Prefacio de Eduardo Mangarelli

XAMARIN.FORMS en acción

Aplicaciones de negocio

Rodrigo Díaz Concha

Alfaomega

libros R

Xamarin.Forms en acción

Aplicaciones de negocio

Xamarin.Forms en acción

Aplicaciones de negocio

Rodrigo Díaz Concha

Alfaomega

Libros R

Diseño de colección y pre-impresión: Grupo RC
Diseño de cubierta: Cuadratín

Datos catalográficos

Díaz, Rodrigo
Xamarin.Forms en acción. Aplicaciones de negocio
Primera Edición
Alfaomega Grupo Editor, S.A. de C.V., México

ISBN: 978-607-622-962-0

Formato: 17 x 23 cm Páginas: 376

Xamarin.Forms en acción. Aplicaciones de negocio
Rodrigo Díaz Concha
ISBN: 978-84-944650-9-3 edición original publicada por RC Libros, Madrid, España.

Primera edición: Alfaomega Grupo Editor, México, julio 2017

Dr. Isidoro Olvera (Eje 2 sur) No. 74, Col. Doctores, 06720, Ciudad de México.

Miembro de la Cámara Nacional de la Industria Editorial Mexicana
Registro No. 2317

Pág. Web: **http://www.alfaomega.com.co**
E-mail: **cliente@alfaomegacolombiana.com**

ISBN: 978-958-778-322-3

Nota importante:
La información contenida en esta obra tiene un fin exclusivamente didáctico y, por lo tanto, no está previsto su aprovechamiento a nivel profesional o industrial. Las indicaciones técnicas y programas incluidos, han sido elaborados con gran cuidado por el autor y reproducidos bajo estrictas normas de control. ALFAOMEGA GRUPO EDITOR, S.A. de C.V. no será jurídicamente responsable por: errores u omisiones; daños y perjuicios que se pudieran atribuir al uso de la información comprendida en este libro, ni por la utilización indebida que pudiera dársele.

Edición autorizada para venta en México y todo el continente americano.

Impreso en Colombia. Printed in Colombia.

Empresas del grupo:

México: Alfaomega Grupo Editor, S.A. de C.V. – Dr. Isidoro Olvera (Eje 2 sur) No. 74, Col. Doctores, C.P. 06720, Del. Cuauhtémoc, Ciudad de México. Tel.: (52-55) 5575-5022 – Fax: (52-55) 5575-2420 / 2490. Sin costo: 01-800-020-4396 E-mail: atencionalcliente@alfaomega.com.mx

Colombia: Alfaomega Colombiana S.A. – Calle 62 No. 20-46, Barrio San Luis, Bogotá, Colombia, Tels.: (57-1) 746 0102 / 210 0122 – E-mail: cliente@alfaomegacolombiana.com

Chile: Alfaomega Grupo Editor, S.A. – Av. Providencia 1443. Oficina 24, Santiago, Chile
Tel.: (56-2) 2235-4248 – Fax: (56-2) 2235-5786 – E-mail: agechile@alfaomega.cl

Argentina: Alfaomega Grupo Editor Argentino, S.A. – Av. Córdoba 1215 piso 10, C.P. 1055, Buenos Aires, Argentina. – Tel./Fax: (54-11) 4811-0887 y 4811 7183 – E-mail: ventas@alfaomegaeditor.com.ar

Para Diego.

Que todos tus caminos siempre estén iluminados de sabiduría. Te amo hijo.

ÍNDICE

PREFACIO

Vivimos en la Era de la Transformación Digital, donde el World Economic Forum habla de que estamos transcurriendo la Cuarta Revolución Industrial[1]. Estas son denominaciones que representan el impacto de la tecnología en las personas, empresas, gobiernos y modelos de negocios, no solo por los cambios, desafíos y oportunidades que generan en cada uno de estos ámbitos, sino por la velocidad sin precedentes con la que estos cambios suceden.

Fenómenos como el poder del Cloud Computing, la disponibilidad y accesibilidad del acceso a Internet, la movilidad y proliferación de múltiples tipos de dispositivos móviles, sumados a los más recientes como la "Inteligencia Artificial", la realidad virtual y realidad aumentada, son factores de innovación, de creación nuevas oportunidades, disrupción de modelos de negocios y creación de nuevas experiencias para los usuarios.

Si hablamos de experiencias para los usuarios, los Smartphones son hoy el primer punto de comunicación e interacción rica de empresas y servicios con usuarios a través de Apps.

Hoy, los desarrolladores de Apps enfrentan el desafío de la heterogeneidad de dispositivos y tecnologías: iOS, Android y Windows componen el mercado de Smartphones. Desarrollar para múltiples plataformas es complejo y costoso. Conocer múltiples sistemas operativos y lenguajes de programación, adquirir los conocimientos necesarios y tenerlos actualizados, mantener múltiples versiones de

[1] https://www.weforum.org/agenda/2016/01/the-fourth-industrial-revolution-what-it-means-and-how-to-respond/

código y diferentes ambientes y administrar varias distribuciones y actualizaciones son parte de los retos a los que los desarrolladores deben enfrentarse.

Xamarin.Forms tiene un lugar de privilegio en este contexto, permitiendo desarrollar, mantener y administrar una línea base de código para iOS, Android y Windows, para luego generar aplicaciones nativas para cada plataforma, logrando que la App tenga la performance y experiencia en cada dispositivo, dando adicionalmente al desarrollador el poder de la reutilización, la productividad y la optimización en la calidad del código, y en consecuencia de la calidad de la aplicación.

Este libro aporta un enorme valor a la comunidad desarrolladora de habla hispana, permitiendo a desarrolladores tener una didáctica, sólida y completa referencia para poder tomar el máximo potencial de Xamarin.Forms y en consecuencia tener éxito en el desarrollo de aplicaciones móviles para múltiples plataformas.

Eduardo Mangarelli
Director de Tecnología e Innovación de Microsoft Latinoamérica
Ingeniero de Sistemas de la Universidad ORT de Uruguay
Profesor de la Universidad ORT de Uruguay
Inversor y Advisor de múltiples empresas

PRÓLOGO

A lo largo de varios años, he tenido la oportunidad de participar en una gran cantidad de proyectos de software dentro y fuera de México, para empresas privadas y gubernamentales. En los últimos dos años, muchos de esos proyectos han sido exitosas aplicaciones móviles multiplataforma de negocio construidas con Xamarin.Forms, por lo que de primera mano estoy convencido y he podido apreciar que Xamarin.Forms es una excelente opción tecnológica en el momento de decidir el rumbo por el que irán las soluciones móviles en tu empresa o institución.

El libro que tienes en tus manos es el resultado de varios meses de arduo trabajo y desvelo. El haber decidido escribir esta obra tiene la firme intención de inspirar a los lectores a construir con calidad todo tipo de aplicación móvil de negocio con Xamarin.Forms.

¿QUIÉN DEBERÍA LEER ESTE LIBRO?

Para que puedas sacar el máximo provecho, este libro presupone que ya tienes experiencia con el lenguaje de programación C# y la plataforma de desarrollo .NET con Visual Studio .NET. Es ideal –mas no un prerrequisito– contar con experiencia previa en el desarrollo de aplicaciones con XAML (WPF, Silverlight, Windows Phone o Universal Windows Platform). El libro está enfocado a desarrolladores amateurs que deseen aprender a construir aplicaciones multiplataforma con Xamarin.Forms, y a desarrolladores profesionales de empresas que deseen iniciar o perfeccionar su estrategia de aplicaciones móviles multiplataforma con esta fascinante tecnología.

EL PROYECTO

A lo largo de los capítulos 4 al 11, construimos una aplicación móvil multiplataforma para la captura de encuestas. Si bien la aplicación hace encuestas

acerca de cuál es el equipo de fútbol favorito de la gente, fácilmente podrías adaptar esta aplicación a cualquier otro tipo de encuesta con muy poco esfuerzo.

SOFTWARE UTILIZADO

Este libro está basado en la versión estable de Xamarin.Forms 2.3.3.180, Prism para Xamarin.Forms 6.3.0 pre-1 y sqlite-net-pcl 1.2.1. Además, se ha usado Visual Studio .NET 2015 Enterprise Update 3, en inglés. Muy probablemente, en el momento de estar leyendo estas líneas, ya tengas a tu alcance versiones más nuevas de los paquetes de NuGet requeridos, así como también ya puedas descargar Visual Studio .NET 2017 RTM.

ESTRUCTURA DEL LIBRO

El libro está organizado y estructurado de tal manera que pueda leerse en orden de inicio a fin, ya que gradualmente en cada capítulo (a partir del 4) implementamos funcionalidades adicionales y modificaciones en la aplicación de encuestas. Para sacar el máximo provecho te sugiero leer los capítulos en orden y sobre todo practiques el código relacionado con cada uno de ellos.

CÓDIGO FUENTE

Todo el código fuente de este libro, así como el script de la base de datos de encuestas, lo puedes descargar de https://github.com/rdiazconcha.

1 INTRODUCCIÓN A XAMARIN

¡Bienvenido al mundo de desarrollo profesional con Xamarin! Si estás leyendo esta primera página seguramente eres de los(as) que les gusta comenzar a entender a detalle las bases de la plataforma de desarrollo que usarás para construir fantásticas soluciones. Yo soy igual que tú, así que te invito a que comencemos desde el principio: ¿qué es Xamarin?

¿Qué es Xamarin?

Para poder comprender bien qué es Xamarin debemos remontarnos a la historia de .NET.

Si ya tienes algunos años el mundo de desarrollo de aplicaciones para el ecosistema Windows, recordarás que en el año 2000 Microsoft anunció la plataforma de desarrollo .NET, la cual fue promovida como una plataforma de desarrollo cuyo objetivo era poder ejecutar aplicaciones en una gran cantidad de dispositivos y sistemas operativos diversos "basada en estándares de Internet" (lo que significara en aquel momento dicha frase rimbombante). En diciembre de ese año, la Infraestructura de Lenguaje Común o CLI –parte fundamental y corazón de .NET– fue publicada como el estándar abierto ECMA-335, abriendo un gran abanico de oportunidades para aquellos que quisieran hacer una implementación independiente. Este estándar define cómo las aplicaciones escritas en múltiples lenguajes de alto nivel pueden ejecutar en diferentes ambientes, sin la necesidad de reescribir esas aplicaciones para tomar en consideración las características únicas de dichos entornos. Una persona llamada Miguel de Icaza, de la empresa Ximian, se inspiró en esta especificación estándar y se dio a la tarea determinar si era posible implementarla también en el ecosistema Linux.

La implementación de esta especificación en el ecosistema Linux se llamó Mono, y permitía a los desarrolladores de ese sistema operativo usar un nuevo y novedoso lenguaje de programación llamado C# para poder construir aplicaciones. En el año 2001 se lanzó Mono como proyecto de código fuente abierto. En el mes de agosto del año 2003, la empresa Ximian –junto con sus fundadores Miguel de Icaza y Nat Friedman– fue adquirida por Novell (sí, aquella empresa responsable de su famoso sistema operativo de red Novell Netware), y al siguiente año (tres años después de haber lanzado el proyecto) Mono 1.0 era liberado en junio de 2004, mientras que Microsoft poco tiempo después lanzaba la versión 2.0 del .NET Framework. Algunos meses después, sería lanzada la versión 3.0 (cuyo nombre de producto originalmente era WinFX), versión que incluía las tecnologías Windows Communication Foundation (WCF), Windows Workflow Foundation (WF) y Windows Presentation Foundation (WPF).

La historia continuó con las versiones 3.5, 4.0, 4.5, etcétera. Microsoft al ser una empresa gigante, con recursos financieros y de capital humano muy grande, tuvo un ciclo de innovación mucho más rápido y mucho más contundente que el proyecto de código fuente abierto Mono, cuyo avance e innovación estaba supeditado a los esfuerzos, tiempo y energía de un limitado número de personas. Por tal motivo, nunca estaría a la par Mono con .NET en términos de funcionalidad e innovación. No obstante, Mono había logrado lo que Microsoft únicamente había podido soñar: el crear y ejecutar código verdaderamente multiplataforma, ya que Mono no se concentró solamente en el sistema operativo Linux, sino que su alcance incluía otras tecnologías y dispositivos como Android, PlayStation 3, Wii, Mac OSX y iOS. A través de los proyectos MonoTouch y MonoDroid ahora era posible escribir y ejecutar aplicaciones escritas con el lenguaje C# en los sistemas operativos móviles iOS y Android, respectivamente.

Eran tiempos fabulosos en el mundo tecnológico, y más para los amantes de .NET y C#.

Y pues, como en la mayoría de historias en los negocios: un pez grande se come a otro más pequeño. En el año 2011 la empresa Novell es adquirida por la empresa Attachmate, produciendo una gran cantidad de recortes de proyectos y personal, entre ellos las personas fundadoras de la empresa Ximian, que unos años atrás Novell adquirió. Pocas semanas después del recorte, sería dada a conocer la buena noticia de que Mono seguiría teniendo soporte a través de Xamarin, una empresa recién fundada y conformada por la gran mayoría de personas que fueron despedidas por parte de Attachmate.

Xamarin comenzó como una startup, obteniendo inyección financiera por parte de inversionistas externos, creando un portafolio de productos y servicios sumamente atractivos. No obstante, como cualquier empresa que está empezando,

el flujo de dinero es importante, así que las tecnologías que alguna vez fueron de código fuente abierto se perfilaron y aterrizaron como productos con un modelo de pago por anualidad, a cambio de soporte técnico, actualizaciones frecuentes, extraordinarias herramientas y una creciente y vibrante comunidad de seguidores. MonoTouch se convirtió en Xamarin.iOS, MonoDroid se convirtió en Xamarin.Android y MonoMac se convirtió en Xamarin.Mac. Su entorno de desarrollo MonoDevelop derivó en Xamarin Studio, con soporte tanto para Mac OSX como Windows.

Había comenzado otra etapa en la contienda de las plataformas de desarrollo multiplataforma para dispositivos móviles.

Ciertamente, Microsoft siempre miró de reojo la progresiva evolución e innovación de Mono y las tecnologías relacionadas. En el mes de febrero 2016, Microsoft anunciaría que había llegado a un acuerdo con la empresa Xamarin para poder adquirirla. Esa noticia, aunque ya era esperada por parte de la comunidad de .NET, fue sorprendente. Pero lo que fue aún más sorprendente fue que Microsoft anunciaba que haría de Xamarin un producto sin costo, incluido incluso en Visual Studio .NET Community (su edición gratuita). Pero si eso era sorprendente, lo que a continuación declaró Microsoft nos dejó a todos mudos: haría de Xamarin un proyecto Open Source. Microsoft con esta adquisición cumplía su sueño y profecía: el hacer realmente de .NET una plataforma de desarrollo para construir aplicaciones que ejecuten en un sinfín de sistemas operativos y dispositivos.

El resto de la historia se sigue escribiendo y la estamos viviendo: yo al escribir estas líneas y tú, al estar leyéndolas.

Bueno ya, entonces ¿qué es Xamarin?

Xamarin permite a los desarrolladores de C# poder construir aplicaciones multiplataforma para los sistemas operativos móviles y dispositivos más importantes del mundo: Android y iOS, además de poder compartir código con aplicaciones del ecosistema de Windows. Una de las características más importantes de Xamarin es que nos permite construir aplicaciones nativas con Interfaz de Usuario y desempeño nativo, tal cual como si estuviéramos desarrollando con Java para Android o con Objective-C o Swift para iOS, pero reutilizando al máximo nuestros conocimientos en el lenguaje C# y en la plataforma de desarrollo .NET.

Técnicamente hablando, Xamarin expone o “refleja” las APIs de cada sistema operativo subyacente, para poderlas utilizar con .NET, el CLR y el lenguaje C#. Adicionalmente a esta característica bien recibida por los desarrolladores de C#, lo que diferencia a Xamarin es su capacidad de compartir una base de código que pueda ser reutilizada por diferentes plataformas.

Métodos para compartir el código

En Xamarin hay tres métodos para compartir el código: Bibliotecas de Clases Portables (o PCL por sus siglas en inglés), los proyectos de Código compartido y los proyectos de .NET Standard.

BIBLIOTECAS DE CLASES PORTABLES

La Biblioteca de Clases Portable (o PCL por su nombre en inglés "Portable Class Library") es el mecanismo más usado y sugerido para construir aplicaciones con Xamarin, ya que compila un ensamblado de tipo DLL y resulta una estrategia más limpia en el momento de construir aplicaciones. Este es el método que utilizaremos a lo largo de todo este libro.

CÓDIGO COMPARTIDO

Con este método se crea un "concentrador" en donde se colocan archivos de código y son automáticamente vinculados a los diferentes proyectos que constituyen la solución. En este caso, el código no compila a un DLL por separado, sino que compila como parte de la aplicación concreta nativa. Debido a esto, el código rápidamente se puede inundar con bucles #IF…#ENDIF, haciendo su lectura un tanto menos eficiente. En mi particular experiencia, este mecanismo no lo recomiendo, a menos que se tenga una excelente razón para ello y que no se pueda resolver a través de una Biblioteca de Clases Portable.

.NET STANDARD

El .NET Standard es un conjunto de APIs que cada runtime de .NET debe implementar. En el momento de estar escribiendo este libro, la versión más ubicua es la 1.6. Por sus cualidades actuales, este no será el mecanismo que usaré en el libro, sin embargo, en un corto plazo definitivamente será la estrategia recomendada cuando se construyan aplicaciones multiplataforma.

El siguiente diagrama muestra el diseño lógico de una aplicación construida con Xamarin:

El diagrama muestra la estrategia "clásica" de construcción de aplicaciones con Xamarin: tener un código común que incluya la lógica de negocio, servicios, acceso a datos, almacenamiento, etcétera., y los elementos de la Interfaz de Usuario y lógica específica para cada sistema operativo subyacente en cada proyecto de la plataforma concreta.

Gracias a este mecanismo, podemos obtener un gran porcentaje de código común entre las diferentes plataformas, por lo que el mantenimiento de código se reduce y se incrementa la velocidad en la implementación de nuevas características o correcciones.

Sin embargo, esta estrategia o modelo clásico de construcción de aplicaciones con Xamarin rápidamente se vuelve no tan eficiente y ágil, sobre todo cuando se trata de aplicaciones de negocio, privadas, que no necesariamente requieren interfaces de usuario sofisticadas o de alta complejidad.

Xamarin se dio cuenta de esto y en el año 2014 creó un framework llamado Xamarin.Forms el cual es una abstracción de los principales elementos visuales de la Interfaz de Usuario de cada sistema operativo, maximizando aún más el código compartido que podemos tener en nuestros proyectos.

FUNDAMENTOS DE XAMARIN.FORMS

¿Qué es Xamarin.Forms?

Xamarin.Forms es un framework multiplataforma de elementos visuales, desplegable a través de NuGet. Estos elementos visuales son una abstracción de los elementos visuales nativos de cada sistema operativo concreto donde ejecuta la aplicación. Xamarin.Forms nos permite compartir hasta el 100% de código multiplataforma, ya que a diferencia del método "clásico", el código compartido también incluye las vistas o Interfaz de Usuario de la aplicación.

¿Xamarin o Xamarin.Forms?

Una de las principales dudas que se tienen alrededor del desarrollo de aplicaciones con Xamarin es si utilizamos Xamarin o Xamarin.Forms, como si se tratasen de dos productos completamente diferentes entre sí. Esto es completamente falso, y sospecho que esta duda está basada en cómo se entendió el mensaje cuando se lanzó la plataforma por primera vez. He escuchado a personas recomendar evitar Xamarin.Forms como opción para el desarrollo de aplicaciones multiplataforma. ¡No podrían estar más equivocados! En realidad, Xamarin.Forms es únicamente la abstracción de algunos de los elementos más relevantes en la Interfaz de Usuario de los diversos sistemas operativos que soporta, pero sigue siendo Xamarin e incluso no podría ejecutar sin él.

Mi recomendación a todo desarrollador profesional o amateur, empleado o independiente, que desee construir soluciones móviles de negocio y/o empresariales

con Xamarin es que comience con Xamarin.Forms y no con Xamarin "clásico", ya que la reutilización de código es mucho mayor, la dificultad se reduce y por lo tanto el costo y el tiempo asociados para construir dichas soluciones también se ven reducidos.

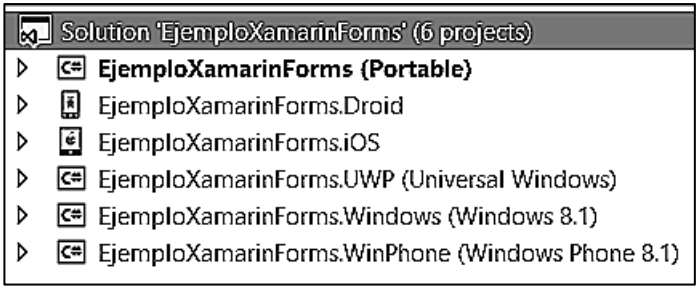
La siguiente tabla nos muestra una guía de cuándo utilizar Xamarin.Forms o cuándo utilizar Xamarin "clásico".

Xamarin.Forms	Xamarin.iOS / Xamarin.Android
• Apps de negocio • Apps de datos • Apps que requieran poca funcionalidad específica de cada plataforma • Apps donde compartir el código sea más importante que una IU sofisticada	• Apps que requieran interacción muy especializada • Apps que usen muchas APIs específicas de una plataforma • Apps en donde la IU sea más importante que compartir el código

ANATOMÍA DE UNA SOLUCIÓN

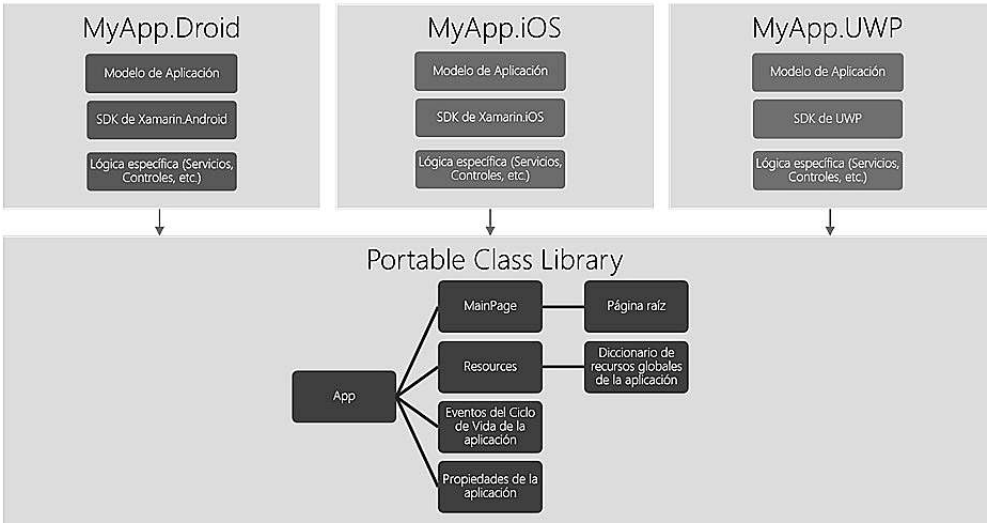
Cuando creamos una aplicación de Xamarin.Forms por medio de Visual Studio .NET usando la plantilla de Biblioteca de Clases Portables, Visual Studio crea diversos proyectos en la solución. El primer proyecto es la Biblioteca de Clases Portable la cual contendrá el código común y compartido multiplataforma que será usado por todo el resto de proyectos. Adicionalmente, Visual Studio .NET creará un proyecto por cada sistema operativo del que encuentre su SDK correctamente instalado y configurado. Ahora bien, nosotros podemos tener más control sobre qué proyectos exactamente crear, si creamos manualmente la solución y sus proyectos.

La siguiente figura muestra el Solution Explorer de Visual Studio .NET después de haber creado un proyecto llamado "EjemploXamarinForms", por medio de la plantilla "Blank Xaml App (Xamarin.Forms Portable)":



ARQUITECTURA DE XAMARIN.FORMS

Comprender a detalle la arquitectura de una aplicación de Xamarin.Forms es necesario para poder construir aplicaciones robustas y elegantes con esta tecnología. El siguiente diagrama muestra la arquitectura de una aplicación de Xamarin.Forms, la cual utiliza el método de Bibliotecas de Clases Portables para compartir el código y que apunta a tres sistemas operativos: Android, iOS y UWP.



Clase Application

Comenzamos hablando de la Biblioteca Clases Portable. Este proyecto contendrá la clase que es el punto de entrada para la aplicación de Xamarin.Forms. Esta clase (regularmente llamada App y contenida en el archivo físico App.cs) es de tipo Xamarin.Forms.Application.

La siguiente tabla describe los miembros más importantes de la clase App:

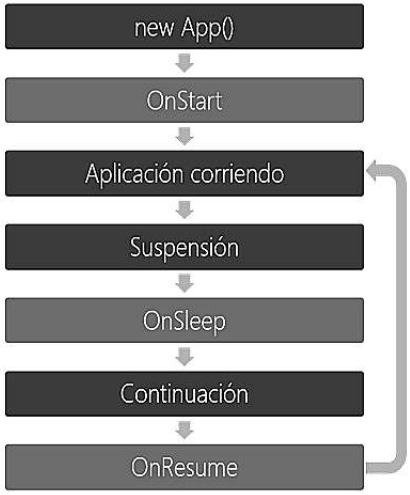
MainPage	Propiedad que representa la raíz visual de la aplicación
Resources	Propiedad de tipo KeyValuePair<string, object> que representa los recursos globales para la aplicación. Para más información acerca de los recursos, consulta el capítulo “El Lenguaje XAML” donde se describen con más detalle
OnStart()	Método virtual que se ejecuta cuando la aplicación se inicia por primera vez

OnSleep()	Método virtual que se ejecuta cuando la aplicación se está yendo a segundo plano
OnResume()	Método virtual que se ejecuta cuando la aplicación reanuda su ejecución después de haber estado en segundo plano
Properties	Propiedad de tipo IDictionary<string, object> en donde podemos almacenar cualquier tipo de objeto serializable para guardar el estado de la aplicación
SavePropertiesAsync()	Método que podemos ejecutar para guardar proactivamente en el almacenamiento local del dispositivo los objetos del diccionario Properties, y de esa manera evitar el riesgo de que no sean guardados ya sea porque ocurrió un error en la app o porque el dispositivo fue apagado en ese justo momento

Cada uno de los proyectos concretos referencia a la Biblioteca de Clases Portable, además de referenciar al SDK de Xamarin para cada sistema operativo. Es importante recordar que si bien Xamarin es una plataforma de desarrollo multiplataforma, cada sistema operativo tendrá sus particularidades, es decir, cada sistema operativo cuenta con un modelo de desarrollo específico.

Ciclo de vida

Como comentábamos anteriormente, el ciclo de vida de las aplicaciones con Xamarin.Forms está expuesto a través de los métodos OnStart(), OnSleep() y OnResume() de la clase Application. En cada método podemos escribir código que responda en el momento que el ciclo de vida tenga. Por ejemplo, si queremos escribir código que ejecute justo cuando inicia la aplicación, podemos ponerlo en el método OnStart(). De igual manera, si queremos ejecutar código en el momento que la aplicación se esté yendo a segundo plano, entonces lo pondríamos en el método OnSleep(), o si queremos ejecutar alguna lógica cuando la aplicación esté reanudando su ejecución después de haber estado en segundo plano, entonces ese código lo implementaríamos en el método OnResume(). El siguiente diagrama muestra el ciclo de vida completo de las aplicaciones con Xamarin.Forms:



Ejecutando Xamarin.Forms en cada proyecto concreto

Ejecutar Xamarin.Forms requiere que se cumplan los siguientes tres pasos:

1. Heredar de una clase base especial
2. Iniciar el motor de Xamarin.Forms
3. Cargar el objeto de tipo Application (App.cs) implementada en la Biblioteca de Clases Portable

A continuación describiremos con más detalle estos pasos requeridos para cada proyecto.

APLICACIÓN ANDROID

Para ejecutar una aplicación de Xamarin.Forms en Android, debemos realizar algunas modificaciones ligeras al proyecto de Xamarin.Android que haya creado la plantilla de Visual Studio .NET o que hayamos creado nosotros manualmente. En el proyecto de Android tenemos una clase llamada MainActivity, la cual es el punto de entrada de la aplicación. Es decir, esto es lo que primero que se interpreta del código cuando decidimos ejecutar nuestra app en el dispositivo. Para que Xamarin.Android ejecute Xamarin.Forms, esta clase MainActivity debe heredar de la clase base Xamarin.Forms.Platform.Android.FormsApplicationActivity. En el constructor, debemos además invocar el método Init() de la clase Xamarin.Forms.Forms y por último debemos ejecutar el método LoadApplication(), pasando como parámetro una instancia del objeto que representa la aplicación en la PCL. En el siguiente fragmento de código nos muestra la clase MainActivity una vez realizados los cambios necesarios para ejecutar Xamarin.Forms:


```
namespace MyApp.Droid
{
    [Activity(Label = "MyApp", Icon = "@drawable/icon",
MainLauncher = true, ConfigurationChanges =
ConfigChanges.ScreenSize | ConfigChanges.Orientation)]
    public class MainActivity :
global::Xamarin.Forms.Platform.Android.FormsApplicationAc
tivity
    {
        protected override void OnCreate(Bundle bundle)
        {
            base.OnCreate(bundle);
            global::Xamarin.Forms.Forms.Init(this,
bundle);
            LoadApplication(new App());
        }
    }
}
```

Nota: Es mejor idea usar la clase FormsAppCompatActivity, pero por cuestiones de sencillez en este momento usaremos FormsApplicationActivity. Más adelante en el libro, retomaremos este importante asunto.

APLICACIÓN IOS

En el proyecto de Xamarin.iOS, encontraremos una clase llamada AppDelegate la cual debe heredar de la clase base Xamarin.Forms.Platform.iOS.FormsApplicationDelegate. En el método FinishedLaunching(), debemos invocar el método Init() de la clase Xamarin.Forms.Forms e inmediatamente después debemos invocar el método LoadApplication(), pasando como parámetro el objeto que representa la aplicación en la PCL. El siguiente fragmento de código muestra estos cambios en la clase AppDelegate del proyecto de Xamarin.iOS:

```
namespace MyApp.iOS
{
    [Register("AppDelegate")]
    public partial class AppDelegate :
global::Xamarin.Forms.Platform.iOS.FormsApplicationDelega
te
    {
        public override bool
FinishedLaunching(UIApplication app, NSDictionary
options)
        {
            global::Xamarin.Forms.Forms.Init();
            LoadApplication(new App());
            return base.FinishedLaunching(app,
options);
        }
    }
}
```

APLICACIÓN UNIVERSAL WINDOWS PLATFORM (UWP)

En el proyecto de UWP, encontraremos la clase App en el archivo físico App.xaml.cs. Esta clase representa el punto de entrada de la aplicación UWP. En dicha clase, encontraremos el método OnLaunched() en donde debemos iniciar el motor de Xamarin.Forms a través de la ejecución del método Init() justo después de haber creado el objeto Frame y antes de establecerlo como la raíz visual de la ventana. Adicionalmente, la clase MainPage debe heredar de la clase base de WindowsPage, y en su constructor debemos de cargar el objeto que representa la aplicación en la PCL, a través de la ejecución del método LoadApplication().

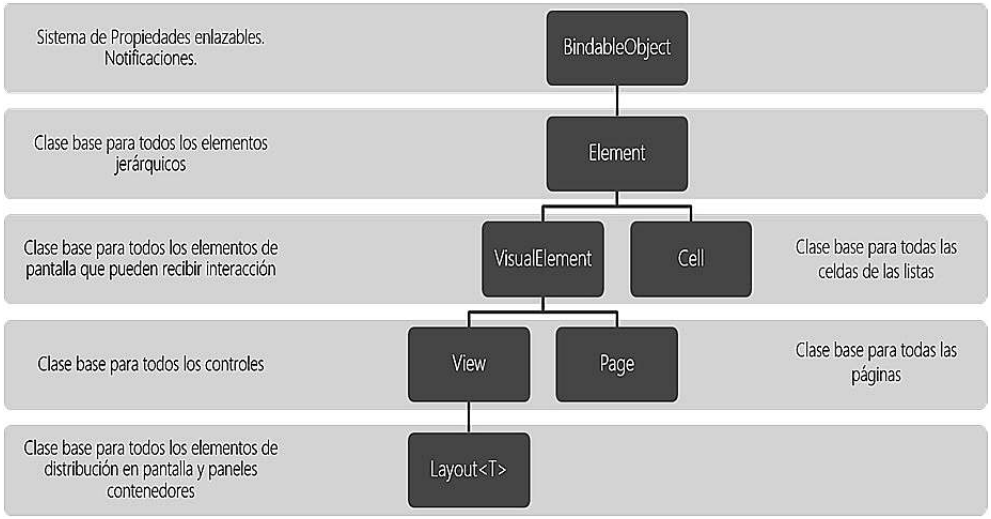
El siguiente fragmento de código muestra los cambios necesarios tanto en la clase App como en MainPage del proyecto UWP:

```
//App.xaml.cs
 protected override void
OnLaunched(LaunchActivatedEventArgs e) {
    …
    rootFrame = new Frame();
```

```
        Xamarin.Forms.Forms.Init(e);
        Window.Current.Content = rootFrame;
        …
        rootFrame.Navigate(typeof(MainPage), e.Arguments);
    }

    //MainPage.xaml.cs
    public sealed partial class MainPage : WindowsPage {
        public MainPage() {
            this.InitializeComponent();
            LoadApplication(new MyApp.App());
        }
    }
```

Jerarquía de clases

El siguiente diagrama nos muestra la jerarquía de las clases más destacables en el API de Xamarin.Forms:



BINDABLEOBJECT

Esta clase es la clase base de todas en el API de Xamarin.Forms. En esta clase podemos encontrar el Sistema de Propiedades Enlazables, así como los miembros necesarios para las notificaciones entre objetos. Toda clase en el API de Xamarin.Forms hereda de manera directa o indirecta de esta clase base, por lo que el Sistema de Propiedades Enlazables y las notificaciones son ubicuas en todo el API.

ELEMENT

La clase Element es la clase base para todos los elementos jerárquicos en Xamarin.Forms.

VISUALELEMENT

La clase VisualElement es la clase base para todos los elementos en pantalla que pueden recibir interacción. Incluye la siguiente funcionalidad:

- Transformaciones
- Comportamientos
- Enfoque
- Navegación
- Distribución y medición de elementos

CELL

La clase Cell es la clase base para toda las celdas en los controles de lista (por ejemplo ListView, que veremos más tarde en este libro).

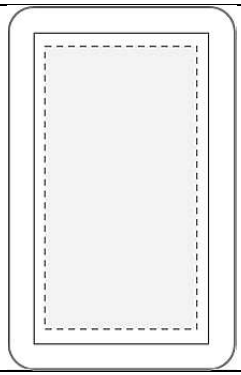
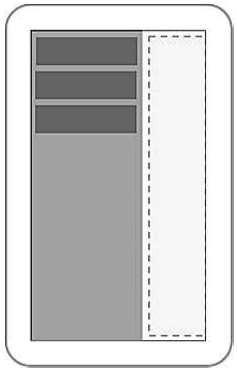
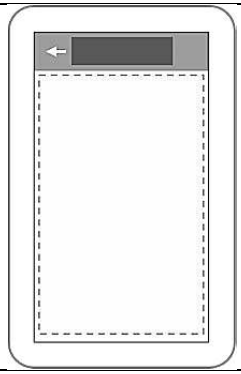
VIEW

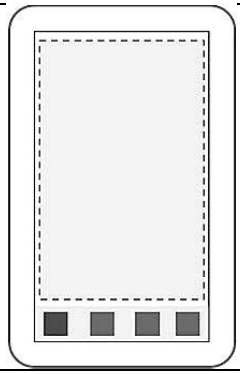
La clase View es la clase base para todos los controles interactivos que usan un "Renderer" para definir su apariencia. De ahí que en muchos foros en idioma inglés y en la documentación de Xamarin.Forms encontraremos el término "view" para representar cualquier tipo de control visual interactivo. Algunos de sus miembros más importantes son las siguientes propiedades:

- HorizontalOptions
- VerticalOptions
- GestureRecognizers

PAGE

Esta clase es la clase base para todas las páginas en Xamarin.Forms. Las páginas representan un elemento visual que ocupa toda la pantalla. A continuación se enlistan los diferentes tipos de páginas incluidas en Xamarin.Forms:

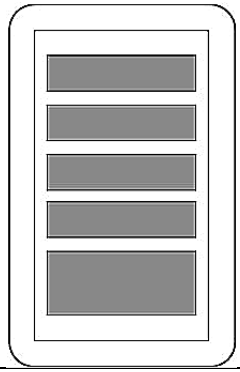
ContentPage	Este tipo de página es el más común ya que su objetivo es desplegar solo un elemento visual. Ese elemento visual a su vez puede ser un contenedor que tenga múltiples hijos	
MasterDetailPage	Este tipo de página puede tener dos paneles de información: uno para un menú (propiedad "Master") y el otro para el contenido relacionado con el elemento del menú seleccionado (propiedad "Detail"). Más adelante en este libro, esta página desempeñará un papel muy importante en la aplicación que construiremos	
NavigationPage	Este es un tipo especial de página, que habilita la funcionalidad de navegación entre diferentes páginas	

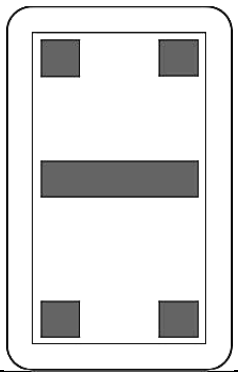
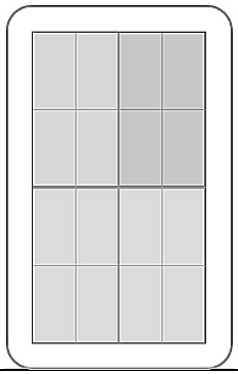
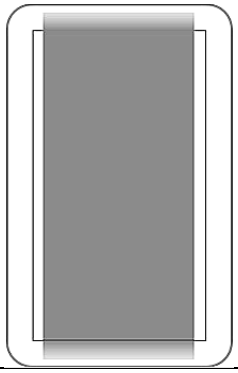
TabbedPage	Este tipo de página puede contener otras y cada una de ellas mostrarlas como pestañas	

LAYOUT<T> Y LAYOUT

Estas clases son las clases base para todos los elementos de posicionamiento en pantalla y paneles contenedores como:

- StackLayout
- AbsoluteLayout
- Grid
- ScrollView
- Etcétera

StackLayout	Contenedor para apilar los elementos hijos de manera vertical u horizontal	

AbsoluteLayout	Contenedor para posicionar los elementos hijos de forma absoluta	
Grid	Contenedor para posicionar los elementos hijos a través de columnas y filas, tal y como si fuera una tabla	
ScrollView	Contenedor que permite hacer scroll cuando el tamaño de sus elementos hijos desborda el tamaño de la pantalla	

SISTEMA DE PROPIEDADES ENLAZABLES

Una de las características más trascendentales de Xamarin.Forms es el Sistema de Propiedades Enlazables, ya que él es el responsable de calcular los valores de las propiedades de los elementos visuales y además notificar cuándo un valor ha cambiado. La clase base BindableObject es la puerta de entrada al Sistema de

Propiedades Enlazables y habilita a todas sus clases derivadas para que puedan usarlas.

¿Qué es una Propiedad Enlazable?

Las Propiedades Enlazables son propiedades cuyo valor no lo determinas tú, sino que su valor está en función de una gran cantidad de estímulos a su alrededor. Imagínate este escenario: tienes un elemento en pantalla que expone una propiedad que representa su ancho. Has declarado que su valor es 200, además estás aplicando un estilo que cambia su valor a 300, luego estás animando dicha propiedad a través de una animación que va de 0 a 400 en 3 segundos. ¿Cuál sería entonces su valor final? O peor aún: ¿me podrías decir cuál sería su valor en el segundo 1.7 de la animación? Tal vez estarás de acuerdo conmigo que sería sumamente difícil determinarlo por nosotros mismos. En vez de eso, en Xamarin.Forms (y en todas las tecnologías basadas en XAML) nos basamos en las Propiedades Enlazables, las cuales nos proporcionan la siguiente funcionalidad:

- Enlace de Datos
- Estilización y Plantillas
- Animación
- Notificación de cambio de valor

PRECEDENCIA DE VALORES

Todos esos "estímulos" de los que hablábamos anteriormente tienen diferente peso o prioridad. El siguiente diagrama nos muestra la precedencia de valores que considera el Sistema de Propiedades Enlazables. Podemos apreciar que el estímulo que tiene más peso es una animación que está activa, y la que tiene menos peso es el valor predeterminado que tenga la Propiedad Enlazable en cuestión.

BindableObject

Regresemos al tema de la clase BindableObject. Como lo mencionamos anteriormente, esta es la clase base para todos los objetos del Sistema de Propiedades Enlazables. Algunos de sus miembros más notables son los siguientes:

GetValue()	Método que regresa el valor de una Propiedad Enlazable
SetValue()	Método que establece el valor de una Propiedad Enlazable
ClearValue()	Método que restablece el valor de una Propiedad Enlazable a su valor predeterminado
BindingContext	Propiedad que representa un contexto de Enlace de Datos

BindableProperty

Una Propiedad Enlazable tiene más funcionalidad que una propiedad tradicional. Esto se debe a que todas las Propiedades Enlazables en Xamarin.Forms son de tipo BindableProperty. Esta clase encapsula todas las funcionalidades que tienen las Propiedades Enlazables, como enlace de datos, notificación en el caso de cambiar de valor, etc. Por lo tanto y como seguramente ya adivinaste, la declaración de este tipo de propiedades debe ser diferente a las propiedades tradicionales. ¡Acertaste! La declaración de las Propiedades Enlazables debe cumplir con los siguientes requisitos:

- Deben ser tipo BindableProperty
- Deben estar declaradas como public
- Deben ser estáticas
- Deben ser de solo lectura
- Su nombre debe tener el sufijo “Property”

El siguiente fragmento de código nos muestra una Propiedad Enlazable que hemos declarado en una clase llamada MyClass. Observa que estamos usando el método estático Create() para declarar la Propiedad Enlazable, indicando el nombre que deseamos que tenga la propiedad (Text), el tipo de dato que devolverá la propiedad (string) y el tipo donde está declarando la propiedad (MyClass):

```
public class MyClass : BindableObject
{
    public static readonly BindableProperty
TextProperty = BindableProperty.Create("Text",
typeof(string),
```

```
            typeof(MyClass));
    }
```

Ahora bien, lo que estás viendo en el fragmento de código anterior es únicamente la declaración del campo estático con la Propiedad Enlazable. Generalmente, las Propiedades Enlazables no se usan de manera directa, sino que accedemos a ellas a través de una propiedad CLR tradicional. Esta propiedad CLR tradicional es únicamente una "fachada" para acceder al valor que nos devuelva el Sistema de Propiedades Enlazables. Entonces ¿cómo le preguntamos al Sistema de Propiedades Enlazables acerca del valor de una Propiedad Enlazable o cómo le indicamos que deseamos establecer un nuevo valor? La respuesta está en los métodos GetValue() y SetValue() de la clase BindableObject que describimos anteriormente. La propiedad CLR tradicional, en vez de guardar o escribir el dato en un campo privado encapsulado, delegará esta responsabilidad al Sistema de Propiedades Enlazables, invocando el método GetValue() en el accesor get{} y el SetValue() en el accesor set{}.

El siguiente fragmento de código muestra la declaración de la propiedad CLR "fachada", que hace uso del Sistema de Propiedades Enlazables para leer y establecer el valor de la propiedad.

```
public string Text
{
    get
    {
        return (string)GetValue(TextProperty);
    }
    set
    {
        SetValue(TextProperty, value);
    }
}
```

Gracias a esta propiedad CLR, podemos usar las Propiedades Enlazables como si fuera cualquier propiedad tradicional. El API de Xamarin.Forms está repleto de ejemplos de Propiedades Enlazables, como lo demuestra la siguiente figura que exhibe el código de la clase base VisualElement:


```
public class VisualElement : Element, IAnimatable, IVisualElementController, IElementController, IResourcesProvider
{
    public static readonly BindableProperty AnchorXProperty;
    public static readonly BindableProperty AnchorYProperty;
    public static readonly BindableProperty BackgroundColorProperty;
    public static readonly BindableProperty BehaviorsProperty;
    public static readonly BindableProperty HeightProperty;
    public static readonly BindableProperty HeightRequestProperty;
    public static readonly BindableProperty InputTransparentProperty;
    public static readonly BindableProperty IsEnabledProperty;
    public static readonly BindableProperty IsFocusedProperty;
    public static readonly BindableProperty IsVisibleProperty;
    public static readonly BindableProperty MinimumHeightRequestProperty;
    public static readonly BindableProperty MinimumWidthRequestProperty;
    public static readonly BindableProperty NavigationProperty;
    public static readonly BindableProperty OpacityProperty;
    public static readonly BindableProperty RotationProperty;
    public static readonly BindableProperty RotationXProperty;
    public static readonly BindableProperty RotationYProperty;
    public static readonly BindableProperty ScaleProperty;
    public static readonly BindableProperty StyleProperty;
    public static readonly BindableProperty TranslationXProperty;
    public static readonly BindableProperty TranslationYProperty;
    public static readonly BindableProperty TriggersProperty;
    public static readonly BindableProperty WidthProperty;
    public static readonly BindableProperty WidthRequestProperty;
    public static readonly BindableProperty XProperty;
    public static readonly BindableProperty YProperty;
```

El siguiente fragmento de código muestra el ejemplo de uso de algunas Propiedades Enlazables vía código:

```
var button1 = new Button();
button1.WidthRequest = 300;
button1.HeightRequest = 200;
button1.Opacity = 0.5;
```

Propiedades Adjuntas

Las Propiedades Adjuntas son un tipo especial de Propiedad Enlazable. Estas propiedades notifican a un elemento padre que un elemento hijo requiere cierto valor de alguna propiedad, cuando dicha propiedad no está declarada en el tipo de clase del hijo en cuestión, sino en el padre. Para entender este concepto, analicemos el siguiente fragmento de código:

```
<ClasePadre>
   <Elemento ClasePadre.Propiedad = "Valor" />
</ClasePadre>
```

Es de sumo interés destacar que la propiedad a la que le estamos estableciendo el valor, no pertenece al elemento hijo, sino al padre. A través de la sintaxis ClasePadre.Propiedad podemos establecer el valor deseado. No obstante, las Propiedades Adjuntas requieren otro tipo de declaración a través del método CreateAttached() también de la clase BindableProperty. Un ejemplo concreto y

contundente acerca de las Propiedades Adjuntas son las propiedades Column, Row, ColumnSpan y RowsSpan de la clase Grid. El siguiente fragmento de código muestra la declaración de la propiedad Column de la clase Grid, tomada del repositorio de código fuente original de Xamarin.Forms en GitHub:

```
  public static readonly BindableProperty ColumnProperty
= BindableProperty.CreateAttached("Column", typeof(int),
typeof(Grid), default(int), validateValue: (bindable,
value) => (int)value >= 0);
```

PROPIEDADES ADJUNTAS VÍA CÓDIGO

Ya que las Propiedades Adjuntas son Propiedades Enlazables, también puedes utilizar los métodos GetValue() y SetValue() para leer y/o escribir el valor de estas propiedades. No obstante, para evitar el boxing/unboxing que resulta en la ejecución de estos métodos (ya que regresan y solicitan object), generalmente las clases que declaran Propiedades Adjuntas también declaran los métodos necesarios GetXXX y SetXXX para leer y escribir los valores de estas propiedades, solicitando el tipo de dato específico, evitando de esta manera un impacto negativo en el desempeño que ocurre con el proceso de boxing y unboxing en el CLR. El siguiente fragmento de código muestra el uso de los métodos GetColumn() y SetColumn() declarados en la clase Grid de Xamarin.Forms:

```
if (Grid.GetColumn(button1) == 0)
{
    Grid.SetColumn(button1, 1);
}
```

MANEJO DE EVENTOS

En Xamarin.Forms podemos manejar los eventos de los elementos visuales a través de tres maneras diferentes:

- Desde XAML
- A través de código usando la sintaxis estándar
- A través de código usando Expresiones Lambda

Manejo de eventos desde XAML

Con esta técnica, declaramos el nombre del manejador del evento directamente en el cuerpo del documento XAML. Visual Studio .NET creará automáticamente el cuerpo del manejador en la clase de code-behind relacionada.

El siguiente fragmento de código muestra el uso de la sintaxis con XAML para manejar el evento Clicked de un botón.

```
<!-- En XAML-->
<Button Clicked="Button_Clicked" />
//En el code-behind
private void Button_Clicked(object sender, EventArgs e)
{
    //Manejador del evento
}
```

Manejo de eventos a través de código usando la sintaxis estándar

Podemos usar la sintaxis tradicional del lenguaje C# para manejar los eventos, no solamente para los elementos visuales sino para cualquier objeto. Esta técnica se basa en el operador sobrecargado +=, y es el mecanismo recomendado la mayoría de las veces, ya que podemos "desasignar" un manejador a través del uso del operador sobrecargado -=, evitando así las fugas de memoria que pudiesen ocurrir. El siguiente fragmento de código muestra el uso de la sintaxis estándar para declarar el manejador del evento Clicked de un botón:

```
this.button1.Clicked += button1_Clicked;
…
private void button1_Clicked(object sender, EventArgs
e)
{
    //Manejo del evento
}
```

Manejo de eventos a través de código usando Expresiones Lambda

Las Expresiones Lambda son un tipo especial de métodos anónimos, los cuales no requieren la declaración del cuerpo del método por separado sino "en línea", cumpliendo con la signatura del delegado en el que se basa el evento en cuestión. El compilador infiere los tipos de los argumentos por lo que tampoco es necesario que se especifiquen de manera explícita.

El siguiente fragmento de código muestra el uso de una Expresión Lambda para manejar el evento Clicked de un botón.

```
//sender es object y args es EventArgs
this.button1.Clicked += (sender, args) =>
{
    //Manejo del evento
};
```

Recuerda que por su naturaleza, no podrás desasignar un manejador declarado por medio de una Expresión Lambda, por lo que si buscas desasignar el manejador para limpiar recursos de manera adecuada, deberás optar por el uso de la sintaxis estándar usando el cuerpo del manejador por separador y el operador sobrecargado +=.

2 EL LENGUAJE XAML

INTRODUCCIÓN

XAML es un extraordinario lenguaje ubicuo en el ecosistema de aplicaciones de Microsoft. Comprender y dominar XAML es imprescindible para cualquier desarrollador que desee construir soluciones sofisticadas en Xamarin.Forms, por lo que en este capítulo nos concentraremos exclusivamente en explicar este lenguaje.

¿Qué es XAML?

XAML es el acrónimo de e**X**tensible **A**pplication **M**arkup **L**anguage. XAML es un lenguaje que tiene como objetivo expresar de manera declarativa las interfaces de usuario en aplicaciones de Xamarin.Forms, WPF, Silverlight, UWP, Windows Phone, etcétera. En su concepto más fundamental, el objetivo de XAML es ser un lenguaje de instanciación de objetos. XAML no es nada nuevo, de hecho lo tenemos disponible en WPF desde el año 2006. Microsoft ha sido muy claro al respecto de su estrategia tecnológica con respecto a la declaración de interfaces de usuario en su ecosistema.

Como todo lenguaje de marcado, XAML se expresa a través de etiquetas y de algunas reglas básicas que a continuación explicaremos.

Reglas básicas de XAML

XAML está basado en XML, y por lo tanto, tiene sus mismas reglas:

- Debe haber un solo elemento raíz
- Todo elemento debe cerrarse
- Sensible a mayúsculas y minúsculas

- Hay elementos y subelementos
- Hay atributos

En un documento de XAML, por el simple hecho de agregar un elemento, estamos instanciando un objeto del tipo de dicho elemento. Por ejemplo, el siguiente fragmento de código XAML instancia un nuevo objeto de tipo Entry, usando su constructor público sin parámetros:

```
<Entry />
```

Los atributos nos permiten cambiar las características de los elementos. Los atributos están mapeados a las propiedades públicas que dicha clase expone. Por ejemplo, en el siguiente fragmento de código podemos apreciar cómo estamos estableciendo el valor "Hola" a la propiedad pública Text de la clase Entry.

```
<Entry Text="Hola" />
```

En XAML, el anidamiento de un elemento dentro de otro implica pertenencia. El objeto representado por el subelemento se almacenará en alguna de las propiedades que exponga la clase padre, dependiendo de cuál sea esta. Por ejemplo, en el siguiente fragmento de código estamos estableciendo el objeto de tipo Entry como un elemento "hijo" del objeto Grid.

```
<Grid>
    <Entry />
</Grid>
```

El resultado del anterior código es que el objeto Entry se agregará a la colección Children de la clase Grid. ¿Cómo supo el intérprete XAML de Xamarin.Forms que debía poner dicho objeto en esa colección? La respuesta está en el atributo ContentProperty, la cual decora algunas de las clases en el API de Xamarin.Forms, principalmente aquellas que son contenedores. El siguiente código muestra el código fuente original de la clase Layout<T> disponible en GitHub.

```
[ContentProperty("Children")]
public abstract class Layout<T> : Layout,
IViewContainer<T> where T : View { … }
```

Como podemos apreciar en el anterior código, el atributo ContentProperty indica que los elementos hijos se agreguen a la colección Children de esta clase base.

Ya que el objetivo de XAML es instanciar objetos de una manera declarativa, todo elemento del documento XAML está mapeado a una clase en el API de Xamarin.Forms. La siguiente tabla muestra algunos ejemplos de elementos y su clase correspondiente:

<Label />	Xamarin.Forms.Label
<Entry />	Xamarin.Forms.Entry
<Button />	Xamarin.Forms.Button
<ListView />	Xamarin.Forms.ListView
<ContentPage />	Xamarin.Forms.ContentPage

¿Cómo sabe el intérprete de XAML que, por ejemplo, Label es un Xamarin.Forms.Label y no alguna otra clase con el mismo nombre que tal vez tengas en tu proyecto? La respuesta está en los espacios de nombres XML.

Espacios de nombres XML

Los espacios de nombres XML se usan para organizar y agrupar de manera lógica las clases. En Xamarin.Forms hay dos espacios de nombres importantes, y la siguiente tabla describe cada uno de ellos:

http://xamarin.com/schemas/2014/forms	Espacio de nombres base. Abarca varios espacios de nombres CLR
http://schemas.microsoft.com/winfx/2009/xaml	Espacio de nombres del lenguaje XAML

Como podemos apreciar en la anterior tabla, la sintaxis de ambos espacios de nombres es similar a un URL de Internet, pero no pierdas el tiempo escribiendo esta dirección en un navegador ya que realmente no son páginas existentes, sino que simplemente es una sintaxis compatible con el documento XAML.

Para poner a nuestro alcance todos los miembros incluidos en un espacio de nombres, utilizamos el atributo xmlns, ya sea en el elemento raíz o en alguno de los elementos hijos del documento. De manera opcional, podemos asignar un alias a dicho espacio de nombres. Esto es similar a la sentencia using en C# o Imports en Visual Basic. El siguiente fragmento de código muestra el uso del atributo xmlns para traer a nuestro ámbito todos los miembros incluidos en ambos espacios de nombres. También cabe destacar que estamos utilizando el alias “x” para el espacio de nombres del lenguaje XAML. Adicionalmente, podrás observar que estamos utilizando el atributo Class del alias “x”, el cual indica el nombre de la clase de código de code-behind relacionada para este documento de XAML.

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage
xmlns="http://xamarin.com/schemas/2014/forms"

xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="Capitulo2.MainPage">
</ContentPage>
```

MIEMBROS DEL ESPACIO DE NOMBRES DEL LENGUAJE XAML

La siguiente tabla muestra algunos de los miembros más importantes del espacio de nombres del lenguaje XAML:

x:Class	Indica el nombre totalmente cualificado de la clase de código relacionada
x:FactoryMethod	Indica el método estático a utilizar para inicializar el objeto en vez del constructor
x:Key	Especifica el identificador único de un objeto dentro de un Diccionario de Recursos
x:Name	Especifica el nombre del objeto para poder referenciarlo en el código
x:Arguments	Especifica los argumentos para un constructor para inicializar un objeto
x:TypeArguments	Indica el tipo de los argumentos para un constructor no predeterminado
x:Null	Representa el valor null

Sintaxis de subelementos

Todas las propiedades de un elemento se pueden expresar como subelementos del mismo. A esta técnica le denominamos la sintaxis de subelementos. El siguiente fragmento de código nos muestra la declaración de un botón, al que se le está asignando algunos valores para las propiedades Text, TextColor y FontSize.

```
<Button Text="Botón" TextColor="Red" FontSize="22" />
```

A continuación, la declaración del mismo botón pero ahora utilizando la sintaxis de subelementos:

```
<Button>
  <Button.Text>Botón</Button.Text>
  <Button.TextColor>Red</Button.TextColor>
  <Button.FontSize>22</Button.FontSize>
</Button>
```

¿Cuál de los dos fragmentos es más fácil de leer? Sin duda alguna el primero. No obstante, la sintaxis de subelementos es necesaria cuando el valor de la propiedad es un objeto complejo que no puede ser expresado con una simple cadena, y cuando no se cuente con un Convertidor de Tipo.

Extensiones de marcado

Las extensiones de marcado son clases que permiten al intérprete de XAML evaluar de manera especial sus valores. Hay diversas extensiones de marcado soportadas en el lenguaje XAML con Xamarin.Forms, tal y como las describe la siguiente tabla:

{StaticResource}	Referencia de manera estática recursos de un diccionario
{DynamicResource}	Referencia de manera dinámica recursos de un diccionario
{Binding}	Expresa un enlace de datos
{TemplateBinding}	Permite enlazar el valor de una propiedad de una plantilla a un valor concreto de la clase que la implementa
{x:Null}	Permite asignar como valor un null
{x:Reference}	Permite declarar un enlace entre elementos
{x:Static}	Permite acceder a un campo estático, constante o enumeración
{x:FactoryMethod} y {x:Arguments}	Permiten invocar un método y establecer parámetros

Adicionalmente, podemos crear nuestras propias extensiones de marcado a través del uso de la interfaz IMarkupExtension. Si deseas conocer más acerca de este tema, consulta la documentación de Xamarin o el sitio de MSDN.

Recursos

Los recursos son un diccionario de objetos que podemos declarar para poder reutilizar en uno o varios documentos de XAML dentro de una aplicación. Como vimos en el capítulo anterior, la clase base ancestral VisualElement es la clase base para todos los elementos visuales que se dibujan en pantalla. Es en esta clase donde podemos encontrar la propiedad Resources, la cual es de tipo ResourceDictionary. Por este motivo, todas las clases que heredan de manera directa o indirecta de VisualElement cuentan con un diccionario de recursos. Ahora bien, como estamos hablando de un diccionario, para agregar recursos debemos primero identificarlos con una clave única a través del atributo x:Key.

El siguiente fragmento de código nos muestra la declaración de un recurso de tipo color en el diccionario de recursos de un Grid.

```
<Grid.Resources>
    <ResourceDictionary>
        <Color x:Key="MiColor">Red</Color>
    </ResourceDictionary>
</Grid.Resources>
```

REFERENCIANDO RECURSOS

Para referenciar recursos tenemos dos opciones: referenciarlos de manera estática o de manera dinámica a través del uso de las extensiones de marcado {StaticResource} o {DynamicResource} respectivamente.

La diferencia entre una y otra opción se basa en si queremos obligar a que el recurso exista previamente o no: con {StaticResource} la referencia al recurso se resuelve durante la construcción de los objetos, por lo tanto el recurso en cuestión debe estar declarado en XAML previamente. Adicionalmente, la referencia estática obliga a obtener el valor solo una vez y cualquier cambio que sucediera en el recurso sería ignorado completamente. Con {DynamicResource} el recurso que estamos referenciando no necesariamente debe existir, sino que puede ser creado de manera dinámica, ya que el recurso será evaluado y resuelto cada vez que el elemento necesita el recurso en cuestión.

Por temas de desempeño, se recomienda siempre que sea posible referenciar recursos de manera estática a través de {StaticResource}. El siguiente fragmento de código muestra cómo se referencia el recurso que declaramos anteriormente en la propiedad Color de un BoxView:


```
<ContentPage.Resources>
    <ResourceDictionary>
        <Color x:Key="MiColor">Red</Color>
    </ResourceDictionary>
</ContentPage.Resources>
<Grid>
    <BoxView Color="{StaticResource MiColor}" />
</Grid>
```

ÁMBITOS DE LOS RECURSOS

Debido a que la propiedad Resources está disponible en cualquier elemento visual, el compilador de XAML siempre referenciará al recurso con nivel de visibilidad más cercano. Tal es el caso del siguiente ejemplo de código, en donde hemos declarado un recurso con clave única "MiColor" tanto en el Grid que contiene el BoxView como en el diccionario de recursos de la página. En este escenario, la referencia es hacia el recurso que está en el diccionario del Grid y no en el de ContentPage.

```
<ContentPage.Resources>
    <ResourceDictionary>
        <Color x:Key="MiColor">Red</Color>
    </ResourceDictionary>
</ContentPage.Resources>
<Grid>
    <Grid.Resources>
        <ResourceDictionary>
            <Color x:Key="MiColor">Blue</Color>
        </ResourceDictionary>
    </Grid.Resources>
    <BoxView Color="{StaticResource MiColor}" />
</Grid>
```

RECURSOS VÍA CÓDIGO

Todo lo que declaramos en XAML lo podemos también hacer vía código en C#. En código usamos la propiedad Resources para obtener o agregar objetos al diccionario de recursos. La propiedad Resources es de tipo ResourceDictionary, el cual es un diccionario de tipo <string, object>, por lo tanto su uso es el de cualquier diccionario normal. El siguiente fragmento de código nos muestra cómo a través de C# podemos inspeccionar el diccionario de recursos, así como agregar un nuevo recurso en él:

```
if (Resources.ContainsKey("MiColor"))
{
    var myColor = (Color)Resources["MiColor"];
}
Resources.Add("pi", 3.1416);
```

OBJETOS CLR EN XAML

En XAML podemos instanciar cualquier objeto de .NET. El único requisito es importar el Espacio de nombres donde se encuentre usando la siguiente sintaxis:

xmlns:*alias="clr-namespace:Espacio de nombres;assembly=Ensamblado"*

El ejemplo más básico lo podemos apreciar con el siguiente fragmento de código:

```
namespace PhoneApp1
{
    public class Persona
    {
        public string Nombre { get; set; }
        public string Pais { get; set; }
    }
}
…
<ContentPage xmlns:local="clr-namespace:PhoneApp1" … />
<ResourceDictionary>
        <local:Persona Nombre="Rodrigo"
                       Pais="México"
                       x:Key="persona1" />
</ResourceDictionary>
```


En el lenguaje XAML de Xamarin.Forms no estamos limitados a usar solamente el constructor público sin parámetros de la clase que deseamos instanciar. Para inicializar el objeto, se pueden usar los siguientes mecanismos, los cuales son sumamente útiles sobre todo cuando no tenemos control sobre la clase:

- Constructor predeterminado
- x:FactoryMethod
- x:Arguments

Ejemplo con x:FactoryMethod

Este atributo nos permite ejecutar un método estático cuyo objetivo sea el crear el objeto en cuestión, de ahí su nombre de "Factory". El siguiente fragmento de código nos muestra la declaración de una clase llamada Datos que incluye un método estático llamado GetInstance() y que justamente nos sirve para crear objetos de tipo Datos.

```
public class Datos {
    private static Datos currentInstance;

    public static Datos GetInstance() {
        if (currentInstance == null) {
            currentInstance = new Datos() {};
        }
        return currentInstance;
    }
}
```

El siguiente fragmento de código ejemplifica el uso del atributo x:FactoryMethod, en donde estamos declarando que es el método GetInstance el miembro que deseamos usar para crear el objeto:

```
<ResourceDictionary>
<local:Datos x:Key="datos1"
             x:FactoryMethod="GetInstance" />
</ResourceDictionary>
```

Ejemplo con x:Arguments

El atributo x:Arguments nos permite usar algún constructor con parámetros que tenga la clase que deseamos instanciar en XAML. En el siguiente fragmento de código podemos apreciar la declaración de una clase llamada Persona, que tiene un constructor público que acepta como parámetro el nombre de la persona:

```
public class Persona {
    public Persona(string nombre) { Nombre = nombre };
    …
}
```

Para utilizar ese constructor en el momento de crear el objeto en XAML, usamos el atributo x:Arguments tal y como se demuestra en el siguiente fragmento de código:

```
<ResourceDictionary>
<local:Persona x:Key="persona1">
  <x:Arguments>
    <x:String>
      Rodrigo
    </x:String>
  </x:Arguments>
</local:Persona>
</ResourceDictionary>
```

Convertidores de tipos

En algunos de los anteriores ejemplos de código en este mismo capítulo, veíamos que un color puede ser declarado simplemente con su nombre en inglés, por ejemplo: Blue, Red, Green, etc. ¿Cómo es que el intérprete de XAML sabe qué hacer con esos nombres de colores? Lo mismo pasa cuando estableces el valor de la propiedad Margin de algún elemento visual (que es de tipo Thickness: ¿Cómo sabe qué valor poner para cada lado del elemento? No es que XAML trabaje mágicamente, sino que se basa en los convertidores de tipo, los cuales son objetos que nos permiten convertir una cadena a cualquier otro tipo.

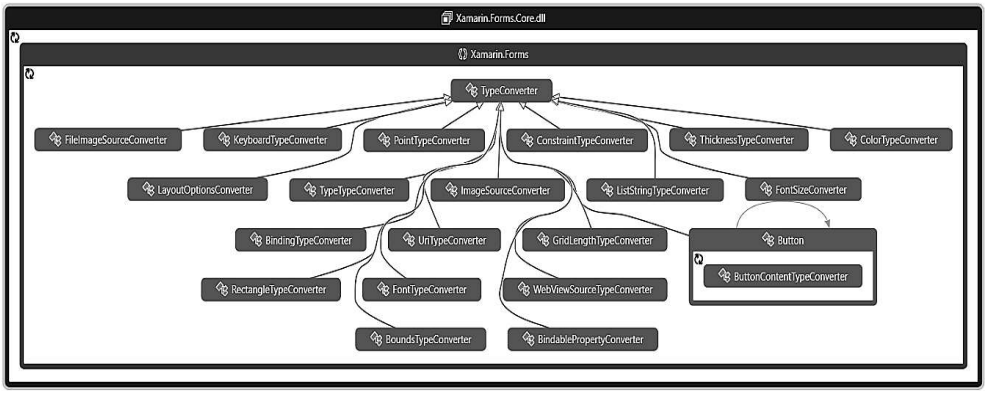
```
<BoxView Color="Red" ... />
```

```
Color.FromRgb(255, 0, 0)
```

```
<BoxView Margin="0, 20, 0, 20" ... />
```

```
new Thickness(0, 20, 0, 20);
```

En pro de la eficiencia y agilidad para escribir código XAML, el API de Xamarin.Forms ya incluye una gran cantidad de convertidores de tipos, tal y como la siguiente figura nos lo demuestra:



CLASE XAMARIN.FORMS.TYPECONVERTER

La clase TypeConverter incluida en el espacio de nombres Xamarin.Forms es la clase base para los convertidores de tipos y que también puedes usar para crear los tuyos propios. La siguiente tabla muestra los métodos virtuales que debes implementar cuando escribes tu propio convertidor:

CanConvertFrom	Determina si se puede convertir la cadena o no
ConvertFromInvariantString	Devuelve el tipo deseado a partir de la cadena especificada en el parámetro

El siguiente fragmento de código muestra un convertidor de tipo llamado PersonaTypeConverter, el cual toma una cadena separada con una coma para poder obtener el nombre y el país, para finalmente devolver un nuevo objeto de tipo

Persona con dichos valores ya establecidos en sus propiedades Nombre y Pais, respectivamente.

```
public class PersonaTypeConverter : TypeConverter
{
    public override bool CanConvertFrom(Type
sourceType)
    {
        return true;
    }
    public override object
ConvertFromInvariantString(string value)
    {
        var personValues = value.Split(',');
        var name = personValues[0];
        var country = personValues[1];
        return new Persona() { Nombre = name, Pais =
country };
    }
}
```

Cabe mencionar que para efectos didácticos, en el anterior código el método CanConvertFrom() simplemente devuelve true de manera fija, pero en ese método sería un lugar adecuado para hacer las validaciones adicionales, para determinar si la conversión puede ser efectuada o no.

ATRIBUTO TYPECONVERTER

Adicionalmente a la clase TypeConverter, en Xamarin.Forms tenemos el atributo TypeConverter (clase Xamarin.Forms.TypeConverterAttribute), el cual nos permite indicar en dónde queremos utilizar un convertidor de tipo. Lo podemos usar a nivel de clase o a nivel también de propiedad. El siguiente fragmento de código muestra el uso del atributo TypeConverter en la clase Persona.

```
[TypeConverter(typeof(PersonaTypeConverter))]
class Persona
{
    public string Nombre { get; set; }
    public string Pais { get; set; }
}
```

El siguiente fragmento de código muestra el uso del atributo TypeConverter en la propiedad Persona dentro de una clase llamada Datos:

```
public class Datos {
    //…
    [TypeConverter(typeof(PersonaTypeConverter))]
    public Persona Persona { get; set; }
}
```

Sea cual sea el lugar de decoración que uses para el atributo TypeConverter, el hacerlo te permite declarar un objeto complejo a través de una simple cadena, tal y como lo demuestra el siguiente fragmento de código XAML:

```
<local:Datos Persona="Rodrigo,México" />
```

XAML compilado

De manera predeterminada, el código XAML de tu aplicación se incluirá como recursos embebidos dentro del ensamblado de la PCL. No obstante, por efectos de un mejor desempeño podemos habilitar la compilación del código XAML, lo cual se logra decorando nuestro ensamblado con el atributo Xamarin.Forms.Xaml.XamlCompilation con el parámetro XamlCompilationOptions.Compile.

El compilar el código XAML mejorará el desempeño de la aplicación, ya que el código XAML no será interpretado cada vez que lo estés usando (por ejemplo, cuando navegas a una página). Es importante mencionar que si bien esto ayudará al desempeño, muy probablemente el tamaño físico del ensamblado PCL se incremente ligeramente.

El siguiente fragmento de código muestra la declaración del atributo XamlCompilation en el archivo AssemblyInfo.cs:

```
[assembly:XamlCompilation(XamlCompilationOptions.Compile)]
```

3 INTERFAZ DE USUARIO

CONTENEDORES

Los contenedores en Xamarin.Forms son aquellos elementos que controlan la renderización de los elementos, en términos de qué tamaño y posición tendrán. Los contenedores los podemos identificar fácilmente ya que son aquellos que heredan de manera directa o indirecta de la clase Layout<T>, tales como:

- StackLayout
- Grid
- AbsoluteLayout
- RelativeLayout

En este capítulo nos concentraremos en los dos primeros: StackLayout y Grid.

StackLayout

Me atrevo a decir que el StackLayout es el contenedor más utilizado en las aplicaciones con Xamarin.Forms, ya que su naturaleza es muy simple y clara: apilar de manera vertical u horizontal sus elementos hijos. Este contenedor es útil en escenarios donde necesitemos alinear controles de una manera rápida y sencilla. El siguiente fragmento de código muestra la declaración de un StackLayout que tiene tres elementos de tipo BoxView dentro de él.

```
<StackLayout>
  <BoxView Color="Red"
           HorizontalOptions="Center" />
  <BoxView Color="Green"
```

```
            HorizontalOptions="Center" />
    <BoxView Color="Blue"
            HorizontalOptions="Center" />
</StackLayout>
```

El resultado visual será el siguiente:

Adicionalmente, si queremos que los elementos se apilen de manera horizontal, simplemente modificamos el valor de la propiedad Orientation:

```
<StackLayout Orientation="Horizontal"> …
```

Y obtendríamos el siguiente resultado:

Grid

El Grid es similar a una tabla y es el más flexible de todos los contenedores, ya que gracias a sus diferentes opciones de tamaño para las columnas y filas, nos permite declarar Interfaces de Usuario que se adapten al tamaño de la pantalla y factor de resolución que tenga el dispositivo. La clase Grid expone las siguientes Propiedades Adjuntas (para mayor información acerca de las Propiedades Adjuntas puedes consultar el capítulo 1 “Introducción a Xamarin”):

- Column
- Row
- ColumnSpan
- RowSpan

El Grid inherentemente tiene una sola celda, pero a través de las propiedades ColumnDefinitions y RowDefinitions podemos definir columnas y filas adicionales. El siguiente fragmento de código muestra la declaración de un Grid que tiene dos columnas y dos filas, es decir, el resultado será un Grid de 2x2:

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
    </Grid.RowDefinitions>
</Grid>
```

Una vez declaradas las columnas y filas que deseamos tener, posicionamos sus elementos hijos a través de las Propiedades Adjuntas que expone la clase Grid. El siguiente fragmento de código muestra el mismo Grid que declaramos anteriormente, pero ahora con cuatro elementos de tipo BoxView ubicados cada uno en una celda diferente.

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>

  <Grid.RowDefinitions>
    <RowDefinition />
    <RowDefinition />
```

```
    </Grid.RowDefinitions>

    <BoxView Color="Red" />
    <BoxView Color="Lime" Grid.Column="1" />
    <BoxView Color="Blue" Grid.Row="1" />
    <BoxView Color="Yellow" Grid.Row="1"
             Grid.Column="1" />
</Grid>
```

Nota: ya que las Propiedades Adjuntas son de tipo entero, y el entero es un Tipo por Valor, implícitamente tendrán un valor de 0, de ahí que no es necesario declarar dicho valor explícitamente.

TIPOS DE TAMAÑOS

Los anchos y altos de las columnas y filas pueden ser de 3 tipos:

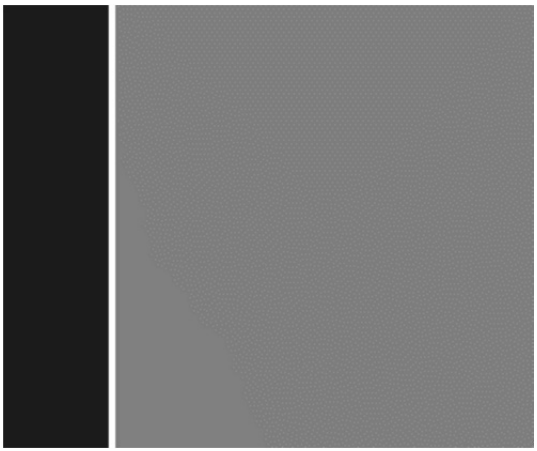
- Fijo
- Automático
- Proporcional (*)

Tamaño fijo

Este tipo de tamaño es un valor absoluto que deseamos que tenga el ancho de la columna o el alto de la fila. El siguiente fragmento de código nos muestra la declaración de un Grid que tiene dos columnas, y la primera columna tiene un ancho fijo de 100. Debido a su naturaleza, si el contenido de una columna con tamaño fijo es más grande que este, el contenido se cortará.

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="100" />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>

   <BoxView Color="Blue" />
</Grid>
```

La siguiente figura muestra el resultado del anterior código:

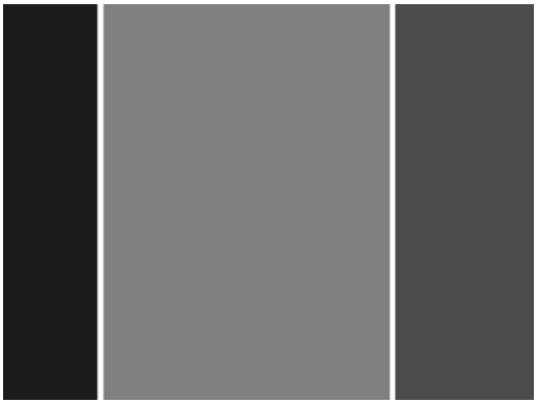


Tamaño automático

Este tipo de tamaño indica que la columna o la fila deben crecer en función de su contenido. Para declarar este tipo de tamaño utilizamos el valor “Auto”. El siguiente fragmento de código nos muestra un Grid con tres columnas en donde la segunda columna tiene el tamaño automático.

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="100" />
    <ColumnDefinition Width="Auto" />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>

   <BoxView Color="Blue" />
   <BoxView Color="Gray"
    Grid.Column="1"
            WidthRequest="300" />
   <BoxView Color="Green" Grid.Column="2" />
</Grid>
```

La siguiente figura muestra el resultado del anterior código:

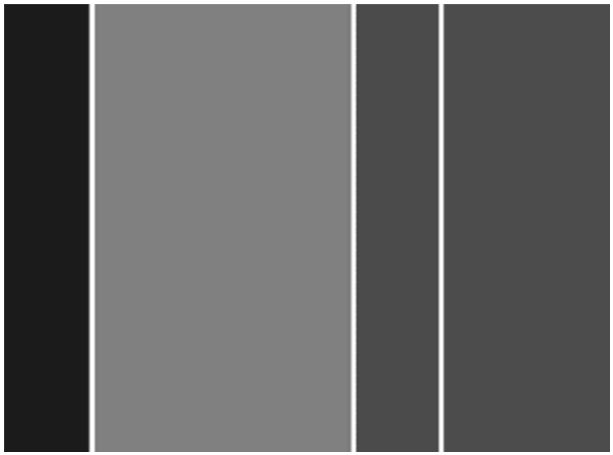


Tamaño proporcional

Este tipo de tamaño es el más flexible ya que es un factor de proporción del espacio restante, después de haber dibujado en pantalla las columnas y filas con tamaño fijo y las de tamaño automático. Este factor de proporción es expresado con un número y el símbolo *. Este es el tamaño predeterminado que utiliza XAML para las columnas y filas. El siguiente fragmento de código nos muestra un Grid que tiene cuatro columnas, donde la tercera y cuarta columnas tienen el tamaño proporcional. Al tener el valor 2*, la cuarta columna indica que será dos veces más ancha que la tercera. También es importante destacar que la tercera columna, al no tener la propiedad Width, implícitamente tendrá un valor de 1*.

```
  <Grid>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="100" />
      <ColumnDefinition Width="Auto" />
      <ColumnDefinition />
      <ColumnDefinition Width="2*" />
    </Grid.ColumnDefinitions>

     <BoxView Color="Blue" />
     <BoxView Color="Gray" Grid.Column="1"
WidthRequest="300" />
     <BoxView Color="Green" Grid.Column="2" />
     <BoxView Color="Red" Grid.Column="3" />
  </Grid>
```

El resultado del anterior código será la siguiente figura:



CONTROLES

El API de Xamarin.Forms cuenta con alrededor de 20 controles que podemos utilizar para construir las Interfaces de Usuario de nuestras aplicaciones. En la siguiente tabla, se enlistan todos los controles de Xamarin.Forms en orden alfabético incluyendo cuál es el Renderer que utiliza cada uno y cuál es el control nativo que finalmente mostrará cada sistema operativo concreto.

Clase	Renderer responsable	iOS	Android	UWP
ActivityIndicator	ActivityIndicatorRenderer	UIActivityIndicator	ProgressBar	ProgressBar
BoxView	BoxRenderer (iOS y Android) BoxViewRenderer en UWP	UIView	ViewGroup	Rectangle
Button	ButtonRenderer	UIButton	Button	Button
CarouselView	CarouselViewRenderer	UIScrollView	RecyclerView	FlipView
DatePicker	DatePickerRenderer	UITextField	EditText	DatePicker
Editor	EditorRenderer	UITextView	EditText	TextBox
Entry	EntryRenderer	UITextField	EditText	TextBox

Image	ImageRenderer	UIImageView	ImageView	Image
Label	LabelRenderer	UILabel	TextView	TextBlock
ListView	ListViewRenderer	UITableView	ListView	ListView
Map	MapRenderer	MKMapView	MapView	MapControl
Picker	PickerRenderer	UITextField	EditText	ComboBox
ProgressBar	ProgressBarRenderer	UIProgressView	ProgressBar	ProgressBar
SearchBar	SearchBarRenderer	UISearchBar	SearchView	AutoSuggestBox
Slider	SliderRenderer	UISlider	SeekBar	Slider
Stepper	StepperRenderer	UIStepper	LinearLayout	Control
Switch	SwitchRenderer	UISwitch	Switch	ToggleSwitch
TableView	TableViewRenderer	UITableView	ListView	ListView
TimePicker	TimePickerRenderer	UITextField	EditText	TimePicker
WebView	WebViewRenderer	UIWebView	WebView	WebView

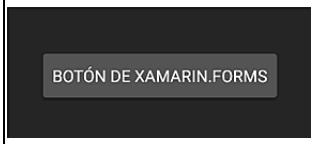
CONTROLES COMUNES

Button

Control que renderiza un botón. Incluye el evento Clicked que se dispara cuando el botón es tocado o pulsado. Además, incluye la propiedad Command que soporta el enlace a un objeto de tipo ICommand. Para mayor información acerca de la interfaz ICommand consulta el capítulo 6 “Comandos”.

El siguiente ejemplo de código muestra la declaración de un botón y el manejo de su evento Clicked:

```
<Button Text="Botón" Clicked="OnClicked" />
```

Android	UWP
	

Stepper

Control para seleccionar el valor de un rango predefinido. El rango se establece con las propiedades Minimum y Maximum. Adicionalmente, podemos establecer un valor entero o decimal para el incremento.

El siguiente ejemplo código muestra la declaración de un Stepper, en donde hemos establecido el rango y el valor de incremento:

```
<Stepper Minimum="0" Maximum="100" Increment="0.5" />
```

Switch

Control para seleccionar un valor verdadero o falso. El valor se establece u obtiene con la propiedad IsToggled.

El siguiente fragmento de código muestra la declaración de un Switch activado:

```
<Switch IsToggled="True" />
```

Slider

Control para seleccionar el valor de un rango predefinido. El rango se establece con las propiedades Minimum y Maximum. El siguiente fragmento código muestra la declaración de un Slider cuyo rango va de 0 a 100:

```
<Slider Minimum="0" Maximum="100" />
```

DatePicker

Control para seleccionar una fecha. El siguiente fragmento de código muestra la declaración de un DatePicker:

```
<DatePicker />
```

TimePicker

Control para seleccionar una hora, minuto y segundo. El siguiente fragmento de código muestra la declaración de un TimePicker:

```
<TimePicker />
```

ProgressBar

Control para mostrar una barra de progreso porcentual. La propiedad Progress indica el valor de progreso, el cual va de 0 a 1. El siguiente fragmento de código muestra la declaración de un ProgressBar cuyo progreso es de 80%:

```
<ProgressBar Progress="0.8" />
```

ActivityIndicator

Control para mostrar un indicador de actividad. Este control es sumamente útil cuando queremos notificar al usuario que la aplicación está ocupada en alguna tarea o proceso. La propiedad IsRunning establece si el indicador está activo o no. El siguiente fragmento de código muestra la declaración de un ActivityIndicator que está ejecutando:

```
<ActivityIndicator IsRunning="True" />
```

Image

Control para mostrar una imagen. La propiedad Source indica el URI de la imagen que deseamos mostrar, pudiendo ser un URI relativo o absoluto. Asimismo, a través de la sintaxis de subelementos podemos declarar una fuente basada en un archivo embebido en el ensamblado. El siguiente fragmento de código muestra la declaración de un Image, cuya fuente es un archivo de imagen expuesta a través de Internet:

```
<Image Source="http://pagina.com/imagen.jpg" />
```

BoxView

Elemento que nos permite dibujar un rectángulo en pantalla. Podemos usar la propiedad Color para establecer el color de relleno del rectángulo, y las propiedades WidthRequest y HeightRequest para establecer el ancho y alto, respectivamente. El siguiente fragmento de código muestra la declaración de un BoxView azul de 200 × 200.

```
  <BoxView Color="Blue" WidthRequest="200"
HeightRequest="200" />
```

Frame

Elemento que dibuja un borde alrededor de un contenido. El siguiente fragmento de código muestra la declaración de BoxView que tiene un borde de color rojo sombreado:

```
<Frame OutlineColor="Red" HasShadow="True">
      <BoxView Color="Blue"
               WidthRequest="200"
               HeightRequest="200" />
</Frame>
```

WebView

Control que presenta contenido HTML. La propiedad Source indica el URI del contenido HTML que deseamos desplegar. El siguiente fragmento de código muestra la declaración de un WebView:

```
<WebView Source="http://rdiazconcha.com" />
```

CONTROLES DE TEXTO

Label

Control para mostrar un texto de solo lectura. Soporta cambiar el tamaño de la fuente, el tipo de fuente, las decoraciones, alineaciones, etc. El siguiente fragmento de código muestra la declaración de un Label, al cual le estamos cambiando el tamaño, la familia de la fuente y el color del texto:

```
<Label Text="Aplicaciones de Negocio con Xamarin.Forms"
       FontSize="44"
       FontFamily="Segoe UI Light"
       TextColor="Red" />
```

Entry

Control para capturar una línea de texto. Soporta múltiples tipos de teclado. La siguiente tabla enlista las propiedades más destacables de este control:

IsPassword	Protege el texto capturado. Ideal para contraseñas
Placeholder	Texto base que mostrará el control
PlaceholderColor	Color del texto base

El siguiente fragmento de código muestra la declaración de un Entry:

```
<Entry Text="..." />
```

Editor

Control que permite capturar múltiples líneas de código. Soporta múltiples tipos de teclado. El siguiente fragmento de código nos muestra la declaración de un Editor:

```
<Editor Text="..." />
```

SearchBar

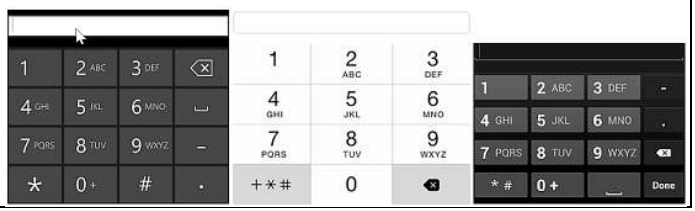
Control que presenta una caja de búsqueda. El siguiente fragmento de código muestra la declaración de SearchBar:

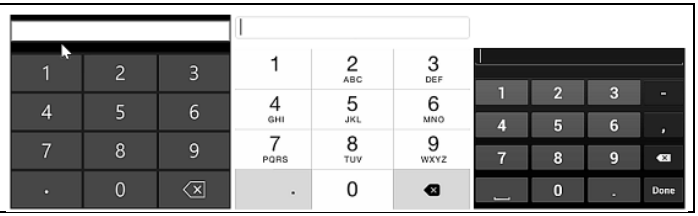
```
<SearchBar Text="Buscar" />
```

Tipos de teclado

Los controles Entry y Editor soportan diferentes formatos de teclado. Esto es muy útil para limitar los caracteres que deseamos mostrar en el teclado del dispositivo donde esté ejecutando la aplicación, para hacer más fácil la captura del texto. Por ejemplo, cuando deseamos teclear un número de teléfono, será más lógico mostrar únicamente números en vez de todo el teclado alfanumérico.

La siguiente tabla enlista los diferentes tipos de teclado que soporta Xamarin.Forms.

Tipo	Ejemplo en UWP, iOS y Android
Default	
Text	
Chat	
Url	
Email	
Telephone	

Numeric	

CONTROLES DE LISTA

Picker

Control que nos permite seleccionar un elemento de una lista. De manera predeterminada está colapsada y a simple vista parece un Entry.

El siguiente fragmento de código muestra la declaración de un Picker que tiene tres elementos:

```
<Picker>
  <Picker.Items>
    <x:String>Uno</x:String>
    <x:String>Dos</x:String>
    <x:String>Tres</x:String>
  </Picker.Items>
</Picker>
```

Nota: A partir de la versión 2.3.4 de Xamarin.Forms el control Picker incluye la propiedad ItemsSource, en donde podremos establecer una fuente de datos a través de un enlace.

ListView

Control que presenta una colección de elementos y permite seleccionar uno. Este control es enlazable a una fuente de datos a través de su propiedad ItemsSource. Adicionalmente, los elementos del control son personalizables a través de Plantillas de Datos. Para mayor información acerca del enlace de datos, contexto de enlace y de las plantillas de datos consulta el capítulo 5 “Enlace de Datos”.

El siguiente fragmento de código muestra un desvío que está siendo enlazado a su contexto de enlace actual.

```
<ListView ItemsSource="{Binding}" />
```

ESTILOS

Los estilos en Xamarin.Forms son similares a los Estilos de cascada (CSS) en las tecnologías Web, en el sentido de que son un grupo de valores que pueden ser aplicados a diversas propiedades en uno o varios elementos visuales.

Los estilos son objetos de tipo Xamarin.Forms.Style, y si bien los puedes declarar vía código, lo más común es declararlos en los Diccionarios de Recursos dentro de un documento de XAML. La clase Style incluye la propiedad Setters, la cual es una colección de objetos de tipo Setter cuya finalidad es indicar el nombre de la propiedad y el valor que deseamos que tenga dicha propiedad después de haber aplicado el Estilo en cuestión. Adicionalmente, la clase Style está decorada con el atributo [ContentProperty("Setters")], por lo que podemos declarar los Setter en un Style a través de la sintaxis de contenido. Para mayor información acerca de la sintaxis de contenido, puedes consultar el capítulo 2 "El lenguaje XAML".

Otra propiedad de suma importancia en la clase Style es TargetType, la cual indica el tipo al que el estilo puede aplicarse.

El siguiente fragmento de código muestra la declaración básica de un estilo. A este estilo, le estamos asignando una clave única llamada "EstiloBotonBase", el tipo al que lo podemos aplicar es Button y además incluye dos objetos Setter que cambian las propiedades WidthRequest y TextColor a 200 y "Yellow" respectivamente:

```
<Style x:Key="EstiloBotonBase" TargetType="Button">
    <Setter Property="WidthRequest" Value="200" />
    <Setter Property="TextColor" Value="Yellow" />
</Style>
```

Usando los estilos

La clase base VisualElement implementa la propiedad pública Style, por lo que todos los elementos visuales y controles en Xamarin.Forms pueden aplicar un estilo. Esto lo logramos referenciando el estilo a través del identificador que le hayamos asignado dentro del Diccionario de Recursos.

Por ejemplo, el siguiente fragmento de código nos muestra cómo un botón está aplicando el estilo llamado EstiloBotonBase, a través de la referenciación estática del mismo usando la extensión de marcado {StaticResource}:

```
<Button Style="{StaticResource EstiloBotonBase}" />
```

Propiedad BasedOn

La clase Style también incluye la propiedad BasedOn, la cual nos permite crear Estilos que estén basados en otros previamente declarados, es decir, estilos con un comportamiento en cascada tal y como sucede en las tecnologías Web.

El siguiente fragmento de código muestra la declaración de dos estilos, en donde el primero llamado "EstiloBotonBase" es la base del segundo, llamado "EstiloBoton". Nota que en "EstiloBoton" estamos usando la propiedad BasedOn para referenciar de manera estática el primero.

```
<Style x:Key="EstiloBotonBase" TargetType="Button">
…
</Style>
<Style x:Key="EstiloBoton"
       BasedOn="{StaticResource EstiloBotonBase}"
       TargetType="Button">
   <Setter Property="TextColor"
           Value="Blue" />
   <Setter Property="FontSize"
                Value="20" />
</Style>
```

Estilos implícitos

Cuando tenemos diversos controles en una página que están utilizando el mismo estilo llega a ser ineficiente que en cada uno de ellos estemos referenciando una y otra vez el estilo de manera estática o de manera dinámica. Tomemos como ejemplo el siguiente código:

```
<StackLayout VerticalOptions="Center"
             HorizontalOptions="Center">
  <StackLayout.Resources>
    <ResourceDictionary>
      <Style x:Key="EstiloBoton" TargetType="Button">
        <Setter Property="FontSize" Value="22" />
        <Setter Property="Margin" Value="10" />
```



```
        </Style>
      </ResourceDictionary>
    </StackLayout.Resources>
    <Button Text="Botón 1"
            Style="{StaticResource EstiloBoton}" />
    <Button Text="Botón 2"
            Style="{StaticResource EstiloBoton}" />
    <Button Text="Botón 3"
            Style="{StaticResource EstiloBoton}" />
</StackLayout>
```

En cada botón estamos repitiendo la misma referencia una y otra vez. Por esta razón, en Xamarin.Forms contamos con los estilos implícitos, los cuales tienen la característica de no utilizar el atributo x:Key, sino simplemente usar el tipo indicado en el atributo TargetType. Cuando tenemos un estilo implícito, todos los elementos visuales del tipo indicado en TargetType se estilizarán automáticamente, incluso si son elementos creados de manera dinámica. Por lo tanto, ya no es necesario referenciar el Estilo en la propiedad Style, tal y como lo muestra el siguiente fragmento de código:

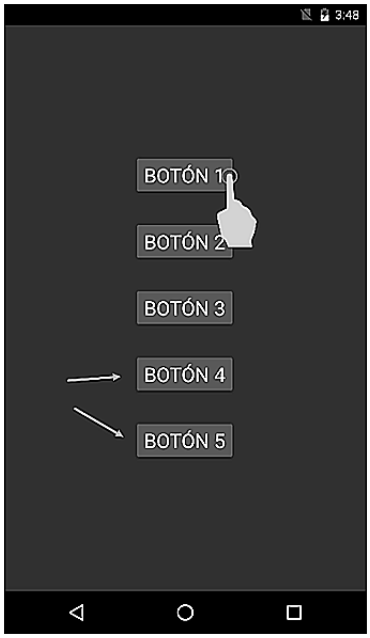
```
<StackLayout x:Name="panel"
VerticalOptions="Center"
             HorizontalOptions="Center">
  <StackLayout.Resources>
    <ResourceDictionary>
      <Style TargetType="Button">
        <Setter Property="FontSize" Value="22" />
        <Setter Property="Margin" Value="10" />
      </Style>
    </ResourceDictionary>
  </StackLayout.Resources>
  <Button Text="Botón 1" />
  <Button Text="Botón 2" />
```

```
    <Button Text="Botón 3" />
</StackLayout>
```

La siguiente figura muestra el resultado de este código en Android:

Para demostrar que un estilo implícito también se aplicaría a los elementos creados de manera dinámica, manejemos el evento Clicked del primer botón, simplemente escribiendo el manejador en XAML y permitiendo que el Intellisense de Visual Studio .NET nos ayude a crear el cuerpo del manejador del evento en el archivo de code-behind. En este método, agregaremos a la colección Children del elemento StackLayout (identificado con el nombre "panel") un nuevo botón cada vez que el botón sea pulsado. Adicionalmente, asignaremos el texto "Botón X" cada vez, siendo "X" el contador de botones totales que tiene el contenedor:

```
  private void Button_OnClicked(object sender, EventArgs
e)
  {
      panel.Children.Add(new Button() { Text = $"Botón
{panel.Children.Count + 1}" });
  }
```



Propiedad ApplyToDerivedTypes

Otra característica importante de los estilos, es que con la propiedad ApplyToDerivedTypes de la clase Style, podemos indicar que un estilo aplique incluso en tipos derivados del especificado en TargetType. Por ejemplo, supongamos que tenemos el siguiente control derivado de Label:

```
public class MyLabel : Label
{
   …
}
```

Si queremos crear un estilo que aplique tanto para los objetos de tipo Label como para los de tipo MyLabel, entonces debemos establecer la propiedad ApplyToDerivedTypes, tal y como lo muestra el siguiente fragmento de código:

```
<Style TargetType="Label" ApplyToDerivedTypes="True">
  <Setter Property="FontFamily" Value="Impact" />
  <Setter Property="FontSize" Value="20" />
  <Setter Property="TextColor" Value="Yellow" />
  <Setter Property="BackgroundColor" Value="Black" />
</Style>
```

En nuestra página, simplemente instanciamos MyLabel como cualquier otro control (recuerda primero importar el xmlns):

```
<Label Text="Texto 1" />
<Label Text="Texto 2" />
<local:MyLabel Text="Texto 3" />
```

Al ejecutar la aplicación, podremos darnos cuenta de que los tres controles fueron estilizados por el mismo objeto Style:

Triggers en estilos

Hasta el momento, hemos visto cómo los estilos nos ayudan cambiar la apariencia de nuestras aplicaciones de una manera muy rápida, sencilla y sobre todo que nos ayudan a que el mantenimiento del código sea más fácil. Adicionalmente, los estilos en Xamarin.Forms pueden tener Triggers o "disparadores", los cuales nos permiten expresar acciones que modifiquen la apariencia de los controles según alguna lógica que decidamos. Para definir estos disparadores, lo hacemos a través de la propiedad Triggers de la clase Style.

Xamarin.Forms incluye las siguientes clases de Triggers:

Trigger	Nos permite cambiar el valor de alguna propiedad, basándose en el valor de otra
DataTrigger	Nos permite cambiar el valor de alguna propiedad, basándose en el valor de alguna propiedad en algún otro control
EventTrigger	Nos permite cambiar el valor de alguna propiedad, basándose en un evento
MultiTrigger	Con este Trigger, podemos definir un grupo de Triggers que se tienen que evaluar como un conjunto para poder cambiar el valor de una propiedad

En este capítulo nos concentraremos en los dos primeros.

TRIGGER

El siguiente fragmento de código muestra la definición de un estilo para los controles de tipo Entry llamado “MyEntryStyle”. A través de la sintaxis de subelementos de XAML, se agrega a la colección de Triggers un Trigger que evalúa la propiedad IsFocused del Entry. Si el valor de dicha propiedad es verdadero, el color del texto del Entry cambiará a color rojo y el tamaño de la fuente cambiará a 30:

```
<Style x:Key="MyEntryStyle" TargetType="Entry">
  <Style.Triggers>
    <Trigger TargetType="Entry" Property="IsFocused"
Value="True">
      <Setter Property="TextColor" Value="Red" />
      <Setter Property="FontSize" Value="30" />
    </Trigger>
  </Style.Triggers>
</Style>
```

Suponiendo que hemos aplicado el anterior estilo a tres elementos Entry dentro de un StackLayout vertical, el resultado de ejecutar la aplicación en Android será el siguiente:

DATATRIGGER

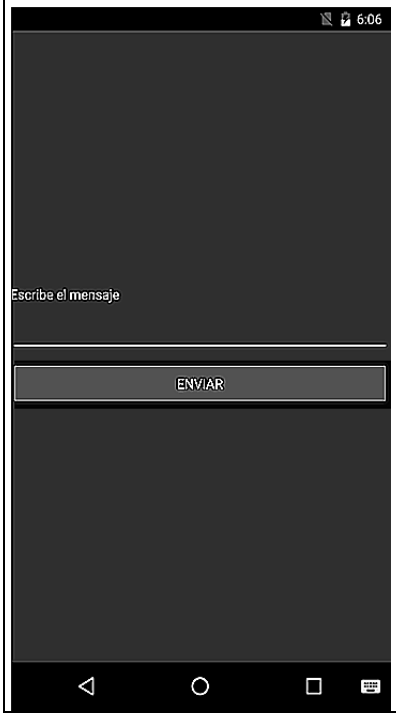
Como lo explicamos anteriormente, el DataTrigger nos permite cambiar el valor de una propiedad en función del valor de una propiedad que está en otro objeto diferente. Esto es muy útil cuando necesitamos modificar de manera automática el estado de un control que está en función de otro, sin necesidad de escribir una línea de código en el code-behind o en el objeto ViewModel relacionado con la página. Por ejemplo, supongamos que tenemos una página que contiene el siguiente código como Interfaz de Usuario para mandar un mensaje:

```
<StackLayout x:Name="panel"
                    VerticalOptions="Center"
                    HorizontalOptions="FillAndExpand">
    <Label Text="Escribe el mensaje" />
    <Entry x:Name="mensaje" Text="" />
    <Button Text="Enviar" />
</StackLayout>
```

Tiene bastante sentido que el botón esté deshabilitado, a menos que el usuario verdaderamente escriba algo en la caja de texto. Este requerimiento lo podemos implementar por medio de un DataTrigger, como el siguiente fragmento de código nos muestra:

```
<Style TargetType="Button">
    <Style.Triggers>
        <DataTrigger
    TargetType="Button"
    Binding="{Binding Text.Length, Source={x:Reference
mensaje}}"      Value="0">
                <Setter Property="IsEnabled"
Value="False" />
        </DataTrigger>
    </Style.Triggers>
</Style>
```

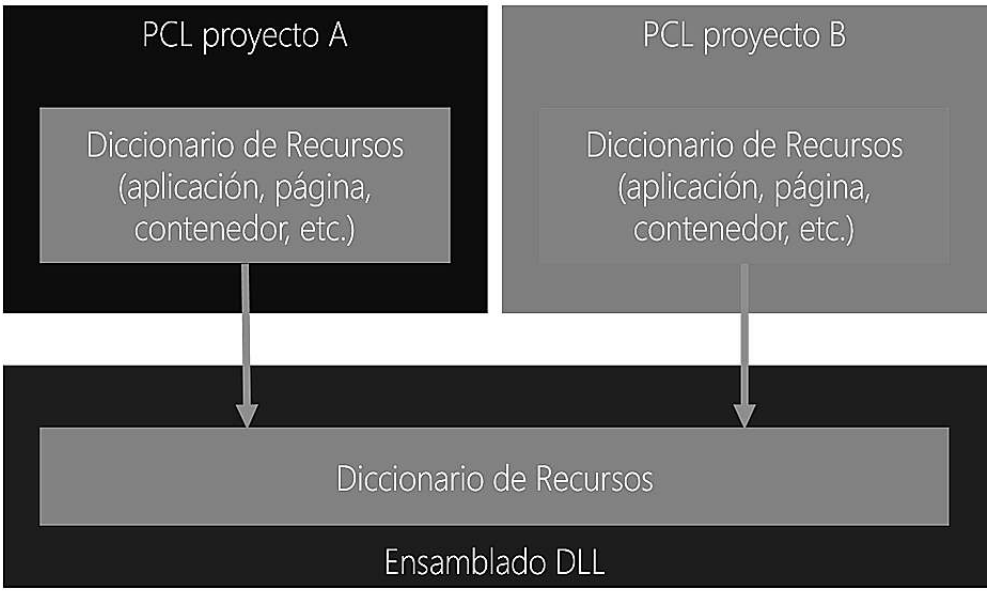
Las siguientes figuras nos muestran la aplicación ejecutándose en Android:

Sin mensaje en la caja de texto	Con mensaje en la caja de texto
	

Diccionarios mezclados

Previamente comentamos que el nivel más global de los Diccionarios de Recursos dentro de una aplicación es la propiedad Resources en la clase Application. No obstante, también podemos compartir recursos entre diferentes proyectos si implementamos un Diccionario de Recursos en un ensamblado de tipo DLL.

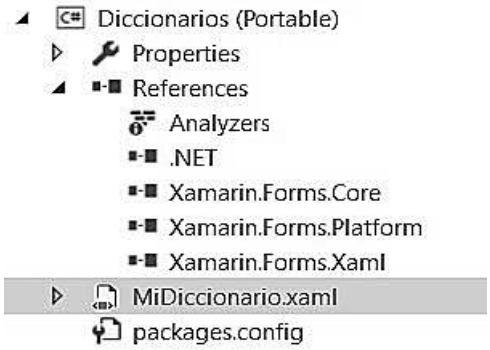
El siguiente diagrama nos muestra la arquitectura de dos Bibliotecas de Clases Portables que están utilizando un mismo Diccionario de Recursos expuesto en un ensamblado DLL externo:



CREANDO UN DICCIONARIO DE RECURSOS EXTERNO

El tipo de proyecto del ensamblado con el Diccionario de Recursos que deseamos reutilizar debe ser de tipo Biblioteca de Clases Portable, apuntando a las mismas plataformas que las PCL que lo van a referenciar (de lo contrario, te mandará un error Visual Studio .NET al tratar de referenciarlo en la PCL de tu aplicación de Xamarin.Forms). Una vez creado el proyecto, debemos agregar la referencia a los ensamblados de Xamarin.Forms a través de NuGet (o usando la plantilla de Biblioteca de Clases Portable para Xamarin.Forms). Posteriormente, agregamos al proyecto un archivo XAML y su archivo de code-behind relacionado. Para mayor facilidad, puedes usar la plantilla de "Forms Xaml Page", sustituyendo el elemento ContentPage por ResourceDictionary, tanto en el archivo de código XAML como en el de C#.

La siguiente figura muestra un proyecto de tipo PCL llamado "Diccionarios", en donde hemos referenciado los ensamblados de Xamarin.Forms y adicionalmente hemos agregado un archivo llamado MiDiccionario.xaml junto con su archivo de code-behind:



El siguiente fragmento de código XAML muestra la declaración del Diccionario de Recursos que incluye un Color llamado “ColorLibro”:

```
<?xml version="1.0" encoding="utf-8" ?>
<ResourceDictionary
xmlns="http://xamarin.com/schemas/2014/forms"

xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
              x:Class="Diccionarios.MiDiccionario">
    <!--Recursos que deseamos reutilizar en diferentes
proyectos-->
    <Color x:Key="ColorLibro">#AFAFAF</Color>
</ResourceDictionary>
```

Del lado del código de C#, tenemos que cambiar el tipo del que hereda la clase a ResourceDictionary. Es muy importante que dejes la invocación al método InitializeComponent(), ya que ese método justamente es el que inicializará los elementos del documento de XAML:

```
using Xamarin.Forms;

namespace Diccionarios
{
    public partial class MiDiccionario :
ResourceDictionary
    {
        public MiDiccionario()
        {
```

```
            InitializeComponent();
        }
    }
}
```

USANDO EL DICCIONARIO DE RECURSOS EXTERNO

Una vez creado el ensamblado con el Diccionario de Recursos externo, lo debemos referenciar en cada proyecto PCL de nuestra aplicación de Xamarin.Forms. Luego, en el documento de XAML en donde desees usar el Diccionario de Recursos externo, deberás importar el espacio de nombres, tal y como lo muestra el siguiente ejemplo:

```
xmlns:dic="clr-
namespace:Diccionarios;assembly=Diccionarios"
```

Finalmente, declaramos la propiedad MergedWith en el elemento ResourceDictionary, indicando que queremos “mezclar” el Diccionario de Recursos local con el externo.

En el siguiente ejemplo de código se muestra la utilización del Diccionario de Recursos externo, además podrás observar cómo el estilo implícito para los BoxView usa el recurso de Color llamado “ColorLibro”:

```
<ResourceDictionary MergedWith="dic:MiDiccionario">
  <Style TargetType="BoxView">
    <Setter Property="Color"
            Value="{DynamicResource ColorLibro}" />
   </Style>
</ResourceDictionary>
```

4 NAVEGACIÓN Y MENSAJERÍA

NAVEGACIÓN

En una aplicación móvil construida con Xamarin.Forms es muy difícil pensar que la aplicación solamente tendrá una sola página. En la vida real, una aplicación sofisticada tiene diversas páginas, por lo tanto necesitamos un mecanismo para poder navegar entre ellas. Xamarin.Forms incluye una infraestructura de navegación que está expuesta a través de la propiedad Navigation, la cual es de tipo INavigation, en la clase base VisualElement. Esta interfaz incluye algunos miembros que nos permiten controlar la navegación dentro de una aplicación. La siguiente tabla muestra los miembros más destacables de la interfaz INavigation:

PushAsync()	Agrega la página especificada en el parámetro a la pila de navegación. La ejecución de este método agrega un registro en NavigationStack
PopAsync()	Remueve la última página de la pila de navegación. La ejecución de este método elimina el último registro de NavigationStack
PushModalAsync()	Navega de manera modal a la página especificada en el parámetro. La ejecución de este método agrega un registro en ModalStack
PopModalAsync()	Remueve la página modal. La ejecución de este método elimina el último registro de ModalStack
NavigationStack	Propiedad que representa la pila de navegación
ModalStack	Propiedad que representa la pila de navegación modal

RemovePage()	Remueve de la pila de navegación la página especificada en el parámetro
InsertPageBefore()	Intercala en la pila la página especificada en el parámetro
PopToRootAsync()	Remueve todas las páginas de la pila, navegando a la página raíz

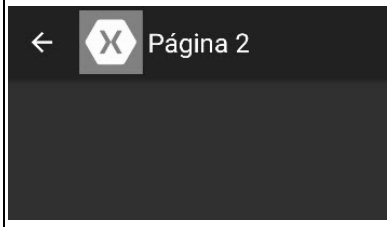
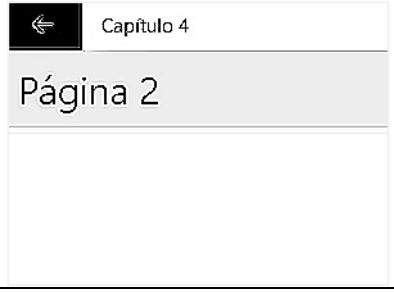
Xamarin.Forms cuenta con dos tipos de navegación: navegación jerárquica y navegación modal.

Navegación jerárquica

Este tipo de navegación nos permite navegar entre diferentes páginas de ida y de vuelta. Por ejemplo, si tienes tres páginas: A, B y C, probablemente querrás navegar de la página A hacia la página B. Asimismo, podrías navegar de la página B hacia la página C o puedes ir de la página B de regreso hacia la página A, pulsando en el botón de "regresar" en pantalla o usando el botón físico del dispositivo. La funcionalidad de navegación jerárquica se basa en cómo trabajan las pilas, ya que es un mecanismo de navegación de tipo LIFO (Last In, First Out por sus siglas en inglés).

Para poder usar la navegación jerárquica requerimos que la aplicación establezca como página raíz una página de tipo NavigationPage. El constructor de la clase NavigationPage solicita como argumento un objeto de tipo Page, por lo que podemos pasar como parámetro cualquier objeto de tipo Page, aunque comúnmente será de tipo ContentPage. Con este método de navegación, la aplicación mostrará una barra de navegación que incluye el botón para navegar de regreso, el título de la página que se haya establecido a través de la propiedad Title y opcionalmente una barra de herramientas con botones, la cual se establece a través de la colección ToolbarItems.

Las siguientes figuras muestran una aplicación que ha navegado a una segunda página:

Android	UWP
	

También es importante mencionar que cada vez que navegamos hacia una página o desde una página, los métodos OnAppearing() y OnDisappearing() de la página serán invocados automáticamente por la infraestructura de navegación. No obstante, como veremos a continuación, el momento en el que estos métodos son invocados varía según la plataforma, y por lo tanto, no se deberá basar ningún tipo de lógica en el orden de estos métodos.

El siguiente fragmento de código establece como página raíz un objeto de tipo NavigationPage a través de la propiedad MainPage, y además indica que la primera página en la pila de navegación es Page1:

```
public class App : Application
{
    public App()
    {
        // Indica que la página raíz será Page1
        MainPage = new NavigationPage(new Page1());
    }
   …
}
```

PUSHASYNC Y POPASYNC

La interfaz INavigationPage incluye los métodos asíncronos PushAsync() y PopAsync() para navegar a una página o regresar de ella, respectivamente. Esto es, PushAsync() agregará a la pila de navegación un nuevo objeto de tipo Page y PopAsync() quitará el último objeto de tipo Page agregado a la pila de navegación.

El siguiente fragmento de código muestra la navegación a una página llamada Page2:

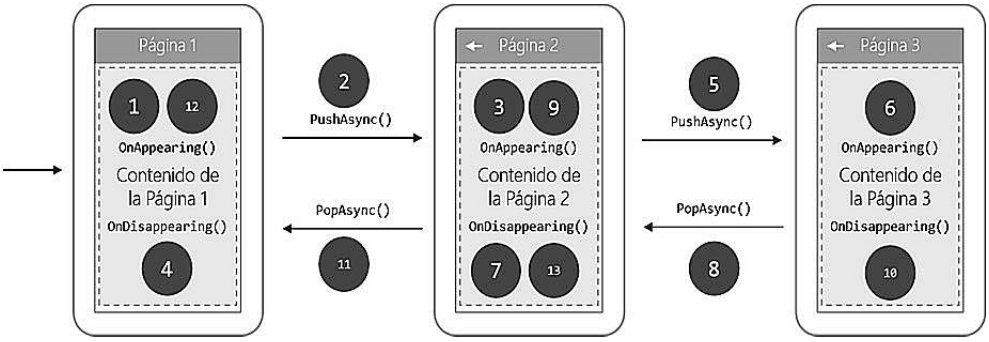
```
await Navigation.PushAsync(new Page2());
```

El siguiente fragmento de código regresa programáticamente de una página por medio de la invocación del método PopAsync():

```
await Navigation.PopAsync();
```

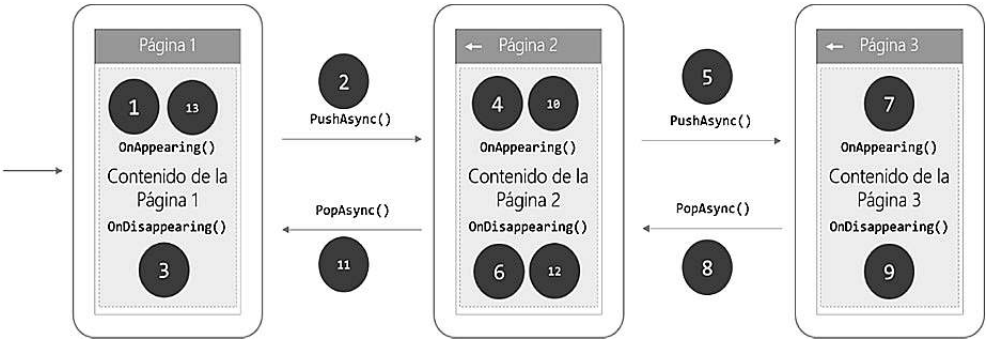
Las siguientes figuras muestran los diferentes momentos en que los métodos OnAppearing() y OnDisappearing() son invocados al estar navegando entre tres diferentes páginas, según el sistema operativo donde esté ejecutando la aplicación de Xamarin.Forms.

Android



Como se puede apreciar en la anterior imagen, el método OnAppearing se invocará antes que el método OnDisappearing.

iOS y UWP



Como podemos apreciar en la anterior imagen, primero se ejecutará el método OnDisappearing() y después OnAppearing().

Navegación modal

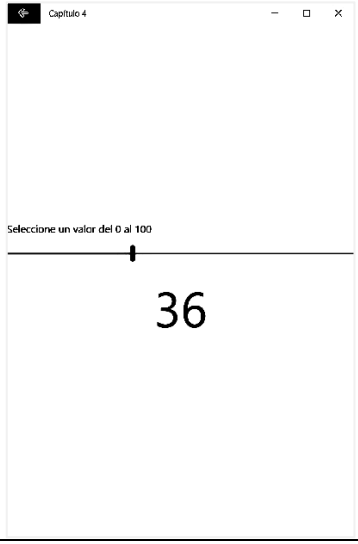
Con este tipo de navegación, podemos mostrar una segunda página de manera modal, es decir, una página que interrumpa el flujo de la aplicación hasta que manual o programáticamente sea cerrada. Este mecanismo es útil cuando deseamos presentar una caja de diálogo interactiva con una Interfaz de Usuario enriquecida. A diferencia de la navegación jerárquica, este tipo de navegación no requiere que la raíz de la aplicación sea de tipo NavigationPage.

La interfaz INavigationPage incluye los métodos asíncronos PushModalAsync() y PopModalAsync(), cuya funcionalidad es prácticamente la misma que sus similares en la navegación jerárquica: agregar una página a la pila de navegación y sacar una página de la pila de navegación, respectivamente.

El siguiente ejemplo muestra el código para navegar de manera modal a una página:

```
await Navigation.PushModalAsync(new Pagina3());
```

Las siguientes figuras muestran la aplicación después de haber navegado de manera modal a Pagina3. Observa que a diferencia de la navegación jerárquica, no se mostrará la barra de título:

Android	UWP
	

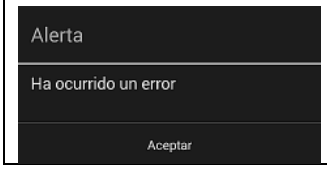
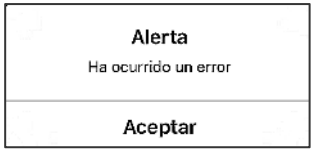
MENSAJERÍA

En el contexto de este capítulo nos referimos al concepto de mensajería con dos cosas: el poder mostrar al usuario ciertas cajas de diálogo, como respuesta a alguna solicitud o para ser notificado de alguna condición de la lógica de la aplicación, y el poder comunicar objetos entre sí de la manera más desacoplada que nos sea posible. La clase Page incluye los métodos DisplayAlert() y DisplayActionSheet() para lograr el primer objetivo, y la clase MessagingCenter nos permite lograr el segundo.

DisplayAlert

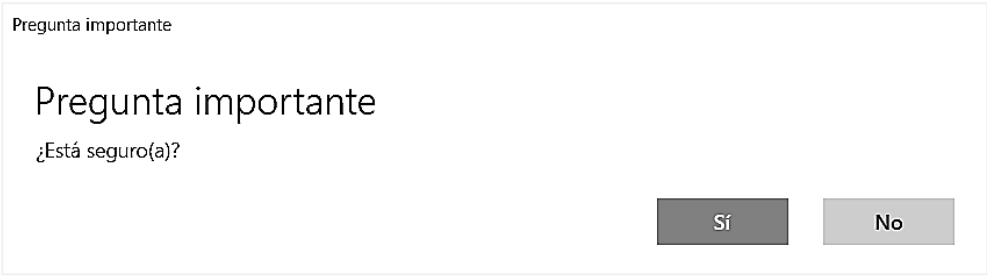
El método asíncrono DisplayAlert() nos permite mostrar una caja de diálogo con un mensaje para el usuario. Esta caja de diálogo puede tener uno o dos botones, además de un título que la identifique. En su forma más simple, podemos usar DisplayAlert() para simplemente presentar un mensaje, tal y como el siguiente código lo demuestra:

```
await DisplayAlert("Alerta", "Ha ocurrido un error",
"Aceptar");
```

Android	iOS	UWP
		

Adicionalmente, podemos usarlo como una función ya que una de sus sobrecargas puede devolver un objeto de tipo bool:

```
var result = await DisplayAlert("Pregunta importante",
"¿Está seguro(a)?", "Sí", "No");
if (result) …
```

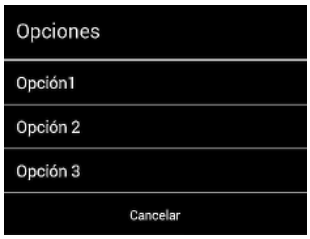
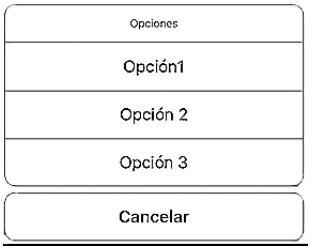
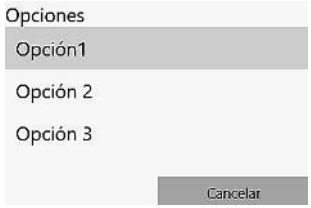


DisplayActionSheet

Este método asíncrono nos permite mostrar una caja de diálogo con una lista de opciones para que el usuario seleccione alguna de ellas. Una desventaja de DisplayActionSheet() es que la lista de opciones únicamente son cadenas, por lo que si se requiere algo más sofisticado se podría usar la opción de navegar a una página de forma modal, y en ella presentar un control de lista.

El siguiente fragmento de código nos muestra una caja de diálogo con tres opciones:

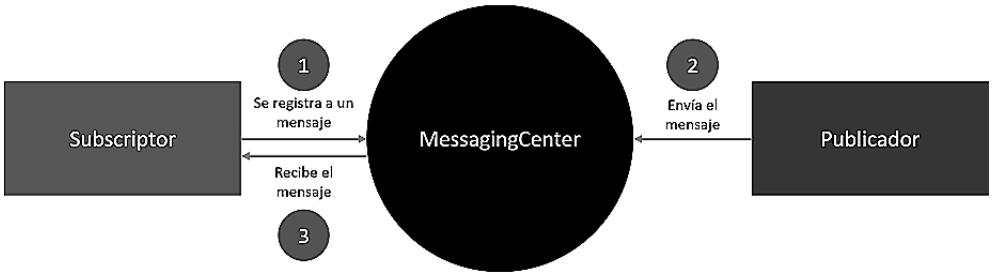
```
  var result =
      await DisplayActionSheet("Opciones", "Cancelar",
null, "Opción1", "Opción 2", "Opción 3");
  //La variable result contiene la opción seleccionada o
Cancelar
```

Android	iOS	UWP
		

Clase MessagingCenter

La clase MessagingCenter nos permite enviar y recibir mensajes entre diferentes objetos de una forma desacoplada. La funcionalidad de MessagingCenter está basada en el patrón de diseño Pub/Sub (Publish/Subscribe por su nombre en inglés), ya que los "publicadores" no envían los mensajes directamente a un receptor específico, sino a todos los "suscriptores" independientemente de cuáles sean estos.

El siguiente diagrama explica el mecanismo de envío y recepción de mensajes que ofrece MessagingCenter:



La siguiente tabla enlista los miembros de la clase MessagingCenter:

Send()	Envía un mensaje con o sin argumentos
Subscribe()	Se suscribe a un mensaje específico
Unsubscribe()	Se desinscribe de un mensaje

ENVÍO DE MENSAJES

Para enviar mensajes usamos el método Send(), en donde su primer parámetro es el tipo del "publicador", por lo que podemos establecer un tipo concreto o un tipo base. En su forma más básica, podemos enviar un mensaje simplemente identificándolo con una cadena:

```
MessagingCenter.Send((ContentPage)this, "Mensaje");
```

En el anterior código, estamos indicando que el "publicador" es ContentPage y no el tipo concreto de la página donde está este código (por ejemplo: Pagina2). Esto es importante para desacoplar al máximo a ambos publicadores y suscriptores.

De manera opcional, también podemos enviar argumentos. Estos pueden ser de cualquier tipo, pero en el siguiente ejemplo se muestra cómo se pasa como parámetro un objeto de tipo Persona:

```
MessagingCenter.Send((ContentPage)this, "Mensaje", new
Persona("Rodrigo"));
```

SUSCRIPCIÓN A MENSAJES

Para suscribirnos a un mensaje, usamos alguna de las sobrecargas del método genérico Subscribe: Subscribe<TSender>() o Subscribe<TSender, TArgs>(), ya que podemos suscribirnos tanto a mensajes que no tengan argumentos, como a los que sí. TSender es el tipo del publicador, de ahí que como mencionamos anteriormente, es importante no dejar acoplados los tipos de objetos. TArgs es el tipo del argumento que se espera recibir en el mensaje.

El siguiente fragmento de código muestra la suscripción a un mensaje llamado "Mensaje", y la implementación de un callback que contiene la lógica que queremos ejecutar cuando el mensaje sea recibido, en este caso, desplegar una caja de diálogo por medio de la invocación a DisplayAlert().

```
MessagingCenter.Subscribe<ContentPage>(this, "Mensaje",
async (s) =>
{
    await DisplayAlert("Mensaje", "Se ha recibido un
mensaje", "Aceptar");
});
```

El siguiente fragmento de código muestra la suscripción a un mensaje que tiene un argumento de tipo Persona:

```
MessagingCenter.Subscribe<ContentPage, Persona>(this,
"Mensaje", async (sender, args) =>
{
    await DisplayAlert("Alerta", $"La persona es:
{args.Nombre}", "Aceptar");
});
```

DESINSCRIPCIÓN A MENSAJES

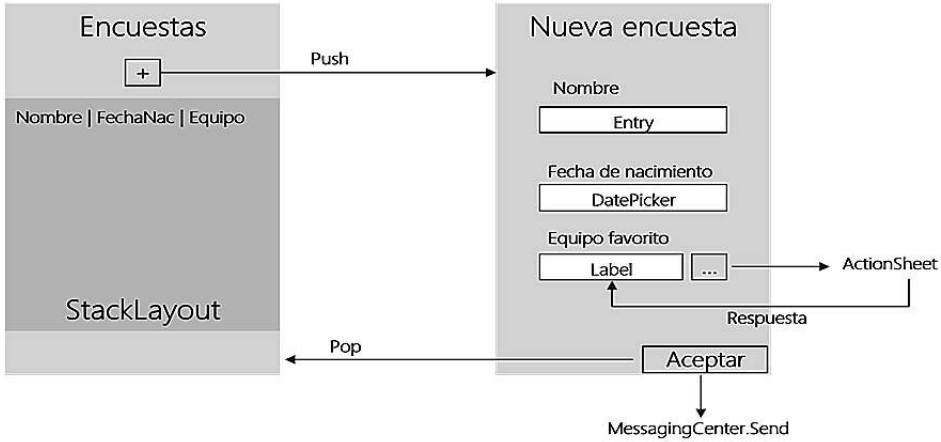
Para desinscribirnos a un mensaje, usamos el método Unsubscribe(), el cual debe repetir el tipo del publicador y del argumento (si hubiera alguno).

El siguiente fragmento de código muestra la desinscripción al mensaje que definimos anteriormente:

```
MessagingCenter.Unsubscribe<ContentPage, Persona>(this,
"Mensaje");
```

MANOS A LA OBRA

En este capítulo, comenzaremos con el desarrollo de la aplicación de encuestas que construiremos con Xamarin.Forms. Nuestro objetivo inmediato será construir la estructura inicial de la solución, e implementar las primeras piezas funcionales de nuestra aplicación. El siguiente diagrama muestra la arquitectura de la aplicación que tendremos una vez finalizados los pasos que a continuación se describen.



Creando la solución

La solución la llamaremos “Surveys” y tendrá la siguiente estructura de proyectos:

Surveys.Core	Biblioteca de Clases Portable
Surveys.Droid	Proyecto de Android
Surveys.iOS	Proyecto de iOS
Surveys.UWP	Proyecto de UWP

PLANTILLAS VS. DESDE CERO

Cuando se instala Xamarin en Visual Studio .NET, se incluirán algunas plantillas que podemos usar para crear nuestras aplicaciones con Xamarin.Forms. El uso de alguna de las plantillas existentes tiene las siguientes ventajas:

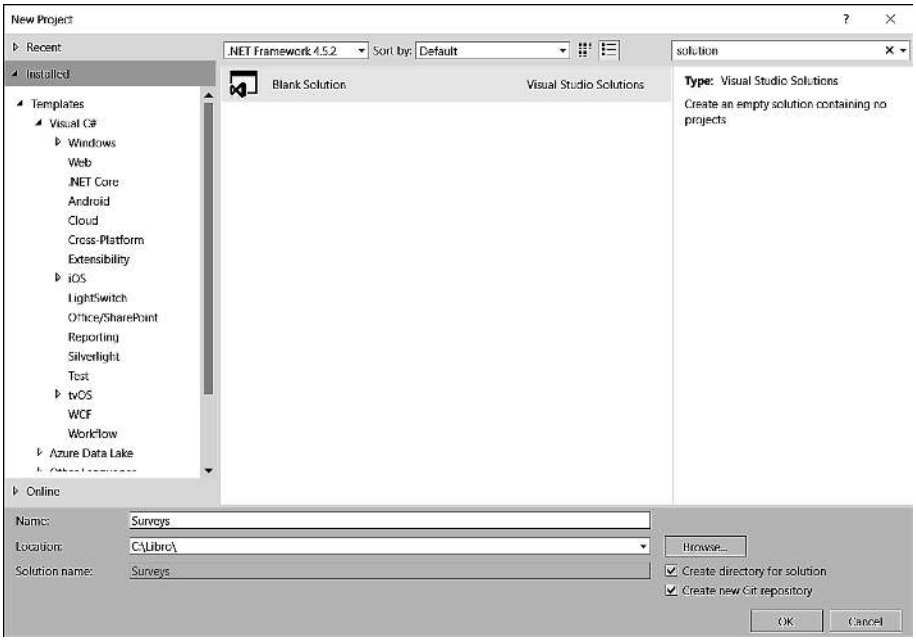
- Desplegará automáticamente los paquetes de NuGet de Xamarin.Forms en todos los proyectos
- Creará automáticamente la aplicación como un archivo App.xaml y su code-behind
- Implementará el código adecuado en MainActivity, AppDelegate y App.xaml/MainPage.xaml para Android, iOS y UWP, respectivamente

Sin embargo, desde mi punto de vista el uso de las plantillas tiene las siguientes desventajas, las cuales pesan más que las ventajas:

- La plantilla incluirá también la creación de los proyectos para Windows 8.1 y Windows Phone 8.1, que, a menos realmente sean plataformas que deseas soportar para tu aplicación con Xamarin.Forms, son un extraequipaje que probablemente no quieras tener
- Por lo anterior, la intersección de APIs a la que la PCL apuntará es más reducida
- El nombre por defecto para el proyecto PCL será el mismo que el de la solución, por lo que frecuentemente causa confusión

Por lo anterior, en este libro nos basaremos en la creación manual de la solución y sus proyectos, así como en la definición manual de los nombres. En mi opinión profesional y experiencia, es mucho más limpio y saludable para un proyecto comenzar de esta manera.

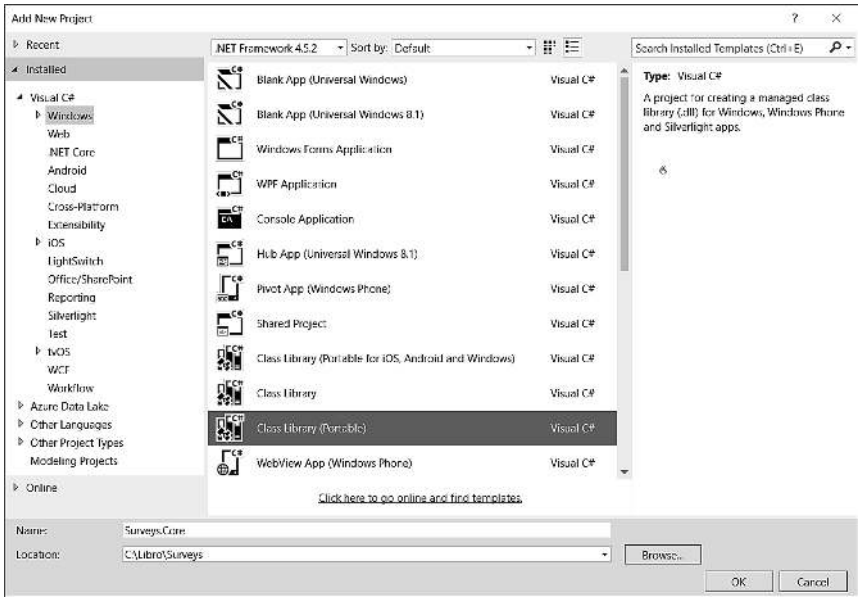
Ejecutaremos Visual Studio .NET y seleccionaremos la opción de crear un nuevo proyecto. En la caja de diálogo de "Nuevo proyecto" buscaremos la plantilla llamada "Blank Solution", y estableceremos el nombre a "Surveys", tal y como lo demuestra la siguiente figura:



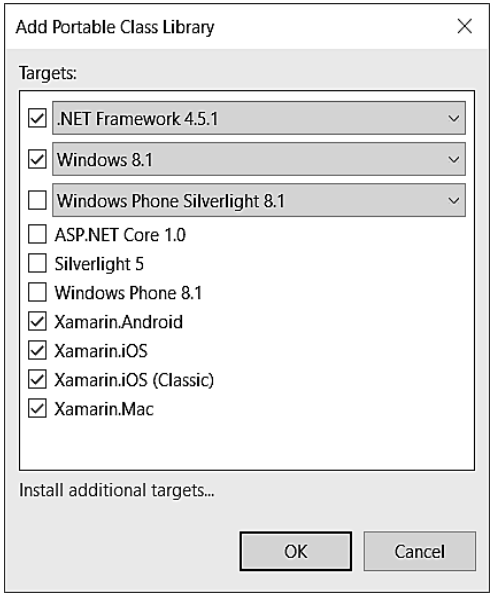
Observa que he seleccionado la opción de crear un nuevo repositorio de Git. Siempre es buena idea respaldarnos en un controlador de versiones para el código fuente de nuestras aplicaciones. Una explicación en el uso de Git o TFVC está fuera del alcance de este libro. Al hacer clic en el botón "OK", Visual Studio .NET creará una solución vacía, lista para contener todos los proyectos.

CREACIÓN DE LA PCL

A esta solución le agregaremos un proyecto de tipo PCL y le asignaremos el nombre "Surveys.Core", tal y como lo demuestra la siguiente figura:



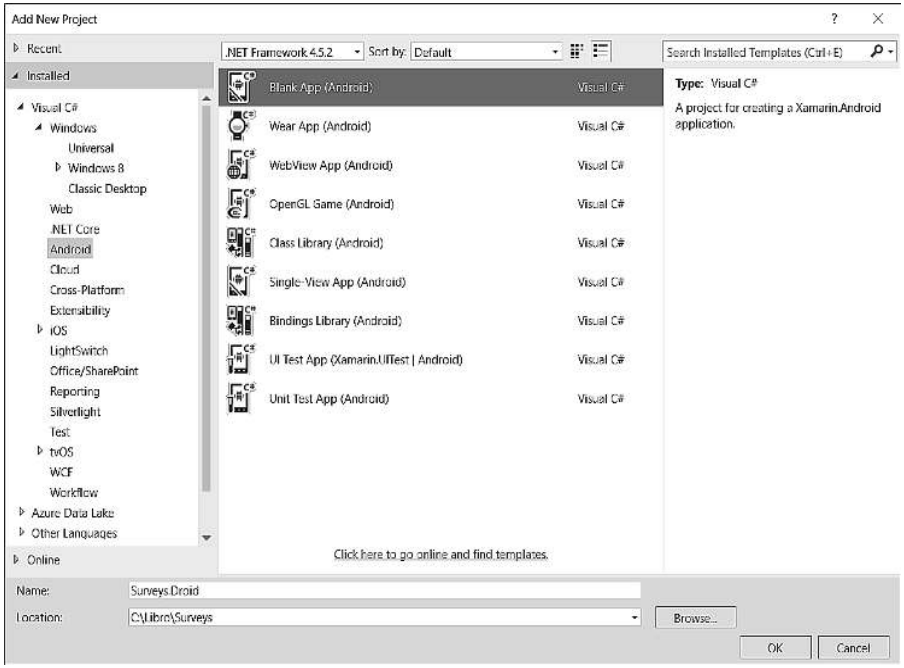
Al pulsar el botón OK, Visual Studio nos mostrará la caja de diálogo para seleccionar a qué plataformas deseamos apuntar en esta Biblioteca de Clases Base. Para este proyecto usaremos los siguientes:



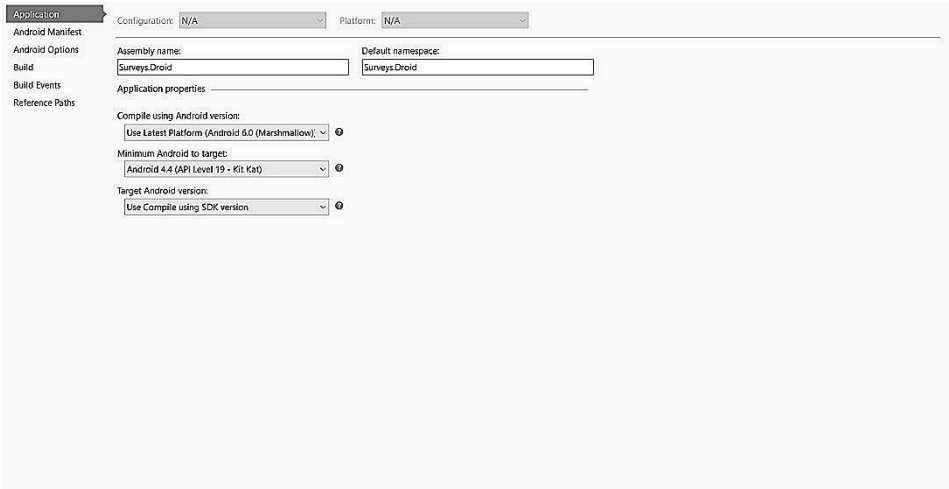
Después de hacer clic en el botón “OK”, Visual Studio creará el proyecto, incluyendo una clase llamada Class1.cs. Este archivo hay que borrarlo, ya que crearemos nuestras propias clases más adelante.

CREACIÓN DEL PROYECTO DE ANDROID

A nuestra solución, agregaremos un nuevo proyecto de tipo “Blank App (Android)” y le asignaremos el nombre “Surveys.Droid”. El sufijo “Droid” es recomendado para evitar la colisión de nombres con el espacio de nombres “Android” que tiene el SDK de dicho sistema operativo.

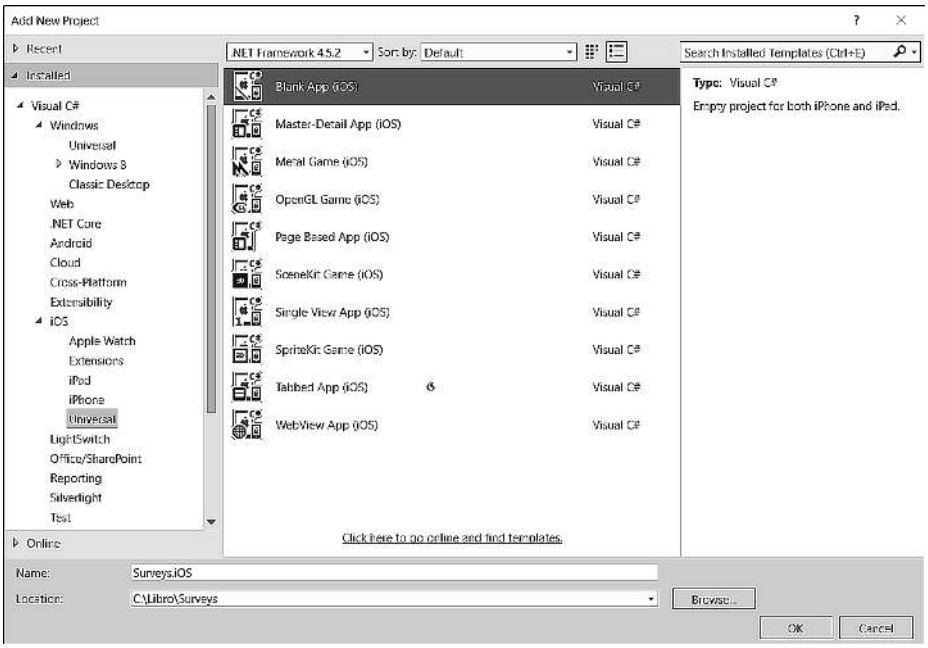


Una vez creado el proyecto de Android, abre sus propiedades y asegúrate de que la versión mínima sea alguna que tenga sentido para tus propósitos. En mi caso, seleccionaré la versión de Android 4.4 Nivel de API 19 (Kit Kat). La siguiente figura muestra las propiedades del proyecto de Android una vez establecida la versión mínima:

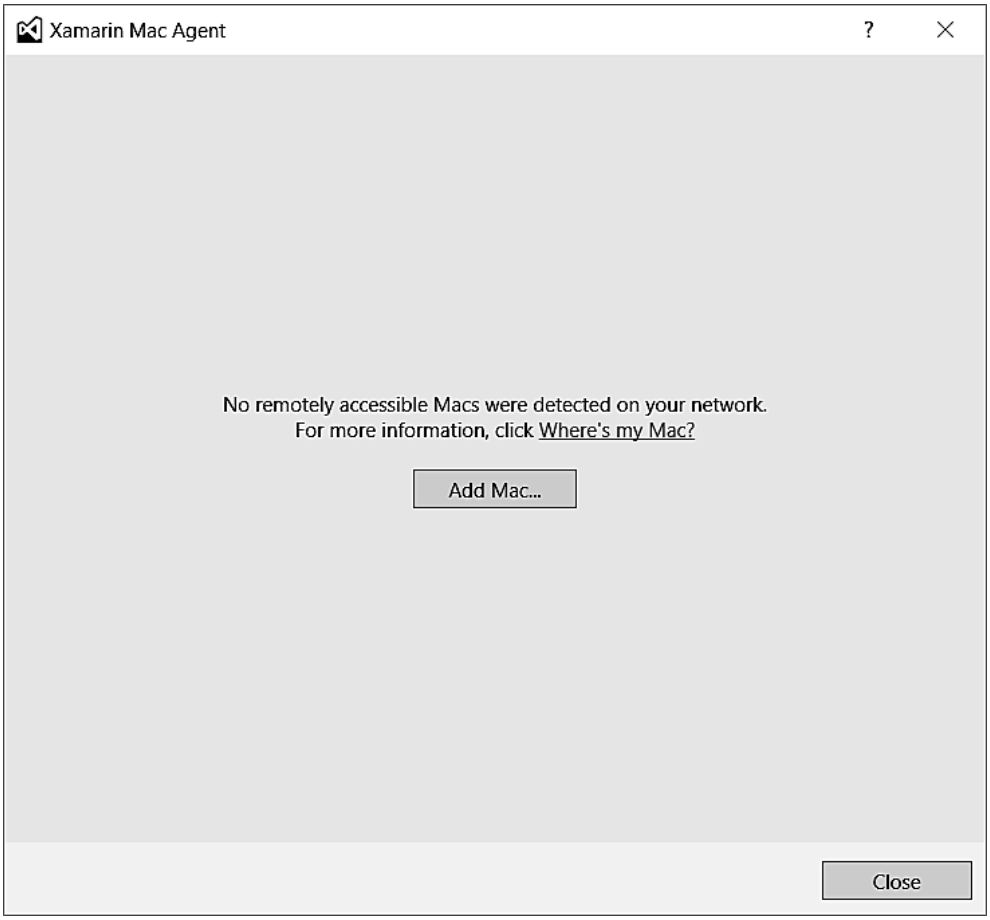


CREACIÓN DEL PROYECTO IOS

Agregaremos a la solución un nuevo proyecto de tipo “Blank App (iOS)”, y le asignaremos el nombre “Surveys.iOS”, tal y como lo demuestra la siguiente figura:



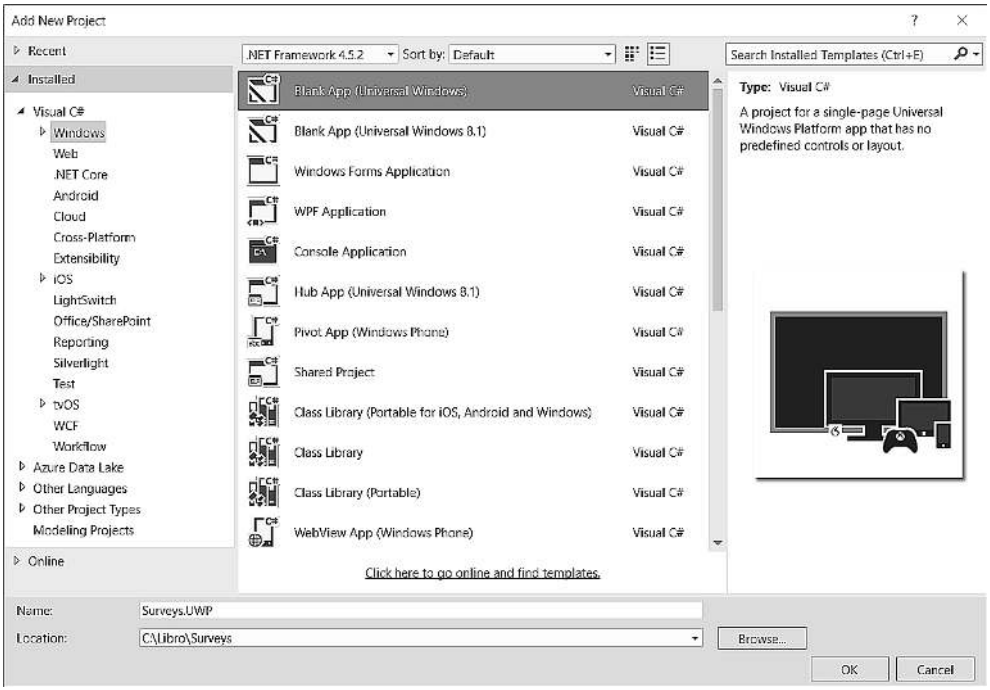
Si es la primera vez que creas un proyecto de tipo Xamarin.iOS en Visual Studio .NET, se mostrará una ventana con las instrucciones necesarias para configurar el equipo Mac que será utilizado para compilar la aplicación. De lo contrario, solo se mostrará una caja diálogo para seleccionar el equipo Mac que será usado como agente de compilación.



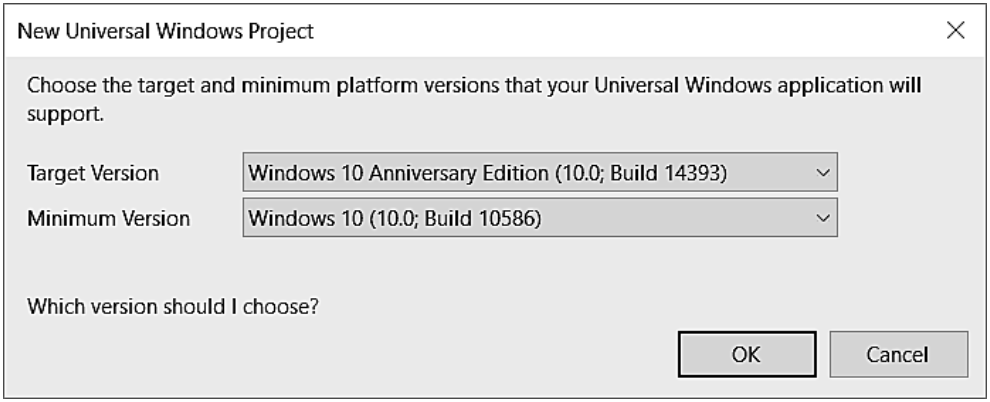
Si no cuentas con esta información en este momento, no te preocupes, ya que podrás configurar el agente de compilación para iOS por medio de las opciones de Xamarin disponibles en el menú Tools -> Options -> Xamarin -> iOS Settings.

CREACIÓN DEL PROYECTO UWP

Agreguemos a nuestra solución un nuevo proyecto de tipo “Blank App (Universal Windows)”, al que le asignaremos el nombre “Surveys.UWP”. La siguiente figura muestra la creación de este proyecto:

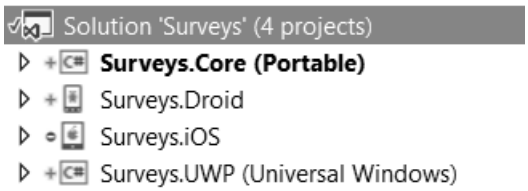


Después de hacer clic en el botón "OK", Visual Studio mostrará una caja de diálogo para seleccionar la versión de la plataforma a la que queremos apuntar. En mi caso, seleccionaré la versión 14393 como versión a la que quiero apuntar y como versión mínima la 10586. La siguiente figura muestra esta configuración:



Después de hacer clic en el botón OK, Visual Studio .NET creará el proyecto para la aplicación UWP.

A este punto, tenemos una solución con cuatro proyectos:



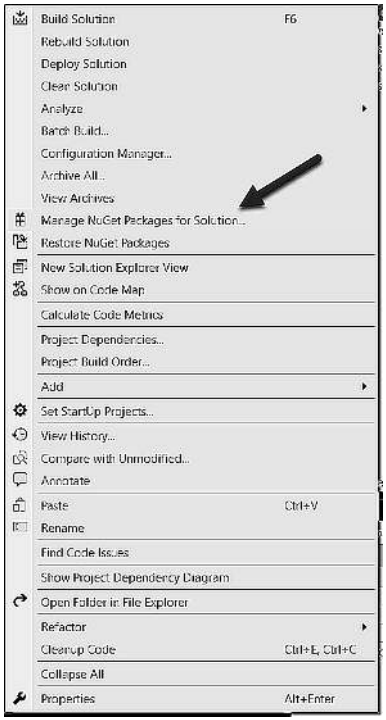
Configuración de la solución

De manera predeterminada, el proyecto de UWP no será compilado y desplegado, pero modificar esta configuración es muy sencilla si hacemos clic secundario sobre la solución, seleccionamos la opción "Configuration Manager…" y nos aseguramos de que estén marcadas las casillas "Build" y "Deploy" para el proyecto UWP. La siguiente figura muestra las modificaciones necesarias en la solución:

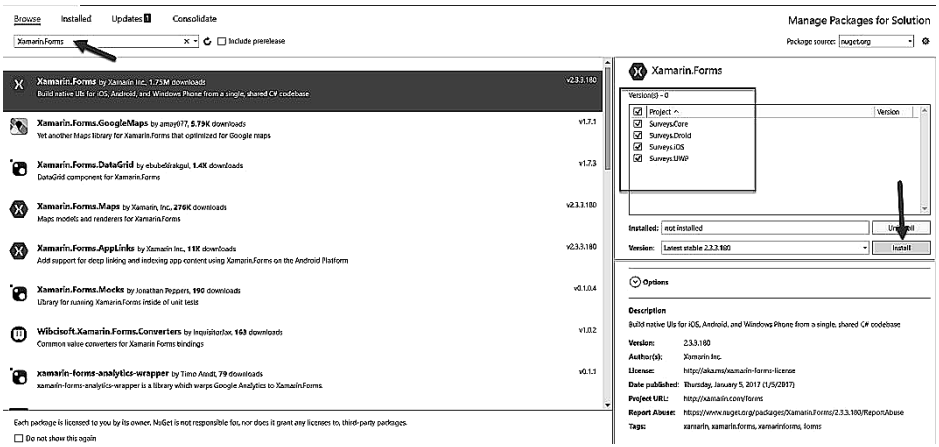
Project	Configuration	Platform	Build	Deploy
Surveys.Core	Debug	Any CPU	☑	☐
Surveys.Droid	Debug	Any CPU	☑	☑
Surveys.iOS	Debug	iPhone	☑	☐
Surveys.UWP	Debug	x86	☑	☑

Despliegue de los ensamblados de Xamarin.Forms

Una vez creada la estructura inicial de la solución, debemos agregar los ensamblados de Xamarin.Forms a través de NuGet. Para hacer esto, haz clic secundario sobre la solución y selecciona la opción "Manage NuGet Packages for Solution…" o su equivalente en el idioma de tu Visual Studio .NET:



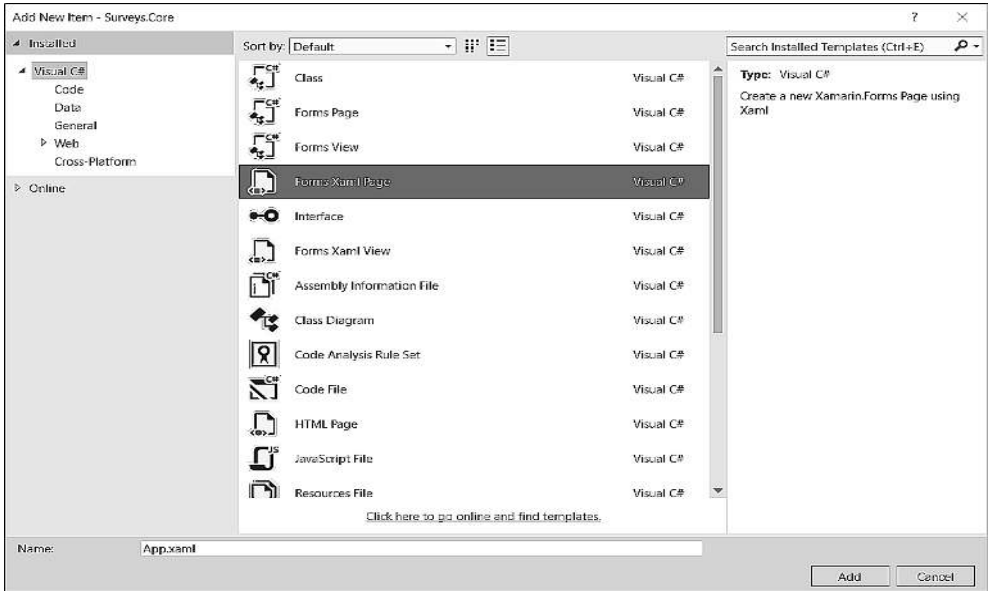
Al seleccionar esa opción, Visual Studio .NET abrirá el panel de NuGet. En él, seleccionaremos la opción "Browse" para hacer una búsqueda en el repositorio público de NuGet en Internet. El nombre del paquete de NuGet es "Xamarin.Forms", distribuido por Xamarin Inc. Una vez que hayas encontrado el paquete correcto, selecciona todos los proyectos de la solución y haz clic en el botón "Install", tal y como lo muestra la siguiente figura:



Nota: En el momento de estar escribiendo estas líneas, la última versión estable de Xamarin.Forms es la 2.3.3.180, pero siempre trata de usar la versión estable más reciente. Para mayor información acerca de las características incluidas en cada versión puedes consultar la sección de Xamarin.Forms en el foro oficial https://forums.xamarin.com/categories/xamarin-forms.

Creación de la aplicación en la PCL

Una vez instalado el paquete de NuGet de Xamarin.Forms en todos los proyectos, necesitamos crear la clase de Aplicación global. Para hacer esto, agregaremos a la PCL un nuevo archivo llamado App.xaml usando la plantilla "Forms Xaml Page". Usaremos esta técnica, ya que la plantilla nos ayudará a crear tanto el archivo XAML como su archivo de code-behind relacionado.



La plantilla en realidad creará un ContentPage, por lo que debemos modificar el elemento raíz en el documento XAML para que sea Application, y además debemos borrar todo su contenido. El siguiente fragmento código muestra el archivo XAML una vez hecho el cambio:

```
<?xml version="1.0" encoding="utf-8" ?>
<Application
xmlns="http://xamarin.com/schemas/2014/forms"
```

```
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
                x:Class="Surveys.Core.App">


</Application>
```

Asimismo, necesitamos modificar la clase de la que hereda App en el archivo de code-behind para que también herede de Application:

```
using Xamarin.Forms;


namespace Surveys.Core
{
    public partial class App : Application
    {
        public App()
        {
            InitializeComponent();
        }
    }
}
```

Modificación del punto de entrada en Android

La plantilla que utilizamos para crear el proyecto de Android agregará un archivo llamado MainActivity.cs, el cual debemos modificar para que herede de la clase FormsApplicationActivity en vez de simplemente Activity. Además, debemos iniciar el motor de Xamarin.Forms y posteriormente cargar la Aplicación global expuesta a través de la clase App en la PCL. Por lo tanto, lo primero que necesitamos hacer es agregar la referencia a la PCL en el proyecto de Android y posteriormente modificar MainActivity como a continuación se muestra en el siguiente fragmento de código:

```
using Android.App;
using Android.OS;
using Xamarin.Forms.Platform.Android;
```

```
namespace Surveys.Droid
{
    [Activity(Label = "Surveys.Droid", MainLauncher =
true, Icon = "@drawable/icon")]
    public class MainActivity :
FormsApplicationActivity
    {
        protected override void OnCreate(Bundle bundle)
        {
            base.OnCreate(bundle);

            Xamarin.Forms.Forms.Init(this, bundle);
            LoadApplication(new Core.App());
        }
    }
}
```

Nota: Más adelante en este libro, usaremos una mejor clase llamada FormsAppCompatActivity. Sin embargo, para efectos prácticos en este momento está bien con FormsApplicationActivity.

Modificación del punto de entrada en iOS

De igual manera que en Android, en el proyecto de iOS necesitamos agregar una referencia a la PCL y modificar la clase AppDelegate para que herede de la clase FormsApplicationDelegate en vez de simplemente UIApplicationDelegate. Posteriormente, en el método FinishedLaunching() iniciamos el motor de Xamarin.Forms y cargamos la aplicación de la PCL:

```
using Foundation;
using UIKit;
using Xamarin.Forms.Platform.iOS;

namespace Surveys.iOS
{
```

```
      [Register("AppDelegate")]
      public class AppDelegate : FormsApplicationDelegate
      {
          public override UIWindow Window { get; set; }

          public override bool
FinishedLaunching(UIApplication application, NSDictionary
launchOptions)
          {
              Xamarin.Forms.Forms.Init();
              LoadApplication(new Core.App());

              return base.FinishedLaunching(application,
launchOptions);
          }
      }
  }
```

Modificación del punto de entrada en UWP

Como comentamos anteriormente, para UWP son un poco diferentes los cambios que necesitamos realizar para poder incorporar Xamarin.Forms. Primero, referenciamos la PCL en el proyecto UWP y posteriormente modificamos el método OnLaunched() de App.xaml para inicializar el motor de Xamarin.Forms justo después de crear el objeto de tipo Frame:

```
  …
  rootFrame = new Frame();

  Xamarin.Forms.Forms.Init(e);
```

Una vez hecho esto, modificamos ambos MainPage.xaml y MainPage.xaml.cs para que su tipo sea WindowsPage. Finalmente, cargamos la aplicación global de la PCL después de la invocación del método InitializeComponent(). El siguiente fragmento de código muestra el archivo de code-behind de MainPage una vez hechas estas modificaciones:

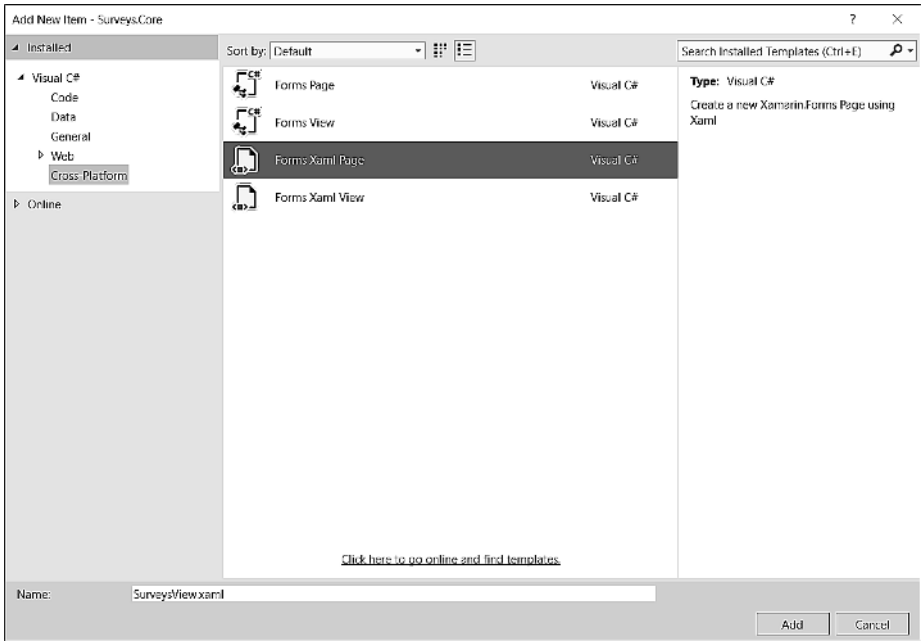
```
using Xamarin.Forms.Platform.UWP;

namespace Surveys.UWP
{
    public sealed partial class MainPage : WindowsPage
    {
        public MainPage()
        {
            this.InitializeComponent();
            LoadApplication(new Core.App());
        }
    }
}
```

Implementando las dos páginas

La aplicación de encuestas requiere las siguientes dos páginas:

SurveysView	Enlista las encuestas capturadas y permite capturar nuevas encuestas
SurveyDetailsView	Captura los datos en una nueva encuesta

Para agregar ambas páginas, usaremos la plantilla “Forms Xaml Page”, como lo muestra el siguiente ejemplo:



Una vez agregadas ambas páginas, modificaremos la Aplicación global para establecer como página raíz una página de tipo NavigationPage, que a su vez establezca a SurveysView como la primera página en la pila de navegación. El siguiente fragmento de código muestra las modificaciones a la clase App en la PCL:

```
using Xamarin.Forms;
namespace Surveys.Core
{
    public partial class App : Application
    {
        public App()
        {
            InitializeComponent();
            MainPage = new NavigationPage(new
SurveysView());
        }
    }
}
```

Implementando SurveysView

SurveysView es la página raíz de la aplicación. En esta página debemos implementar la Interfaz de Usuario necesaria para navegar a la página de captura de encuestas y para enlistar las encuestas capturadas. Basándonos en el diagrama de arquitectura presentado al inicio de esta sección, usaremos un StackLayout como contenedor de las encuestas capturadas, y presentaremos cada encuesta con un Label. Adicionalmente, agregaremos un botón con el texto "+" que nos permita navegar a la página SurveyDetailsView por medio de la infraestructura de navegación.

El siguiente fragmento de código muestra la implementación de SurveysView:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage
xmlns="http://xamarin.com/schemas/2014/forms"

xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="Surveys.Core.SurveysView"
             Title="Encuestas">
  <Grid Margin="10">
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto" />
      <RowDefinition />
    </Grid.RowDefinitions>

    <Button Text="+"
            Clicked="AddSurveyButton_Clicked" />

    <StackLayout Grid.Row="1"
                 x:Name="SurveysPanel" />
  </Grid>
</ContentPage>
```

Es importante destacar las siguientes características de esta implementación:

- A la página se le asigna el título "Encuestas"
- Se utiliza un Grid como contenedor raíz

- El botón está en la primera fila del Grid. Su manejador de evento es "AddSurveyButton_Clicked", implementado en el code-behind
- Se utiliza un StackLayout con el nombre "SurveysPanel" para mostrar las encuestas

Ahora, concentrémonos en el archivo de code-behind de esta página. La única interacción que tiene en este momento el usuario es pulsar el botón "+" para navegar a la página que permite capturar una nueva encuesta. Esto lo logramos a través de la invocación del método asíncrono PushAsync() de la interfaz INavigation disponible en la propiedad Navigation de la página:

```
  private async void AddSurveyButton_Clicked(object
sender, EventArgs e)
  {
      await Navigation.PushAsync(new
SurveyDetailsView());
  }
```

Una vez implementado el código anterior, si probamos la aplicación nos podemos percatar de que efectivamente la aplicación muestra como página raíz a SurveysView y podemos navegar a SurveyDetailsView si pulsamos en el botón "+". Vayamos ahora a la implementación de la segunda página para capturar una nueva encuesta.

Implementando SurveyDetailsView

En esta página debemos mostrar la Interfaz de Usuario necesaria para capturar una nueva encuesta. La siguiente tabla enlista los elementos visuales que necesitamos:

Entry	Captura el nombre del encuestado
DatePicker	Captura la fecha de nacimiento del encuestado
Equipo favorito	Muestra el equipo favorito seleccionado
Botón "…"	Despliega una lista de equipos para seleccionar
Botón "Aceptar"	Envía la encuesta capturada a la primera página

El siguiente fragmento de código muestra la implementación de la página SurveyDetailsView:

```
  <?xml version="1.0" encoding="utf-8" ?>
  <ContentPage
xmlns="http://xamarin.com/schemas/2014/forms"

xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
               x:Class="Surveys.Core.SurveyDetailsView"
               Title="Nueva encuesta">

    <Grid Margin="10">
      <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition Height="Auto" />
      </Grid.RowDefinitions>

      <StackLayout>
        <Label Text="Nombre" />
        <Entry x:Name="NameEntry" />

        <Label Text="Fecha de nacimiento" />
        <DatePicker x:Name="BirthdatePicker" />

        <Label Text="Equipo favorito" />
        <Grid>
          <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition Width="Auto" />
          </Grid.ColumnDefinitions>
          <Label x:Name="FavoriteTeamLabel" />
          <Button Grid.Column="1"
                  Text="..."
                  Clicked="FavoriteTeamButton_Clicked" />
```

```
        </Grid>
      </StackLayout>

      <Button Text="Aceptar"
              Grid.Row="1"
       Clicked="OkButton_Clicked" />
    </Grid>
  </ContentPage>
```

Desplegando los equipos

Uno de los requerimientos de la aplicación es poder mostrar una lista de equipos que el usuario puede seleccionar según la respuesta del encuestado. Antes de implementar el código para enlistar los equipos, me gustaría sugerir la implementación de una clase llamada Literals, ya pienso que es mejor separar las cadenas que serán presentadas en la Interfaz de Usuario en una clase por separado, para que en un futuro sea muy sencilla la localización en otro idioma si así fuese necesario.

Por lo tanto, agregaremos en la PCL una nueva clase llamada Literals.cs, con el siguiente contenido:

```
namespace Surveys.Core
{
    public class Literals
    {
        public const string FavoriteTeamTitle =
"Selección de equipo";
    }
}
```

Ahora, desplegaremos la lista de equipos. Como lo mencionamos anteriormente, a través del uso del método DisplayActionSheet() podemos desplegar una lista de cadenas que el usuario puede seleccionar. El siguiente fragmento de código muestra la implementación del manejador del evento Clicked del botón "…", en donde usamos el método DisplayActionSheet() para mostrar la lista de equipos:

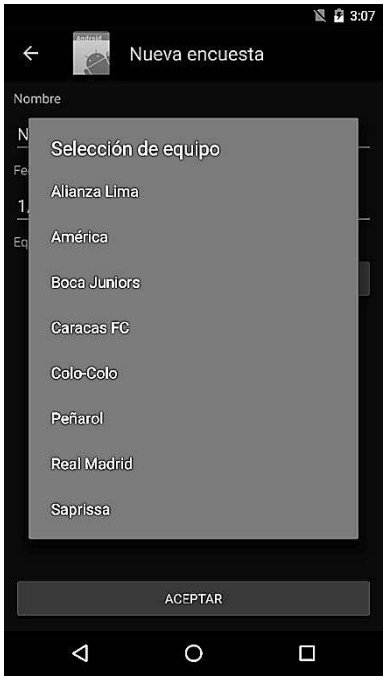
```
using System;
```

```
using Xamarin.Forms;

namespace Surveys.Core
{
    public partial class SurveyDetailsView :
ContentPage
    {
        private readonly string[] teams =
        {
            "Alianza Lima",
            "América",
            "Boca Juniors",
            "Caracas FC",
            "Colo-Colo",
            "Peñarol",
            "Real Madrid",
            "Saprissa"
        };

        public SurveyDetailsView()
        {
            InitializeComponent();
        }

        private async void
FavoriteTeamButton_Clicked(object sender, EventArgs e)
        {
      var favoriteTeam = await
DisplayActionSheet(Literals.FavoriteTeamTitle, null,
null, teams);
```

```
                if
(!string.IsNullOrWhiteSpace(favoriteTeam))
                {
                    FavoriteTeamLabel.Text = favoriteTeam;
                }       }
        }
    }
```

Es importante destacar que el método DisplayActionSheet() es una función y nos devolverá la cadena seleccionada… ¡incluso el texto de los botones! Por este motivo, usamos null para no mostrar ningún botón. Adicionalmente, evaluamos el resultado de la función para solo asignar el texto en el Label "FavoriteTeamLabel" cuando verdaderamente se haya seleccionado un elemento de la lista.

Si probamos la aplicación en este momento, podremos asegurarnos de que el botón despliega correctamente la lista de equipos. La siguiente figura muestra el resultado de esta prueba en Android:



Comunicando ambas páginas

¿Cómo devolvemos la encuesta capturada por el usuario a la página SurveysView para poder agregarla al StackLayout? Aquí podríamos usar diversos mecanismos, pero en este caso usaremos la clase MessagingCenter para publicar un mensaje cuando una encuesta ha sido capturada, para promover la separación de preocupaciones entre ambas páginas y para ejercitar nuestros conocimientos obtenidos en este capítulo.

Podríamos pasar como argumento una cadena con todos los elementos separados por algún carácter especial como "|" y en en el callback de suscripción interpretar esta cadena con el método Split() de la clase System.String. No obstante, usaremos una forma mucho mejor: crearemos una nueva clase llamada "Survey" que nos sirva para modelar una encuesta dentro de esta aplicación. Adicionalmente, remplazaremos el método ToString() para poder devolver una cadena que concatene todas sus propiedades en vez de simplemente regresar el nombre del tipo totalmente cualificado. El siguiente fragmento código muestra la implementación de esta clase:

```
using System;
namespace Surveys.Core
{
    public class Survey
    {
        public string Name { get; set; }

        public DateTime Birthdate { get; set; }

        public string FavoriteTeam { get; set; }

        public override string ToString()
        {
            return $"{Name} |  {Birthdate} |
{FavoriteTeam}";
        }
    }
}
```

Adicionalmente, para seguir promoviendo la separación de preocupaciones en nuestro código crearemos una nueva clase llamada Messages, la cual contendrá la lista de todos los diferentes nombres de los mensajes que usaremos durante la construcción de esta aplicación:

```
namespace Surveys.Core
{
    public class Messages
    {
        public const string NewSurveyComplete =
"NewSurveyComplete";
    }
}
```

El siguiente fragmento de código muestra la implementación del manejador del evento Clicked del botón "Aceptar", para crear un nuevo objeto de tipo Survey –previa validación de los datos– y enviarlo como argumento del mensaje "NewSurveyComplete" a través de la clase MessagingCenter:

```
private async void OkButton_Clicked(object sender,
EventArgs e)
{
    //Evaluamos si los datos están completos
    if (string.IsNullOrWhiteSpace(NameEntry.Text) ||
string.IsNullOrWhiteSpace(FavoriteTeamLabel.Text))
    {
        return;
    }

    //Creamos el nuevo objeto de tipo Survey
    var newSurvey = new Survey()
    {
        Name = NameEntry.Text,
        Birthdate = BirthdatePicker.Date,
        FavoriteTeam = FavoriteTeamLabel.Text
    };
```



```
        //Publicamos el mensaje con el objeto de encuesta
como argumento
        MessagingCenter.Send((ContentPage)this,
Messages.NewSurveyComplete, newSurvey);

        //Removemos la página de la pila de navegación para
regresar inmediatamente
        await Navigation.PopAsync();
    }
```

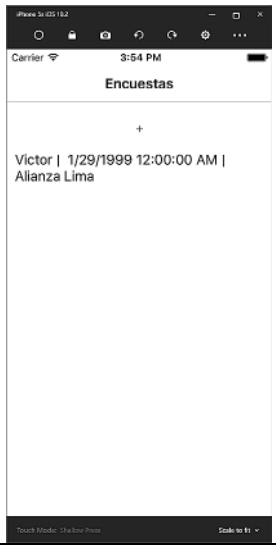
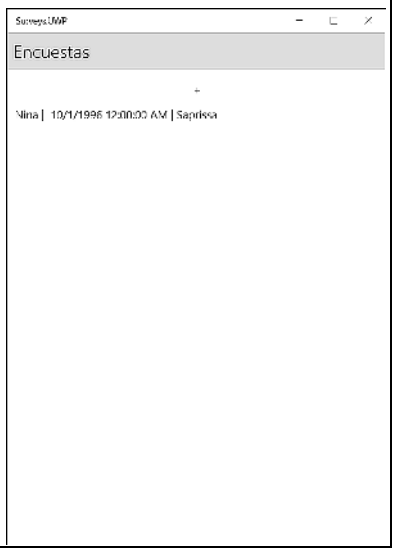
Ya que estamos publicando el mensaje, ahora necesitamos suscribirnos a él.

De regreso a la página SurveysView, implementaremos el siguiente código para suscribirnos al mensaje "NewSurveyComplete". Cuando se reciba el mensaje agregaremos un nuevo objeto de tipo Label al StackLayout. Esta suscripción la haremos en el constructor de clase:

```
    public SurveysView()
    {
        InitializeComponent();

        MessagingCenter.Subscribe<ContentPage,
Survey>(this, Messages.NewSurveyComplete, (sender, args)
=>
        {
            SurveysPanel.Children.Add(new Label()
            {
                Text = args.ToString()
            });
        });
    }
```

Probando la aplicación

Ya que tenemos la implementación lista, podemos probar la aplicación en las diferentes plataformas a las que deseamos apuntar:

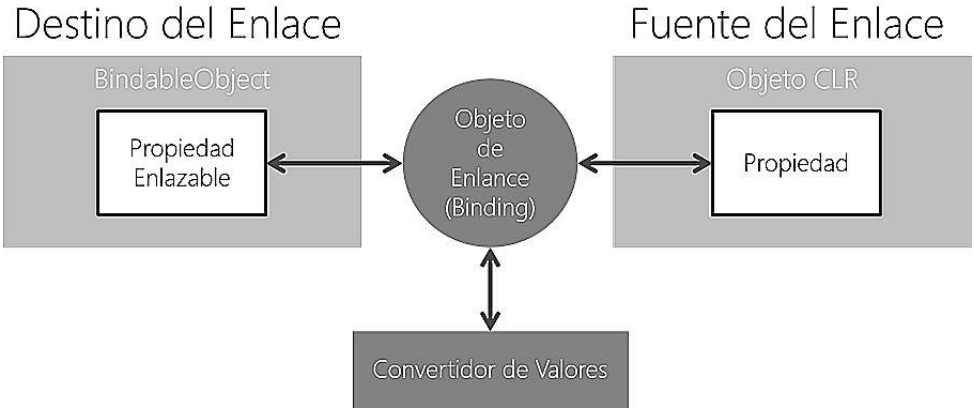
Android	iOS	UWP
		

5 ENLACE DE DATOS

INTRODUCCIÓN

La infraestructura de enlace de datos en Xamarin.Forms nos permite establecer una conexión entre la Interfaz de Usuario y la lógica de negocio. Su modelo es muy simple, ya que se basa en tener cuatro piezas: un objeto fuente en donde has identificado una propiedad a la que quieres enlazar a una propiedad de un control destino, es decir: un objeto fuente, un control destino, una propiedad fuente y una propiedad destino. Una vez identificadas estas piezas, expresamos el enlace de datos a través de la extensión de marcado {Binding}. Opcionalmente, también podemos usar un Convertidor de Valor para que el dato enlazado sea modificado o le apliquemos alguna lógica adicional.

En el siguiente diagrama, se muestra el modelo de enlace de datos en Xamarin.Forms:



El siguiente fragmento de código muestra la declaración básica de un enlace de datos. En él, podemos identificar las cuatro piezas que describimos anteriormente:

- El objeto fuente es album1, el cual referenciamos de manera estática a través de la extensión de marcado {StaticResource}
- La propiedad fuente es “Titulo”
- El objeto destino es un Label
- La propiedad en el objeto destino es “Text”

```
<Label Text="{Binding Source={StaticResource album1},
Path=Titulo}" />
```

La extensión de marcado {Binding} tiene diferentes propiedades que podemos usar para definir cómo queremos que se haga el enlace. Las tres propiedades principales son: Source, Path y Mode.

Source

La propiedad Source indica cuál es el objeto fuente de nuestro enlace de datos. Este objeto fuente puede ser cualquier tipo de objeto CLR.

Path

La propiedad Path indica la ruta para llegar al valor que deseamos enlazar. Esta propiedad es la primera en la posición dentro de la extensión de marcado, por lo que las siguientes declaraciones son equivalentes:

```
<Label Text="{Binding Source={StaticResource album1},
Path=Titulo}" />
```

```
<Label Text="{Binding Titulo, Source={StaticResource
album1}}" />
```

Como podrás apreciar en el segundo ejemplo, lo que establezcas después de “Binding” lo tomará como la propiedad Path. Esta sintaxis es la que usaré en el resto del libro, por ser más compacta y un tanto más elegante.

Mode

La propiedad Mode indica el modo para el enlace de datos. La siguiente tabla describe los cuatro modos de enlace de datos que soporta Xamarin.Forms:

Default	Modo predeterminado establecido por el autor de la propiedad enlazable que estamos utilizando como propiedad destino. Ya que el modo puede variar según la clase y propiedad en la XAML, la mejor práctica es establecer el modo de manera explícita
OneWay	En este modo, el enlace se establece y cualquier cambio que suceda en la propiedad fuente se notificará inmediatamente a la propiedad del control destino
OneWayToSource	En este modo, el enlace se establece y cualquier cambio que suceda en la propiedad del control destino, le notificará inmediatamente a la propiedad del objeto fuente. Un ejemplo de este escenario es el caso de un control interactivo
TwoWay	En este modo, el enlace se establece y cualquier cambio que suceda en la propiedad fuente le notificará a la propiedad destino y viceversa. Es decir, permite una notificación bidireccional

Hagamos entonces en este momento un ejemplo sencillo. Dada la clase que a continuación se muestra, enlacémosla a un par de etiquetas para mostrar sus valores.

```
public class Persona
{
    public string Nombre { get; set; }
    public string Pais { get; set; }
}
```

El siguiente fragmento de código crea una instancia de la clase Persona (previa importación del espacio de nombres) y le asigna valores a sus propiedades Nombre y Pais. Posteriormente, enlaza estas propiedades a la propiedad Text de dos Label:

```
…
<ContentPage.Resources>
  <ResourceDictionary>
    <local:Persona x:Key="persona1"
                   Nombre="Rodrigo"
                   Pais="México" />
  </ResourceDictionary>
```

```
  </ContentPage.Resources>

  <StackLayout Margin="10">
    <Label Text="{Binding Nombre, Source={StaticResource
persona1}, Mode=OneWay}" />
    <Label Text="{Binding Pais, Source={StaticResource
persona1}, Mode=OneWay}" />
  </StackLayout>
```

Como podemos apreciar con el anterior código, "Nombre" y "Pais" los está estableciendo como Path del objeto Binding, no de manera explícita sino usando su posición. Además, estamos usando como objeto fuente el objeto de tipo Persona identificado por medio de la clave única "persona1". Finalmente, establecemos de manera explícita el tipo de modo para que sea de una sola vía (OneWay).

Si el modo es OneWay, ¿qué sucederá si modificamos el valor de las propiedades? Veamos justamente ese escenario. Agreguemos un botón con el texto "Modificar" al StackLayout y manejemos su evento Clicked, tal y como lo muestra el siguiente fragmento de código:

```
  <StackLayout Margin="10">
    <Label Text="{Binding Nombre, Source={StaticResource
persona1}, Mode=OneWay}" />
    <Label Text="{Binding Pais, Source={StaticResource
persona1}, Mode=OneWay}" />
    <Button Text="Modificar"
            Clicked="Button_OnClicked" />
  </StackLayout>
```

En el archivo de code-behind de la página, implementaremos el código necesario para modificar los valores de las propiedades Nombre y Pais del objeto Persona. Para lograrlo, primero obtendremos el recurso llamado "persona1" a través de la propiedad Resources de la página y luego haremos el cambio en los valores. El siguiente fragmento de código muestra la implementación del manejador del evento Clicked:

```
  private void Button_OnClicked(object sender, EventArgs
e)
  {
      var persona1 = (Persona)Resources["persona1"];
```



```
        persona1.Nombre = "Diego";
        persona1.Pais = "Argentina";
    }
```

Si ejecutamos en este momento la aplicación y pulsamos en el botón “Modificar”, veremos que el texto de los Label ¡no cambia!, no obstante, tanto el nombre como el país sí fueron modificados para la ejecución del código. Es decir, la Interfaz de Usuario no se sincroniza. Pero entonces, ¿qué está sucediendo? La respuesta está en que nuestro objeto fuente tiene un requisito: necesita notificar de alguna manera a la infraestructura de enlace de datos que el valor de ambas propiedades “Nombre” y “Pais” ha cambiado. Esto obedece al hecho de que, a diferencia de la propiedad Text del Label que sí es una Propiedad Enlazable y por ende notifica el cambio de valor, las propiedades de la clase Person son simples propiedades CLR tradicionales: no tienen la funcionalidad en sí mismas de notificación. Lo que necesitamos entonces es hacer que nuestra clase Person sea una clase capaz de notificar los cambios.

Interfaces INotifyPropertyChanged y INotifyCollectionChanged

Los modos OneWay y TwoWay requieren que el objeto fuente implemente alguna de las interfaces **INotifyPropertyChanged** o **INotifyCollectionChanged.** INotifyPropertyChanged es una interfaz que utilizamos en nuestras clases que deseemos notifiquen acerca del cambio de valor de alguna de sus propiedades. Por su parte, la interfaz INotifyCollectionChanged es para las clases de tipo colección en las que deseemos notificar que la colección ha sido cambiada, ya sea por agregar un nuevo elemento, borrar un elemento o mover un elemento dentro de la colección.

IMPLEMENTANDO INOTIFYPROPERTYCHANGED

Como cualquier otra interfaz, implementamos INotifyPropertyChanged simplemente cumpliendo con los miembros que exige, pero aquí tenemos buena suerte ya que INotifyPropertyChanged es una interfaz muy sencilla: solo tiene un miembro llamado PropertyChanged, el cual es un evento cuyos argumentos traen consigo el nombre de la propiedad cuyo valor ha sido cambiado. ¿Suena complicado? ¡Para nada! Hagamos la implementación más básica de esta interfaz en la clase Persona, como a continuación lo muestra el siguiente código:

```
public class Persona : INotifyPropertyChanged
{
    public string Nombre { get; set; }
    public string Pais { get; set; }
```

```
    public event PropertyChangedEventHandler
PropertyChanged;
}
```

Simple, ¿cierto? No obstante, aún necesitamos más cosas. De entrada, necesitamos un método que nos permita disparar el evento PropertyChanged cuando el valor de la propiedad haya cambiado. Observa que en el siguiente código estamos solicitando como parámetro el nombre de la propiedad, el cual pasaremos en el constructor del objeto de argumentos que requiere el evento PropertyChanged:

```
public void OnPropertyChanged(string propertyName)
{
    PropertyChanged?.Invoke(this, new
PropertyChangedEventArgs(propertyName));
}
```

Por lo tanto, ya en cada accesor set{} podemos invocar este método. Sin embargo, en este momento nos hemos topado con pared, ya que las propiedades “Nombre” y “Pais” están expresadas en el código como propiedades autoimplementadas. Por esa razón no podemos agregar código en ninguno de sus accesores así que modificaremos su implementación como propiedades tradicionales con un campo privado que sirva para almacenar el valor, y la propiedad con ambos accesores get{} y set{}, usando dicho campo privado. Una vez hecho esto, ya estamos listos para invocar el método OnPropertyChanged() que acabamos de escribir:

```
private string nombre;
public string Nombre
{
    get
    {
        return nombre;
    }
    set
    {
        nombre = value;
        OnPropertyChanged("Nombre");
    }
```



```
}
private string pais;
public string Pais
{
    get
    {
        return pais;
    }
    set
    {
        pais = value;
        OnPropertyChanged("Pais");
    }
}
```

Si ejecutamos la aplicación nuevamente y pulsamos el botón “Modificar” nos podremos percatar de que efectivamente ahora los textos de los Label son sincronizados automáticamente, gracias a que ahora la clase Persona es una clase con la capacidad de notificar los cambios en sus propiedades. ¡Sin haber modificado nada en el XAML! La interfaz INotifyPropertyChanged es el héroe del día.

Esto está muy bien, pero nuestro código tiene muchos lados donde lo podemos mejorar y justamente haremos eso a continuación.

Observa la invocación del método OnPropertyChanged en ambas propiedades “Nombre” y “Pais”. En ambos casos está pasando una cadena con el nombre de la propiedad. Esto será un problema cuando queramos usar alguna herramienta de refactoring para cambiar el nombre de las propiedades, o simplemente cuando deseemos cambiar manualmente el nombre de las propiedades. Y es que la herramienta de refactoring no modificará la cadena, y a ti se te puede olvidar cambiar la cadena, o sencillamente tuviste una falta de ortografía.

Por esta razón, mejoraremos nuestro método OnPropertyChanged(), decorando el parámetro propertyName con el atributo [CallerMemberName]. Este útil atributo infiere el nombre del miembro que está invocando el método en la pila de llamadas del código y pondrá ese nombre en la propiedad propertyName automáticamente. Por lo tanto, nuestra nueva implementación del método será la que muestra el siguiente fragmento de código:

```
  public void OnPropertyChanged([CallerMemberName] string
propertyName = null)
  {
      PropertyChanged?.Invoke(this, new
PropertyChangedEventArgs(propertyName));
  }
```

Gracias a [CallerMemberName] ahora nuestras propiedades podrán invocar OnPropertyChanged() sin parámetro alguno, quedando de la siguiente manera:

```
  private string nombre;
  public string Nombre
  {
      get
      {
          return nombre;
      }
      set
      {
          nombre = value;
          OnPropertyChanged();
      }
  }

  private string pais;
  public string Pais
  {
      get
      {
          return pais;
      }
      set
      {
```

```
            pais = value;
            OnPropertyChanged();
        }
    }
```

Ahora bien, todavía podemos hacer otro cambio en beneficio de nuestra aplicación. Una mejor práctica es implementar la interfaz INotifyPropertyChanged en una clase base abstracta separada, para que diversas clases en tu aplicación puedan heredar de ella y por lo tanto obtener la funcionalidad de notificación cuando cambian sus propiedades, sin necesidad de repetir esa lógica una y otra vez en cada una de ellas. En el siguiente fragmento de código se muestra la implementación de una clase abstracta llamada "Notificable:

```
  public abstract class Notificable :
INotifyPropertyChanged
  {
      public event PropertyChangedEventHandler
PropertyChanged;


      protected virtual void
OnPropertyChanged([CallerMemberName] string propertyName
= null)
      {
          PropertyChanged?.Invoke(this, new
PropertyChangedEventArgs(propertyName));
      }
  }
```

Por lo tanto, podemos modificar la clase Persona para que herede de la clase Notificable, en vez de implementar directamente la interfaz INotifyPropertyChanged:

```
  public class Persona : Notificable
  {
  …
  }
```

Finalmente, otra buena práctica más que quiero compartir para que la tomes en cuenta en tus aplicaciones con Xamarin.Forms: siempre es buena idea notificar únicamente cuando verdaderamente el valor de una propiedad haya sido cambiado,

por lo que te recomiendo validar esto en el accesor set{} de todas tus propiedades. El siguiente fragmento de código muestra esta validación en la propiedad “Nombre” de la clase Persona:

```
private string nombre;
public string Nombre
{
    get
    {
        return nombre;
    }
    set
    {
        if (nombre == value)
        {
            return;
        }
        nombre = value;
        OnPropertyChanged();
    }
}
```

IMPLEMENTANDO INOTIFYCOLLECTIONCHANGED

Como lo mencionamos con anterioridad, la interfaz INotifyCollectionChanged permite que una clase de tipo colección notifique cuando ha cambiado, esto es, cuando se ha agregado un elemento, se ha borrado un elemento o se ha movido un elemento dentro de la colección. Al igual que INotifyPropertyChanged, la interfaz INotifyCollectionChanged solo tiene un miembro: el evento CollectionChanged. Este evento debe ser disparado cuando la colección ha sido cambiada. Por ejemplo, en la clase Collection<T> contamos con los métodos InsertItem(), RemoveItem() y SetItem() que son invocados automáticamente cuando se agrega, borra o mueve un elemento respectivamente.

El siguiente fragmento de código muestra una clase llamada Personas, basada en la clase genérica Collection<Persona>, en donde hemos implementado la interfaz INotifyCollectionChanged:


```
  public class Personas : Collection<Persona>,
INotifyCollectionChanged
  {
      protected override void InsertItem(int index,
Persona item) {
          base.InsertItem(index, item);

OnCollectionChanged(NotifyCollectionChangedAction.Add,
index, item);
      }
      protected override void RemoveItem(int index) {
          base.RemoveItem(index);

OnCollectionChanged(NotifyCollectionChangedAction.Remove,
index);
      }
      protected override void SetItem(int index, Persona
item) {
          Persona oldItem = this[index];


          base.SetItem(index, item);

OnCollectionChanged(NotifyCollectionChangedAction.Replace
, item, oldItem, index);
      }
      public event NotifyCollectionChangedEventHandler
CollectionChanged;
  }
```

¿Significa esto que debes implementar INotifyCollectionChanged en todas tus colecciones? ¡Para nada! A continuación veremos que Xamarin.Forms incluye una mejor manera.

ObservableCollection<T>

El API de Xamarin.Forms incluye una clase genérica llamada ObservableCollection<T>, la cual es el tipo de clase recomendada y sugerida para todas tus colecciones en esta plataforma, ya que esta clase ya implementa INotifyCollectionChanged (y de hecho también INotifyPropertyChanged). Esta clase está ubicada en el espacio de nombres System.Collections.ObjectModel.

Si queremos ejemplificar el uso de ObservableCollection<T>, el usar un solo objeto de tipo Persona no será suficiente. Debido a esto, ahora crearemos una nueva clase llamada "Datos", la cual expondrá una propiedad llamada "Personas" (nota el nombre en plural, para que no te confundas) de tipo ObservableCollection<Persona>. Adicionalmente, la clase Datos heredará de la clase Notificable, para permitir que también la propiedad "Personas" notifique cuando ha sido modificada (por ejemplo, cuando inicializas la propiedad). No confundas esto con el hecho de que la colección también es capaz de notificar.

El siguiente fragmento de código muestra la implementación inicial de la clase Datos:

```
public class Datos : Notificable
{
    private ObservableCollection<Persona> personas;

    public ObservableCollection<Persona> Personas
    {
        get
        {
            return personas;
        }
        set
        {
            if (personas == value)
            {
                return;
            }
            personas = value;
```



```
                OnPropertyChanged();
            }
        }
    }
```

En este momento la colección es null, por lo que crearemos algunos datos de ejemplo que nos sirvan para corroborar la funcionalidad de la clase ObservableCollection<T>. Esto lo haremos en el constructor de la clase Datos de la siguiente manera:

```
    public Datos()
    {
        Personas = new ObservableCollection<Persona>();


        for (int i = 0; i < 5; i++)
        {
            Personas.Add(new Persona() { Nombre = $"Persona
{i}", Pais = $"País {i}" });
        }
    }
```

De regreso a la página, modificaremos el XAML para hacer una instancia de la clase Datos y enlazaremos la propiedad ItemsSource de un control de tipo ListView a la colección expuesta a través de la propiedad Personas.

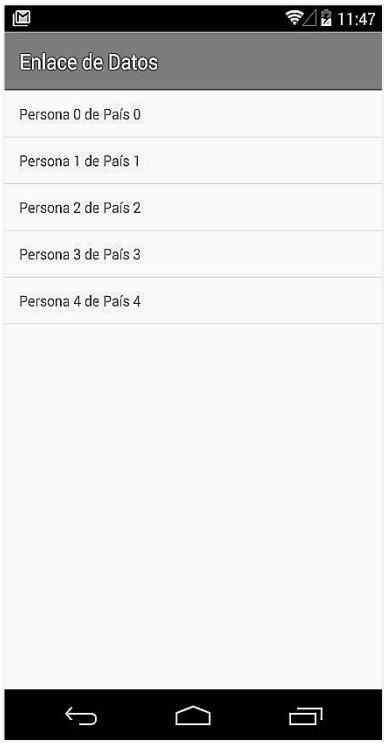
```
    …
    <ContentPage.Resources>
      <ResourceDictionary>
        <local:Datos x:Key="datos1" />
      </ResourceDictionary>
    </ContentPage.Resources>


    <ListView ItemsSource="{Binding Personas,
Source={StaticResource datos1}}" />
```

Si ejecutas en este momento la aplicación, te darás cuenta de que efectivamente el ListView es enlazado a la colección, pero cada elemento en la lista muestra el

nombre totalmente cualificado del objeto Persona. Este es el comportamiento predeterminado del control ListView ya que no hemos asignado ninguna Plantilla de Datos (que veremos más adelante), sin embargo, lo solucionaremos rápidamente reemplazando el método ToString() en la clase Persona, para que devuelva la concatenación de las propiedades “Nombre” y “Pais”:

```
public override string ToString()
{
    return $"{Nombre} de {Pais}";
}
```

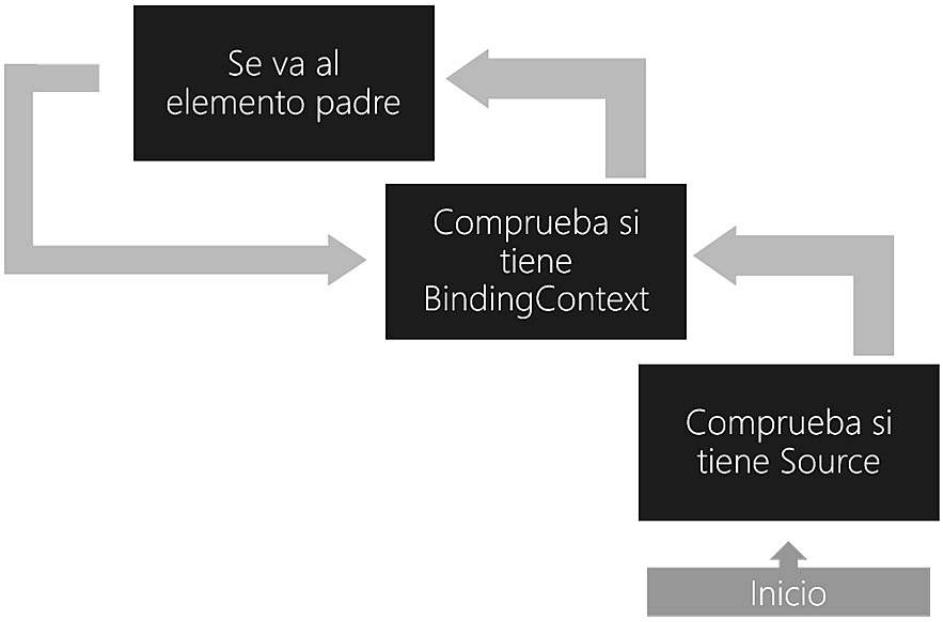
Ahora sí, si ejecutamos nuevamente la aplicación, te darás cuenta de que el control ListView ha sido enlazado a la colección y que además cada elemento muestra en texto la concatenación de ambas propiedades de la clase Persona. La siguiente figura muestra la aplicación que se está procesando en Android:



Contexto de Enlace de Datos

Cuando tenemos diversos controles enlazados a un mismo objeto, resulta ineficiente estar repitiendo el mismo valor para la fuente de datos. Debido a ello, la clase BindingObject expone la propiedad BindingContext, la cual representa un contexto de enlace que deseamos establecer para un elemento contenedor y sus hijos. De esa manera, todas las expresiones de {Binding} que estén dentro de dicho contenedor podrán omitir la propiedad Source, y en vez de ello usar automáticamente el objeto establecido en la propiedad BindingContext del elemento padre.

La siguiente figura muestra la secuencia de pasos que usará una expresión de enlace de datos para determinar su fuente:



Por ejemplo, recordemos el código que tenemos para enlistar las personas en un control ListView:

```
<ContentPage.Resources>
  <ResourceDictionary>
    <local:Datos x:Key="datos1" />
  </ResourceDictionary>
</ContentPage.Resources>
```

```
  <Grid>
    <ListView ItemsSource="{Binding Personas,
Source={StaticResource datos1}}" />
  <Grid />
```

Este código podríamos modificarlo para establecer el contexto de enlace de datos en el Grid, en vez de explícitamente establecer la propiedad Source en el ListView tal y como lo muestra el siguiente fragmento de código:

```
  <Grid BindingContext="{Binding Source={StaticResource
datos1}}">
    <ListView ItemsSource="{Binding Personas}" />
  </Grid>
```

Es importante destacar que en la expresión de enlace para la propiedad BindingContext no estamos usando Path, sino que estamos usando todo el objeto complejo como fuente. Asimismo, en la expresión de enlace de la propiedad ItemsSource en el control ListView, no estamos usando la propiedad Source ya que estamos indicando que el enlace use el objeto establecido como contexto de enlace de datos en el Grid.

OBTENIENDO EL ELEMENTO SELECCIONADO DE UN LISTVIEW

La infraestructura de enlace de datos en Xamarin.Forms es sumamente poderosa. Aprovechando que ahora hemos establecido el contexto de enlace de datos, agregaremos en la clase Datos una nueva propiedad llamada PersonaSeleccionada, la cual recibirá el objeto seleccionado en la lista.

```
private Persona personaSeleccionada;
public Persona PersonaSeleccionada
{
    get
    {
        return personaSeleccionada;
    }
    set
    {
        if (personaSeleccionada == value)
        {
```

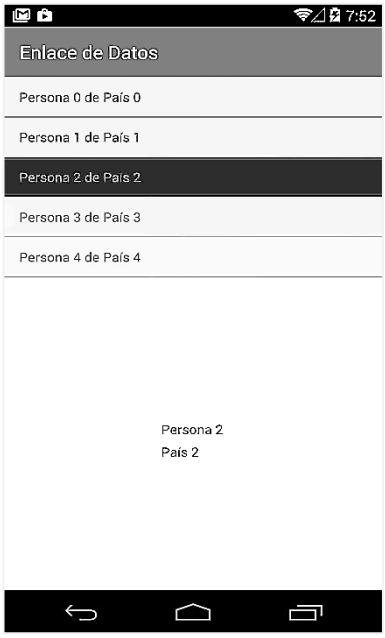


```
                return;
            }
            personaSeleccionada = value;
            OnPropertyChanged();
        }
    }
```

Para poder "empujar" de regreso al objeto Datos la persona seleccionada en la lista, enlazaremos la propiedad SelectedItem del ListView a esta nueva propiedad llamada PersonaSeleccionada, pero esta vez usando el modo bi-direccional (TwoWay). Adicionalmente, agregaremos un par de controles tipo Label que nos permitan mostrar el nombre y el país de la persona seleccionada. El siguiente fragmento de código muestra los cambios requeridos en XAML:

```
  <Grid BindingContext="{Binding Source={StaticResource
datos1}}">
    <Grid.RowDefinitions>
      <RowDefinition Height="2*" />
      <RowDefinition />
    </Grid.RowDefinitions>

    <ListView ItemsSource="{Binding Personas}"
              SelectedItem="{Binding PersonaSeleccionada,
Mode=TwoWay}" />

    <StackLayout Grid.Row="1"
                 BindingContext="{Binding
PersonaSeleccionada}"
                 HorizontalOptions="Center">
      <Label Text="{Binding Nombre}" />
      <Label Text="{Binding Pais}" />
    </StackLayout>
  </Grid>
```

La siguiente figura muestra la ejecución de la aplicación en Android, una vez implementadas las modificaciones anteriores:



Enlace entre elementos

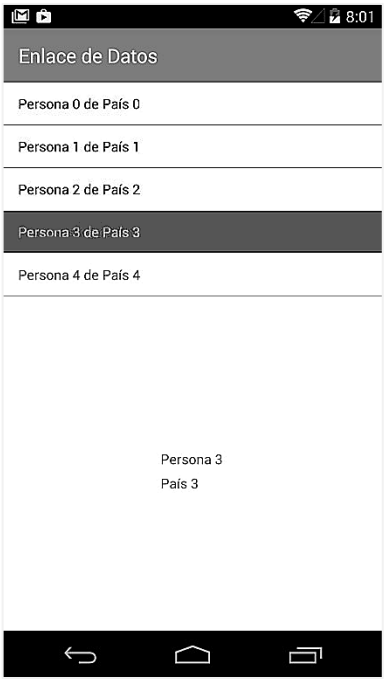
Además de enlazar un control de tipo BindableObject a una fuente de datos como lo hemos estado mostrando en todo este capítulo, también podemos enlazar las propiedades de dos BindableObject entre sí. Para lograr esto, usamos como fuente de datos un objeto que nos devuelva la extensión de marcado {x:Reference}. En esta extensión de marcado indicamos el nombre del control que deseamos usar como fuente, por lo tanto, dicho control previamente deberá haber establecido su nombre por medio del atributo x:Name.

Por ejemplo, en el siguiente fragmento de código modificamos la propiedad BindingContext del StackLayout, para referenciar la propiedad SelectedItem del ListView, control al que le hemos puesto el nombre “ListaPersonas” por medio del atributo x:Name:

```
   <ListView x:Name="ListaPersonas"
             ItemsSource="{Binding Personas}"
             SelectedItem="{Binding PersonaSeleccionada,
Mode=TwoWay}" />
```

```
  <StackLayout Grid.Row="1"
               BindingContext="{Binding SelectedItem,
Source={x:Reference ListaPersonas}}"
               HorizontalOptions="Center">
    <Label Text="{Binding Nombre}" />
    <Label Text="{Binding Pais}" />
  </StackLayout>
```

El resultado de esta modificación hace que la aplicación se comporte exactamente igual que antes:



Entonces, si el resultado es el mismo, tal vez te preguntarás: ¿cuándo usar la primera técnica (enlazar el elemento seleccionado a una propiedad del objeto que ejerce como el contexto de enlace de datos)? Y ¿cuándo usar la segunda técnica (usar el enlace entre elementos, usando la extensión de marcado {x:Reference})? La respuesta estará en función de si necesitas conocer el valor en tu objeto de contexto de enlace o no. Por ejemplo, para realizar alguna validación del elemento seleccionado probablemente la primera técnica sea más adecuada. Por otro lado, si

simplemente estás conectando dos elementos visuales entre sí sin una intención adicional de validación o reglas de negocio, probablemente la segunda técnica sea más sencilla de implementar.

Propiedad StringFormat

La clase Binding en Xamarin.Forms incluye la propiedad StringFormat, la cual nos permite establecer un formato para presentar correctamente un tipo de dato en cadena. Un ejemplo clásico de esta necesidad es cuando requerimos mostrar en pantalla el valor de un número decimal o el valor de una fecha. StringFormat soporta los mismos formatos que el método Format() de la clase String del .NET Framework. Para mayor información, puedes consultar "String.Format" en el sitio de MSDN.

Para demostrar la funcionalidad de esta propiedad, agregaremos a la clase Persona dos nuevas propiedades llamadas FechaNacimiento de tipo DateTime y Saldo de tipo decimal, tal y como lo muestra el siguiente fragmento de código:

```
private DateTime fechaNacimiento;
public DateTime FechaNacimiento
{
    get
    {
        return fechaNacimiento;
    }
    set
    {
        if (fechaNacimiento == value)
        {
            return;
        }
        fechaNacimiento = value;
        OnPropertyChanged();
    }
}
private decimal saldo;
```

```
public decimal Saldo
{
    get
    {
        return saldo;
    }
    set
    {
        if (saldo == value)
        {
            return;
        }
        saldo = value;
        OnPropertyChanged();
    }
}
```

Adicionalmente, modificaremos el método ToString() de la clase Persona para incluir el valor de las nuevas propiedades:

```
public override string ToString()
{
    return
$"{Nombre}|{Pais}|{FechaNacimiento}|{Saldo}";
}
```

Asimismo, modificaremos el constructor de la clase Datos, para poner algunos valores en las nuevas propiedades de cada objeto de tipo Persona que agreguemos a la colección:

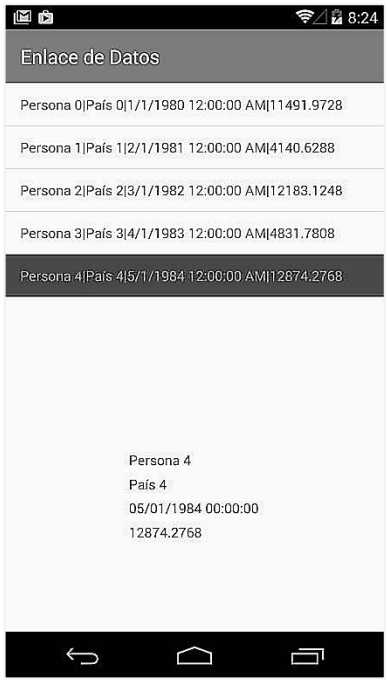
```
public Datos()
{
    Personas = new ObservableCollection<Persona>();
    var random = new Random();
```

```
        for (int i = 0; i < 5; i++)
        {
            Personas.Add(new Persona()
            {
                Nombre = $"Persona {i}",
                Pais = $"País {i}",
                FechaNacimiento = new DateTime(1980 + i, i
+ 1, 1),
                Saldo = (decimal)(random.Next(100, 5000) *
3.1416)
            });
        }
    }
```

Finalmente, modificamos el código de XAML para incluir las nuevas propiedades en el StackLayout que muestra los datos de la persona seleccionada:

```
    <StackLayout Grid.Row="1"
                 BindingContext="{Binding
PersonaSeleccionada}"
                 HorizontalOptions="Center">
      <Label Text="{Binding Nombre}" />
      <Label Text="{Binding Pais}" />
      <Label Text="{Binding FechaNacimiento}" />
      <Label Text="{Binding Saldo}" />
    </StackLayout>
```

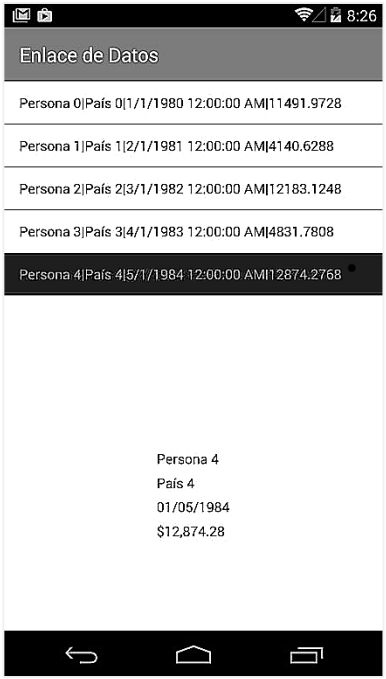
Si ejecutamos la aplicación en este momento, podremos observar que las nuevas propiedades son mostradas en cada elemento dentro de la lista. Recuerda que el motivo por el cual estamos viendo cada objeto como un texto es porque la infraestructura de enlace de datos invocará el método ToString() de cada objeto, ya que aún no tenemos asignada una Plantilla de Datos. Por otro lado, al seleccionar una persona de la lista sus datos se muestran en la parte de abajo, sin embargo, ambas propiedades FechaNacimiento y Saldo se muestran sin formato alguno.



Para corregir el formato de ambas etiquetas, modificaremos las expresiones de enlace de datos asignando la propiedad StringFormat como a continuación se muestra:

```
  <StackLayout Grid.Row="1"
               BindingContext="{Binding
PersonaSeleccionada}"
               HorizontalOptions="Center">
    <Label Text="{Binding Nombre}" />
    <Label Text="{Binding Pais}" />
    <Label Text="{Binding FechaNacimiento,
StringFormat='{}{0:dd/MM/yyyy}'}" />
    <Label Text="{Binding Saldo, StringFormat='{}{0:C}'}"
/>
  </StackLayout>
```

La siguiente figura muestra la aplicación después de haber implementado los anteriores cambios en el código XAML:



¡Mucho mejor! No obstante, aún quedan algunas cosas por hacer ya que la lista aún no muestra los datos formateados. A continuación corregiremos eso a través de la creación de Plantillas de Datos.

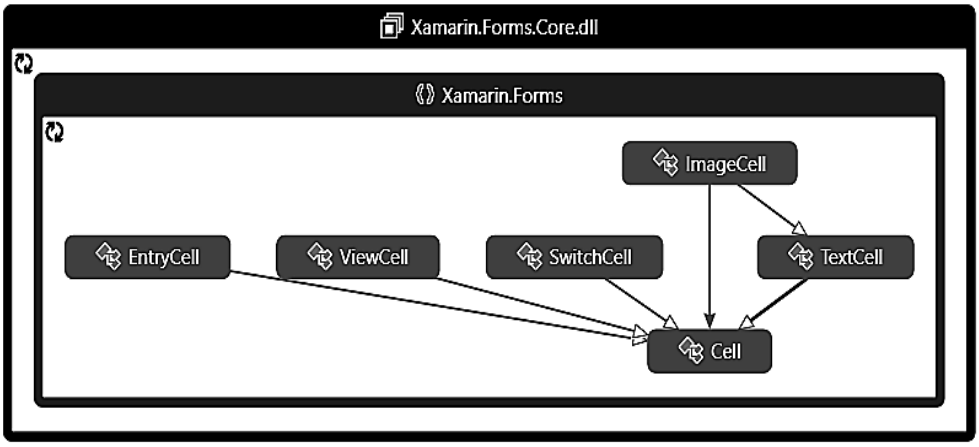
PLANTILLAS DE DATOS

Las Plantillas de Datos expresan cómo queremos que se vea cada elemento dentro de un control de lista, por ejemplo el control ListView. El control ListView tiene la propiedad ItemTemplate en donde podemos asignar una plantilla para modificar la visualización de cada elemento cuando se presenta en pantalla.

Las Plantillas de Datos son objetos de tipo DataTemplate, en donde su elemento raíz debe ser de algún tipo Celda (Cell). Estos DataTemplate pueden ser asignados usando la sintaxis de subelementos en XAML, o podemos agregarlos a un diccionario de recursos y referenciarlos de manera estática o dinámica usando {StaticResource} o {DynamicResource} respectivamente.

Tipos de celdas

Las celdas son clases que heredan de la clase base Cell. El siguiente diagrama muestra la jerarquía de clases de tipo Cell disponibles en el API de Xamarin.Forms:



La siguiente tabla describe los diferentes tipos de celdas:

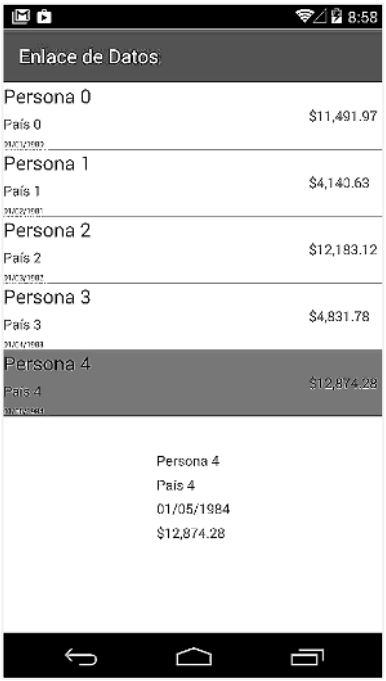
EntryCell	Muestra una caja de texto y un texto
ImageCell	Muestra una imagen y un texto
SwitchCell	Muestra un control de tipo Switch y un texto
TextCell	Muestra dos textos
ViewCell	Celda que nos permite establecer cualquier estructura de elementos en XAML de manera arbitraria

Como podemos observar la tabla anterior, la celda más flexible es ViewCell y es justamente la que utilizaremos para mejorar la lista de personas en nuestra aplicación de ejemplo.

El siguiente fragmento de código muestra la creación de una Plantilla de Datos llamada PersonaDataTemplate, cuyo elemento raíz es de tipo ViewCell. Dentro de la celda usamos diferentes etiquetas enlazadas con los valores de cada objeto Persona. Posteriormente, referenciamos esta Plantilla de Datos en el control ListView a través de la extensión de marcado {StaticResource}. También es importante destacar el uso de la propiedad HasUnevenRows en el control ListView, ya que es requerida en Android para poder permitir que la altura de una celda crezca adecuadamente, de lo contrario los elementos visuales se cortarían.

```
  <DataTemplate x:Key="PersonaDataTemplate">
    <ViewCell>
      <Grid>
        <Grid.ColumnDefinitions>
          <ColumnDefinition Width="4*" />
          <ColumnDefinition />
        </Grid.ColumnDefinitions>

        <StackLayout>
          <Label Text="{Binding Nombre}"
                 FontSize="20" />
          <Label Text="{Binding Pais}" />
          <Label Text="{Binding FechaNacimiento,
StringFormat='{}{0:dd/MM/yyyy}'}"
                 FontSize="8" />
        </StackLayout>

        <Label Text="{Binding Saldo,
StringFormat='{}{0:C}'}"
               Grid.Column="1"
               VerticalTextAlignment="Center" />
      </Grid>
    </ViewCell>
  </DataTemplate>
  …
  <ListView ItemsSource="{Binding Personas}"
            SelectedItem="{Binding PersonaSeleccionada,
Mode=TwoWay}"
            ItemTemplate="{StaticResource
PersonaDataTemplate}"
            HasUnevenRows="True" />
```

Si ejecutamos la aplicación una vez efectuados estos cambios, veremos que ahora los elementos en la lista se muestran de una forma más enriquecida. Las siguientes figuras muestran la aplicación ejecutando en Android y en UWP:

Android	UWP
	

CONVERTIDORES DE VALOR

Como mencionamos al inicio de este capítulo, en una expresión de enlace de datos podemos asignar un Convertidor de Valor para ejecutar alguna lógica adicional durante el proceso de enlace. Los Convertidores de Valor son clases que implementan la interfaz IValueConverter, la cual incluye los siguientes dos miembros:

Convert()	Método que se ejecuta cuando el valor fluye del Objeto fuente hacia el Control destino
ConvertBack()	Método que se ejecuta cuando el valor fluye de regreso, del Control destino hacia el Objeto fuente

Ambos métodos tienen los siguientes argumentos:

value	Argumento de tipo object que representa el valor de entrada
targetType	Argumento de tipo Type que representa el tipo de dato de la propiedad a la que está enlazada la expresión de enlace que está usando el Convertidor de Valor
parameter	Argumento de tipo object que representa un parámetro adicional arbitrario para el Convertidor de Valor
culture	Argumento de tipo CultureInfo que representa la información de cultura que tiene en ese momento el dispositivo donde está ejecutándose la aplicación. Muy útil cuando vas a devolver una cadena que esté en función de la configuración regional del dispositivo.

Creando un Convertidor de Valor

Para crear un Convertidor de Valor, creamos una clase que implemente la interfaz IValueConverter e incluimos la lógica deseada en uno o ambos métodos, según requiera la aplicación en cuestión. El siguiente fragmento de código muestra la creación de un Convertidor de Valor llamado SaldoConverter, el cual evaluará el saldo de una persona y regresar a un color en función de esa cantidad.

```
public class SaldoConverter : IValueConverter
{
    public object Convert(object value, Type
targetType, object parameter, CultureInfo culture)
    {
        var saldo = (decimal)value;
        var color = Color.Green;

        if (saldo >= 5000 && saldo <= 10000)
        {
            color = Color.Orange;
        }
        if (saldo > 10000)
        {
            color = Color.Red;
```



```
            }

            return color;
        }
        public object ConvertBack(object value, Type
targetType, object parameter, CultureInfo culture)
        {
            throw new NotImplementedException();
        }
    }
```

Usando un Convertidor de Valor

Para usar un Convertidor de Valor, debemos instanciarlo en un diccionario de recursos. Posteriormente, lo referenciamos en la propiedad Converter de la expresión de Binding. Opcionalmente y como lo explicamos con anterioridad, también podemos pasarle parámetros al convertidor. Esto lo hacemos a través de la propiedad ConverterParameter de la clase Binding.

El siguiente fragmento de código muestra la instanciación del Convertidor de Valor en el diccionario de recursos. Adicionalmente, podemos observar cómo se está referenciando en la Plantilla de Datos para poder cambiar el color del texto del saldo de una persona en función del valor de esta propiedad.

```
<ResourceDictionary>
    <local:Datos x:Key="datos1" />

    <local:SaldoConverter x:Key="SaldoConverter" />

    <DataTemplate x:Key="PersonaDataTemplate">
      <ViewCell>
        <Grid>
          <Grid.ColumnDefinitions>
            <ColumnDefinition Width="4*" />
            <ColumnDefinition />
          </Grid.ColumnDefinitions>
```

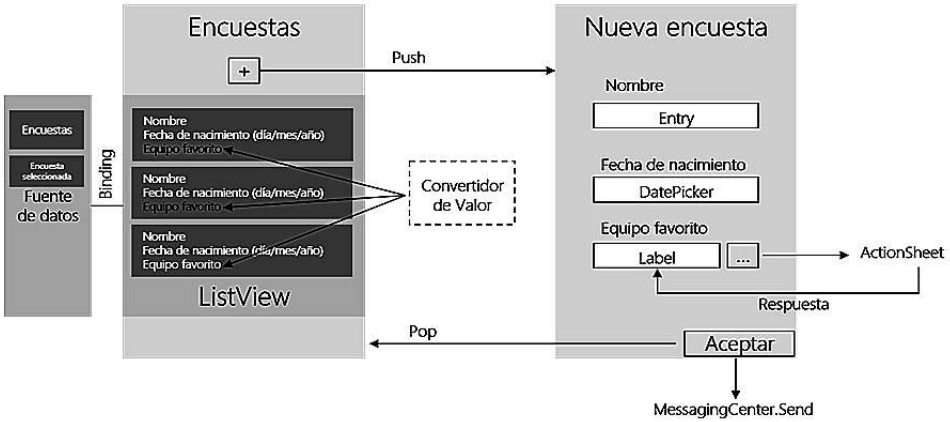
```
          <StackLayout>
            <Label Text="{Binding Nombre}"
                   FontSize="20" />
            <Label Text="{Binding Pais}" />
            <Label Text="{Binding FechaNacimiento,
StringFormat='{}{0:dd/MM/yyyy}'}"
                   FontSize="8" />
          </StackLayout>


          <Label Text="{Binding Saldo,
StringFormat='{}{0:C}'}"
                 Grid.Column="1"
                 VerticalTextAlignment="Center"
        TextColor="{Binding Saldo,
Converter={StaticResource SaldoConverter}}" />
        </Grid>
      </ViewCell>
    </DataTemplate>
  </ResourceDictionary>
```

Al ejecutar la aplicación, podremos observar que el saldo de cada persona tendrá un color en función de la lógica que implementamos en SaldoConverter:

MANOS A LA OBRA

Durante este capítulo realizaremos las siguientes mejoras y modificaciones a nuestra aplicación de encuestas:

- Crearemos un objeto capaz de notificar el cambio de valor en sus propiedades, y que tenga como responsabilidad ser la fuente de datos de la vista principal de encuestas
- Sustituiremos el StackLayout que muestra las encuestas capturadas por un ListView que tenga una Plantilla de Datos y haga uso de un Convertidor de Valor para cambiar el color del texto del equipo favorito



¡Comencemos!

Creando la fuente de datos

En el proyecto PCL, crearemos una nueva clase abstracta llamada NotificationObject. Esta clase implementará la interfaz INotifyPropertyChanged y expondrá un método llamado OnPropertyChanged() el cual será el responsable de disparar el evento PropertyChanged cuando cambie el valor de alguna propiedad en las clases derivadas. El siguiente fragmento de código muestra la implementación completa de esta clase:

```
  public abstract class NotificationObject :
INotifyPropertyChanged
  {
      public event PropertyChangedEventHandler
PropertyChanged;


      protected virtual void
OnPropertyChanged([CallerMemberName] string propertyName
= null)
      {
          PropertyChanged?.Invoke(this, new
PropertyChangedEventArgs(propertyName));
      }
  }
```


Igualmente en el proyecto PCL, agregaremos una nueva clase llamada Data, que tendrá la responsabilidad de ser la fuente de datos para la página SurveysView. Esta clase heredará de NotificationObject e implementaremos un par de propiedades como a continuación se describen:

Surveys	ObservableCollection<Survey>	Colección que guardará las encuestas capturadas
SelectedSurvey	Survey	Propiedad para guardar la encuesta seleccionada en la lista

Adicionalmente, en el constructor de la clase Data implementaremos la lógica de suscripción al mensaje "NewSurveyComplete" que en el capítulo anterior dejamos en el archivo de code-behind de la página SurveysView. De hecho, esa implementación la debemos borrar una vez implementada la lógica en el constructor de la clase Data.

El siguiente fragmento de código muestra la implementación completa de la clase Data:

```
using System.Collections.ObjectModel;
using Xamarin.Forms;

namespace Surveys.Core
{
    public class Data : NotificationObject
    {
        private ObservableCollection<Survey> surveys;

        public ObservableCollection<Survey> Surveys
        {
            get
            {
                return surveys;
            }
            set
            {
```

```
            if (surveys == value)
            {
                return;
            }
            surveys = value;
            OnPropertyChanged();
        }
    }
    private Survey selectedSurvey;

    public Survey SelectedSurvey
    {
        get
        {
            return selectedSurvey;
        }
        set
        {
            if (selectedSurvey == value)
            {
                return;
            }
            selectedSurvey = value;
            OnPropertyChanged();
        }
    }

    public Data()
    {
```

```
                Surveys = new
ObservableCollection<Survey>();
                MessagingCenter.Subscribe<ContentPage,
Survey>(this, Messages.NewSurveyComplete,
                    (sender, args) => { Surveys.Add(args);
});
            }
        }
    }
```

En el código XAML de la página SurveysView, agregamos una nueva instancia de Data en el diccionario de recursos de la página y lo establecemos como contexto de enlace de datos en el Grid raíz. Posteriormente, borramos el elemento StackLayout y lo sustituimos por un control de tipo ListView, el cual estará enlazado a la colección "Surveys" a través de su propiedad ItemsSource. Además, la propiedad SelectedItem del ListView estará enlazada de manera bidireccional a la propiedad "SelectedSurvey" de la fuente de datos, para poder rastrear en dicha propiedad la encuesta seleccionada en la lista.

El siguiente fragmento de código muestra el código XAML de la página una vez implementadas estas modificaciones:

```
    <?xml version="1.0" encoding="utf-8" ?>
    <ContentPage
xmlns="http://xamarin.com/schemas/2014/forms"

xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
                 xmlns:core="clr-
namespace:Surveys.Core;assembly=Surveys.Core"
                 x:Class="Surveys.Core.SurveysView"
                 Title="Encuestas">
      <ContentPage.Resources>
        <ResourceDictionary>
          <core:Data x:Key="data" />
        </ResourceDictionary>
      </ContentPage.Resources>
```

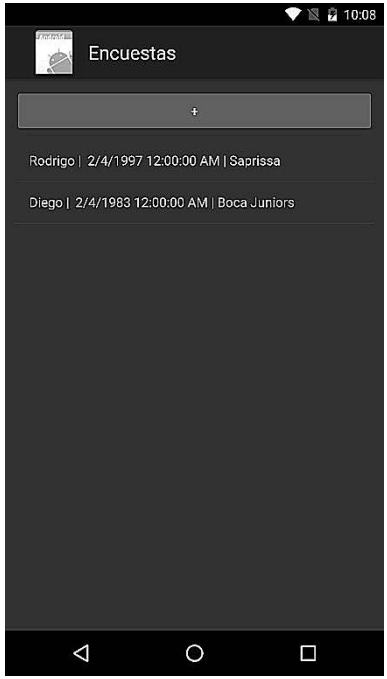
```
    <Grid Margin="10"
          BindingContext="{Binding Source={StaticResource
data}}">
      <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition />
      </Grid.RowDefinitions>

      <Button Text="+"
              Clicked="AddSurveyButton_Clicked" />

      <ListView Grid.Row="1"
                ItemsSource="{Binding Surveys}"
                SelectedItem="{Binding SelectedSurvey,
Mode=TwoWay}" />
    </Grid>
  </ContentPage>
```

Y el archivo de code-behind:

```
using System;
using Xamarin.Forms;

namespace Surveys.Core
{
    public partial class SurveysView : ContentPage
    {
        public SurveysView()
        {
            InitializeComponent();
        }
```

```
        private async void
AddSurveyButton_Clicked(object sender, EventArgs e)
        {
            await Navigation.PushAsync(new
SurveyDetailsView());
        }
    }
}
```

Si ejecutamos la aplicación en este punto, podemos apreciar que la funcionalidad de agregar encuestas sigue completamente vigente, ¡pero ahora sí estamos usando el control adecuado para enlistarlas!



Implementando la Plantilla de Datos

El hecho de que cada encuesta se vea como un texto, hace que la aplicación en este momento esté muy limitada visualmente hablando. Por lo tanto, implementaremos una nueva Plantilla de Datos que incluya la información de cada encuesta pero de una manera mucho más enriquecida que un simple texto.

En el diccionario de recursos de la página SurveysView, agregaremos un nuevo DataTemplate llamado “SurveyDataTemplate”, el cual tendrá como raíz una Celda de tipo ViewCell, ya que usaremos su flexibilidad para usar cualquier estructura de XAML arbitrario para presentar la celda. Una vez implementada la Plantilla de Datos, la referenciaremos de manera estática en la propiedad ItemTemplate del control ListView y además estableceremos la propiedad HasUnevenRows para que pueda renderizarse correctamente la celda en los dispositivos Android.

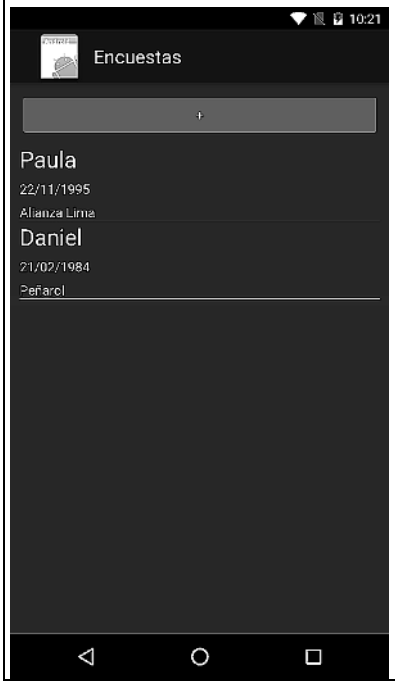
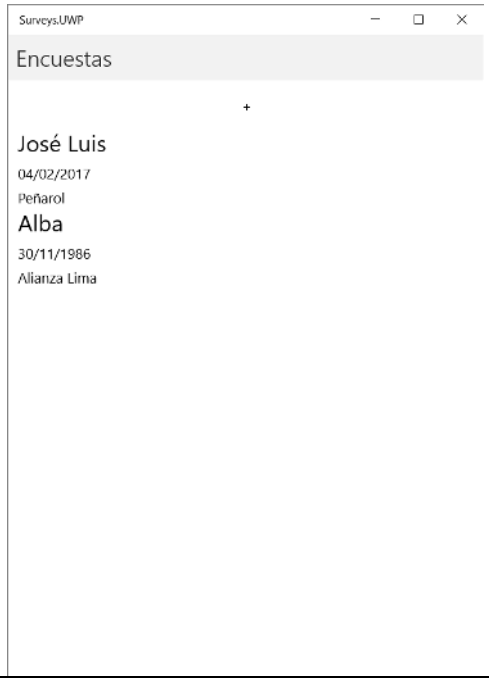
El siguiente fragmento de código muestra la implementación de la Plantilla de Datos, y su referencia en el control ListView:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage
xmlns="http://xamarin.com/schemas/2014/forms"

xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:core="clr-
namespace:Surveys.Core;assembly=Surveys.Core"
             x:Class="Surveys.Core.SurveysView"
             Title="Encuestas">
  <ContentPage.Resources>
    <ResourceDictionary>
      <core:Data x:Key="data" />

      <DataTemplate x:Key="SurveyDataTemplate">
        <ViewCell>
          <StackLayout>
            <Label Text="{Binding Name}"
                   FontSize="24" />
            <Label Text="{Binding Birthdate,
StringFormat='{}{0:dd/MM/yyyy}'}" />
            <Label Text="{Binding FavoriteTeam}" />
          </StackLayout>
        </ViewCell>
      </DataTemplate>
```



```
      </ResourceDictionary>
    </ContentPage.Resources>

    <Grid Margin="10"
          BindingContext="{Binding Source={StaticResource
data}}">
      <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition />
      </Grid.RowDefinitions>

      <Button Text="+"
              Clicked="AddSurveyButton_Clicked" />

      <ListView Grid.Row="1"
                ItemsSource="{Binding Surveys}"
                SelectedItem="{Binding SelectedSurvey,
Mode=TwoWay}"
                ItemTemplate="{StaticResource
SurveyDataTemplate}"
                HasUnevenRows="True" />
    </Grid>
  </ContentPage>
```

Las siguientes figuras muestran la aplicación ejecutándose en Android y UWP respectivamente:

Android	UWP
	

Implementando el Convertidor de Valor

El objetivo de la aplicación es mostrar de un color diferente —representativo de cada uno— el equipo favorito en cada encuesta capturada. Por lo tanto, en la PCL crearemos una nueva clase llamada TeamColorConverter, la cual implementará la interfaz IValueConverter y tendrá la lógica de convertir un nombre de equipo a un objeto de tipo Color.

El siguiente fragmento de código muestra la implementación completa de la clase TeamColorConverter:

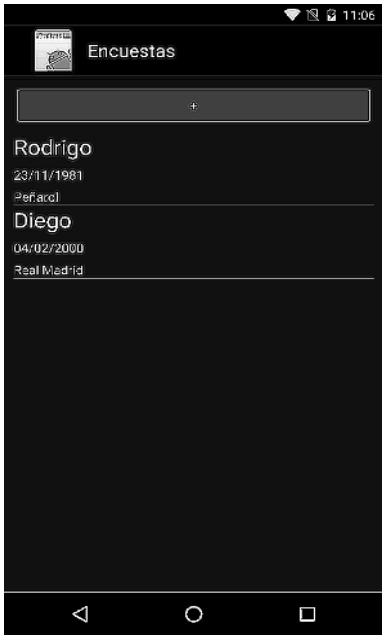
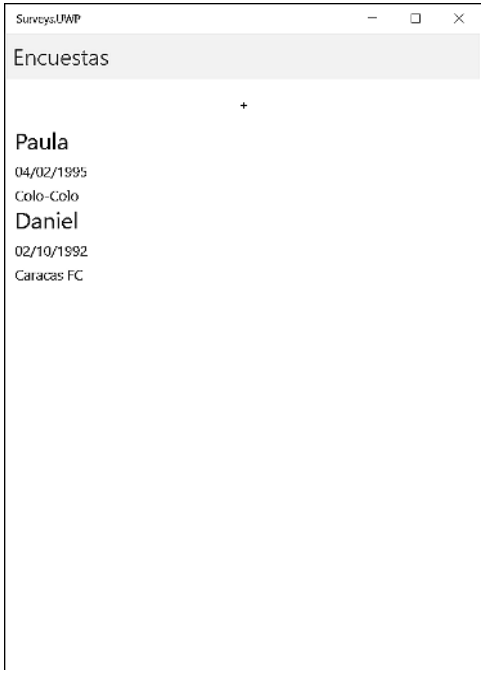
```
using System;
using System.Globalization;
using Xamarin.Forms;

namespace Surveys.Core
{
    public class TeamColorConverter : IValueConverter
```

```
    {
        public object Convert(object value, Type
targetType, object parameter, CultureInfo culture)
        {
            if (value == null)
            {
                return null;
            }

            var team = (string)value;
            var color = Color.Transparent;
            switch (team)
            {
                case "América":
                case "Peñarol":
                    color = Color.Yellow;
                    break;
                case "Boca Juniors":
                case "Colo-Colo":
                case "Alianza Lima":
                    color = Color.Blue;
                    break;
                case "Caracas FC":
                case "Saprissa":
                    color = Color.Purple;
                    break;
                case "Real Madrid":
                    color = Color.Fuchsia;
                    break;
            }

            return color;
        }
```

```
            public object ConvertBack(object value, Type
targetType, object parameter, CultureInfo culture)
            {
                throw new NotImplementedException();
            }
        }
    }
```

Nota: Como mencioné con anterioridad, el uso indiscriminado de "cadenas mágicas" como las utilizadas en el bloque switch no es muy buena idea. No obstante, en capítulos posteriores regresaremos a este código para corregirlo a través del uso de una entidad para los equipos que incluirá además del nombre, el color del equipo.

Si ejecutamos la aplicación, ahora los textos con los nombres de los equipos favoritos tendrán su color representativo, según la lógica implementada en el Convertidor de Valor. Las siguientes figuras muestran la aplicación ejecutándose en Android y UWP:

Android	UWP
	

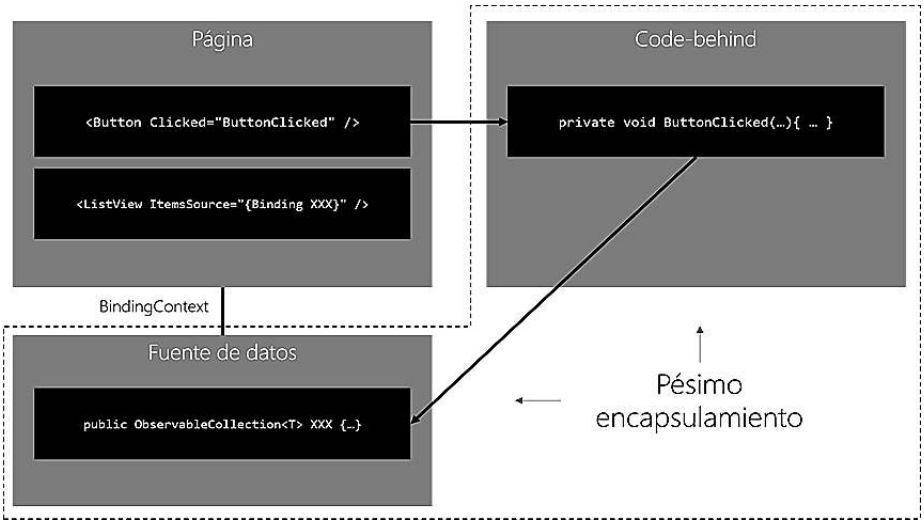
Nota: Debido a la impresión en papel del libro, probablemente no se alcance a diferenciar los colores de los equipos, por lo tanto ¡es buen pretexto para que sigas la implementación del código tú mismo(a)!

6 COMANDOS

INTRODUCCIÓN

La intención de toda aplicación basada en tecnologías XAML será siempre buscar la separación de preocupaciones entre los diferentes objetos que la constituyen. Sin embargo, nos encontramos un obstáculo cuando llega el momento de escribir los manejadores de eventos de los controles interactivos como los botones ya que estamos destinados a escribir dichos manejadores en la clase de code-behind de las páginas.

Este escenario se complica aún más cuando el manejador del evento en el code-behind requiere acceder a las propiedades del objeto que ejerce como contexto de enlace de datos de la página. Tal y como lo muestra la siguiente figura:



Esta falta de encapsulamiento correcto es un nido de múltiples problemas, entre los que destacan el costo de mantenimiento del código y el no poder escribir Pruebas Unitarias de una manera adecuada a nuestra clase de fuente de datos.

Los comandos resuelven justamente este problema ya que los podríamos definir como "funcionalidad enlazable". Los comandos son en realidad clases que implementan la interfaz ICommand, la cual proviene del espacio de nombres System.Windows.Input e incluye los siguientes tres miembros:

Execute()	Método que indica qué acción deseamos ejecute el comando
CanExecute()	Método que determina si un comando puede ejecutar o no. Cuando se trata de controles visuales, si el método devuelve falso el control se deshabilitará automáticamente
CanExecuteChanged	Evento que al dispararse, reevalúa el método CanExecute()

Los comandos están soportados en algunos controles y gestos en Xamarin.Forms. Todas las clases que a continuación se enlistan exponen las propiedades Command y CommandParameter, para asignar un comando y opcionalmente un parámetro respectivamente:

- Button
- ToolbarItem
- SearchBar
- TapGestureRecognizer

Para poder usar un comando, enlazamos la propiedad Command a la propiedad de tipo ICommand de nuestro objeto que está ejerciendo como el contexto de enlace de datos del control en cuestión. Por ejemplo, el siguiente fragmento de código crea un botón que ejecuta un comando llamado AddPersonCommand, pasando como parámetro al comando el texto que tenga un Entry llamado entry1:

```
  <Button Text="Agregar"
          Command="{Binding AddPersonCommand}"
          CommandParameter="{Binding Text,
Source={x:Reference entry1}}" />
```

Creando un comando básico

Como mencioné con anterioridad, los comandos son clases que implementan la interfaz ICommand. Para comprender a detalle el funcionamiento de los comandos, vamos a implementar uno desde cero llamado MyCommand. Lo primero que

tenemos que hacer es implementar en la clase MyCommand la interfaz ICommand. El siguiente ejemplo muestra el código que deja Visual Studio .NET al implementar una interfaz usando su herramienta de refactoring:

```
public class MyCommand : ICommand
{
    public bool CanExecute(object parameter)
    {
        throw new NotImplementedException();
    }

    public void Execute(object parameter)
    {
        throw new NotImplementedException();
    }

    public event EventHandler CanExecuteChanged;
}
```

Si bien la intención de un comando es su "funcionalidad" a la que nos podemos enlazar, el comando no sería muy flexible si lo atamos a una sola funcionalidad o lógica. Por lo tanto, para permitir que el comando ejecute cualquier acción, independientemente de cuál sea esta, implementaremos un constructor que tenga como argumento un delegado de tipo Action y guardaremos esa referencia en un campo privado a nivel de clase:

```
private Action action = null;
public MyCommand(Action action)
{
    this.action = action;
}
```

Ya que estamos guardando el delegado en el campo privado, podemos simplemente ejecutarlo en el método Execute():

```
public void Execute(object parameter)
{
```

```
        action();
    }
```

En este momento, simplemente indicaremos que sí queremos que el comando se ejecute, por lo que devolveremos el valor true en el método CanExecute(). El código completo de esta implementación básica de un comando es el siguiente:

```
public class MyCommand : ICommand
{
    private Action action = null;

    public MyCommand(Action action)
    {
        this.action = action;
    }

    public bool CanExecute(object parameter)
    {
        return true;
    }

    public void Execute(object parameter)
    {
        action();
    }

    public event EventHandler CanExecuteChanged;
}
```

Para utilizar el comando MyCommand en un botón, expondremos una propiedad de tipo ICommand en nuestra clase Datos y le asignaremos en el constructor de la clase una instancia de MyCommand, pasando como parámetro la función que deseamos ejecute el comando cuando el botón sea pulsado. En este ejemplo, la propiedad se llama “AgregarPersonaCommand” y su funcionalidad será justamente

agregar nuevas personas a la colección de tipo ObservableCollection<Persona> llamada Personas.

```
    public ICommand AgregarPersonaCommand { get; set; }
  public Datos()
  {
      AgregarPersonaCommand = new
MyCommand(AddPersonaCommandExecute);
      …
  }
  private void AddPersonaCommandExecute()
  {
      Personas.Add(new Persona()
      {
          Nombre = Guid.NewGuid().ToString(),
          Pais = Guid.NewGuid().ToString(),
          FechaNacimiento = new DateTime(1980, 10, 10),
          Saldo = 0
      });
  }
```

Observa cómo en el anterior código se establece el método AddPersonaCommandExecute como acción a ejecutar, ya que el método cumple con la signatura del delegado Action. Justamente ese método se usará como parámetro en el constructor de la clase MyCommand y será lo que finalmente ejecute el método Execute().

Para usar el comando, enlazamos la propiedad Command de un botón a la propiedad AddPersonaCommand. Esto hace que ya sea innecesario manejar el evento Clicked del botón, encapsulando en una sola clase y de una mejor manera la funcionalidad buscada:

```
<Button Text="Agregar persona"
        Command="{Binding AgregarPersonaCommand}" />
```

Implementando CanExecute()

Ahora bien, el hecho de que el método CanExecute() esté devolviendo siempre true no está muy bien, porque hay veces que querrás que el comando solo se ejecute si se cumplen ciertas condiciones. Por este motivo, modificaremos nuestra implementación de MyCommand agregando un segundo constructor que pida además del Action a ejecutar un Func<bool> que determine si el comando se puede ejecutar o no. El objeto de tipo Func<bool> también lo guardaremos en un campo privado a nivel de clase para poder ejecutarlo y devolver su valor en el método CanExecute().

El siguiente fragmento de código muestra la implementación completa de MyCommand, ahora con el segundo constructor y corrigiendo el método CanExecute():

```
  public class MyCommand : ICommand
  {
      private Action action = null;
      private Func<bool> canExecute = null;

      public MyCommand(Action action)
      {
          this.action = action;
      }
      public MyCommand(Action action, Func<bool>
canExecute) : this(action)
      {
          this.canExecute = canExecute;
      }

      public bool CanExecute(object parameter)
      {
          return canExecute();
      }
      public void Execute(object parameter)
```



```
        {
            action();
        }

        public event EventHandler CanExecuteChanged;
    }
```

Para usar este segundo constructor, asignaremos un método que tenga la lógica para determinar si el comando se puede ejecutar o no, y que cumpla la signatura del delegado Func<bool>. En este ejemplo, nuestra lógica será que el comando solo se podrá ejecutar hasta llegar a 10 personas en la colección:

```
    private bool AddPersonaCommandCanExecute()
    {
        return Personas.Count < 10;
    }
```

Una vez implementado este método, solo es cuestión de modificar la línea que instancia MyCommand, para establecer este segundo parámetro:

```
    AgregarPersonaCommand = new
MyCommand(AddPersonaCommandExecute,
AddPersonaCommandCanExecute);
```

Si ejecutamos este ejemplo, podemos observar que aunque hayamos establecido el delegado para evaluar CanExecute(), esta lógica parecería que no se está evaluando, y es justamente porque aún nos falta una pieza en el rompecabezas: el evento CanExecuteChanged.

Implementando CanExecuteChanged

El evento CanExecuteChanged lo debemos disparar cuando deseemos que el método CanExecute() sea reevaluado. Por esta razón, implementaremos un nuevo método llamado RaiseCanExecuteChanged() en la clase MyCommand, para poder disparar el evento CanExecuteChanged.

El siguiente código muestra la implementación completa de MyCommand con este cambio:

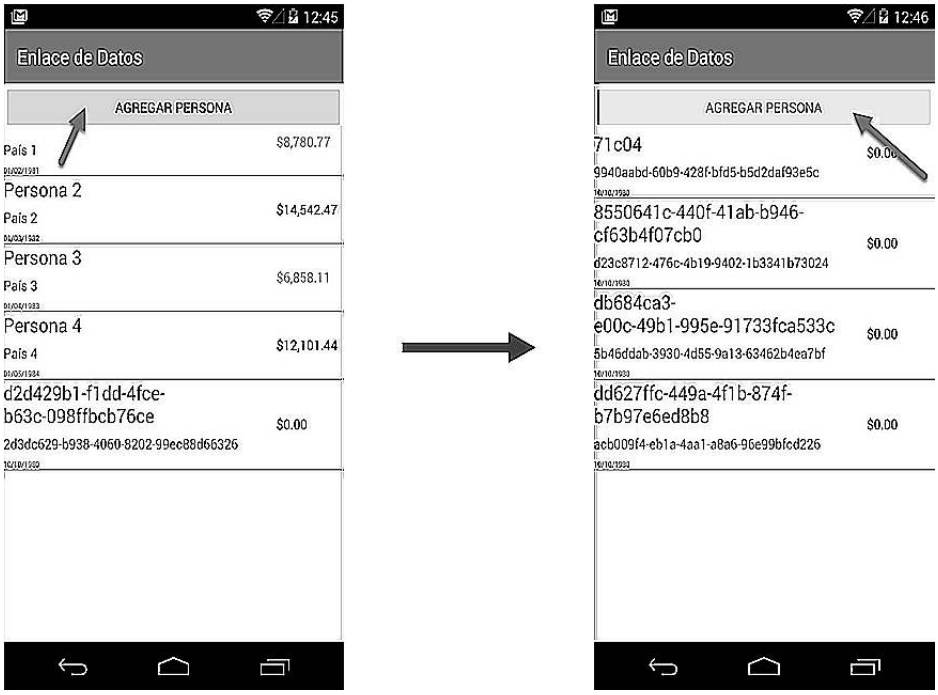
```
    public class MyCommand : ICommand
    {
```

```
        private Action action = null;
        private Func<bool> canExecute = null;

        public MyCommand(Action action)
        {
            this.action = action;
        }

        public MyCommand(Action action, Func<bool>
canExecute) : this(action)
        {
            this.canExecute = canExecute;
        }

        public bool CanExecute(object parameter)
        {
            return canExecute();
        }
        public void Execute(object parameter)
        {
            action();
        }

        public event EventHandler CanExecuteChanged;

        public void RaiseCanExecuteChanged()
        {
            CanExecuteChanged?.Invoke(this,
EventArgs.Empty);
        }
    }
```

Una vez implementado este método, es nuestra responsabilidad ejecutarlo cuando deseemos se reevalúe el método CanExecute() del comando. En este ejemplo, lo implementaremos justamente después de haber agregado el nuevo objeto de tipo Persona a la colección Personas:


```
    private void AddPersonaCommandExecute()
    {
        Personas.Add(new Persona()
        {
            Nombre = Guid.NewGuid().ToString(),
            Pais = Guid.NewGuid().ToString(),
            FechaNacimiento = new DateTime(1980, 10, 10),
            Saldo = 0
        });
        (AgregarPersonaCommand as
MyCommand)?.RaiseCanExecuteChanged();
    }
```

Si ejecutamos la aplicación en este momento, podemos comprobar que efectivamente la aplicación permitirá únicamente la creación de 10 personas, una vez cumplida esta lógica el botón será deshabilitado automáticamente. Las siguientes figuras muestran esta funcionalidad en Android:



Implementaciones existentes recomendadas

A este punto, tendrás un conocimiento y una comprensión mucho más a detalle de cómo funcionan los comandos. No obstante, la buena noticia es que generalmente no tienes que crear tus propios comandos ya que existen diversas implementaciones de estas clases que puedes utilizar en tus aplicaciones:

Xamarin.Forms	Command
Prism	DelegateCommand
MVVM Light	RelayCommand

Por ejemplo, para usar la clase Command de Xamarin.Forms en vez de nuestra clase MyCommand, simplemente modificamos el tipo que estamos instanciando en el constructor de la clase y modificamos el método que reevalúa el método CanExecute() ya que en la clase Command se llama ChangeCanExecute():

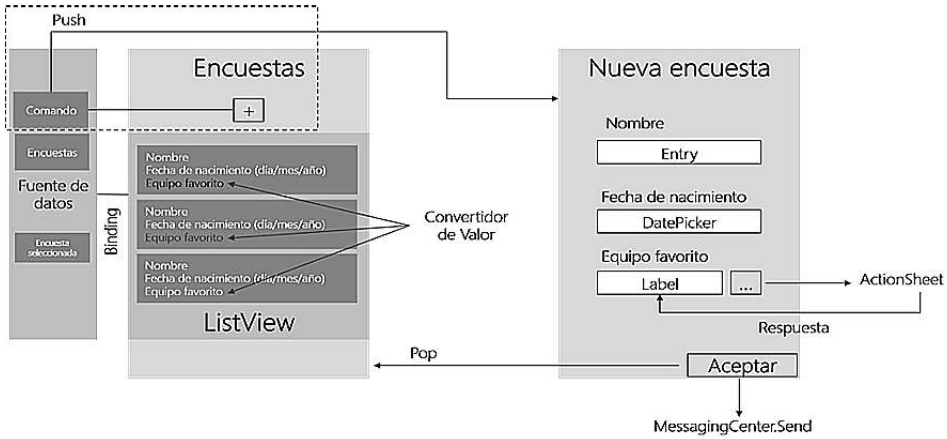
```
  AgregarPersonaCommand = new
Command(AddPersonaCommandExecute,
AddPersonaCommandCanExecute);

  …
  private void AddPersonaCommandExecute()
  {
      …
      (AgregarPersonaCommand as
Command)?.ChangeCanExecute();
  }
```

Si ejecutas la aplicación nuevamente, verás que la funcionalidad sigue siendo exactamente la misma.

MANOS A LA OBRA

En el capítulo anterior, nuestra aplicación de encuestas es capaz de capturar nuevos registros y enlistarlos en un ListView enlazado a un ObservableCollection<Survey> en el objeto Data. No obstante, la lógica del botón sigue implementada en un manejador de evento en el archivo de code-behind de la página SurveysView. En este capítulo corregiremos esa implementación ya que

agregaremos un comando a la clase Data tal y como lo muestra el siguiente diagrama:



Implementando el comando

En la clase Data, implementaremos una nueva propiedad de tipo ICommand llamada NewSurveyCommand. Esta propiedad la inicializaremos en el constructor de la clase y le asignaremos un nuevo objeto de tipo Xamarin.Forms.Command, el cual ejecutará un método llamado NewSurveyCommandExecute() ya que cumple con la signatura del delegado Action.

```
public ICommand NewSurveyCommand { get; set; }
public Data()
{
    NewSurveyCommand= new
Command(NewSurveyCommandExecute);
    …
}
private void NewSurveyCommandExecute()
{
}
```

En la página SurveysView, modificaremos la declaración del Button para enlazar su propiedad Command a la propiedad NewSurveyCommand del objeto Data, en vez de establecer el manejador del evento Clicked. El siguiente fragmento de código muestra la nueva implementación del botón:

```
  <Button Text="+" Command="{Binding NewSurveyCommand}"
/>
```

Ahora bien, ¿qué debemos implementar en el método NewSurveyCommandExecute()? El problema que se nos presenta en este momento es que el comando debe navegar a la página SurveyDetailsView… ¡pero en Xamarin.Forms la propiedad Navigation pertenece a la clase base Page!

La manera que nos permite implementar esta lógica reduciendo el acoplamiento entre ambos objetos SurveysView y Data es hacer uso de la clase MessagingCenter. De esta forma, el método NewSurveyCommandExecute() publicaría un nuevo mensaje al cual estaría suscrita la página SurveysView. Ya que se trata ahora de un nuevo mensaje, lo agregamos a la clase Messages para evitar el uso de "cadenas mágicas" en el código de C#:

```
  public class Messages
  {
      public const string NewSurveyComplete =
"NewSurveyComplete";
      public const string NewSurvey = "NewSurvey";
  }
```

El siguiente fragmento de código muestra la implementación del método NewSurveyCommandExecute():

```
  private void NewSurveyCommandExecute()
  {
      MessagingCenter.Send(this, Messages.NewSurvey);
  }
```

Finalmente, modificamos el archivo de code-behind de la página SurveysView para suscribirnos al mensaje NewSurvey, y en su callback navegamos a la página SurveyDetailsView. El siguiente fragmento de código muestra la implementación completa del archivo de code-behind de SurveysView:

```
  using Xamarin.Forms;
  namespace Surveys.Core
  {
      public partial class SurveysView : ContentPage
      {
```



```
        public SurveysView()
        {
            InitializeComponent();

            MessagingCenter.Subscribe<Data>(this,
Messages.NewSurvey, async (sender) =>
            {
                await Navigation.PushAsync(new
SurveyDetailsView());
            });
        }
    }
}
```

Si ejecutamos en este momento la aplicación, veremos que efectivamente el botón "+" hace uso del Comando, el cual a su vez manda un mensaje a través del MessagingCenter a SurveysView, para que sea esta página la que navegue a SurveyDetailsView.

¿Qué hay de la página SurveyDetailsView? En ella también tenemos implementado prácticamente todo en el archivo de code-behind. Ese código lo corregiremos en el siguiente capítulo después de que veamos el patrón de diseño Model-View-ViewModel.

7 EL PATRÓN DE DISEÑO MODEL-VIEW-VIEWMODEL

INTRODUCCIÓN

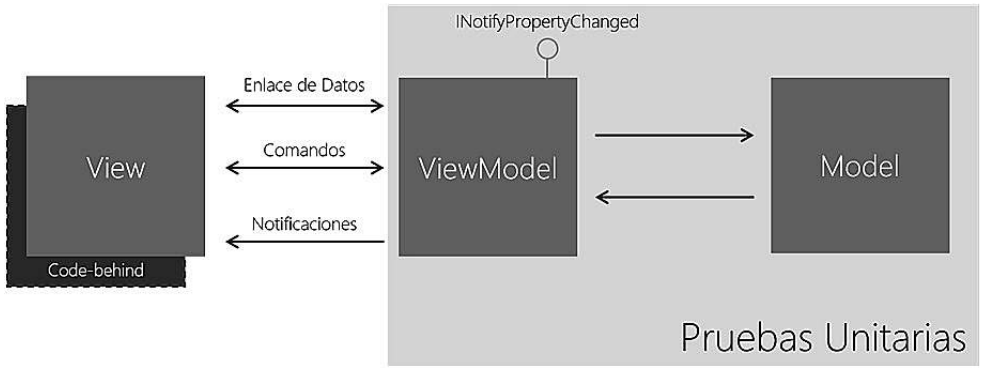
Construir soluciones profesionales altamente flexibles, robustas y de fácil mantenimiento es sinónimo de seguir patrones de diseño comprobados para la tecnología en cuestión. El patrón de diseño Model-View-ViewModel es el patrón ubicuo en todas las tecnologías basadas en XAML.

¿Qué es el patrón de diseño Model-View-ViewModel?

El patrón de diseño Model-View-ViewModel o simplemente MVVM es el patrón recomendado y natural para todas las plataformas que utilizan XAML como tecnología para definir sus interfaces de usuario ya que aprovecha al máximo la infraestructura de enlace de datos. Está basado en el conocido patrón Model-View-Controller o MVC, y también tiene tres piezas: un Modelo o Model una Vista o View y un Modelo para la Vista o ViewModel.

Nota: La traducción más fiel y correcta para el ViewModel es “Modelo para la Vista” o “Modelo de la Vista”, ya que justamente su papel es el de exponer las propiedades necesarias para la Vista en cuestión, es decir, “modela” los datos que requiere una Vista. La simple traducción “Vista-Modelo” es desatinada, errónea y no describe su intención verdadera. Pero, para evitar el uso de la frase completa en español “Modelo para la Vista”, en el resto del libro usaré el término en inglés “ViewModel”.

El siguiente diagrama describe los miembros del patrón de diseño MVVM y su relación que guardan entre sí:



La Vista o View

La Vista o View es la parte del patrón de diseño que define la Interfaz de Usuario de nuestra aplicación. En esta "capa" caen todos los elementos y conceptos relacionados exclusivamente con la vista como las Páginas, los Recursos, los Estilos, las Plantillas de Datos, etcétera. Una de las cosas que debemos aclarar y en contra de lo que mucha gente piensa, la Vista sí puede tener código en el archivo de code-behind. No obstante, la sugerencia es que lo limites únicamente a código que tenga que ver exclusivamente con la Vista y que estés dispuesto a no crearle una Prueba Unitaria a dicho código. Por ejemplo, iniciar o detener una animación, establecer el enfoque en un campo específico al iniciar una página, etcétera.

La Vista se actualiza a través de la infraestructura de enlace de datos de Xamarin.Forms. Nunca deberíamos estar modificando de manera directa las propiedades de los controles que incluye.

El Modelo para la Vista o ViewModel

El "Modelo para la Vista" o ViewModel es una abstracción de la Vista, exponiendo propiedades de datos y/o Comandos para que la Vista sea capaz de enlazarse a ellos. Por lo tanto, podemos decir que el ViewModel "adapta" el Modelo que usa la aplicación a una Vista.

Los ViewModel deben ser capaces de notificar los cambios en los valores de las propiedades públicas que expone hacia la Vista. Por lo tanto, los ViewModel deben implementar la interfaz INotifyPropertyChanged o heredar de alguna clase base abstracta que implemente dicha interfaz.

Adicionalmente, el ViewModel es el responsable de mantener y guardar el estado, además de implementar toda la lógica de presentación necesaria como cambio de estado, validaciones de datos en los controles enlazados, etcétera.

El Modelo

El Modelo o Model, es todas tus clases que representan tu dominio en la aplicación. Estas clases pueden ser de tipo POCO (Plain Old CLR Object) con comportamiento o de tipo DTO (Data Transfer Object), es decir, solo clases que implementen propiedades públicas que modelen la clase en cuestión.

Nota: Hay algunos autores que describen las clases que no tienen comportamiento como POCO, y viceversa.

En esta "capa" también entran los Servicios para la aplicación, capaces de incluir reglas de negocio relacionadas con la consulta y el manejo de los datos de la aplicación.

Ventajas

Usar este patrón nos ofrece diversas ventajas que describiremos a continuación.

SEPARACIÓN DE PREOCUPACIONES Y DESACOPLAMIENTO

La separación de preocupaciones y el desacoplamiento son un resultado inmediato al usar el patrón MVVM, ya que en la vista se define la interfaz de usuario y todos los elementos visuales mientras que la lógica de esa vista queda en una clase completamente separada.

PRUEBAS UNITARIAS

Ya que la lógica de la vista está encapsulada en una clase completamente separada, podemos escribir y ejecutar pruebas unitarias que comprueben la correcta funcionalidad de dicha lógica.

MANTENIMIENTO DE CÓDIGO

El mantenimiento de código se reduce, ya que al no estar mezclando las cosas, queda un código limpio, en donde es más fácil encontrar y resolver los bugs que tenga la lógica de nuestra aplicación.

CONSISTENCIA

Si sigues el patrón en una aplicación, muy probablemente en otra aplicación diferente lo vas a poder replicar. Asimismo, cuando estás trabajando en un equipo

con múltiples desarrolladores y todos se guían por el patrón, el código es consistente independientemente de quién lo haya escrito.

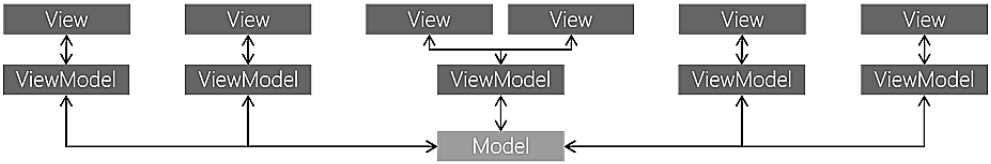
REUTILIZACIÓN DE CÓDIGO

El código queda altamente reutilizable en otros lugares dentro de la misma aplicación incluso para otras aplicaciones.

Cardinalidad entre los Model, Views y ViewModels

Generalmente, una Vista tiene solamente un ViewModel que ejerce como su contexto de enlace de datos. Por otro lado, un ViewModel puede ser usado en más de una Vista si así fuese necesario y tomando en cuenta que el ViewModel modela los datos para una Vista específicamente. Asimismo, un Modelo puede ser usado en uno o más ViewModels y frecuentemente es así en la mayoría de proyectos con Xamarin.Forms que siguen este patrón de diseño.

El siguiente diagrama muestra la cardinalidad entre los diferentes miembros del patrón de diseño MVVM:



Estrategias para relacionar una Vista con su ViewModel

En el desarrollo de aplicaciones con tecnologías XAML, hay dos principales técnicas para relacionar un ViewModel con una Vista:

PRIMERO LA VISTA O "VIEW-FIRST"

En esta técnica, es la Vista quien crea el objeto de tipo ViewModel y lo establece como contexto de enlace de datos a través de la propiedad BindingContext que tienen todos los objetos que heredan de manera directa o indirecta de la clase BindingObject. Esta técnica es la más común y es la que hemos estado usando –a propósito– durante los ejemplos de código y en la aplicación de encuestas.

La creación del objeto ViewModel puede ser a través de XAML, ya sea instanciándolo en un diccionario de recursos y referenciándolo con {StaticResource} o {DynamicResource}, o estableciéndolo directamente como valor de una propiedad por medio de la sintaxis de subelementos del lenguaje XAML. Otros autores incluso

establecen el objeto ViewModel como BindingContext por medio del archivo de code-behind, pero yo no soy partidario de esta técnica.

Otro tipo de arquitectura es inyectar el ViewModel a la Vista a través de su constructor. En el capítulo 9 usaremos esta arquitectura de inyección de ViewModels a través del uso de Unity como contenedor de Inyección de Dependencias.

PRIMERO EL VIEWMODEL O "VIEWMODEL-FIRST"

En esta otra técnica, es el ViewModel quien crea la Vista y se enlaza así mismo a ella. Es un poco más rara pero perfectamente posible en las tecnologías XAML.

Cómo pensar en MVVM

El patrón de diseño MVVM no es más que una guía que te ayuda a separar la Interfaz de Usuario de tu lógica de presentación y las reglas de negocio. Seguir e implementar el patrón de diseño se basa en que implementes el código correcto en el lugar adecuado, pero la piedra angular siempre será el enlace de datos.

Por ejemplo, ¿necesitas cambiar una Vista? Entonces, implementas una propiedad pública notificable en el ViewModel:

- Si deseas deshabilitar un botón, implementas en el ViewModel una propiedad de tipo bool y enlazas la propiedad IsEnabled
- Si deseas cambiar el estado de una Vista, implementas en el ViewModel una propiedad de tipo bool y enlazas la propiedad IsVisible de la Vista
- Si deseas mostrar errores de validación, implementas en el ViewModel una propiedad de tipo string o IEnumerable<string> (si son varios) y enlazas un Label o ListView a dichas propiedades

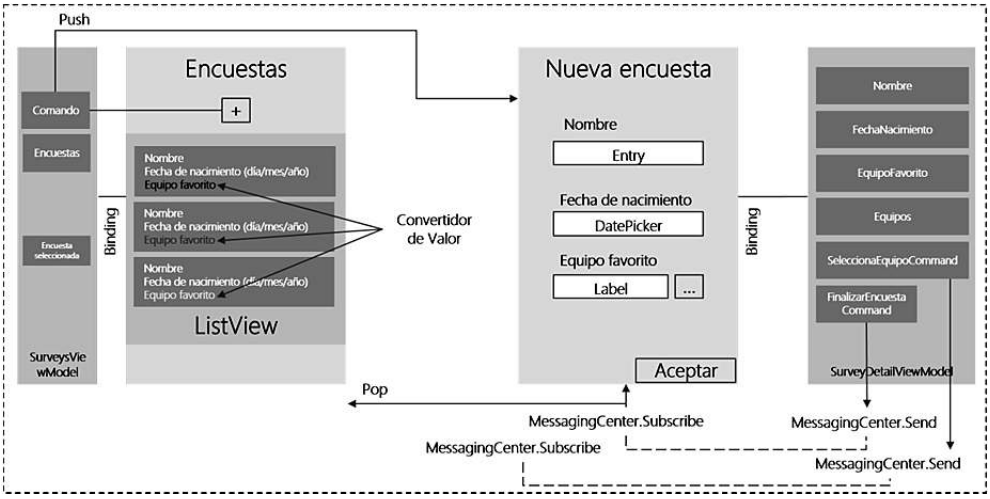
¿Deseas ejecutar código a través de la Vista? Entonces, el ViewModel debe exponer propiedades de tipo ICommand para que las puedas enlazar a las propiedades Command de los controles, por ejemplo del Button o del ToolbarItem.

Como seguramente ya lo dedujiste, la clave está siempre en enlazar la Vista a las propiedades que expone su ViewModel.

También es importante mencionar que es ideal que un ViewModel no dependa de una Vista específicamente. Por ejemplo, el referenciar en el ViewModel un objeto de tipo concreto de una Vista se considera mala práctica. De igual manera, es ideal que una Vista no dependa de un tipo concreto de ViewModel, siempre deberías de ser capaz de sustituir un ViewModel por otro (siempre y cuando expongan las mismas propiedades que espera la Vista).

MANOS A LA OBRA

Es hora de implementar adecuadamente el patrón de diseño Model-View-ViewModel en nuestra aplicación de encuestas. El siguiente diagrama muestra los cambios que haremos en el proyecto de la PCL durante este capítulo:



Renombrando y organizando las clases actuales

A lo largo de los capítulos anteriores, a propósito hemos implementado el proyecto de la PCL usando nociones del patrón de diseño MVVM. Por ejemplo, la clase Survey es el Modelo de la aplicación. Por su parte, la clase "Datos" es el ViewModel de la Vista SurveysView. ¡Ya es buen momento de cambiar los nombres por algo más adecuado!

En el patrón de diseño MVVM es común que los Views y ViewModels se llamen igual, diferenciándolos a través de un sufijo "View" o "ViewModel". Por esta razón, renombraremos Data a SurveysViewModel:

```
public class SurveysViewModel : NotificationObject
{
    …
}
```

Debido a este cambio de nombre, debemos hacer la siguiente modificación en el archivo de code-behind de la página SurveysView, específicamente en el método Subscribe():

```
using Xamarin.Forms;
```



```
namespace Surveys.Core
{
    public partial class SurveysView : ContentPage
    {
        public SurveysView()
        {
            InitializeComponent();


MessagingCenter.Subscribe<SurveysViewModel>(this,
Messages.NewSurvey, async (sender) =>
            {
                await Navigation.PushAsync(new
SurveyDetailsView());
            });
        }
    }
}
```

También, debemos modificar el código XAML de SurveysView, para corregir la instanciación y referenciación de SurveysViewModel:

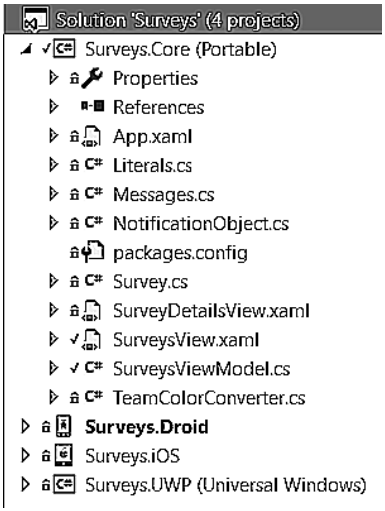
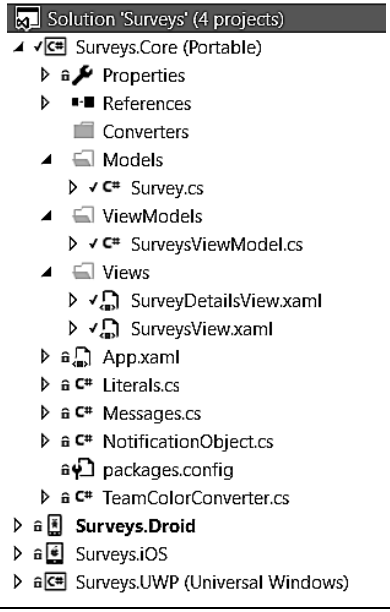
```
<ContentPage.Resources>
  <ResourceDictionary>
    <core:SurveysViewModel x:Key="SurveysViewModel" />
    …
  </ResourceDictionary>
</ContentPage.Resources>

<Grid Margin="10"
      BindingContext="{Binding Source={StaticResource
SurveysViewModel}}">
```

```
    …
</Grid>
```

Una vez efectuados estos cambios, la aplicación seguirá funcionando correctamente. Por otro lado, es buen momento de mejorar el proyecto de la PCL para agregar algunos folders que nos permitan ordenar de mejor manera todas las clases, no solamente las que hemos escrito a este momento, sino también las nuevas clases que agreguemos al proyecto.

Agregaremos los siguientes folders, y moveremos las clases existentes que correspondan a cada uno de esos folders:

- Converters
- Models
- ViewModels
- Views

Antes	Después
	

Implementando SurveyDetailsViewModel

Actualmente la página SurveyDetailsView no sigue el patrón de diseño MVVM. Para lograr esto, agregaremos al proyecto PCL una nueva clase llamada

SurveyDetailsViewModel en el folder "ViewModels". Esta nueva clase deberá heredar también de la clase base NotificationObject, y en ella implementaremos los siguientes miembros:

Name	Propiedad de tipo string para el nombre del encuestado
Birthdate	Propiedad de tipo DateTime para la fecha de nacimiento del encuestado
FavoriteTeam	Propiedad de tipo string para el equipo favorito que ha seleccionado el encuestado
Teams	Propiedad de tipo ObservableCollection<string> con la lista de los equipos
SelectTeamCommand	Propiedad de tipo ICommand para ejecutar la acción de selección de un equipo
EndSurveyCommand	Propiedad de tipo ICommand para ejecutar la acción de finalización de una encuesta

El siguiente fragmento de código muestra la implementación de la clase SurveyDetailsViewModel:

```
using System.Collections.ObjectModel;
using System.Windows.Input;

namespace Surveys.Core.ViewModels
{
    public class SurveyDetailsViewModel :
NotificationObject
    {
        private string name;

        public string Name
        {
            get
            {
                return name;
```

```
        }
        set
        {
            if (name == value)
            {
                return;
            }
            name = value;
            OnPropertyChanged();
        }
    }

    private DateTime birthdate;

    public DateTime Birthdate
    {
        get
        {
            return birthdate;
        }
        set
        {
            if (birthdate == value)
            {
                return;
            }
            birthdate = value;
            OnPropertyChanged();
        }
    }
```

```
private string favoriteTeam;

public string FavoriteTeam
{
    get
    {
        return favoriteTeam;
    }
    set
    {
        if (favoriteTeam == value)
        {
            return;
        }
        favoriteTeam = value;
        OnPropertyChanged();
    }
}
private ObservableCollection<string> teams;

public ObservableCollection<string> Teams
{
    get
    {
        return teams;
    }
    set
    {
        if (teams == value)
        {
```

```
                    return;
                }
                teams = value;
                OnPropertyChanged();
            }
        }

        public ICommand SelectTeamCommand { get; set; }

        public ICommand EndSurveyCommand { get; set; }
    }
}
```

Modificando la página SurveyDetailsView

La clase de code-behind de la página SurveyDetailsView tiene toda la implementación de funcionalidad requerida para llenar una nueva encuesta y regresar a la página principal.

El primer cambio que haremos será modificar el documento XAML de la página SurveyDetailsView para establecer como contexto de enlace de datos un objeto de tipo SurveyDetailsViewModel. Para lograr esto, crearemos una nueva instancia de SurveyDetailsViewModel en el diccionario de recursos de la página y posteriormente enlazaremos la propiedad BindingContext del Grid raíz con este objeto:

```
…
<ContentPage.Resources>
  <ResourceDictionary>
    <viewModels:SurveyDetailsViewModel
x:Key="SurveyDetailsViewModel" />
  </ResourceDictionary>
</ContentPage.Resources>

<Grid Margin="10"
      BindingContext="{Binding Source={StaticResource
SurveyDetailsViewModel}}">
  …
</Grid>
```

Una vez hecho esto, enlazaremos todos los controles con las propiedades expuestas en el objeto SurveyDetailsViewModel. También es buen momento de quitar los atributos x:Name de los controles ya que no los vamos a referenciar más en el archivo de code-behind, así como también removeremos los manejadores del evento Clicked de ambos botones.

El siguiente fragmento de código muestra el código XAML de SurveyDetailsView una vez finalizados los cambios anteriormente descritos:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage
xmlns="http://xamarin.com/schemas/2014/forms"

xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:viewModels="clr-
namespace:Surveys.Core.ViewModels;assembly=Surveys.Core"
             x:Class="Surveys.Core.SurveyDetailsView"
             Title="Nueva encuesta">

  <ContentPage.Resources>
    <ResourceDictionary>
      <viewModels:SurveyDetailsViewModel
x:Key="SurveyDetailsViewModel" />
    </ResourceDictionary>
  </ContentPage.Resources>

  <Grid Margin="10"
        BindingContext="{Binding Source={StaticResource
SurveyDetailsViewModel}}">
      <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition Height="Auto" />
      </Grid.RowDefinitions>

      <StackLayout>
        <Label Text="Nombre" />
```

```
        <Entry Text="{Binding Name, Mode=TwoWay}" />

        <Label Text="Fecha de nacimiento" />
        <DatePicker Date="{Binding Birthdate,
Mode=TwoWay}" />

        <Label Text="Equipo favorito" />
        <Grid>
          <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition Width="Auto" />
          </Grid.ColumnDefinitions>
          <Label Text="{Binding FavoriteTeam}" />
          <Button Grid.Column="1"
                  Text="..."
                  Command="{Binding SelectTeamCommand}"
/>
        </Grid>
      </StackLayout>

      <Button Text="Aceptar"
              Grid.Row="1"
              Command="{Binding EndSurveyCommand}" />
    </Grid>
  </ContentPage>
```

Implementando la colección de equipos

Debemos llenar la colección de equipos expuesta a través de la propiedad Teams. La lista de equipos actualmente está implementada en el code-behind de la página SurveyDetailsView como un campo privado de tipo string[]. Hay que mover dicho código para que en el constructor de la clase SurveyDetailsViewModel quede implementado de la siguiente manera:

```
public SurveyDetailsViewModel()
{
    Teams =
        new ObservableCollection<string>(new[]
        {
            "Alianza Lima", "América", "Boca Juniors",
"Caracas FC", "Colo-Colo", "Peñarol", "Real Madrid",
            "Saprissa"
        });
}
```

Implementando la selección de equipos

El comando SelectTeamCommand es el responsable de ejecutar la acción que permite a un usuario seleccionar un equipo de la lista expuesta en la propiedad Teams. No obstante, nos hemos topado con una pared nuevamente ya que el método DisplayActionSheet() está expuesto en la clase Page y no está disponible en el ViewModel. Tal y como lo resolvimos en el capítulo pasado, utilizaremos el MessagingCenter para desacoplar al máximo la invocación de DisplayActionSheet(), ya que la única manera que tenemos en este momento de usarlo es desde la instancia de la página.

Por lo tanto, implementaremos un nuevo mensaje llamado SelectTeam en la clase Messages el cual publicaremos a través del MessagingCenter. La página SurveyDetailsView será la responsable de suscribirse a este método y en el callback invocará el método DisplayActionSheet(). Sin embargo, ¡nos volvemos a topar con la pared! ya que necesitamos un mecanismo para enviar de regreso al objeto SurveyDetailsViewModel el equipo seleccionado. Esto también lo podemos lograr nuevamente con el MessagingCenter, así que además de SelectTeam, crearemos otro mensaje llamado TeamSelected en la clase Messages. El siguiente fragmento de código muestra la implementación completa de la clase Messages:

```
namespace Surveys.Core
{
    public class Messages
    {
```

```
        public const string NewSurveyComplete =
"NewSurveyComplete";
        public const string NewSurvey = "NewSurvey";
        public const string SelectTeam = "SelectTeam";
        public const string TeamSelected =
"TeamSelected";
    }
}
```

Posteriormente, implementaremos en la clase SurveyDetailsViewModel un nuevo método llamado SelectTeamCommandExecute() que envíe el mensaje SelectTeam, pasando como parámetro el arreglo de strings de la colección Teams. El método lo asignaremos como acción en el constructor de la clase Command. El siguiente fragmento de código muestra la implementación de estos cambios:

```
public SurveyDetailsViewModel()
{
    Teams =
        new ObservableCollection<string>(new[]
        {
            "Alianza Lima", "América", "Boca Juniors",
"Caracas FC", "Colo-Colo", "Peñarol", "Real Madrid",
            "Saprissa"
        });

    SelectTeamCommand = new
Command(SelectTeamCommandExecute);
}

private void SelectTeamCommandExecute()
{
    MessagingCenter.Send(this, Messages.SelectTeam,
Teams.ToArray());
}
```


Ahora, de regreso al code-behind de la página SurveyDetailsView, en el constructor nos suscribiremos al mensaje SelectTeam, obteniendo en el callback el arreglo de strings con los equipos para usarlos en el método DisplayActionSheet(). Una vez obtenido un equipo favor a través del método DisplayActionSheet(), lo enviaremos de regreso a SurveyDetailsViewModel por medio del mensaje TeamSelected, pasando como parámetro el equipo favorito seleccionado.

El siguiente fragmento de código muestra la implementación del constructor de la página SurveyDetailsView, en donde usamos el MessagingCenter tanto para la suscripción como para la publicación de los mensajes:

```
public SurveyDetailsView()
{
    InitializeComponent();


MessagingCenter.Subscribe<SurveyDetailsViewModel,string[]
>(this, Messages.SelectTeam, async (sender, args) =>
    {
        var favoriteTeam = await
DisplayActionSheet(Literals.FavoriteTeamTitle, null,
null, args);


        if (!string.IsNullOrWhiteSpace(favoriteTeam))
        {
            MessagingCenter.Send((ContentPage)this,
Messages.TeamSelected, favoriteTeam);
        }
    });
}
```

El siguiente fragmento de código muestra la suscripción al mensaje TeamSelected en el constructor de la clase SurveyDetailsViewModel:

```
MessagingCenter.Subscribe<ContentPage, string>(this,
Messages.TeamSelected, (sender, args) => { FavoriteTeam =
args; });
```

Implementando el comando para finalizar una encuesta

En el constructor de la clase SurveyDetailsViewModel, asignaremos un nuevo objeto de tipo Command a la propiedad EndSurveyCommand, pasando como parámetros un método llamado EndSurveyCommandExecute() el cual tendrá la acción a ejecutar, y otro llamado EndSurveyCommandCanExecute() para evaluar si el comando se puede ejecutar o no.

El resultado de EndSurveyCommandCanExecute() estará en función de si el usuario ha capturado un nombre y si ha seleccionado un equipo favorito. El siguiente fragmento de código muestra la implementación de este método:

```
  private bool EndSurveyCommandCanExecute()
  {
      return !string.IsNullOrWhiteSpace(Name) &&
!string.IsNullOrWhiteSpace(FavoriteTeam);
  }
```

Ya que estamos interesados en conocer cuándo las propiedades Name y FavoriteTeam cambien para poder reevaluar el comando EndSurveyCommand, debemos solicitar esa reevaluación justamente cuando son modificadas. Una manera rápida sería implementar dicho código en los accesores set{} de ambas propiedades, pero contaminaríamos la propiedad y le estaríamos dando una responsabilidad adicional. Una manera más limpia para resolver esto es manejar el evento PropertyChanged de la clase SurveyDetailsViewModel y evaluar si alguna de esas propiedades son las que han sido modificadas. El siguiente fragmento de código muestra el manejo del evento PropertyChanged en el constructor de la clase SurveyDetailsViewModel:

```
  public SurveyDetailsViewModel()
  {
      …
      PropertyChanged +=
SurveyDetailsViewModel_PropertyChanged;
  }


  private void
SurveyDetailsViewModel_PropertyChanged(object sender,
      System.ComponentModel.PropertyChangedEventArgs e)
```



```
    {
        if (e.PropertyName == nameof(Name) ||
e.PropertyName == nameof(FavoriteTeam))
        {
            (EndSurveyCommand as
Command)?.ChangeCanExecute();
        }
    }
```

Por su parte, el método EndSurveyCommandExecute() creará el nuevo objeto de tipo Survey con los datos disponibles en el ViewModel. Además, mandará el mensaje NewSurveyComplete usando como parámetro la nueva encuesta capturada, tal y como lo muestra el siguiente fragmento de código:

```
    private void EndSurveyCommandExecute()
    {
        var newSurvey = new Survey() { Name = Name,
Birthdate = Birthdate, FavoriteTeam = FavoriteTeam };


        MessagingCenter.Send(this,
Messages.NewSurveyComplete, newSurvey);
    }
```

Cuando terminamos de capturar una encuesta, debemos navegar de regreso a la página SurveysView, por lo que tenemos que ejecutar el método PopAsync() de la propiedad Navigation de la página. Por este motivo, nuevamente, nos basaremos en la funcionalidad del MessagingCenter para publicar un mensaje desde SurveysViewModel y nos suscribiremos a él en el code-behind de la página SurveyDetailsView. Este mensaje será NewSurveyComplete, el cual ya teníamos anteriormente en la clase Messages. El siguiente código muestra la implementación de la suscripción y publicación de mensajes en el archivo code-behind de la página SurveyDetailsView. Es importante destacar también que hemos borrado el código que teníamos anteriormente:

```
    using Surveys.Core.ViewModels;
    using Xamarin.Forms;


    namespace Surveys.Core
```

```
  {
      public partial class SurveyDetailsView :
ContentPage
      {
          public SurveyDetailsView()
          {
              InitializeComponent();


MessagingCenter.Subscribe<SurveyDetailsViewModel,
string[]>(this, Messages.SelectTeam, async (sender, args)
=>
              {
                  var favoriteTeam = await
DisplayActionSheet(Literals.FavoriteTeamTitle, null,
null, args);


                  if
(!string.IsNullOrWhiteSpace(favoriteTeam))
                  {

MessagingCenter.Send((ContentPage)this,
Messages.TeamSelected, favoriteTeam);
                  }
              });


MessagingCenter.Subscribe<SurveyDetailsViewModel,
Survey>(this, Messages.NewSurveyComplete, async (sender,
args) =>
              {
                  await Navigation.PopAsync();
              });
          }
      }
  }
```

Recuerda que en el constructor de la clase SurveysViewModel ya hay una suscripción existente al mensaje NewSurveyComplete. Ya que el patrón de diseño Pub/Sub que ofrece el MessagingCenter permite tener más de un suscriptor por mensaje, seguiremos usando esa suscripción para agregar a la colección Surveys la nueva encuesta capturada, solo necesitamos cambiar el tipo del publicador para usar SurveyDetailsViewModel en vez de ContentPage. El siguiente fragmento de código muestra el constructor de la clase SurveysViewModel después de hacer este cambio:

```
public SurveysViewModel()
{
    Surveys = new ObservableCollection<Survey>();

    NewSurveyCommand = new
Command(NewSurveyCommandExecute);
    MessagingCenter.Subscribe<SurveyDetailsViewModel,
Survey>(this, Messages.NewSurveyComplete,
        (sender, args) => { Surveys.Add(args); });
}
```

Desinscribiendo los mensajes

Finalmente, para evitar problemas en el momento de crear múltiples encuestas, debemos desinscribirnos de ambos mensajes SelectTeam y NewSurveyComplete cuando salimos de la página SurveyDetailsView, ya que de lo contrario estaríamos tratando de usar objetos inexistentes, debido a que cada vez que navegamos de regreso, la página en la que estábamos es destruida. Por este simple motivo, sobrescribiremos el método OnDisappearing() de la página y ahí ejecutaremos el método Unsubscribe() de la clase MessagingCenter, indicando el tipo del publicado y del argumento, además de quién es el suscriptor y el nombre del mensaje.

El siguiente código muestra la implementación final completa del archivo de code-behind de la página SurveyDetailsView:

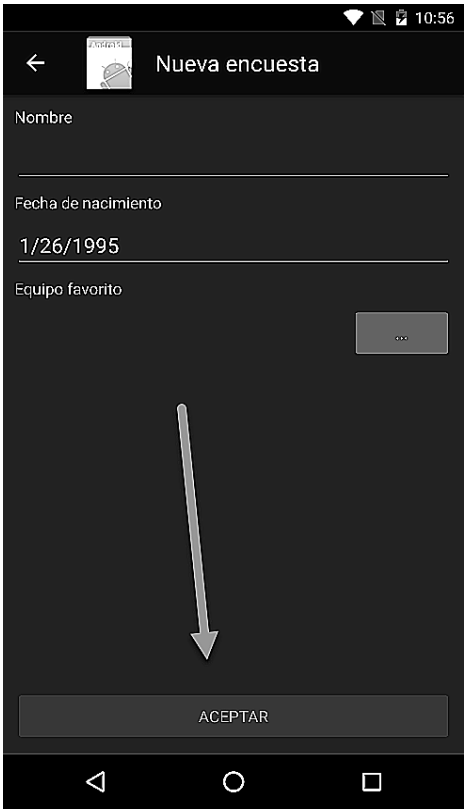
```
using Surveys.Core.ViewModels;
using Xamarin.Forms;

namespace Surveys.Core
{
    public partial class SurveyDetailsView :
ContentPage
```

```
      {
          public SurveyDetailsView()
          {
              InitializeComponent();

MessagingCenter.Subscribe<SurveyDetailsViewModel,
string[]>(this, Messages.SelectTeam,
                  async (sender, args) =>
                  {
                      var favoriteTeam = await
DisplayActionSheet(Literals.FavoriteTeamTitle, null,
null, args);


                      if
(!string.IsNullOrWhiteSpace(favoriteTeam))
                      {

MessagingCenter.Send((ContentPage)this,
Messages.TeamSelected, favoriteTeam);
                      }
                  });


MessagingCenter.Subscribe<SurveyDetailsViewModel,
Survey>(this, Messages.NewSurveyComplete,
                  async (sender, args) => { await
Navigation.PopAsync(); });
          }


          protected override void OnDisappearing()
          {
              base.OnDisappearing();

MessagingCenter.Unsubscribe<SurveyDetailsViewModel,
string[]>(this, Messages.SelectTeam);
```

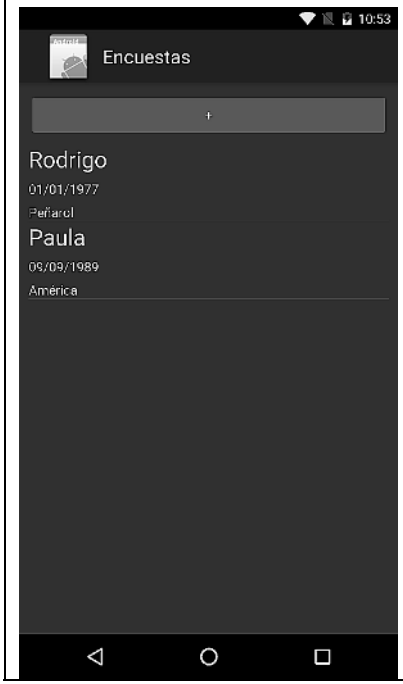
```
MessagingCenter.Unsubscribe<SurveyDetailsViewModel,
Survey>(this, Messages.NewSurveyComplete);
            }
        }
    }
```

Ejecutando la aplicación

Hemos finalizado la implementación de la aplicación usando el patrón de diseño MVVM, usando solamente el API de Xamarin.Forms. Si después de todas estas modificaciones ejecutamos la aplicación, podrás apreciar que efectivamente podemos capturar una o más encuestas. Además, te podrás dar cuenta de que una nueva encuesta solo la podrás agregar a la lista si su nombre y su equipo favorito se han capturado, tal y como lo muestra la siguiente figura:



Una vez capturados el nombre y el equipo favorito, la encuesta es enviada a la colección de encuestas de la página inicial (SurveysView):

Android	UWP
	

¡Uff! Parece ser que es un sinfín de idas y vueltas de mensajes por medio del MessagingCenter. Déjame confesarte una cosa: lo es. La infraestructura de navegación de Xamarin.Forms está ciertamente limitada, por lo que hay mejores opciones que incluso el mismo Xamarin está pensando adoptar y que en el momento de estar escribiendo estas líneas aún no lo ha hecho. Nos referimos a Prism y su sofisticada infraestructura de navegación. Pero antes de tocar el tema de Prism, debemos ver antes un tema muy importante y vital para todas las aplicaciones de negocio serias construidas con Xamarin.Forms: la invocación de funcionalidad específica de cada una de las plataformas concretas.

8 FUNCIONALIDAD NATIVA DE LAS PLATAFORMAS

INTRODUCCIÓN

Generalmente, una aplicación de negocios construida con Xamarin.Forms va a requerir invocar funcionalidades nativas del sistema operativo donde está ejecutando. Esto debido a que Xamarin.Forms no es más que una abstracción de diversos elementos visuales, y fuera de ello, el resto de piezas en nuestra aplicación serán Xamarin "clásico".

Hay diferentes maneras que podemos utilizar en nuestras aplicaciones para determinar en qué plataforma está ejecutándose la aplicación, y muy importante: ejecutar lógica o características que son únicas del sistema operativo.

CLASE DEVICE

La clase Xamarin.Forms.Device incluye diversos miembros que nos permiten ejecutar código específico en una plataforma, así como también determinar algunas características del dispositivo donde está ejecutándose la aplicación.

Idiom

La propiedad Idiom nos devuelve el tipo de dispositivo en donde está ejecutándose la aplicación. El tipo de propiedad es TargetIdiom con alguno de los siguientes valores:

Unsupported	La aplicación está ejecutándose en un dispositivo no soportado
Phone	La aplicación está ejecutándose en un dispositivo tipo teléfono
Tablet	La aplicación está ejecutándose en un dispositivo tipo Tablet
Desktop	La aplicación está ejecutándose en el escritorio (UWP)

OS

La propiedad OS nos devuelve el tipo de sistema operativo en donde está ejecutándose la aplicación. El tipo de propiedad es TargetPlatform con alguno de los siguientes valores:

Other	La aplicación está ejecutándose en una plataforma no conocida
iOS	La aplicación está ejecutándose en iOS
Android	La aplicación está ejecutándose en Android
WinPhone	La aplicación está ejecutándose en Windows Phone
Windows	La aplicación está ejecutándose en la plataforma Windows (UWP)

BeginInvokeOnMainThread()

Hay escenarios en los que necesitamos modificar algún elemento de la Interfaz de Usuario, pero el código en donde lo queremos implementar está en un hilo secundario. El método BeginInvokeOnMainThread() nos permite ejecutar una acción en el hilo de la Interfaz de Usuario. En el siguiente fragmento de código, después de esperar el Task, la ejecución estaría en un hilo secundario por lo que cambiar el texto de un Label mandaría un error, pero el uso de BeginInvokeOnMainThread() lo permitiría:

```
  private async void Button_OnClicked(object sender,
EventArgs e)
  {
      await Task.Delay(1000).ConfigureAwait(false);


      Device.BeginInvokeOnMainThread(() => myLabel.Text =
DateTime.Now.ToString());
  }
```


StartTimer()

Inicia un timer basado en el reloj interno del dispositivo. El siguiente fragmento de código mostraría, cada segundo, la fecha y hora actual del dispositivo en una etiqueta:

```
  private async void Button_OnClicked(object sender,
EventArgs e)
  {
      await Task.Delay(1000).ConfigureAwait(false);


      Device.StartTimer(TimeSpan.FromSeconds(1), () =>
      {
          Device.BeginInvokeOnMainThread(() =>
myLabel.Text = DateTime.Now.ToString());
          return true;
      });
  }
```

OnPlatform() y OnPlatform<T>()

Los métodos OnPlatform() (no genérico y genérico) nos permiten ejecutar una acción u obtener un valor dependiendo de la plataforma donde esté ejecutando la aplicación. En sus argumentos especificamos el valor deseado para el caso que sea iOS, Android o Windows. Por ejemplo, el siguiente fragmento de código establece un texto de un Label en función del sistema operativo donde esté ejecutándose la aplicación:

```
  mietiqueta.Text = Device.OnPlatform("Esto es iOS",
"Esto es Android", "Esto es Windows");
```

Este otro ejemplo mostraría la fecha y hora cada segundo en un Label pero únicamente en el sistema operativo Android:

```
  Device.OnPlatform(Android: () =>
  {
      Device.StartTimer(TimeSpan.FromSeconds(1), () =>
      {
```

```
            Device.BeginInvokeOnMainThread(() =>
myLabel.Text = DateTime.Now.ToString());
            return true;
        });
    });
```

CLASE ONPLATFORM<T>

Esta clase tiene el mismo objetivo que el método OnPlatform<T>() de la clase Device: obtener un valor según el sistema operativo del dispositivo donde está ejecutándose la aplicación. Si bien lo podemos utilizar en código C#, su uso realmente está pensado para XAML.

OnPlatform<T> hace uso del atributo x:TypeArguments para especificar su tipo (indicado en T). Por ejemplo, el siguiente fragmento de código cambia el tamaño de un botón, según el sistema operativo:

```
<Button>
  <Button.FontSize>
    <OnPlatform x:TypeArguments="x:Double"
            iOS="24"
            Android="28"
                        WinPhone="30" />
  </Button.FontSize>
</Button>
```

IMÁGENES

El uso de imágenes en las aplicaciones construidas con Xamarin.Forms debe respetar las ubicaciones, tipo de compilación y nomenclatura de los archivos que cada plataforma exige:

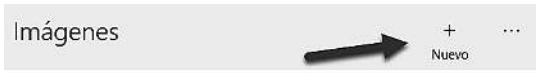
iOS	/Resources Build action: **BundleResource** Adicionalmente, versiones retina con el sufijo @2x y @3x
Android	/Resources/drawable Build action: **AndroidResource**

	Adicionalmente, versiones High y Low DPI en folders en /Resources: • drawable-ldpi • drawable-hdpi • drawable-xhdpi
UWP	/ o cualquier otro folder (puedes usar OnPlatform si está en otra ubicación diferente a /) Build action: **Content**

Por ejemplo, el siguiente fragmento de código establece el icono de un ToolbarItem para que en Android utilice el archivo "newicon.png" de su folder "drawable" mientras que en Windows use el archivo "new.png" localizado en el folder "assets":

```
<ContentPage.ToolbarItems>
  <ToolbarItem Text="Nuevo">
    <ToolbarItem.Icon>
      <OnPlatform x:TypeArguments="FileImageSource"
                  Android="newicon.png"
                  WinPhone="assets/new.png" />
    </ToolbarItem.Icon>
  </ToolbarItem>
</ContentPage.ToolbarItems>
```

Las siguientes figuras muestran una aplicación que incluye el anterior ToolbarItem, tanto en Android como en UWP:

Android	UWP
	

SERVICIO DE DEPENDENCIAS

Si analizamos la arquitectura de un aplicación con Xamarin.Forms, podemos apreciar que prácticamente todo el código lo implementamos en la Biblioteca de Clases Base (PCL), el cual tiene todo el código compartido. Sin embargo, en una

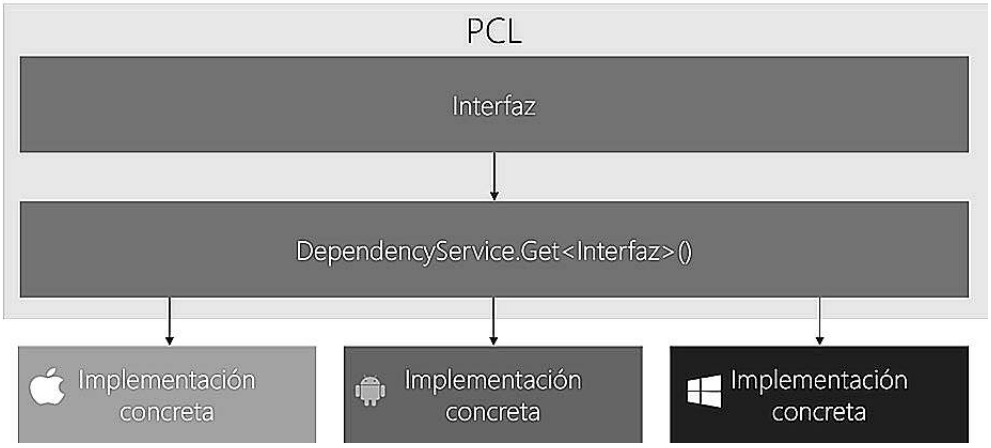
aplicación real es muy difícil que el 100% de los requerimientos sean implementados únicamente en el código compartido, ya que comúnmente vamos a necesitar funcionalidades nativas que tiene el sistema operativo del dispositivo en donde está ejecutándose la aplicación. Es por ello que Xamarin.Forms incluye un servicio de resolución de dependencias que a continuación describiremos.

DependencyService

Esta clase nos permite resolver un objeto que tenga la implementación concreta de alguna lógica de la plataforma donde está ejecutándose la aplicación.

Por su naturaleza, en un proyecto de tipo PCL sería imposible referenciar cualquier API de iOS / Android / Windows. De ahí que la clase DependencyService se base en la definición de interfaces en la PCL que incluyan todos los miembros mínimos esperados para obtener la funcionalidad buscada.

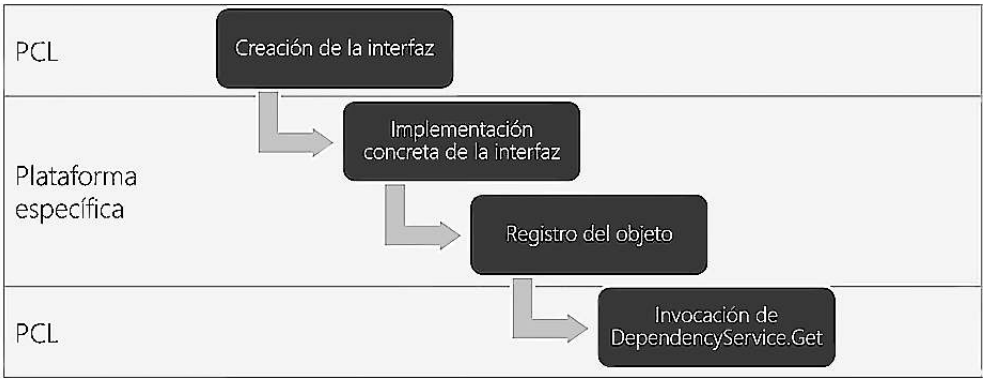
El siguiente diagrama muestra el diseño conceptual de una PCL que define una interfaz y que hace uso de la clase DependencyService para obtener el objeto concreto que implementa dicha interfaz en cada plataforma concreta:



Con lo anterior, podemos deducir que el primer paso para invocar alguna funcionalidad específica del sistema operativo donde está ejecutándose tu aplicación de Xamarin.Forms es definir una interfaz en la PCL. De hecho, antes de eso, es importante identificar la funcionalidad que no tienes a tu alcance en el código compartido de la PCL y que requieres de la plataforma concreta. Ejemplos de esto hay muchos: acceso al sistema de archivos, acceso a los sensores del dispositivo como GPS, acelerómetro, aproximación, etc., acceso a la cámara, acceso al teléfono,

lista de contactos, información del hardware, información del software, y un largo etcétera.

Una vez identificada la funcionalidad que requiere tu aplicación, debemos ahora sí comenzar con la definición de la interfaz. Los pasos completos para invocar una funcionalidad específica de la plataforma concreta son los que describe el siguiente diagrama y que posteriormente describimos uno por uno:



CREACIÓN DE LA INTERFAZ

En el proyecto de la PCL, debemos definir una o varias interfaces cuyo objetivo sea tener el mínimo de miembros necesarios que requiere tu aplicación para obtener la funcionalidad deseada, tomando en cuenta que, ya que estamos en un proyecto de tipo PCL, estamos confinados a la intersección de APIs que conforman las plataformas a las que apunta la PCL. Dicho de otra manera, solamente puedes usar tipos en tu interfaz a los que tenga acceso la PCL. Por otro lado, también es posible que la interfaz la implementes en algún proyecto externo también de tipo PCL si la arquitectura de tu aplicación así lo requiere.

El siguiente fragmento de código define la interfaz llamada IFileSystemService, la cual incluye un método llamado WriteFileAsync() al que le pasamos como parámetro un nombre de archivo y un contenido en cadena que deseamos guardar en el sistema de archivos:

```
public interface IFileSystemService
{
    Task WriteFileAsync(string fileName, string
content);
}
```

IMPLEMENTACIÓN CONCRETA Y REGISTRO

Una vez definida la interfaz, hacemos la implementación concreta en cada uno de los proyectos de las plataformas en donde queremos que ejecute nuestra aplicación. Durante la implementación, ya tendrás total libertad de utilizar las clases nativas de la plataforma.

Con la implementación lista, tenemos que "exponer" la clase en donde has implementado la interfaz para que el servicio de dependencias lo encuentre. Esto lo hacemos por medio del atributo Dependency del espacio de nombres Xamarin.Forms, indicando el tipo de la clase.

Implementación en UWP

El siguiente fragmento de código muestra la clase FileSystemService que hemos agregado al proyecto UWP y que implementa la interfaz IFileSystemService. Por medio del atributo Dependency a nivel de ensamblado estamos exponiendo el tipo de FileSystemService para que el servicio de dependencias lo encuentre. Cabe destacar el uso del método ApplicationData.Current.LocalFolder.CreateFileAsync() ya que es código específico de UWP:

```
[assembly:Dependency(typeof(FileSystemService))]


namespace Capitulo8.UWP
{
    public class FileSystemService : IFileSystemService
    {
        public async Task WriteFileAsync(string
fileName, string content)
        {
            var result = await
ApplicationData.Current.LocalFolder.CreateFileAsync(fileN
ame,

CreationCollisionOption.ReplaceExisting);


            using (var stream = await
result.OpenStreamForWriteAsync())
            {
```

```
                    using (var streamWriter = new
StreamWriter(stream))
                    {
                        await
streamWriter.WriteAsync(content);
                    }
                }
            }
        }
```

Implementación en Android

Este otro fragmento de código muestra la implementación de la clase FileSystemService en Android. Observa que a diferencia de UWP, aquí estamos usando File.Create() para crear el archivo en el sistema de archivos del dispositivo Android:

```
[assembly:Dependency(typeof(FileSystemService))]


namespace Capitulo8.Droid
{
    public class FileSystemService : IFileSystemService
    {
        public async Task WriteFileAsync(string
fileName, string content)
        {
            var path =
Path.Combine(Environment.GetFolderPath(Environment.Specia
lFolder.MyDocuments), fileName);


            if (File.Exists(path))
            {
                File.Delete(path);
            }
```

```
                using (var stream = File.Create(path))
                {
                    using (var streamWriter = new
StreamWriter(stream))
                    {
                        await
streamWriter.WriteLineAsync(content);
                    }
                }
            }
        }
    }
```

RESOLUCIÓN DE LA DEPENDENCIA

Una vez implementadas las clases concretas en cada plataforma, podemos utilizar el método Get<T> de la clase DependencyService para obtener el objeto que implementa la interfaz, en donde T es el tipo de interfaz. El siguiente fragmento de código obtiene el objeto IFileSystemService e invoca el método WriteFileAsync, pasando como parámetros el nombre del archivo y el contenido en cadena que finalmente se escribirá en el archivo:

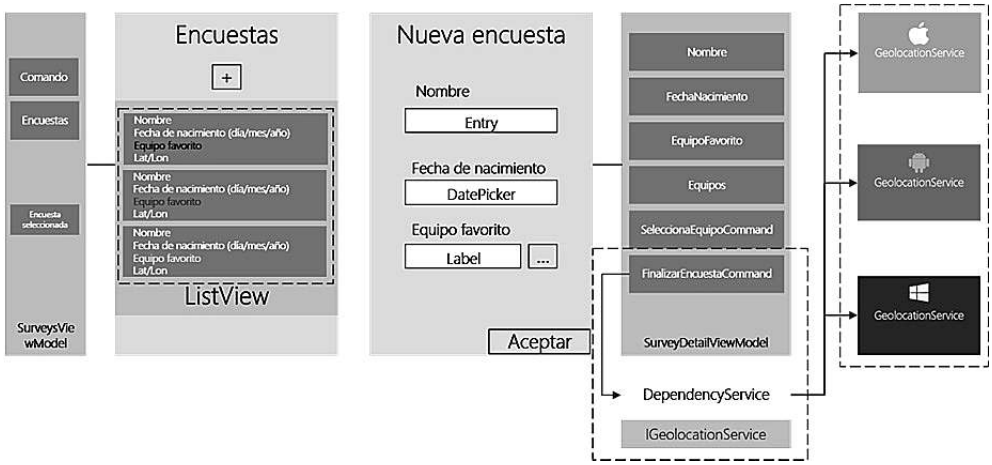
```
  var fileSystemService =
DependencyService.Get<IFileSystemService>();


  await fileSystemService.WriteFileAsync("prueba.txt",
"Hola desde la PCL");
```

PLUGINS DE XAMARIN

Xamarin y la comunidad de desarrolladores han puesto al alcance de cualquier persona una serie de plugins como paquetes de NuGet, los cuales agregan diversas funcionalidades específicas para cada plataforma. Por esta razón, probablemente querrás echar un vistazo a estos componentes para analizar si cumplen con la funcionalidad que tu proyecto requiere. Todos esos componentes de una u otra manera usan el Servicio de Dependencias que explicamos en este capítulo. Para mayor información acerca de estos plugins consulta la página de su repositorio en GitHub: https://github.com/xamarin/XamarinComponents.

MANOS A LA OBRA

Continuando con la implementación de nuestra aplicación de encuestas, en este capítulo implementaremos la funcionalidad para georreferenciar una encuesta cuando es capturada. Para cumplir con este requerimiento, necesitamos modificar la clase Survey para implementar las propiedades de latitud y longitud, además definiremos una interfaz llamada IGeolocationService en la PCL y la implementaremos en cada proyecto de plataforma concreta. El siguiente diagrama muestra los cambios que realizaremos:



Modificando el modelo

El primer cambio que vamos a realizar es modificar la clase Survey, para agregar un par de propiedades de tipo double que almacenen la latitud y la longitud de la encuesta una vez que haya sido capturada. Además, modificaremos el método ToString() para incluir ambas propiedades para que en la depuración podamos corroborar que la coordenada haya sido obtenida. No obstante, recuerda que estamos usando una Plantilla de Datos para presentar la encuesta en el control ListView, así que más adelante también la tendremos que modificar para incluir los valores de estas nuevas propiedades.

El siguiente fragmento muestra el código de la clase Survey una vez efectuados los cambios descritos anteriormente:

```
using System;
namespace Surveys.Core
{
```

```
        public class Survey
        {
            public string Name { get; set; }

            public DateTime Birthdate { get; set; }

            public string FavoriteTeam { get; set; }

            public double Lat { get; set; }

            public double Lon { get; set; }

            public override string ToString()
            {
                return $"{Name} |  {Birthdate} |
{FavoriteTeam} | {Lat} | {Lon}";
            }
        }
    }
```

Creando la interfaz IGeolocationService

En el proyecto PCL, agregaremos un nuevo folder llamado “ServiceInterfaces”, el cual tendrá como objetivo organizar las interfaces de la aplicación. En ese folder agregaremos un nuevo archivo llamado IGeolocationService.cs con la siguiente definición de interfaz:

```
using System;
using System.Threading.Tasks;
namespace Surveys.Core.ServiceInterfaces
{
    public interface IGeolocationService
    {
```



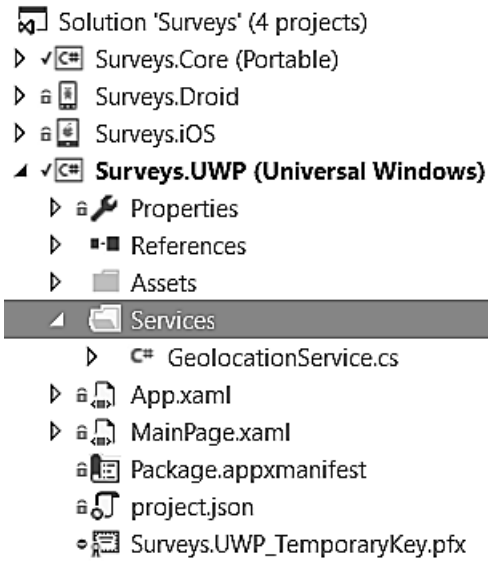
```
        Task<Tuple<double, double>>
GetCurrentLocationAsync();
    }
}
```

Como podrás observar en el anterior código, la interfaz tiene únicamente un miembro llamado GetCurrentLocationAsync(), el cual nos devolverá un objeto de tipo Tuple<double, double> en donde el primer elemento del Tuple será la latitud, y el segundo elemento será la longitud.

Nota: Como probablemente estás imaginando en este momento, otras implementaciones para lograr esta funcionalidad también son posibles. Por ejemplo, podríamos agregar una clase llamada Location con un par de propiedades de tipo double y que el método GetCurrentLocationAsync() devuelva un Task<Location>.

Implementación de IGeolocationService en UWP

En el proyecto UWP, en pro de tener también aquí una organización lógica más adecuada, agregaremos un folder llamado “Services”, el cual tendrá como objetivo organizar las implementaciones concretas de las interfaces de la aplicación. En este folder, agregaremos una nueva clase llamada GeolocationService tal y como lo muestra la siguiente figura:



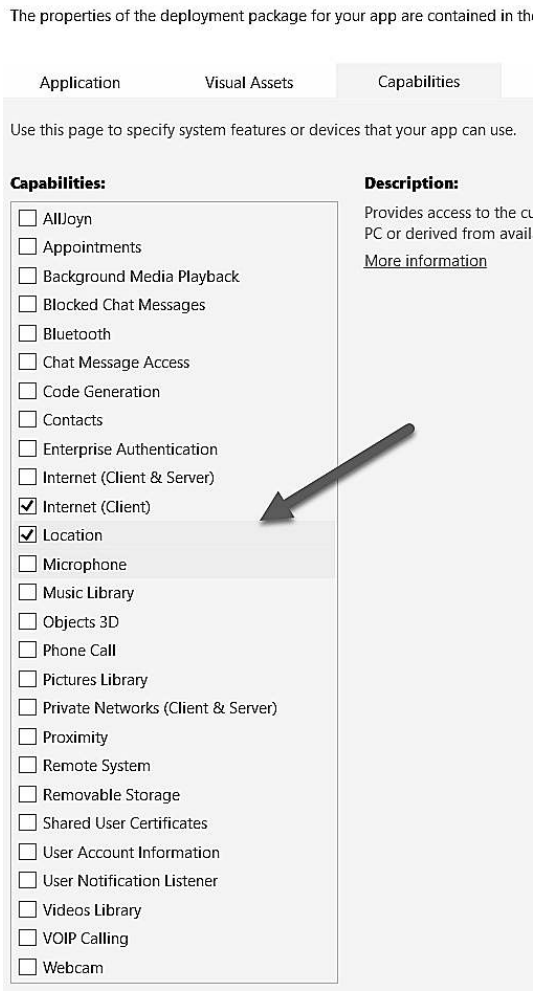
La clase GeolocationService debe implementar la interfaz IGeolocationService y exponer la clase concreta para que el servicio de dependencias lo pueda resolver. En

el caso de esta plataforma, usaremos la clase Windows.Devices.Geolocation.Geolocator para obtener la ubicación geográfica actual que reporta el dispositivo.

El siguiente fragmento de código muestra la implementación de la clase GeolocationService en UWP:

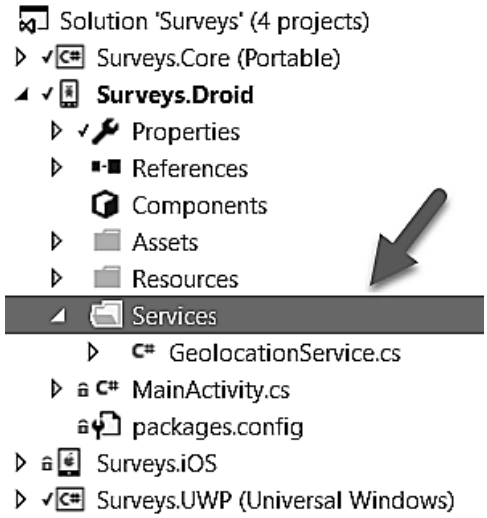
```
using System;
using System.Threading.Tasks;
using Windows.Devices.Geolocation;
using Surveys.Core.ServiceInterfaces;
using Surveys.UWP.Services;
using Xamarin.Forms;

[assembly:Dependency(typeof(GeolocationService))]

namespace Surveys.UWP.Services
{
    public class GeolocationService :
IGeolocationService
    {
        public async Task<Tuple<double, double>>
GetCurrentLocationAsync()
        {
            var geolocator = new Geolocator();
            var position = await
geolocator.GetGeopositionAsync();
            var result = new Tuple<double,
double>(position.Coordinate.Point.Position.Latitude,

position.Coordinate.Point.Position.Longitude);
            return result;
        }
    }
}
```


Finalmente, para que los servicios de geolocalización funcionen en una aplicación UWP, deberás modificar el manifiesto de la aplicación para incluir la capacidad "Location" en tu proyecto, de lo contrario, cuando ejecutes la aplicación te arrojará una excepción.



Implementación de IGeolocationService en Android

En el proyecto de Android, agregaremos también un folder llamado "Services", el cual tendrá como objetivo organizar los servicios concretos que implementemos en la aplicación. Dentro de este folder crearemos una nueva clase llamada GeolocationService, tal y como muestra la siguiente figura:



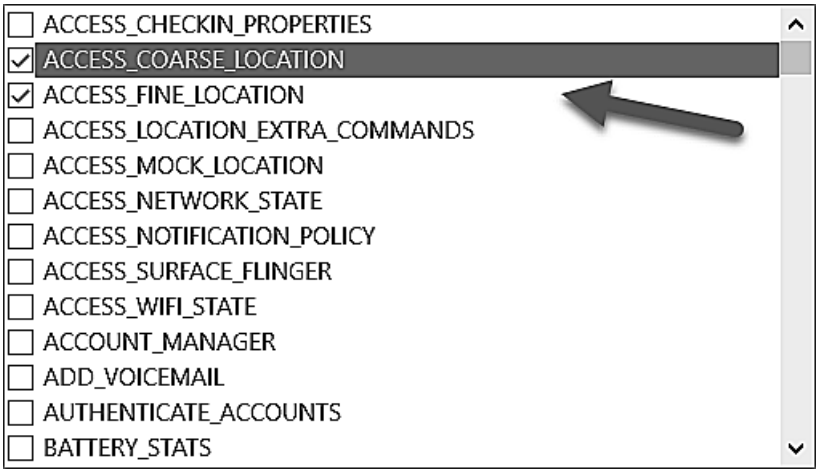
La clase GeolocationService debe implementar la interfaz IGeolocationService y cumplir con la funcionalidad requerida. En el API de Xamarin.Android contamos con la clase LocationManager, la cual nos permite obtener información acerca de los servicios de geolocalización del dispositivo. En nuestro caso, usaremos el método GetLastKnownLocation() para obtener la última ubicación geográfica conocida del proveedor de localización que cumpla con un criterio de exactitud alta. El siguiente código muestra la implementación de la clase GeolocationService:

```
using System;
using System.Threading.Tasks;
using Android.Content;
using Android.Locations;
using Surveys.Core.ServiceInterfaces;
using Surveys.Droid.Services;
using Xamarin.Forms;

[assembly:Dependency(typeof(GeolocationService))]

namespace Surveys.Droid.Services
{
```



```
      public class GeolocationService :
IGeolocationService
      {
          private readonly LocationManager
locationManager = null;

          public GeolocationService()
          {
              locationManager =
Xamarin.Forms.Forms.Context.GetSystemService(Context.Loca
tionService) as LocationManager;
          }

          public Task<Tuple<double, double>>
GetCurrentLocationAsync()
          {
              var provider =
locationManager.GetBestProvider(new Criteria() { Accuracy
= Accuracy.Fine }, true);

              var location =
locationManager.GetLastKnownLocation(provider);

              var result = new Tuple<double,
double>(location.Latitude, location.Longitude);

              return Task.FromResult(result);
          }
      }
  }
```

Igualmente, en el proyecto de Android tenemos que agregar en su manifiesto los permisos ACCESS_COARSE_LOCATION y ACCESS_FINE_LOCATION los cuales nos permiten tener acceso a los servicios de geolocalización. La siguiente figura muestra los cambios requeridos en el manifiesto de la aplicación Android:



Usando el Servicio de Dependencias

En la PCL, usaremos la clase DependencyService para obtener el objeto concreto que implementa la interfaz IGeolocationService, y de esa manera ejecutar la funcionalidad nativa a través del método GetCurrentLocationAsync(). Ya que la idea es georreferenciar cada encuesta una vez que haya sido capturada, el mejor lugar para hacer esto es el método EndSurveyCommandExecute() que es justamente el momento cuando estamos haciendo la instancia del objeto Survey.

El siguiente fragmento de código muestra las modificaciones a este método:

```
private async void EndSurveyCommandExecute()
{
    var newSurvey = new Survey() { Name = Name,
Birthdate = Birthdate, FavoriteTeam = FavoriteTeam };

    var geolocationService =
DependencyService.Get<IGeolocationService>();

    if (geolocationService != null)
    {
        try
        {
```



```
                var currentLocation = await
geolocationService.GetCurrentLocationAsync();
                newSurvey.Lat = currentLocation.Item1;
                newSurvey.Lon = currentLocation.Item2;
            }
            catch (Exception)
            {
                newSurvey.Lat = 0;
                newSurvey.Lon = 0;
            }
        }

        MessagingCenter.Send(this,
Messages.NewSurveyComplete, newSurvey);
    }
```

Modificando la Plantilla de Datos

Una última tarea antes de probar la aplicación: debemos modificar la Plantilla de Datos SurveyDataTemplate implementada en el diccionario de recursos de la página SurveysView. En la Plantilla de Datos agregaremos un StackLayout horizontal que incluya elementos de tipo Label enlazados a las propiedades Lat y Lon del objeto Survey. Además, especificaremos el formato de la cadena para usar únicamente 4 decimales.

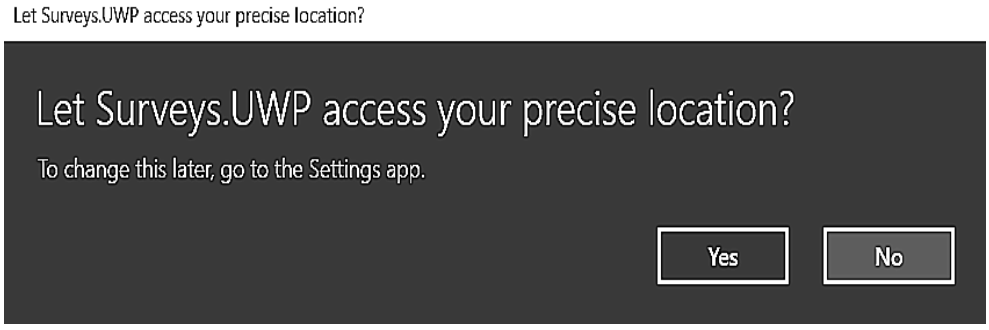
El siguiente fragmento de código muestra la Plantilla de Datos SurveyDataTemplate con las modificaciones:

```
<DataTemplate x:Key="SurveyDataTemplate">
  <ViewCell>
    <StackLayout>
      <Label Text="{Binding Name}"
             FontSize="24" />
      <Label Text="{Binding Birthdate,
StringFormat='{}{0:dd/MM/yyyy}'}" />
```

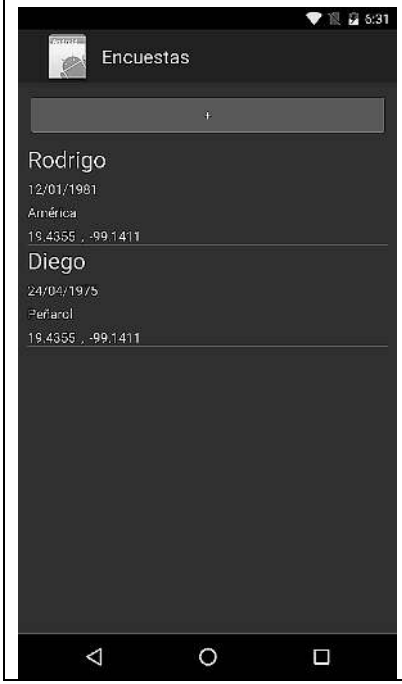
```
        <Label Text="{Binding FavoriteTeam}"
                TextColor="{Binding FavoriteTeam,
Converter={StaticResource TeamColorConverter}}" />
        <StackLayout Orientation="Horizontal">
          <Label Text="{Binding Lat,
StringFormat='{}{0:N4}'}" />
          <Label Text="," />
          <Label Text="{Binding Lon,
StringFormat='{}{0:N4}'}" />
        </StackLayout>
      </StackLayout>
    </ViewCell>
  </DataTemplate>
```

Probando la aplicación

Ha llegado el momento de probar nuestra aplicación. Después de capturar una encuesta vemos que efectivamente la encuesta es georreferenciada usando la funcionalidad que expone la interfaz IGeolocationService. En el caso de UWP en el escritorio, la primera vez que ejecutes la aplicación te preguntará si deseas permitir que dicha aplicación tenga acceso a los servicios de geolocalización, a lo que deberás responder que sí, de lo contrario no podrás obtener la ubicación geográfica:



Las siguientes figuras muestran la aplicación ejecutándose después de haber implementado los cambios de este capítulo:

Android	UWP
	

5:31
Encuestas
+
Rodrigo
12/01/1981
América
19.4355 , -99.1411
Diego
24/04/1975
Peñarol
19.4355 , -99.1411

Surveys UWP
Encuestas
+
Daniel
04/03/1957
Saprissa
19.4354 , -99.1411
Víctor
03/07/1977
Boca Juniors
19.4354 , -99.1411

9 ARQUITECTURA DE APLICACIONES CON PRISM

INTRODUCCIÓN

Hasta este momento, hemos utilizado el API propio de Xamarin.Forms para explicar la construcción de aplicaciones de negocio. No obstante, en aplicaciones empresariales reales, Xamarin.Forms puede estar limitado si lo que queremos es implementar las mejores prácticas arquitectónicas en el espacio del desarrollo móvil. Y es que, hay dos ejemplos muy sencillos de estas limitaciones: la infraestructura de navegación y las cajas de diálogo. Si has seguido en orden los capítulos de este libro, podrás haber apreciado que para navegar de una página a otra forzosamente lo tenemos que hacer a través de las páginas. De igual manera y complicando aún más el escenario, las cajas de diálogo también las podemos invocar únicamente a través de las instancias de las páginas.

Para tratar de solventar un poco estos obstáculos, nos hemos basado en utilizar el patrón de diseño Pub/Sub que ofrece la clase MessagingCenter para poder enviar mensajes entre los ViewModel y las Vistas de la aplicación; sin embargo, rápidamente se vuelve complejo el rastrear la secuencia y el flujo de mensajes entre todos los objetos. Adicionalmente, el código se enreda cada vez más y más, y enredarse no es nada bueno en el mundo del desarrollo de software. Al contrario, siempre debemos buscar la máxima limpieza y sencillez en nuestro código para que el mantenimiento del mismo sea más fácil. Recuerda que los desarrolladores pasamos más tiempo leyendo código que escribiendo código.

Y estos obstáculos no son inherentes únicamente a Xamarin.Forms. En general, e inspirados en estos obstáculos y otros motivos, históricamente se han creado diversos frameworks alrededor de las plataformas de desarrollo, las cuales nos ayudan y habilitan para implementar las mejores prácticas arquitectónicas de la tecnología en cuestión. Tal es el caso de Prism: un kit de desarrollo que incluye diversos componentes para implementar nociones de buenas prácticas de

arquitectura que serían difíciles de lograr por sí mismas en la plataforma. Comencemos entonces contestando: ¿qué es Prism?

¿Qué es Prism?

Prism es el nombre clave de un kit de desarrollo que ayuda al diseño, arquitectura y construcción de aplicaciones flexibles, desacopladas y que busquen como objetivo la facilidad en su mantenimiento. Prism no es nada nuevo, ya que era el nombre clave de un proyecto llamado Composite Application Guidance para WPF y Silverlight en el año 2008. Por costumbre y familiaridad, el nombre simplemente quedó como Prism.

Lo describo como un "kit de desarrollo" ya que no únicamente es un framework con clases que referenciamos en nuestras aplicaciones. También incluye documentación extensa, guías, plantillas para Visual Studio .NET, paquetes de NuGet y ejemplos de aplicaciones completas construidas con este framework. Por si fuera poco, Prism es parte del .NET Foundation y es completamente Open Source.

Prism se basa en algunos patrones de diseño que promueven principios fundamentales de arquitectura de software como la separación de preocupaciones y el desacoplamiento entre componentes, ambos incluidos en los principios básicos de la Programación Orientada a Objetos llamados SOLID (por sus siglas en inglés de: Single responsibility, Open/close, Liskov substitution, Interface segregation y Dependency inversion).

¿Por qué usar Prism?

Hay diversas consecuencias bastante positivas al utilizar un kit de desarrollo como Prism en el momento de crear la arquitectura de nuestras aplicaciones:

PROMUEVE EL DESACOPLAMIENTO ENTRE CLASES

Gracias a sus diversas interfaces y patrones de diseño que se fomentan al usar Prism, se reduce al máximo el acoplamiento entre las clases de tu aplicación.

PROMUEVE LA REUTILIZACIÓN DE FUNCIONALIDAD

Las clases desacopladas pueden ser reutilizadas de una manera más sencilla.

EVOLUCIÓN INDEPENDIENTE DE LAS PIEZAS

Las diversas piezas de tu aplicación pueden ser desarrolladas y probadas de manera independiente. Por lo tanto, cualquier modificación de una pieza no

afectaría al resto de piezas que conforman la aplicación. Además, una pieza podría evolucionar de manera independiente de las demás.

IMPULSA LA CREACIÓN DE PRUEBAS UNITARIAS

Tener clases desacopladas entre sí, permite que sea más fácil la creación de pruebas unitarias que comprueben la funcionalidad de cada una de ellas de forma independiente.

SE REDUCE LA DIFICULTAD DE AGREGAR NUEVAS CARACTERÍSTICAS AL SISTEMA

Siguiendo los principios SOLID, podríamos incorporar una nueva funcionalidad y características en una aplicación sin que eso impacte en el resto de componentes ya construidos.

RESUELVE LAS LIMITACIONES DE XAMARIN.FORMS

Específicamente en Xamarin.Forms, una consecuencia de usar Prism es solventar los límites de la infraestructura de navegación y cajas de diálogo que tiene la plataforma, debido a su total dependencia a las clases de las páginas. Adicionalmente, incluye implementaciones con mayor funcionalidad si las comparamos con algunos de los componentes incluidos en Xamarin.Forms, como veremos a continuación.

Versiones de Prism

Prism puede ser usado prácticamente en todas las plataformas de XAML en el ecosistema de Microsoft. Están disponibles las siguientes versiones:

- Prism para UWP
- Prism para WPF
- Prism para Silverlight
- Prism para Xamarin.Forms

¿Qué ofrece Prism para Xamarin.Forms?

Prism nos brinda diversos componentes y mejores prácticas como las que a continuación se describen:

Inyección de Dependencias con Unity	Prism usa de manera predeterminada Unity como contenedor de Inyección de Dependencias

Infraestructura de Navegación robusta	Prism incluye las interfaces INavigationService y INavigationAware como infraestructura de navegación, evitando el uso de la infraestructura interna de la plataforma (en Xamarin.Forms, la interfaz INavigation de la clase base Page)
Mensajería con EventAggregator	Prism incluye la clase EventAggregator como mecanismo para usar el patrón de diseño Pub/Sub, con más funcionalidad en comparación con la clase MessagingCenter de Xamarin.Forms
Servicios para cajas de diálogos	Prism incluye la interfaz IPageDialogService como mecanismo para invocar las cajas de diálogo desde los ViewModels
DelegateCommand	La clase DelegateCommand es la implementación de ICommand en Prism, no obstante, tiene más funcionalidad que la clase Command de Xamarin.Forms ya que nos permite observar el valor de propiedades para reevaluar el comando
Soporte para el Patrón de Diseño MVVM	Prism incluye la clase abstracta base BindableBase, la cual implementa la interfaz INotifyPropertyChanged
Logging	Prism incluye nociones de logging para auditoría en las aplicaciones

Principios SOLID

Como mencioné en páginas anteriores, Prism promueve los principios básicos en la Programación Orientada a Objetos llamados SOLID, identificados por Robert C. Martin. La siguiente tabla describe con un poco más de detalle a qué se refiere cada uno de ellos:

Single responsibility principle	Principio que sugiere que un elemento (clase, método, etc.) solo debe tener una sola responsabilidad
Open/close principle	Las clases, módulos, funciones, etc., deben estar abiertos a extensión pero cerrados para modificación

Liskov substitution principle	Si **S** es un subtipo de **T**, entonces los objetos de tipo **T** pueden ser reemplazados con objetos de tipo **S** sin alterar alguna de las propiedades deseables, como lo correcto del programa
Interface segregation principle	Las interfaces que son muy largas deben ser divididas en otras más específicas y pequeñas, para que las clases cliente solo necesiten saber acerca de los métodos que ellos usan: ninguna clase cliente debe ser forzada a depender de métodos que no usa
Dependency inversion principle	Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones. Las abstracciones no deben depender de los detalles: los detalles deben depender de abstracciones

Requerimientos

Para instalar Prism en nuestra aplicación de Xamarin.Forms, debemos descargar los siguientes paquetes de NuGet:

Descripción	Nombre del paquete	Versión mínima*
Prism para Xamarin.Forms	Prism.Forms	6.3.0
Unity para Prism para Xamarin.Forms	Prism.Unity.Forms	6.3.0
Prism	Prism.Core	6.3.0
Unity	Unity	4.0.1
Xamarin.Forms	Xamarin.Forms	2.3.3.180

*Nota: En el momento de estar escribiendo este capítulo la versión 6.3.0 de Prism está aún en fase previa (6.3.0 pre-1).

MODELO DE APLICACIÓN

Prism es como un buffet, en donde cada persona se sirve en su plato lo que quiere. Tiene algunas clases que pueden ser usadas de manera aislada, como por ejemplo DelegateCommand, EventAggregator o BindableBase. Sin embargo, si

queremos adoptar su infraestructura de navegación y cajas de diálogo generalmente debemos incorporar el Modelo de Aplicación propio de Prism y dejar a un lado el Modelo de Aplicación de Xamarin.Forms.

Clase PrismApplication

Prism incluye la clase base PrismApplication, la cual tiene como objetivo ser la clase base para las aplicaciones en Xamarin.Forms, sustituyendo la clase Xamarin.Forms.Application que hemos usado hasta este momento en nuestro código.

La clase PrismApplication tiene la particularidad de estar preconfigurada para utilizar Unity como el contenedor de Inyección de Dependencias, aunque también podemos usar cualquier otro como Ninject o Autofac. Incluso, en el sitio de GitHub de Prism podrás encontrar ejemplos de cómo utilizar estos y más contenedores si así lo requiere tu aplicación.

Nota: No confundir Unity (el contenedor de Inyección de Dependencias) con Unity (el motor de videojuegos 3D/2D). Si bien su nombre es el mismo, son cosas completamente diferentes. Unity es un contenedor de Inyección de Dependencias creado por el equipo de Patrones y Prácticas en Microsoft y es al que nos referimos en este libro.

El uso de un contenedor de Inyección de Dependencias en nuestras aplicaciones de Xamarin.Forms cambia totalmente la manera de ver y manejar los diversos componentes que las constituyen, ya que el ciclo de vida de los objetos no lo controlas tú (por ejemplo, instanciando manualmente los objetos), sino que Unity se encarga de inyectarlos a través del constructor de las clases. Más adelante en este mismo capítulo volveremos a este concepto tan valioso que nos brinda Prism en conjunto con Unity.

La clase PrismApplication cuenta con diversos miembros, pero destacan los siguientes:

MÉTODO ONINITIALIZED()

Después del constructor de la clase PrismApplication, este es el siguiente método que toca el flujo de ejecución en una aplicación Xamarin.Forms basada en Prism. Por esta razón, en este método se debe inicializar el diccionario de recursos global de la aplicación, a través de la ejecución del método InitializeComponent() que hemos usado hasta este momento en el código del libro.

Otra responsabilidad de este método será el de inicializar los módulos que contenga la aplicación (si hubiera alguno).

Adicionalmente, en este método navegamos a la página raíz de la aplicación. Con "navegar" me refiero a utilizar la propiedad NavigationService de la clase PrismApplication, la cual es una propiedad de tipo INavigationService y que incluye el método asíncrono NavigateAsync(). Por lo tanto, en una aplicación Prism en Xamarin.Forms ya no es necesario establecer el valor de la propiedad MainPage como lo habíamos hecho con anterioridad.

MÉTODO REGISTERTYPES()

En este método debemos registrar todos los tipos a los que se puede navegar en la aplicación de Xamarin.Forms con Prism, es decir, las Vistas. Esto lo logramos ejecutando el método genérico RegisterTypeForNavigation<T> del contenedor de Unity, expuesto a través de la propiedad Container en la clase PrismApplication.

MÉTODO CONFIGURECONTAINER()

En este método, debemos registrar los servicios adicionales que requiere la aplicación en el contenedor de Inyección de Dependencias expuesto a través de la propiedad Container de la clase base PrismApplication.

En el siguiente fragmento de código se muestra un ejemplo de implementación de PrismApplication en una clase llamada App:

```
public partial class App : PrismApplication
{
    protected override async void OnInitialized()
    {
        //Inicializa el diccionario de recursos global
de la aplicación
        InitializeComponent();

        //Navega a la página raíz de la aplicación
        await
NavigationService.NavigateAsync($"{nameof(LoginView)}");
    }

    protected override void RegisterTypes()
    {
```

```
Container.RegisterTypeForNavigation<LoginView>();
            //Resto de tipos a los que se puede navegar
        }
    }
```

CLASE BINDABLEBASE

Prism incluye la clase abstracta BindableBase, la cual implementa la interfaz INotifyPropertyChanged. Por tal motivo, podemos ahorrarnos la creación de una clase base abstracta con este propósito (tal y como lo hicimos con la clase NotificationObject en capítulos pasados).

El siguiente fragmento de código muestra la implementación de una clase base abstracta llamada ViewModelBase que hereda de BindableBase:

```
public abstract class ViewModelBase : BindableBase
{
    //Resto de funcionalidad base para los ViewModels
}
```

CLASE DELEGATECOMMAND

La clase DelegateCommand es la implementación que hace Prism de la interfaz ICommand. Como vimos en el capítulo 6, podemos definir los comandos como "funcionalidad enlazable", ya que los exponemos como propiedades en los ViewModel y nos enlazamos a ellos en los controles que los soporten, como los Button o los ToolbarItem.

DelegateCommand incluye los métodos ObservesProperty() y ObservesCanExecute(), los cuales extienden la funcionalidad básica de un comando de una manera bastante atractiva, tal y como lo describiremos a continuación.

ObservesProperty()

Con este método, podemos asignar una acción que regrese una propiedad que la infraestructura de enlace de datos estará "observando", de tal manera que si hubiese un cambio en ella, automáticamente se reevaluaría el comando, evitándonos disparar explícitamente el método de reevaluación, ya sea en el accesor set{} de las

propiedades o en el manejador del evento PropertyChanged del ViewModel. ObservesProperty() soporta una sintaxis fluida, por lo que podrás encadenar invocaciones a ObservesProperty() una después de otra.

El siguiente ejemplo indica que deseamos observar las propiedades Name y FavoriteTeam de la clase, y si hubiese algún cambio en ellas, se reevaluaría el comando.

```
  EndSurveyCommand = new
DelegateCommand(EndSurveyCommandExecute).ObservesProperty
(()=> Name).ObservesProperty(()=>FavoriteTeam);
```

ObservesCanExecute()

Similar a ObservesProperty(), este método nos permite definir una expresión booleana para determinar si el comando puede ejecutar o no.

El siguiente ejemplo muestra un código para evaluar que la colección Surveys tenga menos de 15 elementos.

```
  AddSurveyCommand = new
DelegateCommand(AddSurveyCommandExecute).ObservesCanExecu
te(o => Surveys.Count < 15);
```

NAVEGACIÓN

Si Prism fuera un buffet como dijimos anteriormente, entonces la infraestructura de navegación que incluye es el plato fuerte. Incluso, Xamarin ha anunciado que en próximas versiones de Xamarin.Forms adoptarán la infraestructura de navegación de Prism como sustitución a la que se tiene actualmente en la plataforma.

A continuación describiremos las dos interfaces más importantes de esta infraestructura:

INavigationService

La navegación en Prism se basa en la interfaz INavigationService. Esta interfaz incluye los siguientes métodos:

NavigateAsync()	Navega al Uri o string que especifiques en el argumento. La sintaxis para el Uri soporta "Deep Linking", esto es, la capacidad de especificar una secuencia de navegación entre todas las páginas. La Vista o Vistas que represente el Uri o string deberán haber sido registradas previamente en el método RegisterTypes() de la clase PrismApplication. De manera opcional, puedes pasar parámetros a través de un query string, o pasar un objeto de tipo NavigationParameters
GoBackAsync()	Regresa a la página anterior, removiendo de la pila de navegación la última página agregada. Opcionalmente, puedes pasar un objeto de tipo NavigationParameters

INavigationAware

Esta interfaz permite a los ViewModel estar al tanto de cuándo se ha navegado a las páginas de las que son contexto de enlace de datos. Los miembros de esta interfaz son los siguientes:

OnNavigatingTo()	Método que se ejecuta cuando se está navegando hacia la página de donde el ViewModel es contexto de enlace. Se ejecuta antes de OnNavigatedTo(). Tiene como argumento un objeto de tipo NavigationParameters
OnNavigatedTo()	Método que se ejecuta cuando se ha navegado hacia la página de donde el ViewModel es contexto de enlace. Se ejecuta después de OnNavigatingTo(). Tiene como argumento un objeto de tipo NavigationParameters
OnNavigatedFrom()	Método que se ejecuta cuando se ha navegado fuera de la página de donde el ViewModel es contexto de enlace. Tiene como argumento un objeto de tipo NavigationParameters

IPAGEDIALOGSERVICE

El servicio de cajas de diálogo que proporciona Prism está basado en la interfaz IPageDialogService. Esta interfaz nos permite desplegar desde los ViewModels las cajas de diálogo de tipo "alerta" y "action sheet" que tradicionalmente invocaríamos a través de los métodos DisplayAlert() y DisplayActionSheet() de las páginas.

El siguiente fragmento de código muestra la invocación del método DisplayAlertAsync() de la interfaz IPageDialogService:


```
  await pageDialogService.DisplayAlertAsync("Título",
"¿Está seguro(a)?", "Aceptar");
```

Tener ambas cajas de diálogo encapsuladas en esta interfaz es motivo suficiente para usarla. No obstante, por si fuera poco, su método DisplayActionSheet() también permite que usemos objetos que implementen la interfaz IActionSheetButton para la lista de elementos que deseamos desplegar en pantalla, y no únicamente cadenas de texto como sucede con la implementación propia de Xamarin.Forms. Esto es notable, ya que la interfaz IActionSheetButton incluye la propiedad ICommand, y por lo tanto podríamos ejecutar un comando cuando se seleccione un elemento.

INYECCIÓN DE DEPENDENCIAS

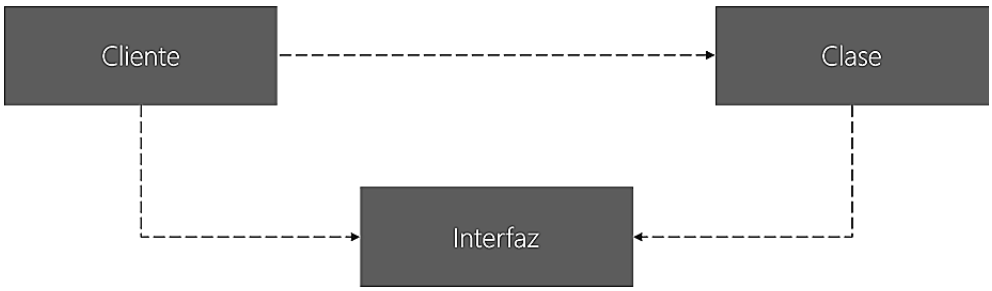
La Inyección de Dependencias es un patrón de diseño que busca desacoplar dos clases entre sí, en un escenario donde una clase es el "cliente" de la otra.

Por ejemplo, el siguiente fragmento de código muestra un acoplamiento total entre ambas clases:

```
//Clase Cliente
var x = new Clase();
```

Esto es debido a que Cliente requiere Clase, la referencia y depende de ella. El siguiente diagrama muestra este escenario:

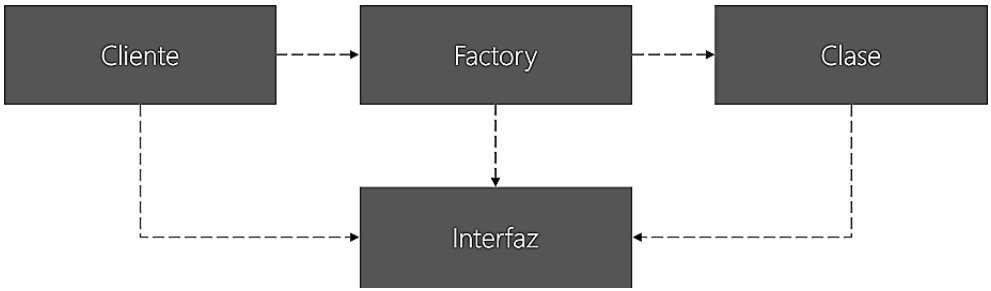
Un primer paso para desacoplar ambas clases sería el derivar una interfaz de Clase:



No obstante, sin un mecanismo adicional, el código sigue estando completamente acoplado, tal y como lo podemos apreciar en el siguiente fragmento de código:

```
Interfaz x = new Clase();
```

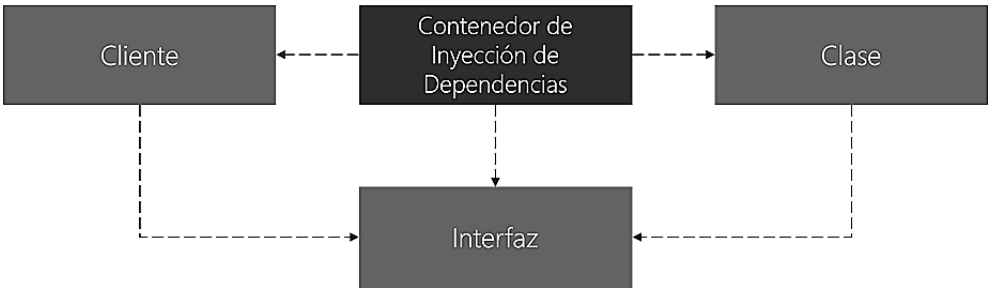
Un siguiente paso en la búsqueda de desacoplar ambas clases sería el implementar el patrón de diseño Factory, el cual incluye una clase (Factory) que crea las instancias del tipo que requiere la clase Cliente. El siguiente diagrama muestra este patrón:



De esta manera, el Cliente dependería de Factory en vez de directamente depender de Clase. Sin embargo, de ahí justamente viene su debilidad de este patrón, ya que Cliente ahora depende de totalmente de la clase responsable de crear los objetos de tipo Clase, es decir, de Factory. Adicionalmente, en este escenario, Cliente, además de las responsabilidades que en sí misma tiene, ahora también tiene la responsabilidad adicional de resolver su dependencia solicitando la instancia a Factory del tipo que requiere (Clase).

El patrón de Inyección de Dependencias corrige las debilidades del anterior escenario, ya que la única dependencia que tendría Cliente sería al tipo Interfaz. Adicionalmente, Cliente no tendría conocimiento de la clase que crea las instancias concretas del tipo Interfaz (Clase).

El siguiente diagrama muestra el patrón de diseño de Inyección de Dependencias:



Con este patrón, el contenedor de Inyección de Dependencias "empuja" o "inyecta" el objeto concreto (Clase) a Cliente, en vez de que Cliente "tire" de una clase "creadora" de objetos (Factory) de tipo Interfaz el objeto concreto. A este modelo que "empuja" los objetos se le denomina Inversión de Control, y la Inyección de Dependencias es una implementación de este mecanismo.

¿Cuándo se usa la Inyección de Dependencias en Prism?

Prism usa de manera predeterminada Unity como contenedor de Inyección de Dependencias. Por esta razón, sumándola a la explicación del patrón que acabamos de explicar, no tenemos que preocuparnos por crear nosotros las instancias de los servicios o de los ViewModels de la aplicación: Unity los creará por ti.

RESOLUCIÓN DE SERVICIOS DE PRISM

En el caso de los servicios de Prism, como por ejemplo INavigationService o IPageDialogService, cuando requieres utilizarlos simplemente los solicitamos como argumento en el constructor de las clases de los ViewModel. Por ejemplo, el siguiente fragmento de código muestra el constructor de la clase SurveysViewModel, el cual solicita como parámetro un objeto de tipo INavigationService. Unity, tomando su papel de contenedor de Inyección de Dependencias, te dará el objeto concreto que implementa la interfaz. En este caso, será un objeto de tipo Prism.Unity.Navigation.UnityPageNavigationService, pero eso no lo tienes que saber necesariamente, ya que esa lógica la encapsula Unity. ¡Esa es la belleza de usar la Inyección de Dependencias!

```
  public SurveysViewModel(INavigationService
navigationService)
  {
      this.navigationService = navigationService;
     //navigationService será de tipo
Prism.Unity.Navigation.UnityPageNavigationService
  }
```

RESOLUCIÓN DE VIEWMODELS

Cuando navegas a una página, Prism hace automáticamente la instancia del ViewModel relacionado con esa página y además lo asigna automáticamente como contexto de enlace de datos, es decir, lo establece como valor de la propiedad BindingContext de la página.

Prism se basa en una convención de nombres de las Vistas y ViewModels para inferir cuál es el ViewModel que debe instanciar. La nomenclatura básica que usa Prism es que ambas clases terminen con el sufijo "View" y "ViewModel", o con "Page" y "PageViewModel". La siguiente tabla muestra algunos nombres de clases válidos para que Prism automáticamente instancie el ViewModel y lo asigne como BindingContext de la Vista:

SurveysView	SurveysViewModel
SurveyDetailsView	SurveyDetailsViewModel
LoginPage	LoginPageViewModel

Nota: En la versión 6.2.0 de Prism para Xamarin.Forms, hay un bug en este proceso de resolución de ViewModels, que se produce cuando las Vistas están en un espacio de nombres diferente a los ViewModels, y se utiliza el sufijo "View" y "ViewModel". En Prism 6.3.0 pre-1 está corregido ese bug y justamente debido a ello es la versión que utilizaremos en este capítulo.

MANOS A LA OBRA

En este capítulo modificaremos la arquitectura de la aplicación para usar los siguientes componentes y nociones de Prism:

- Infraestructura de navegación (INavigationService y INavigationAware)
- Servicio de cajas de diálogo (IPageDialogService)
- DelegateCommand para todos los comandos
- BindableBase como clase base para todos los ViewModel
- Unity como contenedor de Inyección de Dependencias

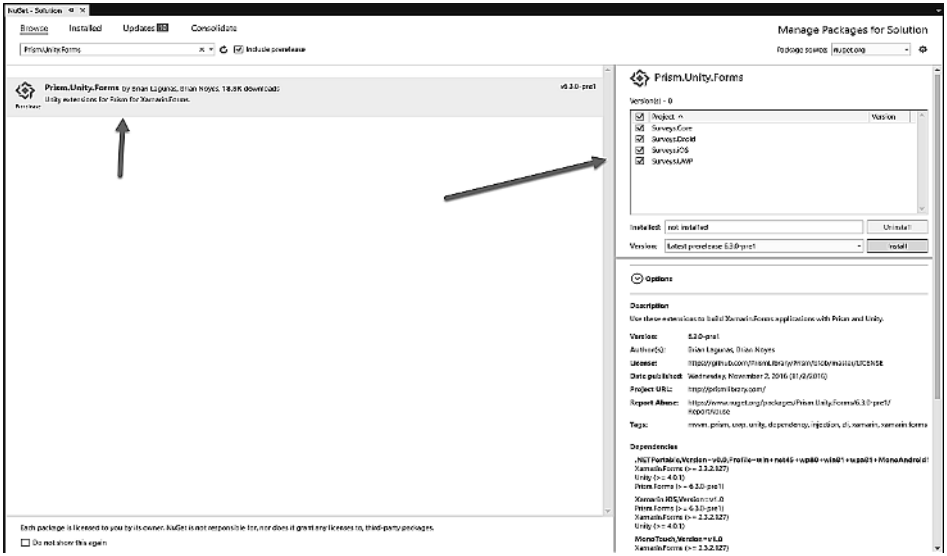
Adicionalmente, complementaremos la aplicación con una nueva estructura visual basada en el paradigma Maestro-Detalle. Por si fuera poco, agregaremos una página para autenticarnos a la aplicación y otra para mostrar datos informativos de la misma.

¡Comencemos!

Instalando Prism

Prism es distribuido a través de paquetes de NuGet. Ya que un paquete de NuGet declara sus dependencias, la manera más rápida de instalar Prism para Xamarin.Forms es buscar la última versión del paquete Prism.Unity.Forms.

Al instalar este paquete, se descargarán todas las dependencias requeridas. La siguiente figura muestra el panel de administración de paquetes NuGet en Visual Studio .NET. Es importante destacar que deberás seleccionar todos los proyectos de tu solución, en la lista que está en la parte izquierda:



Una vez instalados todos los paquetes de NuGet, el archivo packages.config de la PCL quedará de la siguiente manera:

```
<?xml version="1.0" encoding="utf-8"?>
<packages>
  <package id="CommonServiceLocator" version="1.3"
targetFramework="portable46-net451+win81" />
  <package id="Prism.Core" version="6.3.0-pre1"
targetFramework="portable46-net451+win81" />
  <package id="Prism.Forms" version="6.3.0-pre1"
targetFramework="portable46-net451+win81" />
  <package id="Prism.Unity.Forms" version="6.3.0-pre1"
targetFramework="portable46-net451+win81" />
  <package id="Unity" version="4.0.1"
targetFramework="portable46-net451+win81" />
  <package id="Xamarin.Forms" version="2.3.3.180"
targetFramework="portable46-net451+win81" />
</packages>
```

Implementando el Modelo de Aplicación

Prism para Xamarin.Forms incluye la clase base abstracta PrismApplication, la cual es la clase que configura Unity como contenedor de Inyección de Dependencias. Por lo tanto, modificaremos App.xaml y su archivo de code-behind para heredar de esta clase en vez de Xamarin.Forms.Application.

El siguiente fragmento de código muestra las modificaciones al archivo App.xaml. Observa que debemos importar el espacio de nombres que contiene la clase que deseamos instanciar en XAML, en este caso Prism.Unity del ensamblado Prism.Unity.Forms:

```
<?xml version="1.0" encoding="utf-8" ?>
<unity:PrismApplication
xmlns="http://xamarin.com/schemas/2014/forms"

xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
                xmlns:unity="clr-
namespace:Prism.Unity;assembly=Prism.Unity.Forms"
                x:Class="Surveys.Core.App">
    <unity:PrismApplication.Resources>
      <ResourceDictionary>
        <!--Recursos globales para la aplicación-->
      </ResourceDictionary>
    </unity:PrismApplication.Resources>
</unity:PrismApplication>
```

Asimismo, el archivo de code-behind debe ser modificado para heredar de la clase abstracta PrismApplication, y además implementar sus miembros marcados como abstractos. El siguiente fragmento de código muestra esta implementación básica en la clase App:

```
using Prism.Unity;
using Xamarin.Forms;

namespace Surveys.Core
{
    public partial class App : PrismApplication
```



```
    {
        public App()
        {
            InitializeComponent();
            MainPage = new NavigationPage(new
SurveysView());
        }

        protected override void OnInitialized()
        {
            throw new System.NotImplementedException();
        }

        protected override void RegisterTypes()
        {
            throw new System.NotImplementedException();
        }
    }
}
```

IMPLEMENTANDO ONINITIALIZED()

Ya que hemos heredado de PrismApplication, tenemos que implementar el método OnInitialized(), el cual tiene como objetivo inicializar el diccionario de recursos global de la aplicación, así como navegar a la página raíz haciendo uso del servicio INavigationService que está disponible a través de la propiedad NavigationService en PrismApplication.

Ya que nuestra aplicación de encuestas tiene dos páginas a las que navegamos entre sí, usaremos las características de Deep Linking que expone Prism para establecer como página raíz una página de navegación, y que la primera página a la que navegue sea SurveysView. Por lo tanto, debemos agregar a la PCL una nueva página de tipo NavigationPage a la que llamaremos RootNavigationView.

Sin embargo, al usar la plantilla de “Forms Xaml Page”, automáticamente creará una página de tipo ContentPage, por lo que debemos modificar el elemento raíz en el código XAML y eliminar el elemento Label que pone la plantilla. Además, debemos modificar la herencia de la clase en el archivo de code-behind.

El siguiente fragmento de código muestra el código XAML de RootNavigationPage:

```
<?xml version="1.0" encoding="utf-8" ?>
<NavigationPage
xmlns="http://xamarin.com/schemas/2014/forms"

xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"

x:Class="Surveys.Core.Views.RootNavigationView">
</NavigationPage>
```

El siguiente fragmento de código muestra el archivo de code-behind de RootNavigationPage:

```
using Xamarin.Forms;
namespace Surveys.Core.Views
{
    public partial class RootNavigationView :
NavigationPage
    {
        public RootNavigationView()
        {
            InitializeComponent();
        }
    }
}
```

De regreso al método OnInitialized(), navegamos a SurveysView por medio de RootNavigationView. El siguiente fragmento de código muestra el uso de Deep Linking:

```
await
NavigationService.NavigateAsync($"{nameof(RootNavigationV
iew)}/{nameof(SurveysView)}");
```

IMPLEMENTANDO REGISTERTYPES()

En el método RegisterTypes() de la clase PrismApplication debemos registrar todos los tipos “navegables”, es decir, todas las Vistas de la aplicación. Esto es

absolutamente necesario, ya que de lo contrario al ejecutar la aplicación te arrojaría un error indicando precisamente esta omisión.

El siguiente fragmento de código muestra el registro de las tres páginas que tenemos en este momento en nuestra aplicación de encuestas:

```
protected override void RegisterTypes()
{

Container.RegisterTypeForNavigation<RootNavigationView>()
;
      Container.RegisterTypeForNavigation<SurveysView>();

Container.RegisterTypeForNavigation<SurveyDetailsView>();
}
```

Si ejecutamos y probamos la aplicación en este momento, podrás apreciar que efectivamente la primera página será SurveysView y podemos navegar de ida y de regreso a SurveyDetailsView sin problema alguno.

Implementando la clase BindableBase

En la implementación actual de la aplicación de encuestas, tenemos la clase NotificationObject la cual implementa la interfaz INotifyPropertyChanged. No obstante, borraremos esa clase para favorecer el uso de BindableBase. Sin embargo, crearemos primero una clase abstracta llamada ViewModelBase. Esta clase será la clase base para todos los ViewModels de la aplicación.

El siguiente fragmento de código muestra la implementación inicial de ViewModelBase:

```
using Prism.Mvvm;
namespace Surveys.Core
{
    public abstract class ViewModelBase : BindableBase
    {
    }
}
```

Una vez creada esta clase, borramos la clase NotificationObject del proyecto PCL y corregimos la herencia de ambas clases SurveysViewModel y SurveyDetailsViewModel:

```
public class SurveysViewModel : ViewModelBase { … }
public class SurveyDetailsViewModel : ViewModelBase { … }
```

Implementando la infraestructura de navegación

En la implementación actual de la aplicación de encuestas, hacemos uso excesivo de la publicación y suscripción de mensajes a través de MessagingCenter. Esto fue necesario, ya que no teníamos otra forma de poder desacoplar la navegación de las clases ViewModel.

Como lo mencionamos anteriormente, Prism cuenta con la interfaz INavigationService la cual nos permite navegar entre diferentes páginas desde los ViewModels. Por este motivo, debemos hacer que las clases de ViewModels estén alertas cuando se navega hacia o desde las páginas relacionadas, es decir, debemos implementar la interfaz INavigationAware cuyo objetivo es justamente el de dar esta noción de alerta a las clases de ViewModels.

Siendo así, y aprovechando la nueva clase base ViewModelBase que acabamos de crear, es sumamente fácil llevar a cabo esta labor. Adicionalmente, marcaremos los métodos de la interfaz INavigationAware como virtuales, para que los ViewModels concretos sean los que implementen la lógica en cada uno de ellos.

El siguiente código muestra la implementación completa de la clase ViewModelBase:

```
using Prism.Mvvm;
using Prism.Navigation;


namespace Surveys.Core
{
    public abstract class ViewModelBase : BindableBase,
INavigationAware
    {
        public virtual void
OnNavigatedFrom(NavigationParameters parameters)
```



```
            {
            }

            public virtual void
OnNavigatedTo(NavigationParameters parameters)
            {
            }

            public virtual void
OnNavigatingTo(NavigationParameters parameters)
            {
            }
        }
    }
```

Gracias al anterior cambio en ViewModelBase, ahora todos los ViewModel serán alertados cuando se navegue a sus páginas.

Inyección de Dependencias

Con todas las modificaciones al momento, estamos en un buen punto para modificar los ViewModels actuales para aprovechar el patrón de Inyección de Dependencias que promueve Unity a través de Prism. Comenzaremos con la clase SurveysViewModel y su página SurveysView.

SURVEYSVIEWMODEL Y SURVEYSVIEW

En el comando NewSurveyCommand de la clase SurveysViewModel, publicamos el mensaje Messages.NewSurvey, al cual está suscrita la página SurveysView. Como la intención de esa implementación era la de navegar a la página de detalle, ahora ya no será necesario usar MessagingCenter, debido a que todo lo podemos hacer ahora desde el ViewModel. ¿Cómo usar la infraestructura de navegación de Prism expuesta a través de INavigationService? La respuesta está en modificar el constructor de la clase SurveysViewModel y solicitar como argumento en el constructor un objeto de tipo INavigationService. Cuando el flujo de ejecución de la aplicación navegue a SurveysView, Unity inyectará en el constructor de SurveysViewModel el objeto concreto de tipo INavigationService. Al obtenerlo, lo guardaremos en un campo privado a nivel de clase para poder utilizarlo en el resto de miembros.

Adicionalmente, debemos sobrescribir el método OnNavigatedTo para determinar si existe un parámetro que incluya la nueva encuesta que hayamos capturado en SurveyDetailsView. Este método entonces sería el lugar adecuado para agregar a la colección Surveys la nueva encuesta capturada, tal y como lo muestra el siguiente fragmento de código:

```
public override void OnNavigatedTo(NavigationParameters
parameters)
{
    base.OnNavigatedTo(parameters);

    if (parameters.ContainsKey("NewSurvey"))
    {
        Surveys.Add(parameters["NewSurvey"] as Survey);
    }
}
```

El siguiente código muestra la nueva implementación completa de la clase SurveysViewModel. Observa la ausencia de MessagingCenter y la implementación del método OnNavigatedTo:

```
using System.Collections.ObjectModel;
using System.Windows.Input;
using Prism.Navigation;
using Xamarin.Forms;

namespace Surveys.Core
{
    public class SurveysViewModel : ViewModelBase
    {
        private INavigationService navigationService =
null;

        #region Properties

        private ObservableCollection<Survey> surveys;
```



```
public ObservableCollection<Survey> Surveys
{
    get
    {
        return surveys;
    }
    set
    {
        if (surveys == value)
        {
            return;
        }
        surveys = value;
        OnPropertyChanged();
    }
}

private Survey selectedSurvey;

public Survey SelectedSurvey
{
    get
    {
        return selectedSurvey;
    }
    set
    {
        if (selectedSurvey == value)
        {
            return;
```

```
                }
                selectedSurvey = value;
                OnPropertyChanged();
            }
        }

        #endregion

        public ICommand NewSurveyCommand { get; set; }

        public SurveysViewModel(INavigationService
navigationService)
        {
            this.navigationService = navigationService;

            Surveys = new
ObservableCollection<Survey>();

            NewSurveyCommand = new
Command(NewSurveyCommandExecute);
        }

        private async void NewSurveyCommandExecute()
        {
            await
navigationService.NavigateAsync(nameof(SurveyDetailsView)
);
        }

        public override void
OnNavigatedTo(NavigationParameters parameters)
        {
            base.OnNavigatedTo(parameters);

            if (parameters.ContainsKey("NewSurvey"))
            {
```

```
                    Surveys.Add(parameters["NewSurvey"] as
Survey);
                }
            }
        }
    }
```

En el caso del código XAML de SurveysView, ya no será necesario que instanciemos nosotros el objeto de tipo SurveysViewModel y lo asignemos como BindingContext de la página, ya que Prism y Unity se encargarán de ello gracias a que ambas clases respetan la nomenclatura que explicamos con anterioridad. Por lo tanto, borraremos ese código.

El siguiente código muestra el código XAML completo de la página SurveysView una vez hechos estos cambios:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage
xmlns="http://xamarin.com/schemas/2014/forms"

xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
                xmlns:core="clr-
namespace:Surveys.Core;assembly=Surveys.Core"
                x:Class="Surveys.Core.SurveysView"
                Title="Encuestas">
    <ContentPage.Resources>
      <ResourceDictionary>
        <core:TeamColorConverter
x:Key="TeamColorConverter" />

        <DataTemplate x:Key="SurveyDataTemplate">
          <ViewCell>
            <StackLayout>
              <Label Text="{Binding Name}"
                     FontSize="24" />
```

```
              <Label Text="{Binding Birthdate,
StringFormat='{}{0:dd/MM/yyyy}'}" />
              <Label Text="{Binding FavoriteTeam}"
                     TextColor="{Binding FavoriteTeam,
Converter={StaticResource TeamColorConverter}}" />
              <StackLayout Orientation="Horizontal">
                <Label Text="{Binding Lat,
StringFormat='{}{0:N4}'}" />
                <Label Text="," />
                <Label Text="{Binding Lon,
StringFormat='{}{0:N4}'}" />
              </StackLayout>
            </StackLayout>
          </ViewCell>
        </DataTemplate>
      </ResourceDictionary>
    </ContentPage.Resources>

    <Grid Margin="10">
      <ListView ItemsSource="{Binding Surveys}"
                SelectedItem="{Binding SelectedSurvey,
Mode=TwoWay}"
                ItemTemplate="{StaticResource
SurveyDataTemplate}"
                HasUnevenRows="True" />
    </Grid>

    <ContentPage.ToolbarItems>
      <ToolbarItem Text="Nueva"
                   Command="{Binding NewSurveyCommand}">
        <ToolbarItem.Icon>
          <OnPlatform x:TypeArguments="FileImageSource"
                      Android="newcapture.png"
```



```
                    iOS="newcapture.png"
                    WinPhone="assets/newcapture.png" />
        </ToolbarItem.Icon>
      </ToolbarItem>
    </ContentPage.ToolbarItems>
  </ContentPage>
```

SURVEYDETAILSVIEWMODEL Y SURVEYDETAILSVIEW

De igual forma, modificaremos el constructor de la clase SurveyDetailsViewModel. No obstante, en esta clase debemos solicitar ambas interfaces INavigationService y IPageDialogService, ya que como recordarás, necesitamos mostrar el Action Sheet con la lista de equipos. Un cambio inmediato es modificar la acción relacionada con el comando SelectTeamCommand, tal y como lo muestra el siguiente fragmento de código:

```
  private async void SelectTeamCommandExecute()
  {
      var team = await
pageDialogService.DisplayActionSheetAsync(Literals.Favori
teTeamTitle, null, null, Teams.ToArray());
      FavoriteTeam = team;
  }
```

Otra modificación necesaria es remover MessagingCenter en la acción del comando EndSurveyCommand, y ejecutar el método GoBackAsync() del objeto INavigationService. En este método enviamos como parámetro el objeto de la nueva encuesta capturada, tal y como lo muestra el siguiente fragmento de código:

```
  await navigationService.GoBackAsync(new
NavigationParameters { { "NewSurvey", newSurvey } });
```

Al ejecutar la línea de código anterior, la aplicación navegará de regreso a SurveysView, y se invocará el método OnNavigatedTo de SurveysViewModel, obteniendo el parámetro y agregándolo a la colección Surveys para mostrarlo en la lista.

A continuación se muestra el código completo de SurveysViewModel una vez efectuadas las modificaciones:

```
  using System;
  using System.Collections.ObjectModel;
  using System.Linq;
  using System.Windows.Input;
  using Prism.Navigation;
  using Prism.Services;
  using Surveys.Core.ServiceInterfaces;
  using Xamarin.Forms;
  namespace Surveys.Core
  {
      public class SurveyDetailsViewModel : ViewModelBase
      {
          private INavigationService navigationService =
null;
          private IPageDialogService pageDialogService =
null;

          #region Properties

          private string name;

          public string Name
          {
              get
              {
                  return name;
              }
              set
              {
                  if (name == value)
                  {
                      return;
                  }
```

```
            name = value;
            OnPropertyChanged();
        }
    }

    private DateTime birthdate;

    public DateTime Birthdate
    {
        get
        {
            return birthdate;
        }
        set
        {
            if (birthdate == value)
            {
                return;
            }
            birthdate = value;
            OnPropertyChanged();
        }
    }

    private string favoriteTeam;

    public string FavoriteTeam
    {
        get
        {
            return favoriteTeam;
        }
        set
```

```
        {
            if (favoriteTeam == value)
            {
                return;
            }
            favoriteTeam = value;
            OnPropertyChanged();
        }
    }

    private ObservableCollection<string> teams;

    public ObservableCollection<string> Teams
    {
        get
        {
            return teams;
        }
        set
        {
            if (teams == value)
            {
                return;
            }
            teams = value;
            OnPropertyChanged();
        }
    }

    #endregion

    public ICommand SelectTeamCommand { get; set; }
```

```
        public ICommand EndSurveyCommand { get; set; }

        public
SurveyDetailsViewModel(INavigationService
navigationService, IPageDialogService pageDialogService)
        {
            this.navigationService = navigationService;
            this.pageDialogService = pageDialogService;

            Teams =
                new ObservableCollection<string>(new[]
                {
                    "Alianza Lima", "América", "Boca
Juniors", "Caracas FC", "Colo-Colo", "Peñarol", "Real
Madrid",
                    "Saprissa"
                });

            SelectTeamCommand = new
Command(SelectTeamCommandExecute);
            EndSurveyCommand = new
Command(EndSurveyCommandExecute,
EndSurveyCommandCanExecute);


            PropertyChanged +=
SurveyDetailsViewModel_PropertyChanged;
        }

        private void
SurveyDetailsViewModel_PropertyChanged(object sender,

System.ComponentModel.PropertyChangedEventArgs e)
        {
            if (e.PropertyName == nameof(Name) ||
e.PropertyName == nameof(FavoriteTeam))
            {
```

```
                (EndSurveyCommand as
Command)?.ChangeCanExecute();
            }
        }

        private async void SelectTeamCommandExecute()
        {
            var team = await
pageDialogService.DisplayActionSheetAsync(Literals.Favori
teTeamTitle, null, null,
                Teams.ToArray());
            FavoriteTeam = team;
        }

        private async void EndSurveyCommandExecute()
        {
            var newSurvey = new Survey() { Name = Name,
Birthdate = Birthdate, FavoriteTeam = FavoriteTeam };


            var geolocationService =
Xamarin.Forms.DependencyService.Get<IGeolocationService>(
);
            if (geolocationService != null)
            {
                try
                {
                    var currentLocation = await
geolocationService.GetCurrentLocationAsync();
                    newSurvey.Lat =
currentLocation.Item1;
                    newSurvey.Lon =
currentLocation.Item2;
                }
                catch (Exception)
```

```
                {
                    newSurvey.Lat = 0;
                    newSurvey.Lon = 0;
                }
            }

            await navigationService.GoBackAsync(new
NavigationParameters { { "NewSurvey", newSurvey } });
        }

        private bool EndSurveyCommandCanExecute()
        {
            return !string.IsNullOrWhiteSpace(Name) &&
!string.IsNullOrWhiteSpace(FavoriteTeam);
        }
    }
}
```

Por su parte, en el código XAML de la página SurveyDetailsView debemos quitar la instancia de SurveyDetailsViewModel del diccionario de recursos, así como la asignación de la propiedad BindingContext de la página.

El siguiente código XAML muestra la implementación completa de SurveyDetailsView, una vez hechos los cambios anteriormente descritos:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage
xmlns="http://xamarin.com/schemas/2014/forms"

xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="Surveys.Core.SurveyDetailsView"
             Title="Nueva encuesta">

  <Grid Margin="10">
    <StackLayout>
      <Label Text="Nombre" />
      <Entry Text="{Binding Name, Mode=TwoWay}" />
```

```
        <Label Text="Fecha de nacimiento" />
        <DatePicker Date="{Binding Birthdate,
Mode=TwoWay}" />

        <Label Text="Equipo favorito" />
        <Grid>
          <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition Width="Auto" />
          </Grid.ColumnDefinitions>
          <Label Text="{Binding FavoriteTeam}" />
          <Button Grid.Column="1"
                  Text="..."
                  Command="{Binding SelectTeamCommand}"
/>
        </Grid>
      </StackLayout>
    </Grid>

    <ContentPage.ToolbarItems>
      <ToolbarItem Text="Aceptar"
                   Command="{Binding EndSurveyCommand}">
        <ToolbarItem.Icon>
          <OnPlatform x:TypeArguments="FileImageSource"
                      Android="ok.png"
                      iOS="ok.png"
                      WinPhone="assets/ok.png" />
        </ToolbarItem.Icon>
      </ToolbarItem>
    </ContentPage.ToolbarItems>
  </ContentPage>
```

Finalmente, el archivo de code-behind quedaría completamente limpio después de quitar la lógica que utilizaba MessagingCenter:

```
using Xamarin.Forms;

namespace Surveys.Core
{
    public partial class SurveyDetailsView :
ContentPage
    {
        public SurveyDetailsView()
        {
            InitializeComponent();
        }
    }
}
```

Si ejecutamos la aplicación en este momento, podrás observar que la funcionalidad inicial continúa. No obstante, estamos utilizando ya la infraestructura de navegación de Prism y su servicio para desplegar las cajas de diálogo.

Implementando DelegateCommand

Para concluir con la implementación de Prism en nuestra aplicación de encuestas, implementaremos todos los comandos como DelegateCommand, en vez de usar la clase Command de Xamarin.Forms.

SURVEYSVIEWMODEL

En el caso de la clase SurveysViewModel, simplemente modificamos la inicialización del comando NewSurveyCommand, para usar la clase DelegateCommand, tal y como lo muestra el siguiente fragmento de código:

```
NewSurveyCommand = new
DelegateCommand(NewSurveyCommandExecute);
```

SURVEYDETAILSVIEWMODEL

En SurveyDetailsViewModel tenemos los comandos SelectTeamCommand y EndSurveyCommand. Para el caso específico de EndSurveyCommand, estamos manejando el evento PropertyChanged del ViewModel para reevaluar el comando, para conocer cuándo las propiedades Name o FavoriteTeam hayan sido modificadas.

Aprovechando el método adicional ObservesProperty() que incluye la clase DelegateCommand, podríamos evitar el uso de ese manejador del evento

PropertyChanged, con menos código y una implementación más limpia, tal y como se muestra en el siguiente fragmento de código:

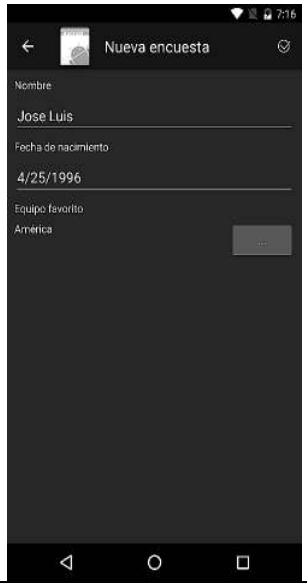
```
  SelectTeamCommand = new
DelegateCommand(SelectTeamCommandExecute);

  EndSurveyCommand =

      new DelegateCommand(EndSurveyCommandExecute,
EndSurveyCommandCanExecute)

          .ObservesProperty(() => Name)

          .ObservesProperty(() => FavoriteTeam);
```

Probando la aplicación con Prism

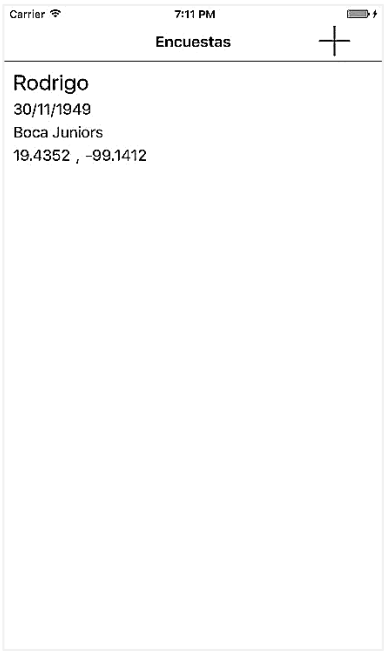
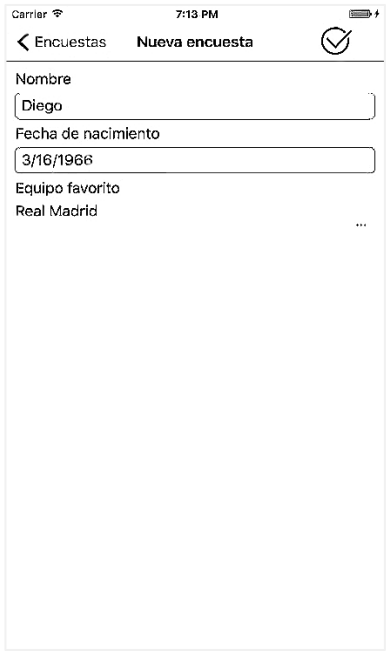
Ha sido un viaje largo, pero ha valido la pena: nuestra aplicación tiene una arquitectura mucho más desacoplada y con mejores prácticas, gracias a la implementación que hemos hecho de Prism y Unity.

Si ejecutas la aplicación en este momento, podrás observar que la funcionalidad de captura de nuevas encuestas sigue intacta, como lo demuestran las siguientes figuras:

ANDROID

SurveysView	SurveyDetailsView
	

IOS

SurveysView	SurveyDetailsView
	

Complementando la aplicación con más páginas

Con solamente dos páginas hemos demostrado una enormidad de características de la plataforma de desarrollo Xamarin.Forms y aún más usando un kit de desarrollo como Prism. Sin embargo, en una aplicación de negocio regularmente necesitamos más páginas y probablemente otra estructura visual más acorde con cualquier aplicación moderna. En consecuencia, agregaremos tres nuevas páginas a nuestra aplicación de encuestas:

MainView	MasterDetailPage	Página que tendrá como función estructurar la aplicación de una forma Maestro–Detalle
LoginView	ContentPage	Página que permita al usuario introducir sus credenciales para autenticación
AboutView	ContentPage	Página para mostrar el "Acerca de…" de la aplicación de encuestas

IMPLEMENTANDO LA PÁGINA DE ACERCA DE…

La página AboutView es una página regular de tipo ContentPage, que tendrá datos informativos de la aplicación de encuestas, como el nombre y una descripción. La agregaremos en el folder "Views" de la PCL usando la plantilla "Forms Xaml Page". Asimismo, agregaremos su ViewModel llamado AboutViewModel en el folder "ViewModels".

El siguiente código muestra la implementación completa de la página AboutView.xaml:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage
xmlns="http://xamarin.com/schemas/2014/forms"

xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="Surveys.Core.Views.AboutView">
  <ContentPage.Resources>
    <ResourceDictionary>
      <Style TargetType="Label">
        <Setter Property="HorizontalTextAlignment"
                Value="Center" />
      </Style>
    </ResourceDictionary>
  </ContentPage.Resources>

  <StackLayout VerticalOptions="Center"
               Margin="10">
    <Label Text="Encuestas"
           FontSize="Large" />
    <Label Text="Una aplicación de ejemplo del libro"
/>
    <Label Text="Xamarin.Forms en acción: Aplicaciones
de Negocio" />
    <Label Text="Escrito por Rodrigo Díaz Concha" />
```



```
    </StackLayout>
</ContentPage>
```

Con respecto a AboutViewModel, en este momento simplemente será una clase que herede de ViewModelBase, ya que no requerimos funcionalidad adicional:

```
namespace Surveys.Core.ViewModels
{
    public class AboutViewModel : ViewModelBase
    {
    }
}
```

IMPLEMENTANDO LA PÁGINA MAESTRO-DETALLE

La página MainView definirá una estructura "Maestro-Detalle" para la aplicación. Su propiedad Master representará la lista de "módulos" a los que podemos navegar. Inicialmente y de manera fija serán "Encuestas" y "Acerca de…" pero gracias a esta estructura, podríamos agregar más opciones de una forma rápida y sencilla, simplemente agregando otro módulo a la colección.

Agregaremos al folder "Views" del proyecto PCL una nueva página llamada MainView usando la plantilla "Forms Xaml Page". Una vez creada la página, modificaremos su elemento raíz en XAML y su herencia en el code-behind para declararla como una página de tipo MasterDetailPage.

Las páginas de tipo MasterDetailPage tienen la particularidad de requerir un valor en sus propiedades "Master" y "Detail", ambas de tipo Page. En nuestra implementación, asignaremos directamente en el código de XAML un ContentPage como valor de la propiedad Master. El contenido de este ContentPage es un control de tipo ListView, cuya fuente de datos está enlazada a la propiedad "Modules" del ViewModel. Asimismo, la propiedad SelectedItem de la lista estará enlazada de manera bidireccional a la propiedad SelectedModule del ViewModel. Esta implementación permitirá rastrear cuál es el módulo que seleccionó el usuario en la lista, de una manera limpia y respetando el patrón de diseño MVVM, a diferencia de manejar el evento ItemSelected del control ListView.

Como el control ListView será el responsable de desplegar la lista de los módulos, implementaremos una Plantilla de Datos que incluya el icono y el título del módulo en cuestión. Además, asignaremos la propiedad Header del ListView para mostrar el nombre de la aplicación.

El siguiente código muestra la implementación completa de la página MainView.xaml:

```
<?xml version="1.0" encoding="utf-8" ?>
<MasterDetailPage
xmlns="http://xamarin.com/schemas/2014/forms"

xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="Surveys.Core.Views.MainView">
    <MasterDetailPage.Master>
      <ContentPage Title="Encuestas">
        <ListView ItemsSource="{Binding Modules}"
                  SelectedItem="{Binding SelectedModule,
Mode=TwoWay}"
                  SeparatorColor="Transparent"
                  HasUnevenRows="True"
                  BackgroundColor="#3D3D3D">
          <ListView.Header>
            <Grid BackgroundColor="#3578FF">
              <Label Text="Encuestas"
                     FontSize="40"
                     TextColor="White"
                     Margin="10, 40, 0, 40" />
            </Grid>
          </ListView.Header>

          <ListView.ItemTemplate>
            <DataTemplate>
              <ViewCell>
                <Grid>
                  <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="Auto" />
                    <ColumnDefinition />
```



```
                    </Grid.ColumnDefinitions>

                    <Image Source="{Binding Icon}"
                           WidthRequest="50"
                           HeightRequest="50" />
                    <Label Text="{Binding Text}"
                           Grid.Column="1"
                           TextColor="White"
                           FontSize="Large"
                           VerticalTextAlignment="Center"
                           Margin="0, 10, 0, 10" />
                </Grid>
              </ViewCell>
            </DataTemplate>
          </ListView.ItemTemplate>
        </ListView>
      </ContentPage>
    </MasterDetailPage.Master>
  </MasterDetailPage>
```

Implementando la clase Module

Debemos implementar una nueva clase llamada “Module”, en el folder “Models” de la PCL. Esta clase Module representa un módulo dentro de la aplicación de encuestas. La siguiente tabla describe los miembros que implementaremos en esta clase:

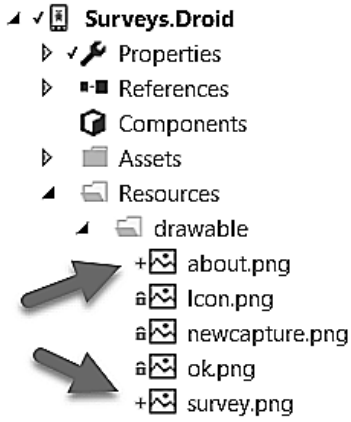
Icon	string	Representa la imagen que usaremos como icono para cada módulo en la lista
Title	string	Representa el título del módulo que será mostrado en la lista
LoadModuleCommand	ICommand	Comando que se ejecutará al seleccionar un módulo en la lista

El siguiente código muestra la implementación completa de esta clase. Observa que no necesitamos hacer esta clase "notificable", ya que sus datos son prácticamente fijos y no cambiarán durante la ejecución de la aplicación.

```
using System.Windows.Input;
namespace Surveys.Core.Models
{
    public class Module
    {
        public string Icon { get; set; }
        public string Title { get; set; }
        public ICommand LoadModuleCommand { get; set; }
    }
}
```

Agregando las imágenes para los módulos

Ya que cada módulo tendrá un icono relacionado, debemos agregar los archivos de imagen a los diferentes proyectos de plataformas concretas en la solución. Estos archivos se llamarán "about.png" y "survey.png". La siguiente figura muestra el proyecto de Android con los nuevos archivos agregados:



Implementando MainViewModel

Agregaremos una nueva clase llamada MainViewModel al folder "ViewModels" del proyecto PCL. En esta clase, implementaremos los miembros que a continuación se describen:

Modules	ObservableCollection<Module>	Representa la colección de módulos de la aplicación. Esta propiedad es la fuente de datos del control ListView en la Vista.
SelectedModule	Module	Representa el módulo seleccionado en la vista. Esta propiedad está enlazada de manera bidireccional a la propiedad SelectedItem del control ListView.

En el constructor de la clase MainViewModel solicitaremos como argumento INavigationService, para poder navegar a la vista adecuada cada vez que se seleccione un módulo. También en el constructor crearemos la colección Modules que queremos presentar en la lista. Cada objeto de tipo Module será el responsable de indicar su icono, su título y la acción que ejecutará al seleccionarlo en la lista. La acción de cada uno será navegar usando Deep Linking a la página principal relacionada con dicho módulo.

El siguiente código muestra la implementación completa de la clase MainViewModel, con todos los requerimientos funcionales que acabo de describir:

```
using System.Collections.ObjectModel;
using Prism.Commands;
using Prism.Navigation;
using Surveys.Core.Models;
using Surveys.Core.Views;
namespace Surveys.Core.ViewModels
{
    public class MainViewModel : ViewModelBase
    {
        private ObservableCollection<Module> modules;

        public ObservableCollection<Module> Modules
        {
            get
            {
```

```
            return modules;
        }
        set
        {
            if (modules == value)
            {
                return;
            }
            modules = value;
            OnPropertyChanged();
        }
    }

    private Module selectedModule;
    public Module SelectedModule
    {
        get
        {
            return selectedModule;
        }
        set
        {
            if (selectedModule == value)
            {
                return;
            }
            selectedModule = value;
            OnPropertyChanged();
        }
    }
```

```
        public MainViewModel(INavigationService
navigationService)
        {
            Modules = new ObservableCollection<Module>
            {
                new Module()
                {
                    Icon = "survey.png",
                    Title = "Encuestas",
                    LoadModuleCommand =
                        new DelegateCommand(
                            async () =>
                                await
navigationService.NavigateAsync(

$"{nameof(RootNavigationView)}/{nameof(SurveysView)}"))
                },
                new Module()
                {
                    Icon = "about.png",
                    Title = "Acerca de...",
                    LoadModuleCommand =
                        new DelegateCommand(
                            async () =>
                                await
navigationService.NavigateAsync(

$"{nameof(RootNavigationView)}/{nameof(AboutView)}"))
                }
            };
```

```
                PropertyChanged +=
MainViewModel_PropertyChanged;
            }

            private void
MainViewModel_PropertyChanged(object sender,
System.ComponentModel.PropertyChangedEventArgs e)
            {
                if (e.PropertyName ==
nameof(SelectedModule))
                {

SelectedModule?.LoadModuleCommand.Execute(null);
                }
            }
        }
    }
```

Como podrás apreciar en el código anterior, el módulo de "Encuestas" navegará a RootNavigationView/SurveysView, mientras que el módulo de "Acerca de…" navegará a RootNavigationView/AboutView. Observa además que para evitar las "cadenas mágicas", estoy utilizando la sentencia nameof() del lenguaje C# 6.0.

IMPLEMENTANDO LA PÁGINA DE LOGIN

La página LoginView será la página inicial cuando ejecute la aplicación, para que primero un usuario introduzca sus credenciales y se validen antes de poder capturar encuestas. Si bien la funcionalidad de autenticación la implementaremos más adelante en este libro, es un excelente momento de incluir la página para estructurar la aplicación y su navegación.

Usando la plantilla "Forms Xaml Page", agregaremos una nueva página llamada "LoginView" al folder "Views" del proyecto PCL. Además, agregaremos la clase "LoginViewModel" al folder "ViewModels" en el mismo proyecto.

En LoginView agregaremos los siguientes controles dentro de un StackLayout:

Label	Label con el texto "Nombre de usuario"
Entry	Entry cuyo texto está enlazado bidireccionalmente a la propiedad Username del ViewModel

Label	Label con el texto “Contraseña”
Entry	Entry de tipo Password cuyo texto está enlazado bidireccionalmente a la propiedad Password del ViewModel
Button	Button con el texto “Login” enlazado al comando LoginCommand del ViewModel

A continuación se muestra el código XAML completo de la página LoginView:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage
xmlns="http://xamarin.com/schemas/2014/forms"

xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="Surveys.Core.Views.LoginView">
  <StackLayout VerticalOptions="Center" Margin="10">
    <Label Text="Nombre de usuario" />
    <Entry Text="{Binding Username, Mode=TwoWay}" />
    <Label Text="Contraseña" />
    <Entry Text="{Binding Password, Mode=TwoWay}"
IsPassword="True" />
    <Button Text="Login" Command="{Binding
LoginCommand}" />
  </StackLayout>
</ContentPage>
```

Implementando LoginViewModel

La clase LoginViewModel debe exponer las propiedades Username y Password de tipo string, además del comando LoginCommand. La ejecución del comando LoginCommand estará en función de si el usuario ha introducido ambos valores. Esto lo podemos lograr nuevamente gracias al método ObservesProperty() de la clase DelegateCommand.

La acción relacionada con el comando LoginCommand deberá navegar a MainView, y mostrar de manera inmediata la página SurveysView. Aprovechando la característica de Deep Linking que ofrece Prism, la navegación sería: MainView/RootNavigationView/SurveysView. Con este Uri, estamos estableciendo

MainView como página raíz, indicando que su detalle es RootNavigationView, y a RootNavigationView le indicamos que navegue a SurveysView.

El siguiente fragmento de código muestra la navegación a ese Uri:

```
await
navigationService.NavigateAsync($"{nameof(MainView)}/{nam
eof(RootNavigationView)}/{nameof(SurveysView)}");
```

El siguiente código muestra la implementación completa de la clase LoginViewCommand:

```
using System.Windows.Input;
using Prism.Commands;
using Prism.Navigation;
using Surveys.Core.Views;
namespace Surveys.Core.ViewModels
{
    public class LoginViewModel : ViewModelBase
    {
        private INavigationService navigationService =
null;

        private string username;

        public string Username
        {
            get
            {
                return username;
            }
            set
            {
                if (username == value)
                {
                    return;
                }
```

```
                username = value;
                OnPropertyChanged();
            }
        }

        private string password;

        public string Password
        {
            get
            {
                return password;
            }
            set
            {
                if (password == value)
                {
                    return;
                }
                password = value;
                OnPropertyChanged();
            }
        }

        public ICommand LoginCommand { get; set; }

        public LoginViewModel(INavigationService
navigationService)
        {
            this.navigationService = navigationService;
            LoginCommand =
```

```
                    new
DelegateCommand(LoginCommandExecute,
LoginCommandCanExecute).ObservesProperty(() => Username)
                        .ObservesProperty(() => Password);
            }

            private async void LoginCommandExecute()
            {
                await navigationService.NavigateAsync(

$"{nameof(MainView)}/{nameof(RootNavigationView)}/{nameof
(SurveysView)}");
            }

            private bool LoginCommandCanExecute()
            {
                return !string.IsNullOrWhiteSpace(Username)
&& !string.IsNullOrWhiteSpace(Password);
            }
        }
    }
```

MODIFICANDO LA CLASE DE APLICACIÓN

Para concluir con esta implementación, modificaremos el método RegisterTypes() de la clase App para incluir las nuevas páginas que hemos creado. Asimismo, modificaremos el método OnInitialized() para navegar a LoginView cuando la aplicación ejecute por primera vez.

El siguiente código muestra implementación completa de la clase App:

```
using Prism.Unity;
using Surveys.Core.Views;
namespace Surveys.Core
{
    public partial class App : PrismApplication
    {
```



```
            protected override async void OnInitialized()
            {
                InitializeComponent();

                await
NavigationService.NavigateAsync($"{nameof(LoginView)}");
            }

            protected override void RegisterTypes()
            {

Container.RegisterTypeForNavigation<RootNavigationView>()
;

Container.RegisterTypeForNavigation<SurveysView>();

Container.RegisterTypeForNavigation<SurveyDetailsView>();

Container.RegisterTypeForNavigation<LoginView>();

Container.RegisterTypeForNavigation<MainView>();

Container.RegisterTypeForNavigation<AboutView>();
            }
        }
    }
```

Implementando Material Design en la aplicación de Android

Para que el botón del menú en MainView se vea correctamente, y en pro de que la aplicación se vea de mejor manera en versiones anteriores de Android, modificaremos la herencia de la clase MainActivity, para que herede de FormsAppCompatActivity en vez de simplemente de FormsApplicationActivity.

La clase FormsAppCompatActivity incluye la propiedad ToolbarResource, la cual podemos usar para establecer el estilo de la barra de herramientas. Por esta razón, además del cambio de herencia, agregaremos un nuevo archivo llamado Toolbar.axml en el folder resources\layout del proyecto Android. Este archivo incluirá el código AXML para estilizarla:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v7.widget.Toolbar
xmlns:android="http://schemas.android.com/apk/res/android
"
      android:id="@+id/toolbar"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:background="?attr/colorPrimary"

android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionB
ar"

android:popupTheme="@style/ThemeOverlay.AppCompat.Light"
/>
```

El siguiente código muestra la implementación de la clase MainActivity una vez hechas las modificaciones anteriores:

```
using Android.App;
using Android.OS;
using Xamarin.Forms.Platform.Android;
namespace Surveys.Droid
{
     [Activity(Label = "Surveys.Droid", Theme =
"@style/MainTheme", MainLauncher = true, Icon =
"@drawable/icon")]
      public class MainActivity : FormsAppCompatActivity
      {
          protected override void OnCreate(Bundle bundle)
          {
              ToolbarResource = Resource.Layout.Toolbar;

              base.OnCreate(bundle);

              Xamarin.Forms.Forms.Init(this, bundle);
              LoadApplication(new Core.App());
```



```
            }
        }
    }
```

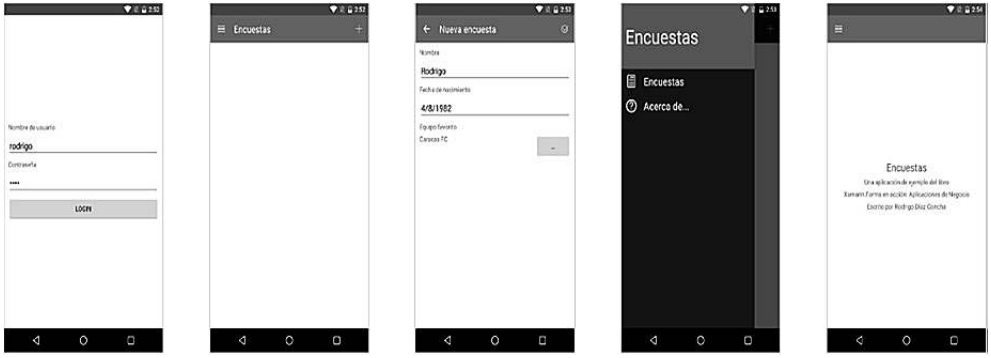
Probando la aplicación con la nueva estructura

La incorporación del esquema Maestro-Detalle en la aplicación ha mejorado la Interfaz de Usuario y las posibilidades de extenderla.

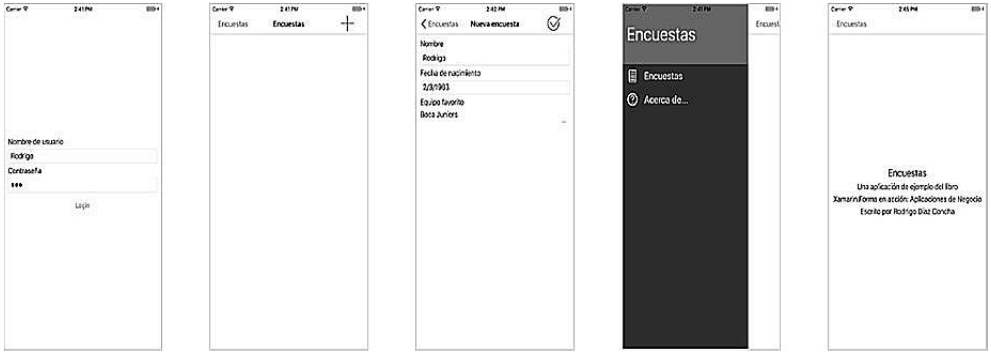
Recuerda que ahora la página inicial al ejecutar por primera vez la aplicación será LoginView. Para poder continuar, simplemente introduce algo en ambas cajas de texto y pulsa el botón “Login”.

Las siguientes figuras muestran las diferentes páginas que ahora tiene la aplicación. Observa el menú desplegable que muestra la lista de módulos disponibles:

ANDROID



IOS



10 ALMACENAMIENTO LOCAL CON SQLITE

INTRODUCCIÓN

Generalmente, en las aplicaciones de negocio requerimos algún tipo de almacenamiento local que nos permita guardar datos y que estos persistan incluso cuando el dispositivo haya sido apagado. En este módulo presentaremos SQLite como motor de base de datos local para nuestras aplicaciones móviles multiplataforma con Xamarin.Forms.

¿Qué es SQLite?

SQLite es un motor de bases de datos para dispositivos móviles. Es transaccional y autocontenida, la cual la hace una excelente opción para las aplicaciones multiplataforma construidas con una plataforma como Xamarin.

Una de las características de SQLite es que no requiere configuración: una simple declaración de la ruta física donde queremos guardar la base de datos será suficiente. Otra característica interesante es que podemos consultar la base de datos a través de expresiones de LINQ, además de poder usar cláusulas de tipo T-SQL.

Ambos Android y iOS ya incluyen SQLite como parte del sistema operativo, por lo que no necesitamos agregar alguna referencia adicional. Por otra parte, en el caso de las aplicaciones de tipo UWP, necesitamos descargar e instalar de manera independiente SQLite y agregar manualmente la referencia en el proyecto.

Instalando y referenciando SQLite en el proyecto UWP

La última versión de SQLite la puedes descargar de la página http://sqlite.org/download.html. En esta página podrás encontrar todas las

plataformas que soporta SQLite. En nuestro caso, descargaremos la extensión instalable para Visual Studio .NET:

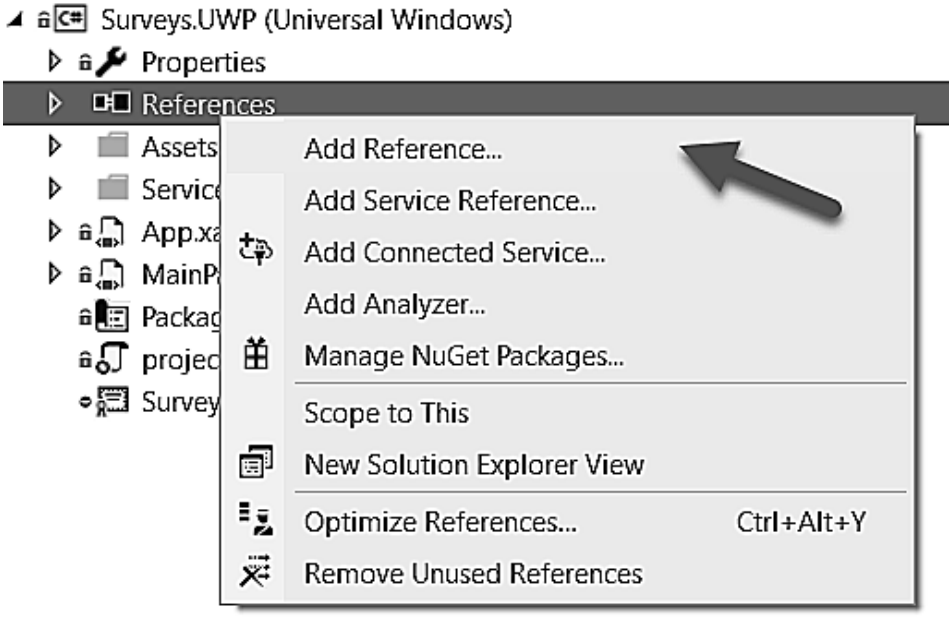
Universal Windows Platform

sqlite-uwp-3170000.vsix (6.24 MiB) VSIX package for Universal Windows Platform development using Visual Studio 2015. (sha1: 340ef8a4ccff5c0a69b375993d580d70377fa253)

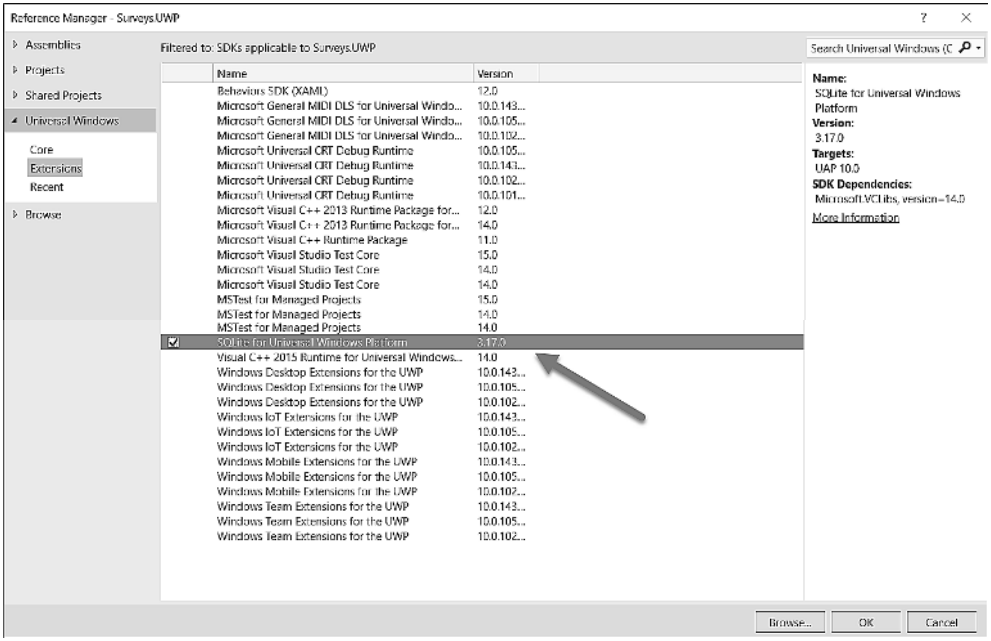
Nota: En el momento de estar escribiendo estas líneas, la versión más reciente de la extensión de SQLite para UWP es la 3.17.0.

Una vez descargado el archivo .vsix, simplemente lo ejecutamos para permitir que se instale como una extensión en Visual Studio .NET.

Para agregar SQLite en el proyecto UWP debemos seleccionar la opción de "Agregar referencia", tal y como se muestra en la siguiente figura:



En la ventana para administrar las referencias, seleccionamos en la parte izquierda la sección "Extensions" de la opción "Universal Windows". En la lista de SDKs seleccionamos "SQLite for Universal Windows Platform" y hacemos clic en el botón "OK" para agregar la referencia:



Instalando el paquete de SQLite para .NET

Además del motor de SQLite, independientemente si ya lo incluye el sistema operativo o se tiene que instalar por separado, necesitamos algún tipo de biblioteca que nos permita comunicarnos al motor de SQLite a través de .NET y el lenguaje C#. Hay varias bibliotecas que podrás encontrar en NuGet que tengan esta funcionalidad, pero la biblioteca oficial con soporte a Bibliotecas de Clases Portables es "SQLite-net PCL" (sqlite-net-pcl). La versión más reciente al estar escribiendo este capítulo es la 1.3.1, sin embargo, utilizaré la versión 1.2.1 ya que no requiere más dependencias adicionales como sí ocurre con la más reciente. Puedes instalar la última versión de este paquete de NuGet usando la consola de administración de paquetes en Visual Studio .NET escribiendo el siguiente comando:

```
Install-Package sqlite-net-pcl
```

Otra forma más fácil es utilizar la ventana de paquetes de NuGet de la solución, buscar el paquete en la caja de búsqueda e instalarlo en todos los proyectos, seleccionándolos en la lista de la parte derecha en la pantalla. Al hacer clic en el botón "Install", Visual Studio .NET descargará e instalará el paquete y sus dependencias en todos los proyectos que hayamos seleccionado.

Nota: Previo a instalar sqlite-net-pcl en UWP, deberás actualizar el paquete Microsoft.NETCore.UniversalWindowsPlatform a la versión 5.2.2.

Creando la conexión a SQLite

Para usar SQLite en una solución con Xamarin.Forms, cada plataforma concreta deberá devolver un objeto de tipo SQLiteConnection, el cual representa una conexión hacia la base de datos local. Una base de datos de SQLite es simplemente un archivo físico almacenado en el sistema de archivos de cada plataforma. Por lo tanto, para utilizar SQLite debemos definir una interfaz en el proyecto PCL y usar el Servicio de Dependencias de Xamarin.Forms para obtener el objeto concreto de dicha interfaz. La clase concreta podrá hacer uso de todas clases de .NET disponibles en la plataforma en cuestión. Para mayor información acerca del Servicio de Dependencias, consulta el capítulo 8 “Funcionalidad nativa de las plataformas”.

La clase SQLiteConnection tiene diversos miembros, entre los que destacan los siguientes:

DatabasePath	Propiedad que devuelve la ruta física de la base de datos en la plataforma concreta donde se está ejecutando el código
TableMappings	Propiedad que devuelve el mapeo de tablas existentes en la base de datos que la conexión está regresando
CreateTable<T>()	Método para crear una tabla en la base de datos
DropTable<T>()	Método para eliminar una tabla de la base de datos
Table<T>()	Devuelve un objeto en el cual podemos ejecutar consultas en la tabla especificada en el tipo T
Insert()	Inserta un objeto en su tabla relacionada
InsertAll()	Inserta una colección de objetos en su tabla relacionada
Update()	Actualiza un objeto, basándose en su Clave Primaria (PK)
Delete()	Borra un objeto, basándose en su Clave Primaria (PK)
CreateCommand()	Crea un comando al que le podemos especificar una sentencia preparada con el lenguaje T-SQL

El siguiente fragmento de código muestra una interfaz llamada ISqliteService, la cual incluye un método llamado GetConnection() cuyo objetivo es devolver una conexión hacia la base de datos local:

```
public interface ISqliteService
{
    SQLiteConnection GetConnection();
}
```


El siguiente código muestra la implementación concreta de la anterior interfaz en Android. Cabe destacar el uso de las clases System.Environment y System.IO.Path para cumplir con la funcionalidad buscada:

```
public class SqliteService : ISqliteService
{
    public SQLiteConnection GetConnection()
    {
        var localDbFile =

System.IO.Path.Combine(System.Environment.GetFolderPath(S
ystem.Environment.SpecialFolder.Personal),
                "surveys.db");
        return new SQLiteConnection(localDbFile);
    }
}
```

Una vez implementadas las clases concretas, podríamos obtener la conexión a SQLite en la PCL a través del siguiente código de ejemplo:

```
var sqliteConnection =
Xamarin.Forms.DependencyService.Get<ISqliteService>().Get
Connection();
```

Creando un servicio para la base de datos

Obtener la conexión a SQLite es solo el primer paso, ya que necesitamos alguna clase que encapsule todas las operaciones hacia la base de datos local. Hay muchas maneras de lograr esto, por ejemplo podríamos crear una clase con el patrón de diseño Singleton que se asegure que solamente hubiera una instancia del objeto durante todo el ciclo de vida de la aplicación, y en esa clase exponer todos los miembros para las operaciones CRUD. Otra manera es crear una interfaz que incluya todos los miembros necesarios para hacer las operaciones CRUD en la base de datos, y registrar esa interfaz y un objeto que la implemente de manera concreta en el contenedor de Inyección de Dependencias de Unity. Esto lo podemos lograr a través del método ConfigureContainer() de la clase PrismApplication, el cual explicamos en el capítulo anterior. De esta forma, podríamos simplemente solicitar la interfaz en el constructor de los ViewModels donde la vayamos a usar, y Unity se encargaría de inyectar el objeto concreto.

Por ejemplo, el siguiente código muestra la definición de una interfaz llamada ILocalDbService con un método para insertar una encuesta en la base de datos:

```
public interface ILocalDbService
{
    Task InsertSurveyAsync(Survey survey);
}
```

El siguiente código muestra el registro de una instancia de un objeto que implementa la interfaz ILocalDbService, en el contenedor de Unity:

```
protected override void ConfigureContainer()
{
    base.ConfigureContainer();

    Container.RegisterInstance<ILocalDbService>(new
LocalDbService());
}
```

MANOS A LA OBRA

Como pudimos observar en el último capítulo, después de capturar una nueva encuesta, si navegamos al módulo de Acerca de… y nuevamente de regreso al módulo de Encuestas, la encuesta recién capturada desaparece. Esto se debe a que nuestra aplicación de encuestas no guarda la encuesta capturada, y de hecho tampoco contamos con una base de datos local donde podamos almacenar los datos capturados. En este capítulo modificaremos la aplicación actual para incluir SQLite como motor de base de datos local multiplataforma.

Implementando ISQLiteService

En el folder "ServiceInterfaces" del proyecto PCL, agregaremos una nueva interfaz llamada ISqliteService. Esta interfaz tendrá como único miembro el método GetConnection(), el cual nos devolverá un objeto de tipo SQLiteConnection.

```
using SQLite;
namespace Surveys.Core.ServiceInterfaces
{
    public interface ISqliteService
    {
```



```
        SQLiteConnection GetConnection();
    }
}
```

Una vez creada la interfaz, la implementaremos en una clase concreta en cada plataforma. Adicionalmente, la clase concreta la expondremos por medio del atributo Xamarin.Forms.Dependency para que el servicio de dependencias de Xamarin.Forms sea capaz de resolver dicho objeto cuando se requiera.

IMPLEMENTACIÓN DE LA CLASE SQLITESERVICE EN ANDROID

En el folder "Services" del proyecto Android agregaremos una nueva clase llamada SqliteService. En esta clase implementaremos la interfaz ISqliteService para devolver el objeto de tipo SQLiteConnection, tal y como lo muestra el siguiente código:

```
using SQLite;
using Surveys.Core.ServiceInterfaces;
using Surveys.Droid.Services;
[assembly:Xamarin.Forms.Dependency(typeof(SqliteService
))]
namespace Surveys.Droid.Services
{
    public class SqliteService : ISqliteService
    {
        public SQLiteConnection GetConnection()
        {
            var localDbFile =

System.IO.Path.Combine(System.Environment.GetFolderPath(S
ystem.Environment.SpecialFolder.Personal),
                    "surveys.db");
            return new SQLiteConnection(localDbFile);
        }
    }
}
```

IMPLEMENTACIÓN DE LA CLASE SQLITESERVICE EN IOS

En el folder "Services" del proyecto iOS, agregaremos una nueva clase llamada SqliteService. En esta clase implementaremos la interfaz ISqliteService para devolver el objeto de tipo SQLiteConnection. El siguiente código muestra la implementación completa de la clase SqliteService de iOS:

```
using SQLite;
using Surveys.Core.ServiceInterfaces;
using Surveys.iOS.Services;
[assembly:Xamarin.Forms.Dependency(typeof(SqliteService
))]
namespace Surveys.iOS.Services
{
    public class SqliteService : ISqliteService
    {
        public SQLiteConnection GetConnection()
        {
            var localDbFile =

System.IO.Path.Combine(System.Environment.GetFolderPath(S
ystem.Environment.SpecialFolder.Personal),
                    "surveys.db");
            return new SQLiteConnection(localDbFile);
        }
    }
}
```

IMPLEMENTACIÓN DE LA CLASE SQLITESERVICE EN UWP

En el folder "Services" del proyecto UWP, agregaremos una nueva clase llamada SqliteService. En esta clase implementaremos la interfaz ISqliteService para devolver el objeto de tipo SQLiteConnection. Observa que a diferencia de las implementaciones en Android y iOS, aquí utilizamos la clase ApplicationData del espacio de nombres Windows.Storage para establecer la ruta donde se creará la base de datos.

El siguiente código muestra la implementación completa de la clase SqliteService de UWP:

```
using Windows.Storage;
using SQLite;
using Surveys.Core.ServiceInterfaces;
using Surveys.UWP.Services;
[assembly:Xamarin.Forms.Dependency(typeof(SqliteService
))]
namespace Surveys.UWP.Services
{
    public class SqliteService : ISqliteService
    {
        public SQLiteConnection GetConnection()
        {
            var conn =
                new
SQLiteConnection(System.IO.Path.Combine(ApplicationData.C
urrent.LocalFolder.Path, "surveys.db"));

            return conn;
        }
    }
}
```

Una vez listas las implementaciones concretas de ISqliteService, crearemos la interfaz que encapsule las operaciones hacia la base de datos. Pero antes, haremos una ligera modificación a nuestra clase de modelo Survey.

Modificando la clase Survey

La clase Survey es nuestro modelo en el Patrón de Diseño MVVM. Agregaremos una nueva propiedad de tipo string llamada Id, la cual nos permitirá asignar un identificador único a cada encuesta. Este identificador único será de gran ayuda en el momento de borrar un registro en la base de datos y al sincronizar.

El siguiente código muestra la implementación completa de la clase Survey. Observa que también he modificado el método ToString() para incluir el valor de la propiedad Id.

```
using System;
namespace Surveys.Core
{
    public class Survey
    {
        public string Id { get; set; }
        public string Name { get; set; }

        public DateTime Birthdate { get; set; }
        public string FavoriteTeam { get; set; }
        public double Lat { get; set; }
        public double Lon { get; set; }
        public override string ToString()
        {
            return $"{Id} {Name} |  {Birthdate} |
{FavoriteTeam} | {Lat} | {Lon}";
        }
    }
}
```

Creación de la interfaz ILocalDbService

La interfaz ILocalDbService encapsulará todos los miembros necesarios para realizar operaciones CRUD en la base de datos. El objeto concreto que implemente esta interfaz será inyectado por Unity en aquellos constructores de los ViewModels que la soliciten como argumento.

Los miembros que tendrá inicialmente la interfaz ILocalDbService se describen a continuación:

GetAllSurveysAsync()	Método que ddevuelve todas las encuestas de la base de datos
InsertSurveyAsync()	Método para insertar un nuevo registro en la tabla Survey
DeleteSurveyAsync()	Método para borrar un registro de la tabla Survey

Una vez identificados los miembros que debe tener, agregaremos la interfaz en el folder "ServiceInterfaces" del proyecto PCL. El siguiente código muestra la implementación completa de la interfaz ILocalDbService:

```
using System.Collections.Generic;
using System.Threading.Tasks;
namespace Surveys.Core.ServiceInterfaces
{
    public interface ILocalDbService
    {
        Task<IEnumerable<Survey>> GetAllSurveysAsync();

        Task InsertSurveyAsync(Survey survey);

        Task DeleteSurveyAsync(Survey survey);
    }
}
```

Una vez implementada la interfaz, crearemos un nuevo folder en el proyecto PCL llamado "Services". En este folder, crearemos una nueva clase llamada LocalDbService la cual implementará de manera concreta la interfaz ILocalDbService. En el constructor de la clase, obtendremos la conexión que resuelve el Servicio de Dependencias de Xamarin.Forms y la guardaremos en un campo privado a nivel de clase, ya que utilizaremos el objeto de conexión en todos los miembros.

La base de datos para nuestra aplicación de encuestas inicialmente tendrá una sola tabla llamada "Survey", la cual tendrá como columnas las mismas propiedades que tiene la clase de modelo Survey. Adicionalmente, nos aseguraremos de que el esquema de la base de datos haya sido creado previo a la utilización de los métodos. Por tal motivo, crearemos un método privado llamado CreateDatabase() el cual usará el método CreateTable() de la clase SQLiteConnection, y cuyo tipo será la clase Survey que tenemos en el folder Models en la PCL. Esto permitirá a SQLite basarse en el modelo e inferir todas las columnas para la tabla Survey.

El siguiente código muestra la implementación completa de la clase LocalDbService:

```
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using SQLite;
using Surveys.Core.ServiceInterfaces;
using Xamarin.Forms;
namespace Surveys.Core.Services
{
    public class LocalDbService : ILocalDbService
    {
        private readonly SQLiteConnection connection =
null;

        public LocalDbService()
        {
            connection =
DependencyService.Get<ISqliteService>().GetConnection();

            CreateDatabase();
        }
        private void CreateDatabase()
        {
            if (connection.TableMappings.All(t =>
t.TableName != nameof(Survey)))
            {
                connection.CreateTable<Survey>();
            }
        }

        public Task<IEnumerable<Survey>>
GetAllSurveysAsync()
```

```
            {
                return Task.Run(() =>
(IEnumerable<Survey>)connection.Table<Survey>().ToArray()
);
            }

            public Task InsertSurveyAsync(Survey survey)
            {
                return Task.Run(() =>
connection.Insert(survey));
            }

            public Task DeleteSurveyAsync(Survey survey)
            {
                return Task.Run(() =>
                {
                    var query = $"DELETE FROM Survey WHERE
Id = '{survey.Id}'";

                    var command =
connection.CreateCommand(query);
                    var result = command.ExecuteNonQuery();
                    return result > 0;
                });
            }
        }
    }
```

Nota: En el método DeleteSurveyAsync() hemos optado por usar directamente una consulta de tipo T-SQL a través del método CreateCommand(). Esto obedece a que no hemos modificado de ninguna manera la clase de modelo Survey para decorar sus propiedades con los atributos que expone SQLite, por ejemplo para identificar la Clave Primaria. Esto es completamente intencional y con el objetivo de seguir manteniendo limpia la declaración del modelo, ya que más adelante la vamos a reutilizar cuando hablemos de la comunicación a servicios REST en el siguiente capítulo.

Registrando la instancia en el contenedor de Unity

Para poder usar la interfaz ILocalDbService, registraremos una instancia de LocalDbService en el contenedor de Inyección de Dependencias de Unity. Esto lo lograremos reemplazando la implementación del método ConfigureContainer() de la clase App en la PCL, tal y como lo muestra el siguiente fragmento de código:

```
…
protected override void ConfigureContainer()
{
    base.ConfigureContainer();


    Container.RegisterInstance<ILocalDbService>(new
LocalDbService());
}
```

Gracias al código anterior, cada vez que un ViewModel solicite en su constructor un objeto de tipo ILocalDbService, Unity se encargará de inyectar la instancia única de LocalDbService que hemos declarado. En otras palabras, LocalDbService ha quedado como un objeto de tipo Singleton ya que solo habrá una instancia de él durante todo el ciclo de vida de la aplicación.

Inyectando el servicio en los ViewModels

Modificaremos la clase SurveyDetailsViewModel para solicitar como argumento un objeto de tipo ILocalDbService. El objeto lo guardaremos en un campo privado a nivel de la clase de ViewModel para poder utilizarlo en sus diferentes miembros.

El siguiente fragmento de código destaca las modificaciones que tendrá la clase SurveyDetailsViewModel:

```
public class SurveyDetailsViewModel : ViewModelBase
{
    private INavigationService navigationService =
null;
    private IPageDialogService pageDialogService =
null;
    private ILocalDbService localDbService = null;

…
```

```
            public
SurveyDetailsViewModel(INavigationService
navigationService, IPageDialogService pageDialogService,
            ILocalDbService localDbService)
        {
            this.navigationService = navigationService;
            this.pageDialogService = pageDialogService;
            this.localDbService = localDbService;

            …
        }
    …
    }
```

El siguiente fragmento de código destaca las modificaciones que haremos en la clase SurveysViewModel:

```
    public class SurveysViewModel : ViewModelBase
    {
        private INavigationService navigationService =
null;
        private ILocalDbService localDbService = null;
    …


        public SurveysViewModel(INavigationService
navigationService, ILocalDbService localDbService = null)
        {
            this.navigationService = navigationService;
            this.localDbService = localDbService;

            …
        }
    …
    }
```

Como podrás observar, en ambos casos la intención es la misma: inyectar el objeto que implementa la interfaz ILocalDbService en el ViewModel, y guardar la referencia en un campo privado para poder reusarlo en cualquier parte de la clase.

Guardando las encuestas en la base de datos

Ya con el objeto ILocalDbService inyectado en SurveyDetailsViewModel, modificaremos la acción relacionada al comando EndSurveyCommand, para insertar en la base de datos la nueva encuesta capturada. Debido a que ya no será necesario pasar como parámetro el objeto de la nueva encuesta cuando estemos navegando de regreso a SurveysView, la invocación del método GoBackAsync() quedará sin parámetro. Adicionalmente, asignaremos como valor a la propiedad Id de la nueva encuesta capturada una cadena que la identifique de manera única. Para ello, usaremos el método NewGuid() de la clase Guid.

El siguiente fragmento de código muestra los cambios en el método EndSurveyCommandExecute():

```
  private async void EndSurveyCommandExecute()
  {
      var newSurvey = new Survey()
      {
          Id = Guid.NewGuid().ToString(),
          Name = Name,
          Birthdate = Birthdate,
          FavoriteTeam = FavoriteTeam
      };

      var geolocationService =
Xamarin.Forms.DependencyService.Get<IGeolocationService>(
);

      if (geolocationService != null)
      {
          try
          {
              var currentLocation = await
geolocationService.GetCurrentLocationAsync();
```



```
                newSurvey.Lat = currentLocation.Item1;
                newSurvey.Lon = currentLocation.Item2;
            }
            catch (Exception)
            {
                newSurvey.Lat = 0;
                newSurvey.Lon = 0;
            }
        }

        await localDbService.InsertSurveyAsync(newSurvey);

        await navigationService.GoBackAsync();
    }
```

Leyendo las encuestas de la base de datos

En la clase SurveysViewModel, modificaremos el método OnNavigatedTo para ejecutar el método GetAllSurveysAsync() del objeto ILocalDbService y el resultado lo usaremos como fuente para crear la colección Surveys. Además, eliminaremos la implementación anterior que obtenía el parámetro "NewSurvey", ya que no será necesaria.

El siguiente fragmento de código muestra las modificaciones en el método OnNavigatedTo de la clase SurveysViewModel:

```
    public override async void
OnNavigatedTo(NavigationParameters parameters)
    {
        base.OnNavigatedTo(parameters);

        var allSurveys = await
localDbService.GetAllSurveysAsync();
        if (allSurveys != null)
        {
            Surveys = new
ObservableCollection<Survey>(allSurveys);
        }
    }
```

Implementando la funcionalidad de borrar encuestas

Una funcionalidad que falta en este momento en nuestra aplicación es el poder borrar encuestas ya capturadas. Esto lo vamos a implementar agregando un nuevo comando llamado DeleteSurveyCommand en la clase SurveysViewModel, y además agregando un nuevo ToolbarItem en SurveysView, enlazado a este nuevo comando.

CREANDO EL TOOLBARITEM

En la colección ToolbarItems de la página, agregaremos un nuevo ToolbarItem tal y como lo muestra el siguiente fragmento de código:

```
<ToolbarItem Text="Borrar"
             Command="{Binding DeleteSurveyCommand}">
  <ToolbarItem.Icon>
    <OnPlatform x:TypeArguments="FileImageSource"
                Android="delete.png"
                iOS="delete.png"
                WinPhone="assets/delete.png" />
  </ToolbarItem.Icon>
</ToolbarItem>
```

Ten en cuenta que deberás agregar en cada proyecto de plataforma concreta un archivo de imagen que será usado como icono en el ToolbarItem. Para que el código anterior funcione, he agregado un archivo llamado “delete.png” en el folder respectivo de cada proyecto concreto, y además he marcado la acción de compilación requerida en cada plataforma.

Una vez modificada la página SurveysView, implementaremos el comando DeleteSurveyCommand en el ViewModel.

IMPLEMENTANDO EL COMANDO DELETESURVEYCOMMAND

En la clase SurveysViewModel, implementaremos una nueva propiedad de tipo ICommand llamada DeleteSurveyCommand. Esta propiedad la inicializaremos en el constructor de la clase, asignándole un objeto de tipo DelegateCommand, el cual estará observando la propiedad SelectedSurvey para determinar si se puede ejecutar o no.

El siguiente fragmento de código muestra la instanciación del objeto DelegateCommand que asignaremos a la propiedad DeleteSurveyCommand:

```
  DeleteSurveyCommand = new
DelegateCommand(DeleteSurveyCommandExecute,
DeleteSurveyCommandCanExecute).ObservesProperty(() =>
SelectedSurvey);
```

Los métodos DeleteSurveyCommandExecute() y DeleteSurveyCommandExecute() determinan la acción a ejecutar y la expresión para determinar si se puede ejecutar o no. En nuestro caso, debemos estar seguros de que efectivamente el usuario haya seleccionado una encuesta de la lista, es decir, que la propiedad SelectedSurvey no sea nula. Además, siempre es buena idea solicitar la confirmación del usuario para realizar esta operación, por lo que adicionalmente en el constructor solicitaremos como argumento la interfaz IPageDialogService, para poder usar su método DisplayAlertAsync() y pedir al usuario que confirme esta operación antes de borrar el registro. Si el usuario decide borrar el registro, lo borraremos haciendo uso del método DeleteSurveyAsync() de la interfaz ILocalDbService, y además cargaremos nuevamente la lista de encuestas. Ya que actualmente tenemos un código en el método OnNavigatedTo para cargar la lista de encuestas, y en pro de la reutilización de código, ese código lo moveremos a su propio método llamado LoadSurveysAsync().

Antes de implementar estos cambios, agregaremos en el archivo Literals.cs las cadenas que usaremos en DisplayAlertAsync(). El siguiente código muestra la implementación completa de la clase Literals después de haber incluido las constantes DeleteSurveyTitle, DeleteSurveyConfirmation y Cancel:

```
  namespace Surveys.Core
  {
      public class Literals
      {
          public const string FavoriteTeamTitle =
"Selección de equipo";
          public const string Ok = "Aceptar";
          public const string DeleteSurveyTitle =
"Borrar";
          public const string DeleteSurveyConfirmation =
"¿Está seguro(a)?";
          public const string Cancel = "Cancelar";
      }
  }
```

El siguiente código muestra la implementación completa de la clase SurveysViewModel después de haber implementado todas las modificaciones recién descritas:

```
using System.Collections.ObjectModel;
using System.Threading.Tasks;
using System.Windows.Input;
using Prism.Commands;
using Prism.Navigation;
using Prism.Services;
using Surveys.Core.ServiceInterfaces;
namespace Surveys.Core
{
    public class SurveysViewModel : ViewModelBase
    {
        private INavigationService navigationService =
null;
        private IPageDialogService pageDialogService =
null;
        private ILocalDbService localDbService = null;

        #region Properties

        private ObservableCollection<Survey> surveys;
        public ObservableCollection<Survey> Surveys
        {
            get
            {
                return surveys;
            }
            set
            {
                if (surveys == value)
```



```
                {
                    return;
                }
                surveys = value;
                OnPropertyChanged();
            }
        }
        private Survey selectedSurvey;

        public Survey SelectedSurvey
        {
            get
            {
                return selectedSurvey;
            }
            set
            {
                if (selectedSurvey == value)
                {
                    return;
                }
                selectedSurvey = value;
                OnPropertyChanged();
            }
        }

        #endregion

        public ICommand NewSurveyCommand { get; set; }

        public ICommand DeleteSurveyCommand { get; set;
}
```

```
        public SurveysViewModel(INavigationService
navigationService, IPageDialogService pageDialogService,
            ILocalDbService localDbService = null)
        {
            this.navigationService = navigationService;
            this.pageDialogService = pageDialogService;
            this.localDbService = localDbService;
            Surveys = new
ObservableCollection<Survey>();
            NewSurveyCommand = new
DelegateCommand(NewSurveyCommandExecute);
            DeleteSurveyCommand =
                new
DelegateCommand(DeleteSurveyCommandExecute,
DeleteSurveyCommandCanExecute).ObservesProperty(
                    () => SelectedSurvey);
        }

        private async void NewSurveyCommandExecute()
        {
            await
navigationService.NavigateAsync(nameof(SurveyDetailsView)
);
        }

        private async void DeleteSurveyCommandExecute()
        {
            if (SelectedSurvey == null)
            {
                return;
            }
            var result = await
pageDialogService.DisplayAlertAsync(Literals.DeleteSurvey
Title,
```



```
                    Literals.DeleteSurveyConfirmation,
Literals.Ok, Literals.Cancel);

                if (result)
                {
                    await
localDbService.DeleteSurveyAsync(SelectedSurvey);

                    await LoadSurveysAsync();
                }
            }

            private bool DeleteSurveyCommandCanExecute()
            {
                return SelectedSurvey != null;
            }

            public override async void
OnNavigatedTo(NavigationParameters parameters)
            {
                base.OnNavigatedTo(parameters);
                await LoadSurveysAsync();
            }
            private async Task LoadSurveysAsync()
            {
                var allSurveys = await
localDbService.GetAllSurveysAsync();
                if (allSurveys != null)
                {
                    Surveys = new
ObservableCollection<Survey>(allSurveys);
                }
            }
        }
    }
```

Agregando un texto amigable en la Interfaz de Usuario

En este momento, cuando la aplicación ejecuta y no hay encuestas capturadas, la lista se muestra completamente vacía. Sin embargo, para mejorar un poco esta Interfaz de Usuario mostraremos un texto que invite al usuario a capturar encuestas. Esto lo lograremos implementando una nueva expresión llamada IsEmpty en SurveysViewModel, y cuyo valor estará en función de la cantidad de elementos de la colección Surveys:

```
  public bool IsEmpty => Surveys == null ||
!Surveys.Any();
```

Además, modificaremos el método LoadSurveysAsync() para notificar manualmente que la propiedad IsEmpty debe ser reevaluada por la infraestructura de enlace de datos:

```
  private async Task LoadSurveysAsync()
  {
      var allSurveys = await
localDbService.GetAllSurveysAsync();
      if (allSurveys != null)
      {
          Surveys = new
ObservableCollection<Survey>(allSurveys);
      }
      OnPropertyChanged(nameof(IsEmpty));
  }
```

Finalmente, modificaremos la página SurveysView para incluir un Grid con el texto que deseamos mostrar. El Grid estará sobrepuesto al ListView y se mostrará únicamente cuando la propiedad IsEmpty sea verdadera. El siguiente fragmento de código muestra la implementación de este nuevo Grid. Pon mucha atención en la expresión de enlace de datos en su propiedad IsVisible:

```
  …
  <Grid Margin="10">
    <ListView ItemsSource="{Binding Surveys}"
              SelectedItem="{Binding SelectedSurvey,
Mode=TwoWay}"
```



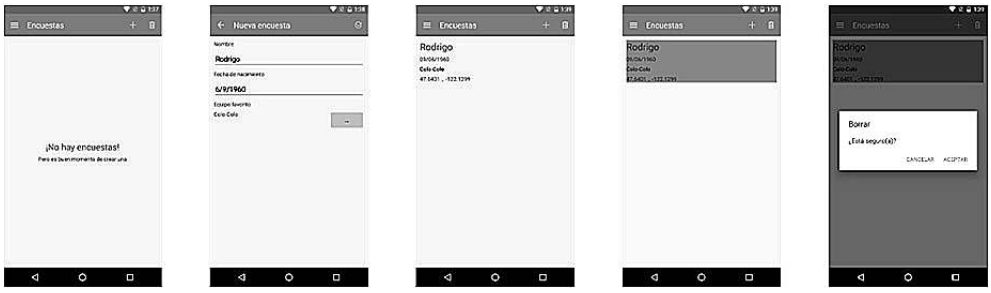
```
                ItemTemplate="{StaticResource
SurveyDataTemplate}"
                HasUnevenRows="True" />

    <Grid IsVisible="{Binding IsEmpty}">
      <StackLayout VerticalOptions="Center">
        <Label Text="¡No hay encuestas¡" FontSize="Large"
                HorizontalTextAlignment="Center" />
        <Label Text="Pero es buen momento de crear una"
                HorizontalTextAlignment="Center" />
      </StackLayout>
    </Grid>
  </Grid>
  …
```

Probando la aplicación

Si ejecutas la aplicación en este momento, podrás comprobar que efectivamente las encuestas son almacenadas en la base de datos local de SQLite. Esto lo puedes verificar si después de capturar una encuesta, navegas al módulo de "Acerca de…" o simplemente cierras la aplicación y vuelves a entrar. Adicionalmente, puedes corroborar que las encuestas son borradas de la base de datos si seleccionas una de ellas en la lista y pulsas el botón Borrar en la barra de herramientas.

Las siguientes figuras muestran la aplicación ejecutando en Android y iOS:

ANDROID



IOS

11 COMUNICACIÓN A SERVICIOS

INTRODUCCIÓN

Prácticamente, en la arquitectura de toda aplicación móvil de negocios se requiere algún tipo de infraestructura de servicios a los que se pueda comunicar ya sea para almacenamiento de datos, la ejecución de alguna lógica, validación de reglas, autenticación de los usuarios y un sinfín de posibilidades.

En este capítulo describiremos algunos de los mecanismos que podemos utilizar para implementar estos servicios y nos concentraremos principalmente en la arquitectura con servicios REST.

SOAP vs. REST

Cualquiera que fuesen tus opciones tecnológicas hoy en día para construir servicios, de alguna u otra manera, todas caerían en alguna de estas dos opciones: servicios SOAP o servicios REST.

SOAP es un protocolo basado en XML que pueden utilizar las aplicaciones distribuidas para intercambiar información. Los servicios SOAP son independientes a la plataforma que los consume y también son independientes al lenguaje que se haya seleccionado para construirlos. Estos servicios son muy generales y tienen su origen en la arquitectura RPC (Remote Procedure Call). En el ecosistema de Microsoft, los servicios SOAP se pueden construir usando la tecnología ASMX o también por medio de Windows Communication Foundation (WCF). Comúnmente, los servicios SOAP se consumen a través de un documento llamado WSDL, el cual describe los miembros que expone el servicio.

Por otro lado, REST (por sus siglas en inglés Representational State Transfer) es un estilo arquitectónico de servicios basado en el protocolo HTTP y en los verbos que

dicho protocolo expone. En REST, cada verbo está relacionado a una operación que un cliente puede ejecutar sobre el servicio. Debido a la ligereza y sencillez de este tipo de servicios, actualmente son el mecanismo adecuado y sugerido para la mayoría de aplicaciones móviles que requieran hacer uso de servicios.

En la siguiente tabla se enlistan algunos verbos HTTP y se describen las operaciones relacionadas con cada uno de ellos:

GET	Verbo utilizado para obtener datos del servicio
POST	Verbo utilizado para insertar o agregar datos en el servicio
PUT	Verbo utilizado para actualizar datos del servicio
DELETE	Verbo utilizado para borrar datos del servicio

Como todo en la arquitectura de software, elegir una u otra opción dependerá de las necesidades específicas del proyecto en cuestión, ya que ambas tienen pros y contras. No obstante, el resto del capítulo nos concentraremos en los servicios REST por tratarse de una arquitectura bastante común y predominantemente ubicua, principalmente en el espacio de sistemas móviles.

Creando un servicio REST con ASP.NET Web API

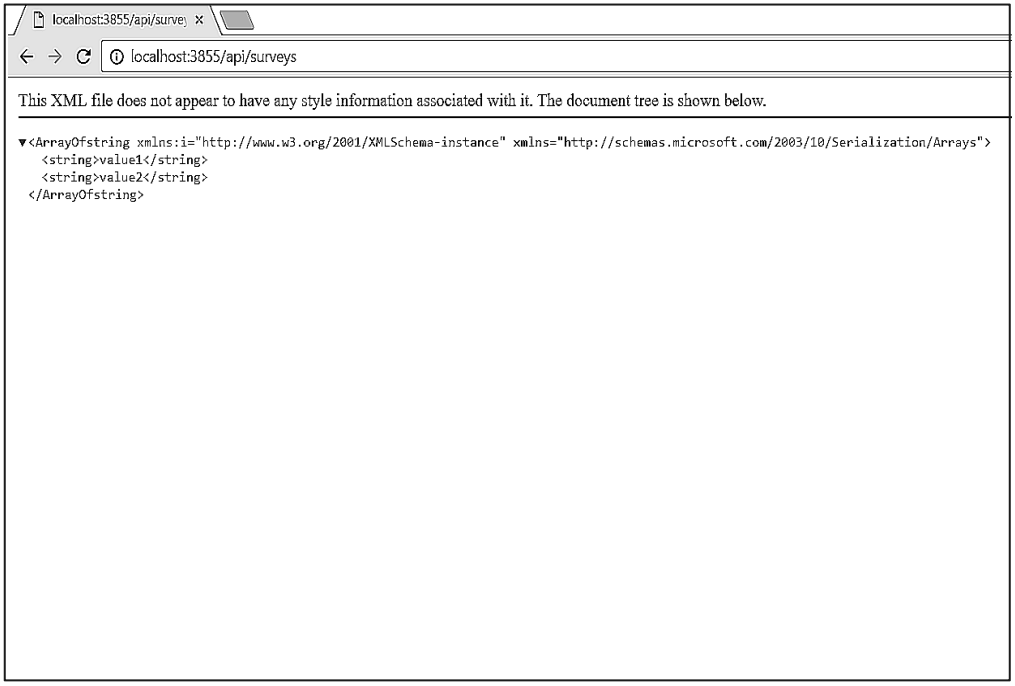
Para construir servicios REST podemos usar un sinfín de tecnologías, herramientas y lenguajes. Muy probablemente, la decisión de qué usar estará supeditada a la experiencia previa que tengas. Así que, debido a que hemos estado utilizando Visual Studio .NET a lo largo de todo este libro (y muy probablemente lo tengas incluso abierto en este momento al estar leyendo estas líneas), la plataforma de desarrollo con la que vamos a construir los servicios REST en este capítulo será ASP.NET Web API. ASP.NET Web API es un framework que nos permite construir fácilmente servicios HTTP para dispositivos móviles, aplicaciones Web, aplicaciones de escritorio, soluciones IoT, y un largo etcétera.

CONTROLADORES

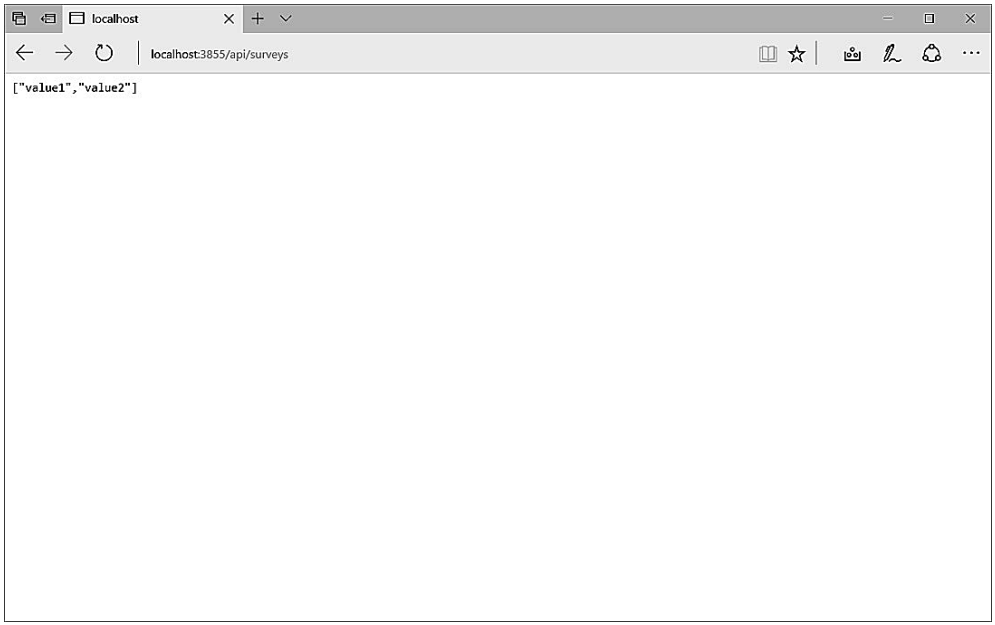
En ASP.NET Web API, un controlador es un objeto que maneja las peticiones del protocolo HTTP. ASP.NET Web API promueve el mecanismo de convención de nombres para ejecutar los controladores a través de HTTP. Esto quiere decir que un controlador llamado SurveysController es identificado a través del URI /api/surveys y otro llamado TeamsController tendrá un URI /api/teams (siempre y cuando /api sea la raíz en donde estén implementados los controladores).

Si ejecutas un proyecto de tipo ASP.NET Web API en Visual Studio .NET, podrás ver que la ejecución abrirá automáticamente el navegador que tengas configurado de forma predeterminada. Si en la barra de direcciones modificas el URI para solicitar una petición hacia un controlador, podrás observar que efectivamente el controlador está ejecutando y enviando una respuesta XML o JSON, dependiendo del navegador que estés utilizando.

La siguiente figura muestra el navegador Google Chrome después de haber creado una solicitud hacia el URI /api/surveys en el servidor Web local que ejecuta Visual Studio .NET cuando depuras una aplicación Web. Como podrás apreciar, en este escenario Chrome obtiene la respuesta en formato XML.



La siguiente figura muestra el navegador Microsoft Edge después de solicitar la misma dirección. En este otro caso, Edge recibe la respuesta en formato JSON.



MANOS A LA OBRA

En este capítulo construiremos el servicio REST para nuestra aplicación de encuestas utilizando ASP.NET Web API. Adicionalmente, implementaremos un mecanismo de autenticación basada en tokens por medio de OAuth y también modificaremos la aplicación para que sincronice las encuestas capturadas a una base de datos centralizada por medio del servicio.

Creando la base de datos

El servicio REST almacenará y consultará los datos en una base de datos relacional. En este capítulo, utilizaremos SQL Server como motor de base de datos relacional, pero fácilmente y con pocas modificaciones, podrás usar cualquier tipo de base de datos incluso aquellas de tipo NoSQL.

La base de datos tendrá dos tablas: Teams y Surveys. La tabla Teams tendrá los datos de los equipos que un usuario puede seleccionar durante una encuesta. La tabla Surveys almacenará los datos de las encuestas capturadas por los usuarios.

TABLA TEAMS

La siguiente tabla muestra el esquema que tendrá la tabla Teams:

Id	int	Identificador único para cada equipo. Esta es la Clave Primaria (PK) de la tabla
Name	nvarchar(50)	El nombre del equipo
Color	nvarchar(50)	El color del equipo
Logo	varbinary(MAX)	El logotipo del equipo en formato .png

Inicialmente, los equipos que agregaremos a esta tabla serán los mismos que están implementados de manera fija en la clase SurveyDetailsViewModel. La siguiente figura muestra la tabla Teams una vez precargados los datos:

	Id	Name	Color	Logo
1	1	Alianza Lima	#0000FF	0x89504E470D0A1A0A0000000D49484452000000BD000000...
2	2	América	#FFFF35	0x89504E470D0A1A0A0000000D4948445200000238000023...
3	3	Boca Juniors	#0000FF	0x89504E470D0A1A0A0000000D4948445200000190000001...
4	4	Caracas FC	#7C0029	0x89504E470D0A1A0A0000000D494844520000011A0000016...
5	5	Colo-Colo	#0000FF	0x89504E470D0A1A0A0000000D4948445200000121000011...
6	6	Peñarol	#FFFF35	0x89504E470D0A1A0A0000000D4948445200000151000001C...
7	7	Real Madrid	#E612E3	0x89504E470D0A1A0A0000000D49484452000001AC000002...
8	8	Saprissa	#7C0029	0x89504E470D0A1A0A0000000D49484452000001E3000001...

Cabe mencionar que también hemos incluido la columna “Logo”, la cual incluye la imagen del logotipo del equipo en cuestión. Esto es importante destacarlo, ya que más adelante modificaremos la aplicación para mostrar el logo de cada equipo en pantalla.

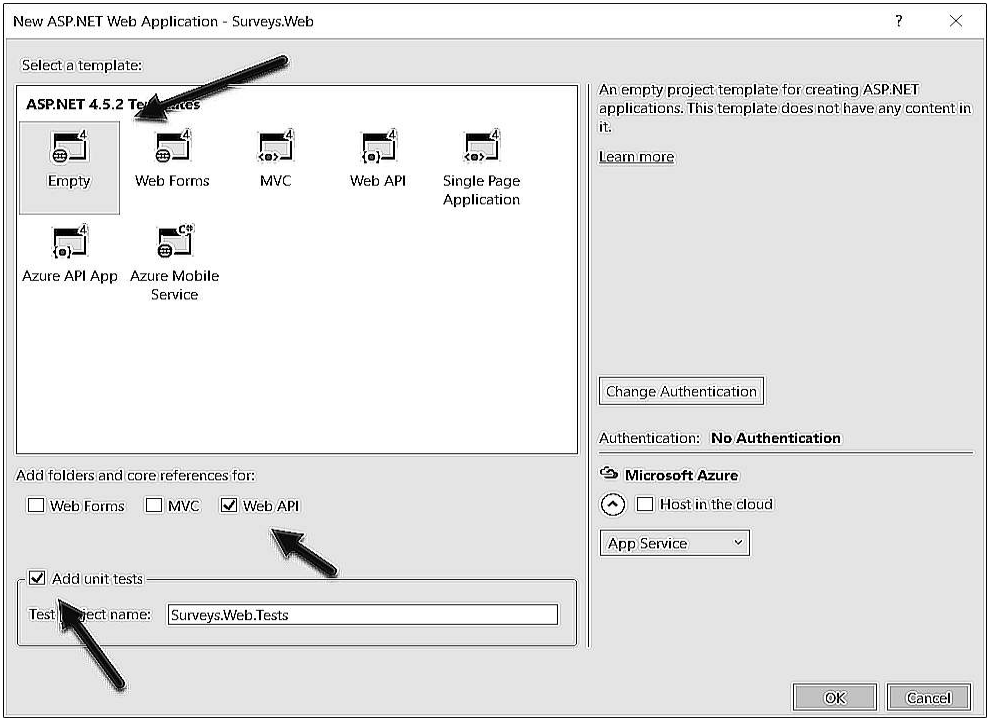
TABLA SURVEYS

La siguiente tabla muestra el esquema que tendrá la tabla Surveys:

Id	nvarchar(50)	Identificador único de la encuesta. Esta es la Clave Primaria (PK) de la tabla
Name	nvarchar(50)	El nombre de la persona que contestó la encuesta
Birthdate	datetime	La fecha de nacimiento de la persona que contestó la encuesta
TeamId	int	El identificador único del equipo que seleccionó la persona
Lat	decimal(18, 4)	La latitud en donde se capturó la encuesta
Lon	decimal(18, 4)	La longitud en donde se capturó la encuesta

Creando el proyecto Web API

En la solución actual llamada Surveys, agregaremos un nuevo proyecto llamado "Surveys.Web" usando la plantilla "ASP.NET Web Application (.NET Framework)". Al crear el proyecto, Visual Studio .NET nos mostrará una caja de diálogo en donde podemos seleccionar la plantilla específica para este proyecto Web. En nuestro caso, usaremos la plantilla "Empty", ya que con ella podemos seleccionar manualmente las referencias que deseamos tener en el proyecto. En nuestro caso, seleccionaremos "Web API". Adicionalmente, marcaremos la opción "Add unit tests" para que Visual Studio .NET también agregue a la solución un proyecto de Pruebas Unitarias para el Web API llamado "Surveys.Web.Tests". Una vez seleccionadas estas opciones, haremos clic en el botón "OK".

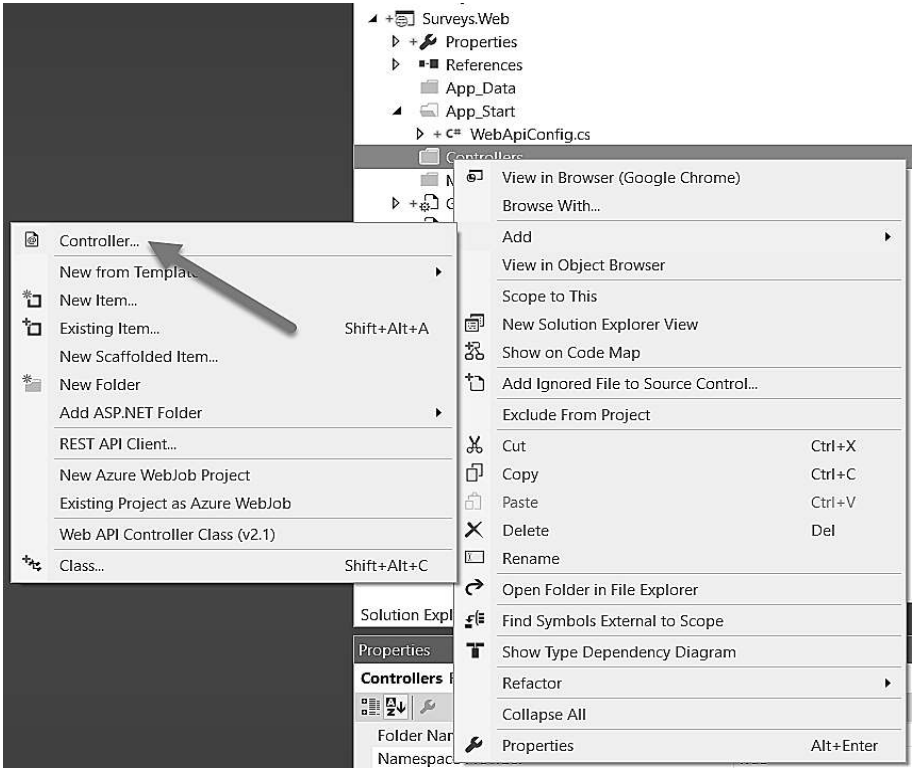


Una vez creados ambos proyectos Surveys.Web y Surveys.Web.Tests, crearemos dos controladores llamados SurveysController y TeamsController en el proyecto Surveys.Web.

Creando los controladores SurveysController y TeamsController

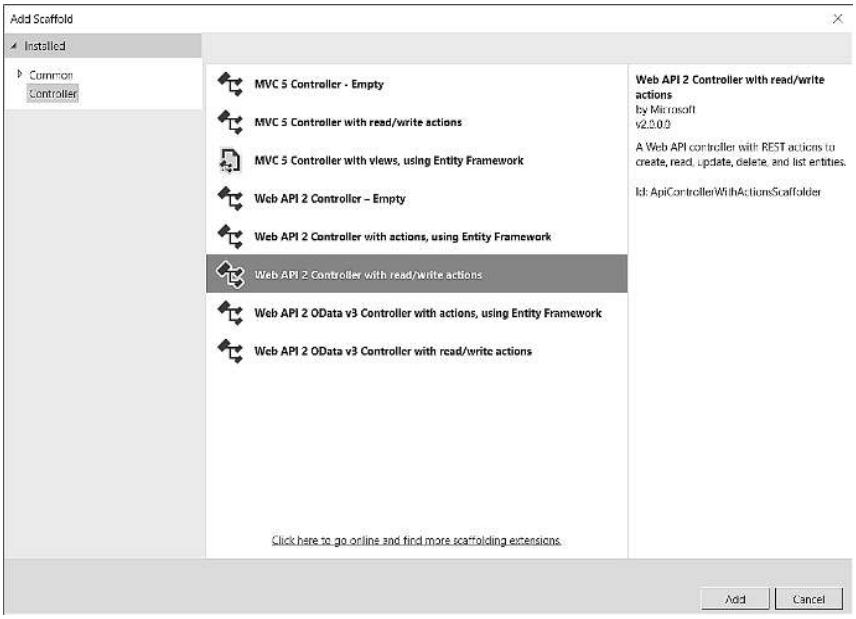
En el folder "Controllers" del proyecto Surveys.Web, agregaremos dos nuevos controladores llamados SurveysController y TeamsController. El primero será el controlador para ejecutar operaciones CRUD sobre los datos de las encuestas (consulta, inserción y borrado principalmente). El segundo será el controlador para consultar los datos de los equipos que serán descargados a la aplicación móvil.

Para crear los controladores, haremos clic secundario en folder y seleccionaremos la opción "Add -> Controller…" tal y como se demuestra en la siguiente figura:



Al seleccionar esta opción, Visual Studio .NET nos muestra una caja de diálogo en donde podemos seleccionar la plantilla de controlador que queremos usar. Hay diferentes tipos de plantillas para los controladores y cada uno de ellos tiene una razón de ser. Sin embargo, nosotros seleccionaremos la plantilla llamada "Web API 2 Controller – Empty", ya que esta plantilla creará una clase completamente limpia y que nos será de gran ayuda para explicar paso a paso la implementación del servicio REST para la aplicación de encuestas.

La siguiente figura muestra la selección de la plantilla que vamos a utilizar en el momento de crear un controlador:



Después de hacer clic en el botón “Add”, Visual Studio .NET creará el controlador deseado. El siguiente código muestra la clase SurveysController:

```
using System.Web.Http;
namespace Surveys.Web.Controllers
{
    public class SurveysController : ApiController
    {
    }
}
```

El siguiente fragmento de código muestra la clase TeamsController:

```
using System.Web.Http;

namespace Surveys.Web.Controllers
{
    public class TeamsController : ApiController
    {
```



```
        }
    }
```

Creando el proyecto Surveys.Entities

Ya que la intención del servicio REST es realizar operaciones CRUD en los datos de encuestas, y además consultar los datos de los equipos, necesitamos un ensamblado que permita implementar las clases que pueden ser transportadas de ida y de vuelta hacia el servicio. Por lo tanto, crearemos un nuevo proyecto en la solución llamado Surveys.Entities, usando la plantilla de Bibliotecas de Clases Portables y seleccionando las mismas plataformas destino que tiene el proyecto PCL Surveys.Core. De esta manera, podremos referenciar tanto en Surveys.Core como en Surveys.Web el proyecto Surveys.Entities.

CLASE TEAM

Una vez creado el proyecto Surveys.Entities, agregaremos una nueva clase llamada Team con las siguientes propiedades:

Id	int	Identificador único del equipo
Name	string	Nombre del equipo
Color	string	Color representativo del equipo. El color debe estar representado en formato hexadecimal
Logo	byte[]	Logotipo del equipo

El siguiente código muestra la implementación completa de la clase Team:

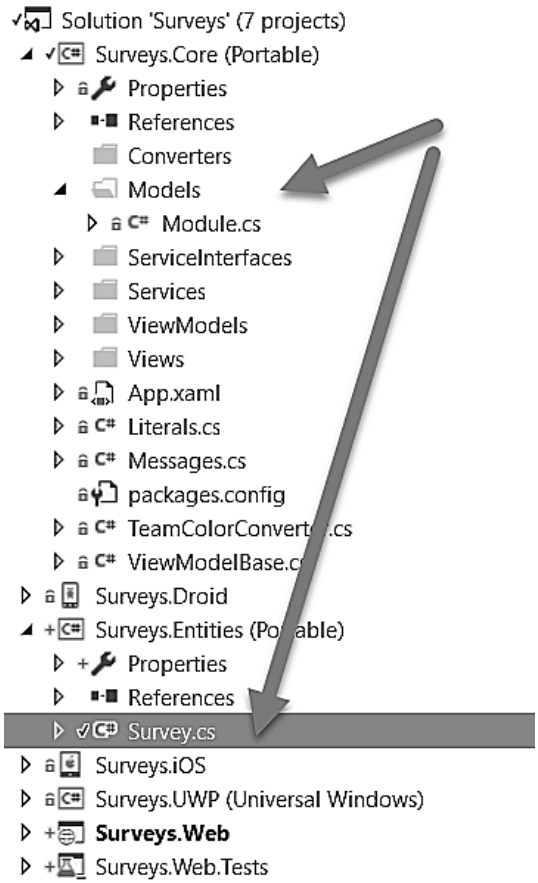
```
namespace Surveys.Entities
{
    public class Team
    {
        public int Id { get; set; }

        public string Name { get; set; }

        public string Color { get; set; }

        public byte[] Logo { get; set; }
```

```
        }
    }
```

CLASE SURVEY

Recordarás que en el proyecto Surveys.Core, tenemos actualmente una clase llamada Survey que representa el modelo para la aplicación. Esta clase la definimos como un DTO (Data Transfer Object), es decir, sin comportamiento, sin estado y también sin referencias duras a ensamblados específicos de Xamarin o Xamarin.Forms. Gracias a eso, la reutilizaremos en el controlador SurveysController del servicio Web API para regresar y solicitar parámetros de tipo Survey.

Moveremos la clase Survey del folder "Models" de Survey.Core a la raíz del proyecto Surveys.Entities, tal y como muestra la siguiente figura:



Después de haber movido la clase Survey entre los proyectos, y para promover un mayor orden en nuestro código, asegúrate de corregir el espacio de nombres de la clase Survey para que sea ahora Surveys.Entities en vez de Suveys.Core. Adicionalmente, renombraremos la propiedad FavoriteTeam a "TeamId" y su tipo de dato en vez de string será int. Esto permitirá correlacionar una encuesta con un equipo del catálogo a través de su identificador único.

El siguiente código muestra la clase Survey una vez hechas las modificaciones correspondientes:

```
using System;


namespace Surveys.Entities
{
    public class Survey
    {
        public string Id { get; set; }

        public string Name { get; set; }

        public DateTime Birthdate { get; set; }

        public int TeamId { get; set; }

        public double Lat { get; set; }

        public double Lon { get; set; }

        public override string ToString()
        {
            return $"{Id} {Name} |  {Birthdate} |
{TeamId} | {Lat} | {Lon}";
        }
    }
}
```

Finalmente, referenciaremos el proyecto Surveys.Entities en ambos proyectos Surveys.Core y Surveys.Web. En el caso específico del proyecto Surveys.Core, tendrás también que corregir los errores que tendrá el código en los ViewModels y

en ILocalDbService, simplemente trayendo a tu alcance los miembros del espacio de nombres Surveys.Entities:

```
using Surveys.Entities;
```

Creando el proyecto Surveys.Web.DAL.SqlServer

Agregaremos a nuestra solución un nuevo proyecto de tipo Biblioteca de Clases llamado Surveys.Web.DAL.SqlServer. En este proyecto implementaremos las clases de proveedores de datos para encuestas y equipos, los cuales serán almacenados en una base de datos SQL Server. Para poder obtener el valor de las cadenas de conexión del archivo Web.config de la aplicación Web, es necesario que agreguemos también la referencia al ensamblado System.Configuration.

IMPLEMENTANDO SQLSERVERPROVIDER

Hay diversas maneras y caminos que podemos usar del lado del servidor para acceder a los datos. No obstante, en este escenario usaremos directamente las clases de ADO.NET, por tratarse de una tecnología probada, con excelente desempeño, facilidad y sobre todo para evitar el uso de capas adicionales que causa el uso de los ORMs como por ejemplo el Entity Framework.

Lo primero que haremos será agregar una nueva clase abstracta llamada SqlServerProvider, que tendrá como objetivo encapsular las operaciones hacia la base de datos SQL Server. El siguiente código muestra la implementación completa de esta clase:

```
using System.Data;
using System.Data.SqlClient;
using System.Threading.Tasks;

namespace Surveys.Web.DAL.SqlServer
{
    public abstract class SqlServerProvider
    {
        public abstract string ConnectionString { get;
set; }


        public virtual Task<int>
ExecuteNonQueryAsync(string query, SqlParameter[]
parameters = null,
```



```
            CommandType commandType = CommandType.Text)
        {
            Task<int> result;
            SqlConnection conn = new
SqlConnection(ConnectionString);
            using (SqlCommand cmd = new
SqlCommand(query, conn))
            {
                cmd.CommandType = commandType;

                if (parameters != null)
                {

cmd.Parameters.AddRange(parameters);
                }
                conn.Open();

                result = cmd.ExecuteNonQueryAsync();

                result.ContinueWith((t) =>
                {
                    cmd.Connection = null;
                    if (conn != null && conn.State !=
ConnectionState.Closed)
                    {
                        conn.Close();
                        conn.Dispose();
                        conn = null;
                    }
                });

                return result;
            }
        }
```

```
        public virtual Task<SqlDataReader>
ExecuteReaderAsync(string query, SqlParameter[]
parameters = null,
            CommandType commandType = CommandType.Text)
        {
            Task<SqlDataReader> result = null;
            SqlConnection conn = new
SqlConnection(ConnectionString);

            using (SqlCommand cmd = new
SqlCommand(query, conn))
            {
                cmd.CommandType = commandType;
                if (parameters != null)
                {
cmd.Parameters.AddRange(parameters);
                }
                conn.Open();
                result =
cmd.ExecuteReaderAsync(CommandBehavior.CloseConnection);
            }

            return result;
        }
    }
}
```

Como podrás observar en el anterior código, estamos implementando dos métodos: ExecuteNonQueryAsync() y ExecuteReaderAsync(), además de una propiedad abstracta llamada ConnectionString. Esta propiedad permitirá a una clase derivada concreta el definir de manera independiente la cadena de conexión hacia la base de datos.

IMPLEMENTANDO TEAMSPROVIDER

Agregaremos ahora una clase llamada TeamsProvider, la cual representa el proveedor de datos para los datos de los equipos. Esta clase hereda de la clase base abstracta SqlServerProvider que acabamos de implementar, por lo que será necesario establecer el valor para la propiedad ConnectionString. Observa que estamos usando la clase ConfigurationManager para obtener el valor de la cadena de conexión llamada "Surveys" del archivo Web.config de la aplicación Web.

TeamsProvider tendrá un solo método llamado GetAllTeamsAsync(), el cual regresará la colección de equipos que obtenga de la tabla Teams en la base de datos:

```
using System;
using System.Collections.Generic;
using System.Data.SqlClient;
using System.Threading.Tasks;
using Surveys.Entities;

namespace Surveys.Web.DAL.SqlServer
{
    public class TeamsProvider : SqlServerProvider
    {
        public override string ConnectionString { get;
set; } =

System.Configuration.ConfigurationManager.ConnectionStrin
gs["Surveys"].ConnectionString;

        public async Task<IEnumerable<Team>>
GetAllTeamsAsync()
        {
            var result = new List<Team>();
            var query = "SELECT * FROM Teams";

            using (var reader = await
ExecuteReaderAsync(query))
            {
                while (reader.Read())
```

```
                    {
result.Add(GetTeamFromReader(reader));
                    }
                }

                return result;
            }

            private Team GetTeamFromReader(SqlDataReader
reader)
            {
                return new Team()
                {
                    Id = (int)reader[nameof(Team.Id)],
                    Name =
reader[nameof(Team.Name)].ToString(),
                    Color =
reader[nameof(Team.Color)].ToString(),
                    Logo = reader[nameof(Team.Logo)] is
DBNull ? null : (byte[])reader[nameof(Team.Logo)]
                };
            }
        }
    }
```

IMPLEMENTANDO SURVEYSPROVIDER

En el proyecto Surveys.Web.DAL.SqlServer, agregaremos una nueva clase llamada SurveysProvider, la cual será la responsable tanto de consultar las encuestas existentes, como de insertar nuevas encuestas en la base de datos, a través de los métodos GetAllSurveysAsync() e InsertSurveyAsync(), respectivamente. Además, usaremos la cadena de conexión “Surveys” del archivo Web.Config y la usaremos para establecer el valor de la propiedad ConnectionString.

El siguiente código muestra la implementación completa de la clase SurveysProvider:


```
using System;
using System.Collections.Generic;
using System.Data.SqlClient;
using System.Threading.Tasks;
using Surveys.Entities;

namespace Surveys.Web.DAL.SqlServer
{
    public class SurveysProvider : SqlServerProvider
    {
        public override string ConnectionString { get;
set; } =
System.Configuration.ConfigurationManager.ConnectionStrin
gs["Surveys"].ConnectionString;

        public async Task<IEnumerable<Survey>>
GetAllSurveysAsync()
        {
            var result = new List<Survey>();
            var query = "SELECT * FROM Surveys";

            using (var reader = await
ExecuteReaderAsync(query))
            {
                while (reader.Read())
                {
result.Add(GetSurveyFromReader(reader));
                }
            }

            return result;
        }
```

```
        public async Task<int> InsertSurveyAsync(Survey
survey)
        {
            if (survey == null)
            {
                return 0;
            }

            var query = @"INSERT INTO Surveys (Id,
Name, Birthdate, TeamId, Lat, Lon)
                          VALUES
                          (@Id, @Name, @Birthdate,
@TeamId, @Lat, @Lon)";

            var parameters = new List<SqlParameter>
            {
                new SqlParameter("@Id",
GetDataValue(survey.Id)),
                new SqlParameter("@Name",
GetDataValue(survey.Name)),
                new SqlParameter("@Birthdate",
survey.Birthdate),
                new SqlParameter("@TeamId",
survey.TeamId),
                new SqlParameter("@Lat", survey.Lat),
                new SqlParameter("@Lon", survey.Lon)
            };

            var result = await
ExecuteNonQueryAsync(query, parameters.ToArray());

            return result;
        }

        private Survey
GetSurveyFromReader(SqlDataReader reader)
```

```
            {
                return new Survey()
                {
                    Id =
reader[nameof(Survey.Id)].ToString(),
                    Name =
reader[nameof(Team.Name)].ToString(),
                    Birthdate =
(DateTime)reader[nameof(Survey.Birthdate)],
                    TeamId =
(int)reader[nameof(Survey.TeamId)],
                    Lat =
(double)reader[nameof(Survey.Lat)],
                    Lon =
(double)reader[nameof(Survey.Lon)]
                };
            }
        }
    }
```

Implementando el controlador de equipos

El controlador de equipos en nuestro servicio REST, implementado en la clase TeamsController, será el responsable de consultar los equipos posibles que podrá elegir un usuario en una encuesta. El controlador tendrá una única operación, como a continuación describe la siguiente tabla:

/api/teams	GET	Regresa toda la lista de equipos

Basándonos en este requerimiento, implementaremos en la clase TeamsController un nuevo método llamado Get(), que devolverá una colección de objetos de tipo Team. Para lograr este propósito, usaremos el proveedor de datos TeamsProvider.

El siguiente código muestra la implementación del método Get() en la clase TeamsController:

```
using System.Collections.Generic;
using System.Threading.Tasks;
using System.Web.Http;
using Surveys.Entities;
using Surveys.Web.DAL.SqlServer;

namespace Surveys.Web.Controllers
{
    public class TeamsController : ApiController
    {
        private readonly TeamsProvider teamsProvider =
new TeamsProvider();

        public async Task<IEnumerable<Team>> Get()
        {
            var allTeams = await
teamsProvider.GetAllTeamsAsync();
            var result = new List<Team>(allTeams);

            return result;
        }
    }
}
```

Implementando el controlador de encuestas

El controlador de encuestas, implementado en la clase SurveysController, será el responsable de exponer las operaciones de consulta e inserción de encuestas. Este controlador tendrá las siguientes operaciones:

/api/surveys	GET	Regresa todas las encuestas
/api/surveys	POST	Envía una colección de encuestas para insertarlas

El siguiente código muestra la clase SurveysController, y la implementación de sus métodos Get() y Post():


```
using System.Collections.Generic;
using System.Threading.Tasks;
using System.Web.Http;
using Surveys.Entities;
using Surveys.Web.DAL.SqlServer;

namespace Surveys.Web.Controllers
{
    public class SurveysController : ApiController
    {
        private readonly SurveysProvider
surveysProvider = new SurveysProvider();

        public async Task<IEnumerable<Survey>> Get()
        {
            var allSurveys = await
surveysProvider.GetAllSurveysAsync();
            return allSurveys;
        }
        public async Task Post([FromBody]
IEnumerable<Survey> surveys)
        {
            if (surveys == null)
            {
                return;
            }

            foreach (var survey in surveys)
            {
                await
surveysProvider.InsertSurveyAsync(survey);
            }
```

```
            }
        }
    }
```

Nota: Como podrás observar en el anterior código, estamos iterando sobre la colección de encuestas que recibe el método Post(), y usamos cada elemento para mandarlo como parámetro al método InsertSurveyAsync() del proveedor SurveysProvider. Probablemente, querrás después modificar el proveedor y agregar un nuevo método que reciba toda la colección y con ella armar el query completo.

Creando el servicio IWebApiService

En nuestra aplicación de encuestas, necesitamos un servicio capaz de comunicarse con el servicio REST que acabamos de construir usando ASP.NET Web API. El objetivo de este servicio será obtener la colección de equipos que expone el servicio REST, y subir las nuevas encuestas capturadas en el dispositivo. Este servicio estará basado en una nueva interfaz llamada IWebApiService en el folder "ServiceInterfaces" del proyecto PCL. La interfaz tendrá los siguientes miembros que a continuación se describen:

GetTeamsAsync()	Método para obtener la colección de equipos
SaveSurveysAsync()	Método para subir todas las encuestas capturadas

El siguiente código muestra la implementación completa de la interfaz IWebApiService:

```
using System.Collections.Generic;
using System.Threading.Tasks;
using Surveys.Entities;

namespace Surveys.Core.ServiceInterfaces
{
    public interface IWebApiService
    {
        Task<IEnumerable<Team>> GetTeamsAsync();

        Task<bool> SaveSurveysAsync(IEnumerable<Survey>
surveys);
```



```
        }
    }
```

Posteriormente, crearemos una nueva clase llamada WebApiService en el folder "Services" del proyecto PCL, la cual implementará de manera concreta la interfaz IWebApiService que acabamos de agregar.

Usaremos la clase HttpClient como mecanismo de comunicación al servicio REST, ya que esta clase nos permite construir peticiones y recibir respuestas a través del protocolo HTTP. La propiedad BaseClient de la clase HttpClient nos permite establecer un URI raíz, el cual será usado por todas las peticiones. Este URI lo implementaremos como constante en la clase Literals de la PCL:

```
    public const string WebApiServiceBaseAddress =
@"http://EL URI DE TU SERVICIO/";
```

Una vez creada la constante en la clase Literals, podremos instanciar el objeto HttpClient en el constructor de la clase WebApiService, así como lo muestra el siguiente fragmento de código:

```
    public class WebApiService : IWebApiService
    {
        private readonly HttpClient client;


        public WebApiService()
        {
            client = new HttpClient { BaseAddress = new
Uri(Literals.WebApiServiceBaseAddress) };
        }
    …
    }
```

Con el objeto HttpClient disponible, implementamos cada método de la interfaz IWebApiService. Ya que los datos del servicio serán transportados de ida y de vuelta en formato JSON, agregaremos en la PCL la referencia a un paquete de NuGet llamado "Newtonsoft.Json", el cual nos permite manejar los datos de JSON en las aplicaciones .NET de una forma sumamente fácil.

En el caso del método GetTeamsAsync(), usaremos el método GetStringAsync() de HttpClient para obtener en cadena el resultado con toda la colección de equipos. Una vez obtenida la cadena, la deserializaremos para obtener el objeto original que

está devolviendo el servicio, en este caso, un IEnumerable<Team>. La deserialización será llevada a cabo por el método DeserializeObject<T>() de la clase JsonConvert. El siguiente fragmento de código muestra la implementación del método GetTeamsAsync():

```
  public async Task<IEnumerable<Team>> GetTeamsAsync()
  {
      IEnumerable<Team> result = null;
      var teams = await
client.GetStringAsync("/api/teams");

      if (!string.IsNullOrWhiteSpace(teams))
      {
          result =
JsonConvert.DeserializeObject<IEnumerable<Team>>(teams);
          return result;
      }

      return result;
  }
```

Por otro lado, en el caso del método SaveSurveysAsync(), usaremos el método PostAsync() de la clase HttpClient para enviar la colección de encuestas del parámetro hacia el servicio REST. Sin embargo, antes de enviar la colección la serializaremos en JSON, y usaremos dicha cadena como contenido de un objeto StringContent que será usado como parámetro al invocar el método PostAsync(). El siguiente fragmento de código muestra la implementación completa del método SaveSurveysAsync():

```
  public async Task<bool>
SaveSurveysAsync(IEnumerable<Survey> surveys)
  {
      var content = new
StringContent(JsonConvert.SerializeObject(surveys),
Encoding.UTF8, "application/json");

      var response = await
client.PostAsync("/api/surveys", content);
```



```
        return response.IsSuccessStatusCode;
    }
```

Registrando el servicio IWebApiService

Acto seguido, registraremos en Unity una instancia de WebApiService cuando el contenedor tenga que inyectar un objeto de tipo IWebApiService. El siguiente fragmento de código muestra la modificación en el método ConfigureContainer() de la clase App para registrar esta instancia:

```
    protected override void ConfigureContainer()
    {
        base.ConfigureContainer();


        Container.RegisterInstance<ILocalDbService>(new
LocalDbService());
        Container.RegisterInstance<IWebApiService>(new
WebApiService());
    }
```

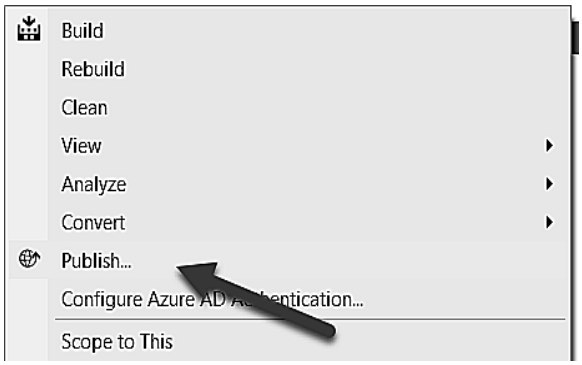
Publicando el servicio REST

Hay diferentes maneras para publicar el servicio REST, y la que elijas dependerá de las condiciones y los requerimientos del proyecto que estés construyendo. Probablemente, en algunos casos ya cuentes con un servidor Web con IIS y el .NET Framework instalado y configurado, y por lo tanto tu intención sea publicar ahí el servicio. En otros casos, usarás algún servicio de cómputo en la nube para lograr este propósito. Para continuar con el desarrollo de nuestra aplicación de encuestas, usaremos Microsoft Azure como plataforma de cómputo en la nube para publicar el servicio.

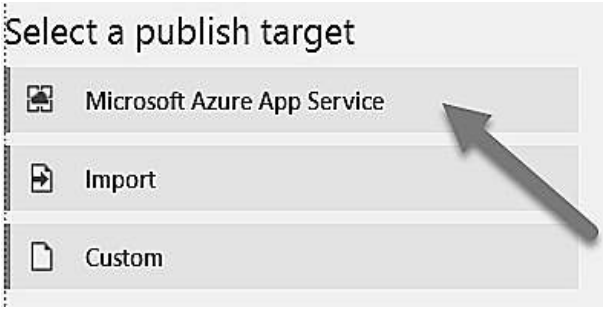
Nota: Si deseas continuar con el desarrollo de este ejemplo usando Azure, deberás tener ya una cuenta. La obtención y apertura de una cuenta en Microsoft Azure está fuera del alcance de este libro, pero te recomiendo visites https://azure.microsoft.com para mayor información al respecto. Para cualquier otro escenario de despliegue de servicios REST construidos con ASP.NET Web API y el .NET Framework, consulta la documentación relacionada con el producto o servicio donde deseas publicarlo.

PUBLICANDO DESDE VISUAL STUDIO .NET

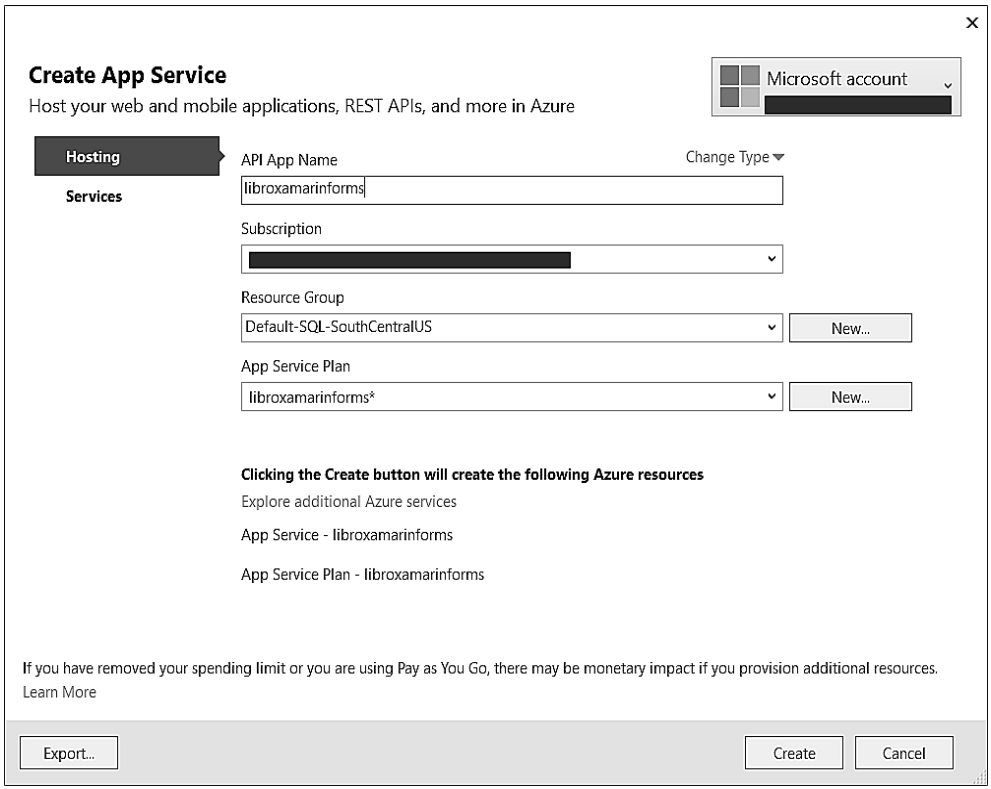
Si controlas el código fuente, la forma más sencilla para publicar un servicio de ASP.NET Web API es a través de Visual Studio .NET. Para hacer esto, haz clic secundario sobre el proyecto Surveys.Web en el explorador de solución y selecciona la opción "Publish…", tal y como se muestra en la siguiente figura:



Después de seleccionar esta opción, Visual Studio .NET te mostrará una caja de diálogo para seleccionar el destino de la publicación. En este caso, seleccionaremos la opción "Microsoft Azure App Service":

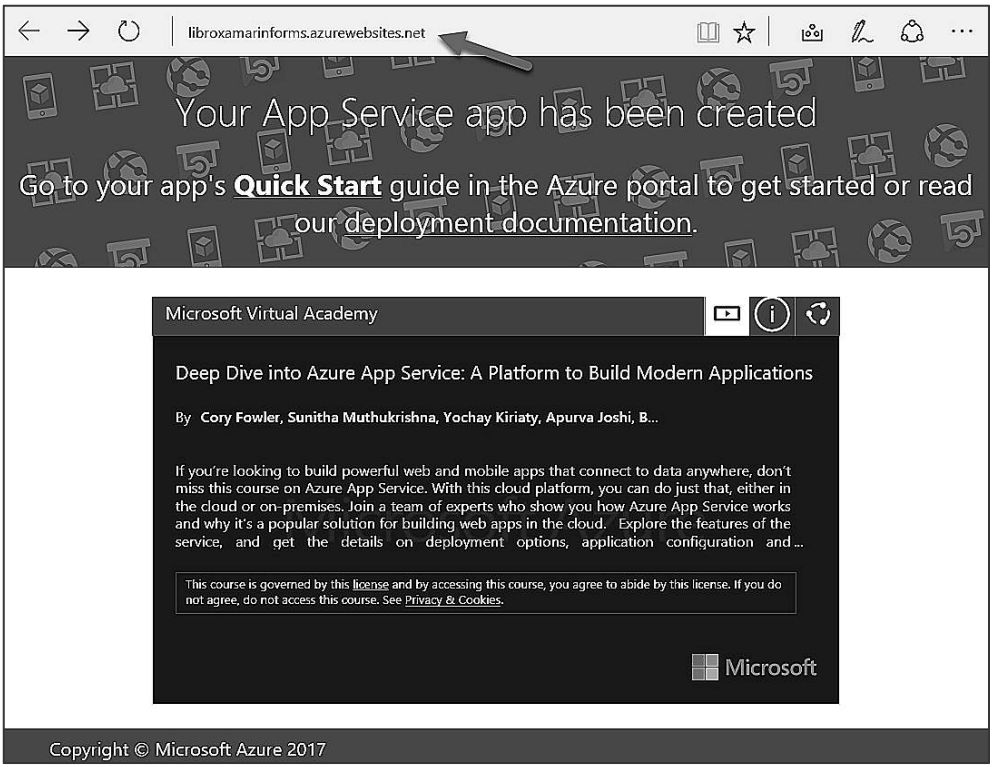


Al seleccionar la opción "Microsoft Azure App Service", Visual Studio .NET te mostrará una caja de diálogo para configurar el App Service en donde deseas publicar tu servicio REST. Deberás seleccionar la cuenta con la que estás registrado en Azure, la suscripción, el grupo de recursos y el nivel de plan. Adicionalmente, deberás especificar el nombre para el servicio. En mi caso, usaré el nombre "libroxamarinforms", tal y como lo muestra la siguiente figura:



Nota: Para mayor información acerca de los App Services de Microsoft Azure, consulta su documentación correspondiente.

Finalmente, haremos clic en el botón “Create” para que el App Service sea creado en la nube y se publique nuestro servicio REST. Si todo ha marchado correctamente, al final Visual Studio .NET abrirá en un navegador la página relacionada con el Web API que acabas de publicar, tal y como lo muestra la siguiente figura:



Observa que el URI final de nuestro servicio es "http://libroxamarinforms.azurewebsites.net/". Si has seguido estos pasos hasta aquí, la dirección que tú hayas especificado es la que deberás poner en la constante WebApiServiceBaseAddress de la clase Literals:

```
public const string WebApiServiceBaseAddress =
@"http://libroxamarinforms.azurewebsites.net/";
```

Probando el servicio REST

Una vez publicado el servicio, simplemente ejecuta una petición a "http://(la dirección de tu servicio)/api/teams" para comprobar que efectivamente el servicio te esté trayendo la lista completa de equipos. En mi caso, haré una petición a "http://libroxamarinforms.azurewebsites.net/api/teams", ya que como recordarás, esa es la dirección del servicio que especifiqué cuando lo publiqué desde Visual Studio .NET.

La siguiente figura muestra la respuesta obtenida con esta petición:

Nota: Todos los caracteres raros que ves en la figura anterior, no son más que los logotipos de los equipos serializados en JSON.

Modificando ILocalDbService

Con el servicio REST publicado, modificaremos la interfaz ILocalDbService y su implementación concreta, para poder agregar los métodos para borrar las encuestas y equipos existentes, además de insertar y obtener los equipos que obtengamos del servicio REST.

Los métodos que agregaremos serán los siguientes:

DeleteAllSurveysAsync()	Borra todas las encuestas de la base de datos local
DeleteAllTeamsAsync()	Borra todos los equipos de la base de datos local
InsertTeamsAsync()	Inserta los equipos en la base de datos local
GetAllTeamsAsync()	Obtiene todos los equipos de la base de datos local

El siguiente código muestra la interfaz ILocalDbService con estos nuevos miembros:

```
using System.Collections.Generic;
using System.Threading.Tasks;
using Surveys.Entities;

namespace Surveys.Core.ServiceInterfaces
{
    public interface ILocalDbService
    {
        Task<IEnumerable<Survey>> GetAllSurveysAsync();

        Task InsertSurveyAsync(Survey survey);

        Task DeleteSurveyAsync(Survey survey);

        Task DeleteAllSurveysAsync();
        Task DeleteAllTeamsAsync();

        Task InsertTeamsAsync(IEnumerable<Team> teams);

        Task<IEnumerable<Team>> GetAllTeamsAsync();
    }
}
```

La implementación concreta de los nuevos miembros de la clase LocalDbService será la siguiente:

```
public Task DeleteAllSurveysAsync()
{
    return Task.Run(() =>
connection.DeleteAll<Survey>() > 0);
}
public Task DeleteAllTeamsAsync()
{
    return Task.Run(() => connection.DeleteAll<Team>()
> 0);
}
public Task InsertTeamsAsync(IEnumerable<Team> teams)
```



```
    {
        return Task.Run(() => connection.InsertAll(teams));
    }

    public Task<IEnumerable<Team>> GetAllTeamsAsync()
    {
        return Task.Run(() =>
(IEnumerable<Team>)connection.Table<Team>().ToArray());
    }
```

Asimismo, debemos asegurarnos de crear la tabla Team en la base de datos local. Esto lo haremos modificando el método CreateDatabase() para agregar el código que invoca el método CreateTable() del objeto SQLiteConnection, tal y como muestra el siguiente fragmento de código:

```
    private void CreateDatabase()
    {
        if (connection.TableMappings.All(t => t.TableName
!= nameof(Survey)))
        {
            connection.CreateTable<Survey>();
        }

        if (connection.TableMappings.All(t => t.TableName
!= nameof(Team)))
        {
            connection.CreateTable<Team>();
        }
    }
```

Creando el módulo de sincronización

La aplicación de encuestas está diseñada para poder usarla aun sin conexión a Internet. Por lo tanto, necesitamos implementar una página donde se puedan sincronizar los datos manualmente. El usuario de la aplicación será el responsable de abrir esta página y sincronizar los datos cuando considere oportuno.

Agregaremos una nueva página llamada SyncView en el folder "Views" y su respectiva clase SyncViewModel en el folder "ViewModels" del proyecto PCL.

IMPLEMENTANDO SYNCVIEWMODEL

En la clase SyncViewModel, agregaremos los siguientes miembros:

Status	Propiedad de tipo string que indica el estatus de la sincronización
IsBusy	Propiedad de tipo bool que indica si está activo el proceso de sincronización
SyncCommand	Comando para ejecutar el proceso de sincronización

En el método OnNavigatingTo(), implementaremos la lógica para determinar cuándo se ejecutó el proceso de sincronización por última vez. Le indicaremos al usuario esta información a través de la propiedad Status.

En la acción relacionada con el comando SyncCommand, primero enviaremos todas las encuestas que hayan sido almacenadas localmente en el dispositivo, y posteriormente obtendremos los equipos que nos devuelva el servicio, para finalmente guardarlos localmente y de esa manera refrescar el catálogo. Además, guardaremos en el diccionario de propiedades de la aplicación la fecha y hora de la última sincronización.

Nota: Este es un buen escenario para usar el diccionario de propiedades de la aplicación, ya que crear una tabla en la base de datos local solo para almacenar una fecha probablemente sea excesivo.

El siguiente código muestra la implementación completa de la clase SyncViewModel:

```
using System.Linq;
using System.Windows.Input;
using Prism.Commands;
using Surveys.Core.ServiceInterfaces;

namespace Surveys.Core.ViewModels
{
    public class SyncViewModel : ViewModelBase
    {
        private IWebApiService webApiService = null;
        private ILocalDbService localDbService = null;
```



```
private string status;

public string Status
{
    get
    {
        return status;
    }
    set
    {
        if (status == value)
        {
            return;
        }
        status = value;
        OnPropertyChanged();
    }
}

private bool isBusy;

public bool IsBusy
{
    get
    {
        return isBusy;
    }
    set
    {
        if (isBusy == value)
        {
```

```
                    return;
                }
                isBusy = value;
                OnPropertyChanged();
            }
        }

        public ICommand SyncCommand { get; set; }

        public SyncViewModel(IWebApiService
webApiService, ILocalDbService localDbService)
        {
            this.webApiService = webApiService;
            this.localDbService = localDbService;

            SyncCommand = new
DelegateCommand(SyncCommandExecute);
        }

        public override void
OnNavigatingTo(NavigationParameters parameters)
        {
            base.OnNavigatingTo(parameters);

            Status =
Application.Current.Properties.ContainsKey("lastSync")
                ? $"Última actualización:
{(DateTime)Application.Current.Properties["lastSync"]}"
                : "No se han sincronizado los datos";
        }

        private async void SyncCommandExecute()
        {
            IsBusy = true;
```

```
                //Envía las encuestas
                var allSurveys = await
localDbService.GetAllSurveysAsync();

                if (allSurveys != null && allSurveys.Any())
                {
                    await
webApiService.SaveSurveysAsync(allSurveys);
                    await
localDbService.DeleteAllSurveysAsync();
                }

                //Consulta los equipos
                var allTeams = await
webApiService.GetTeamsAsync();
                if (allTeams != null && allTeams.Any())
                {
                    await
localDbService.DeleteAllTeamsAsync();
                    await
localDbService.InsertTeamsAsync(allTeams);
                }

                Application.Current.Properties["lastSync"]
= DateTime.Now;
                await
Application.Current.SavePropertiesAsync();
                Status = $"Se enviaron {allSurveys.Count()}
encuestas y se obtuvieron {allTeams.Count()} equipos";
                IsBusy = false;
            }
        }
    }
```

IMPLEMENTANDO SYNCVIEW

La página SyncView tendrá un ToolbarItem para iniciar el proceso de sincronización, además de una etiqueta que nos muestre el estatus, así como también un indicador de actividad. La etiqueta se ocultará cuando el proceso de sincronización esté activo. Esto lo lograremos con un DataTrigger con toda la intención de ahorrarnos un Convertidor de Valor que convirtiera un bool a su valor falso y que pudiéramos usar para enlazar la propiedad IsVisible de la etiqueta a la propiedad IsBusy del ViewModel.

Adicionalmente, agregaremos una nueva imagen llamada "sync.png" a cada proyecto concreto, la cual identificará al ToolbarItem en la barra de herramientas y también al módulo de sincronización que crearemos a continuación.

El siguiente código muestra la implementación completa de la página SyncView.

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage
xmlns="http://xamarin.com/schemas/2014/forms"

xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="Surveys.Core.Views.SyncView">
    <Grid Margin="10">
      <Label Text="{Binding Status}"
VerticalOptions="Center"
HorizontalTextAlignment="Center">
        <Label.Style>
          <Style TargetType="Label">
            <Style.Triggers>
              <DataTrigger TargetType="Label"
Binding="{Binding IsBusy}" Value="True">
                <Setter Property="IsVisible"
Value="False" />
              </DataTrigger>
            </Style.Triggers>
          </Style>
        </Label.Style>
      </Label>
```



```
    <ActivityIndicator IsRunning="{Binding IsBusy}"
                       IsVisible="{Binding IsBusy}"
                       HorizontalOptions="Center"
                       VerticalOptions="Center" />
  </Grid>

  <ContentPage.ToolbarItems>
    <ToolbarItem Command="{Binding SyncCommand}">
      <ToolbarItem.Icon>
        <OnPlatform x:TypeArguments="FileImageSource"
                    Android="sync.png"
                    iOS="sync.png"
                    WinPhone="assets/sync.png" />
      </ToolbarItem.Icon>
    </ToolbarItem>
  </ContentPage.ToolbarItems>
</ContentPage>
```

REGISTRANDO EL TIPO NAVEGABLE

Como recordarás, en Prism es necesario registrar de antemano los tipos a los que se puede navegar. Por esta razón, modificaremos el método RegisterTypes() de la clase App para agregar la siguiente línea de código:

```
Container.RegisterTypeForNavigation<SyncView>();
```

CREANDO EL MÓDULO DE SINCRONIZACIÓN

En MainViewModel, agregaremos un nuevo módulo con el título “Sincronización”, después del módulo de encuestas. Este nuevo módulo navegará a SyncView y usará como icono la imagen sync.png que acabamos de agregar en los proyectos de cada plataforma.

El siguiente fragmento de código muestra la creación de este nuevo módulo:

```
…
new Module()
```

```
{
    Icon = "sync.png",
    Title = "Sincronización",
    LoadModuleCommand =
        new DelegateCommand(
            async () =>
                await
navigationService.NavigateAsync($"{nameof(RootNavigationV
iew)}/{nameof(SyncView)}"))
},
…
```

Implementando la nueva página de selección de equipos

Al capturar una nueva encuesta, utilizamos el método DisplayActionSheetAsync() para mostrar la lista de equipos. No obstante, ahora que tenemos también el logotipo del equipo este mecanismo resulta insuficiente. Precisamente por esta razón, es un excelente momento de implementar una nueva página de selección de equipos la cual mostraremos de manera modal. Esta página la llamaremos TeamSelectionView y su clase de ViewModel será TeamSelectionViewModel. Sin embargo, primero debemos crear un nuevo ViewModel que contenga las propiedades que serán usadas para presentar los equipos.

IMPLEMENTACIÓN DE TEAMVIEWMODEL

En la clase TeamViewModel modelaremos los datos necesarios para presentar un equipo, ya que ahora deseamos mostrar también su logotipo. Recuerda que el patrón de diseño Model-View-ViewModel promueve el "modelado" de datos para una vista en específico. Justamente de ahí proviene el nombre de la parte "ViewModel".

La siguiente tabla describe los miembros que tendrá esta clase:

Id	Propiedad de tipo int que representa el identificador único del equipo
Name	Propiedad de tipo string que representa el nombre del equipo

Color	Propiedad de tipo string que representa el color del equipo
Logo	Propiedad de tipo ImageSource que representa el logotipo del equipo
GetViewModelFromEntity()	Método estático para regresar un objeto de tipo TeamViewModel a partir de un objeto de tipo entidad (Surveys.Entities.Team)

El siguiente código muestra la implementación completa de la clase TeamViewModel:

```
using System.IO;
using Prism.Mvvm;
using Surveys.Entities;
using Xamarin.Forms;

namespace Surveys.Core.ViewModels
{
    public class TeamViewModel : BindableBase
    {
        private int id;

        public int Id
        {
            get
            {
                return id;
            }
            set
            {
                if (id == value)
                {
                    return;
                }
```

```
            id = value;
            OnPropertyChanged();
        }
    }

    private string name;

    public string Name
    {
        get
        {
            return name;
        }
        set
        {
            if (name == value)
            {
                return;
            }
            name = value;
            OnPropertyChanged();
        }
    }

    private string color;

    public string Color
    {
        get
        {
            return color;
        }
        set
```

```
            {
                if (color == value)
                {
                    return;
                }
                color = value;
                OnPropertyChanged();
            }
        }

        private ImageSource logo;

        public ImageSource Logo
        {
            get
            {
                return logo;
            }
            set
            {
                if (logo == value)
                {
                    return;
                }
                logo = value;
                OnPropertyChanged();
            }
        }

        public static TeamViewModel
GetViewModelFromEntity(Team entity)
        {
            var result = new TeamViewModel
```

```
                {
                    Id = entity.Id,
                    Name = entity.Name,
                    Color = entity.Color,
                    Logo = ImageSource.FromStream(() => new
MemoryStream(entity.Logo))
                };

                return result;
            }
        }
    }
```

Presta especial atención en la propiedad Logo. En una entidad Team contamos con el arreglo de bytes que representa la imagen binaria del logotipo. Pero en un elemento Image en Xamarin.Forms no podemos usar directamente ese arreglo como fuente de imagen. Ahí es donde destaca y brilla el patrón de diseño MVVM al sugerirnos modelar esta clase e implementar la lógica necesaria para poder enlazarla a la vista.

IMPLEMENTACIÓN DE TEAMSELECTIONVIEWMODEL

La clase TeamSelectionViewModel es el ViewModel para la página TeamSelectionView. En ella, implementaremos los siguientes miembros:

Teams	Colección de tipo ObservableCollection<TeamViewModel> la cual contiene todos los equipos
SelectedTeam	Propiedad que representa el equipo seleccionado en la lista

En el método OnNavigatingTo(), implementaremos la lógica necesaria para obtener los equipos de la base de datos local, que previamente el usuario debió sincronizar usando el módulo correspondiente. Ya que Teams es de tipo ObservableCollection<TeamViewModel> y nosotros obtenemos un IEnumerable<Team> por medio de ILocalDbService, entonces debemos convertir la colección de objetos Team a una colección de objetos TeamViewModel. Esto lo lograremos por medio del método estático GetViewModelFromEntity() de la clase TeamViewModel, descrito anteriormente.

Una vez lista la colección Teams, la intención es que, cuando un usuario seleccione un equipo, devolvamos un parámetro con el identificador único del

equipo seleccionado. Eso lo haremos mandando como parámetro un objeto de tipo NavigationParameters en el método GoBackAsync() de la interfaz INavigationService.

El siguiente código muestra la implementación completa de la clase TeamSelectionViewModel:

```
using System.Collections.ObjectModel;
using System.Linq;
using Prism.Navigation;
using Surveys.Core.ServiceInterfaces;

namespace Surveys.Core.ViewModels
{
    public class TeamSelectionViewModel : ViewModelBase
    {
        private INavigationService navigationService =
null;
        private ILocalDbService localDbService = null;

        private ObservableCollection<TeamViewModel>
teams;

        public ObservableCollection<TeamViewModel>
Teams
        {
            get
            {
                return teams;
            }
            set
            {
                if (teams == value)
                {
                    return;
                }
```

```
                teams = value;
                OnPropertyChanged();
            }
        }

        private TeamViewModel selectedTeam;

        public TeamViewModel SelectedTeam
        {
            get
            {
                return selectedTeam;
            }
            set
            {
                if (selectedTeam == value)
                {
                    return;
                }
                selectedTeam = value;
                OnPropertyChanged();
            }
        }

        public
TeamSelectionViewModel(INavigationService
navigationService, ILocalDbService localDbService)
        {
            this.navigationService = navigationService;
            this.localDbService = localDbService;

            PropertyChanged +=
TeamSelectionViewModel_PropertyChanged;
        }
```

```
        public override async void
OnNavigatingTo(NavigationParameters parameters)
        {
            base.OnNavigatingTo(parameters);

            var allTeams = await
localDbService.GetAllTeamsAsync();

            if (allTeams != null)
            {
                Teams = new
ObservableCollection<TeamViewModel>(allTeams.Select(TeamV
iewModel.GetViewModelFromEntity));
            }
        }
        private async void
TeamSelectionViewModel_PropertyChanged(object sender,

System.ComponentModel.PropertyChangedEventArgs e)
        {
            if (e.PropertyName == nameof(SelectedTeam))
            {
                if (SelectedTeam == null)
                {
                    return;
                }

                var param = new NavigationParameters {
{ "id", SelectedTeam.Id } };
                await
navigationService.GoBackAsync(param, true);
            }
        }
    }
}
```

IMPLEMENTACIÓN DE TEAMSELECTIONVIEW

La página TeamSelectionView tiene como objetivo mostrar una lista con los equipos que expone TeamSelectionViewModel. Estableceremos como contenido un Grid, el cual tendrá una etiqueta para el título y un ListView enlazado a la colección de equipos.

El siguiente código muestra la implementación completa de TeamSelectionView:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage
xmlns="http://xamarin.com/schemas/2014/forms"

xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"

x:Class="Surveys.Core.Views.TeamSelectionView">
    <Grid Margin="10">
      <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition />
      </Grid.RowDefinitions>

      <Label Text="Seleccione el equipo" FontSize="Large"
Margin="0,0,0,20" />

      <ListView Grid.Row="1"
                ItemsSource="{Binding Teams}"
                SelectedItem="{Binding SelectedTeam,
Mode=TwoWay}">
        <ListView.ItemTemplate>
          <DataTemplate>
            <ViewCell>
              <Grid>
                <Grid.ColumnDefinitions>
                  <ColumnDefinition Width="Auto" />
```



```
                    <ColumnDefinition />
                </Grid.ColumnDefinitions>

                <Image Source="{Binding Logo}"
                       WidthRequest="100"
                       HeightRequest="100" />
                <Label Grid.Column="1" Text="{Binding
Name}" Margin="10,0,0,0" VerticalTextAlignment="Center"
/>
              </Grid>
            </ViewCell>
          </DataTemplate>
        </ListView.ItemTemplate>
      </ListView>
    </Grid>
  </ContentPage>
```

Modificando SurveyDetailsViewModel

Actualmente, en el constructor de la clase SurveyDetailsViewModel, inicializamos la propiedad Teams a través de un arreglo de cadenas con los nombres de los equipos. Ese código lo eliminaremos, ya que la clase que ahora tiene la responsabilidad de mostrar los equipos es TeamSelectionView. Por lo tanto, modificaremos la acción del comando SelectTeamCommand para que presente de manera modal la página TeamSelectionView, en vez de usar IPageDialogService. El siguiente fragmento de código muestra este cambio:

```
  private async void SelectTeamCommandExecute()
  {
      await
navigationService.NavigateAsync(nameof(TeamSelectionView)
, useModalNavigation:true);
  }
```

Adicionalmente, implementaremos en el método OnNavigatedTo() la lógica para determinar si cuando se navega a la vista relacionada es el resultado de haber navegado modalmente a TeamSelectionView, es decir, evaluamos los parámetros

para conocer si existe el parámetro “id” que representa el identificador único del equipo seleccionado. Si es así, usaremos ese valor para obtener de los equipos de la base de datos local el nombre de dicho equipo.

El siguiente fragmento de código explica con más detalle esta modificación:

```
…
private IEnumerable<Team> localDbTeams = null;
…
public override async void
OnNavigatedTo(NavigationParameters parameters)
{
    base.OnNavigatedTo(parameters);

    localDbTeams = await
localDbService.GetAllTeamsAsync();

    if (parameters.ContainsKey("id"))
    {
        FavoriteTeam = localDbTeams.First(t => t.Id ==
(int)parameters["id"]).Name;
    }
}
```

Finalmente, en la acción relacionada con el comando EndSurveyCommand, usaremos la colección de equipos de la base de datos local para establecer el valor de la propiedad TeamId de la nueva encuesta seleccionada:

```
private async void EndSurveyCommandExecute()
{
    var newSurvey = new Survey()
    {
        Id = Guid.NewGuid().ToString(),
        Name = Name,
        Birthdate = Birthdate,
```



```
            TeamId = localDbTeams.First(t => t.Name ==
FavoriteTeam).Id
        };
    …
    }
```

Modificando la página de encuestas

Si ejecutas en este momento la aplicación, podrás observar que en la lista de encuestas capturadas ya no se muestra el equipo seleccionado. Esto se debe a que ahora la clase Survey no tiene el nombre del equipo favorito sino su identificador único.

Para mostrar correctamente los elementos de la lista de encuestas, necesitamos crear una nueva clase que modele los datos de cada encuesta. A esta clase la llamaremos SurveyViewModel.

CREANDO LA CLASE SURVEYVIEWMODEL

Crearemos una nueva clase llamada SurveyViewModel en el folder "ViewModels" del proyecto PCL. Similar a lo ocurrido con TeamViewModel, en esta nueva clase expondremos los miembros necesarios para presentar una encuesta en la lista. La siguiente tabla describe estos miembros:

Name	Propiedad de tipo string que representa el nombre de la persona encuestada
Birthdate	Propiedad de tipo DateTime que representa la fecha de nacimiento de la persona encuestada
Team	Propiedad de tipo TeamViewModel que representa el equipo seleccionado
Lat	Propiedad de tipo double que representa la latitud en donde fue tomada la encuesta
Lon	Propiedad de tipo double que representa la longitud en donde fue tomada la encuesta
GetViewModelFromEntity()	Método estático para obtener un objeto de tipo SurveyViewModel a partir de un objeto de tipo Survey
GetEntityFromViewModel()	Método estático para obtener un objeto de tipo Survey a partir de un objeto de tipo SurveyViewModel

El siguiente código muestra la implementación completa de la clase SurveyViewModel:

```
using System;
using System.Collections.Generic;
using System.Linq;
using Prism.Mvvm;
using Surveys.Entities;

namespace Surveys.Core.ViewModels
{
    public class SurveyViewModel : BindableBase
    {
        public string Id { get; set; }

        private string name;

        public string Name
        {
            get
            {
                return name;
            }
            set
            {
                if (name == value)
                {
                    return;
                }
                name = value;
                OnPropertyChanged();
            }
        }
```

```
private DateTime birthdate;

public DateTime Birthdate
{
    get
    {
        return birthdate;
    }
    set
    {
        if (birthdate == value)
        {
            return;
        }
        birthdate = value;
        OnPropertyChanged();
    }
}

private TeamViewModel team;

public TeamViewModel Team
{
    get
    {
        return team;
    }
    set
    {
        if (team == value)
        {
            return;
```

```
            }
            team = value;
            OnPropertyChanged();
        }
    }

    private double lat;

    public double Lat
    {
        get
        {
            return lat;
        }
        set
        {
            if (lat == value)
            {
                return;
            }
            lat = value;
            OnPropertyChanged();
        }
    }

    private double lon;

    public double Lon
    {
        get
        {
            return lon;
        }
```

```
            set
            {
                if (lon == value)
                {
                    return;
                }
                lon = value;
                OnPropertyChanged();
            }
        }

        public static SurveyViewModel
GetViewModelFromEntity(Survey entity, IEnumerable<Team>
teams)
        {
            var result = new SurveyViewModel
            {
                Id = entity.Id,
                Name = entity.Name,
                Birthdate = entity.Birthdate,
                Team =
TeamViewModel.GetViewModelFromEntity(teams.First(t =>
t.Id == entity.TeamId)),
                Lat = entity.Lat,
                Lon = entity.Lon
            };

            return result;
        }

        public static Survey
GetEntityFromViewModel(SurveyViewModel viewModel)
        {
            var result = new Survey
```

```
                {
                    Id = viewModel.Id,
                    Name = viewModel.Name,
                    Birthdate = viewModel.Birthdate,
                    TeamId = viewModel.Team.Id,
                    Lat = viewModel.Lat,
                    Lon = viewModel.Lon
                };

                return result;
            }
        }
    }
```

MODIFICANDO SURVEYSVIEWMODEL

En la clase SurveysViewModel cambiaremos el tipo de dato de las propiedades Surveys y SelectedSurvey, para que sean de tipo ObservableCollection<SurveyViewModel> y SurveyViewModel, respectivamente.

Adicionalmente, modificaremos el método LoadSurveysAsync() para convertir la colección de objetos de tipo Survey a una colección de objetos de tipo SurveyViewModel, por medio de la invocación del método estático GetViewModelFromEntity(), tal y como se muestra en el siguiente fragmento de código:

```
  private async Task LoadSurveysAsync()
  {
      var allTeams = await
localDbService.GetAllTeamsAsync();
      var allSurveys = await
localDbService.GetAllSurveysAsync();
      if (allSurveys != null)
      {
          Surveys =
              new ObservableCollection<SurveyViewModel>(
                  allSurveys.Select(s =>
SurveyViewModel.GetViewModelFromEntity(s, allTeams)));
```



```
        }
        OnPropertyChanged(nameof(IsEmpty));
    }
```

De igual forma, modificaremos la acción relacionada con el comando DeleteSurveyCommand, específicamente la línea donde se borra la encuesta seleccionada de la base de datos local, para convertir el objeto de tipo SurveyViewModel a un objeto de tipo Survey:

```
    await
localDbService.DeleteSurveyAsync(SurveyViewModel.GetEntit
yFromViewModel(SelectedSurvey));
```

MODIFICANDO SURVEYSVIEW

La página SurveysView deberá también ser modificada en este momento, específicamente la Plantilla de Datos del control ListView que muestra las encuestas capturadas. Esto es debido a que ahora mostraremos también el logo del equipo que eligió la persona encuestada. Otro cambio será eliminar por completo el uso del Convertidor de Valor TeamColorConverter, ya que ahora el color del equipo es parte de los datos que expone el servicio REST.

El siguiente fragmento de código muestra la Plantilla de Datos "SurveyDataTemplate" después de haber realizado las modificaciones correspondientes:

```
<DataTemplate x:Key="SurveyDataTemplate">
  <ViewCell>
    <Grid>
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition />
      </Grid.ColumnDefinitions>

      <Image Source="{Binding Team.Logo}"
             WidthRequest="100"
             HeightRequest="100"
             HorizontalOptions="Start"
             Margin="4" />
```

```
        <StackLayout Grid.Column="1" Margin="10,0,0,0">
          <Label Text="{Binding Name}"
                 FontSize="24" />
          <Label Text="{Binding Birthdate,
StringFormat='{}{0:dd/MM/yyyy}'}" />
          <Label Text="{Binding Team.Name}"
                 TextColor="{Binding Team.Color}" />
          <StackLayout Orientation="Horizontal">
            <Label Text="{Binding Lat,
StringFormat='{}{0:N4}'}" />
            <Label Text="," />
            <Label Text="{Binding Lon,
StringFormat='{}{0:N4}'}" />
          </StackLayout>
        </StackLayout>
      </Grid>
    </ViewCell>
  </DataTemplate>
```

Probando la aplicación

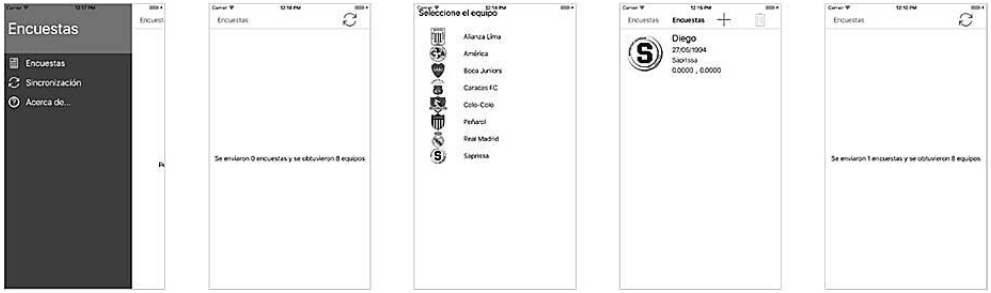
¡Uff! Han sido demasiados cambios los que hemos hecho hasta ahora en el capítulo, pero si todo está implementado correctamente, podremos probar la aplicación para corroborar que:

- La sincronización de datos es requerida inicialmente para bajar el catálogo de equipos
- Las capturas de encuestas siguen siendo almacenadas localmente
- Se muestra una nueva página de selección de equipos
- Se muestra el logotipo del equipo seleccionado en la lista de encuestas
- La sincronización de datos sube las nuevas encuestas y las borra de la base de datos local

Las figuras que a continuación se muestran destacan estas nuevas funcionalidades.

ANDROID

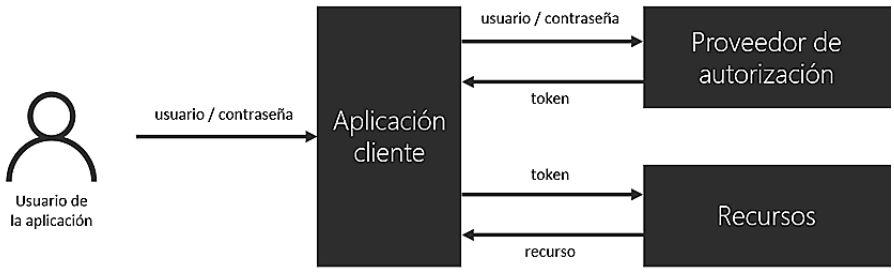
IOS



Protegiendo el servicio con autenticación basada en tokens

A estas alturas, a la aplicación solo le falta agregar la posibilidad de autenticar a los usuarios a través de las credenciales que capturen en la página LoginView. Además, debemos fortalecer la seguridad del lado de los servicios, ya que es posible hacer peticiones y recibir respuestas de ellos sin ningún tipo de seguridad. Por estos motivos, lo último en esta serie de cambios que hemos hecho en la aplicación de encuestas será implementar seguridad en la aplicación a través del mecanismo de autenticación basado en tokens.

El siguiente diagrama muestra el diseño conceptual para este mecanismo:



AGREGANDO LOS PAQUETES DE NUGET

En el proyecto Surveys.Web, agregaremos los siguientes paquetes de NuGet:

- Microsoft.Owin.Security.OAuth
- Microsoft.AspNet.WebApi.Owin

CREANDO LA CLASE APPOAUTHPROVIDER

En el proyecto Surveys.Web, agregaremos un nuevo folder llamado "Providers". En él, implementaremos una nueva clase llamada AppOAuthProvider, la cual hereda de la clase base OAuthAuthorizationServerProvider, disponible en el ensamblado Microsoft.Owin.Security.OAuth. Esta clase tendrá como responsabilidad autenticar el usuario, por medio del nombre de usuario y contraseña que se hayan introducido en la página LoginView. Adicionalmente, esta clase será la responsable de emitir un token en JSON que podrá ser usado por la aplicación de Xamarin.Forms y agregarlo a todas las peticiones subsecuentes.

Por efectos prácticos, en este código solo permitiremos un usuario con las siguientes credenciales:

Username	libro
Password	xamarin.forms

Esta validación la implementaremos en el método GrantResourceOwnerCredentials(). Ten en cuenta que este método es el lugar en donde podrías agregar la lógica para validar la credencial de un usuario en una base de datos de usuarios preexistente, si así fuese necesario.

Si el usuario introduce las credenciales correctas, la clase AppOAuthProvider emitirá el token y agregará un objeto de tipo ClaimsIdentity, en conjunto con una lista de propiedades adicionales relacionadas al usuario que acaba de ser autenticado. En este caso, devolveremos el email pero fácilmente podríamos incluir cualquier otro tipo de dato necesario para la aplicación.

El siguiente código muestra la implementación completa de la clase AppOAuthProvider:

```
  public class AppOAuthProvider :
OAuthAuthorizationServerProvider
  {
      private readonly string publicClientId;

      public AppOAuthProvider(string publicClientId)
```



```
        {
            if (publicClientId == null)
            {
                throw new
ArgumentNullException(nameof(publicClientId), "El
identificador no es válido");
            }

            this.publicClientId = publicClientId;
        }

        public override Task
GrantResourceOwnerCredentials(OAuthGrantResourceOwnerCred
entialsContext context)
        {

context.OwinContext.Response.Headers.Add("Access-Control-
Allow-Origin", new[] { "*" });

            //Aquí va el login REAL

            if (context.UserName == "libro" &&
context.Password == "xamarin.forms")
            {
                //Crea y prepara el objeto ClaimsIdentity
                var identity = new
ClaimsIdentity(OAuthDefaults.AuthenticationType);

                identity.AddClaim(new
Claim(ClaimTypes.Name, "Rodrigo Díaz Concha"));

                var data = new Dictionary<string, string> {
{ "email", "rodrigo@rdiazconcha.com" } };
                var properties = new
AuthenticationProperties(data);

                //Crea un AuthenticationTicket con la
identidad y las propiedades
```

```
                var ticket = new
AuthenticationTicket(identity, properties);

                //Valid y autentica
                context.Validated(ticket);

                return Task.FromResult(true);
            }
            else
            {
                context.SetError("access_denied", "Acceso
denegado");
                return Task.FromResult(false);
            }
        }
        public override async Task
ValidateClientAuthentication(OAuthValidateClientAuthentic
ationContext context)
        {
            await Task.Run(() => context.Validated());
        }

        public override Task
TokenEndpoint(OAuthTokenEndpointContext context)
        {
            foreach (KeyValuePair<string, string> property
in context.Properties.Dictionary)
            {

context.AdditionalResponseParameters.Add(property.Key,
property.Value);
            }

            return Task.FromResult<object>(null);
        }
    }
```


CREANDO LA CLASE OAUTHCONFIG

Una vez implementado el proveedor de tokens de seguridad, crearemos una nueva clase que tenga la responsabilidad de configurar el endpoint que la aplicación cliente pueda invocar para autenticar al usuario. Esta clase la agregaremos en el folder existente "App_Start" del proyecto Surveys.Web y le llamaremos OAuthConfig.

En esta nueva clase, agregaremos un método llamado ConfigureOAuth(), el cual tendrá como responsabilidad el configurar los endpoints /token y /auth. Asimismo, en este método indicaremos que el proveedor que deseamos utilizar es justamente AppOAuthProvider, así como el tiempo de expiración para el token, el cual será de 1 día.

El siguiente código muestra la implementación completa de la clase OAuthConfig:

```
public class OAuthConfig
{
    public static string PublicClientId { get; }
    public static OAuthAuthorizationServerOptions
OAuthOptions { get; private set; }
    public static OAuthBearerAuthenticationOptions
OAuthBearerOptions { get; private set; }

    static OAuthConfig()
    {
        PublicClientId = "LibroXamarinForms";
    }
    public static void ConfigureOAuth(IAppBuilder app,
HttpConfiguration config)
    {
        OAuthOptions = new
OAuthAuthorizationServerOptions
        {
            TokenEndpointPath = new
PathString("/token"),
            AuthorizeEndpointPath = new
PathString("/auth"),
```

```
                Provider = new
AppOAuthProvider(PublicClientId),
                AccessTokenExpireTimeSpan =
TimeSpan.FromDays(1),
                AllowInsecureHttp = true
            };

            OAuthBearerOptions = new
OAuthBearerAuthenticationOptions
            {
                AuthenticationMode =
Microsoft.Owin.Security.AuthenticationMode.Active,
                Realm = "LibroXamarinForms"
            };

            app.UseOAuthAuthorizationServer(OAuthOptions);

app.UseOAuthBearerAuthentication(OAuthBearerOptions);
        }
    }
```

AGREGANDO LA CLASE STARTUP

En el proyecto Surveys.Web, actualmente usamos la clase Global.asax. Este es el mecanismo tradicional en ASP.NET para manejar las variables globales en la aplicación Web. Sin embargo, ya que estamos usando OWIN y sus estándares, usaremos una nueva clase llamada Startup.cs para inicializar la aplicación ya que las versiones más recientes de ASP.NET promueven el uso de inyección de dependencias.

En la clase Startup, agregaremos un solo método llamado Configuration(), el cual tendrá como argumento un objeto de tipo IAppBuilder. Será en este método el lugar en donde invocaremos el método estático ConfigureOAuth() de la clase OAuthConfig que acabamos de implementar, así como también el método Register() de la clase WebApiConfig existente.

El siguiente código muestra la implementación completa de la clase Startup:


```
public class Startup
{
    public void Configuration(IAppBuilder app)
    {
        var config = new HttpConfiguration();
        OAuthConfig.ConfigureOAuth(app, config);

        app.UseWebApi(config);

        WebApiConfig.Register(config);
    }
}
```

MODIFICANDO LOS CONTROLADORES

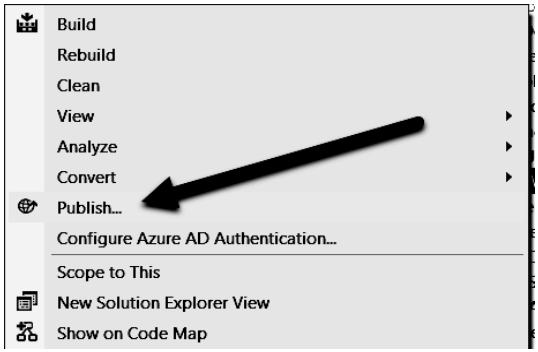
Como último paso en la implementación de seguridad en nuestro servicio REST, modificaremos los controladores SurveysController y TeamsController, para decorar todos los métodos que deseemos proteger con el atributo [Authorize]. El hacer esto impedirá que un cliente sin token JSON de seguridad sea capaz de obtener una respuesta por parte de los diferentes endpoints del servicio REST.

El siguiente fragmento de código muestra el uso de este atributo en el método Get() que nos devuelve toda la lista de encuestas capturadas:

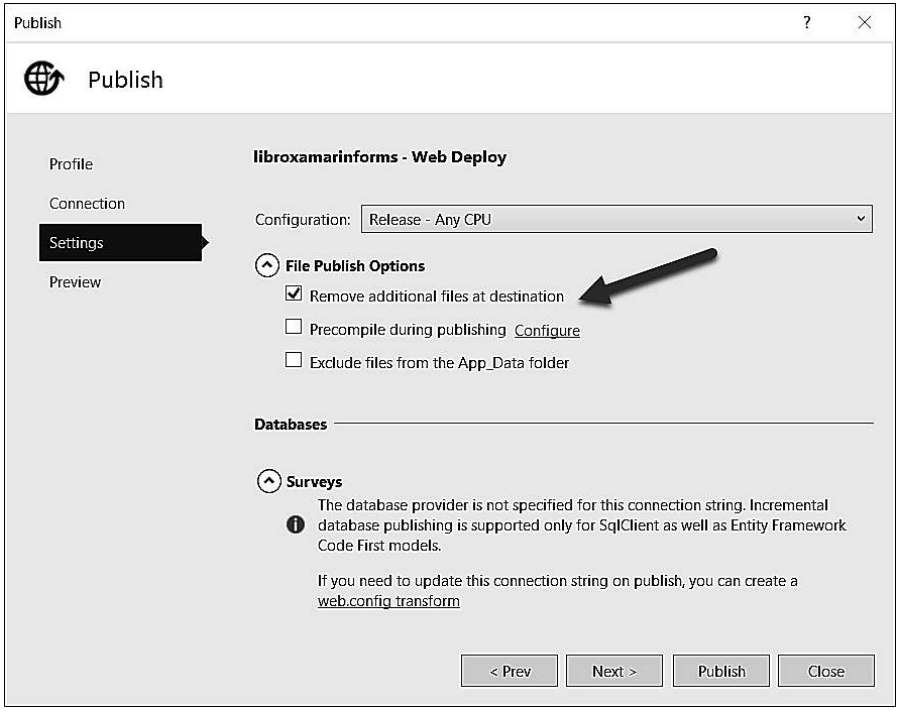
```
[Authorize]
public async Task<IEnumerable<Survey>> Get()
{
    var allSurveys = await
surveysProvider.GetAllSurveysAsync();
    return allSurveys;
}
```

PUBLICANDO EL SERVICIO CON LAS MODIFICACIONES

Una vez implementada la autenticación basada en tokens, publicaremos nuevamente el servicio en Microsoft Azure, haciendo uso de la opción “Publish…” que aparece al hacer clic secundario en el proyecto Surveys.Web:

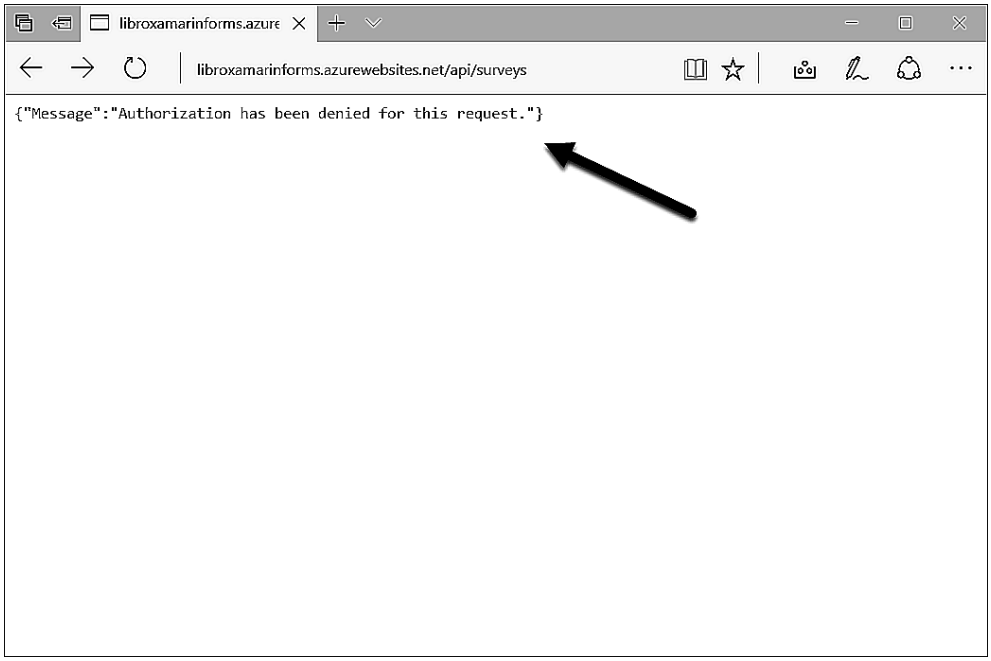


En la sección de “Settings” de la caja de diálogo de publicación, seleccionaremos la opción “Remove additional files at destination”, para asegurarnos de borrar todos los archivos que previamente publicamos en el servicio de Microsoft Azure, tal y como lo demuestra la siguiente figura:



Una vez seleccionada la opción de remover los archivos anteriores, haremos clic en el botón “Publish” para permitir que Visual Studio .NET publique todos los cambios en Microsoft Azure.

Al finalizar la publicación del servicio, podremos observar que ahora ya no es posible recibir respuestas del servicio de forma anónima, ya que requerimos el token JSON de seguridad. La siguiente figura muestra el resultado de una petición a /api/surveys hecha a través del navegador:



MODIFICANDO IWEBAPISERVICE

Ya que ahora las peticiones que hagamos a los endpoints del servicio REST requieren un token JSON válido, necesitamos modificar la aplicación cliente para poder enviar las credenciales al servicio, específicamente al endpoint /token que previamente configuramos. Si la autenticación fue exitosa, /token nos devolverá un token en formato JSON. El token que obtengamos lo usaremos en las peticiones subsecuentes.

Creando la clase TokenModelResponse

Para hacer más fácil la utilización del token que nos devuelva el endpoint /token, deserializaremos ese dato JSON en un objeto de tipo TokenModelResponse. Debido a esto, primero agregaremos la clase TokenModelResponse en el folder “Models” del proyecto PCL. Esta clase incluye todos los miembros que estructuran un token JSON:

```
public class TokenResponseModel
{
    [JsonProperty("access_token")]
    public string AccessToken { get; set; }

    [JsonProperty("token_type")]
    public string TokenType { get; set; }

    [JsonProperty("expires_in")]
    public int ExpiresIn { get; set; }

    [JsonProperty(".issued")]
    public string IssuedAt { get; set; }

    [JsonProperty(".expires")]
    public string ExpiresAt { get; set; }

    [JsonProperty("username")]
    public string Username { get; set; }
}
```

Agregando el método LoginAsync()

Modificaremos la interfaz IWebApiService para agregar un nuevo método llamado LoginAsync(), que reciba como parámetros el nombre de usuario y la contraseña. El siguiente fragmento de código muestra la declaración de este nuevo método que tendrá la interfaz:

```
Task<bool> LoginAsync(string username, string
password);
```

En la implementación concreta de este método en la clase WebApiService, lo primero que haremos será codificar ambos nombre de usuario y la contraseña para escapar todos los caracteres especiales y que sea compatible con un URI. Para ello, usaremos el método UrlEncode de la clase WebUtility, tal y como se muestra en el siguiente fragmento de código:

```
public async Task<bool> LoginAsync(string username,
string password)
{
```



```
        var encodedUsername =
WebUtility.UrlEncode(username);
        var encodedPassword =
WebUtility.UrlEncode(password);
        …
    }
```

Posteriormente, armaremos todo el contenido en cadena que enviaremos al endpoint /token. Este contenido indicará que el tipo de grant_type está basado en un password. Asimismo, enviaremos ambos nombre de usuario y la contraseña e indicaremos que el tipo de contenido a enviar es "application/x-www-form-urlencoded".

El siguiente fragmento de código muestra la creación de este contenido:

```
    var content = new
StringContent($"grant_type=password&username={encodedUser
name}&password={encodedPassword}", Encoding.UTF8,
"application/x-www-form-urlencoded");
```

Una vez creado el objeto con el contenido, lo enviaremos en formato cadena al endpoint /token de nuestro servicio REST a través del método PostAsync() del objeto HttpClient. La respuesta que obtengamos la guardaremos en una variable llamada "value", tal y como lo demuestra el siguiente fragmento de código:

```
    var uri = new
Uri($"{Literals.WebApiServiceBaseAddress}Token");
    using (var response = client.PostAsync(uri.ToString(),
content).Result)
    {
        var value = await
response.Content.ReadAsStringAsync();
        …
    }
```

Acto seguido, verificaremos si la respuesta de la petición tiene un código de estatus exitoso. Si este es el caso, deserializaremos el token JSON en un objeto de tipo TokenResponseModel. Esto lo haremos fácilmente gracias a la clase JsonConvert. En este objeto de tipo TokenResponseModel tendremos todas las propiedades pertenecientes al token que nos devolvió el servicio. La cadena del token la agregaremos al encabezado "Authorization" del objeto HttpClient, para que todas las peticiones subsecuentes lo incluyan, y por lo tanto del lado del servicio se

permita el acceso a los diferentes endpoints de encuestas y equipos (los métodos marcados con el atributo [Authorize] en los controladores).

El siguiente fragmento de código muestra esta implementación:

```
if (response.IsSuccessStatusCode)
{
    var token =
JsonConvert.DeserializeObject<TokenResponseModel>(value);

    var tokenString = token.AccessToken;

    if
(!client.DefaultRequestHeaders.Contains("Authorization"))
    {

client.DefaultRequestHeaders.Add("Authorization", "Bearer
" + tokenString);
    }
    return true;
}
```

MODIFICANDO LA PÁGINA DE LOGIN

Modificando LoginViewModel

Con el método LoginAsync() disponible ahora en IWebApiService, modificaremos la clase LoginViewModel para poder invocar dicho método. Primero, solicitaremos como argumento IWebApiService y también IPageDialogService en el constructor de la clase. Después, agregaremos una nueva propiedad de tipo bool llamada "IsBusy", para notificar a la Interfaz de Usuario si la aplicación está ocupada en el proceso de autenticación. Por último, modificaremos la acción relacionada al comando LoginCommand para invocar el método LoginAsync() del objeto IWebApiService. Si el login es correcto, navegaremos por medio de Deep Linking a "MainView/RootNavigationView/SurveysView" como regularmente lo hacemos, o en caso contrario, mostraremos una caja de diálogo con el error. Es importante destacar que para la navegación, ahora usaremos un URI absoluto en vez de uno relativo. Esto indicará a Prism que deberá restablecer la pila de navegación, ya que la intención es no poder navegar de regreso a la página LoginView una vez autenticado el usuario.

El siguiente fragmento de código muestra las modificaciones realizadas en el método LoginCommandExecute():

```
  private async void LoginCommandExecute()
  {
      IsBusy = true;

      try
      {
          var loginResult = await
webApiService.LoginAsync(Username, Password);

          if (loginResult)
          {
              await navigationService.NavigateAsync(

$"app:///{nameof(MainView)}/{nameof(RootNavigationView)}/
{nameof(SurveysView)}");
          }
          else
          {
              await
pageDialogService.DisplayAlertAsync(Literals.LoginTitle,
Literals.AccessDenied, Literals.Ok);
          }
      }
      catch (Exception e)
      {
          await
pageDialogService.DisplayAlertAsync(Literals.LoginTitle,
e.Message, Literals.Ok);
      }

      IsBusy = false;
  }
```

Nota: Recuerda agregar las cadenas constantes en la clase Literals del proyecto PCL.

Modificando LoginView

En la página LoginView, agregaremos un ActivityIndicator, enlazado a la propiedad IsBusy del ViewModel. Además, crearemos un estilo con un Trigger que oculte el StackLayout cuando la propiedad IsBusy sea verdadera.

El siguiente código de XAML muestra la implementación completa de la página LoginView:

```
  <?xml version="1.0" encoding="utf-8" ?>
  <ContentPage
xmlns="http://xamarin.com/schemas/2014/forms"

xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
               x:Class="Surveys.Core.Views.LoginView">
    <Grid>
      <StackLayout VerticalOptions="Center"
                   Margin="10">
        <StackLayout.Style>
          <Style TargetType="StackLayout">
            <Style.Triggers>
              <DataTrigger TargetType="StackLayout"
                           Binding="{Binding IsBusy}"
                           Value="True">
                <Setter Property="IsVisible"
Value="False" />
              </DataTrigger>
            </Style.Triggers>
          </Style>
        </StackLayout.Style>
        <Label Text="Nombre de usuario" />
        <Entry Text="{Binding Username, Mode=TwoWay}" />
```

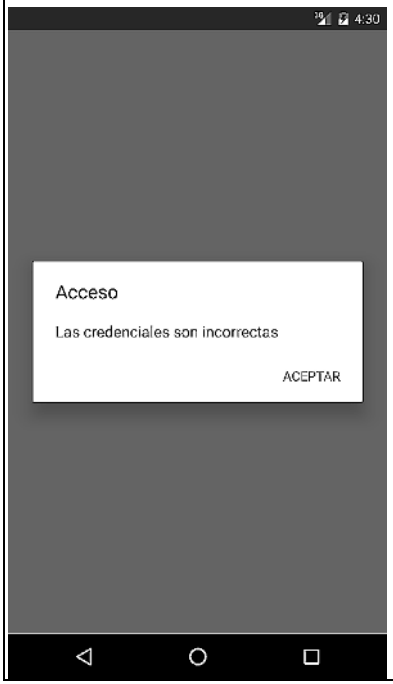


```
        <Label Text="Contraseña" />
        <Entry Text="{Binding Password, Mode=TwoWay}"
               IsPassword="True" />
        <Button Text="Login"
                Command="{Binding LoginCommand}" />
      </StackLayout>

      <ActivityIndicator IsRunning="{Binding IsBusy}"
                         IsVisible="{Binding IsBusy}"
                         HorizontalOptions="Center"
                         VerticalOptions="Center" />
    </Grid>
  </ContentPage>
```

Probando la aplicación

Si llegaste a este punto, ¡felicidades! En este momento tenemos ahora una aplicación de encuestas que es capaz de comunicarse a un servicio REST construido con ASP.NET Web API, protegido, que requiere un token JSON válido para poder usar sus diferentes endpoints, publicado en Windows Azure.

Al ejecutar la aplicación, podrás observar que es absolutamente necesario introducir la credencial “libro” / “xamarin.forms”, de lo contrario la aplicación mostrará una caja de diálogo con un mensaje de error.

Android	iOS
	

ÍNDICE ANALÍTICO

N

O

P

R

S

T

U

V

W

X