->fetchTopTenPosts()
as $post) {
4 echo "{$post['title']} has "
5 . $comment_transactions-
>fetchCountForPost($post['id'])
6 . " comments.";
7 }
8 ?>
```

Ignore for a moment that we need to solve the N+1 query problem presented in the example, and that this might better be solved at the *Transactions* level. How can we disentangle the presentation from the data retrieval?

In this case, we build an equivalent set of code to capture the needed data, then pass that data to the presentation logic, and apply proper escaping:

```
1 <?php
2 // ...
3 $posts = $post_transactions->fetchTopTenPosts();
4 foreach ($posts as &$post) {
5 $count = $comment_transactions-
>fetchCountForPost($post['id']);
6 $post['comment_count'] = $count;
7 }
8 // ...
```

```
9
10 /* PRESENTATION */
11 foreach ($posts as $post) {
12 $title = $this->esc($post['title']);
13 $comment_count = $this-
>esc($post['comment_count']);
14 echo "{$title} has {$comment_count} comments."
15 }
16 ?>
```

Yes, we end up looping over the same data twice -- once in the business logic, and once in the presentation logic. While this may reasonably be called inefficient in some ways, efficiency is not our primary goal. Separation of concerns is our primary goal, and this approach achieves that nicely.

What if a page contains only presentation logic?

Some of the pages in our legacy application may consist mostly or entirely of presentation code. In these cases, it may seem like we don't need a *Response* object.

However, even these page scripts should be converted to use a *Response* and a view file. A later step in our modernization process is going to require a consistent interface to the results of our page scripts, and our *Response* object is the way to ensure that consistency.

Review and next steps

We have now gone through all of our page scripts and extracted the presentation logic to a series of separate files. The presentation code is now executed in a scope completely independent from the page script. This makes it very easy for us to see the remaining logic of the script, as well as test the presentation logic independently.

With the presentation logic extracted to its own layer, our page scripts are dwindling in size. All that remains in them is some setup work and the action logic needed to prepare a response.

Our next step, then, is to extract the remaining action logic from our page scripts to a series of controller classes.

Chapter 11. Extract Action Logic to Controllers

Thus far, we have extracted our model domain logic and our view presentation logic. Only two kinds of logic remain in our page scripts:

- Dependency logic, which uses the application settings to create objects
- Action logic (sometimes called business logic) which uses those objects to perform the page actions

In this chapter, we will extract a layer of `Controller` classes from our page scripts. These will handle the remaining action logic in our legacy application separately from our dependency-creation logic.

Embedded action logic

For an example of embedded action logic mixed with dependency logic, we can look at the ending example code from the last chapter in Appendix G, *Code after Response View File*. Therein, we do a little setup work, then we check some conditions and call different parts of our domain `Transactions`, and at the end we put together a `Response` object to send our response to the client.

As was the problem with mixed-in presentation logic, we cannot test the action logic separately from rest of the page script. Similarly, we cannot easily change the dependency creation logic to make the page script more testable.

We solve the problem of embedded action logic as we did with embedded presentation logic. We must extract the action code to a class of its own to separate the various remaining concerns of our page script. This will also allow us to test the action logic independently from the rest of the application.

The Extraction Process

Extracting the action logic from our page scripts should be a relatively easy task for us now. Because the domain layer has been extracted, along with the presentation layer, the action logic should be obvious. The work itself still rewards attention to detail, in that the main issue will be picking apart the dependency setup portions from the action logic itself.

In general, the process is as follows:

1. Find a page script where action logic is still mixed in with the rest of the code.
2. In that page script, rearrange the code so that all action logic is in its own central block. Spot check the rearranged code to make sure it still works properly.
3. Extract the central block of action logic to a new `Controller` class, and modify the page script to use the new `Controller`. Spot check the page script with the *Controller* in place.
4. Write a unit test for the new `Controller` class and spot check again.
5. Commit the new code and tests, push them to the common repository, and notify QA.
6. Find another page script with embedded action logic and start again; when all page scripts use `Controller` objects, we are done.

Search for Embedded Action Logic

At this point, we should be able to find action logic without having to use our project-wide search facility. Every page script in our legacy application probably has at least a little bit of action logic left in it.

Rearrange the Page Script and Spot Check

When we have a candidate page script, we proceed to rearrange the code so that all setup and dependency-creation work is at the top, all the action logic is in the middle, and the `$response->send()` call is at the bottom. For our starting example here, we will use the code from the end of the last chapter as found in Appendix G, *Code after Response View File*.

Identify Code Blocks

First, we go to the very top of the script and place a `/* DEPENDENCY */` comment on the first line (or perhaps after the inclusion of a setup script). Then we go to the very end of the script, to the `$response->send()` line, and place a `/* FINISHED */` comment above it.

Now we reach a point where we must use our professional judgment. On some line after the setup and dependency work in the page script, we will see that the code begins to perform some sort of action logic. Our assessment of just where this transition occurs may be somewhat arbitrary, since the action logic and setup logic are likely to still be intertwined. Even so, we must pick a point at which we believe the action logic really gets started, and place a `/* CONTROLLER */` comment there.

Move Code to Its Related Block

Once we have identified these three blocks in the page script, we begin rearranging the code so that only setup and dependency-creation work occurs between `/* DEPENDENCY */` and `/* CONTROLLER */`, and only action logic occurs between `/* CONTROLLER */` and `/* FINISHED */`.

In general, we should avoid conditions or loops in the dependency block, and avoid object creation in the controller block. The code in the dependency block should only create objects, and the code in the controller block should only operate on objects that have been created in the dependency block.

Given our starting code in Appendix G, *Code after Response View File*, we can see the result of an example rearrangement in Appendix H, *Code after Controller Rearrangement*. Of note, we moved the `$user_id` declaration down to the controller block, and we moved the `Response` object creation up to the dependency block. The original action logic in the central controller block remains otherwise unchanged.

Spot Check the Rearranged Code

Finally, after rearranging the page script, we need to spot check our changes to make sure everything still works properly. If we have characterization tests, we should run those. Otherwise, we should browse to or otherwise invoke the page script. If it does not work correctly, we need to undo and redo our rearrangement so that we fix whatever errors we have introduced.

When our spot check runs are successful, we may wish to commit our changes so far. This will give us a known-working state to which we can revert if future changes go bad.

Extract a Controller Class

Now that we have a rearranged page script that works properly, we can extract the central controller block to a class of its own. This is not difficult, but we will do it in several sub-steps to make sure everything goes smoothly.

Pick a Class Name

Before we can extract to a class, we need to pick a name for the class we will extract to.

With our domain-layer classes, we chose the top-level namespace *Domain*. Because this is a controller layer, we will use the top-level namespace *Controller*. The namespace we use is not as important as consistently using the same namespace for all controllers.

Personally, I prefer *Controller* because it is broad enough to encompass different kinds of controllers, such as Application Controller.

The class name within that namespace should reflect where the page script is in the URL hierarchy, with namespace separators where there are directory separators in the path. This approach makes it obvious what the original page script directory path was, and keeps the subdirectories organized nicely in the class structure. We also suffix the class name with `Page` to indicate it is a Page Controller.

For example, if the page script is at `/foo/bar/baz.php`, the class name should be `Controller\Foo\Bar\BazPage`. The class file itself would then be placed in our central classes directory under `classes/Controller/Foo/Bar/BazPage.php`.

Create a Skeleton Class File

Once we have a class name, we can create a skeleton class file for it. We add two empty methods as placeholders for later: the `__invoke()` method will receive the action logic from the page script, and the constructor will eventually receive dependencies for the class.

classes/Controller/Foo/Bar/BazPage.php

```
1 <?php
2 namespace Controller\Foo\Bar;
3
4 class BazPage
5 {
6 public function __construct()
7 {
8 }
9
10 public function __invoke()
11 {
12 }
13 }
14 ?>
```

Note

Why __invoke()?

Personally, I enjoy co-opting the `__invoke()` magic method for this purpose, but you may wish to use `exec()` or some other appropriate term to indicate we are executing or otherwise running the controller. Whatever method name we choose, we should use it consistently.

Move the Action Logic and Spot Check

Now we are ready to extract the action logic to our new `Controller` class.

First, we cut the controller block from the page script, and paste it into the `__invoke()` method as-is. We add one line to the end of the action logic, `return $response`, to send the *Response* object back to the calling code.

Next, we go back to the page script. In the place of the extracted action logic, we create an instance of our new `Controller` and call its `__invoke()` method, getting back a *Response* object.

We should `always` use the same variable name for the *Controller* object in all of our page scripts. All the examples here will use the name `$controller`. This is not because the name `$controller` is special, but because this level of consistency will be very important in a later chapter.

At this point, we have successfully decoupled the action logic from the page script. However, this decoupling fundamentally breaks the action logic, because the *Controller* depends on variables from the page script.

With that in mind, we begin a spot-check-and-modify cycle. We browse to or otherwise invoke the page script and discover that a particular variable is not available to the *Controller*. We add it to the `__invoke()` method signature, and spot check again. We continue adding variables to the `__invoke()` method until the *Controller* has everything it needs and our spot check runs become completely successful.

> **Note**
>
> For this part of the process, it would be best if we set `error_reporting(E_ALL)`. That way we will get a PHP notice for every uninitialized variable in the action logic.

Given our rearranged page script in Appendix H, *Code after Controller Rearrangement*, the result of our initial extraction to a *Controller* can be seen in Appendix I, *Code after Controller Extraction*. It turns out that the extracted action logic needed four variables: `$request`, `$response`, `$user`, and `$article_transactions`.

Convert Controller to Dependency Injection and Spot Check

Once we have a working block of action logic in the `__invoke()` method, we will convert the method parameters into constructor parameters so that the *Controller* can use dependency injection.

First, we cut the `__invoke()` parameters and paste them as a whole into the `__construct()` parameters. We then edit the class definition and `__construct()` method to retain the parameters as properties.

Next, we modify the `__invoke()` method to use the class properties instead of the method parameters. That means prefixing each of the needed variables with `$this->`.

Then, we go back to the page script. We cut the arguments to the `__invoke()` call, and paste them into the *Controller* instantiation.

Now that we have converted the *Controller* to dependency injection, we need to spot check the page script again to make sure everything works properly. If it does not, we need to undo and redo our conversion until our tests pass.

At this point, we can remove the `/* DEPENDENCY */`, `/* CONTROLLER */`, and `/* FINISHED */` comments. They have served their purpose and are no longer needed.

Given the `__invoke()` usage in Appendix I, *Code after Controller Extraction*, we can see what converting the *Controller* to dependency injection looks like in Appendix J, *Code after Controller Dependency Injection*. We have moved the *Controller* `__invoke()` parameters up to `__construct()`, retained them as properties, used the new properties in the `__invoke()` method body, and modified the page script to pass the needed variables at `new` time instead of `__invoke()` time.

Once we have a working page script, we may wish to commit our work yet again so that we have a known correct state to which we can revert later, if needed.

Write a Controller Test

Even though we have tested our page script, we need to write a unit test for our extracted *Controller* logic. When we write the test, we will need to inject all the needed dependencies into our *Controller*, preferably as test doubles like fakes or mocks so we can isolate the *Controller* from the rest of the system.

When we make assertions, they should probably be against the *Response* object returned from the `__invoke()` method. We can use `getView()` to make sure the right view file is set, `getVars()` to inspect the variables to be used in the view, and `getLastCall()` to see if the final callable (if any) has been set properly.

Commit, Push, Notify QA

Once we have a passing unit test, and our tests of the original page script also pass, we can commit our new code and tests. Then we push to the common repository and notify QA that we are ready for them to review our work.

Do ... While

Now we go on to the next page script that has embedded action logic and begin the extraction process over again. When all of our page scripts use dependency-injected *Controller* objects, we are done.

Common Questions

Can we pass parameters to the Controller method?

In the examples, we remove all parameters from the `__invoke()` method. However, sometimes we will want to pass a parameter to that method as last-minute information for the controller logic.

In general, we should avoid doing so at this point in our modernization process. This is not because it is a poor practice, but because we need a very high level of consistency in our controller invocations for a later modernization step. The most-consistent thing is for there to be no `__invoke()` parameters at all.

If we need to pass extra information to the *Controller*, we should do so via the constructor. This is especially the case when we are passing request values.

For example, instead of this:

page_script.php

```
1 <?php
2 /* DEPENDENCY */
3 // ...
4 $response = new
\Mlaphp\Response('/path/to/app/views');
5 $foo_transactions = new
\Domain\Foo\FooTransactions(...);
6 $controller = new \Controller\Foo(
7 $response,
8 $foo_transactions
9 );
10
11 /* CONTROLLER */
12 $response = $controller->__invoke('update',
$_POST['user_id']);
13
14 /* FINISHED */
15 $response->send();
16 ?>
```

We could do this:

page_script.php

```
1 <?php
2 /* DEPENDENCY */
3 // ...
4 $response = new
\Mlaphp\Response('/path/to/app/views');
5 $foo_transactions = new
\Domain\Foo\FooTransactions(...);
6 $request = new \Mlaphp\Request($GLOBALS);
7 $controller = new \Controller\Foo(
8 $response,
9 $foo_transactions,
10 $request
11 );
12
13 /* CONTROLLER */
14 $response = $controller->__invoke();
15
16 /* FINISHED */
17 $response->send();
18 ?>
```

The `__invoke()` method body would then use `$this->request->get['item_id']`.

Can a Controller have Multiple actions?

In the examples, our *Controller* objects perform a single action. However, it is often the case that a page controller encompasses multiple actions, such as both inserting and updating a database record.

Our first pass at extracting action logic from the page script should keep the code pretty much intact, making allowances for properties instead of local variables and so on. Once the code is in the class, though, it is perfectly reasonable to split the logic into separate action methods. Then the `__invoke()` method can become little more than a `switch` statement that picks the correct action method. If we do so, we should be sure to update our *Controller* tests, and continue to spot check the page script to make sure our changes do not break anything.

Note that if we create additional *Controller* action methods, we need to avoid calling them from our page script. For the sake of the consistency needed in a later modernization step, the `__invoke()` method should be the only *Controller* method our page script calls in its controller block.

What If the Controller contains include Calls?

Unfortunately, as we go about rearranging a page script, we are likely to discover that we still have several `include` calls in the controller block. (Calls to `include` for setup and dependency purposes are not such a big deal, especially if they are the same in every page script.)

Having `include` calls in the controller block is an artifact of the include-oriented architecture with which our legacy application began. It is a particularly difficult problem to solve. We want to encapsulate action logic in classes, not in files that execute behavior the moment we `include` them.

For now, we must submit ourselves to the idea that `include` calls in the controller block of our page scripts are ugly but necessary. We should avert our eyes if needed and copy them into the `Controller` class with the rest of the controller code from the page script.

As consolation, we will solve the problem of these embedded `include` calls in the next chapter.

Review and next steps

The extraction of the action logic to a layer of *Controllers* completes a huge modernization goal for our legacy application. We now have a full Model View Controller system in place: a domain layer for models, a presentation layer for views, and a controller layer that connects the two.

We should feel very satisfied about our modernization progress. The code that remains in each page script is a shadow of its original self. Most of the logic is wiring code that creates a *Controller* with its dependencies. The remaining logic is the same across all page scripts; it invokes the *Controller* and sends the returned *Response* object.

However, there is a major legacy artifact for us to deal with. To finish a full extraction and encapsulation of controller logic, we need to remove any remaining `include` calls embedded in our *Controller* classes.

Chapter 12. Replace Includes in Classes

Even though we have Model View Controller separation now, we may still have many include calls in our classes. We want our legacy application to be free from the artifacts of its include-oriented heritage, where merely including a file causes logic to be executed. To do so, we will need to replace include calls with method calls throughout our classes.

> **Note**
>
> For the purposes of this chapter, we will use the term include to cover not just `include` but also `require`, `include_once`, and `require_once`.

Embedded include Calls

Let's say we extracted some action logic with an embedded `include` to a *Controller* method. The code receives information on a new user, calls an `include` to perform some common validation functionality, and then deals with success or failure of validation:

classes/Controller/NewUserPage.php

```
1 <?php
2 public function __invoke()
3 {
4 // ...
5 $user = $this->request->post['user'];
6 include
'includes/validators/validate_new_user.php';
7 if ($user_is_valid) {
8 $this->user_transactions->addNewUser($user);
9 $this->response->setVars('success' => true);
```

```
10 } else {
11 $this->response->setVars(array(
12 'success' => false,
13 'user_messages' => $user_messages
14 ));
15 }
16
17 return $this->response;
18 }
19 ?>
```

Here is an example of what the included file might look like:

```
includes/validators/validate_new_user.php
1 <?php
2 $user_messages = array();
3 $user_is_valid = true;
4
5 if (! Validate::email($user['email'])) {
6 $user_messages[] = 'Email is not valid.';
7 $user_is_valid = false;
8 }
9
10 if (! Validate::strlen($foo['username'], 6, 8))
{
11 $user_messages[] = 'Username must be 6-8
characters long.';
12 $user_is_valid = false;
13 }
14
15 if ($user['password'] !==
$user['confirm_password']) {
16 $user_messages[] = 'Passwords do not match.';
17 $user_is_valid = false;
18 }
19 ?>
```

Let us ignore for now the specifics of the validation code. The point here is that the `include` file and any code using it are both tightly coupled to each other. Any code using the file has to initialize a `$user` variable before including it. Any code using the file also has an expectation of getting two new variables introduced into its scope (`$user_messages` and `$user_is_valid`).

We want to decouple this logic so that the logic in the `include` file does not intrude on the scope of the class methods in which is it used. We do this by extracting the logic of the `include` file to a class of its own.

The Replacement process

The difficulty of extracting includes to their own classes depends on the number and complexity of the `include` calls remaining in our class files. If there are very few includes and they are relatively simple, the process will be easy to complete. If there are many complex interdependent includes, the process will be relatively difficult to work through.

In general, the process is as follows:

1. Search the `classes/` directory for an `include` call in a class.
2. For that `include` call, search the entire codebase to find how many times the included file is used.
3. If the included file is used only once, and only in that one class:

    1. Copy the contents of the included file code as-is directly over the `include` call.
    2. Test the modified class, and delete the include file.
    3. Refactor the copied code so that it follows all our existing rules: no globals, no `new`, inject dependencies, return instead of output, and no `include` calls.

4. If the included file is used more than once:

    1. Copy the contents of the included file as-is to a new class method.
    2. Replace the discovered `include` call with inline instantiation of the new class and invocation of the new method.
    3. Test the class in which the `include` was replaced to find coupled variables; add these to the new method signature by reference.
    4. Search the entire codebase for `include` calls to that same file, and replace each with inline instantiation and invocation; spot check modified files and test modified classes.

    5. Delete the original `include` file; unit test and spot check the entire legacy application.
    6. Write a unit test for the new class, and refactor the new class so that it follows all our existing rules: no globals, no superglobals, no `new`, inject dependencies, return-not-output, and no includes.
    7. Finally, replace each inline instantiation of the new class in each of our class files with dependency injection, testing along the way.

5. Commit, push, notify QA.
6. Repeat until there are no `include` calls in any of our classes.

Search for include Calls

First, as we did in a much earlier chapter, we use our project-wide search facility to find `include` calls. In this case, search only the `classes/` directory with the following regular expression:

```
^[ \t]*(include|include_once|require|require_once)
```

This should give us a list of candidate `include` calls in the `classes/` directory.

We pick a single `include` file to work with, then search the entire codebase for other inclusions of the same file. For example, if we found this candidate `include` ...

```
1 <?php
2 require 'foo/bar/baz.php';
3 ?>
```

We would search the entire codebase for `include` calls to the file name `baz.php`:

```
^[ \t]*
(include|include_once|require|require_once).*baz\.
```

```
php
```

We search only for the file name because, depending on where the `include` call is located, the relative directory paths might lead to the same file. It is up to us to determine which of these `include` calls reference the same file.

Once we have a list of `include` calls that we know lead to the same file, we count the number of calls that include that file. If there is only one call, our work is relatively simple. If there is more than one call, our work is more complex.

Replacing a Single include Call

If a file is used as the target of an `include` call only once, it is relatively easy to remove the `include`.

First, we copy the entire contents of the `include` file. We move back to the class where the `include` occurs, delete the `include` call, and paste the entire contents of the `include` file in its place.

Next, we run the unit tests for the class to make sure it still works properly. If they fail, we rejoice! We have found errors to be corrected before we continue. If they pass, we likewise rejoice, and move on.

Now that the `include` call has been replaced, and the file contents have been successfully transplanted to the class, we delete the include file. It is no longer needed.

Finally, we can return to our class file where the newly transplanted code lives. We refactor it according to all the rules we have learned so far: no globals or superglobals, no use of the `new` keyword outside of factories, inject all needed dependencies, return values instead of generating output, and (recursively) no `include` calls. We run our unit tests along the way to make sure we do not break any pre-existing functionality.

Replacing Multiple include Calls

If a file is used as the target of multiple `include` calls, it will take more work to replace them.

Copy include file to Class Method

First, we will copy the `include` code to a class method of its own. To do this, we need to pick a class name appropriate to the purpose of the included file. Alternatively, we may name the class based on the path to the included file so we can keep track of where the code came from originally.

As for the method name, we again pick something appropriate to the purpose of the `include` code. Personally, if the class is going to contain only a single method, I like to co-opt the `__invoke()` method for this. However, if there end up being multiple methods, we need to pick a sensible name for each one.

Once we have picked a class name and method, we create the new class in the proper file location, and copy the `include` code directly into the new method. (We do not delete the include file itself just yet.)

Replace the original include Call

Now that we have a class to work with, we go back to the `include` call we discovered in our search, replace it with an inline instantiation of the new class, and invoke the new method.

For example, say the original calling code looked like this:

Calling Code

```
1 <?php
2 // ...
3 include
'includes/validators/validate_new_user.php';
```

```
4 // ...
5 ?>
```

If we extracted the `include` code to a `Validator\NewUserValidator` class as its `__invoke()` method body, we might replace the `include` call with this:

```
Calling Code
1 <?php
2 // ...
3 $validator = new \Validator\NewUserValidator;
4 $validator->__invoke();
5 // ...
6 ?>
```

> **Note**
>
> Using inline instantiation in a class violates one of our rules regarding dependency injection. We do not want to use the `new` keyword outside of factory classes. We do so here only to facilitate the refactoring process. Later, we will replace this inline instantiation with injection.

Discover coupled variables through testing

We have now successfully decoupled the calling code from the `include` file, but this leaves us with a problem. Because the calling code executed the `include` code inline, the variables needed by the newly-extracted code are no longer available. We need to pass into the new class method all the variables it needs for execution, and to make its variables available to the calling code when the method is done.

To do so, we run the unit tests for the class that called the `include`. The tests will reveal to us what variables are needed by the new

method. We can then pass these into the method by reference. Using a reference makes sure that both blocks of code are operating on the exact same variables, just as if the `include` was still being executed inline. This minimizes the number of changes we need to make to the calling code and the newly extracted code.

For example, say we have extracted the code from an `include` file to this class and method:

classes/Validator/NewUserValidator.php

```
1 <?php
2 namespace Validator;
3
4 class NewUserValidator
5 {
6 public function __invoke()
7 {
8 $user_messages = array();
9 $user_is_valid = true;
10
11 if (! Validate::email($user['email'])) {
12 $user_messages[] = 'Email is not valid.';
13 $user_is_valid = false;
14 }
15
16 if (! Validate::strlen($foo['username'], 6, 8))
{
17 $user_messages[] = 'Username must be 6-8
characters long.';
18 $user_is_valid = false;
19 }
20
21 if ($user['password'] !==
$user['confirm_password']) {
22 $user_messages[] = 'Passwords do not match.';
23 $user_is_valid = false;
24 }
25 }
26 }
27 ?>
```

When we test the class that calls this code in place of an `include`, the tests will fail, because the `$user` value is not available to the new method, and the `$user_messages` and `$user_is_valid` variables are

not available to the calling code. We rejoice at the failure, because it tells us what we need to do next! We add each missing variable to the method signature by reference:

```
classes/Validator/NewUserValidator.php
1 <?php
2 public function __invoke(&$user,
&$user_messages, &$user_is_valid)
3 ?>
```

We then pass the variables to the method from the calling code:

```
classes/Validator/NewUserValidator.php
1 <?php
2 $validator->__invoke($user, $user_messages,
$user_is_valid);
3 ?>
```

We continue running the unit tests until they all pass, adding variables as needed. When all the tests pass, we rejoice! All the needed variables are now available in both scopes, and the code itself will remain decoupled and testable.

> **Note**
>
> Not all variables in the extracted code may be needed by the calling code, and vice versa. We should let the unit testing failures guide us as to which variables need to be passed in as references.

Replace other include Calls and Test

Now that we have decoupled our original calling code from the `include` file, we need to decouple all other remaining code from the same file. Given our earlier search results, we go to each file and replace the relevant `include` call with an inline instantiation of the

new class. We then add a line that calls the new method with the needed variables.

Note that we may be replacing code within classes, or within non-class files such as view files. If we replace code in a class, we should run the unit tests for that class to make sure the replacement does not break anything. If we replace code in a non-class file, we should run the test for that file if it exists (such as a view file test), or else spot check the file if no tests exist for it.

Delete the include file and test

Once we have replaced all `include` calls to the file, we delete the file. We should now run all of our tests and spot checks for the entire legacy application to make sure that we did not miss an `include` call to that file. If a test or spot check fails, we need to remedy it before continuing.

Write a test and refactor

Now that the legacy application works just as it used to before we extracted the `include` code to its own class, we write a unit test for the new class.

Once we have a passing unit test for the new class, we refactor the code in that class according to all the rules we have learned so far: no globals or superglobals, no use of the `new` keyword outside of factories, inject all needed dependencies, return values instead of generating output, and (recursively) no `include` calls. We continue to run our tests along the way to make sure we do not break any pre-existing functionality.

Convert to Dependency Injection and test

When the unit test for our newly refactored class passes, we proceed to replace all our inline instantiations with dependency

injection. We do so only in our class files; in our view files and other non-class files, the inline instantiation is not much of a problem

For example, we may see this inline instantiation and invocation in a class:

classes/Controller/NewUserPage.php

```
1 <?php
2 namespace Controller;
3
4 class NewUserPage
5 {
6 // ...
7
8 public function __invoke()
9 {
10 // ...
11 $user = $this->request->post['user'];
12
13 $validator = new \Validator\NewUserValidator;
14 $validator->__invoke($user, $user_messages, $u
15
16 if ($user_is_valid) {
17 $this->user_transactions->addNewUser($user
18 $this->response->setVars('success' => true
19 } else {
20 $this->response->setVars(array(
21 'success' => false,
22 'user_messages' => $user_messages
23 ));
24 }
25
26 return $this->response;
27 }
28 }
29 ?>
```

We move the `$validator` to a property injected via the constructor, and use the property in the method:

classes/Controller/NewUserPage.php

```
1 <?php
2 namespace Controller;
3
```

```
4 class NewUserPage
5 {
6 // ...
7
8 public function __construct(
9 \Mlaphp\Request $request,
10 \Mlaphp\Response $response,
11 \Domain\Users\UserTransactions
$user_transactions,
12 \Validator\NewUserValidator $validator
13 ) {
14 $this->request = $request;
15 $this->response = $response;
16 $this->user_transactions = $user_transactions;
17 $this->validator = $validator;
18 }
19
20 public function __invoke()
21 {
22 // ...
23 $user = $this->request->post['user'];
24
25 $this->validator->__invoke($user,
$user_messages, $user_is_valid);
26
27 if ($user_is_valid) {
28 $this->user_transactions->addNewUser($user);
29 $this->response->setVars('success' => true);
30 } else {
31 $this->response->setVars(array(
32 'success' => false,
33 'user_messages' => $user_messages
34 ));
35 }
36
37 return $this->response;
38 }
39 }
40 ?>
```

Now we need to search the codebase and replace every instantiation of the modified class to pass the new dependency object. We run our tests as we go to make sure everything continues to operate properly.

Commit, Push, Notify QA

At this point we have either replaced a single `include` call, or multiple `include` calls to the same file. Because we have been testing along the way, we can now commit our new code and tests, push it all to common repository, and notify QA that we have new work for them to review.

Do ... While

We begin again by searching for the next `include` call in a class file. When all `include` calls have been replaced by class method invocations, we are done.

Common QuestionsCan one class receive logic from many include files?

In the examples, we show the `include` code being extracted to a class by itself. If we have many related `include` files, it may be reasonable to collect them into the same class, each with their own method name. For example, the *NewUserValidator* logic might be only one of many user-related validators. We can reasonably imagine the class renamed as *UserValidator* with such methods as `validateNewUser()`, `validateExistingUser()`, and so on.

What about include calls originating in non-class files?

In our search for `include` calls, we look only in the `classes/` directory for the originating calls. It is likely that there are `include` calls that originate from other locations as well, such as the `views/`.

For the purposes of our refactoring, we don't particularly care about `include` calls that originate outside our classes. If an `include` is

called only from non-class files, we can safely leave that `include` in its existing state.

Our main goal here is to remove `include` calls from class files, not necessarily from the entire legacy application. At this point, it is likely that most or all `include` calls outside our classes are part of the presentation logic anyway.

Review and next steps

After we have extracted all the include calls from our classes, we will have finally removed one of the last major artifacts of our legacy architecture. We can load a class without any side effects, and logic is executed only as a result of invoking a method. This is a big step forward for us.

We can now begin paying attention to overarching end-to-end architecture of our legacy application.

As things stand now, the entire legacy application is still located in the web server document root. Users browse to each page script directly. This means that the URLs are coupled to the file system. In addition, each page script has quite a bit of repeated logic: load a setup script, instantiate a controller using dependency injection, invoke the controller, and send the response.

Our next major goal, then, is to begin using a Front Controller in our legacy application. The front controller will be composed of some bootstrapping logic, a router, and a dispatcher. This will decouple our application from the file system and allow us to start removing our page scripts entirely.

But before we do so, we need to separate the public resources in our application from the non-public resources.

Chapter 13. Separate Public and Non-Public Resources

At this point we have made major strides in reorganizing the core of our legacy application. However, the surrounding architecture still leaves much to be desired.

Among other things, our entire application is still embedded in the document root. This means that we need special protections on resources we intend to keep private, or that we need to rely on obscurity to make sure that clients do not browse to resources not intended to be public. Errors in web server configuration—or failure to attend to specific security measures—may reveal parts of our application to the public.

As such, our next step is to extract all public resources to a new document root. This will keep the non-public resources from being delivered by accident and will set up a structure for further refactoring.

Intermingled resources

Currently, our web server acts as a combined front controller, router, and dispatcher for our legacy application. The routes to the page scripts are mapped directly onto the file system, using the web server document root as a base. The web server document root, in turn, is mapped directly to the root of the legacy application.

For example, if the web server document root is `/var/www/htdocs`, it currently doubles as the application root. Thus, the URL path `/foo/bar.php` maps directly to `/var/www/htdocs/foo/bar.php`.

This may be fine for public resources, but there are large parts of our application that we do not want to be directly accessible by

outsiders. For example, directories related to configuration and setup should not be exposed to possible outside examination. An error in the web server configuration may reveal the code itself, making our passwords and other information available to malicious users.

The separation process

Although the process itself is straightforward, the change we are making is a foundational one. It affects the server configuration as well as the legacy application structure. To fully effect this change, we will need to coordinate closely with any operations personnel who are in charge of server deployments.

In general, the process is as follows:

1. Coordinate with operations to communicate our intentions.
2. Create a new document root directory in our legacy application, along with a temporary index file.
3. Reconfigure the server to point to the new document root directory, and spot check the new configuration to see if our temporary index file appears.
4. Remove the temporary index file, then move all public resources to the new document root, and spot check along the way.
5. Commit, push, and coordinate with operations for QA testing.

Coordinate with operations personnel

This is the single most important step in the process. We should never make changes that affect server configurations without discussing our intentions with the people in charge of the servers (our operations personnel).

The feedback from operations will inform us as to the path we need to follow for making sure our change will be effective. They will advise or instruct us as to what the new document root directory name should be, along with what the new server configuration directives should be. They are the ones in charge of deploying the application, so we want to do our best to make their job as easy as possible. If operations is unhappy, then everyone will be unhappy.

Alternatively, if we have no operations personnel and are in charge of our own deployments, our job is both easier and harder. It is easier because we have no coordination and communication costs. It is harder because we need specific, detailed knowledge about server configurations. Proceed carefully in this case.

Create a document root directory

After coordinating with our operations personnel, we create a document root directory in the legacy application structure. Our operations contacts will have advised us on a proper directory name; in this case, let us assume that name is `docroot/`.

For example, if we currently have a legacy application structure that looks like this:

var/www/htdocs/

```
classes/
 ...
css/
 ...
foo/
    bar/
        baz.php
images/
 ...
includes/
 ...
index.php
js/
tests/
 ...
views/
 ...
```

... we add a new `docroot/` directory at the top level of the application. In the new document root directory, we add a temporary `index.html` file. This will let us know, later, if our server reconfiguration works

properly. It can contain any text we like, such as `Rejoice! The new configuration works!`.

When we are done, the new directory structure will look something more like this:

/var/www/htdocs/

```
classes/
 ...
css/
 ...
docroot/
    index.html
foo/
    bar/
        baz.php
images/
 ...
includes/
 ...
index.php
    js/
 ...
tests/
 ...
views/
 ...
```

Reconfigure the server

We now reconfigure our local development web server to point to the new `docroot/` directory. Our operations personnel should have given us some instructions on how to do this.

In Apache, we might edit the configuration file for our local development environment to change the `DocumentRoot` directive in a related `.conf` file from the main application directory:

```
DocumentRoot "/var/www/htdocs"
```

... to our newly created subdirectory within the application:

```
DocumentRoot "/var/www/htdocs/docroot"
```

We then save the file, and reload or restart the server to apply our changes.

> **Tip**
>
> The applicable `DocumentRoot` directive may be in one of many locations. It could be in the main `httpd.conf` file, or perhaps inside a separate configuration file as part of a `VirtualHost` directive. If we are using something other than Apache, the configuration is probably in an entirely different file. Unfortunately, it is beyond the scope of this book to give full instructions on web server administration. Please review the documentation for your particular server for more information.

Once we have applied our configuration changes, we browse to our legacy application to see if the new document root is being honored. We should see the contents of our temporary `index.html` file. If not, we have done something wrong and need to revisit our changes until they work as expected.

Move public resources

Now that we have configured the web server to point to our new `docroot/` directory, we can safely remove our temporary `index.html` file.

After doing so, our next step is to move all of our public resources from their current locations into the new `docroot/` directory. This includes all of our page scripts, style sheets, JavaScript files, images, and so on. It does *not* include anything that users should not

be able to browse to: classes, includes, setup, configuration, command-line scripts, tests, view files, and so forth.

We want to maintain the same relative location in `docroot/` as they had when in the base of the application, so we should not change file names or directory names when moving.

As we move our public resources to their new location, we should occasionally spot check our modified structure by browsing through the application. This will help us discover any problems with our changes earlier rather than later.

Tip

Some of our moved PHP files may still depend on `include` files in specific locations. In these cases, we may need to modify them to point to paths relative to our new `docroot/` directory. Alternatively, we may need to modify our include-path values so that they can find the necessary files.

When we are done, we will have a directory structure that looks a little more like this:

/var/www/htdocs/

```
classes/
  ...
docroot/
   css/
       ...
   foo/
       bar/
          baz.php

   index.php
   js/
     ...
   images/
     ...
```

```
includes/
    ...
tests/
    ...
views/
    ...
```

Commit, push, coordinate

When we have moved all of our public resources to the new `docroot/` directory and the legacy application works properly in this new structure, we commit all of our changes and push them to the common repository.

At this point, we would normally notify QA of our changes for them to test. However, because we have made a foundational change to the server configuration, we need to coordinate the QA testing with our operations personnel. Operations will probably need to deploy the new configuration to the QA servers. Only then will QA be able to effectively check our work.

Common Questions

Is This Really Necessary?

Most of the time it seems harmless to leave the various non-public resources in the document root. But for our next step, it is going to be very important that we have a separation between our public resources and our non-public ones.

Review and next steps

We have now begun to refactor the overarching architecture of our legacy application. By creating a document root that separates our public resources from non-public ones, we can start to put together a front-controller system to control access to our application.

Chapter 14. Decouple URL Paths from File Paths

Even though we have a document root that separates our public and non-public resources, the users of our legacy application still browse directly to our page scripts. This means that our URLs are coupled directly to file system paths on the web server.

Our next step is to decouple the paths so that we can route URLs independently to any target we want. This means putting in place a Front Controller to handle all incoming requests for our legacy application.

Coupled Paths

As we noted in the previous chapter, our web server acts as a combined front controller, router, and dispatcher for our legacy application. The routes to the page scripts are still mapped directly onto the file system, using our `docroot/` directory as the base for all URL paths.

This presents us with some structural problems. For example, if we want to expose a new or different URL, we have to modify the location of the related page script in the file system. Similarly, we cannot change what page script responds to a particular URL. There is no way intercept the incoming request before it is routed.

These and other problems, including the ability to complete future refactorings, mean that we must create a single entry point for all incoming requests. This entry point is called a front controller.

In our first implementation of a front controller for our legacy application, we will add a Router to convert the incoming URL path to a page script path. That will allow us to remove our page scripts

from the document root, thereby decoupling the URLs from the file system.

The Decoupling Process

As with separating our public resources from our non-public ones, we will have to make a change to our web server configuration. Specifically, we will enable URL rewriting so we can point all incoming requests to a front controller. We need to coordinate this refactoring with our operations personnel so they can deploy the changes as easily as possible.

The process, in general, is as follows:

1. Coordinate with operations to communicate our intentions.
2. Create a front controller script in the document root.
3. Create a `pages/` directory for our page scripts, along with a `page not found` page script and controller.
4. Reconfigure the web server to enable URL rewriting.
5. Spot check the reconfigured web server to make sure the front controller and URL rewriting work properly.
6. Move all page scripts from `docroot/` to `pages/`, spot checking along the way.
7. Commit, push, and coordinate with operations for QA testing.

Coordinate with Operations

This is the single most important step in the process. We should never make changes that affect server configurations without discussing our intentions with the people in charge of the servers (i.e., our operations personnel).

In this case, we need to tell our operations personnel that we have to enable URL rewriting. They will advise or instruct us on how to do this for our particular web server.

Alternatively, if we have no operations personnel and are in charge of our own servers, we will need to determine on our own how to enable URL rewriting. Proceed carefully in this case.

Add a Front Controller

Once we have coordinated with our operations personnel, we will add a front controller script. We will also add a `page not found` script, controller, and view.

First, we create the front controller script in our document root. It uses a `Router` class to map the incoming URL to a page script. We call it front.php, or some other name that indicates it is a front controller:

```
docroot/front.php
1 <?php
2 // the router class file
3 require dirname(__DIR__) .
'/classes/Mlaphp/Router.php';
4
5 // set up the router
6 $pages_dir = dirname(__DIR__) . '/pages';
7 $router = new \Mlaphp\Router($pages_dir);
8
9 // match against the url path
10 $path = parse_url($_SERVER['REQUEST_URI'],
PHP_URL_PATH);
11 $route = $router->match($path);
12
13 // require the page script
14 require $route;
15 ?>
```

Note

We `require` the `Router` class file because the autoloader has not been registered yet. That will happen only when we execute the page script, which does not occur until the end of the front controller logic. We will remedy this situation in the next chapter.

Create a pages/ Directory

The front controller references a `$pages_dir`. The idea is that we will move all our page scripts out of the document root and into this new directory.

First, we make a `pages/` directory at the top level of our legacy application, next to the `classes/`, `docroot/`, `views/`, etc. directories.

We then create a `pages/not-found.php` script, along with a corresponding controller and view file. The front controller will call the `not-found.php` script when the `Router` is unable to match a URL path. The `not-found.php` script should set itself up like any other page script in our legacy application, then invoke its corresponding view file for the response:

pages/not-found.php

```
1 <?php
2 require '../includes/setup.php';
3
4 $request = new \Mlaphp\Request($GLOBALS);
5 $response = new
\Mlaphp\Response('/path/to/app/views');
6 $controller = new \Controller\NotFound($request,
$response);
7
8 $response = $controller->__invoke();
9
10 $response->send();
11 ?>
```

classes/Controller/NotFound.php

```
1 <?php
2 namespace Controller;
3
4 use Mlaphp\Request;
5 use Mlaphp\Response;
6
7 class NotFound
8 {
9 protected $request;
```

```
10
11 protected $response;
12
13 public function __construct(Request $request,
Response $response)
14 {
15 $this->request = $request;
16 $this->response = $response;
17 }
18
19 public function __invoke()
20 {
21 $url_path = parse_url(
22 $this->request->server['REQUEST_URI'],
23 PHP_URL_PATH
24 );
25
26 $this->response->setView('not-found.html.php');
27 $this->response->setVars(array(
28 'url_path' => $url_path,
29 ));
30
31 return $this->response;
32 }
33 }
34 ?>
```

views/not-found.html.php

```
1 <?php $this->header('HTTP/1.1 404 Not Found'); ?
>
2 <html>
3 <head>
4 <title>Not Found</title>
5 </head>
6 <body>
7 <h1>Not Found</h1>
8 <p><?php echo $this->esc($url_path); ?></p>
9 </body>
10 </html>
```

Reconfigure the Server

Now that we have our front controller in place and a target location for our page scripts, we reconfigure our local development web

server to enable URL rewriting. Our operations personnel should have given us some instructions on how to do this.

> ## Note
>
> Unfortunately, it is beyond the scope of this book to give full instructions on web server administration. Please review the documentation for your particular server for more information.

In Apache, we would first enable the `mod_rewrite` module. In some Linux distributions, this is as easy as issuing `sudo a2enmod` rewrite. In others, we need to edit the `httpd.conf` file to enable it.

Once URL rewriting is enabled, we need to instruct the web server to point all incoming requests to our front controller. In Apache, we might add a `docroot/.htaccess` file to our legacy application. Alternatively, we may modify one of the Apache configuration files for our local development server. The rewriting logic would look like the following:

docroot/.htaccess

```
1 # enable rewriting
2 RewriteEngine On
3
4 # turn empty requests into requests for the
"front.php"
5 # bootstrap script, keeping the query string
intact
6 RewriteRule ^$ front.php [QSA]
7
8 # for all files and dirs not in the document
root,
9 # reroute to the "front.php" bootstrap script,
10 # keeping the query string intact, and making
this
11 # the last rewrite rule
12 RewriteCond %{REQUEST_FILENAME} !-f
13 RewriteCond %{REQUEST_FILENAME} !-d
14 RewriteRule ^(.*)$ front.php [QSA,L]
```

Note

For example, if the incoming request is for `/foo/bar/baz.php`, the web server will invoke the `front.php` script instead. This will be the case for every request. The various superglobals values will remain unchanged, so `$_SERVER['REQUEST_URI']` will still indicate `/foo/bar/baz.php`.

Finally, after we have enabled URL rewriting, we restart or reload the web server to make our changes take effect.

Spot check

Now that we have enabled URL rewriting to point all requests to our new front controller, we should browse to our legacy application, using a URL path that we know does not exist. The front controller should show us the output from our `not-found.php` page script. This indicates that our changes are working properly. If not, we need to review and revise our changes up to this point and try to fix whatever went wrong.

Move Page scripts

Once we are sure that the URL rewriting and front controller are operating properly, we can begin to move all of our pages scripts out of `docroot/` and into our new `pages/` directory. Note that we are moving only page scripts. We should leave all the other resources in `docroot/`, including the `front.php` front controller.

For example, if we start out with this structure:

```
/path/to/app/
docroot/
css/
foo/
bar/
baz.php
```

```
front.php
images/
index.php
js/
pages/
not-found.php
```

We should end up with this structure instead:

```
/path/to/app/
docroot/
css/
front.php
images/
js/
pages/
foo/
bar/
baz.php
index.php
not-found.php
```

We have moved only the page scripts. Images, CSS files, Javascript files, and the front controller all remain in `docroot/`.

Because we are moving files around, we may need to change our include-path values to point to the new relative directory locations.

As we move each file or directory from `docroot/` to `pages/`, we should spot check our changes to make sure the legacy application continues to work correctly.

Due to the rewriting rules described earlier, our page scripts should continue to work whether they are in `docroot/` or `pages/`. We want to make sure to move all page scripts to `pages/` before we continue.

Commit, Push, Coordinate

When we have moved all of our page scripts to the new `pages/` directory, and our legacy application works properly in this new

structure, we commit all of our changes and push them to the common repository.

At this point, we would normally notify QA of our changes for them to test. However, because we have made a change to the server configuration, we need to coordinate the QA testing with our operations personnel. Operations will probably need to deploy the new configuration to the QA servers. Only then will QA be able to effectively check our work.

Common Questions

Did we really Decouple the Paths?

Astute observers will note that our *Router* still uses the incoming URL path to find the page scripts. The only difference between this and the original setup is that the path is mapped onto the `pages/` directory instead of the `docroot/` directory. Have we actually decoupled the URLs from the file system after all?

Yes, we have achieved our decoupling goal. This is because we now have an interception point between the URL path and the page script that gets executed. Using the *Router*, we could create an array of routes where URL paths are the keys and file paths are values. That mapping array would allow us to route the incoming URL path to any page script we like.

For example, if we want to map a URL path like `/foo/bar.php` to a page script like `/baz/dib.php`, we could do so via the `setRoutes()` method on the *Router*:

```
1 $router->setRoutes(array(
2 '/foo/bar.php' => '/baz/dib.php',
3 ));
```

Then when we `match()` the incoming URL path of `/foo/bar.php` against the *Router*, our returned route will be `/baz/dib.php`. We can

then execute that route as the page script for the incoming URL. We will use a variation on this technique in the next chapter.

Review and next steps

With the decoupling of our URLs from our page scripts, we are nearly finished with our modernization work. Only two refactorings remain. First, we will move repeated logic in our page scripts up to the front controller. Then we will remove the page scripts entirely and replace them with a dependency injection container.

Chapter 15. Remove Repeated Logic in Page Scripts

As things are now, the logic in our page scripts is highly repetitive. They all look very similar. Each one loads a setup script, instantiates a series of dependencies for a page controller, invokes that controller, and sends the response.

Our front controller gives us a place where we can execute the common elements of each page script and remove that repetition. Once the repetition has been removed, we can begin to eliminate the page scripts themselves.

Repeated logic

In essence, each of our page scripts follows this organizational flow:

```
Generic Page Script
1 <?php
2 // one or more identical setup scripts
3 require 'setup.php';
4
5 // a series of dependencies to build a
controller
6 $request = new \Mlaphp\Request($GLOBALS);
7 $response = new
\Mlaphp\Response('/path/to/app/views');
8 $controller = new \Controller\PageName($request,
$response);
9
10 // invoke the controller and send the response
11 $response = $controller->__invoke();
12 $response->send();
13 ?>
```

Because we have been diligent about always using the same variable name for our controller object (`$controller`), always using

the same method name for invoking it (`__invoke()`), and always using the same variable name for the response (`$response`), we can see that the only part of each page script that is different is the central section. That central block builds the controller object. Everything before and after is identical.

Further, because we have a front controller to handle all incoming requests, we now have a place to put the common before and after logic of every page script. That is what we will do here.

The Removal Process

In general, the removal process is as follows:

1. Modify the front controller to add setup, controller invocation, and response sending.
2. Modify each page script to remove the setup, controller invocation, and response sending.
3. Spot check, commit, push, and notify QA.

Modify the Front controller

First, we modify the front controller logic to perform the logic common to every page script. We change it from the code listed in the previous chapter to something more like this:

docroot/front.php

```
1 <?php
2 // page script setup
3 require dirname(__DIR__) .
'/includes/setup.php';
4
5 // set up the router
6 $pages_dir = dirname(__DIR__) . '/pages';
7 $router = new \Mlaphp\Router($pages_dir);
8
9 // match against the url path
10 $path = parse_url($_SERVER['REQUEST_URI'],
PHP_URL_PATH);
```

```
11 $route = $router->match($path);
12
13 // require the page script
14 require $route;
15
16 // invoke the controller and send the response
17 $response = $controller->__invoke();
18 $response->send();
19 ?>
```

We have replaced the line that requires the `Router` class file with a line that requires the setup script. (Way back in the chapter on autoloading, we put the autoloader into our setup script, so it should be autoloading the `Router` class for us now.)

We have also added two lines after requiring the file `$route` to the page script. These invoke the controller and set the response. We use the common variable names for the controller and response objects in this shared logic. (If you chose something other than `$controller` and `$response` in the page scripts, replace those in the above script. Similarly, if you used a common controller method other than `__invoke()`, replace that as well.)

> **Note**
>
> Note that the setup work is going to be specific to our legacy application. As long as the setup work is the same for every page script (which it should be at this point) placing the common setup work here will be just fine.

Remove Logic from Page Scripts

Now that we have added setup, controller invocation, and response-sending work to the front controller, we can remove that same work from each page script. Doing so should be as easy as doing a project-wide search in the `pages/` directory and deleting the found lines.

Finding the setup lines probably requires a regular expression, since the relative location of the setup scripts may result in lines that use relative directory traversals. The following regular expression will find `includes/setup.php`, `../includes/setup.php`, `dirname(__DIR__) . /includes/setup.php`, and so on:

Search for setup:

```
1 ^\s*(require|require_once|include|include_once)
.*includes/setup\.php.*$
```

However, finding the controller invocation and response-sending lines should not require a regular expression, since they should be identical in every page script.

Search for controller invocation …

```
1 $response = $controller->__invoke();
```

Search for response sending …

```
1 $response->send();
```

In each case, delete the found line. It is no longer needed now that the logic has been moved to the front controller.

Spot Check, Commit, Push, Notify QA

Once the repeated page script logic has been removed in favor of the same logic placed in the front controller, we can spot check the application. We do so by running our characterization tests if they exist, or by browsing to or otherwise invoking each page in the application if they do not.

After we are sure that the application still works properly, we commit our new code and push it to the common repository. Then we notify QA that we have new work for them to review.

Common Questions

What if the Setup Work Is Inconsistent?

In the examples throughout this book, we have shown only a single script doing the setup work for each page script. Some legacy applications may use more than one setup script. As long as the setup work is identical across each page script, even if it is composed of more than one script, then we can move all the setup work to the front controller.

However, if the setup work is not identical across each page script, we have a problem to deal with. If the page scripts do not enjoy an identical setup process at this point, we should do what we can to address that before continuing.

It is imperative that we make the setup work identical in all page scripts. This may mean including all the different setup work from all page scripts in the front controller, even if some scripts don't need all that setup work. We can remedy this overlap the next chapter if necessary.

If we cannot enforce an identical single-stage setup process, we may have to pursue a dual or two-stage setup process. First, we consolidate common setup work into the front controller and remove it from the page scripts. Extraneous, special-case, or page-specific setup work can remain with the page script as a degenerate but necessary part of the dependency creation work.

What if we used inconsistent naming?

In previous chapters, this book emphasizes the importance of consistent naming. This chapter is the point at which that consistency pays off.

If we discover we have been inconsistent in our naming of the controller object variable and/or the controller method name, all is not lost. We will not be able to do a one-pass search-and-replace, but we can still work through each page script manually and change the names to be consistent. Then the newly consistent names can be used by the front controller.

Review and next steps

With this step, we have reduced our page scripts to a bare core of logic. All they do now is set up dependencies for, and then create, a controller object. The front controller does everything before that, and everything after.

As it happens, even this logic can be extracted from the page scripts. An object called a dependency injection container can receive the object creation logic as series of closures, one closure per page script. The container can handle the object creation for us and we can remove the page scripts entirely.

Thus, our final refactoring will extract all object creation logic to a dependency injection container. We will also modify our front controller to instantiate controller objects instead of requiring page scripts. In doing so, we will have removed all of our page scripts and our application will have a fully modernized architecture.

Chapter 16. Add a Dependency Injection Container

We have reached the final step in our modernization process. We will remove the last vestiges of our page scripts by moving their remaining logic into a dependency injection container. The container will be responsible for coordinating all the object creation activity in our application. In doing so, we will modify our front controller again, and begin adding routes that point to controller classes instead of file paths.

> **Note**
>
> For this final step in the modernization process, it is best if we have PHP 5.3 or later installed. This is because we need closures for critical parts of the application logic. If we do not have access to PHP 5.3, there is a less viable but still workable option for implementing a dependency injection container. We address that situation as the last of the "Common Questions" in this chapter.

What is a Dependency Injection Container?

Dependency injection as a technique is something we have been practicing since early in this book. To reiterate, the idea behind dependency injection is that we push dependencies into an object from the outside. This is as opposed to creating dependency objects while inside a class via the new keyword, or reaching out of the current scope to bring in dependencies via the `globals` keyword.

Note

For an overview of inversion of control in general and dependency injection in specific, read Fowler's article on containers at http://martinfowler.com/articles/injection.html.

To accomplish our dependency injection activities, we have been manually creating the necessary objects in a page script. For any object that needed a dependency, we created that dependency first, then we created the object that depended on it and passed in the dependency. This creation process has sometimes been deeply layered, as when the dependencies have dependencies. Regardless of the complexity and depth, the logic for doing so is currently embedded in page scripts.

The idea behind a dependency injection *container* is to keep all that object creation logic in a single place, so that we are no longer required to use a page script to set up our objects. We can place each piece of object creation logic in the container under a unique name, called a service.

We can then tell the container to return a new instance of any defined service object. Alternatively, we can tell the container to create and return a shared instance of that service object, so that each time we get it, it is always the same instance. Careful combinations of new instances and shared instances of container services will allow us to pare down our dependency creation logic.

Note

At no point will we be passing the container into any of the objects that need dependencies. To do so would be using a pattern called Service Locator. We avoid Service Locator activity because doing so is a violation of scope. When the container is inside an object, and that object uses it to retrieve dependencies,

we are only one step removed from where we started; that is, with the `global` keyword. As such, we do not pass the container around -- it stays entirely outside the scope of the objects it creates.

There are many different container implementations in PHP land, each with its own strengths and weaknesses. To keep things tailored to our modernization process, we will use the *Mlaphp\Di*. This is a stripped down container implementation that is well-suited to our transitional needs.

Adding a DI Container

The process for adding a DI container, in general, is as follows:

1. Add a new `services.php` include file to create the container and manage its services.
2. Define a `router` service in the container.
3. Modify the front controller to include the `services.php` file and use the `router` service, then spot check the application.
4. Extract creation logic from each page script to the container:

    1. Create a service in the container named for the page script controller class.
    2. Copy the logic from the page script into the container service. Rename variables as needed to use DI container properties.
    3. Route the page URL path to the container service name (i.e., the controller name).
    4. Spot check and commit the change.
    5. Continue until all page scripts have been extracted to the container.

5. Remove the empty `pages/` directory, commit, push, and notify QA.

Add a DI Container Include File

To keep our existing setup files from growing even larger, we will introduce a new `services.php` setup file. Yes, this means adding another `include` to the front controller, but if we have been diligent, there are few if any includes remaining in our application. This one will be of little import.

First, we need to pick an appropriate location for the file. It is probably best if it goes along with any other setup files we already have, perhaps in an existing `includes/` directory.

Then we create the file with the following line. (We will add much more to this file as we continue.) Because the file will be loaded as the last of our setup files, we can presume that autoloading will be active, so there is no need to load the `Di` class file:

includes/services.php

```
1 <?php
2 $di = new \Mlaphp\Di($GLOBALS);
3 ?>
```

What happens as a result is that the new `$di` instance is loaded with all the existing global variable values. These values are retained as properties on the container. For example, if our setup files create a `$db_user` variable, we can now additionally access that value as `$di->db_user`. These are copies, not references, so changes to one will not affect the other.

Note

Why do we retain the existing variables as properties?

Currently, our page scripts access the global variables directly for their creation work. However, in a later step, the creation logic will no longer be in the global scope. It will be "inside" the DI

container. Thus, we populate the DI container with a copy of the variables that would have been available otherwise.

Add a Router Service

Now that we have a DI container in place, let's add our first service.

Recall that the purpose of a DI container is to create objects for us. Currently, the front controller creates a *Router* object, so we will add a `router` service to the container. (In the next step, we will have the front controller use this service instead of creating a *Router* on its own.)

In the `services.php` file, add the following lines:

includes/services.php

```
1 <?php
2 // set a container service for the router
3 $di->set('router', function () use ($di) {
4 $router = new
\Mlaphp\Router('/path/to/app/pages');
5 $router->setRoutes(array());
6 return $router;
7 });
8 ?>
```

Let's examine the service defination a little bit.

- The service name is `router`. We will use all-lowercase names for service objects intended to be created once as shared instances, and fully-qualified class names for service objects intended to be created as new instances each time. Thus, in this case, our intent is that only a single shared `router` will be available via the container. (This is a convention, not a rule that is enforced by the container.)
- The service definition is a callable. In this case, it is a closure. The closure receives no parameters, but it does use the `$di` object from the current scope. This makes it possible for the

definition code to access container properties and other container services while building the service object.
- We create and then return the object represented by the service name. We do not need to check if the object already exists in the container; the container internals will do that for us if we ask for a shared instance.

With this bit of code, the container now knows how to create a `router` service. It is lazy-loaded code that will only be executed when we call `$di->newInstance()` (to get a new instance of the service object) or `$di->get()` (to get a shared instance of the service object).

Modify the Front Controller

Now that we have a DI container and a `router` service definition, we modify the front controller to load the container and use the `router` service.

```
docroot/front.php
1 <?php
2 require dirname(__DIR__) .
'/includes/setup.php';
3 require dirname(__DIR__) .
'/includes/services.php';
4
5 // get the shared router service
6 $router = $di->get('router');
7
8 // match against the url path
9 $path = parse_url($_SERVER['REQUEST_URI'],
PHP_URL_PATH);
10 $route = $router->match($path);
11
12 // container service, or page script?
13 if ($di->has($route)) {
14 // create a new $controller instance
15 $controller = $di->newInstance($route);
16 } else {
17 // require the page script
18 require $route;
19 }
20
```

```
21 // invoke the controller and send the response
22 $response = $controller->__invoke();
23 $response->send();
24 ?>
```

We have made the following changes from the previous implementation:

- We added a `require` for the `services.php` container file as the very last of our setup includes.
- Instead of creating a *Router* object directly, we `get()` a shared instance of the `router` service object from the `$di` container.
- We have changed our dispatching logic somewhat. After we get a `$route` from the `$router`, we check to see if the `$di` container `has()` a matching service. If so, it treats the `$route` as a service name for a new `$controller` instance; otherwise, it treats the `$route` as a file in `pages/` that creates a `$controller`. Either way, the code then invokes the controller and sends the response.

After these changes, we spot check the application to make sure the new `router` service works properly. If it does not, we undo and redo our changes up to this point until the application works as it did before.

We may wish to commit our changes once the application works. This is so that if future changes go bad, we have a known-working state to which we can revert.

Extract Page Scripts to Services

Now comes the final push in modernizing our legacy application. We are going to remove our page scripts one-by-one and put their logic into the container.

Create a Container Service

Pick any page script and determine what class it uses to create its `$controller` instance. Then, in the DI container, create an empty

service definition for that class name.

For example, if we have this page script:

```
pages/articles.php
1 <?php
2 $db = new Database($db_host, $db_user,
$db_pass);
3 $articles_gateway = new ArticlesGateway($db);
4 $users_gateway = new UsersGateway($db);
5 $article_transactions = new ArticleTransactions(
6 $articles_gateway,
7 $users_gateway
8 );
9 $response = new
\Mlaphp\Response('/path/to/app/views');
10 $controller = new \Controller\ArticlesPage(
11 $request,
12 $response,
13 $user,
14 $article_transactions
15 );
16 ?>
```

We the controller class being instantiated is `Controller\ArticlesPage`. In our `services.php` file, we create an empty service definition with that name:

```
includes/services.php
1 <?php
2 $di->set('Controller\ArticlesPage', function ()
use ($di) {
3 });
4 ?>
```

Next, we move the page script setup logic into the service definition. When we do so, we should note any variables that we expected from the global scope, and prefix them with `$di->` to reference the appropriate container properties. (Recall that these were loaded from `$GLOBALS` early in the `services.php` file.) We also return the controller instance at the end of the definition.

When we are done, the service definition will look something like this:

includes/services.php

```
<?php
$di->set('Controller\ArticlesPage', function ()
use ($di) {
// replace `$variables` with `$di->` properties
$db = new Database($di->db_host, $di->db_user,
$di->db_pass);
// create dependencies
$articles_gateway = new ArticlesGateway($db);
$users_gateway = new UsersGateway($db);
$article_transactions = new ArticleTransactions(
$articles_gateway,
$users_gateway
);
$response = new
\Mlaphp\Response('/path/to/app/views');
// return the new instance
return new \Controller\ArticlesPage(
$request,
$response,
$user,
$article_transactions
);
});
?>
```

Once we have copied the logic over to the container, we delete the orginal page script file from `pages/`.

Route the URL Path to the Container Service

Now that we have removed the page script in favor of the container service, we need to make sure the *Router* points to the container service instead of the now-missing page script. We do this by adding an array element to the `setRoutes()` method parameter where the key is the URL path and the value is the service name.

For example, if the URL path is `/articles.php` and our new container service is named `Controller\ArticlesPage`, we would modify our

`router` service like so:

```
includes/services.php
1 <?php
2 // ...
3 $di->set('router', function () use ($di) {
4 $router = new \Mlaphp\Router($di->pages_dir);
5 $router->setRoutes(array(
6 // add a route that points to a container
service name
7 '/articles.php' => 'Controller\ArticlesPage',
8 ));
9 return $router;
10 });
11 ?>
```

Spot Check and Commit

Finally, we check to see if the conversion from page script to container service works as we expect. We spot check the URL path to the old page script by browsing to or otherwise invoking that URL. If it works, then we know the container service has successfully taken the place of the now-deleted page script.

If not, we need to undo and redo our changes to see where things went wrong. The most common errors I see here are:

- Failure to replace `$var` variables in the page script with `$di->var` properties in the service definition
- Failure to return the object from the service definition
- Mismatches between the controller service name and the mapped route value

Once we are sure the application routes the URL to the new container service, and that the service works properly, we commit our changes.

Do ... While

We proceed to the next page script and begin the process over again. When all page scripts have been converted to container services and then deleted, we are done.

Remove pages/, Commit, Push, Notify QA

After we have extracted all of our page scripts to the DI container, the `pages/` directory should be empty. We can now safely remove it.

With that, we commit our work, push to the common repository, and notify QA that we have new changes for them to review.

Common Questions

How can we refine our service definitions?

When we are done extracting our object creation logic to the container, each service definition is likely to be rather long, and probably repetitive. It would be nice to reduce the repetition and refine the service definition so as to make them short and succinct. We can do so by further extracting each part of object creation logic to its own service.

For example, if we have several services that use a *Request* object, we can extract the object creation logic to its own service and then reference that service in other services. We can name it to show our intent that it be used as can be intended as a shared service (`request`) or as a new instance (`Mlaphp\Request`). Other services can then use `get()` or `newInstance()` instead of creating the request internally.

Given our earlier `Controller\ArticlesPage` service, we could split it up into several reusable services like so:

```
includes/services.php
1 <?php
2 // ...
```

```
3
4 $di->set('request', function () use ($di) {
5 return new \Mlaphp\Request($GLOBALS);
6 });
7
8 $di->set('response', function () use ($di) {
9 return new
\Mlaphp\Response('/path/to/app/views');
10 });
11
12 $di->set('database', function () use ($di) {
13 return new \Database(
14 $di->db_host,
15 $di->db_user,
16 $di->db_pass
17 );
18 });
19
20 $di->set('Domain\Articles\ArticlesGateway',
function () use ($di) {
21 return new
\Domain\Articles\ArticlesGateway($di-
>get('database'));
22 });
23
24 $di->set('Domain\Users\UsersGateway', function
() use ($di) {
25 return new \Domain\Users\UsersGateway($di-
>get('database'));
26 });
27
28 $di->set('Domain\Articles\ArticleTransactions',
function () use ($di) {
29 return new
\Domain\Articles\ArticleTransactions(
30 $di-
>newInstance('Domain\Articles\ArticlesGateway'),
31 $di->newInstance('Domain\Users\UsersGateway'),
32 );
33 });
34
35 $di->set('Controller\ArticlesPage', function ()
use ($di) {
36 return new \Controller\ArticlesPage(
37 $di->get('request'),
38 $di->get('response'),
39 $di->user,
40 $di-
```

```
>newInstance('Domain\Articles\ArticleTransactions'
)
41 );
42 });
43 ?>
```

Notice how the `Controller\ArticlesPage` service now references other services in the container to build its own object. When we get a new instance of the `Controller\ArticlesPage` service object, it addresses the `$di` container to obtain the shared request and response objects, the `$user` property, and a new instance of the *ArticleTransactions* service object. That, in turn, recursively addresses the `$di` container to obtain the dependencies for that service object, and so on.

What if there are includes In the Page Script?

Even though we have done our best to remove them, it is possible that we still have some include files in our page scripts. When we copy the page script logic to the container, we have little choice but to copy them as well. However, once all our page scripts have been converted to the container, we can look for commonalities and begin extracting the include logic either to a setup script or to separate classes (which themselves can become services if needed).

Can we reduce the size of the services.php file?

Depending on the number of page scripts in our application, our DI container may end up with tens or hundreds of service definitions. This can be a lot to manage or scan through in a single file.

If we like, it is perfectly reasonable to split the container into multiple files, and make the `services.php` a series of include calls to bring in the various definitions.

Can we reduce the size of the router service?

As a subset of the DI container file length, the `router` service in particular is likely to become very long. This is because we map every URL in the application to a service; if there are hundreds of URLs, there will be hundreds of `router` lines.

As an alternative, we can create a separate `routes.php` file and have it return an array of routes. We can then include that file in the `setRoutes()` call:

includes/routes.php

```
1 <?php return array(
2 '/articles.php' => 'Controller\ArticlesPage',
3 ); ?>
```

includes/services.php

```
1 <?php
2 // ...
3 $di->set('router', function () use ($di) {
4 $router = new \Mlaphp\Router($di->pages_dir);
5 $router->setRoutes(include
'/path/to/includes/routes.php');
6 return $router;
7 });
8 ?>
```

That at least will reduce the size of the `services.php` file, even though it does not reduce the size of the routes array.

What if we cannot update to PHP 5.3?

The examples in this chapter show a DI container that uses closures to encapsulate object creation logic. Closures only became available in PHP 5.3, so if we are stuck on an earlier version of PHP, it looks like using a DI container is simply not an option.

This turns out not to be true. With some extra effort and a greater toleration for inelegance, we can still build a DI container for PHP 5.2 and earlier.

First, we need to extend the DI container so that we can add methods to it. Then, instead of creating service definitions as closures, we create them as methods on our extended container:

classes/Di.php

```
1 <?php
2 class Di extends \Mlaphp\Di
3 {
4 public function database()
5 {
6 return new \Database(
7 $this->db_host,
8 $this->db_user,
9 $this->db_pass
10 );
11 }
12 }
13 ?>
```

(Notice how we use `$this` instead of `$di` in the method.)

Then in our `services.php` file, the callable becomes a reference to this method, instead of an inline closure:

includes/services.php

```
1 <?php
2 $di->set('database', array($di, 'database'));
3 ?>
```

This is messy but workable. It can also get pretty verbose. Our earlier example of splitting up the `Controller\ArticlesPage` ends up looking more like this:

includes/services.php

```
1 <?php
2 // ...
3 $di->set('request', array($di, 'request'));
4 $di->set('response', array($di, 'response'));
5 $di->set('database', array($di, 'database'));
6 $di->set('Domain\Articles\ArticlesGateway',
array($di, 'ArticlesGateway'));
7 $di->set('Domain\Users\UsersGateway', array($di,
```

```
'UsersGateway'));
8 $di->set(
9 'Domain\Articles\ArticleTransactions',
10 array($di, 'ArticleTransactions')
11 );
12 $di->set('Controller\ArticlesPage', array($di,
'ArticlesPage'));
13 ?>
```

classes/Di.php

```
1 <?php
2 class Di extends \Mlaphp\Di
3 {
4 public function request()
5 {
6 return new \Mlaphp\Request($GLOBALS);
7 }
8
9 public function response()
10 {
11 return new
\Mlaphp\Response('/path/to/app/views');
12 }
13
14 public function database()
15 {
16 return new \Database(
17 $this->db_host,
18 $this->db_user,
19 $this->db_pass
20 );
21 }
22
23 public function ArticlesGateway()
24 {
25 return new
\Domain\Articles\ArticlesGateway($this-
>get('database'));
26 }
27
28 public function UsersGateway()
29 {
30 return new \Domain\Users\UsersGateway($this-
>get('database'));
31 }
32
33 public function ArticleTransactions()
```

```
34 {
35 return new
\Domain\Articles\ArticleTransactions(
36 $this->newInstance('ArticlesGateway'),
37 $this->newInstance('UsersGateway'),
38 );
39 }
40
41 public function ArticlesPage()
42 {
43 return new \Controller\ArticlesPage(
44 $this->get('request'),
45 $this->get('response'),
46 $this->user,
47 $this->newInstance('ArticleTransactions')
48 );
49 }
50 }
51 ?>
```

Unfortunately, we may have to break with some of our style conventions to keep the service names looking like their related method names. We also have to shorten the service method names intended for new instances down to their ending class names, instead of their fully qualified names. Otherwise we find ourselves with overly long and confusing method names.

This can get confusing fast, but it does work. In all, it really is better if we can upgrade to PHP 5.3 or later.

Review and next steps

At last, we have completed our modernization process. We no longer have any page scripts. All of our application logic has been converted to classes, and the only remaining include files are part of the bootstrap and setup process. All of our object creation logic exists inside a container, where we can modify it directly without having to disturb the internals of our objects.

What could the next step possibly be after this? The answer is continuous improvement and it is going to last for the rest of your career.

Chapter 17. Conclusion

Let us think back on our progress.

We started out with a spaghetti mess of a legacy application. The entire application was based in the document root, and users browsed directly to page scripts. It used an include-oriented architecture, where merely including a file caused logic to be executed. Global variables were scattered everywhere, which made debugging difficult to impossible. There were no tests of any sort, much less unit tests, so every change raised the possibility of breaking something else. There was no clear separation of model, view, and controller layers. SQL statements were embedded throughout our code, and domain logic was mixed in with presentation and action logic.

Now, after a great deal of effort, dedication, and commitment, we have modernized our legacy application. The document root consists only of public resources and a front controller. All the page scripts have been distilled into separate model, view, and controller layers. These layers are represented by a well-structured series of classes, each with its own set of unit tests. The application objects are built inside a dependency injection container, keeping their operation separate from their creation.

What could possibly be left to do?

Opportunities for improvement

Even though we have modernized our application, it is still not perfect. Frankly, it will *never* be perfect (whatever that may happen to mean). There will always be some opportunity to improve it. Indeed, the modernization process itself has revealed many opportunities for us.

- The data source layer is composed of a series of Gateways. While they serve our purpose nicely for now, it may be better to restructure these as Data Mappers that interact more cleanly with our domain objects.
- The domain layer is built on top of Transaction Scripts. These, too, are fine in their own way, but as we work with them, we may realize that they are insufficient for our needs. They combine too many aspects of our domain logic into monolithic classes and methods. We will probably want to begin to tease out the different aspects of our domain logic into a Domain Model and wrap it with a series of Service Layers.
- The presentation layer is still relatively monolithic in nature. We may want to convert our view files to a Two Step View system. That will give us a unified layout across the entire application, provide a reusable series of "partial" templates, and help us reduce each view file to its central core.
- Our controllers may be handling several unrelated actions as an artifact of the legacy architecture. We may wish to reorganize them into more sensible groups for quicker understanding. Indeed, each controller may be doing too much work (i.e., a fat controller instead of a skinny one) that could better be handled by helper classes or a Service Layer.
- The response system combines the concerns of content building with HTTP response building. We may wish to refactor the entire response-sending process to two or more separate layers: one that deals with the body of the response and one that deals with the response headers. Indeed, we may wish to represent the response as a Data Transfer Object that describes our intentions, but leaves the actual response building-and-sending to a separate handler.
- The routing system is definitely transitional. We probably still depend on query parameters in the URL to pass client request information into the application, instead of using "pretty URLs" where the parameters are represented as parts of the path information. The routes themselves merely describe the class to be invoked, and do not carry as much information as they should about the action that the application should be executing.

We will want to replace this basic router with a more powerful one.
- The front controller is acting as our dispatcher, instead of handing off route dispatch to a separate object. We may wish to separate the task of discovering the route information from the task of dispatching that route.
- Finally, our dependency injection container is very "manual" in nature. We may wish to find a container system that automates some of the more basic aspects of object creation, allowing us to concentrate on the more complex aspects of service definition.

In other words, we have the problems of a modern codebase, not a legacy one. We have traded the low-quality problems of a spaghetti mess for the "high-quality" problems of an autoloaded, dependency-injected, unit-tested, layer-separated, front-controlled codebase.

Because we have modernized our codebase, we can address these problems in a completely different way than we did under a legacy regime. We can bring the tools of Refactoring to bear against the code. Now that we have better separation of concerns, we can make small changes in limited sections of the code to improve the quality of that code. Each change can be tested for regressions via our unit test suite.

Each new feature that we add can be plugged into our new application architecture using the techniques we have acquired while modernizing it. No longer do we toss in a new page script copied-and-modified from some previous one. Instead, we add a unit-tested controller class and route to it through the front controller. New functionality in our domain logic is added as a unit-tested class or method in the domain layer. Changes and additions to presentation can be tested separately from our models and controllers via our view layer.

Conversion to Framework

There is also the possibility of converting our application over to the latest, hottest framework. Although switching to a public framework is a little bit like an application rewrite, it should be much easier now that we have a well-separated series of application layers. We should be able to determine which parts would be ported over to a public framework, and which are merely incidental to the operation of our particular architecture.

I advise neither for nor against this approach. I will only point out that, in the course of modernizing our legacy application, we have essentially built our own custom-tailored framework. We have done so with what is probably a more disciplined and rigorous approach than most public frameworks in PHP land. While we gain the community that goes along with a public framework, we also gain the baggage of the framework developers themselves. These and other tradeoffs are beyond my ability to judge on your behalf; you will have to decide for yourself if the benefits outweigh the costs.

Review and next steps

Regardless of how we proceed from here, there can be no doubt that the improvement of the *application* has led to an improvement in our quality of life and in our professional approach. The time we have invested in the code has paid off not only in terms of our employment, where we now spend less time feeling frustrated and demoralized, and more time feeling competent and productive. It has paid off in terms of our skills, our knowledge, and our experience regarding application architecture, patterns, and practices.

Our goals now are to continue improving our code, to continue improving ourselves, and to help others to improve as well. We need to share our knowledge. In doing so, we will reduce the amount of suffering in the world that comes from having to deal with legacy applications. As more people learn to apply what we have learned here, we can ourselves go on to bigger and better and more interesting issues in our professional lives.

So go forth and spread the good news to your colleagues, compatriots, and co-workers, that they need not suffer with legacy applications if they do not want to. They, can also modernize their codebases, and improve their own lives in doing so.

Appendix A. Typical Legacy Page Script

```
<?php
2 include("common/db_include.php");
3 include("common/functions.inc");
4 include("theme/leftnav.php");
5 include("theme/header.php");
6
7 define("SEARCHNUM", 10);
8
9 function letter_links()
10 {
11 global $p, $letter;
12 $lettersArray = array(
13 '0-9', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H',
'I',
14 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R',
'S',
15 'T', 'U', 'V', 'W', 'X', 'Y', 'Z'
16 );
17 foreach ($lettersArray as $let) {
18 if ($letter == $let)
19 echo $let.' ';
20 else
21 echo '<a class="letters" '
22 . 'href="letter.php?p='
23 . $p
24 . '&letter='
25 . $let
26 . '">'
27 . $let
28 . '</a> ';
29 }
30 }
31
32 $page = ($page) ? $page : 0;
33
34 if (!empty($p) && $p!="all" && $p!="none") {
35 $where = "`foo` LIKE '%$p%'";
36 } else {
37 $where = "1";
38 }
```

```
39
40 if ($p=="hand") {
41 $where = "`foo` LIKE '%type1%'"
42 . " OR `foo` LIKE '%type2%'"
43 . " OR `foo` LIKE '%type3%'";
44 }
45
46 $where .= " AND `bar`='1'";
47 if ($s) {
48 $s = str_replace(" ", "%", $s);
49 $s = str_replace("'", "", $s);
50 $s = str_replace(";", "", $s);
51 $where .= " AND (`baz` LIKE '%$s%')";
52 $orderby = "ORDER BY `baz` ASC";
53 } elseif ($letter!="none" && $letter) {
54 $where .= " AND (`baz` LIKE '$letter%'"
55 . " OR `baz` LIKE 'The $letter%')";
56 $orderby = "ORDER BY `baz` ASC";
57 } else {
58 $orderby = "ORDER BY `item_date` DESC";
59 }
60 $query = mysql_query(
61 "SELECT * FROM `items` WHERE $where $orderby
62 LIMIT $page,".SEARCHNUM;
63 );
64 $count = db_count("items", $where);
65 ?>
66
67 <td align="middle" width="480" valign="top">
68 <img border="0" width="480" height="30"
69 src="http://example.com/images/example1.gif">
70 <table border="0" cellspacing="0" width="480"
71 cellpadding="0" bgcolor="#000000">
72 <tr>
73 <td colspan="2" width="480" height="50">
74 <img border="0"
75 src="http://example.com/images/example2.gif">
76 </td>
77 </tr>
78 <tr>
79 <td width="120" align="right" nowrap>
80 <img border="0"
81 src="http://example.com/images/example3.gif">
82 </td>
83 <td width="360" align="right" nowrap>
84 <div class="letter"><?php letter_links(); ?>
</div>
85 </td>
```

```
86 </tr>
87 </table>
88
89 <form name="search" enctype="multipart/form-
data"
90 action="search.php" method="POST" margin="0"
91 style="margin: 0px;">
92 <table border="0" style="border-collapse:
collapse"
93 width="480" cellpadding="0">
94 <tr>
95 <td align="center" width="140">
96 <input type="text" name="s" size="22"
97 class="user_search" title="enter your
search..."
98 value="<?php
99 echo $s
100 ? $s
101 : "enter your search..."
102 ;
103 ?>" onFocus=" enable(this); "
104 onBlur=" disable(this); ">
105 </td>
106 <td align="center" width="70">
107 <input type="image" name="submit"
108
src="http://example.com/images/user_search.gif"
109 width="66" height="17">
110 </td>
111 <td align="right" width="135">
112 <img border="0"
113 src="http://example.com/images/list_foo.gif"
114 width="120" height="26">
115 </td>
116 <td align="center" width="135">
117 <select size="1" name="p"
onChange="submit();">
118 <?php
119 if ($p) {
120 ${$p} = 'selected="selected"';
121 }
122 foreach ($foos as $key => $value) {
123 echo '<option value="'
124 . $key
125 . '" '
126 . ${$key}
127 . '>'
128 . $value
```

```
129 . '</option>';
130 }
131 ?>
132 </select>
133 </td>
134 </tr>
135 </table>
136 <?php if ($letter) {
137 echo '<input type="hidden" name="letter" '
138 . 'value="' . $letter . '">';
139 } ?>
140 </form>
141
142 <table border="0" cellspacing="0" width="480"
143 cellpadding="0" style="border-style: solid;
border-color:
144 #606875; border-width: 1px 1px 0px 1px;">
145 <tr>
146 <td>
147 <div class="nav"><?php
148 $pagecount = ceil(($count / SEARCHNUM));
149 $currpage = ($page / SEARCHNUM) + 1;
150 if ($pagecount)
151 echo ($page + 1)
152 . " to "
153 . min(($page + SEARCHNUM), $count)
154 . " of $count";
155 ?></div>
156 </td>
157 <td align="right">
158 <div class="nav"><?php
159 unset($getstring);
160 if ($_POST) {
161 foreach ($_POST as $key => $val) {
162 if ($key != "page") {
163 $getstring .= "&$key=$val";
164 }
165 }
166 }
167 if ($_GET) {
168 foreach ($_GET as $key => $val) {
169 if ($key != "page") {
170 $getstring .= "&$key=$val";
171 }
172 }
173 }
174
175 if (!$pagecount) {
```

```
176 echo "No results found!";
177 } else {
178 if ($page >= (3*SEARCHNUM)) {
179 $firstlink = " | <a class=\"searchresults\"
180 href=\"?page=0$getstring\">1</a>";
181 if ($page >= (4*SEARCHNUM)) {
182 $firstlink .= " ... ";
183 }
184 }
185
186 if ($page >= (2*SEARCHNUM)) {
187 $prevpages = " | <a class=\"searchresults\""
188 . " href=\"?page="
189 . ($page - (2*SEARCHNUM))
190 . "$getstring\">"
191 . ($currpage - 2)
192 ."</a>";
193 }
194
195 if ($page >= SEARCHNUM) {
196 $prevpages .= " | <a class=\"searchresults\""
197 . " href=\"?page="
198 . ($page - SEARCHNUM)
199 . "$getstring\">"
200 . ($currpage - 1)
201 . "</a>";
202 }
203
204 if ($page==0) {
205 $prevlink = "&laquo; Previous";
206 } else {
207 $prevnum = $page - SEARCHNUM;
208 $prevlink = "<a class=\"searchresults\""
209 . " href=\"?page=$prevnum$getstring\">"
210 . "&laquo; Previous</a>";
211 }
212
213 if ($currpage==$pagecount) {
214 $nextlink = "Next &raquo;";
215 } else {
216 $nextnum = $page + SEARCHNUM;
217 $nextlink = "<a class=\"searchresults\""
218 . " href=\"?page=$nextnum$getstring\">"
219 . "Next &raquo;</a>";
220 }
221
222 if ($page < (($pagecount - 1) * SEARCHNUM))
223 $nextpages = " | <a class=\"searchresults\""
```

```
224 . " href=\"?page="
225 . ($page + SEARCHNUM)
226 . "$getstring\">"
227 . ($currpage + 1)
228 . "</a>";
229
230 if ($page < (($pagecount - 2)*SEARCHNUM)) {
231 $nextpages .= " | <a class=\"searchresults\""
232 . " href=\"?page="
233 . ($page + (2*SEARCHNUM))
234 . "$getstring\">"
235 . ($currpage + 2)
236 . "</a>";
237 }
238
239 if ($page < (($pagecount - 3)*SEARCHNUM)) {
240 if ($page < (($pagecount - 4)*SEARCHNUM))
241 $lastlink = " ... of ";
242 else
243 $lastlink = " | ";
244 $lastlink .= "<a class=\"searchresults\""
245 . href=\"?page="
246 . (($pagecount - 1)*SEARCHNUM)
247 . "$getstring\">"
248 . $pagecount
249 . "</a>";
250 }
251
252 $pagenums = " | <b>$currpage</b>";
253 echo $prevlink
254 . $firstlink
255 . $prevpages
256 . $pagenums
257 . $nextpages
258 . $lastlink
259 . ' | '
260 . $nextlink;
261 }
262 ?></div>
263 </td>
264 </tr>
265 </table>
266
267 <table border="0" cellspacing="0" width="100%"
268 cellpadding="0" style="border-style: solid;
border-color:
269 #606875; border-width: 0px 1px 0px 1px;">
270
```

```
271 <?php while($item = mysql_fetch_array($query))
{
272
273 $links = get_links(
274 $item[id],
275 $item[filename],
276 $item[fileinfotext]
277 );
278
279 $dls = get_dls($item['id']);
280
281 echo '
282 <tr>
283 <td class="bg'.(($ii % 2) ? 1 : 2).'"
align="center">
284
285 <div style="margin:10px">
286 <table border="0" style="border-collapse:
287 collapse" width="458" id="table5"
cellpadding="0">
288 <tr>
289 <td rowspan="3" width="188">
290 <table border="0" cellpadding="0"
291 cellspacing="0" width="174">
292 <tr>
293 <td colspan="4">
294 <img border="0"
295 src="http://www.example.com/common/'
296 .$item[thumbnail].'"
297 width="178" height="74"
298 class="media_img">
299 </td>
300 </tr>
301 <tr>
302 <td style="border-color: #565656;
303 border-style: solid; border-width: 0px
304 0px 1px 1px;" width="18">
305 <a target="_blank"
306 href="'.$links[0][link].'"
307 '.$links[0][addlink].'>
308 <img border="0"
309 src="http://example.com/images/'
310 .$links[0][type].'.gif"
311 width="14" height="14"
312 hspace="3" vspace="3">
313 </a>
314 </td>
315 <td style="border-color: #565656;
```

```
border-style: solid; border-width: 0px
0px 1px 0px;" align="left" width="71">
<a target="_blank"
href="'.$links[0][link].'"
class="media_download_link"
'.$links[0][addlink].'>'
.(round($links[0][filesize]
/ 104858) / 10).' MB</a>
</td>
<td style="border-color: #565656;
border-style: solid; border-width: 0px
0px 1px 0px;" width="18">
'.(($links[1][type]) ? '<a
target="_blank"
href="'.$links[1][link].'"
'.$links[1][addlink].'><img
border="0"
src="http://example.com/images/'
.$links[1][type].'.gif"
width="14" height="14" hspace="3"
vspace="3">
</td>
<td style="border-color: #565656;
r339 border-style: solid; border-width: 0px
1px 1px 0px;" align="left" width="71">
<a target="_blank"
href="'.$links[1][link].'"
class="media_download_link"
'.$links[1][addlink].'>'
.(round($links[1][filesize]
/ 104858) / 10).' MB</a>' :
' </td><td> ').'
</td>
</tr>
</table>
</td>
<td width="270" valign="bottom">
<div class="list_title">
<a
href="page.php?id='.$item[rel_id].'"
class="list_title_link">'.$item[baz].'</a>
</div>
</td>
</tr>
<tr>
<td align="left" width="270">
<div class="media_text">
'.$item[description].'
```

```
364 </div>
365 </td>
366 </tr>
367 <tr>
368 <td align="left" width="270">
369 <div class="media_downloads">'
370 .number_format($dls)
371 .' Downloads
372 </div>
373 </td>
374 </tr>
375 </table>
376 </div>
377 </td>
378 </tr>';
379 $ii++;
380 } ?>
381 </table>
382
383 <table border="0" cellspacing="0" width="480"
384 cellpadding="0" style="border-style: solid;
border-color:
385 #606875; border-width: 0px 1px 1px 1px;">
386 <tr>
387 <td>
388 <div class="nav"><?php
389 if ($pagecount)
390 echo ($page + 1)
391 . " to "
392 . min(($page + SEARCHNUM), $count)
393 . " of $count";
394 ?></div>
395 </td>
396 <td align="right">
397 <div class="nav"><?php
398 if (!$pagecount) {
399 echo "No search results found!";
400 } else {
401 echo $prevlink
402 . $firstlink
403 . $prevpages
404 . $pagenums
405 . $nextpages
406 . $lastlink
407 . ' | '
408 . $nextlink;
409 }
410 ?></div>
```

```
411 </td>
412 </tr>
413 </table>
414 </td>
415
416 <?php include("theme/footer.php"); ?>
```

Appendix B. Code before Gateways

This appendix shows a partial page script for a legacy application. It has been sanitized and anonymized from an actual application.

This script was part of a system allowed journalism students to write articles for review, and to provide feedback on articles from fellow students. The students would offer "credits" for other students to review their work, and would receive credits from others by reviewing their articles. Because credits were paid per review, the students would limit the maximum number of reviews to make sure they did not run out of credits. Finally, they were allowed to provide notes indicating what the reviewer should pay attention to.

This is a version of the page script before being converted to using Gateway classes. It contains only the domain logic and data source interactions, not the preliminary setup or any display code.

```
1 <?php
2 // ...
3 require 'includes/setup.php';
4 // ...
5
6 $article_types = array(1, 2, 3, 4, 5);
7 $failure = array();
8 $now = time();
9
10 // sanitize and escape the user input
11 $input = $_POST;
12 $input['body'] = strip_tags($input['body']);
13 $input['notes'] = strip_tags($input['notes']);
14 foreach ($input as $key => $val) {
15 $input[$key] = mysql_real_escape_string($val);
16 }
17
18 if (isset($input['ready']) && $input['ready']
== 'on') {
19 $input['ready'] = 1;
```

```
20 } else {
21 $input['ready'] = 0;
22 }
23
24 // nothing less than 0.01 credits per rating
25 $input['credits_per_rating'] = round(
26 $input['credits_per_rating'],
27 2
28 );
29
30 $credits = round(
31 $input['credits_per_rating'] *
$input['max_ratings'],
32 2
33 );
34
35 // updating an existing article?
36 if ($input['id']) {
37
38 // make sure this article belongs to the user
39 $stm = "SELECT *
40 FROM articles
41 WHERE user_id = '{$user_id}'
42 AND id = '{$input['id']}'
43 LIMIT 1";
44 $result = mysql_query($stm);
45
46 if (mysql_num_rows($result)) {
47
48 // get the existing article from the database
49 $row = mysql_fetch_assoc($result);
50
51 // don't charge unless the article is ready
52 $decrement = false;
53
54 // is the article marked as ready?
55 if ($input['ready'] == 1) {
56
57 // did they offer at least the minimum?
58 if (
59 $credits > 0
60 && $input['credits_per_rating'] >= 0.01
61 && is_numeric($credits)
62 ) {
63
64 // was the article previously ready for review?
65 // (note 'row' not 'input')
66 if ($row['ready'] == 1) {
```

```
67
68 // only subtract (or add back) the difference
to their
69 // account, since they already paid something
70 if (
71 is_numeric($row['credits_per_rating'])
72 && is_numeric($row['max_ratings'])
73 ) {
74 // user owes $credits, minus whatever they paid
already
75 $amount = $row['credits_per_rating']
76 * $row['max_ratings']
77 $credits = $credits - $amount;
78 }
79
80 $decrement = true;
81
82 } else {
83 // article not ready previously, so they hadn't
84 // had credits deducted. if this is less than
their
85 // in their account now, they may proceed.
86 $residual = $user->get('credits') - $credits;
87 $decrement = true;
88 }
89
90 } else {
91 $residual = -1;
92 $failure[] = "Credit offering invalid.";
93 $decrement = false;
94 }
95
96 } else {
97
98 // arbitrary positive value; they can proceed
99 $residual = 1;
100
101 // if it was previously ready but is no
longer, refund them
102 if (
103 is_numeric($row['credits_per_rating'])
104 && is_numeric($row['max_ratings'])
105 && ($row['ready'] == 1)
106 ) {
107 // subtract a negative value
108 $amount = $row['credits_per_rating']
109 * $row['max_ratings']
110 $credits = -($amount);
```

```
111 $decrement = true;
112 }
113 }
114
115 if ($residual >= 0) {
116
117 if (strlen($input['notes'])>0) {
118 $notes = "notes = '{$input['notes']}'";
119 } else {
120 $notes = "notes = NULL";
121 }
122
123 if (strlen($input['title'])>0) {
124 $title = "title = '{$input['title']}'";
125 } else {
126 $title = "title = NULL";
127 }
128
129 if (! in_array(
130 $input['article_type'],
131 $article_types
132 )) {
133 $input['article_type'] = 1;
134 }
135
136 $stm = "UPDATE articles
137 SET
138 body = '{$input['body']}',
139 $notes,
140 $title,
141 article_type = '{$input['article_type']}',
142 ready = '{$input['ready']}',
143 last_edited = '{$now}',
144 ip = '{$_SERVER['REMOTE_ADDR']}',
145 credits_per_rating =
'{$input['credits_per_rating']}',
146 max_ratings = '{$input['max_ratings']}'
147 WHERE user_id = '{$user_id}'
148 AND id = '{$input['id']}'";
149
150 if (mysql_query($stm)) {
151 $article_id = $input['id'];
152
153 if ($decrement) {
154 // Charge them
155 $stm = "UPDATE users
156 SET credits = credits - {$credits}
157 WHERE user_id = '{$user_id}'";
```

```
158 mysql_query($stm);
159 }
160 } else {
161 $failure[] = "Could not update article.";
162 }
163 } else {
164 $failure[] = "You do not have enough credits
for ratings.";
165 }
166 }
167
168 } else {
169
170 // creating a new article. do not decrement
until specified.
171 $decrement = false;
172
173 // if the article is ready, we need to
subtract credits.
174 if ($input['ready'] == 1) {
175
176 // if this is greater than or equal to 0, they
may proceed.
177 if (
178 $credits > 0
179 && $input['credits_per_rating']>=0.01
180 && is_numeric($credits)
181 ) {
182 // minimum offering is 0.01
183 $residual = $user->get('credits') - $credits;
184 $decrement = true;
185 } else {
186 $residual = -1;
187 $failure[] = "Credit offering invalid.";
188 }
189
190 } else {
191 // arbitrary positive value if they are not
done with their article.
192 // no deduction made yet.
193 $residual = 1;
194 }
195
196 // can user afford ratings on the new article?
197 if ($residual >= 0) {
198
199 // yes, insert the article
200 $stm = "INSERT INTO articles (
```

```
201 user_id,
202 ip,
203 last_edited,
204 article_type
205 ) VALUES (
206 '{$user_id}',
207 '{$_SERVER['REMOTE_ADDR']}',
208 '$now',
209 '$input['article_type']'
210 )";
211
212 if (mysql_query($stm)) {
213 $article_id = mysql_insert_id();
214 if ($decrement) {
215 // Charge them
216 $stm = "UPDATE users
217 SET credits = credits - {$credits}
218 WHERE user_id='{$user_id}'";
219 mysql_query($stm);
220 }
221 } else {
222 $failure[] = "Could not update credits.";
223 }
224
225 $stm = "UPDATE articles
226 SET
227 body = '{$input['body']}',
228 $notes,
229 $title,
230 article_type = '{$input['article_type']}',
231 ready = '{$input['ready']}',
232 last_edited = '$now',
233 ip = '{$_SERVER['REMOTE_ADDR']}',
234 credits_per_rating =
'{$input['credits_per_rating']}',
235 max_ratings = '{$input['max_ratings']}'
236 WHERE
237 user_id = '{$user_id}'
238 AND id = '$article_id'
239 ";
240
241 if (! mysql_query($stm)) {
242 $failure[] = "Could not update article.";
243 }
244
245 } else {
246
247 // cannot afford ratings on new article
```

```
248 $failure[] = "You do not have enough credits
for ratings.";
249 }
250 }
251 ?>
```

Appendix C. Code after Gateways

This appendix shows a version of the page script from Appendix B after being converted to use Gateway classes. Note how very little of it has changed. Even though the SQL statements have been removed, the domain business logic remains embedded in the page script.

The Gateway classes are provided below the page script, and show a conversion to PDO-style bound parameters. Also note that there have been minor modifications to the `if()` conditions in the page script: whereas previously they checked to see if a query succeeded, they now check for a return value from the Gateway.

page_script.php

```
<?php
2 // ... $user_id value created earlier
3
4 $db = new Database($db_host, $db_user,
$db_pass);
5 $articles_gateway = new ArticlesGateway($db);
6 $users_gateway = new UsersGateway($db);
7
8 $article_types = array(1, 2, 3, 4, 5);
9 $failure = array();
10 $now = time();
11
12 // sanitize and escape the user input
13 $input = $_POST;
14 $input['body'] = strip_tags($input['body']);
15 $input['notes'] = strip_tags($input['notes']);
16
17 if (isset($input['ready']) && $input['ready']
== 'on') {
18 $input['ready'] = 1;
19 } else {
20 $input['ready'] = 0;
21 }
22
```

```
23 // nothing less than 0.01 credits per rating
24 $input['credits_per_rating'] = round(
25 $input['credits_per_rating'],
26 2
27 );
28
29 $credits = round(
30 $input['credits_per_rating'] *
$input['max_ratings'],
31 2
32 );
33
34 // updating an existing article?
35 if ($input['id']) {
36
37 $row = $articles_gateway-
>selectOneByIdAndUserId($input['id'], $user_id);
38
39 if ($row) {
40
41 // don't charge unless the article is ready
42 $decrement = false;
43
44 // is the article marked as ready?
45 if ($input['ready'] == 1) {
46
47 // did they offer at least the minimum?
48 if (
49 $credits > 0
50 && $input['credits_per_rating'] >= 0.01
51 && is_numeric($credits)
52 ) {
53
54 // was the article previously ready for review?
55 // (note 'row' not 'input')
56 if ($row['ready'] == 1) {
57
58 // only subtract (or add back) the difference
to their
59 // account, since they already paid something
60 if (
61 is_numeric($row['credits_per_rating'])
62 && is_numeric($row['max_ratings'])
63 ) {
64 // user owes $credits, minus whatever they paid
already
65 $amount = $row['credits_per_rating']
66 * $row['max_ratings']
```

```
$credits = $credits - $amount;
}

$decrement = true;

} else {
// article not ready previously, so they hadn't
// had credits deducted. if this is less than
their
// in their account now, they may proceed.
$residual = $user->get('credits') - $credits;
$decrement = true;
}

} else {
$residual = -1;
$failure[] = "Credit offering invalid.";
$decrement = false;
}

} else {

// arbitrary positive value; they can proceed
$residual = 1;

// if it was previously ready but is no longer,
refund them
if (
is_numeric($row['credits_per_rating'])
&& is_numeric($row['max_ratings'])
&& ($row['ready'] == 1)
) {
// subtract a negative value
$amount = $row['credits_per_rating']
* $row['max_ratings']
$credits = -($amount);
$decrement = true;
}
}

if ($residual >= 0) {

$input['ip'] = $_SERVER['REMOTE_ADDR'];
$input['last_edited'] = $now;

if (! in_array(
$input['article_type'],
$article_types
```

```
113 )) {
114 $input['article_type'] = 1;
115 }
116
117 $result = $articles_gateway-
>updateByIdAndUserId(
118 $input['id'],
119 $user_id,
120 $input
121 );
122
123 if ($result) {
124 $article_id = $input['id'];
125
126 if ($decrement) {
127 $users_gateway->decrementCredits($user_id,
$credits);
128 }
129 } else {
130 $failure[] = "Could not update article.";
131 }
132 } else {
133 $failure[] = "You do not have enough credits
for ratings.";
134 }
135 }
136
137 } else {
138
139 // creating a new article. do not decrement
until specified.
140 $decrement = false;
141
142 // if the article is ready, we need to
subtract credits.
143 if ($input['ready'] == 1) {
144
145 // if this is greater than or equal to 0, they
may proceed.
146 if (
147 $credits > 0
148 && $input['credits_per_rating']>=0.01
149 && is_numeric($credits)
150 ) {
151 // minimum offering is 0.01
152 $residual = $user->get('credits') - $credits;
153 $decrement = true;
154 } else {
```

```
155 $residual = -1;
156 $failure[] = "Credit offering invalid.";
157 }
158
159 } else {
160 // arbitrary positive value if they are not
done with their article.
161 // no deduction made yet.
162 $residual = 1;
163 }
164
165 // can user afford ratings on the new article?
166 if ($residual >= 0) {
167
168 // yes, insert the article
169 $input['last_edited'] = $now;
170 $input['ip'] = $_SERVER['REMOTE_ADDR'];
171 $article_id = $articles_gateway-
>insert($input);
172
173 if ($article_id) {
174 if ($decrement) {
175 // Charge them
176 $users_gateway->decrementCredits($user_id,
$credits);
177 }
178 } else {
179 $failure[] = "Could not update credits.";
180 }
181
182 $result = $articles_gateway-
>updateByIdAndUserId(
183 $article_id,
184 $user_id,
185 $input
186 );
187
188 if (! $result) {
189 $failure[] = "Could not update article.";
190 }
191
192 } else {
193
194 // cannot afford ratings on new article
195 $failure[] = "You do not have enough credits
for ratings.";
196 }
```

```
197 }
198 ?>
```

classes/Domain/Articles/ArticlesGateway.php

```
1 <?php
2 namespace Domain\Articles;
3
4 class ArticlesGateway
5 {
6 protected $db;
7
8 public function __construct(Database $db)
9 {
10 $this->db = $db;
11 }
12
13 public function selectOneByIdAndUserId($id,
$user_id)
14 {
15 $stm = "SELECT *
16 FROM articles
17 WHERE user_id = :user_id
18 AND id = :id
19 LIMIT 1";
20
21 return $this->db->query($stm, array(
22 'id' => $id,
23 'user_id' => $user_id,
24 ))
25 }
26
27 public function updateByIdAndUserId($id,
$user_id, $input)
28 {
29 if (strlen($input['notes']) > 0) {
30 $notes = "notes = :notes";
31 } else {
32 $notes = "notes = NULL";
33 }
34
35 if (strlen($input['title']) > 0) {
36 $title = "title = :title";
37 } else {
38 $title = "title = NULL";
39 }
40
41 $input['id'] = $id;
```

```
42 $input['user_id'] = $user_id;
43
44 $stm = "UPDATE articles
45 SET
46 body = :body,
47 $notes,
48 $title,
49 article_type = :article_type,
50 ready = :ready,
51 last_edited = :last_edited,
52 ip = :ip,
53 credits_per_rating = :credits_per_rating,
54 max_ratings = :max_ratings
55 WHERE user_id = :user_id
56 AND id = :id";
57
58 return $this->query($stm, $input);
59 }
60
61 public function insert($input)
62 {
63 $stm = "INSERT INTO articles (
64 user_id,
65 ip,
66 last_edited,
67 article_type
68 ) VALUES (
69 :user_id,
70 :ip,
71 :last_edited,
72 :article_type
73 )";
74 $this->db->query($stm, $input);
75 return $this->db->lastInsertId();
76 }
77 }
78 ?>
```

classes/Domain/Users/UsersGateway.php

```
1 <?php
2 namespace Domain\Users;
3
4 class UsersGateway
5 {
6 protected $db;
7
8 public function __construct(Database $db)
```

```
9 {
10 $this->db = $db;
11 }
12
13 public function decrementCredits($user_id,
$credits)
14 {
15 $stm = "UPDATE users
16 SET credits = credits - :credits
17 WHERE user_id = :user_id";
18 $this->db->query($stm, array(
19 'user_id' => $user_id,
20 'credits' => $credits,
21 ));
22 }
23 }
24 ?>
```

Appendix D. Code after Transaction Scripts

This Appendix shows a version of the code from appendices B and C that extracts the domain logic from the page script into a *Transactions* class. Note how the original page script is now reduced to being a object creation and injection mechanism, and hands off most logic to the *Transactions* classes. Note also how the `$failure`, `$credits`, and `$article_types` variables are now properties on the Transactions class, and how the normalization/sanitizing logic and credit-calculation logic is part of the *Transactions* logic.

page_script.php

```
<?php
2
3 // ... $user_id value created earlier
4
5 $db = new Database($db_host, $db_user,
$db_pass);
6 $articles_gateway = new ArticlesGateway($db);
7 $users_gateway = new UsersGateway($db);
8 $article_transactions = new ArticleTransactions(
9 $articles_gateway,
10 $users_gateway
11 );
12
13 if ($_POST['id']) {
14 $article_transactions-
>updateExistingArticle($user_id, $_POST);
15 } else {
16 $article_transactions-
>submitNewArticle($user_id, $_POST);
17 }
18
19 $failure = $article_transactions->getFailure();
20 ?>
```

classes/Domain/Articles/ArticleTransactions.php

```
1 <?php
2 namespace Domain\Articles;
3
```

```
4 use Domain\Users\UsersGateway;
5
6 class ArticleTransactions
7 {
8 protected $article_types = array(1, 2, 3, 4, 5);
9
10 protected $failure = array();
11
12 protected $input = array();
13
14 public function __construct(
15 ArticlesGateway $articles_gateway,
16 UsersGateway $users_gateway
17 ) {
18 $this->articles_gateway = $articles_gateway;
19 $this->users_gateway = $users_gateway;
20 }
21
22 public function getInput()
23 {
24 return $this->input;
25 }
26
27 public function getFailure()
28 {
29 return $this->failure;
30 }
31
32 public function getCredits()
33 {
34 return round(
35 $this->input['credits_per_rating'] * $this-
>input['max_ratings'],
36 2
37 );
38 }
39
40 public function filterInput($input)
41 {
42 $input['body'] = strip_tags($input['body']);
43 $input['notes'] = strip_tags($input['notes']);
44
45 if (isset($input['ready']) && $input['ready']
== 'on') {
46 $input['ready'] = 1;
47 } else {
48 $input['ready'] = 0;
49 }
```

```
50
51 // nothing less than 0.01 credits per rating
52 $input['credits_per_rating'] = round(
53 $input['credits_per_rating'],
54 2
55 );
56
57 // return the filtered input
58 return $input;
59 }
60
61 public function updateExistingArticle($user_id,
$input)
62 {
63 $this->input = $this->filterInput($input);
64 $now = time();
65 $this->failure = array();
66 $credits = $this->getCredits();
67
68 $row = $this->articles_gateway-
>selectOneByIdAndUserId(
69 $this->input['id'],
70 $user_id
71 );
72
73 if ($row) {
74
75 // don't charge unless the article is ready
76 $decrement = false;
77
78 // is the article marked as ready?
79 if ($this->input['ready'] == 1) {
80
81 // did they offer at least the minimum?
82 if (
83 $credits > 0
84 && $this->input['credits_per_rating'] >= 0.01
85 && is_numeric($credits)
86 ) {
87
88 // was the article previously ready for review?
89 // (note 'row' not 'input')
90 if ($row['ready'] == 1) {
91
92 // only subtract (or add back) the difference
to their
93 // account, since they already paid something
94 if (
```

```
95 is_numeric($row['credits_per_rating'])
96 && is_numeric($row['max_ratings'])
97 ) {
98 // user owes $credits, minus whatever they paid
99 // already
100 $amount = $row['credits_per_rating']
101 * $row['max_ratings']
102 $credits = $credits - $amount;
103 }
104
105 $decrement = true;
106
107 } else {
108 // article not ready previously, so they
hadn't
109 // had credits deducted. if this is less than
their
110 // in their account now, they may proceed.
111 $residual = $user->get('credits') - $credits;
112 $decrement = true;
113 }
114
115 } else {
116 $residual = -1;
117 $this->failure[] = "Credit offering invalid.";
118 $decrement = false;
119 }
120
121 } else {
122
123 // arbitrary positive value; they can proceed
124 $residual = 1;
125
126 // if it was previously ready but is no
longer, refund them
127 if (
128 is_numeric($row['credits_per_rating'])
129 && is_numeric($row['max_ratings'])
130 && ($row['ready'] == 1)
131 ) {
132 // subtract a negative value
133 $amount = $row['credits_per_rating']
134 * $row['max_ratings']
135 $credits = -($amount);
136 $decrement = true;
137 }
138 }
139
```

```
140 if ($residual >= 0) {
141
142 $this->input['ip'] = $_SERVER['REMOTE_ADDR'];
143 $this->input['last_edited'] = $now;
144
145 if (! in_array(
146 $this->input['article_type'],
147 $this->article_types
148 )) {
149 $this->input['article_type'] = 1;
150 }
151
152 $result = $this->articles_gateway-
>updateByIdAndUserId(
153 $this->input['id'],
154 $user_id,
155 $this->input
156 );
157
158 if ($result) {
159 $article_id = $this->input['id'];
160
161 if ($decrement) {
162 $this->users_gateway->decrementCredits(
163 $user_id,
164 $credits
165 );
166 }
167 } else {
168 $this->failure[] = "Could not update
article.";
169 }
170 } else {
171 $this->failure[] = "You do not have enough
credits for ratings.";
172 }
173 }
174 }
175
176 public function submitNewArticle($user_id,
$input)
177 {
178 $this->input = $this->filterInput($input);
179 $now = time();
180 $this->failure = array();
181 $credits = $this->getCredits();
182
183 $decrement = false;
```

```
184
185 // if the article is ready, we need to
subtract credits.
186 if ($this->input['ready'] == 1) {
187
188 // if this is greater than or equal to 0, they
may proceed.
189 if (
190 $credits > 0
191 && $this->input['credits_per_rating']>=0.01
192 && is_numeric($credits)
193 ) {
194 // minimum offering is 0.01
195 $residual = $user->get('credits') - $credits;
196 $decrement = true;
197 } else {
198 $residual = -1;
199 $this->failure[] = "Credit offering invalid.";
200 }
201
202 } else {
203 // arbitrary positive value if they are not
done with their article.
204 // no deduction made yet.
205 $residual = 1;
206 }
207
208 // can user afford ratings on the new article?
209 if ($residual >= 0) {
210
211 // yes, insert the article
212 $this->input['last_edited'] = $now;
213 $this->input['ip'] = $_SERVER['REMOTE_ADDR'];
214 $article_id = $this->articles_gateway-
>insert($this->input);
215
216 if ($article_id) {
217 if ($decrement) {
218 // Charge them
219 $this->users_gateway-
>decrementCredits($user_id, $credits);
220 }
221 } else {
222 $this->failure[] = "Could not update
credits.";
223 }
224
225 $result = $this->articles_gateway-
```

```
>updateByIdAndUserId(
226 $article_id,
227 $user_id,
228 $this->input
229 );
230
231 if (! $result) {
232 $this->failure[] = "Could not update
article.";
233 }
234
235 } else {
236
237 // cannot afford ratings on new article
238 $this->failure[] = "You do not have enough
credits for ratings.";
239 }
240 }
241 }
242 ?>
```

Appendix E. Code before Collecting Presentation Logic

articles.php

```
1 <?php
2 require "includes/setup.php";
3
4 $current_page = 'articles';
5
6 include "header.php";
7
8 $id = isset($_GET['id']) ? $_GET['id'] : 0;
9 if ($id) {
10 $page_title = "Edit An Article";
11 } else {
12 $page_title = "Submit An Article";
13 }
14
15 ?><h1><?php echo $page_title ?></h1><?php
16
17 $user_id = $user->getId();
18
19 $db = new Database($db_host, $db_user,
$db_pass);
20 $articles_gateway = new ArticlesGateway($db);
21 $users_gateway = new UsersGateway($db);
22 $article_transactions = new
ArticleTransactions(
23 $articles_gateway,
24 $users_gateway
25 );
26
27 if ($id) {
28 $article_transactions-
>updateExistingArticle($user_id, $_POST);
29 } else {
30 $article_transactions-
>submitNewArticle($user_id, $_POST);
31 }
32
33 $failure = $article_transactions->getFailure();
34 $input = $article_transactions->getInput();
35
```

```
36 ?>
37
38 <?php
39 if ($failure) {
40 $failure_text = implode("<br />\n", $failure);
41 echo "<h2>Failure</h2>";
42 echo "<p>We could not save the article.<br />";
43 echo $failure_text. "</p>";
44 } else {
45 echo "
46 <h2>Success</h2>
47 <p>We saved the article.</p>
48 ";
49 }
50 ?>
51
52 <form method="POST" action="<?php echo
$_SERVER['PHP_SELF']?>">
53
54 <input type="hidden" name="id" value="<?php
echo $id ?>" />
55
56 <h3>Title</h3>
57 <input type="text" name="title" value="<?php
58 echo $input['title']
59 ?>" size="100">
60
61
62 <h3>Article</h3>
63 <textarea name="body" cols="80" rows="30"><?php
64 echo stripslashes($input['body'])
65 ?></textarea>
66
67 <h3>Ratings</h3>
68 <p>How many rated reviews do you want?</p>
69 <select name='max_ratings'>
70 <?php for ($i = 1; $i <= 10; $i ++) {
71 echo "<option value='$i' ";
72 if ($input['max_ratings'] == $i) {
73 echo 'selected="selected"';
74 }
75 echo ">$i</option>\n";
76 } ?>
77 </select>
78
79 <p>How many credits will you give for each
rating?</p>
80 <input type='text' name='credits_per_rating'
```

```
value='<?php
81 echo $input['credits_per_rating'];
82 ?>' size='5' />
83
84 <h3>Notes for Reviewers</h3>
85 <input type="text" name="notes" value="<?php
86 echo $input['notes']
87 ?>" size="100">
88 <label><input type="checkbox" name="ready" <?
php
89 echo $input['ready'] ? 'checked="checked"' :
'';
90 ?> /> This article is ready to be rated.
</label>
91
92 <p align="center">
93 <input type="submit" value="Save"
name="submit">
94 </p>
95
96 </form>
97
98 <?php
99 include "footer.php";
100 ?>
```

Appendix F. Code after Collecting Presentation Logic

articles.php

```
1 <?php
2 require "includes/setup.php";
3
4 $user_id = $user->getId();
5
6 $db = new Database($db_host, $db_user,
$db_pass);
7 $articles_gateway = new ArticlesGateway($db);
8 $users_gateway = new UsersGateway($db);
9 $article_transactions = new ArticleTransactions(
10 $articles_gateway,
11 $users_gateway
12 );
13
14 $id = isset($_GET['id']) ? $_GET['id'] : 0;
15 if ($id) {
16 $article_transactions-
>updateExistingArticle($user_id, $_POST);
17 } else {
18 $article_transactions-
>submitNewArticle($user_id, $_POST);
19 }
20
21 $failure = $article_transactions->getFailure();
22 $input = $article_transactions->getInput();
23 $action = $_SERVER['PHP_SELF'];
24
25 /** PRESENTATION */
26
27 $current_page = 'articles';
28
29 include "header.php";
30
31 if ($id) {
32 $page_title = "Edit An Article";
33 } else {
34 $page_title = "Submit An Article";
35 }
36 ?>
```

```
37
38 <h1><?php echo $page_title ?></h1><?php
39
40 if ($failure) {
41 $failure_text = implode("<br />\n", $failure);
42 echo "<h2>Failure</h2>";
43 echo "<p>We could not save the article.<br />";
44 echo $failure_text. "</p>";
45 } else {
46 echo "
47 <h2>Success</h2>
48 <p>We saved the article.</p>
49 ";
50 }
51 ?>
52
53 <form method="POST" action="<?php echo $action
?>">
54
55 <input type="hidden" name="id" value="<?php
echo $id ?>" />
56
57 <h3>Title</h3>
58 <input type="text" name="title" value="<?php
59 echo $input['title']
60 ?>" size="100">
61
62
63 <h3>Article</h3>
64 <textarea name="body" cols="80" rows="30"><?php
65 echo stripslashes($input['body'])
66 ?></textarea>
67
68 <h3>Ratings</h3>
69 <p>How many rated reviews do you want?</p>
70 <select name='max_ratings'>
71 <?php for ($i = 1; $i <= 10; $i ++) {
72 echo "<option value='$i' ";
73 if ($input['max_ratings'] == $i) {
74 echo 'selected="selected"';
75 }
76 echo ">$i</option>\n";
77 } ?>
78 </select>
79
80 <p>How many credits will you give for each
rating?</p>
81 <input type='text' name='credits_per_rating'
```

```
value='<?php
82 echo $input['credits_per_rating'];
83 ?>' size='5' />
84
85 <h3>Notes for Reviewers</h3>
86 <input type="text" name="notes" value="<?php
87 echo $input['notes']
88 ?>" size="100">
89 <label><input type="checkbox" name="ready" <?
php
90 echo $input['ready'] ? 'checked="checked"' :
'';
91 ?> /> This article is ready to be rated.
</label>
92
93 <p align="center">
94 <input type="submit" value="Save"
name="submit">
95 </p>
96
97 </form>
98
99 <?php
100 include "footer.php";
101 ?>
```

Appendix G. Code after Response View File

articles.php

```
1 <?php
2 require "includes/setup.php";
3
4 $user_id = $user->getId();
5
6 $db = new Database($db_host, $db_user,
$db_pass);
7 $articles_gateway = new ArticlesGateway($db);
8 $users_gateway = new UsersGateway($db);
9 $article_transactions = new ArticleTransactions(
10 $articles_gateway,
11 $users_gateway
12 );
13
14 $id = isset($_GET['id']) ? $_GET['id'] : 0;
15 if ($id) {
16 $article_transactions-
>updateExistingArticle($user_id, $_POST);
17 } else {
18 $article_transactions-
>submitNewArticle($user_id, $_POST);
19 }
20
21 $response = new
\Mlaphp\Response('/path/to/app/views');
22 $response->setView('articles.html.php');
23 $response->setVars(array(
24 'id' => $id,
25 'failure' => $article_transactions-
>getFailure(),
26 'input' => $article_transactions->getInput(),
27 'action' => $_SERVER['PHP_SELF'],
28 ));
29 $response->send();
30 ?>
```

views/articles.html.php

```
1 <?php
2 $current_page = 'articles';
3
```

```
4 include "header.php";
5
6 if ($id) {
7 $page_title = "Edit An Article";
8 } else {
9 $page_title = "Submit An Article";
10 }
11 ?>
12
13 <h1><?php echo $page_title ?></h1><?php
14
15 if ($failure) {
16 echo "<h2>Failure</h2>";
17 echo "<p>We could not save the article.<br />";
18 foreach ($failure as $failure_text) {
19 echo $this->esc($failure_text) . "<br />";
20 }
21 echo "</p>";
22 } else {
23 echo "
24 <h2>Success</h2>
25 <p>We saved the article.</p>
26 ";
27 }
28 ?>
29
30 <form method="POST" action="<?php echo $this-
>esc($action) ?>">
31
32 <input type="hidden" name="id" value="<?php
echo $this->esc($id) ?>" />
33
34 <h3>Title</h3>
35 <input type="text" name="title" value="<?php
36 echo $this->esc($input['title'])
37 ?>" size="100">
38
39
40 <h3>Article</h3>
41 <textarea name="body" cols="80" rows="30"><?php
42 echo stripslashes($this->esc($input['body']))
43 ?></textarea>
44
45 <h3>Ratings</h3>
46 <p>How many rated reviews do you want?</p>
47 <select name='max_ratings'>
48 <?php for ($i = 1; $i <= 10; $i ++) {
49 $i = $this->esc($i);
```

```
50 echo "<option value='$i' ";
51 if ($input['max_ratings'] == $i) {
52 echo 'selected="selected"';
53 }
54 echo ">$i</option>\n";
55 } ?>
56 </select>
57
58 <p>How many credits will you give for each
rating?</p>
59 <input type='text' name='credits_per_rating'
value='<?php
60 echo $this->esc($input['credits_per_rating']);
61 ?>' size='5' />
62
63 <h3>Notes for Reviewers</h3>
64 <input type="text" name="notes" value="<?php
65 echo $this->esc($input['notes'])
66 ?>" size="100">
67 <label><input type="checkbox" name="ready" <?
php
68 echo ($input['ready']) ? 'checked="checked"' :
'';
69 ?> /> This article is ready to be rated.
</label>
70
71 <p align="center">
72 <input type="submit" value="Save"
name="submit">
73 </p>
74
75 </form>
76
77 <?php
78 include "footer.php";
79 ?>
```

Appendix H. Code after Controller Rearrangement

articles.php

```
1 <?php
2 require "includes/setup.php";
3
4 /* DEPENDENCY */
5
6 $db = new Database($db_host, $db_user,
$db_pass);
7 $articles_gateway = new ArticlesGateway($db);
8 $users_gateway = new UsersGateway($db);
9 $article_transactions = new ArticleTransactions(
10 $articles_gateway,
11 $users_gateway
12 );
13 $response = new
\Mlaphp\Response('/path/to/app/views');
14
15 /* CONTROLLER */
16
17 $user_id = $user->getId();
18
19 $id = isset($_GET['id']) ? $_GET['id'] : 0;
20 if ($id) {
21 $article_transactions-
>updateExistingArticle($user_id, $_POST);
22 } else {
23 $article_transactions-
>submitNewArticle($user_id, $_POST);
24 }
25
26 $response->setView('articles.html.php');
27 $response->setVars(array(
28 'id' => $id,
29 'failure' => $article_transactions-
>getFailure(),
30 'input' => $article_transactions->getInput(),
31 'action' => $_SERVER['PHP_SELF'],
32 ));
33
34 /* FINISHED */
```

```
35
36 $response->send();
37 ?>
```

Appendix I. Code after Controller Extraction

articles.php

```
1 <?php
2 require "includes/setup.php";
3
4 /* DEPENDENCY */
5
6 $db = new Database($db_host, $db_user,
$db_pass);
7 $articles_gateway = new ArticlesGateway($db);
8 $users_gateway = new UsersGateway($db);
9 $article_transactions = new ArticleTransactions(
10 $articles_gateway,
11 $users_gateway
12 );
13 $response = new
\Mlaphp\Response('/path/to/app/views');
14 $controller = new \Controller\ArticlesPage();
15
16 /* CONTROLLER */
17
18 $response = $controller->__invoke(
19 $request,
20 $response,
21 $user,
22 $article_transactions
23 );
24
25 /* FINISHED */
26
27 $response->send();
28 ?>
```

classes/Controller/ArticlesPage.php

```
1 <?php
2 namespace Controller;
3
4 use Domain\Articles\ArticleTransactions;
5 use Mlaphp\Request;
6 use Mlaphp\Response;
7 use User;
8
```

```
9 class ArticlesPage
10 {
11 public function __construct()
12 {
13 }
14
15 public function __invoke(
16 Request $request,
17 Response $response,
18 User $user,
19 ArticleTransactions $article_transactions
20 ) {
21 $user_id = $user->getId();
22
23 $id = isset($request->get['id'])
24 ? $request->get['id']
25 : 0;
26
27 if ($id) {
28 $article_transactions->updateExistingArticle(
29 $user_id,
30 $request->post
31 );
32 } else {
33 $article_transactions->submitNewArticle(
34 $user_id,
35 $request->post
36 );
37 }
38
39 $response->setView('articles.html.php');
40 $response->setVars(array(
41 'id' => $id,
42 'failure' => $article_transactions-
>getFailure(),
43 'input' => $article_transactions->getInput(),
44 'action' => $request->server['PHP_SELF'],
45 ));
46
47 return $response;
48 }
49 }
50 ?>
```

Appendix J. Code after Controller Dependency Injection

articles.php

```
1 <?php
2 require "includes/setup.php";
3
4 /* DEPENDENCY */
5
6 $db = new Database($db_host, $db_user,
$db_pass);
7 $articles_gateway = new ArticlesGateway($db);
8 $users_gateway = new UsersGateway($db);
9 $article_transactions = new ArticleTransactions(
10 $articles_gateway,
11 $users_gateway
12 );
13 $response = new
\Mlaphp\Response('/path/to/app/views');
14 $controller = new \Controller\ArticlesPage(
15 $request,
16 $response,
17 $user,
18 $article_transactions
19 );
20
21 /* CONTROLLER */
22
23 $response = $controller->__invoke();
24
25 /* FINISHED */
26
27 $response->send();
28 ?>
```

classes/Controller/ArticlesPage.php

```
1 <?php
2 namespace Controller;
3
4 use Domain\Articles\ArticleTransactions;
```

```
5 use Mlaphp\Request;
6 use Mlaphp\Response;
7 use User;
8
9 class ArticlesPage
10 {
11 protected $user;
12
13 protected $article_transactions;
14
15 protected $request;
16
17 protected $response;
18
19 public function __construct(
20 Request $request,
21 Response $response,
22 User $user,
23 ArticleTransactions $article_transactions
24 ) {
25 $this->user = $user;
26 $this->article_transactions =
$article_transactions;
27 $this->request = $request;
28 $this->response = $response;
29 }
30
31 public function __invoke()
32 {
33 $user_id = $this->user->getId();
34
35 $id = isset($this->request->get['id'])
36 ? $this->request->get['id']
37 : 0;
38
39 if ($id) {
40 $article_transactions->updateExistingArticle(
41 $user_id,
42 $this->request->post
43 );
44 } else {
```
Appendix J: Code After Controller Dependency Injection 217
```
45 $article_transactions->submitNewArticle(
46 $user_id,
47 $this->request->post
48 );
49 }
```

```
50
51 $this->response->setView('articles.html.php');
52 $this->response->setVars(array(
53 'id' => $id,
54 'failure' => $this->article_transactions-
>getFailure(),
55 'input' => $this->article_transactions-
>getInput(),
56 'action' => $this->request->server['PHP_SELF'],
57 ));
58
59 return $this->response;
60 }
61 }
62 ?>
```

Bibliography

This Learning Path is a blend of content, all packaged up keeping your journey in mind. It includes content from the following Packt products:

- *PHP 7 Programming Cookbook, Doug Bierer*
- *Learning PHP 7 High Performance, Altaf Hussain*
- *Modernizing Legacy Applications in PHP, Paul Jones*

Index

A

- $aad parameter / There's more...
- *AMP packages
    - XAMPP, reference link / Installing a *AMP package
    - AMPPS, reference link / Installing a *AMP package
    - MAMP, reference link / Installing a *AMP package
    - WampServer, reference link / Installing a *AMP package
    - Zend Server, reference link / Installing a *AMP package
    - EasyPHP, reference link / Installing a *AMP package
- abstract method / How to do it...
- Accept-Language header
    - reference link / See also
- Access Control List (ACL) mechanism
    - implementing, with middleware / Using middleware to implement access control, How to do it..., How it works..., See also
- Active Record
    - about / Domain logic patterns
- Adapter / Creating a simple REST client
- Advanced Encryption Standard (AES)
    - about / How to do it...
    - reference link / See also
- Alternate PHP Cache (APC) / There's more...

- Altorouter
  - reference link / See also
- Amazon Mechanical Turk
  - reference link / See also
- AMP
  - about / Installing a *AMP package
- anonymous class / How it works...
- anonymous classes
  - about / Implementing anonymous classes
  - implementing / Implementing anonymous classes, How to do it..., How it works...

  / The anonymous classes
- anonymous classes;about / Introduction
- Apache
  - about / Apache
- ApacheBench (ab)
  - about / ApacheBench (ab)
- Apache JMeter
  - about / Apache JMeter
  - reference link / Apache JMeter
- Apigility
  - about / Apigility
  - reference link / Apigility

- API key
    - creating, reference link / How it works...
    - reference link / See also
- Apple Developer Tools / Installing directly from source
- Application Programming Interface (API) / Using interfaces, Using PDO to connect to a database, Creating a simple REST server
- array
    - creating, to object hydrator / Creating an array to object hydrator, How it works...
- array hydrator
    - object, building to / Building an object to array hydrator, How to do it..., How it works...
- Artisan CLI, Laravel
    - about / Artisan CLI
    - make*controller command / Artisan CLI
    - make*model command / Artisan CLI
    - make*event command / Artisan CLI
    - make*listener command / Artisan CLI
    - make*migration command / Artisan CLI
    - php artisan migrate command / Artisan CLI
    - php artisan optimize command / Artisan CLI
    - php artisan down command / Artisan CLI
    - php artisan up command / Artisan CLI
    - php artisan cache*clear command / Artisan CLI
    - php artisan db*seed command / Artisan CLI
    - php artisan view*clear command / Artisan CLI
    - reference link / Artisan CLI
- assertions

    - about / Writing a simple test
    - reference link / See also...

- AST
    - about / Understanding the abstract syntax tree, How to do it...
    - working / How it works..., See also
    - reference link / See also

- asynchronous communications / Implementing the Pub/Sub design pattern
- Asynchronous JavaScript and XML (AJAX)
    - about / Implementing jQuery DataTables PHP lookups
    - reference link / There's more...

- Aura.Router
    - reference link / See also

- Authenticated Encrypt with Associated Data (AEAD) / There's more...
- authority / How to do it...
- autoloader
    - coding / Add Autoloader Code
    - creating, as global function / As a Global Function
    - creating, as closure / As a Closure
    - creating, as static and instance method / As a Static or Instance method
    - prioritizing / Autoloader Priority
    - performance implications / What are the Performance Implications Of Autoloading?

- Autoloader include Call
    - removing / Should we remove the autoloader include call?

- autoloading process / How to do it...

B

- backwards incompatible changes
    - reference link / See also
- base collectors / PHP DebugBar
- base Gateway class
    - about / Can we extend from a base Gateway class?
- best practices, PHP Programming
    - coding styles / Coding styles
    - test-driven development (TDD) / Test-driven development (TDD)
    - design patterns / Design patterns
    - service-oriented architecture (SOA) / Service-oriented architecture (SOA)
    - object-oriented / Being object-oriented and reusable always
    - reusable feature / Being object-oriented and reusable always
    - PHP frameworks / PHP frameworks
    - version control system (VCS) / Version control system (VCS) and Git
    - Git / Version control system (VCS) and Git
    - deployment / Deployment and Continuous Integration (CI)
    - Continuous Integration (CI) / Deployment and Continuous Integration (CI)
- binary search
    - reference link / See also
- binary search class

    - building / Building a binary search class, How to do it..., How it works..., See also

- Blade templates, Laravel
    - about / Blade templates
    - reference link / Blade templates
- block cipher
    - reference link / See also
- BlowFish (BF) / How to do it...
- browser data
    - locale, obtaining / Getting the locale from browser data, How to do it..., How it works...
- brute force attacks
    - about / Building a secure password generator
    - reference link / See also
- bubble sort
    - building / Building a bubble sort, How to do it..., How it works...
- built-in PHP web server
    - using / Using the built-in PHP web server, How to do it...
- business logic
    - versus presentation logic / Embedded presentation logic

C

- cache
    - used, for improving performance / Improving performance using the cache, How to do it..., How it works...
- CakePHP
    - reference link / Framework-based Legacy Applications
- Cake PHP framework
    - reference link / See also
- callable / Understanding the abstract syntax tree
- CAMELLIA / How to do it...
- candidate include calls
    - files, picking / How should we pick files for candidate include calls?
- CAPTCHA
    - about / Introduction, Safeguarding forms with a CAPTCHA
    - used, for safeguarding forms / Safeguarding forms with a CAPTCHA
    - reference link / There's more...
- CAST5 / How to do it...
- characterization test
    - about / Test suite
- characterization tests
    - about / What about our earlier characterization tests?
- Cipher Block Chaining - Message Authentication Code (CC-MAC) / How to do it...
- class

    - inline definition / What if a Class or Function is defined inline?

- class-to-file mapping
    - examples / How Do Class Names Map To File Names?

- class autoloading
    - implementing / Implementing class autoloading, How to do it..., How it works...

- classes
    - developing / Developing classes, How to do it..., How it works..., See also...
    - extending / Extending classes, How to do it..., How it works...
    - directory location, selecting / A Single Location for Classes
    - autoloader, coding / Add Autoloader Code
    - autoloader, creating as global function / As a Global Function
    - autoloader, creating as closure / As a Closure
    - autoloader, creating as static and instance method / As a Static or Instance method
    - aliasing, with use statement / Convert instantiations to use parameters
    - testing, difficulties / What about hard-to-test classes?

- class files
    - consolidating / Consolidate Class Files
    - candidate include, searching / Find a candidate include
    - moving / Move the class file
    - related include calls, removing / Remove the related include calls
    - codebase, spot checking / Spot check the codebase
    - code, committing / Commit, Push, Notify QA

    - code, pushing / Commit, Push, Notify QA
    - QA team, notifying / Commit, Push, Notify QA
    - single class, converting from include to autoloading / Do ... While
    - functions, consolidating into / Consolidate functions into class files

- code coverage
    - about / What about code coverage?
    - reference link / What about code coverage?

- coding styles
    - about / Coding styles

- Comma Separated Values (CSV) / Uploading a spreadsheet into a database
- Community Enterprise Operating System (CentOS)
    - setting up / Setting up CentOS
    - NGINX, installing / Installing NGINX
    - PHP 7, installing / Installing PHP 7
    - Percona Server, installing / Installing Percona Server

- complex characters
    - converting / Converting complex characters, How it works...

- complex Query strings
    - about / What about complex query strings?

- complex result structures
    - about / What about multiple queries and complex result structures?

- Compose

    - reference link / How it works...

- composer
    - reference link / PHP DebugBar

- Composer
    - reference link / How it works..., How to do it...
    - about / Composer – A dependency manager for PHP
    - dependencies, download link / Composer – A dependency manager for PHP
    - installation / Composer installation
    - using / Using Composer

- composer.json file directives
    - reference link / See also...

- constant visibility
    - reference link / See also

- constructor parameters
    - global properties, converting to / Convert global properties to constructor parameters

- container service
    - creating / Create a Container Service
    - URL path, routing / Route the URL Path to the Container Service

- Content Delivery Network (CDN)
    - about / Content Delivery Network (CDN)
    - features / Content Delivery Network (CDN)
    - using / Using CDN

    - reference links / Using CDN

- context sensitive lexer
    - about / Building an OOP SQL query builder
    - reference link / See also
- Continuous Integration (CI) / Deployment and Continuous Integration (CI)
- cookies
    - about / What about Headers and Cookies?
- coupled paths
    - about / Coupled Paths
    - decoupling / Did we really Decouple the Paths?
- C program
    - compiling, reference link / See also
- Create Read Update Delete (CRUD) / Design patterns
- Cross-site scripting (XSS) / How to do it...
- cross-site scripting (XSS)
    - about / Add Proper Escaping
    - reference / Add Proper Escaping
- Cross Site Request Forgery (CSRF) / How to do it...
    - about / Introduction, Securing forms with a token
    - reference link / See also
- Cryptographically Secure Pseudo Random Number Generator (CSPRNG) / How to do it...
- CSPRNG / How to do it...
- CSS and JavaScript optimization

    - about / CSS and JavaScript optimization
    - merging process / Merging
    - minifying process / Minifying
    - Minify / Minify
    - GRUNT / Grunt

- currency

    - handling, by locale / Handling currency by locale
    - formatting, by locale / How to do it..., How it works...

- cursor / How to do it...
- custom autoloader

    - using / What If I Already Have An Autoloader?

D

- database

    - spreadsheet, uploading into / Uploading a spreadsheet into a database, How to do it..., How it works...
    - connecting, PHP Data Objects (PDO) used / Using PDO to connect to a database, How to do it..., How it works..., See also

- Database class

    - about / What if there is no Database Class?

- database tables

    - representation, by defining entities / Defining entities to match database tables, How to do it..., How it works...

- Data Encryption Standard (DES) / How to do it...
- data mapper / Defining a mapper
- Data Source Name (DSN) / How to do it...

- Data Transfer Object design pattern / Creating an array to object hydrator
- data types
    - hinting at / Hinting at data types, How to do it..., How it works...
- date/time
    - formatting, by locale / Formatting date/time by locale, How to do it..., How it works...
- date formats
    - reference link / See also
- Debian
    - reference link / Installing PHP 7 from pre-compiled binaries
    - setting up / Setting up Debian or Ubuntu
    - about / Debian
- debugging
    - with Sublime Text / Debugging with Sublime Text
    - with Eclipse / Debugging with Eclipse
- decryption
    - without using mcrypt / Encrypting/decrypting without mcrypt, How to do it..., How it works..., There's more...
- deep iteration / How to do it...
- deep web scanner
    - building / Building a deep web scanner, How to do it..., See also
- definition file

    - logic, executing / What if a definition file also executes logic?
    - similar class names / What if two classes have the same name?

- delegating generator / How to do it...
- DELETE statement

    - about / What about INSERT, UPDATE, and DELETE Statements?

- dependency injection

    - issues / Embedded instantiation
    - replacement process / The replacement process
    - extra code / Isn't this a lot of code?
    - automating / Can we automate all these Injections?

- dependency injection container

    - about / What is a Dependency Injection Container?
    - reference link / What is a Dependency Injection Container?
    - adding / Adding a DI Container
    - include file, adding / Add a DI Container Include File
    - router service, adding / Add a Router Service
    - front controller, modifying / Modify the Front Controller
    - page scripts, extracting to services / Extract Page Scripts to Services

- deployment

    - about / Deployment and Continuous Integration (CI)
    - setup / Deployment and Continuous Integration (CI)

- dereferencing process / Introduction
- design patterns

    - about / Design patterns

- development environment
    - Windows, setting up / Setting up Windows
    - Debian, setting up / Setting up Debian or Ubuntu
    - Ubuntu, setting up / Setting up Debian or Ubuntu
    - CentOS, setting up / Setting up CentOS
    - Vagrant, setting up / Setting up Vagrant
- DocBlock / How to do it...
- Docker
    - reference link / See also
- Doctrine
    - reference links / See also
- domain logic
    - print / Are printing and echoing part of Domain Logic?
    - echo / Are printing and echoing part of Domain Logic?
    - in Gateway classes / What about Domain Logic in Gateway classes?
    - in non-domain classes / What about Domain logic embedded in Non-Domain classes?
- domain logic patterns
    - about / Domain logic patterns
    - transaction script / Domain logic patterns
    - domain model / Domain logic patterns
    - table module / Domain logic patterns
    - service layer / Domain logic patterns
    - extracting / The Extraction Process
    - Gateway classes, searching / Search for uses of Gateway
    - relevant domain logic, discover / Discover and Extract Relevant Domain Logic

    - relevant domain logic, extracting / Discover and Extract Relevant Domain Logic
    - extraction, example / Example Extraction
    - code, checking / Spot check the remaining original code
    - extracted transactions, testing / Write tests for the extracted transactions
    - checking / Spot check again, Commit, Push, Notify QA
    - committing / Spot check again, Commit, Push, Notify QA
    - Gateway calls, testing / Do ... While

- domain model
    - about / Defining entities to match database tables
    - reference links / See also

- Drupal 8 / Drupal 8

E

- Eclipse
    - used, for debugging / Debugging with Eclipse

- editor/IDE
    - about / Editor/IDE

- eloquent ORM, Laravel
    - about / Eloquent ORM
    - reference link / Eloquent ORM

- embedded action logic
    - about / Embedded action logic
    - extraction process / The Extraction Process
    - searching for embedded action logic / Search for Embedded Action Logic

    - page script, rearraning / Rearrange the Page Script and Spot Check
    - spot check, rearraning / Rearrange the Page Script and Spot Check
    - code blocks, identifying / Identify Code Blocks
    - code, moving to related block / Move Code to Its Related Block
    - rearranged code, spot checking / Spot Check the Rearranged Code
    - Controller class, extracting / Extract a Controller Class
    - class name, picking / Pick a Class Name
    - Skeleton class file, creating / Create a Skeleton Class File
    - action logic, moving / Move the Action Logic and Spot Check
    - Controller, converting to dependency injection / Convert Controller to Dependency Injection and Spot Check
    - Controller test, writing / Write a Controller Test
    - code, commiting / Commit, Push, Notify QA
    - tests, pushig to repository / Commit, Push, Notify QA
    - notify QA / Commit, Push, Notify QA
    - Do ... While / Do ... While
    - parameters, passing to Controller method / Can we pass parameters to the Controller method?
    - multiple actions, performing with Controller / Can a Controller have Multiple actions?
    - include calls, in controller block / What If the Controller contains include Calls?

- embedded domain logic
    - about / Embedded Domain Logic

- embedded include calls
    - about / Embedded include Calls

- embedded instantiation

    - about / Embedded instantiation

- embedded presentation logic

    - about / Embedded presentation logic
    - extraction process / The Extraction process
    - searching for / Search for Embedded presentation logic
    - view files tests, writing / Write View File Tests
    - template system / What if we already have a Template system?
    - streaming content / What about Streaming Content?
    - presentation variables / What if we have lots of Presentation variables?
    - class methods / What about class methods that generate output?
    - business logic / What about Business Logic Mixed into the presentation?
    - reviewing / Review and next steps

- embedded SQL statements

    - about / Embedded SQL Statements
    - extraction process / The extraction process
    - SQL statements, searching / Search for SQL statements
    - Gateway class method / An initial Gateway class method

- emoji

    - using, in view script / Using emoticons or emoji in a view script, How to do it..., How it works...
    - codes, reference link / See also

- emoticons

    - using, in view script / Using emoticons or emoji in a view script, How it works...

- encryption

    - without using mcrypt / Encrypting/decrypting without mcrypt, How to do it..., How it works..., There's more...

- encryption operation modes
    - reference link / See also
- entities
    - defining, to match database tables / Defining entities to match database tables, How to do it..., How it works..., See also
- entity classes
    - about / Defining entities to match database tables
    - linking, to RDBMS queries / Tying entity classes to RDBMS queries, How to do it..., How it works...
- errors, configuration stage
    - about / Installing directly from source
    - reference link / Installing directly from source
- exceptions
    - about / What About Exceptions and SPL Classes?
- eXtensible Markup Language (XML) / Introduction
- extraction process
    - about / The Extraction process
    - searching for embedded presentation logic / Search for Embedded presentation logic
    - page script, rearranging / Rearrange the Page script and Spot Check
    - spot check, rearranging / Rearrange the Page script and Spot Check

F

- factory
  - collections, creating / Should a factory create collections?
- Factory design pattern / How to do it...
- Factory object
  - about / Embedded instantiation
- fake test data
  - generating / Generating fake test data, How to do it..., How it works..., There's more...
  - reference links / There's more...
- FastRoute
  - reference link / See also
- Fedora / Red Hat
  - reference link / Installing PHP 7 from pre-compiled binaries
- fetch mode / How to do it...
- fetchObject()
  - reference link / There's more...
- file structure
  - about / File Structure
- filters
  - reference link / See also
- filter_input_array()

    - reference link / See also

- fluent interface / How it works...
- FLUSHALL command / Redis – the key-value cache store
- FLUSHDB command / Redis – the key-value cache store
- font protection
    - reference link / See also
- foreach() handling
    - differences / Understanding differences in foreach() handling, How to do it..., How it works...
- form
    - validation, connecting to / Tying validation to a form, How to do it..., How it works...
- form factory
    - implementing / Implementing a form factory, How to do it..., How it works...
- forms
    - securing, with token / Securing forms with a token, How to do it..., How it works...
    - safeguarding, with CAPTCHA / Safeguarding forms with a CAPTCHA
- forms, safeguarding
    - CAPTCHA, using / How to do it...
    - text CAPTCHA, generating / Generating a text CAPTCHA
    - image CAPTCHA, generating / Generating an image CAPTCHA, How it works..., There's more...
- framework

    - legacy application, converting / Conversion to Framework
- front controller
    - modifying / Modify the Front Controller
- full page caching
    - about / Full page caching
- function
    - inline definition / What if a Class or Function is defined inline?
- functional programming / Introduction
- function file conversion, to class file
    - about / Convert the function file to a class file
    - function calls, changing to static method calls / Change function calls to static method calls
    - static method calls, spotchecking / Spot check the static method calls
    - class file, moving / Move the class file
- functions
    - about / Introduction
    - developing / Developing functions, How to do it..., How it works...
    - best practice / How to do it...
- functions, consolidating into class files
    - about / Consolidate functions into class files
    - candidate include, searching / Find a candidate include
    - function file, converting to class file / Convert the function file to a class file

    - single class file, converting from include to autoloading / Do ... While

G

- $GLOBALS
    - about / What about $GLOBALS?
- Gateway class
    - SQL statements, extracting / Move SQL to a Gateway class
- Gateway classes
    - domain logic / What about Domain Logic in Gateway classes?
- Gateway class method
    - about / An initial Gateway class method
    - SQL Injection, defeating / Defeating SQL Injection
    - test, writing / Write a test
    - code, replacing / Replace the original code
    - testing / Test, Commit, Push, Notify QA
    - pushing / Test, Commit, Push, Notify QA
    - committing / Test, Commit, Push, Notify QA
    - SQL statements, searching / Do ... While
- generator
    - used, for writing own iterator / Writing your own iterator using generators, How it works...
    - about / Writing your own iterator using generators
- Generator / How to do it...
- generators

    - reference link / See also

- generic form element generator
    - creating / Creating a generic form element generator, How to do it...
    - working / How it works...

- getter
    - about / Using getters and setters
    - using / How to do it..., How it works...

- gettext
    - avoiding, in translation / Handling translation without gettext, How to do it..., How it works...
    - reference link / See also

- Git
    - about / Version control system (VCS) and Git, Git – A version control system
    - reference link / Version control system (VCS) and Git, Desktop tools to manage repositories
    - installation / Git installation
    - using / Using Git
    - new branches, creating / Creating new branches and merging
    - new branches, merging / Creating new branches and merging
    - repository, cloning / Cloning a repository
    - webhooks / Webhooks
    - desktop tools, used for managing repositories / Desktop tools to manage repositories

- GitHub Desktop

    - download link / Desktop tools to manage repositories

- glob() function / Recursive directory iterator
- global dependencies

    - creating / Global Dependencies
    - replacement process / The replacement process
    - global variable, searching / Find a global variable
    - global variable, converting to properties / Convert global variables to properties
    - testing / Spot check the class
    - global properties, converting to constructor parameters / Convert global properties to constructor parameters
    - checking / Spot check, Commit, Push, Notify QA
    - QA, notifying / Spot check, Commit, Push, Notify QA
    - classes, converting to use dependency injection / Do ... While

- global properties

    - converting, to constructor parameters / Convert global properties to constructor parameters
    - instantiations, converting / Convert instantiations to use parameters

- global variable

    - searching / Find a global variable
    - converting, to properties / Convert global variables to properties

- GNU Compiler Collection (GCC) / Installing directly from source
- GNU project / Installing directly from source
- Google Translation API

    - reference link / See also

- group use declaration

    - about / Namespaces and group use declaration
    - non mixed group use declarations / Non mixed group use declarations
    - mixed group use declarations / Mixed group use declarations
    - compound namespace declaration / The compound namespace declaration
- group use feature / How to do it..., How it works...
- GRUNT
    - about / Grunt
    - reference link / Grunt
- grunt watch
    - about / Grunt watch
- GZIP compression
    - about / GZIP compression
    - in Apache / Apache
    - in NGINX / NGINX

H

- HAProxy
    - load balancing / HAProxy load balancing, HAProxy load balancing
    - installation / HAProxy installation
- headers
    - about / What about Headers and Cookies?
- heredoc / How it works...
- Homestead Vagrant box

    - reference link / Xdebug

- host / How to do it...
- HTML international calendar generator
    - creating / Creating an HTML international calendar generator, How to do it...
    - internationalized output, refining / Refining internationalized output, How it works...
- HTML radio element generator
    - creating / Creating an HTML radio element generator, How to do it..., How it works..., There's more...
- HTML select element generator
    - creating / Creating an HTML select element generator, How to do it..., How it works...
- HTTP persistent connection
    - about / HTTP persistent connection
    - benefits / HTTP persistent connection
    - in Apache / Apache
    - in NGINX / NGINX
    - GZIP compression / GZIP compression
    - PHP, using as separate service / Using PHP as a separate service
    - unused modules, disabling / Disabling unused modules
    - web server resources / Web server resources
- HTTP PUT
    - reference link / See also
- HTTP server optimization
    - about / HTTP server optimization

    - static files, caching / Caching static files

- HTTP status codes
    - reference link / How to do it...

- hydration / How to do it...
- Hydrator pattern / Creating an array to object hydrator
- HyperText Transfer Protocol (HTTP) / Creating a simple REST client

- improvements
    - fast parameter parsing, reference link / There's more...
    - PHP NG, reference link / There's more...
    - dead weight, reference link / There's more...

- include
    - multiple class, defining / What if an include defines more than one class?

- include calls
    - originating, in non-class files / What about include calls originating in non-class files?

- include calls, replacement process
    - about / The Replacement process
    - include calls, searching / Search for include Calls
    - single include call, replacing / Replacing a Single include Call
    - multiple include calls, replacing / Replacing Multiple include Calls

    - include file, copying to class method / Copy include file to Class Method
    - original include call, replacing / Replace the original include Call
    - coupled variables, discovering through testing / Discover coupled variables through testing
    - other include calls, replacing / Replace other include Calls and Test
    - test, replacing / Replace other include Calls and Test
    - include file, deleting / Delete the include file and test
    - test, deleting / Delete the include file and test
    - test, writing / Write a test and refactor
    - code, refactoring / Write a test and refactor
    - Dependency Injection, converting / Convert to Dependency Injection and test
    - committing / Commit, Push, Notify QA
    - pushing / Commit, Push, Notify QA
    - include calls, replacing / Do ... While

- include files
    - logic, receiving in class / Common QuestionsCan one class receive logic from many include files?

- inconsistent naming
    - discovering / What if we used inconsistent naming?

- infrastructure
    - about / The infrastructure
    - web servers / Web servers
    - database server / The database server
    - load balancer (LB) / Load balancer (LB)
    - HAProxy, load balancing / HAProxy load balancing

- initialization vector (IV) / How to do it...
- InnoDB storage engine

    - features / The InnoDB storage engine
    - innodb_buffer_pool_size / innodb_buffer_pool_size
    - innodb_buffer_pool_instances / innodb_buffer_pool_instances
    - innodb_log_file_size / innodb_log_file_size

- INSERT statement
    - about / What about INSERT, UPDATE, and DELETE Statements?

- installation
    - Composer / Composer installation
    - Git / Git installation

- installing
    - Percona Server / Installing the Percona Server

- instance methods
    - using, instead of static methods / For functions, can we use instance methods instead of static methods?
    - process, automating / Can we automate this process?

- instantiation calls
    - modifying / Change instantiation calls
    - checking / Spot Check, Commit, Push, Notify QA
    - new keyword, searching / Do ... While

- inter-framework system calls
    - making / Making inter-framework system calls, How to do it..., How it works...

- interfaces

    - using / Using interfaces, How to do it..., How it works..., Using Traits and Interfaces, How to do it..., How it works...
- intermediary dependencies
    - about / What about Intermediary Dependencies?
- intermingled resources
    - about / Intermingled resources
- International Components for Unicode (ICU)
    - reference links / See also
- Internationalization (I18n) / How to do it...
- International Standards Organization (ISO)
    - reference link / See also

  / See also
- Internet Engineering Task Force (IETF) / Introduction
- IntlCalendar field constants
    - reference link / See also
- IntlDateFormatter
    - TopicnsetPattern()Topicnreference link / See also
- iterator
    - about / Using iterators, How to do it..., How it works...

J

- JavaScript Object Notation (JSON) / How to do it...
- Joomla! installation

    - modifying / How to do it...
- Joomla framework
    - reference link / See also
- jQuery
    - reference link / There's more...
- jQuery DataTables
    - reference link / There's more...
- jQuery DataTables;about / Implementing jQuery DataTables PHP lookups
- jQuery DataTables PHP lookups
    - implementing / Implementing jQuery DataTables PHP lookups, How to do it..., How it works..., There's more...

K

- KEYS command / Redis – the key-value cache store

L

- Laravel
    - about / Laravel
    - installation / Installation
    - features / Features, Other features
    - routing / Routing
    - reference link / Routing
    - eloquent ORM / Eloquent ORM
    - Artisan / Artisan CLI
    - Artisan CLI / Artisan CLI

    - migrations / Migrations
    - Blade templates / Blade templates

- Laravel/Lumen framework
    - reference link / See also

- Last In First Out (LIFO) / Implementing a stack
- Late Static Binding
    - about / How to do it...
    - reference link / See also

- legacy application
    - opportunities, for improvement / Opportunities for improvement
    - converting, to framework / Conversion to Framework

- legacy applications, prerequisites
    - revision control / Revision control
    - PHP 5.0 / PHP version
    - editor/IDE / Editor/IDE
    - style guide / Style Guide
    - test suite / Test suite

- legacy frameworks
    - about / Legacy Frameworks
    - framework-based legacy applications / Framework-based Legacy Applications
    - refactoring to / Refactoring to a Framework

- LimitIterator class
    - reference link / See also

- linked list

    - implementing / Implementing a linked list, How to do it..., How it works...
    - reference link / There's more...
    - about / How to do it...

- locale

    - about / Introduction
    - obtaining, from browser data / Getting the locale from browser data, How to do it..., How it works...
    - number, formatting by / Formatting numbers by locale, How to do it..., How it works...
    - currency, handling by / Handling currency by locale, How to do it..., How it works...
    - date/time, formatting by / Formatting date/time by locale, How to do it..., How it works...

- Lumen

    - about / Lumen
    - reference link / Lumen

M

- Magento 2 / Magento 2
- magic method / How to do it...
- mapper

    - defining / Defining a mapper, How to do it..., How it works...

- massive file

    - iterating through / Iterating through a massive file, How to do it..., How it works...

- mcrypt

    - avoiding, in encryption / Encrypting/decrypting without mcrypt, How to do it..., How it works...
    - avoiding, in decryption / Encrypting/decrypting without mcrypt, How to do it..., How it works...
    - reference link / See also

- Memcached
    - key-value cache store / Memcached key-value cache store

- methods
    - about / Developing classes
    - using / Using static properties and methods, How to do it..., How it works..., See also

- middleware
    - about / Introduction
    - used, for authentication / Authenticating with middleware, How to do it..., How it works..., See also
    - used, for implementing Access Control List (ACL) mechanism / Using middleware to implement access control, How to do it..., How it works..., See also
    - used, for crossing languages / Using middleware to cross languages, How to do it...

- MinGW / Installing directly from source
    - reference link / See also

- Minify
    - reference links / Minify

- mock classes / Using mock classes
- Model View Controller (MVC) / Design patterns
- Model View Controller (MVC) design pattern

    - about / The MVC design pattern
    - model layer / Model, Views
    - controllers / Controllers

- mode of operation / How to do it...
- modes
    - Electronic Code Book (ECB) / How to do it...
    - Cipher Block Chaining (CBC) / How to do it...
    - Cipher Feedback (CFB) / How to do it...
    - Output Feedback (OFB) / How to do it...
    - Counter (CTR) / How to do it...
    - Counter with CBC-MAC (CCM) / How to do it...
    - Galois/Counter Mode (GM) / How to do it...
    - XTS / How to do it...
- msgid (message ID)
    - building / Handling translation without gettext
- msgstr (message string) / Handling translation without gettext
- multi-dimensional array
    - displaying / Displaying a multi-dimensional array and accumulating totals, How to do it..., How it works...
- multiple queries
    - about / What about multiple queries and complex result structures?
- MyISAM storage engine
    - features / The MyISAM storage engine
- MySQL database / How to do it...
    - about / The MySQL database
    - query caching / Query caching

- MysqlDatabase class
    - using / What if there is no Database Class?
- MySQL performance monitoring tools
    - about / MySQL performance monitoring tools
    - phpMyAdmin / phpMyAdmin
    - MySQL workbench / The MySQL workbench
    - Percona Toolkit / Percona Toolkit
- MySQL workbench / The MySQL workbench

N

- N+1 problem
    - about / What about multiple queries and complex result structures?
    - URL / What about multiple queries and complex result structures?
- named placeholders / How it works...
- namespace / How to do it...
- namespaces
    - using / Using namespaces, How to do it..., How it works...
- naming collision / How to do it...
- new keyword
    - about / Convert instantiations to use parameters, The replacement process
- NGINX
    - installing / Installing NGINX
    - about / NGINX and Apache, NGINX

- NGINX web server
    - reference link / See also
- NGINX Windows binaries
    - download link / Setting up Windows
- non-domain classes
    - domain logic / What about Domain logic embedded in Non-Domain classes?
- non-Gateway classes
    - queries / What about queries inside non-Gateway classes?
- nullable types / How to do it...
    - reference link / See also
- null coalesce operator (??) / How it works...
- null coalesce operator(??) / The null coalesce operator(??)
- number
    - formatting, by locale / Formatting numbers by locale, How to do it..., How it works...

O

- object
    - building, to array hydrator / Building an object to array hydrator, How it works...
- object-oriented programming (OOP) / Implementing class autoloading, Introduction
- object-relational mapping

    - implementing / Implementing object-relational mapping
    - all child information, preloading / Technique #1 - pre-loading all child information
    - secondary lookups, embedding / Technique #2 - embedding secondary lookups, How it works..., See also

- object hydrator
    - array, creating / Creating an array to object hydrator, How it works...

- Object Oriented Programming (OOP) / Introduction
- Object Oriented Programming(OOP) / Introduction
- Observer / There's more...
- oclHashcat
    - reference link / See also

- one-class-per-file rule / What if the one-class-per-file rule is disagreeable?
- OOP features
    - about / OOP features
    - type hints / Type hints
    - namespaces / Namespaces and group use declaration
    - group use declaration / Namespaces and group use declaration
    - anonymous classes / The anonymous classes
    - old-style constructor deprecation / Old-style constructor deprecation
    - throwable interface / The throwable interface

- OOP SQL query builder
    - building / Building an OOP SQL query builder, How it works..., See also

- OpenSUSE

    - reference link / Installing PHP 7 from pre-compiled binaries
- operators, PHP
    - about / New operators
    - Spaceship operator (<=>) / The Spaceship operator (<=>)
    - null coalesce operator(??) / The null coalesce operator(??)
- OS platforms
    - security considerations, reference link / See also
- own iterator
    - writing, with generator / Writing your own iterator using generators, How to do it..., How it works...

P

- packt-git
    - reference link / Using Git
- page scripts
    - about / Page Scripts
    - container service, creating / Create a Container Service
    - URL path, routing to container service / Route the URL Path to the Container Service
    - spot checking / Spot Check and Commit
    - comitting / Spot Check and Commit
    - conversion / Do ... While
    - committing / Remove pages/, Commit, Push, Notify QA
    - removing / Remove pages/, Commit, Push, Notify QA
    - pushing / Remove pages/, Commit, Push, Notify QA
    - QA, notifying / Remove pages/, Commit, Push, Notify QA
    - include files, removing / What if there are includes In the Page Script?

- pagination
    - about / Handling pagination
    - handling / How to do it..., How it works...
- parameters / How to do it...
- parsing
    - differences / Understanding differences in parsing, How to do it..., How it works..., See also
- password cracking approach
    - reference link / See also
- path / How to do it...
- path converter library
    - download link / Minify
- Patterns of Enterprise Application Architecture (PoEAA)
    - about / Domain logic patterns
- Percona Server
    - installing / Installing Percona Server, Installing the Percona Server
    - about / The Percona Server - a fork of MySQL
    - reference link / Installing the Percona Server
- Percona Toolkit
    - about / Percona Toolkit
    - reference link / Percona Toolkit, pt-duplicate-key-checker
    - pt-query-digest / pt-query-digest
    - pt-duplicate-key-checker / pt-duplicate-key-checker
- Percona XtraDB Cluster (PXC)

    - about / Percona XtraDB Cluster (PXC)
- performance
    - improving, with PHP 7 enhancements / Improving performance using PHP 7 enhancements, How to do it..., How it works..., There's more...
    - improving, with cache / Improving performance using the cache, How to do it..., How it works...
- Personal Package Archive (PPA)
    - security considerations, reference link / There's more...
- Personal Package Archive (PPA);about / There's more...
- phar (PHP archive) file / How to do it...
- PHP
    - converting, to XML / Converting between PHP and XML, How to do it..., How it works...
    - reference link, for testing / Setting up a test suite
- php.ini session directives
    - reference link / How to do it...
- PHP 5 to PHP 7 code converter
    - creating / Creating a PHP 5 to PHP 7 code converter, How to do it..., How it works..., See also
- PHP 5.0
    - about / PHP version
- PHP 5.3
    - updating / What if we cannot update to PHP 5.3?

- PHP 7 / Introduction
    - acquiring, ways / PHP 7 installation considerations
    - installation considerations / How to do it...
    - installing, directly from source / Installing directly from source
    - installing, from pre-compiled binaries / Installing PHP 7 from pre-compiled binaries
    - *AMP package, installing / Installing a *AMP package
    - installing / Installing PHP 7
- php7 / Installing PHP 7 from pre-compiled binaries
- PHP 7 enhancements
    - used, for improving performance / Improving performance using PHP 7 enhancements, How to do it..., How it works..., There's more...
- PHP 7, features
    - OOP features / OOP features
    - operators / New operators
    - uniform variable syntax / Uniform variable syntax
    - about / Miscellaneous features and changes
    - constant arrays / Constant arrays
    - multiple default cases, in switch statement / Multiple default cases in the switch statement
    - session_start function, options array / The options array for session_start function
    - filtered unserialize function / Filtered unserialize function
- php7cookbook.sql file
    - reference link / How to do it...
- PHP application
    - about / The typical PHP application

    - file structure / File Structure
    - page scripts / Page Scripts
    - rewriting / Rewrite or Refactor?
    - refactoring / Rewrite or Refactor?
    - rewrites, pros and cons / The Pros and Cons of Rewriting
    - rewrites, limitations / Why Don't Rewrites Work?

- PHP Archive / Installing PHPUnit
- PHP Data Objects (PDO) / Getting ready..., How to do it...

    - about / Introduction
    - used, for connecting database / Using PDO to connect to a database
    - PDO class / Using PDO to connect to a database
    - PDOStatement class / Using PDO to connect to a database
    - PDOException class / Using PDO to connect to a database
    - PDODriver class / Using PDO to connect to a database
    - reference link / See also
    - predefined constants, reference link / See also

- PHP DebugBar

    - about / PHP DebugBar
    - reference link / PHP DebugBar

- phpDocumentor

    - reference link / See also

- PHP Framework Interop Group (PHP-FIG) / Making inter-framework system calls

    - about / Coding styles
    - reference link / Coding styles

- PHP frameworks / PHP frameworks
- PHP Locale class

    - reference link / See also

- phpMyAdmin / phpMyAdmin
- PHP NG (Next Generation) / There's more...
- PHPRedis
    - reference link / Redis – the key-value cache store
- PHP reference page
    - reference link / See also
- PHP RFC
    - reference link / There's more...
- PHP session
    - safeguarding / Safeguarding the PHP session, How to do it..., How it works..., See also
- PHP Standard Recommendation number 7 (PSR-7) / Introduction
- PHP Standards Recommendations / Using middleware to cross languages
- PHP Standards Recommendations (PSR-7) / There's more...
- PHP streams
    - reference link / See also
- PHPUnit
    - installing / Installing PHPUnit, Install PHPUnit
    - reference link / Installing PHPUnit, How to do it..., Test-driven development (TDD)
    - about / Writing a simple test
- PHPUnit documentation

    - reference link / See also...
- PHPUnit test suites
    - reference link / See also...
- PHP Windows binaries
    - download link / Setting up Windows
- port / How to do it...
- positional placeholders / How it works...
- POST
    - reference link / See also
- PostgreSQL / How to do it...
- pre-compiled binaries / Installing PHP 7 from pre-compiled binaries
- presentation, extracting to view file and spot check
    - about / Extract Presentation to View file and Spot Check
    - views/ Directory, creating / Create a views/ Directory
    - view file name, picking / Pick a View File name
    - presentation block, moving to view file / Move Presentation Block to View file
    - proper escaping, adding / Add Proper Escaping
- presentation logic
    - versus business logic / Embedded presentation logic
- private methods
    - testing / Should we test private and protected methods?
- procedural / Introduction
- profiling

    - with Xdebug / Profiling with Xdebug

- properties / Developing classes
- protected methods
    - testing / Should we test private and protected methods?
- PSR
    - reference link / See also, See also
- PSR-0
    - about / PSR-0
- PSR-6
    - reference link / See also
- PSR-7
    - about / Introduction
    - value object classes, implementing / Implementing PSR-7 value object classes, How to do it..., How it works...
    - reference link / How to do it..., See also
    - Request class, developing / Developing a PSR-7 Request class, How to do it..., How it works..., See also
    - Response class, defining / Defining a PSR-7 Response class, How to do it..., How it works..., See also
- PSR-7 interfaces
    - reference link / How it works...
- PSR-7 middleware classes
    - Guzzle, reference link / See also
    - Relay, reference link / See also
    - Radar, reference link / See also

    - NegotiationMiddleware, reference link / See also
    - psr7-csrf-middleware, reference link / See also
    - oauth2-server, reference link / See also
    - zend-diactoros, reference link / See also

- Pub/Sub design pattern

    - about / Implementing the Pub/Sub design pattern
    - implementing / How to do it..., How it works..., There's more...
    - reference link / See also

- public resources, separation process

    - about / The separation process
    - operations personnel, coordinating with / Coordinate with operations personnel
    - document root directory, creating / Create a document root directory
    - server, reconfiguring / Reconfigure the server
    - public resources, extracting / Move public resources
    - committing / Commit, push, coordinate
    - coordinating / Commit, push, coordinate
    - pushing / Commit, push, coordinate
    - importance / Is This Really Necessary?

- PuPHPet

    - reference link / Setting up Windows

- PuTTY

    - download link / Setting up Vagrant

Q

- query caching

    - enabling / Query caching

- query results

    - secondary lookups, embedding into / Embedding secondary lookups into query results, How to do it..., How it works...

R

- rainbow tables / Building a secure password generator
- Rasmus PHP7dev VagrantBox

    - reference link / Setting up Vagrant

- RDBMS queries

    - entity classes, linking to / Tying entity classes to RDBMS queries, How to do it..., How it works...

- real-world applications

    - load testing / Load testing real-world applications

- real-world applications, load testing

    - about / Load testing real-world applications
    - Magento 2 / Magento 2
    - WordPress 4 / WordPress 4
    - Drupal 8 / Drupal 8

- recurring events generator

    - building / Building a recurring events generator, How to do it..., How it works...

- recursion / How to do it...
- recursive directory iterator / Recursive directory iterator, How to do it..., How it works...

- recursive iterators / How to do it...
- Redis
    - reference link / Redis – the key-value cache store
    - about / Redis – the key-value cache store
    - installation link / Redis – the key-value cache store
    - connecting, with server / Connecting with the Redis server
    - data, storing / Storing and fetching data from the Redis server
    - data, fetching / Storing and fetching data from the Redis server
    - management tools / Redis management tools
    - features / Redis management tools
- Redis Desktop Manager (RDM) / Redis management tools
- Relational Database Management System (RDMS) / The MySQL database
- Relational Database Management Systems (RDBMS)
    - about / Using PDO to connect to a database
    - reference link / There's more...
- relational model
    - reference link / There's more...
- relational model;about / Tying entity classes to RDBMS queries
- repeated domain logic
    - about / What about repeated Domain Logic?
- repeated logic
    - in page scripts / Repeated logic
    - removing / The Removal Process
    - front controller, modifying / Modify the Front controller
    - logic, removing from page scripts / Remove Logic from Page Scripts

    - spot checking / Spot Check, Commit, Push, Notify QA
    - committing / Spot Check, Commit, Push, Notify QA
    - pushing / Spot Check, Commit, Push, Notify QA
    - QA, notifying / Spot Check, Commit, Push, Notify QA

- repetitive SQL strings
    - about / What about Repetitive SQL strings?

- replacement process, dependency injection
    - new keyword, searching / Find a new keyword
    - one-time creation, extracting / Extract One-Time creation to dependency injection
    - repeated creation, extracting to factory / Extract repeated creation to factory

- repository / How to do it...
- Representational State Transfer (REST) / Introduction
- Request class, PSR-7
    - developing / Developing a PSR-7 Request class, How to do it..., How it works..., See also
    - about / Developing a PSR-7 Request class

- resolution operator / How it works...
- Response class, PSR-7
    - reference link / See also
    - defining / Defining a PSR-7 Response class, How to do it..., How it works..., See also

- Response object
    - need for / Embedded presentation logic

- REST client
    - creating / Creating a simple REST client, How to do it...

    - streams-based REST client, creating / Creating a streams-based REST client
    - cURL-based REST client, defining / Defining a cURL-based REST client
    - working / How it works...
- RESTClient
    - reference link / How it works...
- REST server
    - creating / Creating a simple REST server, How to do it...
    - working / How it works...
- REST server implementation
    - reference link / There's more...
- return type declarations
    - reference links / See also
- return value data typing
    - using / Using return value data typing, How to do it..., How it works..., There's more...
    - reference link / There's more...
- revision control
    - about / Revision control
- rewrites, limitations
    - context-switching problem / The Context-switching problem
    - knowledge problem / The Knowledge problem
    - schedule problem / The Schedule Problem
    - iterative refactoring / Iterative Refactoring

- RFC
  - reference link / See also
- rfc3339
  - reference link / See also
- Rivest Cipher (RC) / How to do it...
- Ron's Code / How to do it...
- root locale code / How to do it...
- router service
  - adding / Add a Router Service
  - size, reducing / Can we reduce the size of the router service?
- routing
  - about / Implementing routing
  - implementing / Implementing routing, How to do it..., How it works...
- Ruby on Rails
  - about / Domain logic patterns

S

- scalar / How to do it...
- scalar type hinting
  - reference link / See also
- scheme / How to do it...
- scrollable cursor / How to do it...
- search engine

    - implementing / Implementing a search engine, How to do it..., How it works...

- Search Engine Optimization (SEO) / Implementing routing
- secondary lookups
    - embedding, into query results / Embedding secondary lookups into query results, How to do it..., How it works...
- secure password generator
    - building / Building a secure password generator, How to do it..., How it works...
- SEED / How to do it...
- SELECT command / Redis – the key-value cache store
- service-oriented architecture (SOA) / Service-oriented architecture (SOA)
- services
    - page scripts, extracting / Extract Page Scripts to Services
    - refining / How can we refine our service definitions?
- services.php file
    - size, reducing / Can we reduce the size of the services.php file?
- session-related php.ini directives
    - reference link / See also...
- session hijacking
    - reference link / See also
- sessions

    - customizing, with session_start params / Customizing sessions using session_start parameters, How to do it..., How it works...

- session_start function / The options array for session_start function
- session_start params
    - used, for customizing sessions / Customizing sessions using session_start parameters, How to do it..., How it works...
- setlocale()
    - reference link / See also
- setter
    - about / Using getters and setters
    - using / How to do it..., How it works...
- setup work
    - inconsistency / What if the Setup Work Is Inconsistent?
- set_exception_handler() function
    - reference link / See also
- shallow iteration / How to do it...
- Siege
    - about / Siege
    - reference link / Siege
- simple test
    - writing / Writing a simple test, How to do it...
    - running / Running simple tests, Running simple tests

    - database model classes, testing / Testing database Model classes, Testing database model classes
    - mock classes, using / Using mock classes, Using mock classes
    - anonymous classes, using as mock objects / Using anonymous classes as mock objects
    - Mock Builder, using / Using Mock Builder
    - assertions test operations / There's more...

- singleton / How to do it...
- Singleton / How to do it...
- singleton pattern / Design patterns
- Slim framework
    - reference link / See also
- SOAP client
    - creating / Creating a simple SOAP client, How to do it...
    - working / How it works..., See also
- SOAP clients and servers / Introduction
- SOAP server
    - creating / Creating a simple SOAP server, How to do it..., How it works..., See also
- SOAP service
    - reference link / How it works...
- software design patterns / Introduction
- spaceship operator / How to do it...
- Spaceship operator (<=>) / The Spaceship operator (<=>)
    - reference link / The Spaceship operator (<=>)
- SPL classes

    - about / What About Exceptions and SPL Classes?

- spreadsheet
    - uploading, into database / Uploading a spreadsheet into a database, How to do it..., How it works...

- SQL Injection / How to do it...
    - about / An initial Gateway class method

- SQlite / How to do it...
- SQL statements
    - searching / Search for SQL statements
    - extracting, to Gateway class / Move SQL to a Gateway class
    - namespace, determining / Namespace and Class names
    - class names, determining / Namespace and Class names
    - method names, selecting / Method names

- SQL transactions
    - about / Are we talking about SQL transactions?

- stack
    - about / Implementing a stack
    - implementing / Implementing a stack, How it works...

- Standard PHP Library (SPL) / How to do it..., How to do it..., Recursive directory iterator, How to do it..., How to do it...
- State Snapshot Transfer (SST) / Percona XtraDB Cluster (PXC)
- static / Using static properties and methods
- static class method
    - global variable, using / What if we find a global in a static method?

    - instantiations, converting / Is there an alternative conversion process?

- static files
    - caching / Caching static files
    - Apache configuration / Apache
    - NGINX / NGINX

- static properties
    - using / Using static properties and methods

  / How to do it..., How it works..., See also
- storage engines
    - about / Storage engines
    - reference link / Storage engines
    - MyISAM storage engine / The MyISAM storage engine
    - InnoDB storage engine / The InnoDB storage engine

- strategy pattern
    - implementing / Implementing a strategy pattern, How to do it..., How it works...

- stream / How to do it...
- Streams
    - about / Creating a streams-based REST client
    - reference link / See also

- Structured Query Language (SQL) / Introduction
- style guide
    - for codebase / Style Guide

- sub-generator / How to do it...
- Sublime Text

    - used, for debugging / Debugging with Sublime Text
- superglobals
    - about / What about superglobals?
- Symfony framework
    - reference link / See also
- Synaptic / Installing PHP 7 from pre-compiled binaries

T

- $tag parameter / There's more...
- $tag_length parameter / There's more...
- $this keyword
    - reference link / See also...
- test-driven development (TDD)
    - about / Test-driven development (TDD)
- test MySQL database
    - defining / Defining a test MySQL database, How to do it...
    - reference link / Defining a test MySQL database
- tests
    - writing / Fighting test resistance, The way of Testivus
    - writing, for application classes / Pick a class to test
    - passing / Do ... While
    - pushing / Do ... While
    - committing / Do ... While
    - importance / Come On, Really, Can We Do This Later?
    - modifying / Can we change a test after we write it?

- test suite
    - about / Writing a test suite
    - writing / How to do it..., How it works...
    - selecting / Test suite
    - setting up / Setting up a test suite
    - PHPUnit, installing / Install PHPUnit
    - tests/ directory, creating / Create a tests/ directory
    - test case, writing / Write a test case
- TextMate / The Pros and Cons of Rewriting
- third-party libraries
    - about / What about third-party libraries?
    - system-wide libraries / What about system-wide libraries?
    - testing / Do we need to test Third-party libraries?
- third-party libraries;about / What about third-party libraries?
- thread safe
    - versus non thread safe / Installing PHP 7 from pre-compiled binaries
- throwable interface
    - about / The throwable interface
    - error / Error
- token
    - used, for securing forms / Securing forms with a token, How to do it..., How it works...
- totals
    - accumulating / Displaying a multi-dimensional array and accumulating totals, How to do it..., How it works...
- traits

    - using / Using traits, How to do it..., How it works..., Using Traits and Interfaces, How to do it..., How it works...

- transactions
    - approaches / Can a transaction be a class instead of a Method?
- translation
    - handling, without gettext / Handling translation without gettext, How to do it..., How it works...
- traverse / Using iterators
- TreeRoute
    - reference link / See also
- Turbo C compiler / Installing directly from source
- type hint / Hinting at data types, How to do it...
- type hints
    - about / Type hints
    - scala type hints / Scalar type hints
    - return type hints / Return type hints

U

- Ubuntu
    - reference link / Installing PHP 7 from pre-compiled binaries
    - setting up / Setting up Debian or Ubuntu, Ubuntu
    - about / Ubuntu
- Uncaught Type Error / Scalar type hints
- Unicode escape syntax / Introduction
- Uniform Resource Indicator (URI)

    - reference link / How to do it...
- Uniform Variable Syntax
    - reference link / See also
- uniform variable syntax
    - null coalesce operator(??) / The null coalesce operator(??), Uniform variable syntax
    - about / Uniform variable syntax
- unit testing
    - about / Writing a simple test
    - reference link / See also...
- universal error handler
    - using / Universal error handler, How to do it..., How it works...
- universal exception handler
    - using / Universal exception handler, How to do it..., How it works..., See also
- unserialize() function / Filtered unserialize function
- unused modules
    - disabling / Disabling unused modules
    - disabling, in Apache / Apache
    - disabling, in NGINX / NGINX
- UPDATE statement
    - about / What about INSERT, UPDATE, and DELETE Statements?

- URL paths, decoupling process
    - about / The Decoupling Process
    - operations, coordinating with / Coordinate with Operations
    - front controller, creating / Add a Front Controller
    - pages/ directory, creating / Create a pages/ Directory
    - server, reconfiguring / Reconfigure the Server
    - spot checking / Spot check
    - page scripts, moving / Move Page scripts
    - committing / Commit, Push, Coordinate
    - pushing / Commit, Push, Coordinate
    - coordinating / Commit, Push, Coordinate
- URL rewriting / How to do it...
- user info / How to do it...
- UTF-8 / How to do it...

V

- Vagrant
    - reference link / See also
    - about / Setting up Windows
    - setting up / Setting up Vagrant
    - download link / Setting up Vagrant
- validation
    - and filtering, differentiating between / How to do it...
    - connecting, to form / Tying validation to a form, How to do it..., How it works...
- value object classes, PSR-7
    - implementing / Implementing PSR-7 value object classes, How to do it..., How it works..., See also
    - reference link / How it works...

- variables
    - with class names / What about class names in variables?
    - retaining / Add a DI Container Include File
- Varnish
    - about / Varnish
    - reference link / Varnish
- version control system (VCS) / Version control system (VCS) and Git
- view file tests
    - writing / Write View File Tests, Writing a View File Test
    - tests/views/ directory / The tests/views/ directory
    - correctness of content, asserting / Asserting Correctness Of Content
    - code, pushing to repository / Commit, Push, Notify QA
    - code, commiting / Commit, Push, Notify QA
    - notify QA / Commit, Push, Notify QA
    - Do ... While / Do ... While
- view script
    - emoticons, using / Using emoticons or emoji in a view script, How to do it..., How it works...
    - emoji, using / Using emoticons or emoji in a view script, How it works..., See also
- visibility
    - defining / Defining visibility, How to do it..., How it works...
- Visual Studio / Installing directly from source



- web server resources
  - about / Web server resources
  - in NGINX / NGINX
- Web Services Definition Language (WSDL) / How it works..., How to do it...
- website
  - hoovering / Hoovering a website, How to do it..., How it works...
- website vulnerabilities
  - reference link / See also
- Windows
  - reference link / Installing PHP 7 from pre-compiled binaries
  - setting up / Setting up Windows
- WordPress 4 / WordPress 4
- World Wide Web (WWW) / How to do it...
- WSDL generators for PHP
  - reference link / See also

X

- ?XDEBUG_PROFILE=on
  - reference link / Profiling with Xdebug
- Xcode IDE / Installing directly from source
- Xdebug
  - about / Xdebug
  - used, for profiling / Profiling with Xdebug

- XLS / See also
- XML
    - PHP, converting to / Converting between PHP and XML, How to do it..., How it works...

  / See also

Y

- YaST / Installing PHP 7 from pre-compiled binaries

Z

- Zend Expressive / How to do it...
- Zend Framework 2 / Coding styles
- Zend Framework 3/Expressive framework
    - reference link / See also
- Zend\Escaper
    - reference / Add Proper Escaping
- zero configuration data
    - reference link / There's more...