

XAMARIN.FORMS

SUCCINCTLY

BY ALESSANDRO
DEL SOLE

Xamarin.Forms

Sucintamente

Por Alessandro

Del Sole

Prólogo de Daniel Jebaraj



Copyright © 2017 por Syncfusion, Inc.

2501 Centro aérea Parkway

suite 200

Morrisville, NC 27560

Estados Unidos

Todos los derechos reservados.

Información adicional de licencia. Por favor lee.

Este libro está disponible para su descarga gratuita desde www.syncfusion.com al término de una formulario de inscripción.

Si ha obtenido este libro de cualquier otra fuente, por favor registrarse y descargar una copia gratuita de www.syncfusion.com.

Este libro tiene licencia para sólo lectura si se obtiene de www.syncfusion.com.

Este libro se licencia exclusivamente para uso personal o educativo.

Está prohibida la redistribución en cualquier forma.

Los autores y los titulares de derechos de autor proporcionan absolutamente ninguna garantía de ninguna información proporcionada.

Los autores y los titulares de derechos de autor no se hace responsable de cualquier reclamación, daños y perjuicios, o cualquier otra responsabilidad derivada de, fuera de, o en relación con la información de este libro.

Por favor, no use este libro si los términos que se enumeran son inaceptables.

Uso constituirá la aceptación de los términos mencionados.

Syncfusion, de manera sucinta, ofreciendo innovación con facilidad, esencial y .NET

Esenciales son marcas comerciales registradas de Syncfusion, Inc.

Crítico técnica: James McCaffrey

Editor de copia: Jacqueline Bieringer, productor de contenidos, Syncfusion, Inc.

Coordinador de adquisiciones: Tres Watkins, gerente de desarrollo de contenidos, Syncfusion, Inc.

Corrector de pruebas: Graham alta, productor de contenidos de alto nivel, Syncfusion, Inc.

Tabla de contenido

La historia detrás de la serie de libros Sucintamente	8
Sobre el Autor	10
Introducción	11
Capítulo 1 Introducción a Xamarin.Forms	12
La introducción de Xamarin y Xamarin.Forms	12
Configuración del entorno de desarrollo	13
Configuración de un Mac	14
La creación de soluciones Xamarin.Forms	15
La biblioteca Xamarin.Forms	18
El proyecto Xamarin.Android	18
El proyecto Xamarin.iOS	20
El proyecto Plataforma Windows universal	24
Depuración y aplicaciones de pruebas a nivel local	25
Configurar el simulador de iOS	27
Ejecución de aplicaciones en dispositivos físicos	27
La aplicación Xamarin jugador vivo	28
Análisis y perfiles de aplicaciones	28
Resumen del capítulo	29
Capítulo 2 Código intercambio de información entre plataformas	30
Introducción a las estrategias de código compartido	30
Compartir código con bibliotecas de clases portátiles	30
Compartir código con proyectos compartidos	31
Compartir código con bibliotecas .NET Standard	34
Resumen del capítulo	37

Capítulo 3 Creación de la interfaz de usuario con XAML	38
La estructura de la interfaz de usuario en Xamarin.Forms	38
Codificación de la interfaz de usuario en C #	39
La forma moderna: el diseño de la interfaz de usuario con XAML	40
En respuesta a eventos	42
convertidores de tipos comprensión	44
Xamarin.Forms vista previa de	45
Consejos para XAML estándar	0.46
Resumen del capítulo	46
Capítulo 4 La organización de la interfaz de usuario con formatos	47
Comprender el concepto de diseño	47
Alineación y espaciado opciones	48
los StackLayout	49
los Cuadrícula	51
Espaciado y proporciones para las filas y columnas	53
La introducción de tramos	54
los AbsoluteLayout	54
los Disposición relativa	56
los ScrollView	57
los Marco	58
los ContentView	59
Resumen del capítulo	61
Capítulo 5 Xamarin.Forms controles comunes	62
Comprender el concepto de vista	62
Vistas propiedades comunes	62
La introducción de controles comunes	63

La entrada del usuario con el Botón	63
Trabajar con el texto: Etiqueta, de entrada, Editor	64
Trabajar con fechas y horas: Selector de fechas y TimePicker	sesenta y cinco
Viendo el contenido HTML con WebView	67
La implementación de selección de valor: Switch, Slider, paso a paso	68
Presentación de la Barra de búsqueda	70
operaciones de larga duración: ActivityIndicator y Barra de progreso	71
Trabajar con imágenes	72
La introducción de los reconocedores gesto	74
Viendo alertas	75
Resumen del capítulo	76
Capítulo 6 Páginas y Navegación	77
La introducción y la creación de páginas	77
vistas individuales con el Pagina de contenido	78
La división de contenidos con el MasterDetailPage	78
Viendo el contenido dentro de las pestañas con el TabbedPage	80
páginas con la swiping CarouselPage	81
La navegación entre páginas	83
objetos que pasan por las páginas	84
Animación de transiciones entre páginas	85
La gestión del ciclo de vida página	85
La manipulación del botón de hardware volver	85
Resumen del capítulo	86
Capítulo 7 Recursos y enlace de datos	87
Trabajar con recursos	87
recursos que declaran	87

La introducción de estilos	88
Trabajar con el enlace de datos	90
Trabajando con colecciones y con la Vista de la lista	93
La introducción de Model-View-ViewModel	103
Resumen del capítulo	108
Capítulo 8 Acceso a las API específicas de la plataforma	109
los Dispositivo clase y la OnPlatform Método	109
Trabajar con el servicio de dependencia	111
Trabajando con los plugins	113
Trabajar con vistas nativos	115 ..
Incrustación de vistas nativos en XAML	116
Trabajar con procesadores personalizados	117
Consejos para los efectos	120
Resumen del capítulo	120
Capítulo 9 Administración del ciclo de vida de la aplicación	121
Presentación de la Aplicación clase	121
La gestión del ciclo de vida de aplicaciones	121
Enviar y recibir mensajes	124
Resumen del capítulo	125
Apéndice: Recursos útiles	126
Trabajar con bases de datos SQLite	126
El consumo de servicios web y servicios en la nube	126
La publicación de aplicaciones	126
Los ejemplos de código y kits de iniciación	126
Actualización de herramientas Xamarin	127

La historia detrás de la *Sucintamente Serie* de libros

Daniel Jebaraj, vicepresidente
Syncfusion, Inc.

STaying en el borde de corte

Como muchos de ustedes saben, Syncfusion es un proveedor de componentes de software para la plataforma Microsoft. Esto nos pone en la posición apasionante pero difícil de estar siempre a la vanguardia.

Siempre que las plataformas o herramientas están enviando fuera de Microsoft, que parece estar a punto cada dos semanas en estos días, tenemos que educarnos, de forma rápida.

La información es abundante, pero más difícil de digerir

En realidad, esto se traduce en una gran cantidad de pedidos de libros, búsquedas de blogs, Twitter y las exploraciones.

Mientras más información se está convirtiendo disponible en Internet y cada vez más se están publicando libros, incluso en temas que son relativamente nuevos, un aspecto que nos sigue es inhibir la imposibilidad de encontrar tecnología concisa los libros de resumen.

Nos enfrentamos por lo general con dos opciones: leer varios libros de 500 páginas, o buscar en la web para las entradas del blog pertinentes y otros artículos. Al igual que todos los demás que tiene un trabajo que hacer y servir a los clientes, nos encontramos con esta bastante frustrante.

los *Sucintamente serie*

Esta frustración se tradujo en un profundo deseo de producir una serie de libros técnicos concisos que se dirigirían a los desarrolladores que trabajan en la plataforma Microsoft.

Creemos firmemente, teniendo en cuenta los conocimientos básicos tales desarrolladores tienen, que la mayoría de los temas se pueden traducir en libros que son entre 50 y 100 páginas.

Esto es exactamente lo que nos dispusimos a lograr con el *Sucintamente serie*. No es todo lo maravilloso nace de un profundo deseo de cambiar las cosas para mejor?

Los mejores autores, los mejores contenidos

Cada autor se ha escogido de un grupo de expertos con talento que comparten nuestra visión. El libro que tiene en sus manos, y los otros disponibles en esta serie, son el resultado del trabajo incansable de los autores. Va a encontrar el contenido original que está garantizado para que pueda ponerse en funcionamiento en aproximadamente el tiempo que se tarda en tomar unas tazas de café.

Siempre libre

Syncfusion va a trabajar para producir libros sobre varios temas. Los libros siempre serán libres. Cualquier actualización que publicamos también serán libres.

¿Gratis? ¿Cuál es la trampa?

No hay truco aquí. Syncfusion tiene un interés personal en este esfuerzo.

Como proveedor de componentes, nuestra pretensión única siempre ha sido que ofrecemos marcos más amplios y profundos que cualquier otro en el mercado. Formación de desarrolladores en gran medida nos ayuda a comercializar y vender en contra de proveedores de la competencia que prometen "habilitar el soporte de AJAX con un solo clic," o "pasar la luna de queso!"

Háganos saber lo que piensas

Si usted tiene cualquiera de los temas de interés, pensamientos o comentarios, por favor enviarlos a nosotros en succinctly-series@syncfusion.com.

Esperamos sinceramente que disfrute de este libro y que le ayuda a comprender mejor el tema de estudio. Gracias por leer.

Por favor, siga con nosotros en Twitter y "Me gusta" en la cara-libro para ayudar a difundir la
palabra sobre el *Sucintamente* ¡serie!



Sobre el Autor

Alessandro Del Sole es un desarrollador certificado Xamarin móvil y ha sido un MVP de Microsoft desde 2008. Otorgado MVP del Año en 2009, 2010, 2011, 2012, y 2014, es considerado internacionalmente un experto en Visual Studio .NET y una autoridad.

Alessandro ha sido autor de muchos libros impresos y libros electrónicos sobre la programación con Visual Studio, que incluye [Visual Studio 2017 Sucintamente](#), [Visual Basic 2015 Unleashed](#), y [Visual Studio Código Sucintamente](#). Ha escrito toneladas de artículos técnicos sobre .NET, Visual Studio y otras tecnologías de Microsoft en italiano y en Inglés para muchos portales de desarrollo, incluyendo MSDN Magazine y el Centro de desarrolladores de Visual Basic de Microsoft. Es un orador frecuente en conferencias italianos, y él ha dado a conocer una serie de aplicaciones de Windows Store. También ha producido una serie de videos de instrucción, tanto en Inglés e italiano. Alessandro funciona como mayor desarrollador .NET, formador y consultor. Lo puedes seguir en Twitter en [@progalex](#).

Introducción

Para los desarrolladores de aplicaciones móviles y las empresas que desean estar representados en el mercado de las aplicaciones móviles, la necesidad de publicar Android, versiones de iOS y Windows de aplicaciones se ha incrementado dramáticamente en los últimos años. Para las empresas que siempre han trabajado con plataformas nativas y herramientas de desarrollo, esto podría no ser un problema. Es un problema, sin embargo, para las empresas que siempre han construido con el software .NET, C #, y, más en general, con la pila de Microsoft. Por tanto, una empresa podría contratar a los desarrolladores especializados o esperar a los desarrolladores para alcanzar las habilidades necesarias y los conocimientos existentes para trabajar con plataformas nativas, pero, en ambos casos, hay enormes costos y riesgos con el tiempo. La solución ideal es que los desarrolladores pueden reutilizar sus habilidades .NET y C # existentes para construir aplicaciones móviles nativas. Aquí es donde entra en juego Xamarin.

En este e-libro, usted aprenderá cómo Xamarin.Forms permite el desarrollo multiplataforma, que permite crear aplicaciones móviles para Android, iOS y Windows desde una única base de código C #, y por lo tanto volver a utilizar sus habilidades .NET existentes. Usted aprenderá cómo se hacen las soluciones Xamarin.Forms, lo que hace que sea posible compartir código, cómo crear la interfaz de usuario, la forma de organizar los controles dentro de los contenedores, y cómo implementar la navegación entre páginas. También aprovechar técnicas avanzadas como el enlace de datos y el acceso a las API nativas de código de plataforma cruzada.

Vale la pena mencionar que Xamarin.Forms también es compatible con el lenguaje de programación # F, C #, pero obviamente la opción más común y se le proporcionará por lo tanto todas las explicaciones y ejemplos basados en C #. También vale la pena mencionar que, en el pasado, Xamarin.Forms compatibles de Windows 8.x de teléfono y 8.x de Windows como plataformas de destino, y el apoyo para la plataforma Windows Universal de Windows 10 se añadió recientemente. En Visual Studio 2017, Xamarin.Forms sólo es compatible con uwp para el desarrollo de Windows. Por esta razón, cuando me refiero a Windows a partir de ahora, me refiero a Windows 10 y la plataforma Windows universal, no las versiones anteriores.

En este libro electrónico, asumiré que tiene al menos un conocimiento básico de C # y el IDE de Visual Studio. También se sugiere que el funcionario marcarlo [documentación Xamarin](#) portal de referencia rápida. El código fuente de este libro electrónico está disponible en [bitbucket](#). Los nombres de archivo son fáciles de entender para que sea más fácil para que usted siga los ejemplos, especialmente para los capítulos 4, 5 y 6. Antes de empezar a escribir código, es necesario configurar el entorno de desarrollo. Este es el tema del primer capítulo.

Capítulo 1 Introducción a Xamarin.Forms

Antes de empezar a escribir aplicaciones para dispositivos móviles con Xamarin.Forms, primero tiene que entender el estado de desarrollo de aplicaciones móviles de hoy y cómo Xamarin cabe en él. Además, es necesario configurar el entorno de desarrollo para poder construir, probar, depurar y desplegar sus aplicaciones a los dispositivos Android, iOS y Windows. Este capítulo presenta Xamarin como un conjunto de herramientas y servicios, Xamarin.Forms como la plataforma que va a utilizar y, a continuación se presentan las herramientas y hardware necesarios para el desarrollo del mundo real.

La introducción de Xamarin y Xamarin.Forms

Xamarin es el nombre de una compañía que Microsoft adquirió en 2016 y, al mismo tiempo, el nombre de un conjunto de herramientas y servicios de desarrollo que los desarrolladores pueden utilizar para crear aplicaciones nativas para iOS, Android y Windows en C#. El objetivo principal de Xamarin es hacer que sea más fácil para los desarrolladores de .NET para crear aplicaciones nativas para Android, iOS y Windows reutilizar sus habilidades existentes. La razón detrás de este objetivo es simple: la creación de aplicaciones para Android que requiere conocer Java y Android Studio o Eclipse; la creación de aplicaciones para iOS se requiere saber de Objective-C o Swift y Xcode; la creación de aplicaciones para Windows requiere que sepas C# y Visual Studio. Como ya existente

.NET desarrollador, ya sea que tenga experiencia o un principiante, conocer todas las posibles plataformas, lenguajes y entornos de desarrollo es extremadamente difícil y los costes son muy altos para esto.

Xamarin le permite construir aplicaciones nativas con C#, basados en una multi-plataforma, portabilidad de código abierto de .NET Framework llama [Mono](#). Desde el punto de vista del desarrollo, Xamarin ofrece una serie de sabores: Xamarin.iOS y Xamarin.Mac, bibliotecas que las API de Apple envoltura nativos se pueden utilizar para construir aplicaciones para iOS y MacOS usando C# y Visual Studio; Xamarin.Android, una biblioteca que envuelve APIs Java y Google nativas se pueden utilizar para crear aplicaciones para Android usando C# y Visual Studio; Xamarin.Forms, una biblioteca de código abierto que le permite compartir código a través de plataformas y construir aplicaciones que se ejecutan en Android, iOS y Windows desde una única base de código C#. El mayor beneficio de Xamarin.Forms es que se escribe código una vez y se ejecutará en todas las plataformas soportadas, sin costo adicional. Como veremos a lo largo de este libro electrónico, Xamarin.Forms consiste en una capa que envuelve los objetos comunes a todas las plataformas soportadas en objetos de C#. Accediendo nativa, es específica de una plataforma objetos y API posible de varias maneras, todas discutido en los capítulos siguientes, pero requiere un poco de trabajo extra. Además, Xamarin se integra con el IDE de Visual Studio en Windows y es parte del Visual Studio recién lanzado para Mac, por lo que no sólo puede crear soluciones multiplataforma, sino también escribir código en diferentes sistemas.

La oferta también incluye Xamarin [Universidad Xamarin](#), Un servicio de pago que le permite asistir a clases en vivo en línea y ver videos de instrucción que le ayudarán a prepararse para obtener el desarrollador insignia Xamarin Certificado móvil. También incluye la [Test Cloud Xamarin](#) servicio para la automatización de pruebas. Microsoft también ha comenzado recientemente la [Visual Studio Center Mobile](#), Una solución de nube completa para la gestión del ciclo de vida de aplicaciones de automatización de construcción para la integración continua, pruebas, análisis y mucho más (nota que Internet Explorer no es compatible). Este libro se centra en Xamarin.Forms y se dirige a Visual Studio 2017 en Windows 10, pero todos los medios técnicos

conceptos se aplican a Visual Studio para Mac también. El autor de este libro electrónico también ha registrado una [serie de video para Syncfusion](#) que proporciona una visión general de lo que ofrece Xamarin y de Xamarin.iOS y Xamarin.Android, así como Xamarin.Forms.

Configuración del entorno de desarrollo

Con el fin de crear aplicaciones móviles nativas con Xamarin.Forms, necesita Windows 10 como sistema operativo y **Microsoft Visual Studio 2017 como su entorno de desarrollo. Puede descargar e instalar el [Visual Studio 2017 Comunidad](#) edición gratis y recibe todas las herramientas necesarias para el desarrollo Xamarin.** Cuando se inicia la instalación, tendrá que seleccionar el **desarrollo móvil con .NET** carga de trabajo en el instalador de Visual Studio (véase la figura

1).

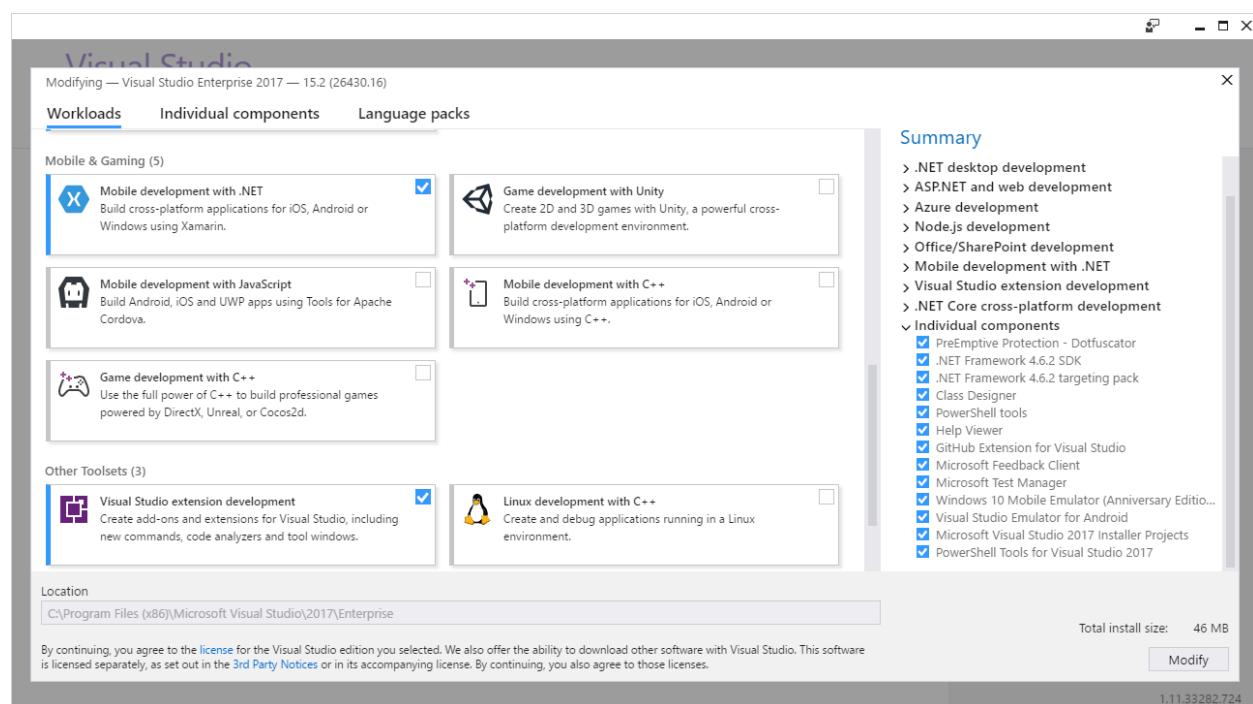


Figura 1: Instalación de herramientas de desarrollo Xamarin

Al seleccionar esta carga de trabajo, el instalador de Visual Studio se encargará de descargar e instalar todas las herramientas necesarias para construir aplicaciones para Android, iOS y Windows. iOS en realidad requiere una configuración adicional se describe en la siguiente sección. Además, para el desarrollo de Windows 10, necesita herramientas y SDK adicionales, que se obtiene al seleccionar también la

Desarrollo Plataforma Windows universal carga de trabajo. Si selecciona la **Componentes individuales** ficha, tendrá una opción para comprobar si Android y Windows emuladores se han seleccionado o para hacer una selección manual (ver Figura 2).

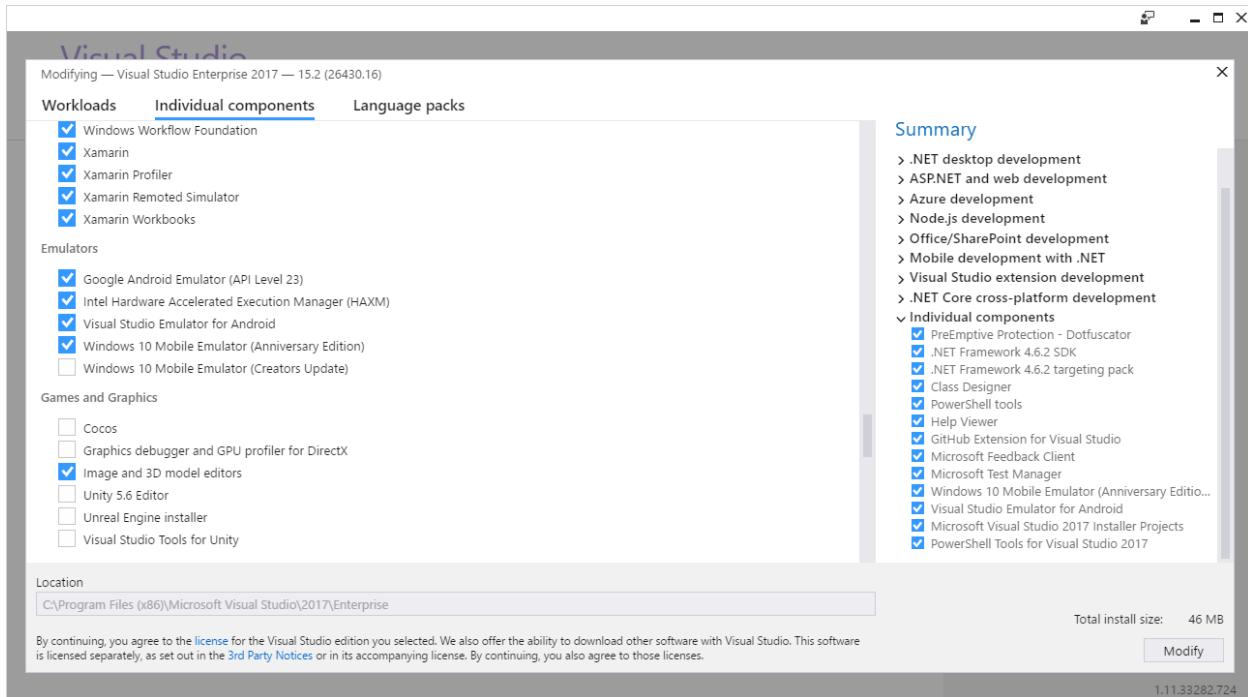


Figura 2: Selección de emuladores

Si usted piensa que va a utilizar Visual Studio 2017 y Visual Studio para Mac, sugiero instalar el emulador de Google, que tiene una apariencia idéntica y el comportamiento en ambos sistemas. Si sólo trabaja en Windows, una buena opción es el emulador de Visual Studio para Android de Microsoft. Este emulador requiere Hyper-V para funcionar, ya que es una máquina virtual completa. Los planes de Microsoft para apoyar este emulador en el futuro son aún incierto, por lo que mi sugerencia es que se trata de usar hasta que ya no está disponible, ya que es más rápido y más potente. Para el emulador de Windows 10, mi sugerencia es para descargar la versión más antigua (en la figura 2, es el aniversario de actualización) de manera que puede orientar versiones anteriores de Windows 10 también. Seguir adelante con la instalación y esperar a que se complete.

Configuración de un Mac

las políticas de Apple establecen que un ordenador Mac se requiere para construir una aplicación. Esto se debe a que sólo el entorno de desarrollo Xcode de Apple y el SDK están autorizados para ejecutar el proceso de generación. Para la depuración y pruebas locales, **Xamarin ha introducido recientemente el [Xamarin jugador vivo](#) aplicación, que se puede descargar e instalar en su dispositivo Android o iOS y vincularse con Visual Studio para propósitos de depuración.** se le proporcionarán más detalles sobre esta aplicación en breve. Sin embargo, se hace insuficiente para el desarrollo serio. Todavía es necesario un ordenador Mac para la firma de código, creación de perfiles, y la publicación de una aplicación a la App Store. Puede usar un Mac local en su red, lo que también permite depurar y aplicaciones de prueba en un dispositivo físico o un Mac remoto. En ambos casos, MacOS deben configurarse con los siguientes requisitos de software:

- macOS “El Capitán” (10.11) o superior.
- Xcode de Apple y el SDK, que puede obtener de la App Store de forma gratuita.
- El motor Xamarin.iOS. La manera más fácil de conseguir Xamarin configurado correctamente en un Mac es mediante la instalación [Comunidad de Visual Studio para Mac](#).

Visual Studio se conectará a la Mac para lanzar el compilador Xcode y SDK, y por lo tanto las conexiones remotas debe ser habilitado para este último. La documentación oficial Xamarin tiene una [página específica](#) que le ayudará a configurar un ordenador Mac y le recomiendo que lea con cuidado, sobre todo porque explica cómo configurar los perfiles y certificados, y cómo usar Xcode para realizar las configuraciones preliminares. En realidad, se trata de la documentación Xamarin.iOS, pero los mismos pasos se aplican a Xamarin.Forms.

La creación de soluciones Xamarin.Forms

Suponiendo que ha instalado y configurado el entorno de desarrollo, el siguiente paso es la apertura de Visual Studio para ver cómo se crea soluciones Xamarin.Forms y lo que estas soluciones consisten. Las plantillas de proyecto para Xamarin.Forms están disponibles en el **Visual C #, CrossPlatform nodo de la Nuevo proyecto** de diálogo (ver Figura 3).

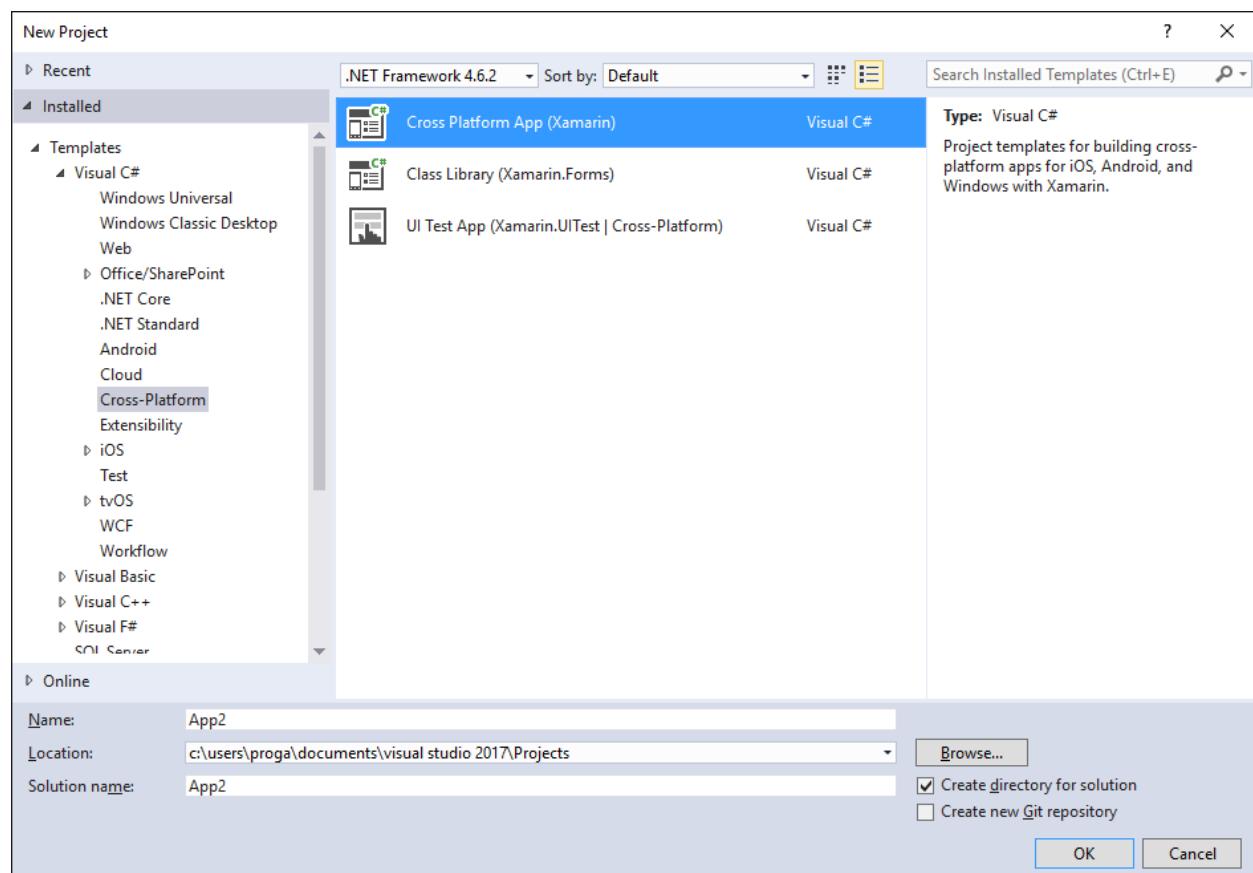


Figura 3: plantillas de proyecto para Xamarin.Forms

los **Cruz plataforma de aplicaciones (Xamarin)** plantilla es la que utiliza para construir aplicaciones móviles. El (Xamarin.Forms) plantilla de biblioteca de clases le permite crear una biblioteca de clases reutilizables que se puede consumir en proyectos Xamarin.Forms, y la plantilla de interfaz de usuario de la aplicación de prueba se utiliza para crear la interfaz de usuario pruebas automatizadas, pero estas dos plantillas no será discutido en este libro electrónico. Selecciona el

Cruz plataforma de aplicaciones plantilla y especificar un nombre de proyecto (esto no es relevante en este momento), a continuación, haga clic **DE ACUERDO**. En este punto, Visual Studio le pedirá que seleccione entre el blanco y las plantillas de aplicaciones maestro detalle (ver Figura 4). Este último genera una interfaz de usuario básica basada en páginas y los elementos visuales que se explicarán más adelante, con algunos datos de la muestra. No es un buen punto de partida menos que ya tenga experiencia con Xamarin, por lo que seleccionar el **Aplicación en blanco**

modelo.

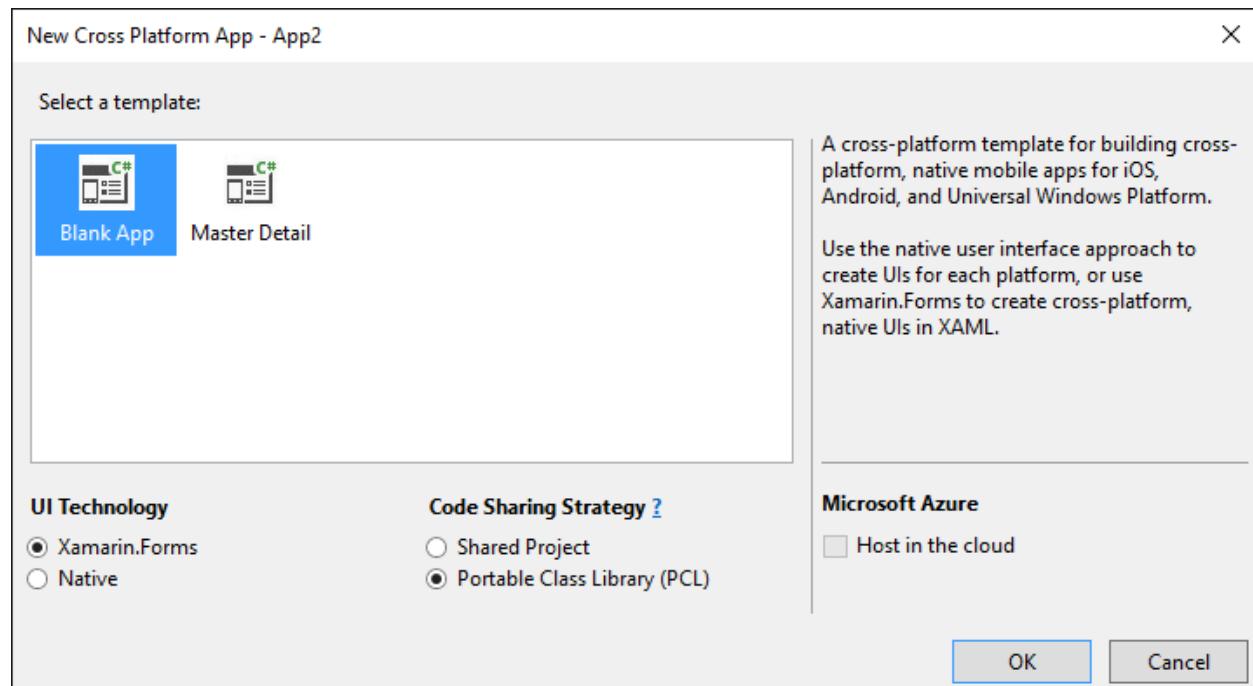


Figura 4: Selección de la plantilla, la tecnología de interfaz de usuario, y la estrategia de código compartido para un nuevo proyecto

En el **La tecnología de interfaz de usuario** grupo, seleccione **Xamarin.Forms**. Si selecciona nativo, su solución se centrará en proyectos específicos de la plataforma en lugar de código de plataforma cruzada. En el **Estrategia de código compartido** grupo, se puede elegir entre Proyecto compartido y la biblioteca de clases portátil. La estrategia de código compartido es un tema muy importante en Xamarin.Forms y Capítulo 2 proporcionará una explicación detallada. Por ahora, seleccione la **Biblioteca de clases portátiles** opción y haga clic **DE ACUERDO** cuando esté listo. Después de unos segundos, mientras que la generación de la solución, Visual Studio le pedirá que especifique la versión de destino de Windows 10 para su nueva aplicación. Deje la selección por defecto sin cambios y seguir adelante. También mostrará un cuadro de diálogo de bienvenida donde, en la segunda pantalla, tendrá una opción para especificar la ubicación de un ordenador Mac. Omite este paso por ahora, ya que se tratará en la siguiente sección. En el Explorador de soluciones, se verá que la solución se compone de cuatro proyectos, como se demuestra en la Figura 5.

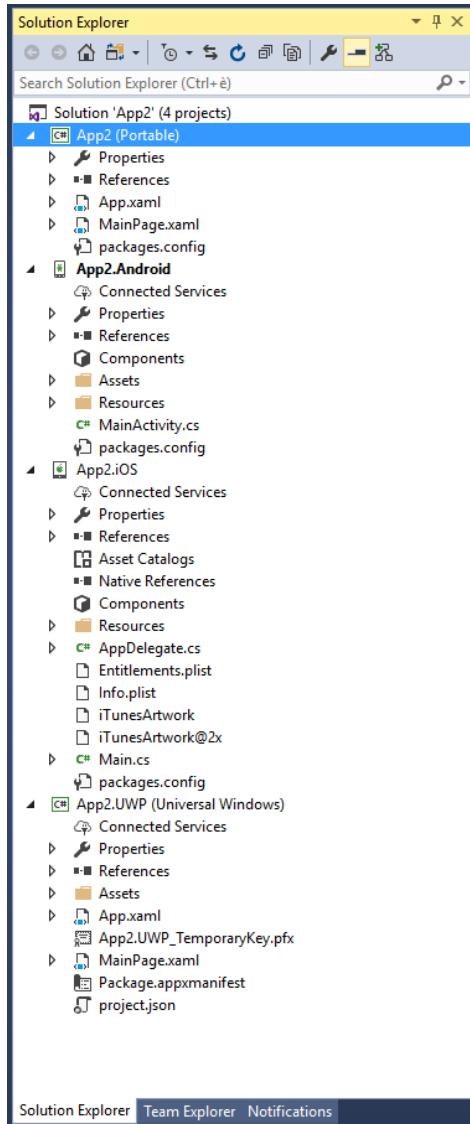


Figura 5: La estructura de una solución Xamarin.Forms

El primer proyecto es una biblioteca de clases ya sea portátil o un proyecto compartido, dependiendo de su selección en la creación del proyecto. Este proyecto contiene todo el código que puede ser compartido a través de plataformas y su implementación será discutido en el capítulo siguiente. Por ahora, lo que hay que saber es que este proyecto es el lugar donde se va a escribir toda la interfaz de usuario de la aplicación, y todo el código que no requiere la interacción con las API nativas. El segundo proyecto, cuyo sufijo es Android, es un proyecto nativa Xamarin.Android. Cuenta con una referencia al código compartido y de Xamarin.Forms, e implementa la infraestructura requerida para su aplicación a ejecutar en los dispositivos Android. El tercer proyecto, cuyo sufijo es iOS, es un proyecto nativa Xamarin.iOS. Éste también tiene una referencia al código compartido y de Xamarin.Forms, e implementa la infraestructura requerida para su aplicación a ejecutar en el iPhone y el IPAD. El cuarto y último proyecto es un proyecto universal nativa de Windows (UWP) que tiene una referencia al código compartido e implementa la infraestructura para su aplicación para funcionar en Windows 10 dispositivos, tanto para el escritorio y dispositivos móviles. Ahora voy a dar más detalles sobre cada proyecto de plataforma, para que tenga un conocimiento básico de sus propiedades. Esto es muy importante, ya que necesitará para afinar las propiedades del proyecto cada vez que cree una nueva solución Xamarin.Forms. Ahora voy a dar más detalles sobre cada proyecto de plataforma, para que tenga un conocimiento básico de sus propiedades. Esto es muy importante, ya que necesitará para afinar las propiedades del proyecto cada vez que cree una nueva solución Xamarin.Forms. Ahora voy a dar más detalles sobre cada proyecto de plataforma, para que tenga un conocimiento básico de sus propiedades. Esto es muy importante, ya que necesitará para afinar las propiedades del proyecto cada vez que cree una nueva solución Xamarin.Forms.

La biblioteca Xamarin.Forms

Técnicamente hablando, Xamarin.Forms es una biblioteca .NET que expone todos los objetos analizados en este libro electrónico a través de un espacio de nombres raíz llamada **Xamarin.Forms**. Recientemente se ha [de código abierto](#) y se distribuye como una paquete NuGet que Visual Studio se instala de forma automática a todos los proyectos cuando se crea una nueva solución. El Administrador de NuGet paquete en Visual Studio y luego notificarle de las actualizaciones disponibles. Dado que las soluciones que crean Xamarin.Forms debe permitir incluso si su PC está desconectado, Visual Studio instala la versión del paquete NuGet en la caché local, que no suele ser la última versión disponible. Por esta razón, se recomienda que actualice los Xamarin.Forms y otros paquetes Xamarin en la solución a la última versión, y, obviamente, también se recomienda que sólo se actualiza a versiones estables. Aunque alfa y beta versiones intermedias están a menudo disponibles, su uso sólo se diseñó es para experimentar con nuevas características todavía en desarrollo. Al escribir estas líneas, la versión 2.3.4.247 es la última versión estable.

El proyecto Xamarin.Android

Xamarin.Android hace posible que su solución Xamarin.Forms se ejecute en los dispositivos Android. Los **MainActivity.cs** archivo representa la actividad de inicio de la aplicación para Android que genera Xamarin. En Android, una actividad puede ser pensado como una única pantalla con una interfaz de usuario, y cada aplicación tiene al menos uno. En este archivo, Visual Studio agrega código de inicio que no se debe cambiar, especialmente el código de inicialización para Xamarin.Forms que se ven en las dos últimas líneas de código. En este proyecto, se puede añadir código que requiere el acceso a las API nativas y las características específicas de la plataforma, como aprenderá en el capítulo 8. La **recursos carpeta** también es muy importante, ya que contiene las subcarpetas donde se puede añadir iconos e imágenes para diferentes resoluciones de pantalla. El nombre de este tipo de carpetas comienza con *dibujable* y cada uno representa una resolución de pantalla particular. El Xamarin [documentación](#) explica a fondo cómo proporcionar iconos e imágenes de diferentes resoluciones en Android. Los **propiedades** elemento en el Explorador de soluciones le permite acceder a las propiedades del proyecto, como lo haría con cualquier solución de C#. En el caso de Xamarin, en el

Solicitud pestaña (ver Figura 6) se puede especificar la versión del SDK de Android que Visual Studio debe utilizar para construir el paquete de la aplicación.

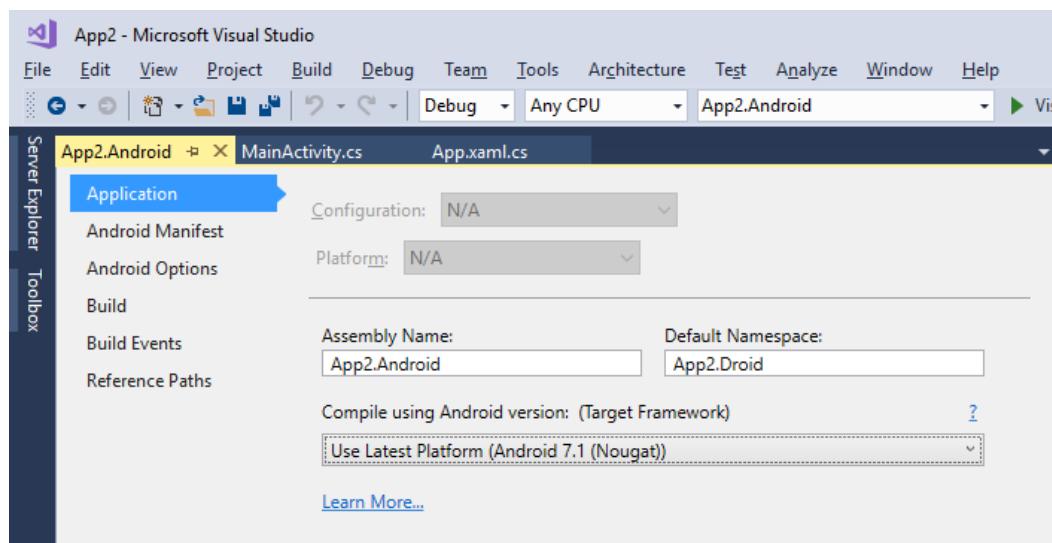


Figura 6: Selección de la versión del SDK de Android para la compilación

Visual Studio selecciona automáticamente la versión más alta disponible en su máquina. Esto no afecta a la versión mínima de Android que desea orientar; sino que se relaciona con la versión de las herramientas de construcción que utilizará Visual Studio. Mi recomendación es dejar la selección por defecto sin cambios.



Consejo: Puede administrar versiones del SDK instalado usando el Administrador de Android SDK, una herramienta que se puede poner en marcha tanto del menú de programas de Windows y desde Visual Studio seleccionando Herramientas, Android, Gestor de SDK de Android.

los **Manifiesto Android** pestaña es aún más importante. Aquí se especifica metadatos de su aplicación, tales como nombre, número de versión, el ícono y permisos que el usuario debe conceder a la aplicación. La Figura 7 muestra un ejemplo.

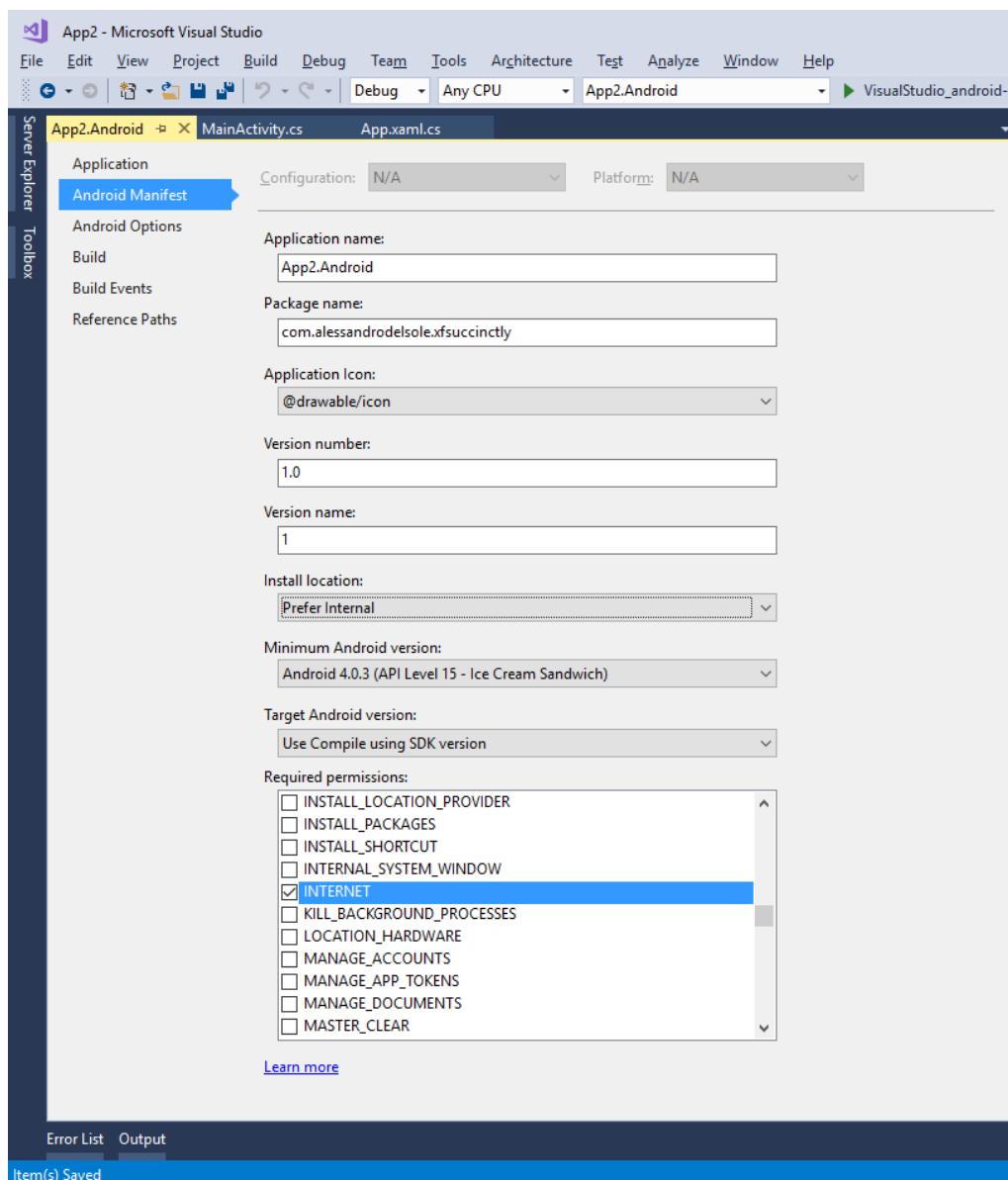


Figura 7: El manifiesto Android

La información que proporcione en el manifiesto de Android también es importante para su publicación a Google Play. Por ejemplo, el nombre del paquete identifica de forma exclusiva el paquete de aplicación en la tienda de Google Play y, por convención, es el siguiente formulario: com.companyname.appname, que es fácil de entender (. Com es un prefijo convencional). El nombre de la versión es su versión de la aplicación, mientras que el número de versión es una cadena de un solo dígito que representa actualizaciones. Por ejemplo, es posible que tenga nombre de la versión 1.0 y la versión 1, nombre de la versión 1.1 y la versión 2, nombre de la versión

1,2 y el número de la versión 3, y así sucesivamente.

los **ubicación de instalación** opción le permite especificar si su aplicación debe ser instalado sólo en el almacenamiento interno o si las tarjetas de memoria están permitidos, pero recuerda que, a partir de Android 6.0, las aplicaciones no se puede instalar en un almacenamiento extraíble por más tiempo. En el **Mínimo versión de Android desplegable**, puede seleccionar la versión mínima de Android que desea orientar.

Es importante que prestar especial atención a la **permisos necesarios** lista. Aquí hay que especificar todos los permisos que su aplicación debe ser concedida con el fin de acceder a recursos tales como la Internet, la cámara, otros dispositivos de hardware, sensores, y mucho más. Recuerde que, a partir de Android 6.0, el sistema operativo le pedirá confirmación al usuario antes de acceder a un recurso que requiere uno de los permisos que ha marcado en el manifiesto, y la aplicación se bloqueará si se intenta acceder a un recurso sensible, pero la relacionada el permiso no fue seleccionado en el manifiesto.

En el **Opciones para Android** pestaña, usted será capaz de gestionar la depuración y construir opciones. Sin embargo, no voy a caminar a través de todas las opciones disponibles aquí. Es de destacar la **Utilice despliegue rápido** opción, sin embargo, que está activada de forma predeterminada. Cuando está activado, el despliegue de la aplicación para un dispositivo físico o emulado sólo reemplazar los archivos modificados. A menudo, esto puede hacer que la aplicación no funcione correctamente o no a ponerse en marcha, así que mi sugerencia es deshabilitar esta opción. Las otras pestañas son los mismos que para otros proyectos de .NET.

El proyecto Xamarin.iOS

Al igual que en el proyecto Xamarin.Android, el proyecto Xamarin.iOS hace posible que sus soluciones Xamarin.Forms que se ejecutan en el iPhone y el IPAD. Suponiendo que haya configurado un Mac, Visual Studio necesitará saber su dirección en la red. Visual Studio normalmente pide esto después de la creación de una nueva o la apertura de una solución Xamarin.Forms existente, pero puede introducir manualmente la dirección MAC con **Herramientas, IOS, Agente Mac Xamarin**. En el **Agente Mac Xamarin** de diálogo, Visual Studio debe ser capaz de enumerar cualquier sistema Mac detectados en la red. Sin embargo, se recomienda encarecidamente que volver a añadir un Mac, proporcionando su dirección IP en lugar de su nombre. Por ejemplo, la Figura 8 muestra el cuadro de diálogo Xamarin Mac Agente mostrar mi máquina Mac Mini tanto con su nombre y su dirección IP, pero Visual Studio establece una conexión basada en el IP, no el nombre.

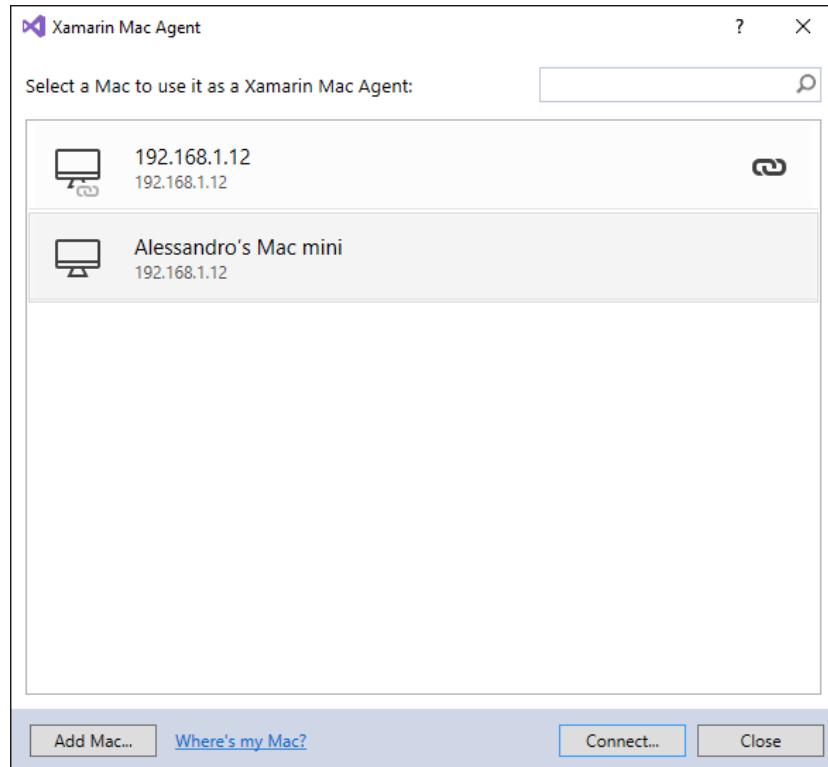


Figura 8: Conexión de Visual Studio para Mac

Si no se detecta un Mac, haga clic **Añadir Mac** e introduzca su dirección IP por primera vez. Entonces, cuando se le solicite, introduzca las mismas credenciales que utiliza para iniciar sesión en el Mac. Si la conexión se realiza correctamente, Visual Studio mostrará un mensaje de éxito en la barra de estado. Para el proyecto Xamarin.iOS, la

AppDelegate.cs archivo contiene el código de inicialización Xamarin.Forms y no debe ser cambiado. Usted puede agregar todo el código que requiere el acceso a las API nativas y las características específicas de la plataforma en este proyecto, como aprenderá en el capítulo 8. En el **info.plist** de archivos (ver Figura 9), con cada ficha, puede configurar sus metadatos de aplicación, la versión objetivo mínimo, dispositivos y orientaciones compatibles, características (como el centro de juego y mapas de integración), los activos visuales tales como imágenes de inicio y los iconos, y otra características avanzadas.

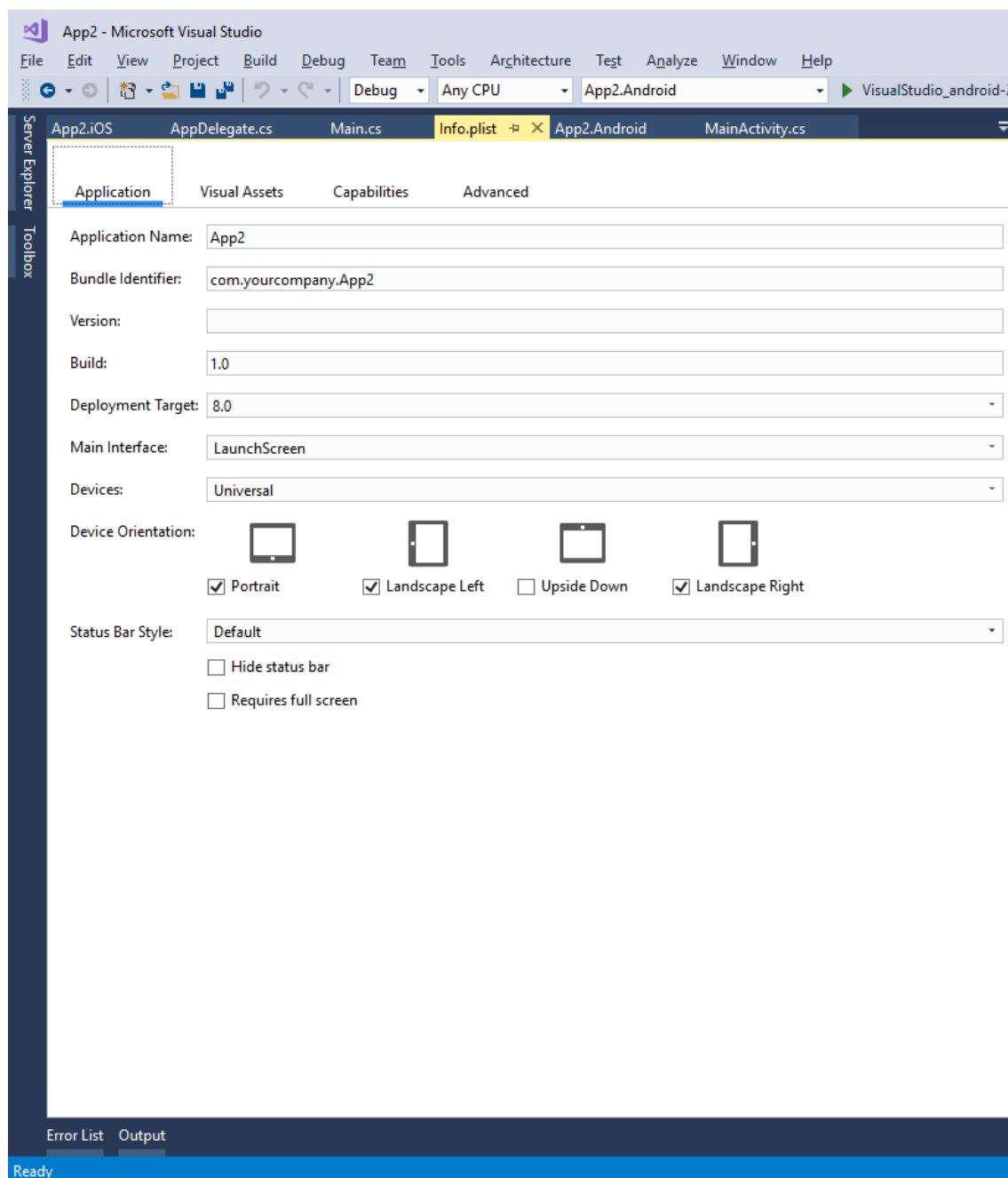


Figura 9: El archivo Info.plist

los **info.plist** archivo representa manifiesto de la aplicación en iOS y por lo tanto no está relacionada con sólo Xamarin.iOS. De hecho, si tiene experiencia con Xcode y el desarrollo nativo de iOS, ya saben este archivo. A diferencia de Android, el sistema operativo iOS incluye las directivas de restricción que se aplican automáticamente a los recursos más sensibles, especialmente las relacionadas con la seguridad y la privacidad. Además, existen diferencias entre iOS 8.x, 9.x, 10.x en cómo el sistema operativo se encarga de estas opciones. los [referencia info.plist](#) le ayudará a entender cómo configurar correctamente las excepciones. Entre las propiedades del proyecto, lo más importante es, sin lugar a dudas, el IOS paquete de firma. Se utiliza el paquete de firma para especificar la identidad que las herramientas de Apple deben utilizar para firmar el paquete de la aplicación, y especifique el archivo de suministro que se utiliza para asociar un equipo de

desarrolladores a un identificador de aplicación. Configuración de identidades firma y perfiles es particularmente importante cuando la preparación de una aplicación para la edición. La Figura 10 muestra las propiedades de firma iOS paquete.

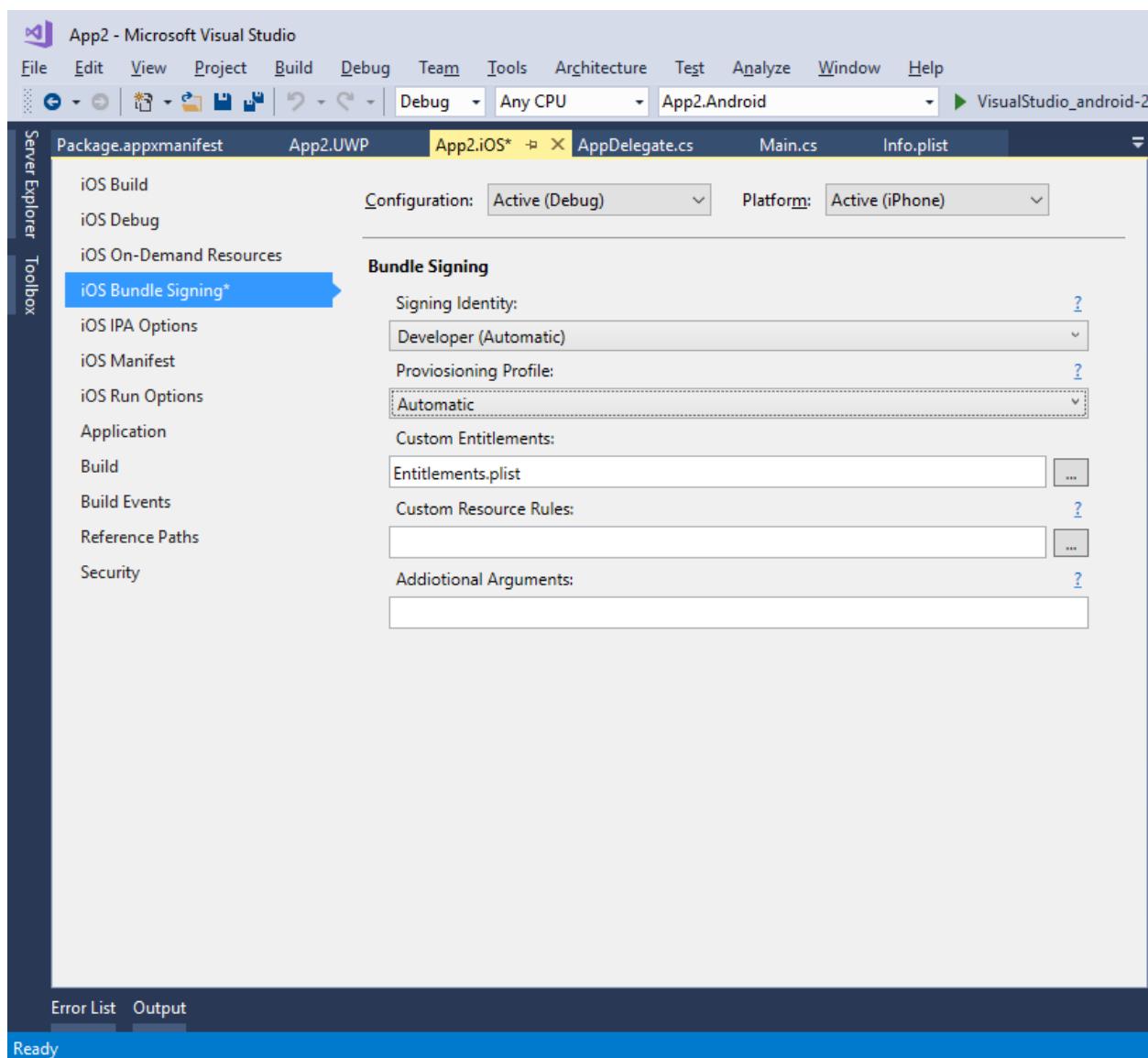


Figura 10: Las opciones de firma iOS Bundle

Como se puede imaginar, Visual Studio puede detectar la firma de identidades y perfiles de datos sólo cuando está conectado a un Mac disponible, ya que esta información se genera a través de Xcode. Más detalles sobre la configuración de un proyecto Xamarin.iOS se ofrecen a través de la [documentación](#).

El proyecto Plataforma Windows universal

El proyecto Plataforma Windows universal en una solución Xamarin.Forms no es más que un proyecto uwp normal con una referencia al código compartido y al paquete Xamarin.Forms. En el **App.xaml.cs** archivo, se puede ver el código de inicialización que no se debe cambiar. Lo que se necesita para configurar, en cambio, es el manifiesto de aplicación, que se puede editar haciendo doble clic en el **Package.appxmanifest** presentar en el Explorador de soluciones. Visual Studio tiene un buen editor para UWP se manifiesta, y que será, al menos, configurar los metadatos de aplicaciones (ver Figura 11), los activos visuales tales como iconos y logotipos, y capacidades (ver Figura 12). Estos incluyen permisos necesarios para especificar antes de la prueba y la distribución de sus aplicaciones y de Windows 10 le pedirá confirmación al usuario antes de acceder a los recursos que requieren un permiso.

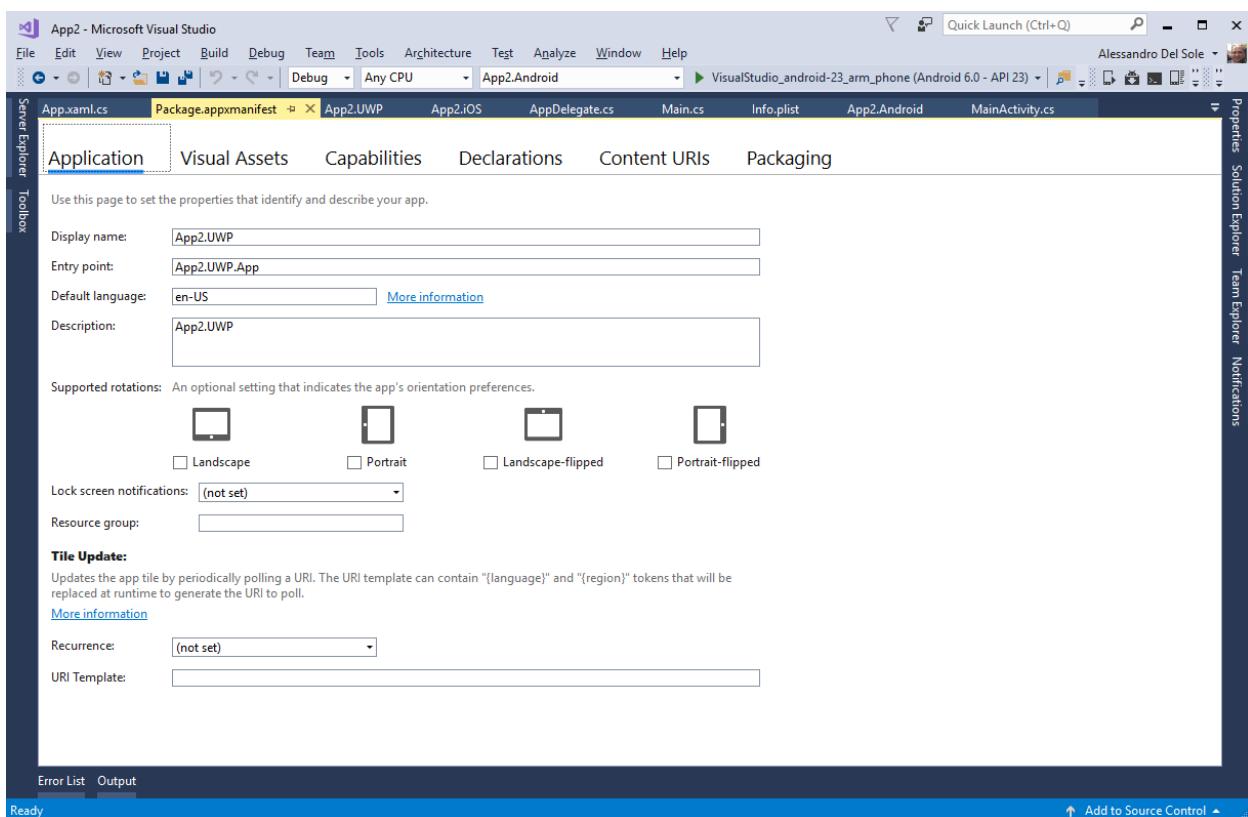


Figura 11: Edición de metadatos en proyectos UWP

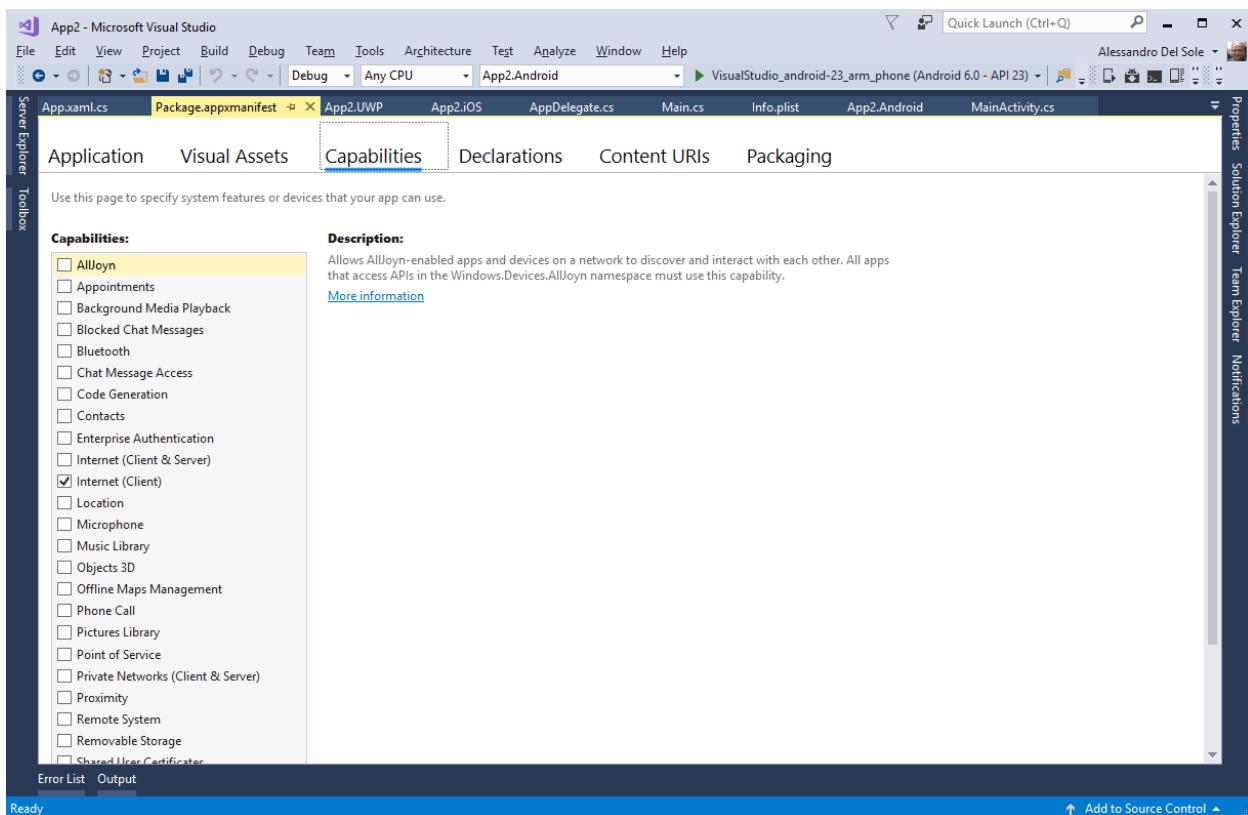


Figura 12: capacidades especificando en proyectos UWP

La documentación oficial se explica cómo configurar otras opciones. Sin embargo, recuerde que necesita una suscripción de pago a la tienda de Windows con el fin de completar la configuración de embalaje que se proporcionan en la preparación para su publicación.

Depuración y prueba aplicaciones localmente

A partir aplicaciones que construir con Xamarin.Forms para la depuración y pruebas no podía ser más sencillo: sólo tiene que seleccionar uno de los proyectos de plataforma en el Explorador de soluciones como el proyecto de inicio, a continuación, seleccionar el dispositivo de destino y presione F5. No se olvide de reconstruir su solución antes de la depuración por primera vez. Cuando se inicia una aplicación para la depuración, Visual Studio va a construir su solución e implementar el paquete de aplicaciones en el dispositivo físico seleccionado o emulador. El resultado del proceso de construcción es un archivo APK para Android, un archivo .ipa para iOS, y un archivo .appx para Windows 10. Cuando se inicia la aplicación, ya sea en un dispositivo físico o en un emulador, Visual Studio concede una instancia de el depurador y usted será capaz de utilizar todas las conocidas, potentes herramientas de depuración del IDE, incluyendo (pero no limitados a) los puntos de interrupción, puntas de datos, ventanas de herramientas, ver las ventanas, y más. La forma más fácil para seleccionar la plataforma de destino y la configuración es mediante el uso de la barra de herramientas estándar, que se puede ver en la figura 13.

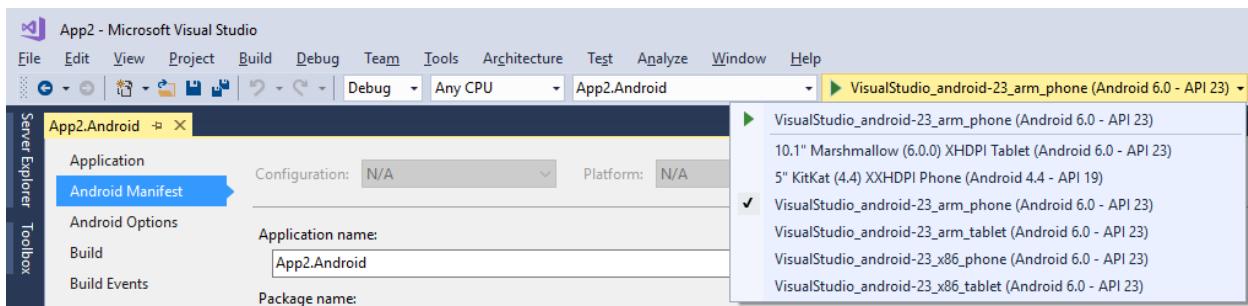


Figura 13: Selección de la plataforma y dispositivos de destino para la depuración

La configuración de depuración es la elección apropiada durante el desarrollo. Usted selecciona Ad-hoc cuando se prepara para su distribución en iOS y Android, o la versión para Windows 10. La arquitectura objetivo es normalmente Cualquier CPU para Android y Windows, mientras que es iPhoneSimulator para depurar una aplicación para iOS en el simulador de iOS o iPhone para la depuración una aplicación en un iPhone o iPad física (recuerde que un dispositivo de Apple física debe estar asociado a tu Mac a través de Xcode y conectada al Mac, no el PC). También puede seleccionar rápidamente el proyecto de inicio y especificar el dispositivo de destino. Por ejemplo, en la figura 13 se puede ver una lista de configuraciones emulador de Android.



NOTA: En este libro electrónico, voy a dar cifras que muestran todas las plataformas soportadas en acción cuando sea relevante para hacerlo. En otros casos, voy a mostrar una única plataforma en la acción, lo que significa que se espera que el mismo comportamiento en todas las plataformas.

La Figura 14 muestra la aplicación en blanco creado anteriormente se ejecuta en las tres plataformas dentro de los respectivos emuladores. Nótese cómo en el fondo de Visual Studio muestra el **Salida** ventana en la que recibe los mensajes del depurador; usted será capaz de utilizar todas las otras herramientas de depuración de manera similar.

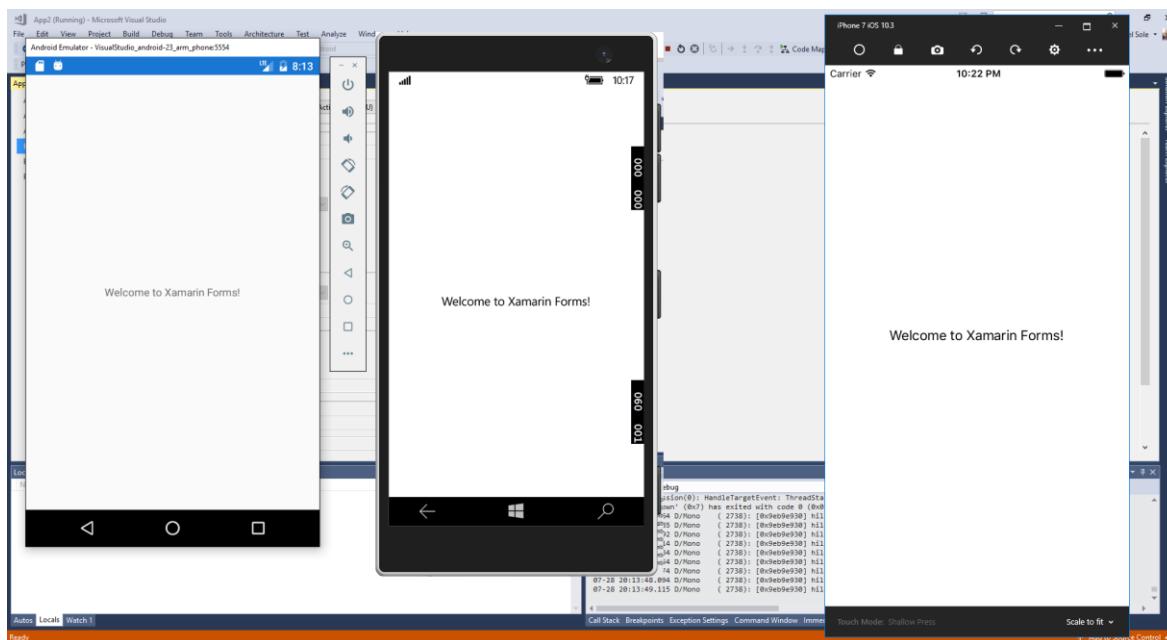


Figura 14: Una aplicación construida con Xamarin.Forms se ejecuta en todas las plataformas



Nota: Aunque Visual Studio 2017 permite ejecutar varias instancias de una aplicación en múltiples plataformas, la Figura 14 ha sido capturada con la versión de Android en marcha, con el emulador de Windows y capturas de pantalla simulador de iOS añaden por separado.

Como se puede imaginar, con el fin de construir el iOS, Visual conectado al Mac y puesto en marcha el Estudio de Apple SDK. En la figura 14, se ve un ejemplo del simulador de iOS, que merece un poco más de consideración.

Configurar el simulador de iOS

A diferencia de Android y Windows, con el que los emuladores de ejecutar localmente en el equipo de desarrollo de Windows, el simulador de iOS se ejecuta en su Mac. Sin embargo, si usted tiene Visual Studio 2017 Enterprise, puede también descargar e instalar el [Remoted simulador de iOS](#) para ventanas. Cuando esto se instala, el simulador se ejecutará en el equipo de Windows en lugar de ejecutar en el Mac. El simulador de iOS en Windows no está habilitado de forma predeterminada, por lo que necesita para abrir **Herramientas**, **Opciones**, **Xamarin**, **Ajustes de iOS**, y seleccione el **Simulador remoto a Windows** opción. Un ejemplo del simulador de iOS está disponible en la Figura 14, pero verá otros en los próximos capítulos. Si tiene Visual Studio 2017 Comunidad o profesional, entonces usted necesita para utilizar el simulador en el Mac.

Ejecución de aplicaciones en dispositivos físicos

Visual Studio se puede implementar fácilmente un paquete de aplicaciones para dispositivos Windows y Android físicas. Para Android, primero se necesita habilitar el modo de programador, que a lograr con los siguientes pasos:

1. Abra la **ajustes** aplicación.
2. Pulse en el **Acerca de** ít.
3. En la lista que aparece, busque el número de versión del sistema operativo y pulse este artículo siete veces.

En este punto, sólo tiene que conectar el dispositivo en el puerto USB de su PC y Visual Studio reconocerá inmediatamente. Será visible en la lista de dispositivos disponibles que se pueden ver en la Figura 13. Para Windows, primero debe permitir que tanto el **equipo como los dispositivos para el desarrollo, y el funcionario** [documentación](#) proporciona orientación sobre esto. Entonces usted será capaz de conectar sus dispositivos en el puerto USB de su PC y Visual Studio reconocerlos como dispositivos de destino disponibles. Para los dispositivos móviles de Apple, lo que necesita para conectar su iPhone o iPad al ordenador Mac, asegurándose de que estén visibles a través de Xcode. Entonces, cuando se inicia la depuración de Visual Studio, su aplicación se implementa en el iPhone o el IPAD a través de la Mac.

La aplicación Xamarin jugador vivo

Microsoft ha publicado recientemente una vista previa de la [Xamarin jugador vivo](#) para Android y para iOS. El objetivo de esta aplicación es que sea más fácil de depurar y probar aplicaciones de Android y iOS en un dispositivo físico conectado a la misma red que el equipo de desarrollo y evitar la necesidad de tener un Mac para compilar y depurar una aplicación en iOS. Actualmente, esta aplicación sólo funciona con [Visual Studio 2017 Vista previa Versión 15.3](#). Básicamente, se le tendrá que vincular la aplicación Xamarin jugador vivo con Visual Studio a través de un código de barras, y luego Visual Studio será capaz de implementar los paquetes de aplicaciones para el dispositivo físico a través de la red. Debido a que la aplicación está en vista previa y requiere una versión preliminar de Visual Studio 2017, no se tratará en este capítulo. Hemos de tener en cuenta que es una buena alternativa si no tiene un Mac todavía. Sin embargo, recuerde que hay un número importante de [limitaciones](#) y que todavía se necesita un ordenador Mac para firmar y distribuir aplicaciones iOS.

El análisis y perfiles de aplicaciones

La caja de herramientas Xamarin se ha enriquecido recientemente con tres herramientas asombrosas: [Xamarin libros](#), [Xamarin inspector](#) y [Xamarin Profiler](#). La primera herramienta le permite explorar una serie de plataformas de desarrollo .NET y Mono con ejemplos interactivos. El inspector Xamarin le permite inspeccionar el árbol visual de su Xamarin aplicaciones muy bien y realizar cambios en la interfaz de usuario en tiempo real (ver el [documentación](#)).

El Xamarin Profiler es una suite completa de herramientas de análisis en un programa que se puede utilizar para perfilar sus aplicaciones móviles y **analizar el rendimiento, uso de memoria, el consumo de CPU, y mucho más**. En Visual Studio, puede seleccionar **Herramientas, Xamarin Profiler**, y poner en marcha cualquier versión de la plataforma de su aplicación para el perfilado, en lugar de pulsar F5. Despues de la aplicación se ha desplegado en el dispositivo seleccionado y antes de la puesta en marcha, el Analizador le preguntará qué tipo de análisis que desea ejecutar en contra de la aplicación. La Figura 15 muestra un ejemplo basado en la selección de todas las herramientas instrumentación disponibles.

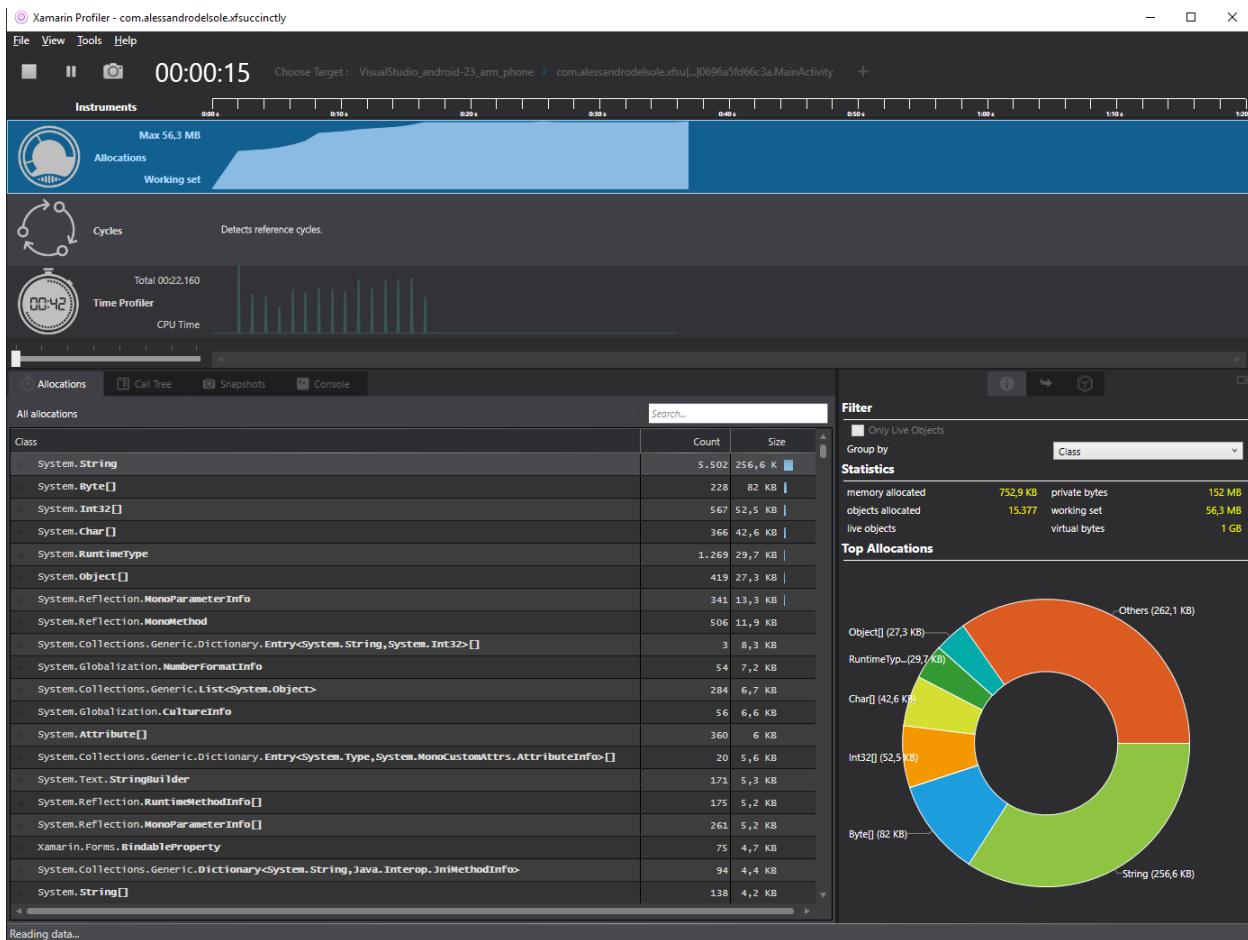


Figura 15: Análisis de rendimiento de las aplicaciones con Xamarin Profiler

También puede tomar y comparar instantáneas de la memoria en diferentes momentos para ver si la asignación de memoria puede causar problemas potenciales. Esta es una excelente herramienta de análisis de rendimiento y la [documentación](#) proporcionará todos los detalles que necesita para mejorar el rendimiento de su aplicación.

Resumen del capítulo

En este capítulo se introduce la plataforma Xamarin.Forms y sus objetivos, la descripción de las herramientas de desarrollo necesarias y ofrecer una visión general de una solución Xamarin.Forms, pasando a través de los proyectos de plataforma y sus valores más importantes. También ha visto cómo iniciar una aplicación para depurar utilizando emuladores y cómo tomar ventaja de la nueva aplicación Xamarin jugador vivo que le permite depurar una aplicación de iOS sin tener un Mac (que todavía se requiere para el desarrollo grave). Ahora que tiene una visión general de Xamarin.Forms y las herramientas de desarrollo, el siguiente paso es la comprensión de lo que es en su núcleo: compartir código entre plataformas.

Capítulo 2 Código intercambio de información entre plataformas

Xamarin.Forms le permite construir aplicaciones que se ejecutan en Android, iOS y Windows desde una única base de código C#. Esto es posible porque, en su núcleo, Xamarin.Forms permite el intercambio entre las plataformas de todo el código de la interfaz de usuario y todo el código que no es específico de la plataforma. Hay diferentes maneras de compartir código entre plataformas, cada una con sus ventajas y desventajas. Este capítulo explica las estrategias de código compartido disponibles en Xamarin.Forms, destacando sus características de manera que será más fácil para usted para decidir qué estrategia es la mejor para sus soluciones.

Introducción a las estrategias de código compartido

En el capítulo 1, he explicado cómo crear una solución Xamarin.Forms en Visual Studio 2017 y que se compone de cuatro proyectos: tres proyectos de plataforma (Android, iOS, y UWP) y un proyecto que permite compartir código entre plataformas. Con este enfoque, los desarrolladores pueden compartir todo el código de la interfaz de usuario y todo el código que no está acoplado a las API de cada plataforma, maximizando la reutilización de código y simplificar el proceso de creación de tres aplicaciones nativas diferentes desde una única base de código C#. En esta explicación, introduce brevemente la biblioteca de clases portátil (PCL) como un tipo de proyecto que permite compartir código. Sin embargo, Xamarin.Forms permite compartir código entre plataformas de tres maneras diferentes: Clase Bibliotecas portátil, proyectos compartidos, y bibliotecas .NET estándar. Este capítulo contiene una discusión a fondo de estas tres estrategias de código compartido, proporcionar más información sobre el tipo de proyecto PCL. Vale la pena mencionar que, en el momento de la escritura, Visual Studio 2017 incluye plantillas de proyecto basado en PCLs y proyectos compartidos, mientras que se necesita algún manual, sin embargo sencillos pasos para .NET estándar. En la sección sobre .NET Standard, aprenderá cómo convertir un PCL en una biblioteca .NET Standard fácilmente.

Compartir código con bibliotecas de clases portátiles

Como su nombre lo indica, bibliotecas de clases portátiles (PCL) son las bibliotecas que se pueden consumir en múltiples plataformas. Más específicamente, pueden ser consumidos en múltiples plataformas sólo si se dirigen a un subconjunto de APIs disponibles en todas las plataformas. PCLs han existido durante muchos años, y desde luego no son exclusivos de Xamarin. De hecho, pueden ser utilizados en muchos otros escenarios de desarrollo. Por ejemplo, un PCL se podría utilizar para compartir una arquitectura Model-View-ViewModel entre un proyecto WPF y un proyecto UWP. Las características más importantes de un PCL son los siguientes:

- Producen una, montaje .dll compilado reutilizable.
- Ellos pueden hacer referencia a otras bibliotecas y tienen dependencias tales como paquetes NuGet.
- Pueden contener archivos XAML para la definición de interfaz de usuario y archivos de C#.
- No pueden exponer código que aprovecha las API específicas de una determinada plataforma, de lo contrario ya no sería portátil.
- Ellos son una mejor opción cuando se necesita para poner en práctica los patrones arquitectónicos como MVVM, la fábrica, la inversión de control (IoC) con inyección de dependencias, y localizador de servicios.

- Con respecto a Xamarin.Forms, pueden utilizar el modelo de servicio de localización para implementar una abstracción e invocar las API específicas de la plataforma a través de proyectos de plataforma (esto será discutido en el capítulo 8).
- Son más fáciles de probar la unidad.

Por ejemplo, un PCL que se utiliza para compartir código entre proyectos WPF y UWP nunca podrían contener código que tiene acceso al sensor de posición de un dispositivo, ya que este no es compatible con WPF y requiere Windows 10 APIs que WPF no puede acceder. En cambio, un PCL se puede utilizar para acceder a los recursos de Internet a través de la **HttpClient** clase en múltiples plataformas, ya que este es comúnmente disponibles. Normalmente, debe crear un proyecto PCL manualmente y luego agregar las referencias necesarias para y desde otros proyectos de la solución. En el caso de Xamarin.Forms, que en vez de decidir una estrategia de código compartido cuando se crea un nuevo proyecto (véase la Figura 4) y luego Visual Studio 2017 generará automáticamente un proyecto PCL que hace referencia a los proyectos de plataforma en la solución y que tiene una dependencia en el paquete de Xamarin.Forms NuGet.



Nota: En todos los ejemplos de este libro electrónico, voy a utilizar el PCL como la estrategia de código compartido por las siguientes razones: que hace que sea más fácil de usar y administrar otras bibliotecas, es una mejor opción con proyectos del mundo real que requieren arquitecturas más complejas, y es más fácil de cambiar en una biblioteca .NET estándar.

Compartir código con proyectos compartidos

proyectos compartidos, así como PCLs, no son específicos de Xamarin y han existido durante muchos años. proyectos compartidos son esencialmente surtidos sueltos de archivos que se pueden compartir con otros proyectos. A continuación se presenta una lista de las características más importantes de un proyecto compartido, también destaca las diferencias con un PCL:

- No producen un ensamblaje .dll compilado reutilizable.
- Ellos no pueden hacer referencia a otras bibliotecas y tienen dependencias tales como paquetes NuGet.
- Pueden contener archivos XAML para la definición de interfaz de usuario y archivos de C #.
- Pueden contener código específico de la plataforma que se puede utilizar directivas de compilación y preprocesador condicionales.

Con el fin de seleccionar un proyecto compartido como la estrategia de código compartido, en el **New Cross plataforma de aplicaciones** de diálogo (ver Figura 4) se selecciona **Proyecto compartido** en el **Estrategia de código compartido** grupo. Cuando la solución está lista en el Explorador de soluciones, verá el proyecto compartido de aspecto similar a la Figura 16.

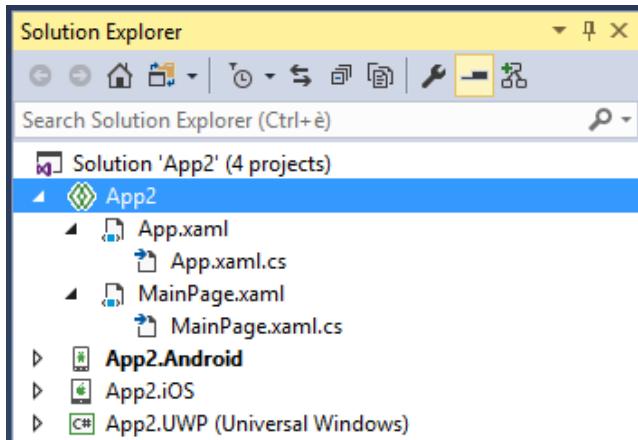


Figura 16: El proyecto compartido en una solución Xamarin.Forms

Una nota importante aquí es que los proyectos de plataforma (Android, iOS, y UWP) tienen una referencia al proyecto compartido, pero el proyecto compartido no puede tener referencias o dependencias. Además, las propiedades de proyectos compartidos no tienen propiedades a nivel de proyecto; en cambio, sólo se puede acceder a las propiedades de los archivos individuales que contienen. Como era de esperar, no hay ningún nodo Propiedades o referencias de proyectos compartidos en el Explorador de soluciones. Proyectos compartidos pueden contener un número infinito de diferentes archivos y recursos, incluyendo archivos XAML para la interfaz de usuario y los archivos de código C#. Esto es posible porque los proyectos compartidos no se compilan, en vez resuelve el compilador archivos y recursos de código cuando toda la solución está construida. Dado que los proyectos compartidos no hacen referencia a ninguna biblioteca, que podría ser difícil de recordar los tipos y miembros que se pueden utilizar en C#.

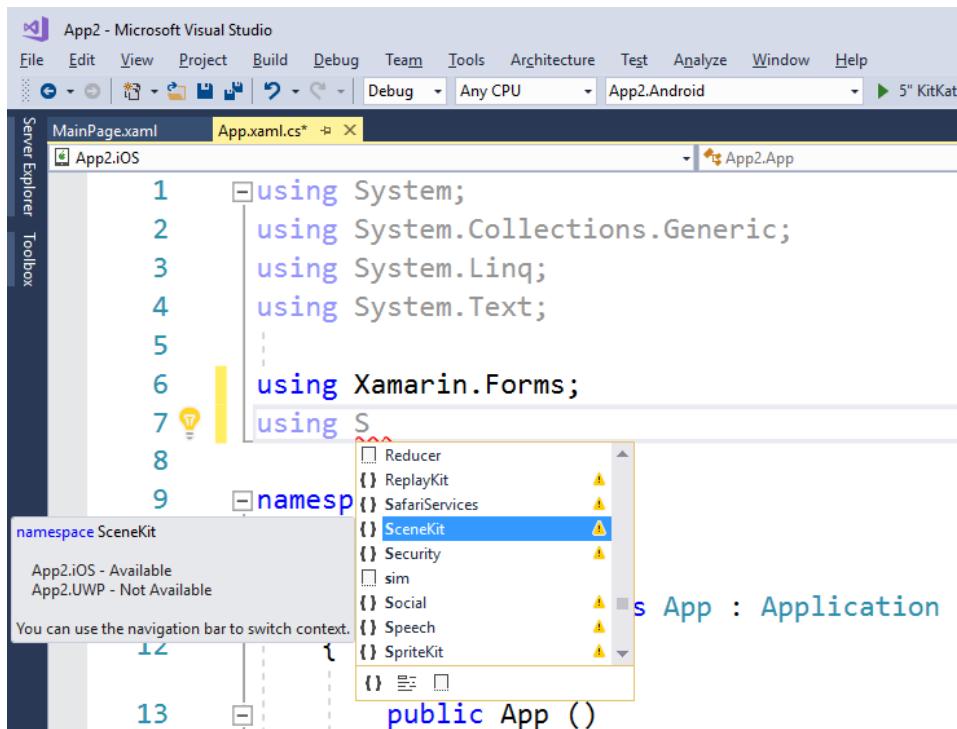


Figura 17: IntelliSense muestra tipos disponibles

El mayor beneficio de proyectos compartidos es que permiten la escritura de código específico de la plataforma sin necesidad de utilizar patrones tales como el localizador de servicios, como lo hace con PCLs. Esto se logra usando las directivas de preprocesador como # Si, #elif, #más y los símbolos de compilación condicional como se demuestra en Listado de Código 1.

Listado de Código 1

```
privado cuerda GetFolderPath () {  
  
    cuerda path = "";  
  
    # Si __ANDROIDE__  
    path = Environment.GetFolderPath (nts Environment.SpecialFolder.MyDocume);  
  
    # elif __IOS__  
    path = Environment.GetFolderPath (nts Environment.SpecialFolder.MyDocume);  
  
    # elif TELEFONO WINDOWS  
    path = Windows.Storage.KnownFolders.DocumentsLibrary.Path;  
    #terminara si  
    regreso camino; }
```

Como se puede ver, sólo tiene que comprobar la plataforma de su aplicación se está ejecutando en las directivas del preprocesador y luego el compilador resolverá el código específico de la plataforma adecuada sin tener que lidiar con la complejidad de los patrones de otros. Cada plataforma está representada por un símbolo de la compilación condicional, definido en las opciones de generación de las propiedades del proyecto. La Tabla 1 resume los símbolos disponibles en Visual Studio 2017 y las plataformas que representan.

Tabla 1: símbolos de compilación condicional en Xamarin.Forms

Símbolo	Descripción
__ANDROIDE__	Representa la plataforma Android.
__IOS__	Representa la plataforma iOS.
TELEFONO WINDOWS	Representa la Plataforma Universal de Windows, Windows 8.1 y Windows Phone 8.1 plataformas.
__TVOS__	Representa la plataforma de Apple TVOS.
__WATCHOS__	Representa la plataforma de Apple Seguir este sistema operativo.
NETCORE_FX	Representa la plataforma .NET Core.

En las versiones de Visual Studio que todavía soportan Windows Phone 8 proyectos, como Visual Studio 2013 y 2015, un símbolo adicional llamada **SILVERLIGHT** está disponible, pero no está dirigido en Visual Studio 2017. Un ejemplo interesante de código específico de la plataforma que utiliza símbolos de compilación condicional y las directivas de preprocesador es el [SQLite.cs](#) archivo, que implementa el acceso de datos contra la base de datos SQLite popular en C#. Una solución de la muestra completa basada en **proyectos compartidos y el enfoque descrito anteriormente está disponible en la documentación oficial Xamarin y se llama Tasky**. Muestra cómo crear una sencilla aplicación móvil de tareas. Tener la opción de escribir código específico de la plataforma con el enfoque que viste en Listado de Código 1 es sin duda atractiva, pero se debe preferir PCLs (y bibliotecas .NET Standard) en al menos las siguientes situaciones:

- Es necesario tener acceso a muchos recursos específicos de la plataforma y su código puede llegar a ser muy difícil de mantener.
- Es necesario implementar arquitecturas basadas en uno o más patrones. En tales situaciones, no sólo se comparten proyectos no es la mejor opción, pero es común tener varias bibliotecas portátiles, lo que también hace que sea más fácil para los equipos que trabajan en diferentes partes de una solución.
- ¿Quieres unidad de probar el código de manera eficiente.

Los puntos antes mencionados, junto con las consideraciones que hice en la sección acerca de las bibliotecas portátiles, deben aclarar aún más la razón que usted encontrará ejemplos basados en PCLs en lugar de proyectos compartidos en este libro electrónico.

Compartir código con bibliotecas .NET Standard

Los [.NET Standard](#) proporciona un conjunto de especificaciones formales para las API que todas las plataformas de desarrollo .NET, tales como .NET Framework, .NET Core, y Mono, debe implementar. Esto permite la unificación de las plataformas .NET y evita la fragmentación futuro. Mediante la creación de una biblioteca .NET Standard, que se asegurará de que su código se ejecutará en cualquier plataforma .NET sin la necesidad de seleccionar cualquier blanco. Esto también resuelve un problema común con las bibliotecas portátiles, ya que cada biblioteca portátil puede dirigirse a un conjunto diferente de las plataformas, lo que implica una incompatibilidad potencial entre las bibliotecas y los proyectos. Microsoft tiene una interesante [entrada en el blog sobre .NET Standard, sus objetivos y sus implementaciones](#), que aclarará cualquier duda acerca de esta especificación.

En el momento de escribir esto, la versión 1.7 de .NET estándar está disponible y completa la unificación de .NET Framework, .NET Core, y se espera Mono para la versión 2.0 (actualmente disponible en la vista previa). Por ahora, Visual Studio 2017 es compatible con las versiones hasta 1.7. La documentación le ayudará a elegir la versión de .NET estándar basado en la versión mínima de la plataforma de su aplicación está destinada a funcionar en. Con Xamarin.Forms en Visual Studio 2017, la versión 1.3 es la opción que recomiendo. Visual Studio 2017 no incluye .NET estándar como una estrategia de intercambio de código al crear una nueva solución Xamarin.Forms, y esto tiene sentido, ya que, como descubrirán en breve, por el momento, sólo las versiones preliminares de apoyo Xamarin.Forms. Estándar NET. Una vez que las versiones estables de Xamarin.Forms soportan .NET Standard, es razonable esperar que esta estrategia de código compartido se incluya como una opción posible. Lo que puede hacer ahora es convertir un proyecto de biblioteca de clases portátil en una biblioteca .NET estándar. Para lograr esto, es necesario seguir estos pasos:

1. En el Explorador de soluciones, haga clic en el nombre del proyecto PCL y seleccione **Manejo de NuGet Paquetes**.

2. En la interfaz del gestor de paquetes NuGet, verá instalado en el proyecto actual del paquete Xamarin.Forms. Usted tiene que desinstalar este paquete en este punto, porque de lo contrario Visual Studio no será capaz de cambiar el PCL a una biblioteca .NET estándar.
3. Abra las propiedades del proyecto PCL y, en el **Biblioteca** pestaña, haga clic en el **Plataforma .NET objetivo** **Estándar** hipervínculo en la parte inferior de la lista de plataformas dirigidos actualmente. Un mensaje de confirmación aparecerá. Hacer clic **Sí**.
4. Como se puede ver en la figura 18, el proyecto apunta ahora a la versión estándar .NET 1.0. Cambiar a la versión 1.3 y clic **Sí** cuando aparece un mensaje diciendo que el proyecto debe ser recargada.
5. En el Explorador de soluciones, haga clic en el nombre de la solución y seleccione **Administrar paquetes NuGet**. En la interfaz NuGet, seleccione **Vistazo** y buscar el paquete Xamarin.Forms NuGet. Asegúrate que **Incluir preliminar** se selecciona bandera.
6. Haga clic en el **Consolidar** pestaña, seleccionar todos los proyectos en la lista de la derecha, a continuación, seleccione la más alta posible versión preliminar del paquete NuGet. Por último, haga clic **instalar** (véase la Figura 19).

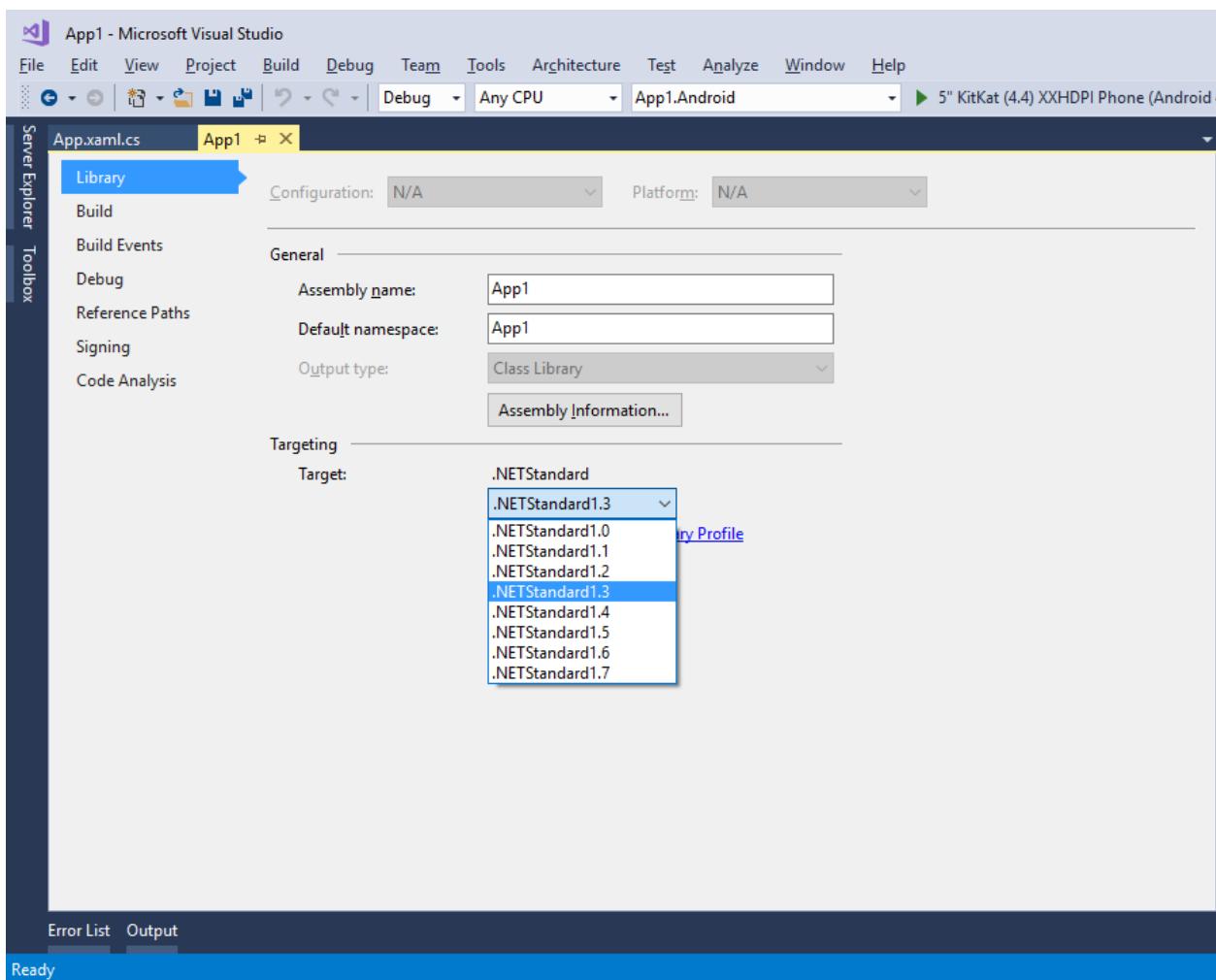


Figura 18: Selección de una versión diferente de .NET Standard

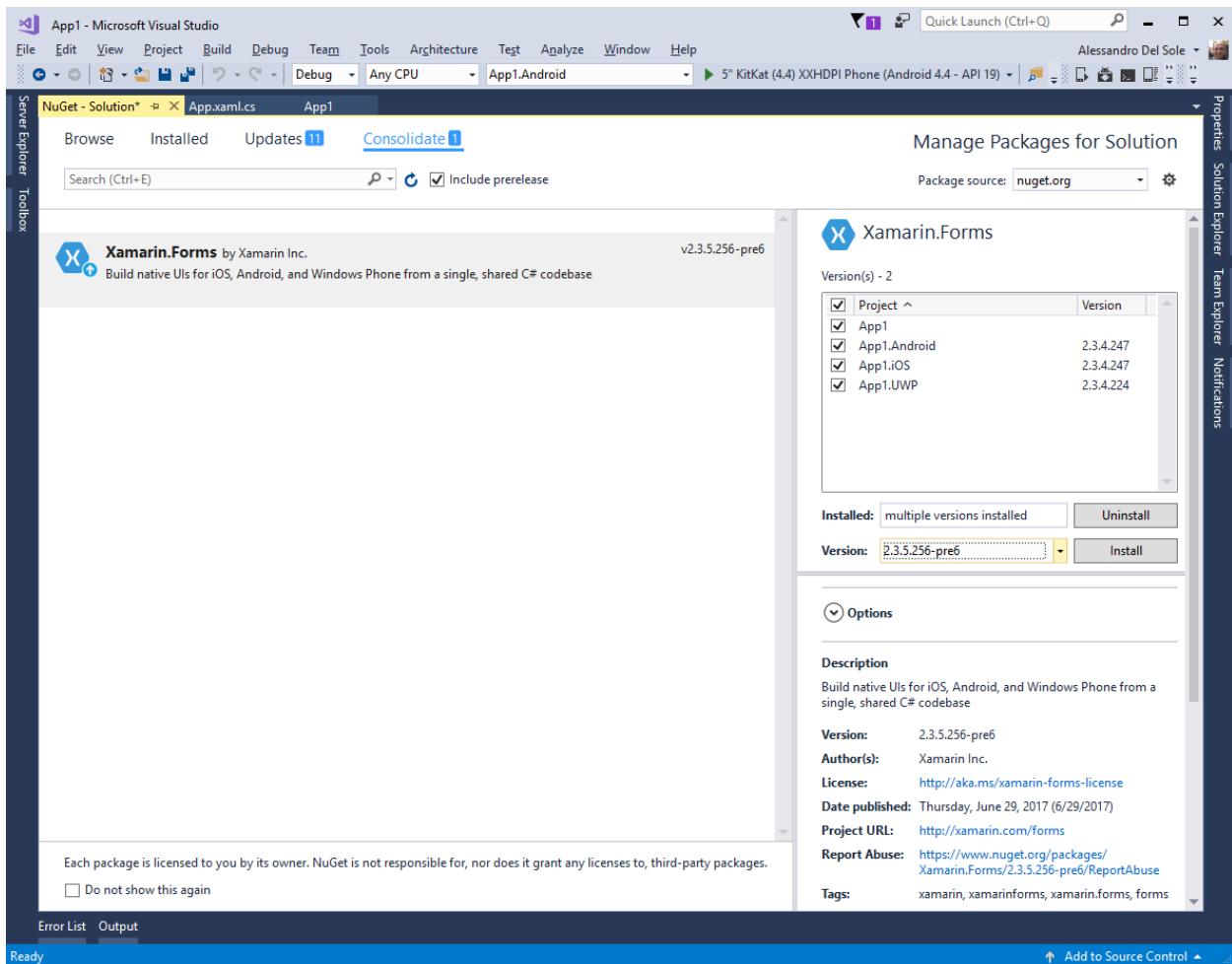


Figura 19: Volver a instalar una versión actualizada de Xamarin.Forms

Al instalar la última versión preliminar de Xamarin.Forms es necesario porque no todas las versiones de este paquete de soporte de .NET estándar, y algunas de las versiones más recientes sólo son compatibles con las versiones más bajas de .NET estándar. Por lo tanto, si desea orientar .NET Standard 1.3, la versión mínima del Xamarin.Forms que es necesario instalar es 2.3.5.256-pre6. Por supuesto, las futuras versiones estables serán compatibles con .NET Standard 1.3 y versiones superiores.

En este punto, reconstruir su solución. Ahora usted tiene una biblioteca estándar de .NET que pueden contener código que sin duda funcionar en todas las plataformas que implementan la especificación, que incluyen mono, UWP, Xamarin.iOS, Xamarin.Android y Xamarin.Mac.



Nota: Aunque .NET estándar aún no se incluye como una estrategia de compartición de códigos al crear una solución Xamarin.Forms, no hay duda de que esta será la estrategia de código compartido de elección en un futuro próximo. Por esta razón, te recomiendo echar un vistazo a la [documentación](#) y tomar algún tiempo para explorar las especificaciones API.

Resumen del capítulo

Este capítulo introduce las estrategias de código compartido disponibles que Xamarin.Forms pueden utilizar para compartir archivos de interfaz de usuario y el código independiente de la plataforma, tales como bibliotecas portátiles de las clases, proyectos compartidos y bibliotecas .NET estándar. PCLs producen montajes reutilizables, permite la aplicación de mejores arquitecturas, y no pueden contener código específico de la plataforma. proyectos compartidos se contienen código específico de la plataforma con las directivas de preprocesador y símbolos de compilación condicional, pero que no producen montajes reutilizables, y el mantenimiento del código es más difícil si tienen acceso a muchos recursos nativos. bibliotecas .NET estándar representan el futuro de código compartido a través de plataformas, se basan en un conjunto formal de las especificaciones API, y se aseguran de que su código se ejecutará en todas las plataformas que soportan la versión seleccionada de .NET estándar.

Capítulo 3 Creación de la interfaz de usuario con XAML

Xamarin.Forms es, en esencia, una biblioteca que permite crear interfaces de usuario nativa de una única base de código C # mediante el intercambio de código. Este capítulo proporciona las bases para la construcción de la interfaz de usuario en una solución Xamarin.Forms. Luego, en los próximos tres capítulos, aprenderá con más detalle acerca de los diseños, controles, páginas y la navegación.

La estructura de la interfaz de usuario en Xamarin.Forms

El mayor beneficio de Xamarin.Forms es que se puede definir toda la interfaz de usuario de la aplicación dentro del proyecto que ha seleccionado para el intercambio de código, que puede ser (por el momento) o bien un PCL o un proyecto compartido. Las aplicaciones nativas para iOS, Android y Windows que se obtiene cuando se genera una solución hará que la interfaz de usuario con los diseños nativos adecuados y controles en cada plataforma. Esto es posible porque Xamarin.Forms mapas controles nativos en clases de C # que luego se encarga de representar el elemento visual apropiado dependiendo de la plataforma de la aplicación se está ejecutando. Estas clases representan en realidad los elementos visuales tales como páginas, diseños y controles.

Debido a que el PCL o proyecto compartido sólo pueden contener código que sin duda funcionar en todas las plataformas, Xamarin.Forms mapas sólo aquellos elementos visuales comunes a todas las plataformas. Por ejemplo, iOS, Android y Windows, proporcionan cuadros de texto y etiquetas, por lo que puede proporcionar la Xamarin.Forms

Entrada y Etiqueta controles que representan cuadros de texto y etiquetas, respectivamente. Sin embargo, cada plataforma hace y gestiona los elementos visuales de forma diferente una de otra, con diferentes propiedades y comportamiento. Esto implica que los controles en Xamarin.Forms exponen sólo las propiedades y eventos que son comunes a todas las plataformas, tales como el tamaño de fuente y el color del texto.

En el capítulo 8, aprenderá cómo utilizar los controles nativos directamente, pero por ahora vamos a centrarnos en cómo Xamarin.Forms permite la creación de interfaces de usuario con elementos visuales proporcionados fuera de la caja. La interfaz de usuario de iOS, Android y Windows tiene una estructura jerárquica hecha de páginas que contienen los diseños que contienen controles. Los diseños pueden ser considerados como contenedores de controles que permiten disponer dinámicamente la interfaz de usuario de diferentes maneras. Sobre la base de esta consideración, Xamarin.Forms proporciona un número de tipos de páginas, diseños y controles que se pueden representar en cada plataforma. Cuando se crea una solución Xamarin.Forms, si usted elige un PCL o un proyecto compartido, el proyecto ha seleccionado para compartir código contendrá una página raíz que se puede llenar con elementos visuales. A continuación, se puede diseñar una interfaz de usuario más compleja mediante la adición de otras páginas y elementos visuales. Para lograr esto, puede utilizar C # y el lenguaje extensible de marcado de aplicaciones (XAML). Vamos a discutir las dos cosas más.

Codificación de la interfaz de usuario en C

En Xamarin.Forms, puede crear la interfaz de usuario de una aplicación en C # código. Por ejemplo, Listado de Código 2 se muestra cómo crear una página con un diseño que organiza los controles en una pila que contiene una etiqueta y un botón. Por ahora, no se centran en los nombres de elementos y sus propiedades (que se explicarán en el siguiente capítulo). Por el contrario, se centran en la jerarquía de los elementos visuales que introduce el código.

Listado de Código 2

```
var newPage = nuevo Pagina de contenido ();
newPage.Title = "Nueva pagina";

var newLayout = nuevo StackLayout (); newLayout.Orientation = StackOrientation .Vertical;
newLayout.Padding = nuevo Espesor (10);

var newLabel = nuevo Etiqueta (); newLabel.Text = "Bienvenido a
Xamarin.Forms!" ;

var newButton = nuevo Botón (); newButton.Text = "Pulse aquí" ;
newButton.Margin = nuevo Espesor (0, 10, 0, 0);

newLayout.Children.Add (newLabel);
newLayout.Children.Add (newButton);

newPage.Content = newLayout;
```

Aquí tienes el pleno apoyo de IntelliSense. Sin embargo, como se puede imaginar, crear una interfaz de usuario compleja enteramente en C # puede ser un reto para al menos las siguientes razones:

- Que representa una jerarquía visual hecha de toneladas de elementos de código C # es extremadamente difícil.
- Debe escribir el código de una manera que le permita distinguir entre la definición de la interfaz de usuario y otro código imperativo.
- Como consecuencia de ello, el C # se vuelve mucho más complejo y difícil de mantener.

En los primeros días de Xamarin.Forms, la definición de la interfaz de usuario sólo podría hacerse en código C #. Afortunadamente, ahora tiene una forma mucho más versátil del diseño de la interfaz de usuario con XAML, como aprenderá en la siguiente sección. Obviamente, todavía hay situaciones en las que pueda necesitar para crear elementos visuales en C #; por ejemplo, si es necesario agregar nuevos controles en tiempo de ejecución, aunque este es el único escenario para el que le sugiero que codificar los elementos visuales en C #.

La forma moderna: el diseño de la interfaz de usuario con XAML

XAML es el acrónimo de *el lenguaje extensible de marcado de aplicaciones*. Como su nombre lo indica, XAML es un lenguaje de marcado que se puede utilizar para escribir la definición de la interfaz de usuario de una manera declarativa. XAML no es nuevo en Xamarin.Forms, desde que se introdujo por primera vez hace más de diez años con Windows Presentation Foundation, y siempre ha estado disponible en plataformas como Silverlight, Windows Phone, y la plataforma Windows universal. XAML deriva de XML y, entre otros, ofrece los siguientes beneficios:

- XAML hace que sea fácil para representar estructuras de elementos de una manera jerárquica, donde páginas, diseños y controles se representan con elementos XML y propiedades con atributos XML.
- Se proporciona una separación limpia entre la definición de interfaz de usuario y la lógica # C.
- Al ser un lenguaje declarativo separado de la lógica, que permite a los diseñadores profesionales para trabajar en la interfaz de usuario sin interferir con el código imperativo.

La forma de definir la interfaz de usuario con XAML se unifica todas las plataformas, lo que significa que el diseño de la interfaz de usuario una vez y se ejecutará en iOS, Android y Windows.



Nota: XAML en Xamarin.Forms adhiere a XAML 2009 especificaciones de Microsoft, pero su vocabulario es diferente de XAML en otras plataformas, como WPF o UWP. Por lo tanto, si usted tiene experiencia con estas plataformas, se dará cuenta de muchas diferencias en cómo los elementos y sus propiedades visuales son nombrados. Microsoft está trabajando en vocabularios XAML unificadores, como aprenderá en la sección [Consejos para XAML Estándar](#). Además, recuerde que XAML entre mayúsculas y minúsculas para los nombres de objetos y sus propiedades y miembros.

Por ejemplo, cuando se crea una solución Xamarin.Forms, se puede encontrar un archivo en el proyecto PCL llamada MainPage.xaml, cuyo formato se representa en Listado de Código 3.

Listado de Código 3

```
<? Xml version = "1.0" encoding = "UTF-8"?> < Pagina de contenido xmlns =  
"http://xamarin.com/schemas/2014/forms"  
    xmlns : x = "Http://schemas.microsoft.com/winfx/2009/xaml"  
    xmlns : local = "CLR-espacio de nombres: App1"  
    x : Clase = "App1.MainPage">  
  
    < Etiqueta Texto = "Bienvenido a las formas Xamarin!"  
        VerticalOptions = "Centro"  
        HorizontalOptions = "Center" />  
  
</ Pagina de contenido >
```

Un archivo XAML en Xamarin.Forms contiene normalmente una página o una vista personalizada. El elemento raíz es una **Pagina de contenido** objeto, que representa su contraparte clase C # y se representa como una página individual. En XAML, el **Contenido** propiedad de una página está implícito, lo que significa que no es necesario escribir una **ContentPage.Content** elemento. El compilador asume que los elementos visuales que encierran entre el **Pagina de contenido** etiquetas se asignan a la **ContentPage.Content** propiedad.

los **Etiqueta** elemento, por el contrario, representa la **Etiqueta** clase en C#. Propiedades de esta clase se asignan con los atributos XML, como **Texto**, **VerticalOptions**, y **HorizontalOptions**.

Es probable que ya tiene la percepción inmediata de una mejor organización y la representación visual de la estructura de la interfaz de usuario. Si nos fijamos en el elemento raíz, se puede ver una serie de atributos cuya definición se inicia con **xmlns**. Estos se conocen como espacios de nombres XML y son importantes porque hacen posible declarar elementos visuales definidos dentro de los espacios de nombres específicos o esquemas XML. Por ejemplo, **xmlns** señala al espacio de nombres XAML raíz definido dentro de un esquema XML específico y permite la adición a la definición de interfaz de usuario todos los elementos visuales definidas por Xamarin.Forms; **xmlns:x** apunta a un esquema XML que expone los tipos predefinidos; y **xmlns:locales** apunta a la asamblea de la aplicación, por lo que es posible el uso de objetos definidos en el proyecto.

Cada página o el diseño sólo puede contener un elemento visual. En el caso de la página MainPage.xaml autogenerado, no se puede agregar otros elementos visuales de la página a menos a organizarlos en un diseño. Por ejemplo, si desea añadir un **botón debajo de la Etiqueta**, que tendría que incluir tanto el **Etiqueta** y el **Botón** dentro de un contenedor tal como el **StackLayout**, como se demuestra en Listado de Código 4.



Consejo: IntelliSense le ayudará a agregar elementos visuales más rápido viendo los nombres de elementos y propiedades a medida que escribe. A continuación, puede simplemente pulsar la tecla Tab o haga doble clic para insertar rápidamente un elemento. Si usted tiene experiencia existente con Xamarin.Forms en Visual Studio 2015, se dará cuenta de un gran número de pequeñas pero significativas mejoras de IntelliSense para XAML en VS 2017.

Listado de Código 4

```
<? Xml version = "1.0" encoding = "UTF-8"?> < Pagina de contenido xmlns =
"http://xamarin.com/schemas/2014/forms"
    xmlns : x = "Http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns : local = "CLR-espacio de nombres: App1"
    x : Clase = "App1.MainPage">

    < StackLayout Orientación = "Vertical" Relleno = "10">
        < Etiqueta Texto = "Bienvenido a las formas Xamarin!" 
            VerticalOptions = "Centro"
            HorizontalOptions = "Center" />

        < Botón x : Nombre = "Button1" Texto = "Toque aquí!"
            Margen = "0,10,0,0" /> </ StackLayout
    >

</ Pagina de contenido >
```

Si usted no incluyó ambos controles dentro de la disposición, Visual Studio generará un error. Puede anidar otros diseños dentro de un **diseño de matriz y crear jerarquías complejas de elementos visuales**. Note la **x: Nombre** asignación para el Botón. En términos generales, la **x: Nombre** puede asignar un identificador a cualquier elemento visual para que pueda interactuar con ella de código C #, por ejemplo, si usted necesita para recuperar un valor de propiedad.

Si usted nunca ha visto antes XAML, puede que se pregunte cómo puede interactuar con los elementos visuales en C # en este punto. En el Explorador de soluciones, si expande la **MainPage.xaml** archivo, verá un archivo llamado anidado **MainPage.xaml.cs**. Este es el llamado archivo de código subyacente, y contiene todo el código imperativo para la página actual. En este caso, la forma más simple de un archivo de código subyacente, el código contiene la definición de la **Página principal** clase, que hereda de

Página de contenido, y el constructor de la página, lo que hace una invocación al **InitializeComponent** método de la clase base e inicializa la página. Va a acceder al archivo de código subyacente a menudo desde el Explorador de soluciones, pero Visual Studio 2017 introduce otra manera fácil de que está relacionado con un requisito muy común: responder a eventos generados por la interfaz de usuario.

En respuesta a eventos

Los eventos son fundamentales para la interacción entre el usuario y la aplicación, y los controles en Xamarin.Forms provocar eventos como ocurre normalmente en cualquier plataforma. Los eventos se manejan de la C # archivo de código subyacente. Visual Studio 2017 hace que sea mucho más sencillo para crear controladores de eventos que sus predecesores con una experiencia evolucionada IntelliSense. Por ejemplo, supongamos que desea realizar una acción cuando el usuario pulsa el botón definido en el código anterior. los **Botón**

Control expone un evento llamado **hecho clic** que se asigna el nombre de un controlador de eventos de la siguiente manera:

```
<Botón x: Name = texto "Button1" = "Toque aquí!" Margen = "0,10,0,0" Seguido =
"Button1_Clicked" />
```

Sin embargo, cuando se escribe **Seguido =**, Visual Studio ofrece un acceso directo que permite la generación de un controlador de eventos en C # basado en el nombre del control, como se muestra en la Figura 20.

The screenshot shows the Microsoft Visual Studio interface for a Xamarin Forms project named "App1". The main window displays the XAML code for the MainPage.xaml file. The code defines a ContentPage with a StackLayout containing a Label and a Button. The Button has its name set to "Button1" and its text set to "Tap here!". The Clicked event of the Button is highlighted with a yellow selection bar, and a tooltip appears stating: "<New Event Handler> Bind event to a newly created method called 'Button1_Clicked'. Use 'Go To Definition' to navigate to the newly created method." The status bar at the bottom indicates the current zoom level is 176%, and the cursor is positioned at Ln 13, Col 27, Ch 27, with the INS key active.

```
<?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:local="clr-namespace:App1"
    x:Class="App1.MainPage">

    <StackLayout Orientation="Vertical">
        <Label Text="Welcome to Xamarin Forms!" 
            VerticalOptions="Center" 
            HorizontalOptions="Center" />

        <Button x:Name="Button1" Text="Tap here!" 
            Clicked=" "/>
    </StackLayout>
</ContentPage>
```

Figura 20: Generación de un controlador de eventos

Si pulsa **Lengüeta**, Visual Studio insertará el nombre del nuevo controlador de eventos y generará el controlador de eventos C # en el código subyacente. Puede ir rápidamente al controlador de eventos pulsando el botón derecho de su nombre y seleccionando **Ir a definición**. Usted será redirigido a la definición controlador de eventos en el C # de código subyacente, como se muestra en la Figura 21.

```
App1 - Microsoft Visual Studio
File Edit View Project Build Debug Team Data Lake Tools Architecture Test Analyze Window Help
Alessandro Del Sole
Quick Launch (Ctrl+Q) Debug Any CPU App1.Android 5" KitKat (4.4) XXHDPI Phone (Android 4.4 - API 19)
Server Explorer Toolbox
MainPage.xaml.cs* packages.config MainPage.xaml* App.xaml.cs
App1 App1.MainPage
Properties Solution Explorer Team Explorer Notifications
8     namespace App1
9     {
10    public partial class MainPage : ContentPage
11    {
12        public MainPage()
13        {
14            InitializeComponent();
15        }
16
17        private void Button1_Clicked(object sender, EventArgs e)
18        {
19        }
20    }
21
22
23
```

Figura 21: La definición controlador de eventos en C #

En este punto, usted será capaz de escribir el código que realiza la acción que desea ejecutar, tal y como sucede con otras plataformas como .NET WPF o UWP. En términos generales, las firmas de eventos manipuladores requieren dos parámetros, uno de tipo **objeto** que representa el control que provoca el evento, y un objeto de tipo **EventArgs** que contiene información sobre el evento. En muchos casos, los controladores de eventos trabajan con versiones derivadas de **EventArgs**, pero éstos se resaltarán cuando sea apropiado. Como se puede imaginar, Xamarin.Forms expone eventos que están comúnmente disponibles en todas las plataformas soportadas.

convertidores de tipos comprensión

Si nos fijamos en Listado de Código 3, se verá que la **Orientación** propiedad de la **StackLayout** es de tipo **StackOrientation**, el **Relleno** propiedad es de tipo **Espesor**, y el **Margen** propiedad asignada a la **Botón** es también de tipo **Espesor**. Sin embargo, como se puede ver en el Listado de Código 4, las mismas propiedades son asignados con valores pasados en forma de cadenas en XAML. Xamarin.Forms (y todas las otras plataformas basadas en XAML) poner en práctica la llamada **convertidores de tipos**, que convertir automáticamente una cadena en el valor adecuado para una serie de tipos conocidos. Resumiendo aquí todos los convertidores de tipos disponibles y los tipos de blancos conocidos no es posible ni necesario en este punto; sólo hay que recordar que, en la mayoría de los casos, las cadenas que asigne como valor de las propiedades se convierten automáticamente en el tipo apropiado en su nombre.

Xamarin.Forms previewer

Xamarin.Forms no tiene un diseñador que le permite dibujar la interfaz de usuario visualmente con el ratón, la caja de herramientas y ventanas interactivas como usted está acostumbrado a hacer con plataformas como WPF, Windows Forms, y UWP. Esto implica que tiene que escribir todos sus XAML manualmente. Sin embargo, Visual Studio 2017 trae una adición importante, conocida como la vista previa de Xamarin.Forms. Esta es una ventana de herramientas se habilita con **Ver, Otras ventanas de vista previa de Xamarin.Forms**, y muestra una vista previa de la interfaz de usuario en tiempo real, a medida que edita su XAML. La Figura 22 muestra la Xamarin.Forms Previewer en acción.

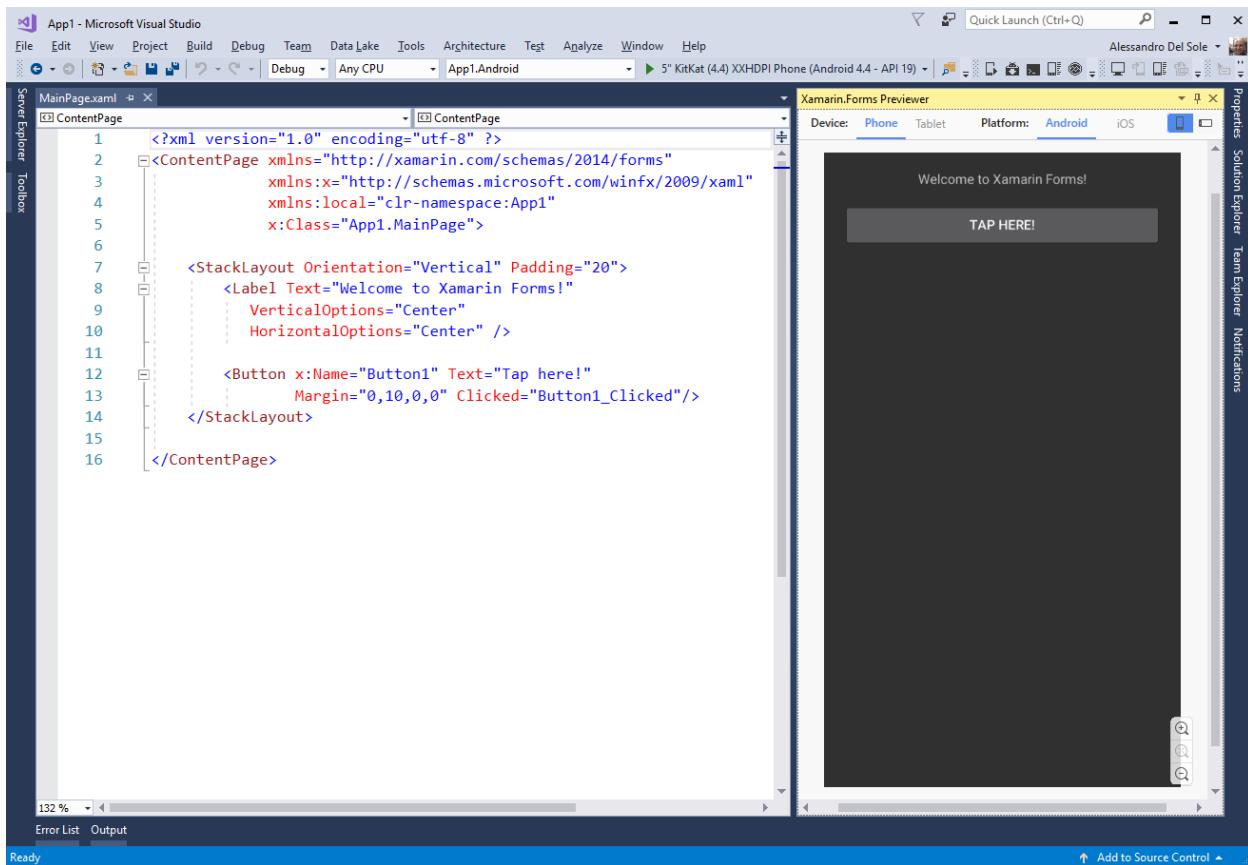


Figura 22: El Xamarin.Forms Previewer



Consejo: Recuerde que debe reconstruir su solución antes de abrir la vista previa de Xamarin.Forms por primera vez.

En la esquina inferior derecha, la vista previa proporciona controles de zoom. En la parte superior, se puede seleccionar el factor de dispositivo (teléfono o tableta), la plataforma que sirve para hacer la vista previa (Android o iOS) y la orientación (vertical u horizontal). Si desea hacer la vista previa basado en IOS, recuerde que necesita Visual Studio para ser conectado a un Mac. Si hay algún error en su XAML o si, por cualquier razón, la vista previa no es capaz de hacer que la vista previa, se mostrará un mensaje de error detallado. La vista previa de Xamarin.Forms es una adición importante, ya que evita la necesidad de ejecutar la aplicación cada vez que realice cambios significativos en la interfaz de usuario, como se requiere en

el pasado. En los próximos capítulos, voy a menudo utilizar la vista previa para demostrar cómo se ve la interfaz de usuario en lugar de correr el emulador.

Consejos para XAML Estándar

XAML en Xamarin.Forms sigue el Microsoft XAML 2009 especificaciones, pero su vocabulario es diferente de otras plataformas basadas en XAML. Por ejemplo, un cuadro de texto está representado por la **Caja de texto** el control de WPF y UWP, pero en Xamarin.Forms tiene una **Entrada**. Una vez más, la **Botón** el control de WPF y uwp expone un evento llamado **Hacer clic**, que en cambio se llama **hecho clic** en Xamarin.Forms. Recientemente, Microsoft ha anunciado [XAML Estándar](#), Una unificación de dialectos XAML a través de plataformas. XAML estándar está todavía en sus primeras etapas, por lo que no está disponible todavía. Sin embargo, se puede seguir el progreso en GitHub y se puede leer una introducción [entrada en el blog](#) que explica los objetivos de XAML estándar con más detalle.

Resumen del capítulo

Compartir la interfaz de usuario a través de plataformas es el objetivo principal de Xamarin.Forms y este capítulo proporciona una descripción de alto nivel de cómo se define la interfaz de usuario con XAML, basada en una jerarquía de elementos visuales. Usted ha visto cómo agregar elementos visuales y de cómo asignar sus propiedades; usted ha visto cómo el tipo de convertidores permiten pasar los valores de cadena en XAML y cómo el compilador los convierte en los tipos apropiados; y ha tenido un primer vistazo a la vista previa de Xamarin.Forms para conseguir un tiempo real, la representación integrada de la interfaz de usuario a medida que edita su XAML. Después de esta visión general de cómo la interfaz de usuario se define en Xamarin.Forms, es el momento para discutir importantes conceptos de interfaz de usuario con más detalle, y vamos a empezar por la organización de la interfaz de usuario con diseños.

Capítulo 4 La organización de la interfaz de usuario con formatos

Los dispositivos móviles, tales como teléfonos, tabletas y computadoras portátiles tienen diferentes tamaños de pantalla y factores de forma. También apoyan ambas orientaciones horizontal y vertical. Por lo tanto, la interfaz de usuario en aplicaciones móviles debe adaptarse dinámicamente al sistema, la pantalla, y el dispositivo de modo que los elementos visuales pueden cambiar de tamaño o reordenar automáticamente en función del factor de forma y la orientación del dispositivo. En Xamarin.Forms, esto se logra con diseños, que es el tema de este capítulo.

Comprender el concepto de diseño



Consejo: Si usted tiene experiencia previa con WPF o UWP, el concepto de diseño es el mismo que el concepto de paneles tales como el Cuadrícula y el StackPanel.

Uno de los objetivos de Xamarin.Forms es proporcionar la capacidad de crear interfaces dinámicas que se pueden reorganizar de acuerdo con las preferencias del usuario o para el tamaño del dispositivo y la pantalla. Debido a esto, los controles en las aplicaciones móviles que construir con Xamarin no deben tener un tamaño fijo o una posición de la interfaz de usuario, excepto en un número muy limitado de escenarios. Para hacer esto posible, los controles Xamarin.Forms están dispuestos dentro de contenedores especiales, conocidos como **diseños**. Disposiciones son clases que permiten que dispone elementos visuales en la interfaz de usuario, y Xamarin.Forms proporciona muchos de ellos. En este capítulo, usted aprenderá acerca de los diseños disponibles y cómo utilizarlos para organizar los controles. Lo más importante que hay que tener en cuenta es que los controles en Xamarin.Forms tienen una lógica jerárquica; Por lo tanto, se puede anidar múltiples paneles para crear experiencias de usuario complejas. La Tabla 2 resume diseños disponibles. Luego, en las siguientes secciones, aprenderá acerca de ellos con más detalle.

Tabla 2: Distribución de en Xamarin.Forms

Diseño	Descripción
StackLayout	Le permite colocar elementos visuales cerca uno del otro en horizontal o vertical.
Cuadrícula	Permite organizar los elementos visuales dentro de las filas y columnas.
AbsoluteLayout	Una disposición colocada en una posición especificada, fija.
Disposición relativa	Una disposición cuya posición depende de las limitaciones relativas.
ScrollView	Le permite desplazarse a los elementos visuales que contiene.

Diseño	Descripción
Marco	Dibuja un borde y añade espacio alrededor del elemento visual que contiene.
ContentView	Un diseño especial que puede contener jerarquías de elementos visuales y se puede utilizar para crear controles personalizados en XAML.

Recuerde que el diseño de una sola raíz se asigna a la **Contenido** propiedad de una página, y que a continuación, diseño anidado pueden contener elementos visuales y diseños.

Las opciones de alineación y espaciamiento

Como regla general, ambos diseños y controles se pueden alinear mediante la asignación de la **HorizontalOptions** y **VerticalOptions** propiedades con uno de los valores de propiedad de la **LayoutOptions** estructura, que se resumen en la Tabla 3. Proporcionar una opción de alineación es muy común. Por ejemplo, si sólo tiene el diseño raíz en una página, tendrá que asignar **VerticalOptions** con **StartAndExpand** por lo que el diseño se pone todo el espacio disponible en la página (recuerda esta consideración cuando experimente con presentaciones y vistas en este capítulo y el siguiente).

Tabla 3: Las opciones de alineación en Xamarin.Forms

Alineación	Descripción
Centrar	Alinea el elemento visual en el centro.
CenterAndExpand	Alinea el elemento visual en el centro y se expande de sus límites para llenar el espacio disponible.
comienzo	Alinea el elemento visual a la izquierda.
StartAndExpand	Alinea el elemento visual a la izquierda y expande sus límites para llenar el espacio disponible.
Fin	Alinea el elemento visual a la derecha.
EndAndExpand	Alinea el elemento visual a la derecha y se expande sus límites para llenar el espacio disponible.

También puede controlar el espacio entre los elementos visuales con tres propiedades: **El relleno, el espaciado y Margen**, resumen en la Tabla 4.

Tabla 4: Las opciones de espaciado en Xamarin.Forms

Espaciado	Descripción
Margen	Representa la distancia entre el elemento visual actual y sus elementos adyacentes, ya sea con un valor fijo para todos los cuatro lados, o con valores separados por comas para el izquierdo, superior, derecho e inferior. Es de tipo Espesor y XAML ha construido en un convertidor de tipos para ello.
Relleno	Representa la distancia entre un elemento visual y sus elementos secundarios. Se puede ajustar con un valor fijo para todos los cuatro lados, o con valores separados por comas para el izquierdo, superior, derecho e inferior. Es de tipo Espesor y XAML ha construido en un convertidor de tipos para ello.
Espaciado	Disponible sólo en el StackLayout contenedor, que le permite ajustar la cantidad de espacio entre cada elemento secundario, con un valor predeterminado de 6,0.

Te recomiendo que pasar algún tiempo experimentando con la forma de alineación y espaciamiento opciones de trabajo con el fin de entender cómo obtener el resultado apropiado en sus interfaces de usuario.

los StackLayout

los **StackLayout** recipiente permite la colocación de los controles de cerca uno del otro, como en una pila que puede disponerse tanto horizontal como verticalmente. Al igual que con otros recipientes, la **StackLayout** puede contener paneles anidados. El código siguiente muestra cómo se puede organizar controles horizontalmente y verticalmente. Listado de Código 5 muestra un ejemplo con una raíz **StackLayout** y dos diseños anidados.

Listado de Código 5

```
<? Xml version = "1.0" encoding = "UTF-8"?> < Pagina de contenido xmlns =
"http://xamarin.com/schemas/2014/forms"
    xmlns : x = "Http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns : local = "CLR-espacio de nombres: App1"
    x : Clase = "App1.MainPage">

    < StackLayout Orientación = "Vertical">
        < StackLayout Orientación = "Horizontal" Margen = "5">
            < Etiqueta Texto = "controles de ejemplo" Margen = "5" /> < Botón Texto
            = "Botón de prueba" Margen = "5" /> </ StackLayout > < StackLayout Orientación
            = "Vertical" Margen = "5">

            < Etiqueta Texto = "controles de ejemplo" Margen = "5" />
```

```

< Botón Texto = "Botón de prueba" Margen = "5" /> </ StackLayout
>
</ StackLayout >
</ Página de contenido >

```

El resultado de la XAML en Listado de Código 5 se muestra en la Figura 23.



Figura 23: Disposición de los elementos visuales con el StackLayout

los **Orientación** propiedad se puede definir como **Horizontal** o **Vertical**, y esto influye en el diseño final. Si no se especifica, **Vertical** es el valor predeterminado. Uno de los principales beneficios de código XAML es que los nombres de los elementos y propiedades se explican por sí, y este es el caso en **StackLayout**'s propiedades, también. Recuerde que controla dentro de una **StackLayout** se redimensionan automáticamente según la orientación. Si no te gusta este comportamiento, es necesario especificar **WidthRequest**

y **HeightRequest** propiedades en cada control, que representan la anchura y la altura, respectivamente. **Espaciado** es una propiedad que se puede utilizar para ajustar la cantidad de espacio entre los elementos secundarios; esto se prefiere el ajuste del espacio en los controles individuales con el **Margen** propiedad.

los Cuadrícula

los **Cuadrícula** es uno de los diseños más fáciles de entender y, probablemente, el más versátil. Se le permite crear tablas con filas y columnas. De esta manera, se pueden definir las células y cada célula puede contener un control u otra disposición de almacenamiento de los controles anidados. los **Cuadrícula** es versátil ya que sólo se puede dividir en filas o columnas o ambos. El siguiente código define una **Cuadrícula** que se divide en dos filas y dos columnas:

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

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

RowDefinitions es una colección de **RowDefinition** objetos, y lo mismo es cierto para **ColumnDefinitions** y **ColumnDefinition**. Cada elemento representa una fila o una columna dentro de la **Cuadrícula**, respectivamente. También puede especificar una **Anchura** o una **Altura** propiedad de delimitar de fila y columna dimensiones; Si no se especifica nada, ambas filas y columnas están dimensionados en el tamaño máximo disponible. Al cambiar el tamaño del contenedor primario, filas y columnas se reorganizan automáticamente.

El código anterior crea una tabla con cuatro células. Para colocar los controles en el **Cuadrícula**, se especifica la posición de la fila y la columna a través de la **Grid.Row** y **Grid.Column** propiedades, conocido como propiedades adjuntas , En el control. **propiedades adjuntas** permiten la asignación de propiedades del contenedor primario del elemento visual actual. El índice de ambos está basado en cero, lo que significa que 0 representa la primera columna de la izquierda y la primera fila de la parte superior. Puede colocar los diseños anidados dentro de una célula o de una sola fila o columna. El código en el Listado de Código 6 muestra cómo anidar una rejilla en una cuadrícula de raíz con los controles de los niños.



Propina: *Grid.Row = "0" y Grid.Column = "0" puede ser omitido.*

Listado de Código 6

```
<? Xml version = "1.0" encoding = "UTF-8"?> < Pagina de contenido xmlns =
"http://xamarin.com/schemas/2014/forms"
  xmlns : x = "Http://schemas.microsoft.com/winfx/2009/xaml"
  x : Clase = "Layouts.GridSample"> < ContentPage.Content
> < Cuadrícula > < Grid.RowDefinitions > < RowDefinition
/> < RowDefinition /> </ Grid.RowDefinitions >
```

```
< Grid.ColumnDefinitions >< ColumnDefinition />< ColumnDefinition /></ Grid.ColumnDefinitions
> < Botón Texto = "Primer botón" /> < Botón Grid.Column = "1" Texto =
"Segundo botón" />

< Cuadrícula Grid.Row = "1">
< Grid.RowDefinitions >< RowDefinition />< RowDefinition /></ Grid.RowDefinitions
> < Grid.ColumnDefinitions >< ColumnDefinition />< ColumnDefinition
/></ Grid.ColumnDefinitions >< Botón Texto = "Botón de 3" /> < Botón
Texto = "Botón 4" Grid.Column = "1" /></ Cuadrícula >

</ Cuadrícula >
</ ContentPage.Content >
</ Pagina de contenido >
```

La Figura 24 muestra el resultado de este código.

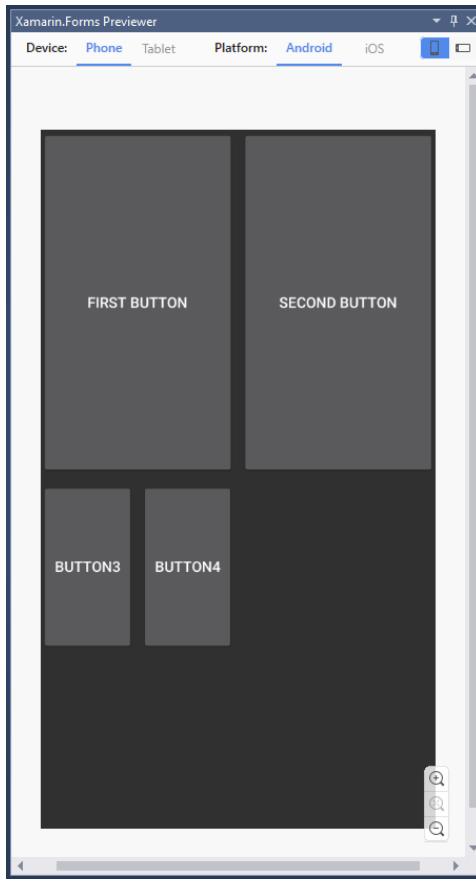


Figura 24: Disposición de los elementos visuales con la cuadrícula

los **Cuadrícula** diseño es muy versátil y es también una buena opción (cuando sea posible) en términos de rendimiento.

Espaciado y proporciones para las filas y columnas

Usted tiene el control preciso sobre el tamaño, espacio y proporciones de filas y columnas. los **Altura y Anchura** propiedades de la **RowDefinition** y **ColumnDefinition** los objetos pueden ser fijados con los valores de la **GridUnitType** enumeración como sigue:

- **Auto:** ajusta automáticamente el tamaño para encajar el contenido de la fila o columna.
- **Estrella:** Los tamaños de las columnas y filas como proporción del espacio restante.
- **Absoluto:** Tamaños columnas y filas con altura específica, fija y valores de anchura.

XAML tiene convertidores de tipos para el **GridUnitType** valores, por lo que simplemente pasa ningún valor para **Auto**, a * de **Estrella**, y el valor numérico fijo para **Absoluto**, como:

```
<Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition Anchura = "*" />
    <ColumnDefinition width = "20" />
</Grid.ColumnDefinitions>
```

La introducción de tramos

En algunas situaciones, es posible que tenga elementos que deben ocupar más de una fila o columna. En estos casos, se puede asignar el **Grid.RowSpan** y **Grid.ColumnSpan** propiedades adjuntas con el número de filas y columnas de un elemento visual debe ocupar.

los AbsoluteLayout

los **AbsoluteLayout** contenedor le permite especificar dónde exactamente en la pantalla que desea que aparezcan los elementos secundarios, así como su tamaño y límites. Hay algunas maneras diferentes para establecer los límites de los elementos secundarios basados en el **AbsoluteLayoutFlags** enumeración utilizado durante este proceso. los **AbsoluteLayoutFlags** enumeración contiene los siguientes valores:

- **Todas**: Todas las dimensiones son proporcionales.
- **HeightProportional**: La altura es proporcional a la disposición.
- **Ninguna**: Ninguna interpretación que se hace.
- **PositionProportional**: Las cosechadoras **XProportional** y **YProportional**.
- **SizeProportional**: Las cosechadoras **WidthProportional** y **HeightProportional**.
- **WidthProportional**: La anchura es proporcional a la disposición.
- **XProportional**: X propiedad es proporcional a la disposición.
- **YProportional**: Y propiedad es proporcional a la disposición.

Una vez que haya creado sus elementos secundarios, para ponerlos en una posición absoluta dentro del contenedor tendrá que asignar el **AbsoluteLayout.LayoutFlags** propiedad adjunta. También tendrá que asignar el **AbsoluteLayout.LayoutParams** propiedad adjunta para dar los elementos de sus límites. Desde Xamarin.Forms es una capa de abstracción entre Xamarin y las implementaciones específicas del dispositivo, los valores de posición puede ser independiente de los píxeles del dispositivo. Aquí es donde las banderas de diseño mencionados anteriormente entran en juego. Listado de Código 7 proporciona un ejemplo basado en dimensiones proporcionales y posición absoluta para controles secundarios.

Listado de Código 7

```
<? Xml version = "1.0" encoding = "UTF-8"?> < Pagina de contenido xmlns =
"http://xamarin.com/schemas/2014/forms"
    xmlns : x = "Http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns : local = "CLR-espacio de nombres: App1"
    x : Clase = "App1.MainPage">

    < AbsoluteLayout > < Etiqueta Texto = "Primera
etiqueta"
        AbsoluteLayout.LayoutParams = "0, 0, 0,25, 0,25"
        AbsoluteLayout.LayoutFlags = "Todos" Color de texto = "Red" /> < Etiqueta Texto =
"Segunda etiqueta"
        AbsoluteLayout.LayoutParams = "0.20, 0.20, 0.25, 0.25"
        AbsoluteLayout.LayoutFlags = "Todos" Color de texto = "Orange" /> < Etiqueta Texto =
"Tercera etiqueta"
        AbsoluteLayout.LayoutParams = "0.40, 0.40, 0.25, 0.25"
        AbsoluteLayout.LayoutFlags = "Todos" Color de texto = "Violeta" />
```

```
< Etiqueta Texto = "Cuarto Etiqueta"  
    AbsoluteLayout.LayoutBounds = "0.60, 0.60, 0.25, 0.25"  
    AbsoluteLayout.LayoutFlags = "Todos" Color de texto = "Yellow" />  
</ AbsoluteLayout >  
</ Pagina de contenido >
```

La Figura 25 muestra el resultado de la **AbsoluteLayout** ejemplo.

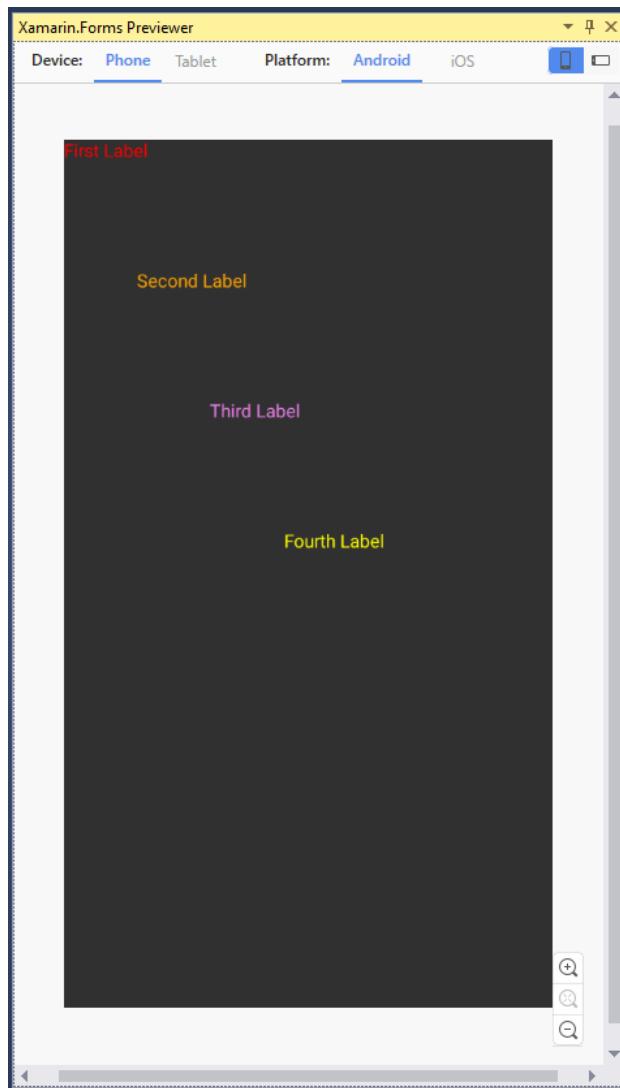


Figura 25: El posicionamiento absoluto con **AbsoluteLayout**

los Disposición relativa

los **Disposición relativa** contenedor proporciona una forma de especificar la ubicación de los elementos secundarios relativa ya sea entre sí o con el control de los padres. ubicaciones relativas se resuelven a través de una serie de **Restricción** objetos que definen la posición relativa de cada elemento de niño en particular a otro. En XAML, **Restricción** objetos se expresan a través de la **ConstraintExpression** extensión de marcado, que se utiliza para especificar la ubicación o el tamaño de un niño, vista como una constante, o en relación con un parent o otra vista con nombre. extensiones de marcado son muy comunes en XAML, y verá muchos de ellos en el capítulo 7 acerca del enlace de datos, pero discutirlos en detalle está fuera del alcance aquí. La documentación oficial tiene una muy detallada [página](#) en su sintaxis y la aplicación que os animo a leer.

En el **Disposición relativa** clase, hay propiedades con nombre **XConstraint** y **YConstraint**. En el siguiente ejemplo, verá cómo asignar un valor a estas propiedades desde el interior de otro elemento XAML, a través de las propiedades asociadas. Esto se demuestra en Listado de Código 8, donde conocer la **BoxView**, un elemento visual que le permite dibujar un cuadro de color. En este caso, es útil para darle una percepción inmediata de cómo se organiza la disposición.

Listado de Código 8

```
<? Xml version = "1.0" encoding = "UTF-8"?> < Pagina de contenido xmlns =
"http://xamarin.com/schemas/2014/forms"
    xmlns : x = "Http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns : local = "CLR-espacio de nombres: App1"
    x : Clase = "App1.MainPage">

    < Disposición relativa > < BoxView Color = "Red" x : Nombre = "REDBOX"

        RelativeLayout.YConstraint = "{ ConstraintExpression Tipo = RelativeToParent,
            Propiedad = Altura, Factor = .15 , Constante = 0 }"
        RelativeLayout.WidthConstraint = "{ ConstraintExpression
            Tipo = RelativeToParent, Propiedad = Ancho, Factor = 1 , Constante = 0 }"
        RelativeLayout.HeightConstraint = "{ ConstraintExpression
            Tipo = RelativeToParent, Propiedad = Altura, Factor = .8 , Constante = 0 }"/> < BoxView Color = "Blue"

        RelativeLayout.YConstraint = "{ ConstraintExpression Tipo = RelativeToView,
            ElementName = REDBOX, Propiedad = Y, Factor = 1 , Constante = 20 }"
        RelativeLayout.XConstraint = "{ ConstraintExpression Tipo = RelativeToView,
            ElementName = REDBOX, Propiedad = X, Factor = 1 , Constante = 20 }"

        RelativeLayout.WidthConstraint = "{ ConstraintExpression
            Tipo = RelativeToParent, Propiedad = Ancho, Factor = .5 , Constante = 0 }"
        RelativeLayout.HeightConstraint = "{ ConstraintExpression
            Tipo = RelativeToParent, Propiedad = Altura, Factor = .5 , Constante = 0 }"/> </ Disposición relativa >

</ Pagina de contenido >
```

El resultado de Listado de Código 8 se muestra en la Figura 26.

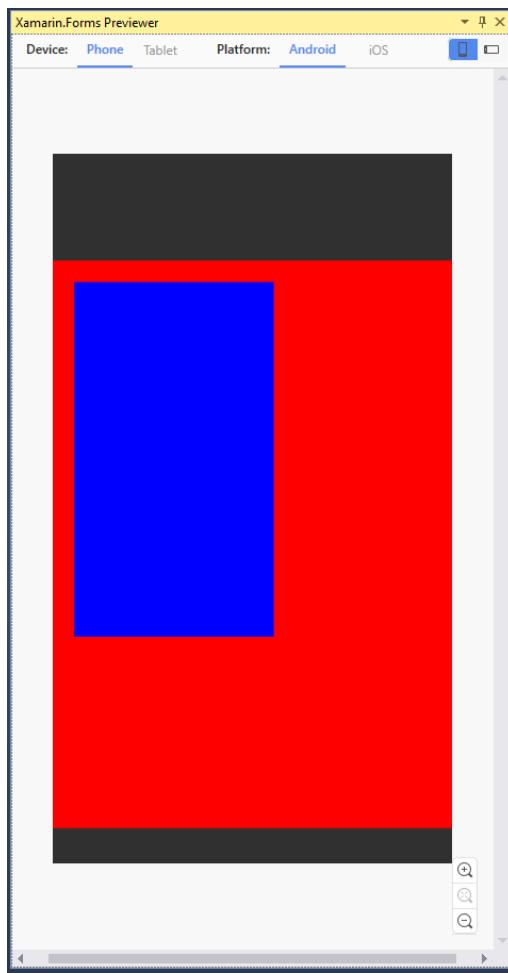


Figura 26: Disposición de los elementos visuales con el RelativeLayout



Consejo: La Disposición relativa **contenedor** tiene pobre rendimiento de la representación y la documentación recomienda que evite esta disposición siempre que sea posible, o al menos se debe evitar más de una Disposición relativa por página.

los ScrollView

El diseño especial **ScrollView** le permite presentar contenidos que no pueden caber en una pantalla y por lo tanto debe ser desplazado. Su uso es muy sencillo:

```
<ScrollView x: Name = "Scroll1">
    <StackLayout>
        <Etiquetas de texto = "Mi color favorito:" x: Name = "Label1" /> <BoxView
            BackgroundColor = "verde" HeightRequest = "600" /> </ StackLayout> </ ScrollView>
```

Es, básicamente, añadir un diseño o elementos visuales dentro de la **ScrollView** y, en tiempo de ejecución, el contenido será desplazable si su área es más grande que el tamaño de la pantalla. Además, puede especificar el **Orientación** propiedad (con valores **Horizontal** o **Vertical**) para establecer el **ScrollView** para desplazarse sólo en horizontal o sólo vertical. La razón por la disposición tiene un nombre en el uso de la muestra es que se puede interactuar con el **ScrollView** programación, invocando su **ScrollToAsync**

Método para mover su posición sobre la base de dos opciones diferentes. Tenga en cuenta las siguientes líneas:

```
ScrollView1.ScrollToAsync (0, 100, true);
```

```
ScrollView1.ScrollToAsync (Label1, ScrollToPosition.Start, true);
```

En el primer caso, el contenido en 100px desde la parte superior es visible. En el segundo caso, el **ScrollView** mueve el control especificado en la parte superior de la vista y establece la posición actual en la posición del control. Los valores posibles para la **ScrollToPosition** enumeración son:

- **Centrar:** Se desplaza por el elemento hasta el centro de la parte visible de la vista.
- **Fin:** Se desplaza por el elemento al final de la parte visible de la vista.
- **makeVisible:** Hace que el elemento visible dentro de la vista.
- **Comienzo:** Desplaza el elemento del comienzo de la parte visible de la vista.

Tenga en cuenta que nunca se debe nido **ScrollView** diseños, y nunca deben incluir la **Vista de la lista** y **WebView** controles dentro de una **ScrollView** debido a que ambos ya poner en práctica el desplazamiento.

los Marco

los **Marco** Es un diseño muy especial en Xamarin.Forms, ya que proporciona una opción para dibujar un borde de color alrededor del elemento visual que contiene, y opcionalmente añadir espacio adicional entre el **Marco** 'S límites y el elemento visual. Listado de Código 9 proporciona un ejemplo.

Listado de Código 9

```
<? Xml version = "1.0" encoding = "UTF-8"?> < Pagina de contenido xmlns =
"http://xamarin.com/schemas/2014/forms"
    xmlns : x = "Http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns : local = "CLR-espacio de nombres: App1"
    x : Clase = "App1.MainPage">

    < Marco OutlineColor = "Red" Radio de esquina = "3" HasShadow = "True"
Margen = "20">
        < Etiqueta Texto = "Etiqueta en un marco"
            HorizontalOptions = "Centro"
            VerticalOptions = "Center" />
    </ Marco >
</ Pagina de contenido >
```

los **OutlineColor** la propiedad se asigna con el color de la frontera, la **Radio de esquina** la propiedad se le asigna un valor que le permite dibujar curvas circulares, y el **HasShadow** la propiedad le permite mostrar una sombra. La Figura 27 proporciona un ejemplo basado en la versión UWP del proyecto, puesto que el Xamarin.Forms Previewer no hace los marcos de manera apropiada en algunas situaciones, tales como con temas oscuros.

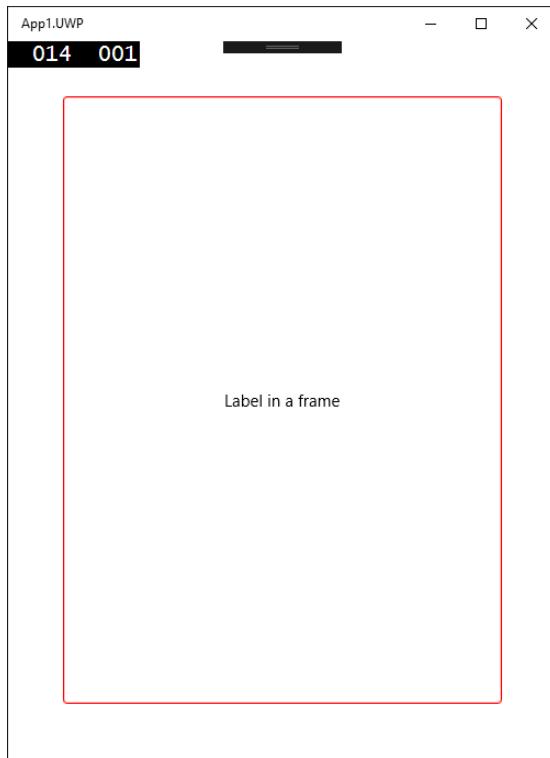


Figura 27: Dibujo de un Frame

los **Marco** se cambiará el tamaño proporcionalmente en función del tamaño del contenedor primario.

los ContentView

El recipiente especial **ContentView** permite la agregación de múltiples vistas en una única vista y es útil para crear, controles personalizados reutilizables. Porque el **ContentView** representa un elemento visual autónomo, Visual Studio hace que sea más fácil para crear una instancia de este recipiente con una plantilla de elementos específicos. En el Explorador de soluciones, puede hacer clic en el nombre de PCL y seleccione

Agregar ítem nuevo. En el **Agregar ítem nuevo** de diálogo, seleccione la **Xamarin.Forms** nodo, entonces la

Ver el contenido elemento, como se muestra en la Figura 28. Asegúrese de que no se selecciona el elemento llamado Vista de contenido (C #), de lo contrario obtendrá un nuevo archivo de clase que se necesita para llenar en código C # en lugar de XAML.

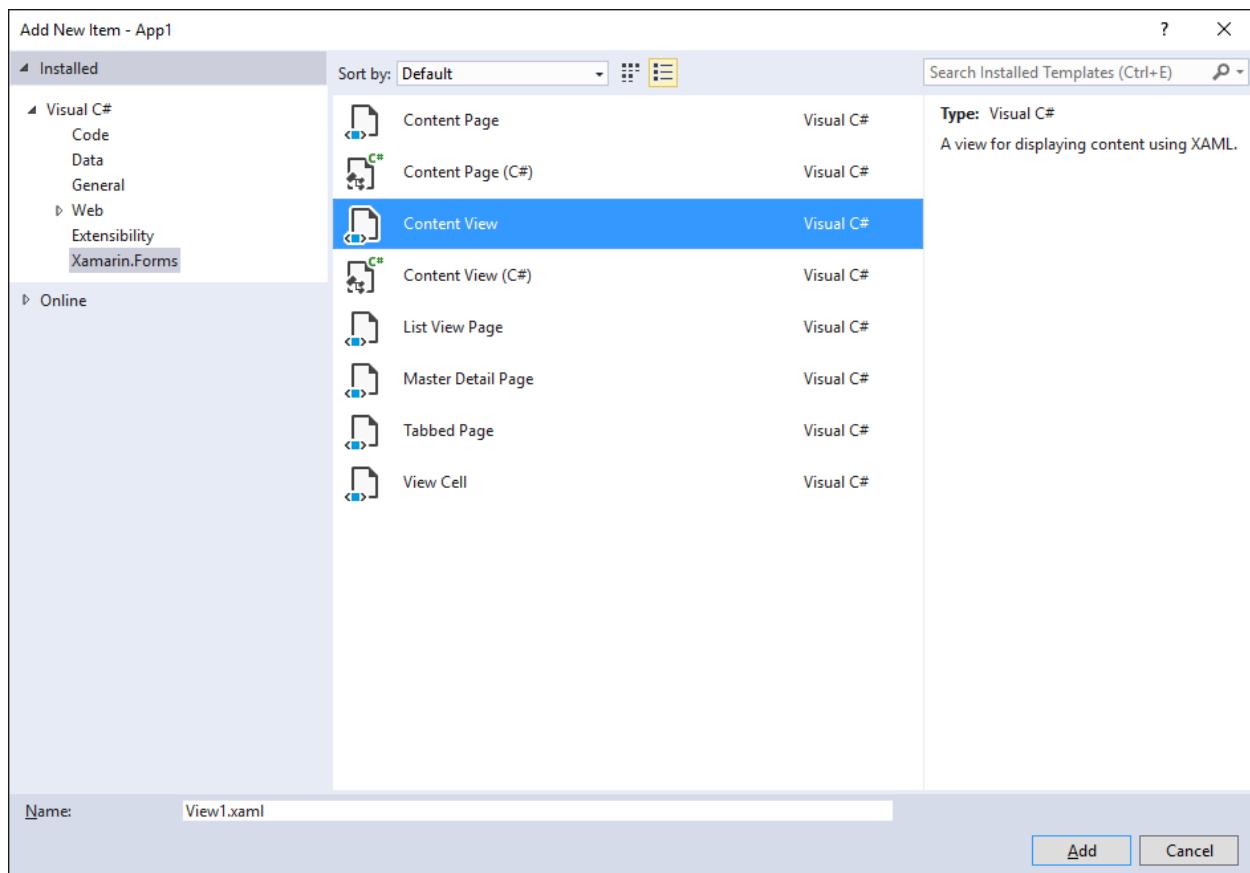


Figura 28: Adición de un ContentView

Cuando se añade el nuevo archivo al proyecto, el editor XAML muestra contenido básico del hecho **ContentView** elemento raíz y una **Etiqueta**. Puede agregar varios elementos visuales, como se muestra en Listado de Código 10, y luego se puede utilizar el **ContentView** como lo haría con un control o el diseño individual.

Listado de Código 10

```
<? Xml version = "1.0" encoding = "UTF-8"?> < ContentView xmlns =
"http://xamarin.com/schemas/2014/forms"
    xmlns : x = "Http://schemas.microsoft.com/winfx/2009/xaml"
    x : Clase = "App1.View1">

< ContentView.Content > < StackLayout > < Etiqueta Texto = "Introduzca su dirección de
correo electrónico:" /> < Entrada x : Nombre = "EmailEntry" /> < / StackLayout >

< / ContentView.Content >
< / ContentView >
```

Vale la pena mencionar que los elementos visuales dentro de una **ContentView** puede levantar y gestionar eventos y datos de soporte de unión, lo que hace que el **ContentView** muy versátil y perfecta para la construcción de vistas reutilizables.

Resumen del capítulo

Las aplicaciones móviles requieren interfaces de usuario dinámicas que pueden adaptarse automáticamente al tamaño de la pantalla de diferentes factores de forma del dispositivo. En Xamarin.Forms, la creación de interfaces de usuario dinámicas es posible a través de un número de los denominados **diseños**. Los **StackLayout** le permite organizar los controles de cerca uno de otro, tanto horizontal como verticalmente; el **Cuadrícula** le permite organizar los controles dentro de las filas y columnas; el **AbsoluteLayout** le permite dar controles una posición absoluta; el

Disposición relativa le permite organizar los controles basados en el tamaño y la posición de otros controles o contenedores; el **ScrollView** diseño le permite desplazarse por el contenido de los elementos visuales que no caben en una sola página; el **Marco** diseño le permite dibujar un borde alrededor de un elemento visual; el **ContentView** le permite crear vistas reutilizables. Ahora que tiene un conocimiento básico de los diseños, es el momento para discutir los controles comunes en Xamarin.Forms que le permiten construir las funcionalidades de la interfaz de usuario, dispuestas dentro de los diseños que ha aprendido en este capítulo.

Capítulo 5 Xamarin.Forms Común controles

Xamarin.Forms se envía con un rico conjunto de controles comunes que se pueden utilizar para construir interfaces de usuario multiplataforma con facilidad y sin la necesidad de que la complejidad de las características específicas de la plataforma. Como se puede imaginar, el beneficio de estos controles comunes es que se ejecutan en Android, iOS y Windows desde la misma base de código. En este capítulo, usted aprenderá acerca de los controles comunes, sus propiedades y sus eventos. Otros controles serán introducidos en el capítulo 7, especialmente controla cuyo propósito es que se muestran los datos.



Nota: Con el fin de seguir los ejemplos de este capítulo, crear una nueva solución Xamarin.Forms basado en la estrategia de código compartido PCL. El nombre es de usted. Cada vez que se discute un nuevo control, simplemente limpiar el contenido de la raíz Pagina de contenido objeto en el archivo XAML y eliminar cualquier código C # específicos de un solo control o añadir un nuevo archivo de tipo Pagina de contenido al proyecto.

Comprender el concepto de vista

En Xamarin.Forms, una vista es la piedra angular de cualquier aplicación móvil. Sucintamente, es una vista de un control que representa lo que podría llamarse un widget de Android, una vista en IOS, y un control en Windows. Vistas derivan de la **Xamarin.Forms.View** clase. En realidad, desde un punto de vista técnico, los diseños son vistas a sí mismos y se derivan de **Diseño**, un objeto intermedio en la jerarquía que se deriva de **Ver** e incluye una **Niños** propiedad que le permite añadir múltiples elementos visuales para el diseño en sí. El concepto de vista es también importante desde el punto de vista terminología. De hecho, en Xamarin.Forms y su documentación, usted más a menudo encontrar la palabra **ver** que **controlar**. A partir de ahora, Me va a utilizar tanto ver y controlar de manera intercambiable, pero recuerda que la documentación y tutoriales a menudo se refieren a puntos de vista.

Vistas propiedades comunes

Vistas comparten una serie de propiedades que son importantes para que usted sepa de antemano. Estos se resumen en la Tabla 5.

Tabla 5: propiedades comunes Vistas

Propiedad	Descripción
HorizontalOptions	Igual que Tabla 3 .
VerticalOptions	Igual que Tabla 3 .
HeightRequest	de tipo doble , obtiene o establece la altura de la vista.

Propiedad	Descripción
WidthRequest	de tipo doble , obtiene o establece el ancho de una vista.
Es visible	de tipo bool , determina si un control es visible en la interfaz de usuario.
Está habilitado	de tipo bool , permite la activación o desactivación de un control, manteniéndola visible en la interfaz de usuario.
GestureRecognizers	Una colección de GestureRecognizer objetos que permiten a los gestos táctiles en los controles que no son compatibles directamente táctil. Estas serán discutidas más adelante en este capítulo.



Consejo: Los controles también exponen la Margen propiedad descrita en [Tabla 4](#).

Por lo tanto, si desea cambiar la anchura o la altura de miras, recuerde que necesita el **WidthRequest** y **HeightRequest** propiedades, en lugar de **Altura** y **Anchura** que son de sólo lectura y regresar la altura y la anchura actual.

La introducción de controles comunes

Esta sección proporciona una descripción de alto nivel de los controles comunes de Xamarin.Forms y sus propiedades más utilizados. Recuerde agregar el [documentación oficial](#) acerca de la interfaz de usuario a sus marcadores para una referencia más detallada.

La entrada del usuario con el Botón

los **Botón** control es sin duda uno de los controles más utilizados en cada interfaz de usuario. Tras observar un par de ejemplos de la **Botón** anteriormente, pero aquí es un resumen rápido. Este control expone las propiedades se resumen en la Tabla 6, y se declara así:

```
<Botón x: Name = texto "Button1" = "Pulse aquí" TextColor = BorderColor "Naranja" = "red"
BorderWidth = "2" BorderRadius = "2" Seguido = "Button1_Clicked" />
```

Tabla 6: Propiedades de Button

Propiedad	Descripción
Texto	El texto en el botón.
Color de texto	El color del texto en el botón.
Color del borde	Dibuja un borde de color alrededor del botón.

Propiedad	Descripción
Ancho del borde	El ancho del borde del botón.
BorderRadius	El radio de los bordes alrededor del botón.
Imagen	Una imagen opcional que se encuentra cerca del texto.

Observe cómo se puede especificar un nombre con **x: Nombre** para que pueda interactuar con el botón en C #, que es el caso cuando se establece el **hecho clic** evento con un controlador. Este control también expone una **Fuente** propiedad cuyo comportamiento se explica en la siguiente sección sobre el texto.

Trabajar con el texto: Etiqueta, de entrada, Editor

Visualización de texto y solicitando la entrada del usuario en forma de texto es muy común en todas las aplicaciones móviles.

Xamarin.Forms ofrece la **Etiqueta** de control para visualizar sólo lectura del texto y

Entrada y **Editor** controles para recibir mensajes de texto. Los **Etiqueta** control tiene algunas propiedades útiles, como se muestra en el siguiente XAML:

```
<Etiquetas de texto = "Viendo algo de texto" LineBreakMode = "WordWrap"
    TextColor = "Blue" xalign = "centro" yalign = "center" />
```

LineBreakMode te permite truncar o envuelva una cadena larga y se puede asignar un valor de la **LineBreakMode** enumeración.

Por ejemplo, **Ajuste de línea** divide una cadena larga en varias líneas proporcionales al espacio disponible. Si no se especifica, **nowrap** es el valor predeterminado. **xalign** y **yalign**

especificar la alineación horizontal y vertical para el texto. Los **Entrada** control en lugar le permite introducir una sola línea de texto y se declara de la siguiente manera:

```
<Entrada x: Name = "entry1" marcador de posición = "Introduzca un texto ..."
    TextColor = teclado "verde" = "Chat" Completado = "Entry1_Completed" />
```

Los **marcador de posición** la propiedad le permite mostrar un texto específico en la entrada hasta que el usuario escribe algo. Es útil para explicar el propósito del cuadro de texto. Cuando el usuario toca la **Entrada**,

se visualiza el teclado en pantalla. La apariencia del teclado puede ser controlado a través de la **Teclado** propiedad, lo que le permite mostrar las claves más adecuadas en función de la

Entrada's propósito. Los valores admitidos son **Chat**, **correo electrónico**, **numérico**, **Teléfono**, y **Número**. Si

Teclado No se asigna, **Defecto** se supone. Este control también expone dos eventos:

Terminado, que se dispara cuando los usuarios finalizar el texto pulsando la tecla de retorno, y

TextChanged, lo que se cuece a cada golpe de teclado. Usted proporciona controladores de eventos de la forma habitual, de la siguiente manera:

```
Entry1_Completed vacío (object sender, EventArgs e) {privado
```

```
    // Entry1.Text contiene el texto completo}
```

Entrada también proporciona la **IsPassword** propiedad para enmascarar el **Entrada** contenido 's, que se utiliza cuando el usuario debe introducir una contraseña. La combinación de la **Etiqueta** y **Entrada** controles es visible en la Figura 29.

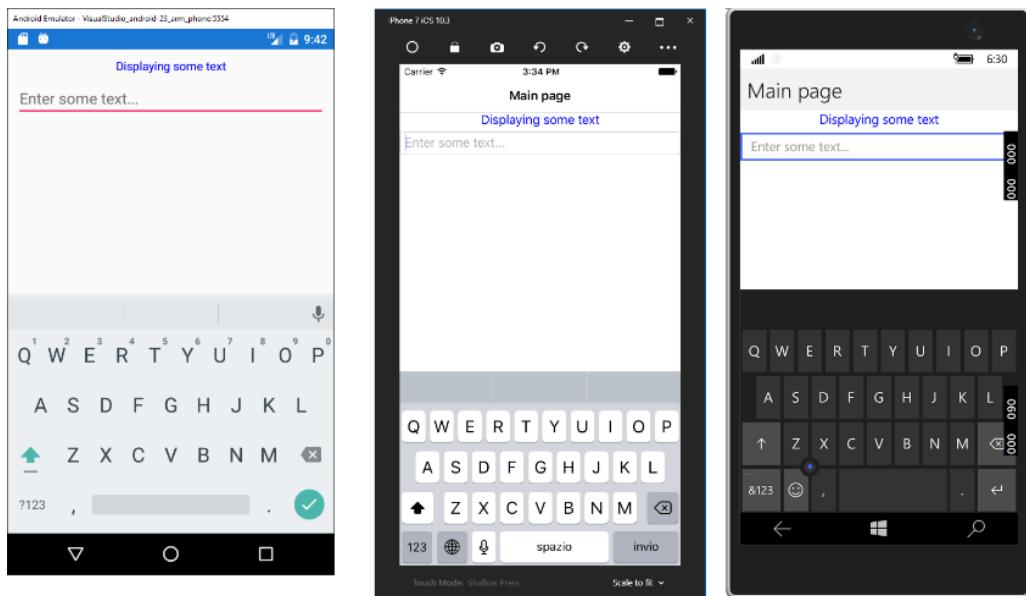


Figura 29: la etiqueta y de introducción de controles

los **Editor** el control es muy similar a **Entrada** En términos de comportamiento, eventos y propiedades, pero permite introducir varias líneas de texto. Por ejemplo, si coloca el editor dentro de un diseño, puede configurar su **HorizontalOptions** y **VerticalOptions** con propiedades **Llenar** por lo que tomará todo el espacio disponible en la matriz.

administración de los tipos

Los controles que muestra parte del texto (incluyendo el **Botón**) o que espera la entrada del usuario a través del teclado también exponen algunas propiedades relacionadas con las fuentes, tales como **FontFamily**, **FontAttributes**,

y **Tamaño de fuente**. **Familia tipográfica** especifica el nombre de la fuente que desea utilizar. **FontAttributes**

muestra el texto como **Itálico** o **Negrita** y, si no se especifica, **Ninguna** se supone. Con **Tamaño de fuente**, puede especificar el tamaño de la fuente, ya sea con un valor numérico o con una llamada *tamaño llamado*, basado en el

Micro, **Pequeña**, **Medio**, y **Grande** valores de la **NamedSize** enumeración. Con esta enumeración, **Xamarin.Forms** elige el tamaño apropiado para la plataforma actual. Por ejemplo, se permiten las siguientes dos opciones para configurar el tamaño de letra:

```
<Etiquetas de texto = "Un poco de texto" Tamaño de Letra = "72" /> <texto de
etiqueta = "Un poco de texto" Tamaño de Letra = "grande" />
```

A menos que usted está escribiendo una aplicación para una sola plataforma, le recomiendo que evite el uso de valores numéricos. Utilice el tamaño nombrado en su lugar.

Trabajar con fechas y horas: Selector de fechas y TimePicker

Otro requisito común en las aplicaciones móviles está trabajando con fechas y horas: **Xamarin.Forms** proporciona la **Selector de fechas** y **TimePicker** vistas por eso. En cada plataforma, éstas se representan con los correspondientes selectores de fecha y hora. **Selector de fechas** expone el **Fecha**, **MinimumDate**,

y **MaximumDate** propiedades que representan la fecha seleccionada / corriente, la fecha mínima, y la fecha máxima, respectivamente, todos los tipo **Fecha y hora**. Se expone un evento llamado

DateSelected, que se eleva cuando el usuario selecciona una fecha. Puede manejar esto para recuperar el valor de la **Fecha** propiedad. La vista puede ser declarado de la siguiente manera:

```
<DatePicker x: Name = "DATEPICKER1" MinimumDate = "17/07/2017"  
          MaximumDate = "12/31/2017"  
          DateSelected = "DatePicker1_DateSelected" />
```

Y luego en el código subyacente puede recuperar la fecha seleccionada como esto:

```
private void DatePicker1_DateSelected (remitente del objeto, DateChangedEventArgs e) {  
  
    DateTime selectedDate = e.NewDate; }
```

los **DateChangedEventArgs** objeto almacena la fecha seleccionada en el **NewDate** la propiedad y la fecha anterior en el **OldDate** propiedad. La Figura 30 muestra la **Selector de fechas** en las tres plataformas.

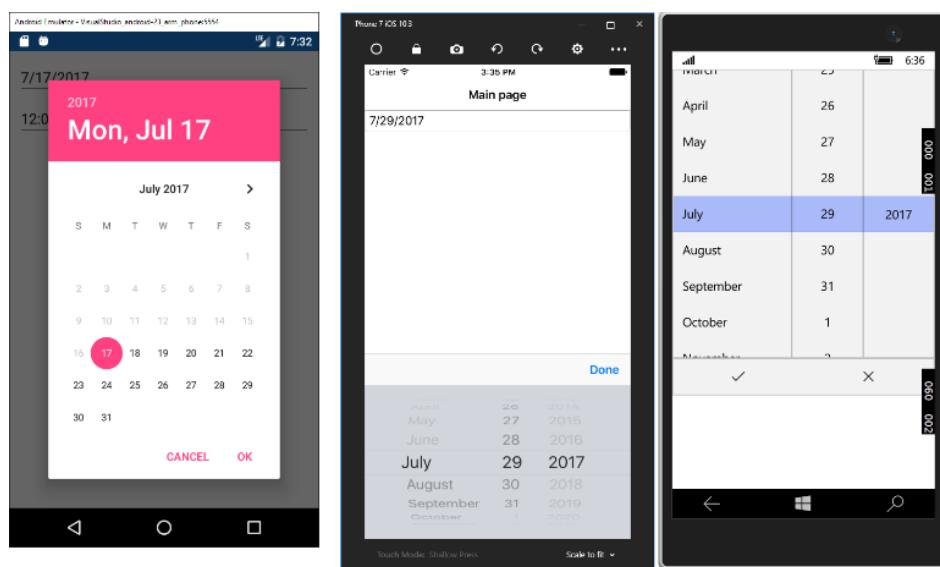


Figura 30: El DatePicker en acción

los **TimePicker** expone una propiedad llamada **Hora**, de tipo **Espacio de tiempo**, pero no expone un evento específico para la selección de tiempo, por lo que necesita para utilizar la **PropertyChanged** evento. En términos de XAML, se declara una **TimePicker** Me gusta esto:

```
<TimePicker x: Name = "TimePicker1"  
           PropertyChanged = "TimePicker1_PropertyChanged" />
```

Luego, en el código subyacente, es necesario detectar cambios en la **Hora** propiedad de la siguiente manera:

```
private void TimePicker1_PropertyChanged (remitente del objeto,  
                                         System.ComponentModel.PropertyChangedEventArgs e) {  
  
    si (e.PropertyName == TimePicker.TimeProperty.PropertyName) {
```

```
TimeSpan selectedTime = TimePicker1.Time; }}
```

TimeProperty es una propiedad de dependencia, un concepto que será discutido en el capítulo 7. La Figura 31 muestra la **TimePicker** en acción.

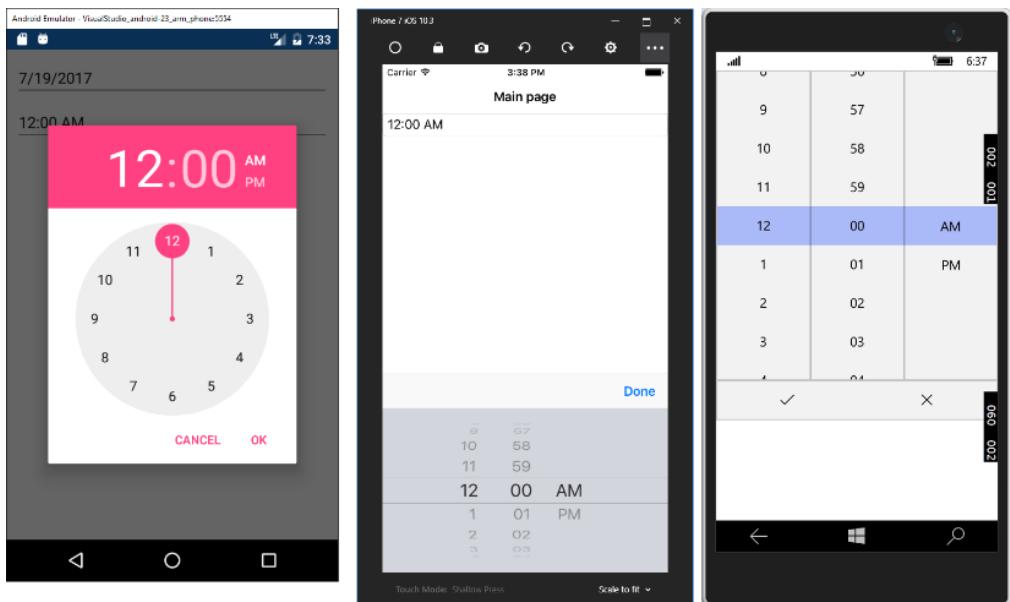


Figura 31: El TimePicker en acción



Consejo: También puede asignar una fecha u hora a los recolectores en el C# de código subyacente. Por ejemplo, en el constructor de la página que los declara.

Viendo el contenido HTML con WebView

los **WebView** control permite mostrar el contenido HTML, incluyendo páginas web y el formato HTML estático. Este control expone la **Navegando** y **navegado** eventos que se plantean cuando la navegación se inicia y completa, respectivamente. El poder real está en su **Fuente** propiedad, del tipo de **WebViewSource**, que pueden ser asignados con una variedad de contenido, como URLs o cadenas que contienen el formato HTML. Por ejemplo, el siguiente XAML abre el sitio web especificado:

```
<WebView x: Name = "WebView1" Fuente = "https://www.xamarin.com" />
```

El siguiente ejemplo muestra cómo en lugar puede asignar el **Fuente** propiedad con una cadena:

```
WebView1.Source = "<div> <h1> Header </ h1> </ div>";
```

Para el dimensionamiento dinámico, una mejor opción es que encierra el **WebView** dentro de una **Cuadrícula** diseño. Si en lugar de utilizar el **StackLayout**, es necesario proporcionar la altura y la anchura de forma explícita. Al navegar por el contenido en Internet, es necesario tener el permiso de Internet en el manifiesto de Android y el permiso de Internet (cliente) en el manifiesto UWP. La Figura 32 muestra cómo aparece el WebView.

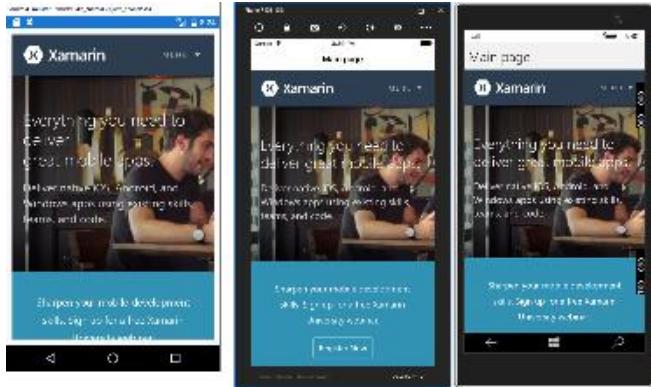


Figura 32: muestra el contenido HTML con WebView

Si la página web que muestre le permite navegar a otras páginas, puede aprovechar el built-in **Regresa** y **GoForward** métodos, junto con el **CanGoBack** y **CanGoForward** propiedades booleanas para controlar de forma automática la navegación entre páginas.



Consejo: Si es necesario implementar la navegación a direcciones URL, podría valer la pena considerar la llamada [deep linking](#) función, disponible desde Xamarin.Forms 2.3.1.

Aplicación de Seguridad de Transporte en IOS

A partir de IOS 9, Apple presentó algunas restricciones para acceder a los recursos en red, incluyendo sitios web, que permite la navegación sólo a través del protocolo HTTPS por defecto. Esta función se conoce como la aplicación de Transporte de Seguridad (ATS). ATS puede ser controlado en las propiedades del proyecto de iOS, y permite la introducción de algunas excepciones, porque puede que tenga que examinar el contenido HTTP a pesar de las restricciones. Más detalles acerca de ATS y excepciones están disponibles en el [documentación](#); Sin embargo, recuerde que si el **WebView** no muestra el contenido en iOS, la razón podría ser ATS.

La implementación de selección de valor: Switch, Slider, paso a paso

Xamarin.Forms ofrece una serie de controles para la entrada del usuario en base a la selección de valores. El primero de ellos es el **Cambiar**, que proporciona un valor comutado y es útil para la selección de valores como verdadero / falso, encendido / apagado, y activado / desactivado. Se expone la **IsToggled** propiedad, lo que convierte el interruptor cuando cierto, y el **toggled** evento, que se genera cuando el usuario cambia la posición del interruptor. Este control no tiene etiqueta incorporada, por lo que necesita para usarlo en conjunción con una **Etiqueta** como sigue:

```
<StackLayout Orientación = "horizontal">
<Etiquetas de texto = "Activar el plan de datos" />
```

```
<Commutación x: Name = "Switch1" IsToggled = "True" toggled = "Switch1_Toggled"
    Margen = "5,0,0,0" />
</ StackLayout>
```

los **toggled** evento almacena el nuevo valor en el **ToggledEventArgs** objeto que se utiliza de la siguiente manera:

```
private void Switch1_Toggled (object sender, ToggledEventArgs e) {
    bool isToggled = e.Value; }
```

los **deslizador** permite la entrada de un valor lineal. Se expone la **Valor, Mínimo, y Máximo** propiedades, todo de tipo **doble**, que representan el valor actual, el valor mínimo y el valor máximo. Como el **Cambiar**, que no tiene una etiqueta incorporada, por lo que se puede utilizar junto con una **Etiqueta** como sigue:

```
<StackLayout Orientación = "horizontal">
    <Etiquetas de texto = "Seleccione su edad:" />
    <Deslizante x: Name = "Slider1" Maximum = "85" Mínimo = "13" Value = "30" ValueChanged =
        "Slider1_ValueChanged" /> </ StackLayout>
```



Consejo: Sorprendentemente, si se escribe el Mínimo antes de Máximo, se producirá un error de ejecución. Así, tanto para el deslizador y el paso a paso, las cuestiones de orden.

los **ValueChanged** evento se genera cuando el usuario mueve el selector de la **deslizador** y el nuevo valor se envía a la **Nuevo valor** propiedad de la **ValueChangedEventArgs** objeto que se obtiene en el controlador de eventos. El último control es el **paso a paso**, que permite que el suministro de valores discretos con un incremento especificado. También permite la especificación de valores mínimos y máximos. Se utiliza el **Valor, Valor mínimo, mínimo, y Máximo** inmuebles del tipo **doble** como sigue:

```
<StackLayout Orientación = "horizontal">
    <Etiquetas de texto = "Seleccione su edad:" />
    <Paso a paso x: Name = Incremento "Stepper1" = "1" Maximum = "85" Mínimo = "13"
        Value = "30" ValueChanged = "Stepper1_ValueChanged" /> <label x:
        Name = "StepperValue" /> </ StackLayout>
```

Nótese que tanto el **paso a paso** y **deslizador** sólo proporcionan una manera de incrementar y disminuir un valor, por lo que es su responsabilidad para mostrar el valor actual, por ejemplo, con una **Etiqueta** que puede manejar a través de la **ValueChanged** evento. El código siguiente muestra cómo lograr esto con el **Paso a paso**:

```
Stepper1_ValueChanged vacío (object sender, ValueChangedEventArgs e) {privado
    StepperValue.Text = e.NewValue.ToString (); }
```

La Figura 33 muestra un resumen de todos los puntos de vista anteriormente mencionados.

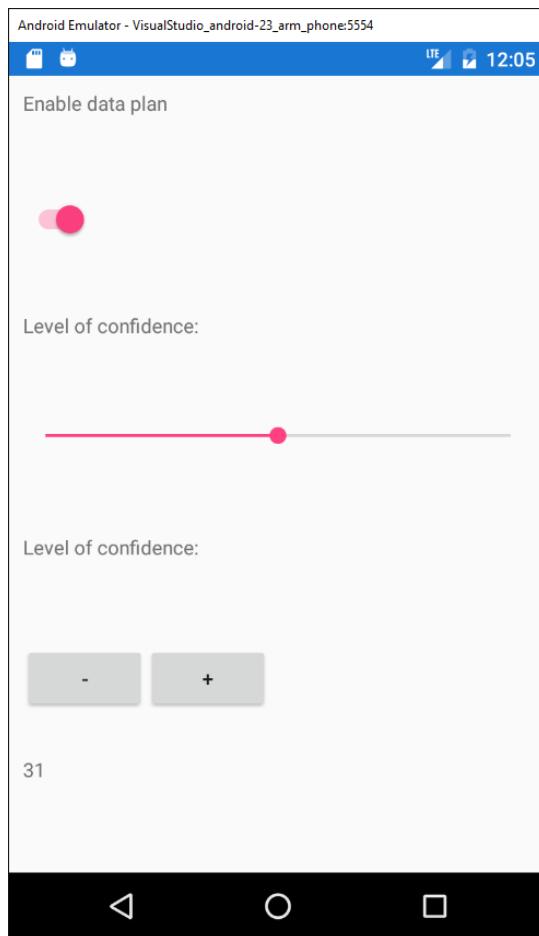


Figura 33: una vista de resumen del interruptor, Slider, y paso a paso controles

Presentación de la Barra de búsqueda

Una de las mejores vistas en Xamarin.Forms, la **Barra de búsqueda** muestra un cuadro de búsqueda nativo con un ícono de búsqueda que los usuarios pueden acceder. Este punto de vista expone la **SearchButtonPressed** evento. Puede controlar este evento para recuperar el texto que el usuario escribió en el cuadro y luego realizar su lógica de búsqueda; por ejemplo, la ejecución de una consulta LINQ contra de un conjunto de datos de **recogida o de filtrado en memoria de la tabla de una base de datos local**. También expone la **TextChanged** evento, que se planteó en cada golpe de teclado, y la **marcador de posición** propiedad, lo que le permite especificar un texto de marcador de posición como el mismo nombre de la propiedad **Entrada** controlar. Se declara de la siguiente manera:

```
<SearchBar x: Name = "earchBar1" marcador de posición = "Introduzca su clave de búsqueda ..." SearchButtonPressed = "earchBar1_SearchButtonPressed" />
```

La figura 34 muestra un ejemplo.

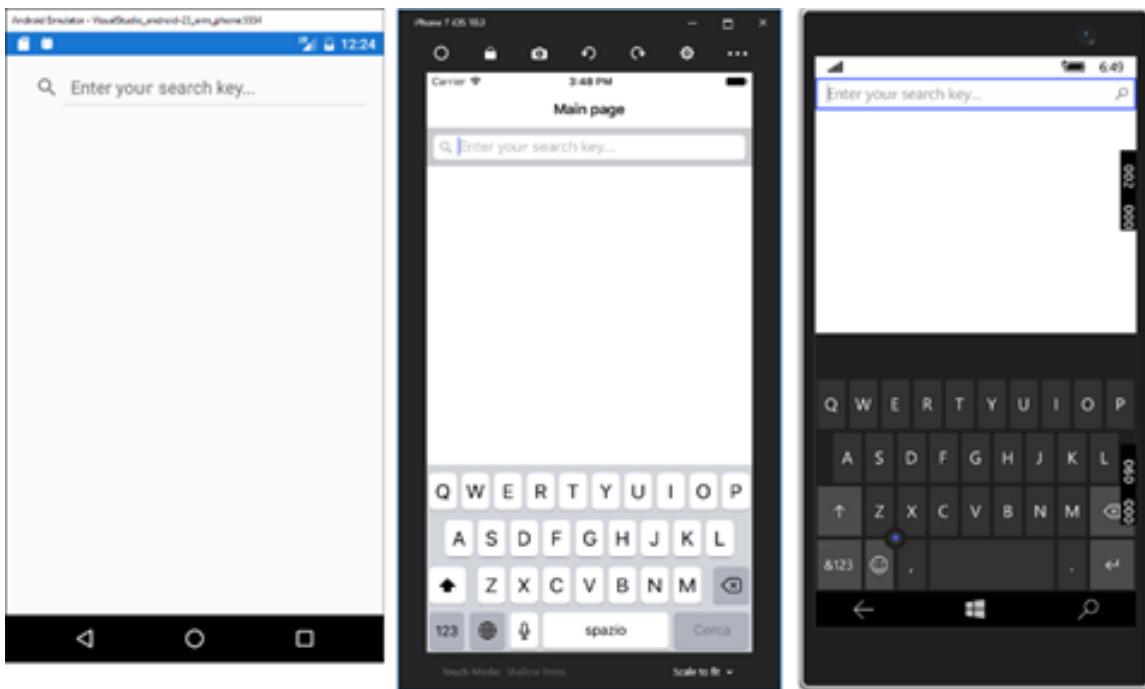


Figura 34: La vista SearchBar

En el capítulo 7, usted aprenderá cómo mostrar listas de elementos a través de la **Vista de la lista** controlar. Los **Barra de búsqueda** puede ser un buen compañero en el que se puede utilizar para filtrar una lista de elementos en función de la clave de búsqueda introducido por el usuario.

operaciones de larga duración: ActivityIndicator y Barra de progreso

En algunas situaciones, la aplicación puede ser que necesite para realizar operaciones potencialmente larga ejecución, tales como la descarga de contenidos a partir de los datos de Internet o de carga de una base de datos local. En tales situaciones, es una buena práctica para informar al usuario de que una operación está en curso. Esto puede lograrse con dos puntos de vista, la **ActivityIndicator** o el **Barra de progreso**. Este último expone una propiedad llamada **Progreso**, de tipo **doble**. Este control no se utiliza muy a menudo, ya que implica que son capaces de calcular la cantidad de tiempo o datos necesarios para completar una operación. los

ActivityIndicator en su lugar muestra un indicador simple, animada que se muestra cuando una operación se está ejecutando, sin la necesidad de calcular su progreso. Se habilita al establecer su **Esta corriendo** propiedad a **cierto**; También puede ser que desee para que sea visible sólo cuando en funcionamiento, hace asignando **Es visible** con cierto. Por lo tanto, normalmente se declara que en XAML de la siguiente manera:

```
<ActivityIndicator x: Name = "ActivityIndicator1" />
```

Luego, en el código subyacente, se puede controlar de la siguiente manera:

```
// comienzo de la operación ...
ActivityIndicator1.isVisible = true;
ActivityIndicator1.isRunning = true;

// La ejecución de la operación ...
```

```
// La operación se realizó  
ActivityIndicator1.IsRunning = false;  
ActivityIndicator1.isVisible = false;
```

Como sugerencia personal, recomiendo siempre establezca tanto **Es visible** y **Esta corriendo**. Esto le ayudará a mantener un comportamiento coherente en todas las plataformas. La Figura 35 muestra un ejemplo basado en Android.

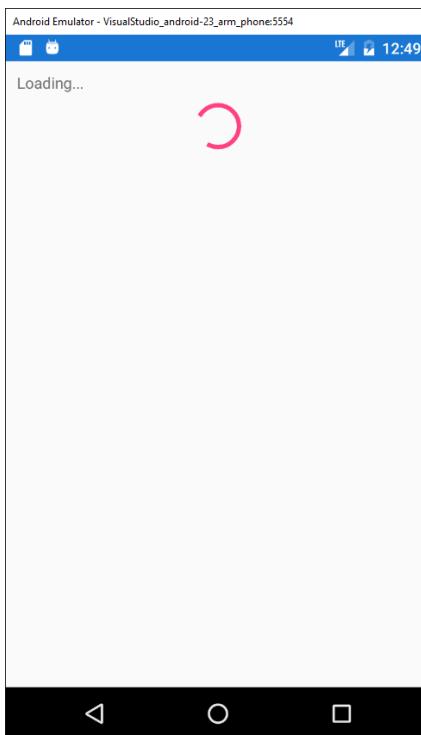


Figura 35: El ActivityIndicator muestra que una operación en curso



Propina: Página objetos, tales como la Pagina de contenido, exponer una propiedad llamada **Está ocupado** que permite a un indicador de actividad cuando se asigna con cierto. Dependiendo de su situación, también podría considerar esta opción.

Trabajar con imágenes

El uso de imágenes es crucial en aplicaciones móviles, ya que tanto enriquecen la apariencia de la interfaz de usuario y permiten aplicaciones para apoyar los contenidos multimedia. Xamarin.Forms proporciona una **Imagen** el control se puede utilizar para visualizar imágenes de Internet, archivos locales, y los recursos incrustados. La visualización de imágenes es muy simple, mientras que la comprensión de cómo se carga y el tamaño de las imágenes es más complejo, especialmente si usted no tiene experiencia previa con XAML y las interfaces de usuario dinámicas. Se declara un **Imagen** como sigue:

```
<Image Source = "https://www.xamarin.com/content/images/pages/branding/assets/x amarin-logo.png"  
Aspecto = "AspectFit" />
```

Como se puede ver, se asigna el **Fuente** propiedad con la ruta de la imagen, que puede ser una URL o el nombre de un archivo local o un recurso. **Fuente** puede ser asignado ya sea en XAML o en el código subyacente. Se le asignará esta propiedad en C # código cuando es necesario asignar la propiedad en tiempo de ejecución. Esta propiedad es de tipo **Fuente de imagen** y, mientras XAML tiene un convertidor de tipos para ello, en C # tiene que utilizar métodos específicos dependiendo de la fuente de la imagen: **Desde el archivo** requiere una ruta de archivo que se puede resolver en cada plataforma, **FromUri** requiere una **System.Uri** objeto, y **FromResource** le permite especificar una imagen en los recursos de aplicaciones embebidas.



Nota: Cada plataforma tiene su propia forma de trabajar con imágenes locales y recursos incrustados, lo que requiere más explicación. Debido a que esto va más allá del alcance de este libro electrónico, le recomiendo que lea el [documentación oficial](#), Que también se explica cómo administrar imágenes para diferentes propósitos en iOS, Android y Windows.

los **Aspecto** propiedad determina cómo el tamaño y estirar una imagen dentro de los límites que se está mostrando en. Se requiere un valor de la **Aspecto** enumeración:

- **Llenar:** Estira la imagen para que llene la pantalla completa y exactamente igual. Esto puede dar lugar a la imagen que se está distorsionada.
- **AspectFill:** Clips de la imagen para que ocupe toda el área de visualización, preservando el aspecto.
- **AspectFit:** Letterboxes la imagen (si es necesario) de manera que toda la imagen encaja en el área de visualización, con espacio en blanco añadido a la parte superior, parte inferior, o lados, dependiendo de si la imagen es de ancho o alto.

También puede establecer la **WidthRequest** y **HeightRequest** propiedades para ajustar el tamaño de la **Imagen** controlar. La figura 36 muestra un ejemplo.



Figura 36: Visualización de imágenes con la imagen de Visión

Formatos de imagen soportados son .jpg, .png, .gif, .bmp y .tif. Trabajar con imágenes también implica iconos y pantallas de inicio, que son totalmente dependientes de la plataforma y por lo tanto se requieren para leer el funcionario [documentación](#). Además, para gráficos mejorados en 2-D, es posible que desee considerar la adopción de una mirada a la [biblioteca SkiaSharp](#), Una biblioteca portátil que funciona muy bien con Xamarin.Forms y es alimentado por la biblioteca Skia de Google.

La introducción de los reconocedores gesto

Vistas como **Imagen** y **Etiqueta** no incluyen soporte para gestos táctiles de forma nativa, pero a veces es posible que desee permitir a los usuarios aprovechar una imagen o un texto para realizar una acción como la navegación a una página o sitio web. En Xamarin.Forms, puede aprovechar [reconocedores gesto](#) para añadir soporte táctil para puntos de vista que no es necesario incluirlos fuera de la caja. Vistas exponen una colección llamada

GestureRecognizers, de tipo **IList <GestureRecognizer>**. reconocedores gesto soportados son:

- **TapGestureRecognizer**: Permite el reconocimiento de los grifos.
- **PinchGestureRecognizer**: Permite el reconocimiento del gesto de pellizcar para zoom.
- **PanGestureRecognizer**: Permite el arrastre de objetos con el gesto sartén.

Por ejemplo, el siguiente XAML muestra cómo agregar una **TapGestureRecognizer** a una **Imagen** controlar:

```
<Image Source = "https://www.xamarin.com/content/images/pages/branding/assets/x amarin-logo.png"  
Aspecto = "AspectFit">  
    <Image.GestureRecognizers>  
        <TapGestureRecognizer x: Name = "ImageTap" NumberOfTapsRequired = "1"  
        roscado = "ImageTap_Tapped" /> </Image.GestureRecognizers> </ Imagen>
```

Puede asignar el **NumberOfTapsRequired** propiedad (auto-explicativo) y la **aprovechado** evento con un controlador que se invoca cuando el usuario toca la imagen. Eso va a tener este aspecto:

```
ImageTap_Tapped private void (object sender, EventArgs e) {  
  
    // Hacer tus cosas aquí ...}
```

reconocedores gesto le dan una gran flexibilidad y le permiten mejorar la experiencia del usuario en sus aplicaciones móviles mediante la adición de soporte táctil donde se requiera.

Viendo alertas

Todas las plataformas pueden mostrar mensajes emergentes de alerta con mensajes informativos o recibir la entrada del usuario con las opciones comunes, como Aceptar o Cancelar. Artículos en la Xamarin.Forms proporcionan un método asíncrono llamada **DisplayAlert**, que es muy fácil de usar. Por ejemplo, supongamos que desea mostrar un mensaje cuando el usuario pulsa un botón. El código siguiente muestra cómo lograr esto:

```
asíncrono privada vacío Button1_Clicked (object sender, EventArgs e) {
```

```
    DisplayAlert espera ( "Título", "Este es un informativo pop-up", "OK"); }
```

Como un método asíncrono, se llama **DisplayAlert** con el **esperar** operador, marcando el método que contiene como **asíncrono**. El primer argumento es el título emergente, el segundo argumento es el mensaje de texto, y el tercer argumento es el texto que desea mostrar en el único botón que aparece. Actualmente, **DisplayAlert** tiene una sobrecarga que puede esperar a la entrada del usuario y de retorno

cierto o **falso** dependiendo de si el usuario selecciona la opción Aceptar o la opción Cancelar:

```
bool resultado =  
    DisplayAlert espera ( "Título", "¿Desea continuar?", "OK", "Cancelar");
```

Usted es libre de escribir el texto que desea para el OK y Cancelar opciones, e IntelliSense le ayuda a comprender el orden de estas opciones en la lista de parámetros. Si el usuario selecciona OK,

DisplayAlert devoluciones **cierto**. La figura 37 muestra un ejemplo de la alerta.

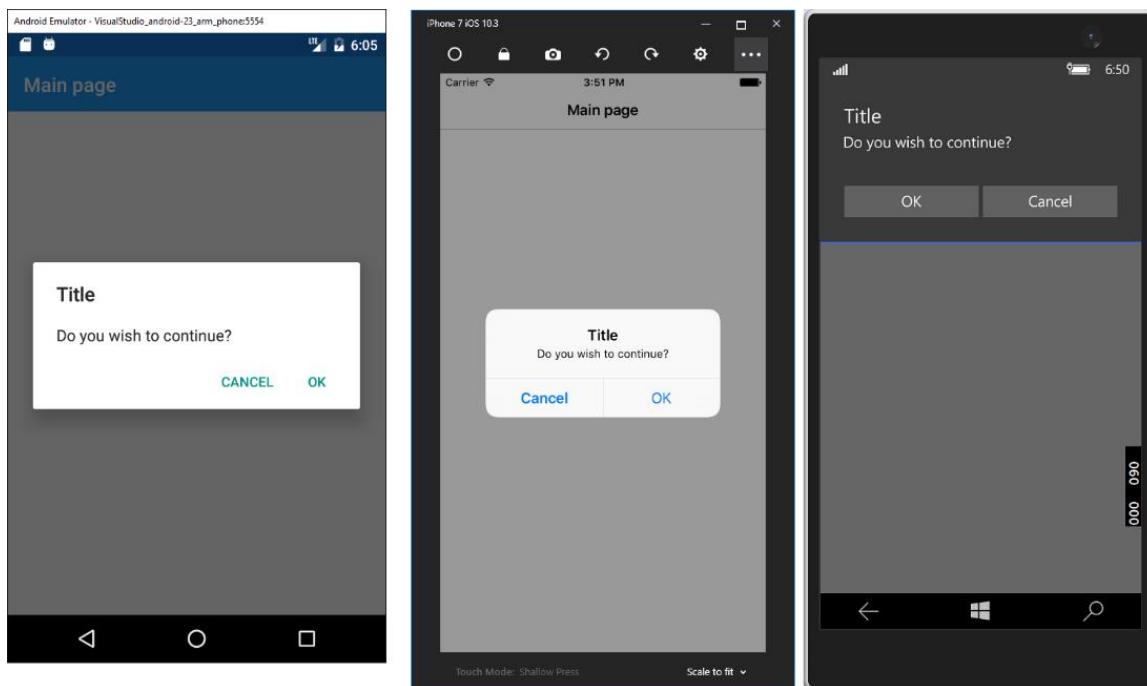


Figura 37: Viendo alertas

Resumen del capítulo

Este capítulo introduce los conceptos de vista y puntos de vista comunes en Xamarin.Forms, los bloques de construcción para cualquier interfaz de usuario en sus aplicaciones para móviles. Usted ha visto cómo obtener la entrada del usuario con el Botón, Entrada, Editor, y Barra de búsqueda controles; que han visto cómo mostrar la información con el Etiqueta y cómo utilizar en conjunción con otros puntos de vista de entrada tales como Deslizador, paso a paso, y Cambiar. Usted ha visto cómo el Selector de fechas y TimePicker vistas le permiten trabajar con fechas y horas. Usted ha visto cómo mostrar imágenes con la Imagen ver; se ha utilizado la WebView para mostrar contenido HTML; y has visto cómo informar al usuario de la marcha de las operaciones de funcionamiento largo con ActivityIndicator y Barra de progreso. Por último, usted ha visto cómo añadir soporte de gestos a puntos de vista que no es necesario incluirlo fuera de la caja, y cómo mostrar alertas para fines informativos y para permitir que las elecciones del usuario.

Ahora usted tiene todo lo que necesita para construir interfaces de usuario de alta calidad con diseños y puntos de vista, y que ha visto cómo utilizar todos estos bloques de construcción en una sola página. Sin embargo, la mayoría de las aplicaciones móviles están hechos de varias páginas. El siguiente capítulo explica todos los tipos de páginas disponibles en Xamarin.Forms y la infraestructura de navegación.

Capítulo 6 páginas y Navegación

En los capítulos anteriores, fuimos los fundamentos de la disposición y puntos de vista, los bloques de construcción fundamentales de la interfaz de usuario en aplicaciones móviles. Sin embargo, he demostrado cómo utilizar presentaciones y vistas dentro de una sola página, mientras que las aplicaciones móviles en el mundo real están hechas de varias páginas. Android, iOS y Windows proporcionan una serie de diferentes páginas que le permiten visualizar el contenido de varias maneras y para proporcionar la mejor experiencia de usuario posible basado en el contenido que necesita presentar. Xamarin.Forms proporciona modelos unificados de páginas que puede utilizar desde su única, compartida C # código base que trabajan multiplataforma. También proporciona un marco de navegación fácil de usar, la infraestructura que se utiliza para moverse entre las páginas. Páginas y la navegación son los temas de este capítulo y las últimas piezas del marco de interfaz de usuario que necesita saber para construir hermosa,



Nota: Con el fin de seguir los ejemplos de este capítulo, crear una nueva solución Xamarin.Forms basado en la estrategia de código compartido PCL. El nombre es de usted. Cada vez que se discute una nueva página, simplemente limpiar el contenido de los archivos y MainPage.xaml MainPage.xaml.cs (excepto por el constructor) y escribir el nuevo código.

La introducción y la creación de páginas

Xamarin.Forms ofrece muchos objetos de la página que se puede utilizar para configurar las interfaces de usuario de sus aplicaciones. Las páginas son elementos raíz de la jerarquía visual, y cada página le permite agregar sólo un elemento visual, típicamente un diseño raíz con otros diseños y elementos visuales anidadas dentro de la raíz. Desde un punto de vista técnico, todos los objetos de la página en Xamarin.Forms derivan de lo abstracto **Página** clase, que proporciona la infraestructura básica de cada página, incluidas las propiedades comunes, tales como **Contenido**, definitivamente la propiedad más importante que se asigna con el elemento visual de la raíz. La Tabla 7 describe las páginas disponibles en Xamarin.Forms.

Tabla 7: Artículos en Xamarin.Forms

Tipo de página	Descripción
Pagina de contenido	Muestra un único objeto de vista.
TabbedPage	Facilita la navegación entre las páginas secundarias utilizando fichas.
CarouselPage	Facilita mediante el gesto de deslizar entre las páginas secundarias.
MasterDetailPage	Gestiona dos paneles separados, que incluye un control de menú lateral.
NavigationPage	Proporciona la infraestructura para la navegación entre páginas.

Las siguientes secciones describen las páginas disponibles en más detalle. Recuerde que Visual Studio proporciona plantillas de elementos de diferentes tipos de páginas, por lo que puede hacer clic en el proyecto PCL en el Explorador de soluciones, seleccione **Agregar ítem nuevo**, y en el cuadro de diálogo Agregar nuevo elemento, verá las plantillas para cada página se describe en la Tabla 7.

vistas individuales con el Pagina de contenido

los **Pagina de contenido** objeto es la página más sencilla posible y permite visualizar un único elemento visual. Ya miraste algunos ejemplos de la **Pagina de contenido** anteriormente, pero vale la pena mencionar su **Título** propiedad. Esta propiedad es particularmente útil cuando el **Pagina de contenido** se utiliza en las páginas con la navegación incorporada, tales como **TabbedPage** y **CarouselPage**, porque ayuda a identificar la página activa. El núcleo de la **Pagina de contenido** es el **Contenido** propiedad, que se asigna con el elemento visual que desea visualizar. El elemento visual puede ser o bien un único control o un diseño; este último permite crear jerarquías visuales complejas e interfaces de usuario en el mundo real. En XAML, la etiqueta para el **Contenido** propiedad puede ser omitida, que es también una práctica común (también cuenta **Título**):

```
<? 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: locales =
    "CLR-espacio de nombres: App1" title = "página principal" x: Class =
    "App1.MainPage">

    <Etiquetas de texto = "Una página de contenido" />

</ ContentPage>
```

los **Pagina de contenido** se pueden utilizar individualmente o como los contenidos de otras páginas discutidos en las siguientes secciones.

La división de contenidos con el MasterDetailPage

los **MasterDetailPage** Es una página muy importante, ya que permite dividir el contenido en dos categorías separadas: genéricas y detalladas. La interfaz de usuario proporcionada por el **MasterDetailPage** es muy común en Android e iOS. Dispone de un menú lateral de la izquierda (la parte principal) que se puede deslizar para mostrar y ocultar, y una segunda zona de la derecha que muestra el contenido más detallado (la parte detalle). Por ejemplo, un escenario muy común para este tipo de página se muestra una lista de temas o ajustes en el maestro y el contenido para el tema seleccionado o entorno en el detalle. Tanto el maestro y los detalles de las piezas están representados por **Pagina de contenido**

objetos. Una declaración típica para una **MasterDetailPage** parece Listado de Código 11.

Listado de Código 11

```
<? Xml version = "1.0" encoding = "UTF-8"?> < MasterDetailPage xmlns =
"http://xamarin.com/schemas/2014/forms"
    xmlns : x = "Http://schemas.microsoft.com/winfx/2009/xaml"
```

```

    xmlns : local = "CLR-espacio de nombres: App1"
    Título = "Página Principal"
    x : Clase = "App1.MainPage">

< MasterDetailPage.Master > < Pagina de contenido > < Etiqueta Texto = "Este es el Maestro" HorizontalOptions
= "Centro"

    VerticalOptions = "Center" />
</ Pagina de contenido >
</ MasterDetailPage.Master > < MasterDetailPage.Detail > < Pagina de contenido > < Etiqueta Texto = "Se
trata de los detalles" HorizontalOptions = "Centro"

    VerticalOptions = "Center" />
</ Pagina de contenido >
</ MasterDetailPage.Detail >
</ MasterDetailPage >

```

Como se puede ver, se rellenan de la **Dominar** y **Detalle** propiedades con el apropiado

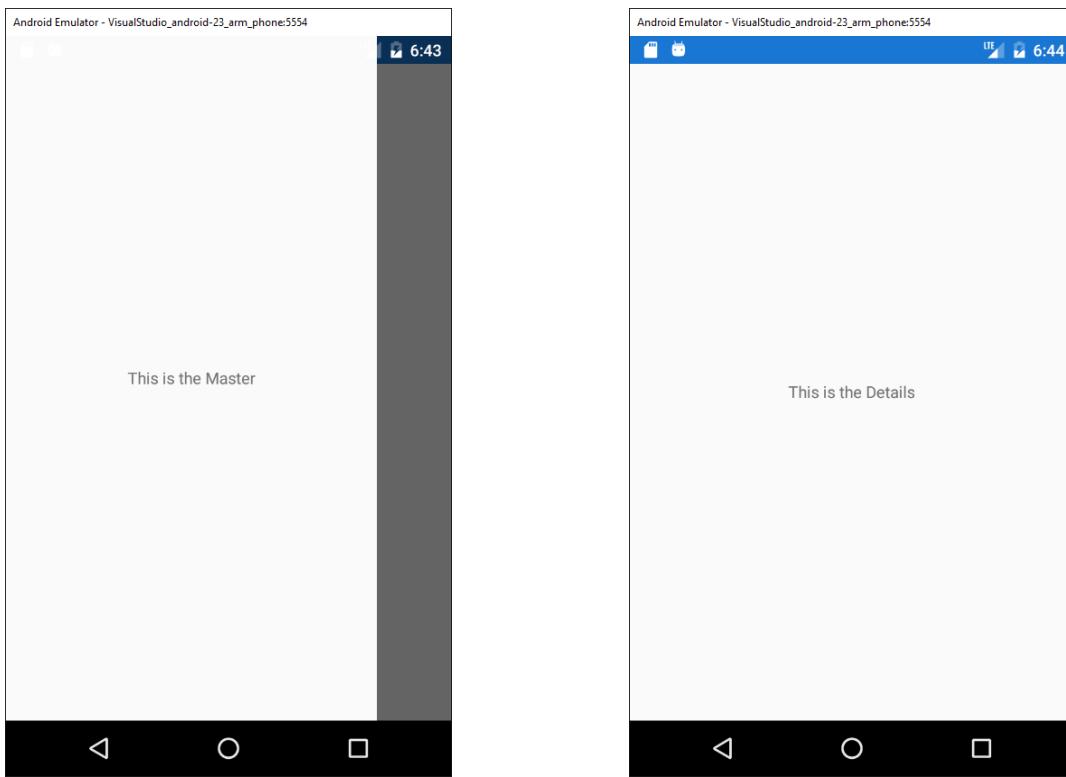
Pagina de contenido objetos. En aplicaciones del mundo real, es posible que tenga una lista de temas en el **Dominar**, entonces es posible mostrar los detalles de un tema en el **Detalle** cuando el usuario toca la una de la **Dominar** 'Contenido de s.



Nota: Cada vez que cambie la página raíz de Pagina de contenido a otro tipo de página, como MasterDetailPage, también es necesario cambiar la herencia en el código subyacente. Por ejemplo, si abre el archivo C # MainPage.xaml.cs, se verá que Pagina principal hereda de Pagina de contenido, pero en XAML se ha sustituido con este objeto MasterDetailPage. Por lo tanto, también es necesario hacer Pagina principal heredar de MasterDetailPage. Si se olvida esto, el compilador informará de un error. Esta nota es válida para las páginas discutidas en las siguientes secciones también.

Las figuras 38 y 39 muestran las partes maestra y detalle, respectivamente. Puede simplemente pase desde la izquierda para activar el menú lateral principal y, a continuación, deslizar hacia atrás para ocultarlo. También puede controlar el menú lateral mediante programación mediante la asignación de la **Es presentado** propiedad con cierto (**visible**) o falso

(oculto). Esto es útil cuando la aplicación está en modo horizontal, porque el menú lateral se abre automáticamente por defecto.



*Figura 38: MasterDetailPage: el menú lateral
del maestro*

Figura 39: MasterDetailPage: El detalle

Viendo el contenido dentro de las pestañas con el TabbedPage

A veces puede que tenga que categorizar varias páginas por tema o por tipo de actividad. Cuando usted tiene una pequeña cantidad de contenido, se puede aprovechar la **TabbedPage**, que puede múltiple grupo **Página de contenido** objetos en pestañas para facilitar la navegación. los **TabbedPage** puede ser declarado como se muestra en Listado de Código 12.

Listado de Código 12

```
<? Xml version = "1.0" encoding = "UTF-8"?> < TabbedPage xmlns =
"http://xamarin.com/schemas/2014/forms"
    xmlns : x = "Http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns : local = "CLR-espacio de nombres: App1"
    Título = "Página Principal"
    x : Clase = "App1.MainPage">

    < TabbedPage.Children > < Página de contenido Título =
    "Primera">
        < Etiqueta Texto = "Esta es la primera página" HorizontalOptions = "Centro"
            VerticalOptions = "Center" />
```

```

</ Pagina de contenido > < Pagina de contenido Título =
"Segundo">
    < Etiqueta Texto = "Esta es la segunda página" HorizontalOptions = "Centro
"
        VerticalOptions = "Center" />
</ Pagina de contenido > < Pagina de contenido Título =
"Tercera">
    < Etiqueta Texto = "Esta es la tercera página" HorizontalOptions = "Centro"
        VerticalOptions = "Center" />
</ Pagina de contenido >
</ TabbedPage.Children >
</ TabbedPage >

```

Como se puede ver, se rellenan de la **Niños** colección con múltiples **Pagina de contenido** objetos. proporcionando una **Título** a cada **Pagina de contenido** es de vital importancia, ya que el texto del título se muestra en cada pestaña, como se demuestra en la Figura 40.

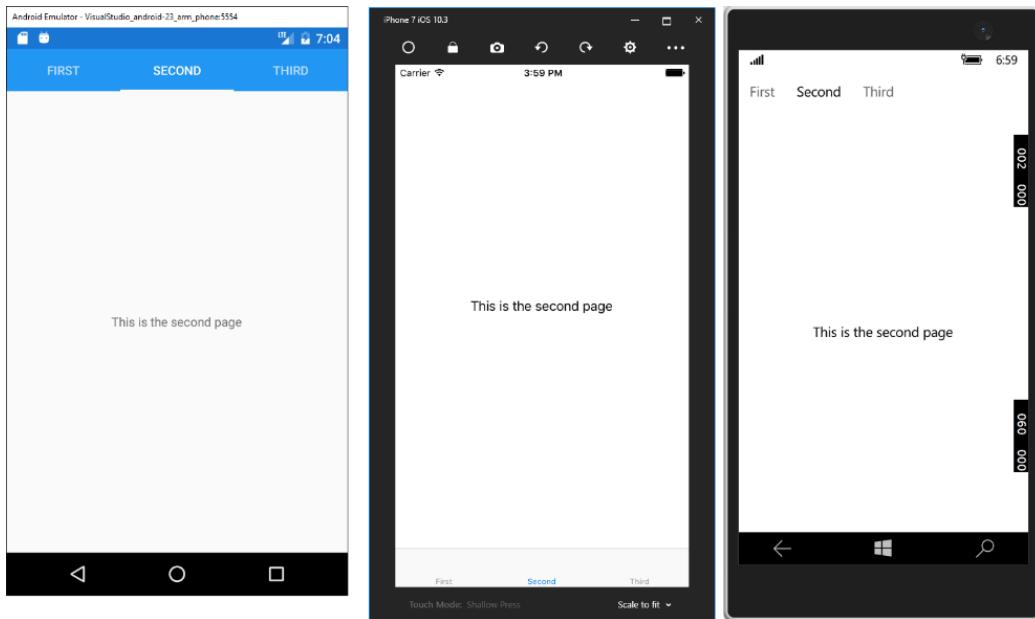


Figura 40: Viendo contenidos agrupados con el TabbedPage

Por supuesto, la **TabbedPage** funciona bien con un pequeño número de páginas secundarias, normalmente entre tres y cuatro páginas.

páginas con la swiping CarouselPage

los **CarouselPage** es similar a la **TabbedPage**, pero en lugar de tener las pestañas, se puede utilizar el gesto de deslizar para cambiar entre las páginas hijas. Por ejemplo, el **CarouselPage** podría ser perfecta para mostrar una galería de fotos. Listado de Código 13 muestra cómo declarar una **CarouselPage**.

Listado de Código 13

```
<? Xml version = "1.0" encoding = "UTF-8"?> < CarouselPage xmlns =
"http://xamarin.com/schemas/2014/forms"
    xmlns : x = "Http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns : local = "CLR-espacio de nombres: App1"
    Título = "Página Principal"
    x : Clase = "App1.MainPage">

    < CarouselPage.Children > < Pagina de contenido Título
        = "Primera">
        < Etiqueta Texto = "Esta es la primera página" HorizontalOptions = "Centro"
            VerticalOptions = "Center" />
    </ Pagina de contenido > < Pagina de contenido Título =
        "Segundo">
        < Etiqueta Texto = "Esta es la segunda página" HorizontalOptions = "Centro"
        VerticalOptions = "Center" />
    </ Pagina de contenido > < Pagina de contenido Título =
        "Tercera">
        < Etiqueta Texto = "Esta es la tercera página" HorizontalOptions = "Centro"
            VerticalOptions = "Center" />
    </ Pagina de contenido >
</ CarouselPage.Children >
</ CarouselPage >
```

La Figura 41 muestra cómo el **CarouselPage** aparece.



Figura 41: El borrar de contenido con el CarouselPage

La navegación entre páginas



Nota: En el espíritu de la Sucintamente serie, en esta sección se explican los conceptos y temas de navegación de páginas más importantes. Sin embargo, hay consejos y consideraciones que son específicos para cada plataforma que usted tiene que saber cuando se trata de la navegación en Xamarin.Forms. Con respecto a esto, no se pierda la salida a la documentación oficial.

La mayoría de las aplicaciones móviles ofrecen su contenido a través de múltiples páginas. En Xamarin.Forms, navegar por sus páginas es muy simple debido a un marco de navegación incorporado. En primer lugar, en Xamarin.Forms que la navegación a través de la cuenta de apalancamiento **NavigationPage** objeto. Este tipo de página debe ser instanciada, pasando una instancia de la primera página en la pila de la navegación a su constructor. Esto normalmente se realiza en el archivo App.xaml.cs, donde se reemplaza la asignación de la **Página principal** propiedad con el siguiente código:

```
Aplicación público () {  
  
    InitializeComponent ();  
  
    MainPage = new NavigationPage (nuevo MainPage ()); }
```

Envolviendo una página raíz en una **NavigationPage** no sólo permitirá a la pila de navegación, sino que también permitirá a la barra de navegación en Android, iOS y escritorio de Windows (pero no en Windows 10 móvil que se basa en el botón de hardware espalda), cuyo texto será el valor de la **Título**

propiedad del objeto de página actual, representado por el **Página actual** propiedad de sólo lectura. Ahora supongamos que agregó otra página de tipo **Página de contenido** al proyecto PCL, llamado SecondaryPage.xaml. El contenido de esta página no es importante en este momento, acaba de establecer su **Título**

propiedad con un poco de texto. Si desea navegar desde la primera hasta la segunda página, se utiliza el **PushAsync** método como sigue:

```
esperar Navigation.PushAsync (nuevo SecondaryPage ());
```

los **Navegación** propiedades, expuestos por cada uno **Página** objeto, representa la pila de navegación a nivel de aplicación y proporciona métodos para la navegación entre las páginas de un enfoque LIFO (último en entrar, primero en salir). **PushAsync** navega a la instancia de página especificado; **PopAsync**, invocado desde la página actual, elimina la página actual de la pila y vuelve a la página anterior. Similar, **PushModalAsync** y **PopModalAsync** le permiten navegar por las páginas de forma modal. Las siguientes líneas de código demuestran esto:

```
// elimina SecondaryPage de la pila y se remonta a la página anterior
```

```
esperar Navigation.PopAsync ();
```

```
// Muestra la página especificada como una página modal
```

```
esperar Navigation.PushModalAsync (nuevo SecondaryPage ());
```

```
esperar Navigation.PopModalAsync ();
```

La figura 42 muestra cómo aparece la barra de navegación en Android y iOS cuando se navega a otra página.

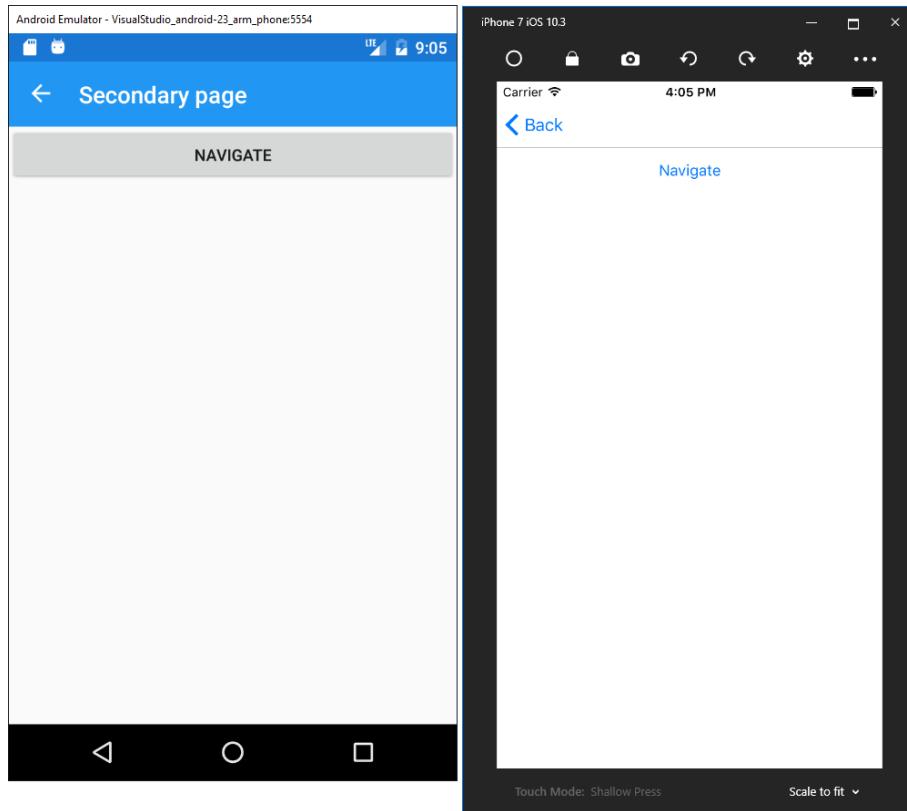


Figura 42: La barra de navegación que ofrece el objeto *NavigationPage*

Los usuarios pueden simplemente toque el botón Atrás de la barra de navegación para volver a la página anterior. Sin embargo, cuando se implementa la navegación modal, no se puede aprovechar el mecanismo integrado de navegación que ofrece la barra de navegación, por lo que es su responsabilidad para implementar el código que permite volver a la página anterior. Modal de navegación puede ser útil si tiene que ser capaz de interceptar un toque en el botón Atrás de cada plataforma. De hecho, los dispositivos Android y Windows tienen un hardware integrado en la parte trasera botón que se puede manejar con los acontecimientos, pero no lo hace iOS. En iOS, es suficiente con el botón de retroceso proporcionada por la barra de navegación, pero esto no se puede manejar con todos los eventos. Por lo tanto, en este caso, la navegación modal puede ser una buena opción para interceptar las acciones del usuario.

objetos que pasa entre páginas

La necesidad de intercambiar datos entre las páginas no es infrecuente. Puede cambiar o sobrecargar un **Página 'S constructor y requieren un parámetro del tipo deseado. Entonces, cuando se llama PushAsync** y pasar la instancia de la nueva página, usted será capaz de suministrar el argumento de que es necesario el constructor de la nueva página.

Animación de transiciones entre páginas

Por defecto, la navegación incluye una animación que hace que la transición de una página a otra más agradable. Sin embargo, puede desactivar animaciones simplemente pasando **falso** como argumento de **PushAsync** y **PushModalAsync**.

La gestión del ciclo de vida página

Cada Página objeto expone la **OnAppearing** y **OnDisappearing** eventos, plantearon justo antes de la página se representa y justo antes de que la página se elimina de la pila, respectivamente. Su código es el siguiente:

```
protected override OnAppearing void () {
```

```
    // Reemplazar con su código ...
```

```
    base.OnAppearing (); }
```

```
protected override OnDisappearing void () {
```

```
    // Reemplazar con su código ...
```

```
    base.OnDisappearing (); }
```

En realidad, estos eventos no están estrechamente relacionadas con la navegación, ya que están disponibles a cualquier página, incluyendo las páginas individuales. Sin embargo, es con la navegación que se vuelven muy importante, especialmente cuando se necesita para ejecutar un código en momentos específicos en la página del ciclo de vida. Para una mejor comprensión del flujo, piensa en la página constructor: esto se invoca la primera vez que se crea una página. Entonces, **OnAppearing** se eleva justo antes de la página se representa en la pantalla. Cuando la aplicación se desplaza a otra página, **OnDisappearing** se invoca, pero esto no destruye la instancia de página actual (y esto tiene mucho sentido). Cuando la aplicación se desplaza de vuelta de la segunda página para la primera página, esto no se crea de nuevo, ya que todavía está en la pila de navegación, por lo que no se invocará su constructor, mientras **OnAppearing** será. Por lo tanto, dentro de la

OnAppearing cuerpo del método, puede escribir código que se ejecuta cada vez que se muestra la página, mientras que en el constructor, puede escribir código que se ejecutará sólo una vez.

La manipulación del botón de hardware de vuelta

dispositivos Android y los teléfonos de Windows tienen un hardware integrado en la parte trasera botón que los usuarios pueden utilizar en lugar del botón de retroceso en la barra de navegación. Puede detectar si el usuario presiona el botón de hardware de vuelta por el manejo de la **OnBackButtonPressed** evento como sigue:

```
protected bool override OnBackButtonPressed () {
```

```
    base.OnBackButtonPressed retorno (); // reemplazar con su lógica aquí ...}
```

En pocas palabras su lógica en el cuerpo del método. El comportamiento por defecto es suspender la aplicación, así que sería bueno para anular esto con **PopAsync** para volver a la página anterior. Este evento no intercepta presionando el botón de retroceso en la barra de navegación, lo que implica que no tiene efecto en los dispositivos IOS.

Resumen del capítulo

Este capítulo introduce las páginas disponibles en Xamarin.Forms, explicando cómo puede mostrar una sola vista de contenido con el **Página de contenido** objeto, contenido de grupo en pestañas con el **TabPage**, contenido de golpe con la **CarouselPage**, y contenidos grupo en dos categorías con el **MasterDetail** objeto de página. En la segunda parte del capítulo, se examinó la forma en la **NavigationPage** objeto proporciona un marco de navegación incorporado que no sólo muestra una barra de navegación, sino que también permite la navegación entre páginas mediante programación. Por último, se analizó cómo funciona el ciclo de vida de la página, incluyendo la diferencia entre la creación de la página y la renderización de páginas. En el siguiente capítulo, que se verá en la información sobre dos características importantes y poderosos en Xamarin.Forms: recursos y el enlace de datos.

Capítulo 7 Recursos y enlace de datos

XAML es un lenguaje declarativo muy potente y que muestra toda su potencia con dos escenarios particulares: trabajar con los recursos y el enlace de datos. Si usted tiene experiencia con plataformas como WPF, Silverlight y plataforma Windows universal existente, usted estará familiarizado con los conceptos descritos en este capítulo. Si esta es la primera vez, usted podrá apreciar inmediatamente la forma en XAML simplifica las cosas difíciles en ambos escenarios.

Trabajar con recursos

En términos generales, en las plataformas basadas en XAML como WPF, Silverlight, la plataforma de Windows universal, y Xamarin.Forms, los recursos son piezas reutilizables de información que se puede aplicar a los elementos visuales en la interfaz de usuario. recursos XAML típicos son los estilos, las plantillas de control, las referencias a objetos, y las plantillas de datos. Xamarin.Forms apoya los estilos y las plantillas de datos, por lo que estos serán discutidos en este capítulo.



Consejo: Recursos en XAML son muy diferentes de los recursos en plataformas como Windows Forms, donde generalmente consume Resx archivos para incrustar cadenas, imágenes, íconos o archivos. Mi sugerencia es que usted no debe hacer ninguna comparación entre los recursos XAML y otros recursos de .NET.

La declaración de recursos

Cada Página objeto y el diseño expone una propiedad llamada **recursos**, una colección de recursos XAML que se puede llenar con uno o más objetos de tipo **ResourceDictionary**. UN **ResourceDictionary** es un contenedor de recursos XAML, como estilos, plantillas de datos, y las referencias a objetos. Por ejemplo, se puede añadir una **ResourceDictionary** a una página de la siguiente manera:

```
<ContentPage.Resources>
    <ResourceDictionary>
        <! - Añadir recursos aquí -> </
    ResourceDictionary>
</ContentPage.Resources>
```

Los recursos tienen alcance. Esto implica que los recursos que se agregan a nivel de página están disponibles para toda la página, mientras que los recursos que se agrega a nivel de diseño está disponible sólo para el diseño actual, como en el siguiente fragmento:

```
<StackLayout.Resources>
    <ResourceDictionary>
        <! - Los recursos están disponibles sólo a este diseño, no en el exterior -> </
    ResourceDictionary> </StackLayout.Resources>
```

A veces es posible que desee hacer que los recursos disponibles para toda la aplicación. En este caso, se puede aprovechar la **App.xaml** archivo. El código por defecto para este archivo se muestra en Listado de Código 14.

Listado de Código 14

```
<? Xml version = "1.0" encoding = "UTF-8"?> < Solicitud xmlns =
"http://xamarin.com/schemas/2014/forms"
    xmlns : x = "Http://schemas.microsoft.com/winfx/2009/xaml"
    x : Clase = "App1.App"> < Application.Resources
>

    <! - diccionario de recursos de aplicaciones ->
    < ResourceDictionary >

        </ ResourceDictionary >
    </ Application.Resources >
</ Solicitud >
```

Como se puede ver, el código autogenerado de este archivo ya contiene una **Application.Resources** nodo con un anidado **ResourceDictionary**. Recursos que ponen dentro de este diccionario de recursos serán visibles a cualquier página, el diseño y la vista en la aplicación. Ahora que tiene conocimiento de donde se declaran los recursos y su alcance, es el momento para ver cómo funcionan los recursos, a partir de estilos. Otros recursos, como las plantillas de datos, se discutirán más adelante en este capítulo.

La introducción de estilos

En el diseño de la interfaz de usuario, en algunas situaciones, es posible tener varias vistas del mismo tipo y, para cada uno de ellos, es posible que tenga que asignar las mismas propiedades con los mismos valores. Por ejemplo, es posible que tenga dos botones con la misma anchura y altura, o dos o más etiquetas con la misma anchura, la altura y la configuración de fuente. En tales situaciones, en lugar de asignar las mismas propiedades muchas veces, se puede tomar ventaja de estilos. Un estilo le permite asignar un conjunto de propiedades a las vistas del mismo tipo. Estilos deben ser definidas dentro de una

ResourceDictionary y deben especificar el tipo se destinan a un y un identificador. El código siguiente muestra cómo definir un estilo para **Etiqueta** puntos de vista:

```
<ResourceDictionary>
    <X Estilo: Clave = "LabelStyle" TargetType = "Etiqueta">
        <Setter propiedad = "TextColor" Value = "Green" /> <Setter propiedad
        = "Tamaño de Letra" Value = "grande" /> </ style>

</ ResourceDictionary>
```

Se asigna un identificador que tiene el **x: Key** expresión y el tipo de destino con **TargetType**, pasando el nombre tipo para la vista de destino. Los valores de propiedad se asignan con **Setter** elementos, cuya **Propiedad** propiedad representa el nombre de la propiedad de destino y **Valor** representa el valor de la propiedad. A continuación, asigne el estilo de **Etiqueta** puntos de vista de la siguiente manera:

```
<Etiquetas de texto = "Introduzca un texto:" style = "{} StaticResource LabelStyle" />
```

Por tanto, un estilo se aplica mediante la asignación de la **Estilo** propiedad de una vista con una expresión que encierra la **StaticResource** extensión de marcado y el identificador de estilo entre llaves. A continuación, puede asignar la **Estilo** propiedad de cada vista de ese tipo en lugar de asignar manualmente las mismas propiedades en todo momento. Con estilos, XAML soporta tanto **StaticResource** y

DynamicResource extensiones de marcado. En el primer caso, si un cambio de estilo, la vista de destino no se actualizará con el estilo fresco. En el segundo caso, la vista será actualizada refleja los cambios en el estilo.

herencia de estilos

Estilos apoyan herencia; Por lo tanto, puede crear un estilo que se deriva de otro estilo. Por ejemplo, puede definir un estilo que se dirige a lo abstracto **Ver** escribir como sigue:

```
<X Estilo: Clave = "ViewStyle" TargetType = "Ver">
    <Setter propiedad = "HorizontalOptions" Value = "center" /> <Setter propiedad
    = "VerticalOptions" Valor = "center" /> </ style>
```

Este estilo se puede aplicar a cualquier punto de vista, independientemente del tipo de hormigón. A continuación, puede crear un estilo más especializado mediante el **Residencia** en propiedad de la siguiente manera:

```
<Style x: Key = "LabelStyle" TargetType = "Etiqueta"
    BasedOn = "{} StaticResource ViewStyle"> <Setter propiedad
    = "TextColor" Value = "Green" /> </ style>
```

Los objetivos segundo estilo **Etiqueta** puntos de vista, sino que también hereda la configuración de propiedades del estilo padre. Poner de manera sucinta, la **LabelStyle** asignará el **HorizontalOptions**, **VerticalOptions**, y **Color de texto** propiedades en el objetivo **Etiqueta** puntos de vista.

estilo implícita

Puntos de vista' **Estilo** propiedad permite la asignación de un estilo definido dentro de los recursos. Esto le permite asignar de forma selectiva el estilo sólo a ciertos puntos de vista de un tipo dado. Sin embargo, si desea que el mismo estilo que se aplicará a todos los puntos de vista del mismo tipo en la interfaz de usuario, la asignación de la

Estilo propiedad para cada vista de forma manual puede ser molesto. En este caso, se puede tomar ventaja de la llamada *estilo implícita*. Esta característica le permite asignar automáticamente un estilo a todos los puntos de vista del tipo especificado con el **TargetType** propiedad sin la necesidad de establecer el **Estilo** propiedad. Para lograr esto, sólo tiene que evitar la asignación de un identificador con **x: Key**, como en el ejemplo siguiente:

```
<Estilo TargetType = "Etiqueta">
    <Setter propiedad = "HorizontalOptions" Value = "center" /> <Setter propiedad
    = "VerticalOptions" Valor = "center" /> <Setter propiedad = "TextColor" Value =
    "Green" /> </ style>
```

Estilos con ningún identificador se aplicarán automáticamente a toda la **Etiqueta** vistas en la interfaz de usuario (de acuerdo con el alcance del diccionario de recursos que contiene) y usted no tendrá que asignar el **Estilo** propiedad en el **Etiqueta** definiciones.

Trabajar con el enlace de datos

El enlace de datos es un mecanismo incorporado que permite a los elementos visuales para comunicarse con los datos de modo que la interfaz de usuario se actualiza automáticamente cuando cambian los datos y viceversa. El enlace de datos está disponible en todas las plataformas de desarrollo más importantes, y Xamarin.Forms no es una excepción. De hecho, sus datos de motor de enlace se basa en el poder de XAML y la forma en que funciona es similar en todas las plataformas basadas en XAML. Xamarin.Forms soporta la unión a un objeto a elementos visuales, una colección de elementos visuales, y elementos visuales a otros elementos visuales. En este capítulo se describen los dos primeros escenarios. Debido a que el enlace de datos es un tema muy complejo, la mejor manera de empezar es con un ejemplo. Supongamos que se desea enlazar una instancia de la siguiente **Persona**

clase a la interfaz de usuario, de modo que se establece un flujo de comunicación entre el objeto y vistas:

```
Public class Persona {
```

```
    NombreCompleto cadena pública {get; conjunto; } Public
    DateTime DateOfBirth {get; conjunto; } Cadena de dirección
    pública {get; conjunto; }}
```

En la interfaz de usuario, tendrá que permitir que el usuario introduzca su nombre completo, fecha de nacimiento y dirección a través de una **Entrada**, un **Selector de fechas**, y otro **Entrada**, respectivamente. En XAML, esto se puede lograr con el código que se muestra en **Listado de Código 15**.

Listado de Código 15

```
<? Xml version = "1.0" encoding = "UTF-8"?> < Pagina de contenido xmlns =
"http://xamarin.com/schemas/2014/forms"
    xmlns : x = "Http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns : local = "CLR-espacio de nombres: App1"
    Título = "Página Principal"
    x : Clase = "App1.MainPage">

    < StackLayout Orientación = "Vertical" Relleno = "20">
        < Etiqueta Texto = "Nombre:" /> < Entrada Texto = "{ Unión Nombre
        completo } "/>

        < Etiqueta Texto = "Fecha de nacimiento:" /> < Selector de fechas Fecha = "{ Unión Fecha de nacimiento , Modo
        = TwoWay} "/>

        < Etiqueta Texto = "Dirección:" /> < Entrada Texto = "{ Unión
        Dirección } "/> </ StackLayout >

    </ Pagina de contenido >
```

Como se puede ver, el **Texto** vivendas en **Entrada** puntos de vista y la **Fecha** propiedad de la **Selector de fechas** tener una expresión marcado como su valor. Tal expresión se compone de la **Unión** literal seguido de la propiedad que desea realizar la vinculación desde el objeto de datos. En realidad, la forma expandida de esta sintaxis podría ser { Binding Path = **PropertyName**} pero **Camino** puede ser omitido. El enlace de datos puede ser de cuatro tipos:

- **TwoWay**: Las vistas se pueden leer y escribir datos.
- **Un camino**: Vistas sólo pueden leer los datos.
- **OneWayToSource**: Vistas sólo pueden escribir datos.
- **Defecto**: Xamarin.Forms resuelve el modo apropiado de forma automática, en base a la vista (véase la explicación que sigue).

TwoWay y **Un camino** son los modos más utilizados, y en la mayoría de los casos no es necesario especificar el modo de forma explícita porque Xamarin.Forms resuelve automáticamente el modo adecuado en función de la vista. Por ejemplo, la unión en el **Entrada** control es **TwoWay** porque este tipo de vista se puede utilizar para leer y escribir datos, mientras que la unión en el **Etiqueta** control es **Un camino** porque este punto de vista puede leer sólo los datos. Sin embargo, con el **Selector de fechas**, es necesario establecer explícitamente el modo de unión, por lo que utilizar la siguiente sintaxis:

```
<DatePicker Fecha = "{Binding DateOfBirth, Modo = TwoWay}" />
```

propiedades de las opiniones de los que están obligados a propiedades de un objeto se conocen como *propiedades enlazables* (o las propiedades de dependencia si viene de la WPF o el mundo UWP).



Consejo: propiedades enlazables son muy potentes, pero un poco más complejo en la arquitectura. En este capítulo, voy a explicar cómo usarlos, pero para más detalles acerca de su aplicación y cómo se puede utilizar en los objetos personalizados, puede hacer referencia a la [documentación oficial](#).

propiedades enlazables se actualizará automáticamente el valor de la propiedad del objeto dependiente y se actualizará automáticamente su valor en la interfaz de usuario si se actualiza el objeto. Sin embargo, esta actualización automática es posible sólo si el objeto enlazado a datos implementa el **INotifyPropertyChanged** interfaz, que permite que un objeto para enviar notificaciones de cambio. Como consecuencia, debe extender la **Persona** definición de clase como se muestra en Listado de Código 16.

Listado de Código 16

```
utilizando Sistema;
utilizando System.ComponentModel;
utilizando System.Runtime.CompilerServices;

espacio de nombres app1 {

    pública clase Persona : INotifyPropertyChanged
    {
        privédo cuerda nombre completo;
        pública cuerda Nombre completo {

            obtener
```

```

{
    regreso nombre completo; }

conjunto
{
    fullName = valor ;
    OnPropertyChanged (); }}

privado Fecha y hora fecha de nacimiento;

público Fecha y hora Fecha de nacimiento {

obtener
{
    regreso fecha de nacimiento; }

conjunto
{
    fechaDeNacimiento = valor ;
    OnPropertyChanged (); }}

privado cuerda dirección;
público cuerda dirección {

obtener
{
    regreso dirección; }

conjunto
{
    dirección = valor ;
    OnPropertyChanged (); }}

evento público PropertyChangedEventHandler PropertyChanged;

private void OnPropertyChanged ([ CallerMemberName ] cuerda nombre de la propiedad
                            = nulo )
{
    PropertyChanged?.Invoke ( esta ,
                            nuevo PropertyChangedEventArgs (nombre de la propiedad)); }}}
```

mediante la implementación **INotifyPropertyChanged**, definidores de propiedad pueden elevar una notificación de cambio a través de la **PropertyChanged** evento. vistas con destino serán notificados de cualquier cambio y se actualizará su contenido.



Consejo: Con la `CallerMemberName` atributo, el compilador resuelve automáticamente el nombre del miembro de la persona que llama. Esto evita la necesidad de pasar el nombre de la propiedad en cada colocador y ayuda a mantener el código mucho más limpio.

El siguiente paso es vinculante una instancia de la **Persona** clase a la interfaz de usuario. Esto se puede lograr con las siguientes líneas de código, normalmente colocado en el interior constructor de la página o en su **OnAppearing** controlador de eventos:

```
Persona persona = new Persona ();
this.BindingContext = persona;
```

Páginas y diseños exponen la **BindingContext** propiedad, del tipo de **objeto**, que representa la fuente de datos de la página o el diseño y es lo mismo que **DataContext** en WPF o UWP. vistas secundarias que son los datos ligados a las propiedades de un objeto buscará una instancia del objeto en el **BindingContext** valor de la propiedad y se unen a las propiedades de este ejemplo. En este caso, el

Entrada y el **Selector de fechas** buscará una instancia de objeto en el interior **BindingContext** y que se unirán a las propiedades de esa instancia. Recuerde que XAML entre mayúsculas y minúsculas, por lo que la unión a

Nombre completo es diferente de la unión a **Nombre completo**. El tiempo de ejecución será una excepción si se intenta unirse a una propiedad que no existe o tiene un nombre diferente. Si ahora se intenta ejecutar la aplicación, no sólo los datos de trabajo vinculante, pero la interfaz de usuario también se actualizará automáticamente si cambia la fuente de datos. Es posible pensar en vistas unión a una única instancia de objeto, al igual que en el ejemplo anterior, como la unión a una fila de una tabla de base de datos.

Trabajando con colecciones y con la Vista de la lista

A pesar de trabajar con una sola instancia de objeto es un escenario común, otra situación muy común está trabajando con colecciones que se muestran como listas en la interfaz de usuario. Xamarin.Forms apoya unión sobre colecciones a través de los datos **ObservableCollection** **<T>** objeto. Esta colección funciona exactamente igual que el **Lista <T>**, pero también plantea una notificación de cambio cuando los artículos se añaden o eliminan de la colección. Las colecciones son de gran utilidad, por ejemplo, cuando se quiere representar filas de una tabla de base de datos. Por ejemplo, suponga que tiene la siguiente colección de **Persona** objetos:

```
person1 persona = new Persona {NombreCompleto = "Alessandro"}; person2 persona = new Persona
{NombreCompleto = "James"}; persona3 persona = new Persona {NombreCompleto = "Jacqueline"};
var personas = nueva ObservableCollection <persona> () {person1, person2,
    persona3};

this.BindingContext = pueblo;
```

El código asigna la colección a la **BindingContext** propiedad del contenedor raíz, pero en este punto, se necesita un elemento visual que es capaz de mostrar el contenido de esta colección. Aquí es donde el **Vista de la lista** el control entra en acción. La **Vista de la lista** puede recibir la fuente de datos, ya sea del **BindingContext** de su contenedor o mediante la asignación de su **ItemsSource** propiedad, y cualquier objeto que implemente la **IEnumerable** interfaz se puede utilizar con el **Vista de la lista**. Por lo general, va a asignar **ItemsSource** directamente si la fuente de datos para el **Vista de la lista** no es la misma fuente de datos que para los otros puntos de vista en la página.

El problema a resolver con la **Vista de la lista** es que no sabe cómo presentar los objetos en una lista. Por ejemplo, pensar en la colección de personas que contiene las instancias de la **Persona** clase. Cada instancia expone la **Nombre completo**, **Fecha de nacimiento**, y **Dirección** propiedades, pero la **Vista de la lista** no sabe cómo presentar estas propiedades, por lo que es su trabajo para explicar cómo a ella. Esto se logra con la llamada *plantillas de datos*. Un modelo de datos es un conjunto estático de puntos de vista que están enlazados a las propiedades en el objeto. Se instruye a la **Vista de la lista** sobre la forma de presentar artículos. plantillas de datos en Xamarin.Forms se basan en el concepto de células. Las células pueden mostrar información de una manera específica y se resumen en la Tabla 8.

Tabla 8: Las células en Xamarin.Forms

Tipo de célula	Descripción
TextCell	Muestra dos etiquetas, una con una descripción y una con un valor de texto enlazado a datos.
EntryCell	Muestra una etiqueta con una descripción y una Entrada con un valor de texto enlazado a datos. También permite un marcador de posición que se muestra.
ImageCell	Muestra una etiqueta con una descripción y una Imagen control con una imagen enlazada a datos.
SwitchCell	Muestra una etiqueta con una descripción y una Cambiar de control unido a una bool valor.
ViewCell	Permite la creación de plantillas de datos personalizados.



Consejo: Las etiquetas dentro de las células también son propiedades enlazables.

Por ejemplo, si sólo se tiene que mostrar y editar el **Nombre completo** propiedad, se podría escribir la siguiente plantilla de datos:

```
<Cuadrícula>
  <ListView x: Name = "PeopleList" ItemsSource = "{Binding}">
    <ListView.ItemTemplate>
      <DataTemplate>
        <EntryCell Etiqueta = "Nombre completo:" Text = "{Binding NombreCompleto}" /> </
        DataTemplate> </ListView.ItemTemplate> </ ListView> </ Cuadrícula>
```



Consejo: La DataTemplate definición siempre se define dentro de la ListView.ItemTemplate elemento.

Como regla general, si se ha asignado la fuente de datos a la BindingContext propiedad, el ItemsSource debe ser ajustado con el { Unión} valor, lo que significa que su fuente de datos es la misma que la de sus padres. Con este código, el Vista de la lista Se muestran todos los elementos de la colección unida, que muestran dos células de cada elemento. Sin embargo, cada Persona también expone una propiedad de tipo Fecha y hora, y ninguna célula es adecuado para eso. En tales situaciones, puede crear una célula personalizado mediante el ViewCell, como se muestra en Listado de Código 17.

Listado de Código 17

```
<? Xml version = "1.0" encoding = "UTF-8"?> < Pagina de contenido xmlns =
"http://xamarin.com/schemas/2014/forms"
    xmlns : x = "Http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns : local = "CLR-espacio de nombres: App1"
    Título = "Página Principal" Relleno = "20"
    x : Clase = "App1.MainPage">

< StackLayout > < Vista de la lista x : Nombre = "PeopleList" ItemsSource = "{ Unión }"

    HasUnevenRows = "True"> < ListView.ItemTemplate >
        < DataTemplate > < ViewCell > < ViewCell.View > < StackLayout
            Margen = "10">

                < Etiqueta Texto = "Nombre completo:" /> < Entrada Texto = "{ Unión Nombre
            completo } "/> < Etiqueta Texto = "Fecha de nacimiento:" /> < Selector de
            fechas Fecha = "{ Unión Fecha de nacimiento ,
                Modo = TwoWay} "/>
                < Etiqueta Texto = "Dirección:" /> < Entrada Texto = "{ Unión
            Dirección } "/> </ StackLayout >

            </ ViewCell.View >
        </ ViewCell >
    </ DataTemplate >
</ ListView.ItemTemplate >

</ Vista de la lista >
</ StackLayout >
</ Pagina de contenido >
```

Como se puede ver, el ViewCell le permite crear la costumbre y plantillas de datos complejos, contenida en el ViewCell.View propiedad, por lo que puede mostrar cualquier tipo de información que necesita. Note la HasUnevenRows propiedad: si cierto en Android y Windows, esta dinámica cambia el tamaño de la altura de una celda en función de su contenido. En iOS, esta propiedad se debe establecer en

falso y debe proporcionar una altura de fila fija mediante el establecimiento de la **Altura de la fila** propiedad. En el capítulo 8, aprenderá cómo tomar ventaja de la **OnPlatform** oponerse a tomar decisiones de interfaz de usuario basado en la plataforma.



Consejo: La Vista de la lista es una visión muy potente y versátil, y hay mucho más a él, como la interactividad, agrupación y clasificación, y las personalizaciones. Le recomiendo que lea el [documentación oficial sobre el Vista de la lista y esto artículo](#) que describe cómo mejorar el rendimiento, que es extremadamente útil con Android.

La Figura 43 muestra el resultado para el código descrito en esta sección. Observe que el **Vista de la lista** incluye una función de la capacidad de desplazamiento y nunca debe ser encerrado dentro de una **ScrollView**.

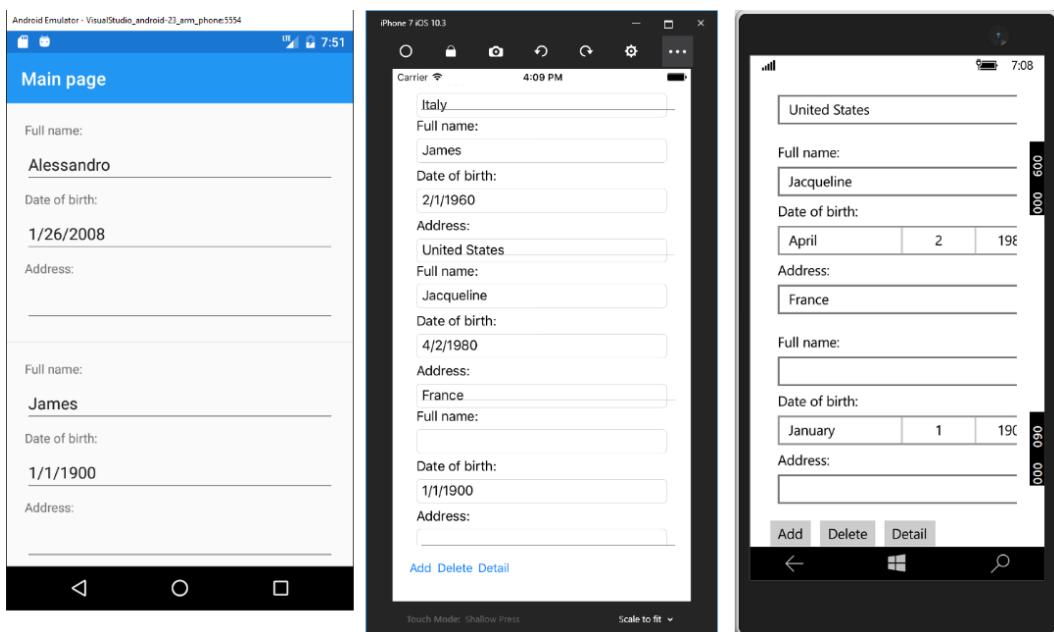


Figura 43: Un ListView enlazado a datos

Una plantilla de datos se puede colocar dentro de la página o aplicación de recursos para que sea reutilizable. A continuación, se asigna el **ItemTemplate** propiedad en el **Vista de la lista** con la definición **StaticResource** expresión, como se muestra en Listado de Código 18.

Listado de Código 18

```
<? Xml version = "1.0" encoding = "UTF-8"?> < Página de contenido xmlns =
"http://xamarin.com/schemas/2014/forms"
    xmlns : x = "Http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns : local = "CLR-espacio de nombres: App1"
    Título = "Página Principal"
    x : Clase = "App1.MainPage">

< ContentPage.Resources > < ResourceDictionary > < DataTemplate
    x : Llave = "MiPlantilla">
```

```

< ViewCell > < ViewCell.View > < StackLayout Margen = "10" Orientación = "Vertical"

        Relleno = "10">
        < Etiqueta Texto = "Nombre completo:" /> < Entrada Texto = "{ Unión Nombre completo } " /> < Etiqueta
        Texto = "Fecha de nacimiento:" /> < Selector de fechas Fecha = "{ Unión Fecha de
        nacimiento , Modo = Dos

    Camino} " />
        < Etiqueta Texto = "Dirección:" /> < Entrada Texto = "{ Unión
        Dirección } " /> < / StackLayout >

    </ ViewCell.View >
</ ViewCell >
</ DataTemplate >
</ ResourceDictionary >
</ ContentPage.Resources >

< Vista de la lista x : Nombre = "PeopleList" VerticalOptions = "FillAndExpand"
    HasUnevenRows = "True" ItemTemplate = "{ StaticResource MiPlantilla
} " /> </ Pagina de contenido >

```

Trabajar con el TableView

Cuando tenga que presentar una lista de parámetros, los datos en un formulario, o datos que es diferente de una fila a otra, se puede considerar la [TableView](#) controlar. Los **TableView** se basa en secciones y se puede mostrar el contenido a través de las mismas células que se describen anteriormente. Con este punto de vista, es necesario especificar un valor para su **Intención** propiedad, que básicamente representa el tipo de información que necesita para mostrar. Los valores posibles son (**ajustes** lista de ajustes), **Los datos** (para visualizar entradas de datos), **Formar**

(Cuando la vista de tabla actúa como un formulario), y **menú** (para presentar un menú de selecciones). Listado de Código 19 proporciona un ejemplo.

Listado de Código 19

```

<? Xml version = "1.0" encoding = "UTF-8"?> < Pagina de contenido xmlns =
"http://xamarin.com/schemas/2014/forms"
    xmlns : x = "Http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns : local = "CLR-espacio de nombres: App1"
    Título = "Página Principal"
    x : Clase = "App1.MainPage">

< ContentPage.Content > < TableView Intención =>
    "Configuración"
    < TableRoot > < TableSection Título = "Sección de la red">

        < SwitchCell Texto = "Mascotas" En = "True" /> < / TableSection
    >

```

```

< TableSection Título = ""> notificaciones Push
    < SwitchCell Texto = "Mascotas" En = "True" /> </ TableSection
>
</ TableRoot >
</ TableView >
</ ContentPage.Content >
</ Pagina de contenido >

```

Puede dividir el **TableView** en múltiples **TableSection**s. Cada uno contiene una celda para mostrar el tipo de información requerida, y, por supuesto, se puede utilizar una **ViewCell** para un encargo, plantilla más compleja. La figura 44 muestra un ejemplo de **TableView** A partir del listado anterior.

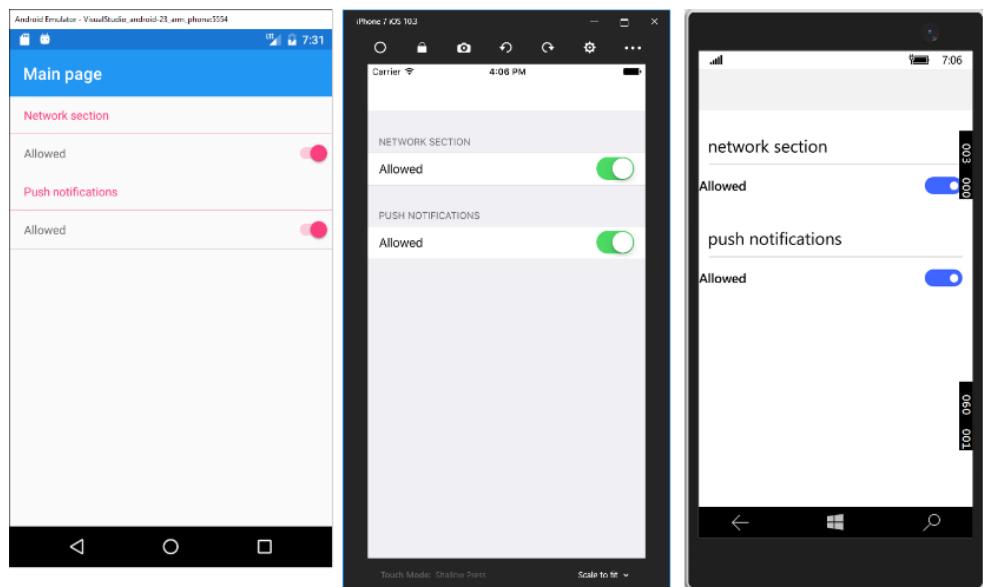


Figura 44: Un **TableView** en acción

Obviamente, puede enlazar las propiedades de células a objetos en lugar de establecer su valor de forma explícita como en el ejemplo anterior.

Mostrar y seleccionar los valores con el Recogedor ver

Con las aplicaciones móviles, es común proporcionar al usuario una opción para seleccionar un elemento de una lista de valores, que se puede lograr con la **Recogedor ver**. **Xamarin.Forms apoyo vinculante 2.3.4 datos** ha introducido en el **Recogedor**. Ahora puede unirse fácilmente a una **Lista <T>** o

ObservableCollection <T> a su **ItemsSource** propiedad y recuperar el elemento seleccionado a través de su **Item seleccionado** propiedad. Por ejemplo, suponga que tiene la siguiente **Fruta** clase:

```
Frutas public class {
```

```

Nombre cadena pública {get; conjunto; } Public
String color {get; conjunto; }
```

Ahora, en la interfaz de usuario, supongamos que desea pedir al usuario que seleccione una fruta de una lista con el XAML muestra en Listado de Código 20.

Listado de Código 20

```
<? Xml version = "1.0" encoding = "UTF-8"?> < Pagina de contenido xmlns =
"http://xamarin.com/schemas/2014/forms"
    xmlns : x = "Http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns : local = "CLR-espacio de nombres: App2"
    x : Clase = "App2.MainPage">

< ContentPage.Content >

    < StackLayout VerticalOptions = "FillAndExpand">
        < Etiqueta Texto = "Seleccione su fruta favorita:" /> < Recogedor x : Nombre = "FruitPicker" ItemDisplayBinding
        = "{ Unión Nombre }"
        "
            SelectedIndexChanged = "FruitPicker_SelectedIndexChanged"
        />
    </ StackLayout >
</ ContentPage.Content >
</ Pagina de contenido >
```

Como se puede ver, el **Recogedor** expone el **SelectedIndexChanged** evento, que se genera cuando el usuario selecciona un elemento. Con el **ItemDisplayBinding**, especifica qué propiedad del objeto dependiente que necesita para mostrar: en este caso, el nombre de la fruta. Los **ItemsSource** propiedad puede, en cambio, se asignará ya sea en XAML o en el código subyacente. En este caso, una colección puede ser asignado en C #, como se demuestra en Listado de Código 21.

Listado de Código 21

```
público clase parcial Pagina principal : Pagina de contenido
{
    público Pagina principal() {

        InitializeComponent ();

        var manzana = nuevo Fruta {Nombre = "Manzana" , Color = "Verde" };
        var = fresa nuevo Fruta {Nombre = "Fresa" , Color = "Rojo" };
        var naranja = nuevo Fruta {Nombre = "Naranja" , Color = "Naranja" };

        var fruitList = nuevo ObservableCollection < Fruta > () {Manzana, fresa, naranja};

        esta .FruitPicker.ItemsSource = fruitList;
    }

    privado vacío asíncrono FruitPicker_SelectedIndexChanged ( objeto remitente,
                                                    EventArgs mi)
    {

```

```

var currentFruit = esta .FruitPicker.SelectedItem como Fruta ;
Si (CurrentFruit! = nulo )
    esperar DisplayAlert ( "Selección" ,
        $ "Seleccionó { CurrentFruit.Name " , "DE ACUERDO" );
}

```

Al igual que el del mismo nombre propiedad en el **ListView**, **ItemsSource** es de tipo **objeto** y se puede unir a cualquier objeto que implementa la **IEnumerable** interfaz. Observe cómo se puede recuperar el elemento seleccionado manejo de la **SelectedIndexChanged** evento y la fundición

Picker.SelectedItem propiedad para el tipo que usted espera. En tales situaciones, es conveniente utilizar el **como** operador, lo que devuelve null si falla la conversión, en lugar de una excepción. La Figura 45 muestra cómo el usuario puede seleccionar un elemento en el selector.

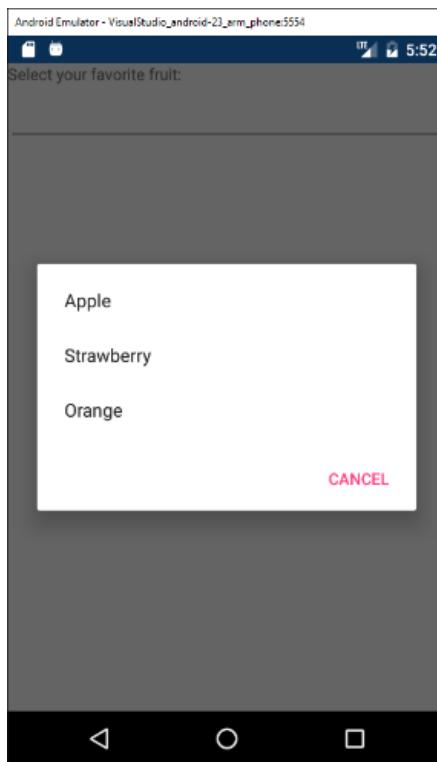


Figura 45: Selección de artículos con una Selector

En realidad, se añadió soporte de enlace de datos a la **Recogedor** Sólo con Xamarin.Forms 2.3.4. En versiones anteriores, sólo se podía rellenar manualmente su **Artículos** a través de la propiedad **Añadir** método y, a continuación, manejar índices. Esta es la verdadera razón por la **SelectedIndexChanged** Existe evento, pero sigue siendo útil con el nuevo enfoque. El enlace de datos de una lista a la **Recogedor** Es muy común, pero que sin duda puede todavía llenar la lista manualmente y manipular el índice.

imágenes vinculantes

La visualización de imágenes en escenarios de enlace de datos es muy común y Xamarin.Forms lo hace fácil. Usted sólo tendrá que obligar a la **Fuente de Imagen** propiedad a un objeto de tipo **Fuente de Imagen** o a una URL que se puede representar por tanto una cuerda y una Uri. Por ejemplo, suponga que tiene una clase con una propiedad que almacena la URL de una imagen de la siguiente manera:

```
Imagen public class {
```

```
    pública Uri PictureUrl {get; conjunto; }}
```

Cuando tenga una instancia de esta clase, se puede asignar el **PictureUrl** propiedad:

```
var picture1 = new Picture ();
picture1.PictureUrl = new Uri ("http://mystorage.com/myimage.jpg");
```

Suponiendo que usted tiene una **Imagen** Ver en el código XAML y una **BindingContext** asignado a una instancia de la clase, de unión tendría las características siguientes datos:

```
<Image Source = "{Binding} PictureUrl" />
```

XAML tiene un convertidor de tipos para el **Fuente de imagen** propiedad, por lo que se resuelve automáticamente las cadenas y Uri instancias en el tipo apropiado.

Consejos para convertidores de valores

La última frase de la sección anterior acerca del enlace imagen pone de relieve la existencia de convertidores de tipos que resolver tipos específicos en el tipo apropiado para el **Fuente de imagen** propiedad. En realidad, esto sucede con muchos otros puntos de vista y tipos. Por ejemplo, si enlaza un valor entero a la **Texto** propiedad de una **Entrada punto de vista**, un número entero tal se convierte en una cadena por un convertidor de tipo XAML. Sin embargo, hay situaciones en las que se necesita para unir objetos que XAML tipo convertidores no pueden convertir automáticamente en el tipo miras espera. Por ejemplo, es posible que desee unirse a una **Color** valor a una **Etiqueta 's Texto** propiedad, lo cual no es posible fuera de la caja. En estos casos, puede crear *convertidores de valores*. Un convertidor de valores es una clase que implementa la **IValueConverter** Interfaz y que debe exponer la **Convertir** y

ConvertBack métodos. **Convertir** traduce el tipo de original en un tipo que la vista puede recibir, mientras **ConvertBack** hace lo contrario. Listado de Código 22 muestra un ejemplo de un convertidor de valores que convierte una cadena que contiene el formato HTML en un objeto que se puede unir a la

WebView controlar. **ConvertBack** no se ha implementado, porque se supone que este convertidor de valores para ser utilizado en un escenario de sólo lectura y por lo tanto no es necesaria una conversión de ida y vuelta.

Listado de Código 22

```
utilizando Sistema;
utilizando System.Collections.Generic;
utilizando System.Globalization;
utilizando System.Linq;
utilizando System.Text;
utilizando System.Threading.Tasks;
```

```

utilizando Xamarin.Forms;

espacio de nombres app1 {

    público clase HTMLConverter: IValueConverter
    {
        público objeto Convertir( objeto valor, Tipo targetType,
            objeto parámetro, CultureInfo cultura) {

            tratar
            {
                var fuente = nuevo HtmlWebViewSource ();
                cuerda OriginalValue = ( cuerda )valor;

                source.Html = OriginalValue;
                regreso fuente; }

            captura ( Excepción ) {

                regreso valor; } }

        público objeto ConvertBack ( objeto valor, Tipo targetType,
            objeto parámetro, CultureInfo cultura)
        {
            arrojar nueva NotImplementedException (); } }
}

```

Ambos métodos siempre se reciben los datos de convertir como instancias de objeto, y entonces usted necesita para convertir el objeto en un tipo especializado para la manipulación. En este caso, **Convertir** crea una

HtmlWebViewSource objeto, convierte la recibida **objeto** en un **cuerda**, y rellena el

html propiedad con la cadena que contiene el código HTML. El convertidor de valores debe entonces ser declarada en los recursos del archivo XAML en los que desea utilizarlo (o en App.xaml). Listado de Código 23 proporciona un ejemplo que muestra también cómo utilizar el convertidor de valores.

Listado de Código 23

```

<? Xml version = "1.0" encoding = "UTF-8"?> < Pagina de contenido xmlns =
"http://xamarin.com/schemas/2014/forms"
    xmlns : x = "Http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns : local = "CLR-espacio de nombres: App1"
    Título = "Página Principal"
    x : Clase = "App1.MainPage">

< ContentPage.Resources >

```

```

< local : HTMLConverter x : Llave = "HTMLConverter" /></ ContentPage.Resources>
>

<! - supone que tiene un objeto .NET enlazado a datos que expone una propiedad llamada htmlContent ->
< WebView Fuente = "{ Unión htmlContent , Convertidor = { StaticResource HTMLConverter } } " /> </ Pagina de contenido >

```

Se declara el convertidor como lo haría con cualquier otro recurso. Luego, su unión también contendrá el **Convertidor** expresión, que apunta al convertidor de valores con la sintaxis típica que se utiliza con otros recursos.

La introducción de Model-View-ViewModel

Modelo-Vista-ViewModel (MVVM) es un patrón de arquitectura utilizada en plataformas basadas en XAML que permite la separación limpia entre los datos (modelo), la lógica (vista del modelo), y la interfaz de usuario (vista). Con MVVM, páginas sólo contienen código relacionado con la interfaz de usuario, que tiene muy en cuenta el enlace de datos, y la mayor parte del trabajo se hace en el modelo de vista. MVVM puede ser bastante complejo si usted nunca ha visto antes, así que voy a tratar de simplificar las explicaciones tanto como sea posible, pero debería utilizar Xamarin de [documentación MVVM](#) como una referencia. Vamos a empezar con un ejemplo sencillo y con una solución Xamarin.Forms frescas sobre la base de la estrategia de código compartido PCL. Imagine que desea trabajar con una lista de **Persona** objetos. Este es el modelo y se puede volver a utilizar el **Persona** clase de antes. Añadir una nueva carpeta llamada **Modelo** a su proyecto, y añadir un nuevo **Person.cs** archivo de clase a esta carpeta, pegar en el código de la **Persona** clase. A continuación, añadir una nueva carpeta llamada **ViewModel** al proyecto y añadir un nuevo archivo de clase llamada **PersonViewModel.cs**.

Antes de escribir el código para ello, vamos a resumir algunas consideraciones importantes:

- El modelo de vista contiene la lógica de negocio, actúa como un puente entre el modelo y la vista, y expone las propiedades a las que la vista se puede unir.
- Entre tales propiedades, uno ciertamente ser una colección de **Persona** objetos.
- En el modelo de vista, se pueden cargar datos, filtrar los datos, ejecutar operaciones de guardar, y los datos de la consulta.

La carga, depuración, el ahorro y la consulta de datos son ejemplos de acciones de un modelo de vista puede ejecutar en los datos. En un enfoque de desarrollo clásico, manejaría **hecho clic** eventos en **Botón**

puntos de vista y escribir el código que ejecuta una acción. Sin embargo, en MVVM, puntos de vista deben sólo contienen código relacionado con la interfaz de usuario, no código que ejecuta acciones en contra de los datos. Así, en MVVM, modelos de vista exponen en la denominada **comandos**. Un comando es una propiedad de tipo **Yo ordeno** que pueden ser datos ligados a puntos de vista tales como **Botón**, **SearchBar**, **ListView**, y

TapGestureRecognizer objetos. En la interfaz de usuario, se enlaza un objeto de un comando en el modelo de vista. De esta manera, la acción se ejecuta en el modelo de vista en lugar de en el código de la vista atrás. Listado de Código 24 muestra el **PersonViewModel** definición de clase.

Listado de Código 24

```

utilizando MvvmSample.Model;
utilizando Sistema;

```

```

utilizando System.Collections.ObjectModel;
utilizando System.Windows.Input;
utilizando Xamarin.Forms;

espacio de nombres MvvmSample.ViewModel {

    público clase PersonViewModel
    {
        público ObservableCollection < Persona > Las personas { obtener ; conjunto ; }
        público Persona SelectedPerson { obtener ; conjunto ; }

        público Yo ordeno addPerson { obtener ; conjunto ; }
        público Yo ordeno DeletePerson { obtener ; conjunto ; }

        público PersonViewModel () {

            esta La gente = nuevo ObservableCollection < Persona >();

            // Data de muestra
            Persona person1 =
                nuevo Persona {FullName = "Alessandro" , Dirección = "Italia" ,
                DateOfBirth = nuevo Fecha y hora (1977,5,10)};

            Persona person2 =
                nuevo Persona {FullName = "James" , Dirección = "Estados Unidos" ,
                DateOfBirth = nuevo Fecha y hora (1960,2,1)};

            Persona persona3 =
                nuevo Persona {FullName = "Jacqueline" , Dirección = "Francia" ,
                DateOfBirth = nuevo Fecha y hora (1980,4,2)};

            esta .People.Add (person1);
            esta .People.Add (Person2);
            esta .People.Add (persona3);

            esta .AddPerson =
                nuevo Mando ((() => esta .People.Add ( nuevo Persona ())));

            esta .DeletePerson =
                nuevo Mando < Persona > ((Persona) => esta .People.Remove (persona))
        ;
    }
}

```

los **Gente** y **SelectedPerson** propiedades exponen una colección de **Persona** objetos y un solo **Persona**, respectivamente, y el último estará sujeto a la **Item seleccionado** propiedad de una **Vista de la lista**, como se verá en breve. Observe cómo las propiedades de tipo **Yo ordeno** son asignados con las instancias de la **Mando** clase, a la que se puede pasar un **Acción** delegar a través de una expresión lambda que ejecuta la operación deseada. Los **Mando** proporciona una aplicación fuera de la caja de la **Yo ordeno** interfaz y su constructor también puede recibir un parámetro, en cuyo caso debe utilizar su sobrecarga genérica (ver **DeletePerson** asignación). En ese caso, el **Mando** trabaja con objetos de tipo **Persona** y la acción se ejecuta contra el objeto recibido. Comandos y otras propiedades son los datos ligados a las vistas en la interfaz de usuario.



Nota: Aquí he demostrado el uso más básico de los comandos. Sin embargo, también ordena exponer una CanExecute método booleano que determina si una acción se puede ejecutar o no. Además, puede crear comandos personalizados que implementan Yo ordeno y debe poner en práctica de manera explícita la Ejecutar y CanExecute métodos, donde Ejecutar se invoca para ejecutar una acción. Para más detalles, ver el [documentación oficial](#).

Ahora es el momento de escribir el código XAML para la interfaz de usuario. Listado de Código 25 muestra cómo utilizar una **Vista de la lista** para esto y cómo enlazar dos **Botón** vistas a los comandos.

Listado de Código 25

```
<? Xml version = "1.0" encoding = "UTF-8"?> < Pagina de contenido xmlns =
"http://xamarin.com/schemas/2014/forms"
    xmlns : x = "Http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns : local = "CLR-espacio de nombres: MvvmSample"
    x : Clase = "MvvmSample.MainPage" Relleno = "20">

    < StackLayout > < Vista de la lista x : Nombre = "PeopleList"
        ItemsSource = "{ Unión Gente }"
        HasUnevenRows = "True"
        Item seleccionado = "{ Unión SelectedPerson } "> < ListView.ItemTemplate
            > < DataTemplate > < ViewCell > < ViewCell.View > < StackLayout
                Margen = "10">

                    < Etiqueta Texto = "Nombre completo:" /> < Entrada Texto = "{ Unión Nombre
completo } "/> < Etiqueta Texto = "Fecha de nacimiento:" /> < Selector de
fechas Fecha = "{ Unión Fecha de nacimiento ,
Modo = TwoWay} "/>
                    < Etiqueta Texto = "Dirección:" /> < Entrada Texto = "{ Unión
Dirección } "/> </ StackLayout >

                </ ViewCell.View >
            </ ViewCell >
        </ DataTemplate >
```

```

</ ListView.ItemTemplate >
</ Vista de la lista > < StackLayout Orientación = "Horizontal">

    < Botón Texto = "Añadir" Mando = "{ Unión addPerson }" /> < Botón Texto = "Borrar" Mando
    = "{ Unión DeletePerson }"
        CommandParameter = "{ Unión Fuente = { x : Referencia PeopleLi
st },
        Camino = SelectedItem} "/>
    </ StackLayout >
</ StackLayout >
</ Pagina de contenido >

```



Consejo: Recuerde que debe establecer HasUnevenRows a falso y para proporcionar una Altura de la fila para el Vista de la lista en IOS.

los **Vista de la lista** es muy similar al ejemplo mostrado en la introducción de los datos de unión a las colecciones. Sin embargo, observe cómo:

- los **ListView.ItemsSource** la propiedad está ligado a la **Gente** colección en el modelo de vista.
- los **ListView.SelectedItem** la propiedad está ligado a la **SelectedPerson** propiedad en el modelo de vista.
- El primero **Botón** está unido a la **addPerson** comando en el modelo de vista.
- El segundo **Botón** está unido a la **DeletePerson** comando y pasa el seleccionado **Persona** objeto en el **Vista de la lista** con una expresión de unión especial: **Fuente** representa la fuente de datos, en este caso el **Vista de la lista**, que se refiere a **x: Referencia**; **Camino** apunta a la propiedad de la fuente que expone el objeto que desea pasar a la orden como un parámetro (denominado simplemente parámetro de comando).

El paso final es la creación de una instancia de la modelo de vista y asignarlo a la **BindingContext**

La página, que se puede hacer en la página de código subyacente, como se demuestra en Listado de Código 26.

Listado de Código 26

```

utilizando MvvmSample.ViewModel;
utilizando Xamarin.Forms;

espacio de nombres MvvmSample {

    público clase parcial Pagina principal : Pagina de contenido
    {
        // No usar un campo aquí ya que las propiedades // están
        optimizados para el enlace de datos.
        privado PersonViewModel ViewModel { obtener ; conjunto ; }

        público Pagina principal() {

            InitializeComponent ();

```

```

esta .ViewModel = nuevo PersonViewModel ();
esta .BindingContext = esta .ViewModel;
}}

```

Si ahora ejecuta la aplicación (ver Figura 46), verá la lista de **Persona** objetos, usted será capaz de utilizar los dos botones, y el beneficio real es que toda la lógica está en el modelo de vista. Con este enfoque, si cambia la lógica en las propiedades o en los comandos, no será necesario cambiar el código de la página. En la figura 46, se puede ver un nuevo **Persona** objeto añadió a través de comandos de unión.

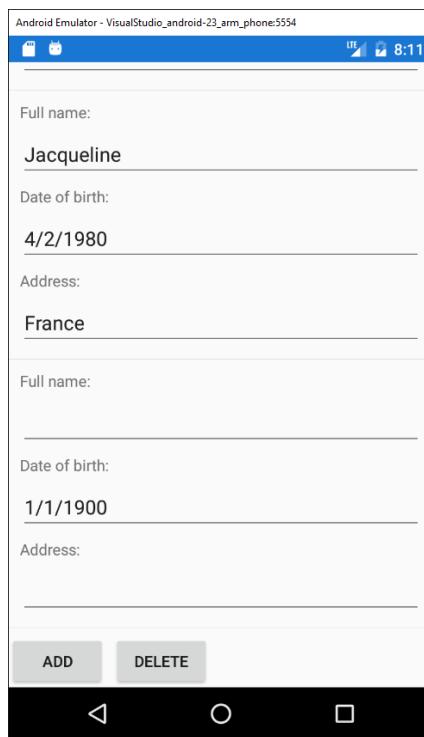


Figura 46: Mostrar una lista de las personas y la adición de una nueva persona con MVVM

MVVM es muy potente, pero las implementaciones del mundo real puede ser muy complejo. Por ejemplo, si usted quiere navegar a otra página y tiene comandos, el modelo de vista debe contener código relacionado con la interfaz de usuario (el lanzamiento de una página) que no se adhiere a los principios de MVVM. Obviamente, hay soluciones para este problema que requiere un mayor conocimiento del patrón, por lo que recomiendo nos fijamos en libros y artículos en Internet para su posterior estudio. Además, mi recomendación es no reinventar la rueda: muchas bibliotecas robusta y popular MVVM ya existen y es posible que desee elegir uno de entre los siguientes:

- [Prisma](#)
- [Kit de herramientas de MVVM Light](#)
- [FreshMvvm](#)
- [MvvmCross](#)

He trabajado personalmente con FreshMvvm, pero todas las alternativas antes mencionadas son lo suficientemente potente como para ahorrarle mucho tiempo.

Resumen del capítulo

XAML juega un papel fundamental en Xamarin.Forms y permite la definición de los recursos reutilizables y para los escenarios de enlace de datos. Los recursos son reutilizables estilos, plantillas de datos, y las referencias a objetos se declaran en XAML. En particular, estilos le permiten configurar las mismas propiedades a todas las vistas del mismo tipo y se pueden extender a otros estilos con la herencia. XAML también incluye un potente motor de enlace de datos que le permite enlazar rápidamente los objetos a los elementos visuales en un flujo de comunicación de dos vías.

En este capítulo, usted ha visto cómo enlazar tanto un objeto único y una colección de elementos visuales individuales y para la **Vista de la lista**, respectivamente. Usted ha visto cómo definir plantillas de datos para que el **Vista de la lista** puede tener conocimiento de cómo se deben presentar artículos, y que ha aprendido sobre convertidores de valores, objetos especiales que vienen a ayudar cuando se quiere unir objetos de un tipo que es diferente del tipo de una vista apoya. En la segunda parte del capítulo, que entramos por una introducción al patrón Model-View-ViewModel, centrándose en la separación de la lógica de la interfaz de usuario y la comprensión de nuevos objetos y conceptos tales como comandos. Hasta el momento, sólo se ha trabajado con los objetos y las vistas que ofrece Xamarin.Forms fuera de la caja, pero más a menudo que no, tendrá que poner en práctica las funciones más avanzadas que requieren API nativas. Esto lo que aprenderá en el siguiente capítulo.

API específicas de la plataforma Capítulo 8 Acceso a las

Hasta ahora, usted ha visto lo que Xamarin.Forms ofrece en términos de características que están disponibles en cada plataforma soportada, caminando a través de las páginas, diseños y controles que mostrar las propiedades y capacidades que sin duda se ejecutan en Android, iOS y Windows. Aunque esto simplifica el desarrollo multiplataforma, no es suficiente para construir aplicaciones móviles en el mundo real. De hecho, más a menudo que no, aplicaciones móviles necesitan acceder a los sensores, el sistema de archivos, la cámara, y la red; enviar notificaciones push; y más. Cada sistema operativo gestiona estas características con las API nativas que no se pueden compartir a través de plataformas y, por tanto, que Xamarin.Forms no pueden asignar en objetos multiplataforma.

Sin embargo, si Xamarin.Forms no proporcionan una manera de acceder a las API nativas, no sería muy útil. Por suerte, Xamarin.Forms ofrece múltiples maneras de acceder a las API específicas de la plataforma que se pueden utilizar para acceder a prácticamente todo de cada plataforma. Por lo tanto, no hay límite a lo que puede hacer con Xamarin.Forms. Con el fin de acceder a las funciones de la plataforma, tendrá que escribir código C # en cada proyecto de plataforma. Esto es lo que este capítulo se explica, junto con todas las opciones que tiene para acceder a las API de iOS, Android y Windows desde su base de código compartido.

los Dispositivo clase y la OnPlatform método

los **Xamarin.Forms** espacio de nombres expone una clase importante llamada **Dispositivo**. Esta clase le permite detectar la plataforma de su aplicación se está ejecutando en el lenguaje y dispositivo (tableta, teléfono, escritorio). Esta clase es particularmente útil cuando se necesita para ajustar la interfaz de usuario basada en la plataforma. El código siguiente muestra cómo tomar ventaja de la

Device.RuntimePlatform propiedad para detectar la plataforma de funcionamiento y para tomar decisiones relacionadas con la interfaz de usuario, en base a su valor:

```
// Label1 es una vista Label en el comutador de interfaz de
usuario (Device.RuntimePlatform) {

    Device.iOS caso:
        Label1.FontSize = Device.GetNamedSize (NamedSize.Large, Label1); descanso;

    Device.Android caso:
        Label1.FontSize = Device.GetNamedSize (NamedSize.Medium, Label1); descanso;

    Device.WinPhone caso:
        Label1.FontSize = Device.GetNamedSize (NamedSize.Medium, Label1); descanso;

    Device.Windows caso:
        Label1.FontSize = Device.GetNamedSize (NamedSize.Large, Label1); descanso; }
```

RuntimePlatform es de tipo **cuerda** y se puede comparar fácilmente contra constantes específicas llamadas **iOS**, **Androide**, **WinPhone**, y **ventanas** que representan, con nombres autoexplicativos, las plataformas soportadas. Los **GetNamedSize** método resuelve automáticamente el **De manera predeterminada, la micro, pequeña, Medio, y Grande** plataforma de tamaño de la fuente y devuelve el correspondiente **doble**, lo que evita la necesidad de suministrar valores numéricos que sería diferente para cada plataforma. Los

Device.Idiom la propiedad le permite determinar si el dispositivo actual de la aplicación se está ejecutando en un teléfono, tableta o PC de escritorio (UWP solamente), y devuelve uno de los valores de la **TargetIdiom** enumeración:

```
interruptor (Device.Idiom) {
```

```
    TargetIdiom.Desktop caso:
```

```
        // ruptura de escritorio  
        uwp;
```

```
    TargetIdiom.Phone caso:
```

```
        // Móviles  
        rompen;
```

```
    TargetIdiom.Tablet caso:
```

```
        // Tablets  
        rompen;
```

```
    TargetIdiom.Unsupported caso:
```

```
        // dispositivos no compatibles se  
        rompen; }
```

También puede decidir cómo ajustar los elementos de interfaz de usuario basado en la plataforma y lenguaje en XAML. Listado de Código 27 muestra cómo ajustar el **Relleno** propiedad de una página, basado en la plataforma.

Listado de Código 27

```
<? Xml version = "1.0" encoding = "UTF-8"?> < Página de contenido xmlns =  
"http://xamarin.com/schemas/2014/forms"  
    xmlns : x = "Http://schemas.microsoft.com/winfx/2009/xaml"  
    xmlns : puntos de vista = "clr-namespace: App1.Views"  
    x : Clase = "App1.Views.MainPage"> < ContentPage.Padding  
> < OnPlatform x : TypeArguments = "Espesor"  
  
    iOS = "0, 20, 0, 0"  
    Androide = "0, 10, 0, 0"  
    WinPhone = "0, 10, 0, 0" />  
  
</ ContentPage.Padding >  
< Página de contenido >
```

Con el **OnPlatform** etiqueta, puede especificar un valor de propiedad diferente, basado en el **iOS**, **Android**, y **WinPhone** plataformas. El valor de la propiedad depende de la **x: TypeArguments** atributo, que representa el tipo de .NET para la propiedad, **Espesor** en este caso particular. Del mismo modo, se puede trabajar con **OnIdiom** y el **TargetIdiom** enumeración en XAML, también.



Consejo: En iOS, es más práctico para definir un acolchado de la página 20 de la parte superior, como en el fragmento anterior. La razón es que si no lo hace, su página se superpondrá a la barra del sistema.

Trabajar con el servicio de dependencia

La mayoría de las veces, las aplicaciones móviles necesitan ofrecer interacción con el hardware del dispositivo, sensores, aplicaciones del sistema, y el sistema de archivos. El acceso a estos rasgos de código compartido no es posible, debido a que sus APIs tienen implementaciones únicas en cada plataforma. Sin embargo, Xamarin.Forms ofrece una solución simple a este problema que se basa en el modelo de servicio de localización: en el proyecto compartido, se escribe un interfaz que define las funcionalidades requeridas, a continuación, dentro de cada proyecto de plataforma, que escribe las clases que implementan la interfaz nativa a través API. Por último, se utiliza el

DependencyService clase y su **Obtener** Método para recuperar la correcta aplicación basada en la plataforma de su aplicación se está ejecutando. Por ejemplo, supongamos que su aplicación necesita para trabajar con bases de datos SQLite locales. Suponiendo que usted ha instalado el [SQLite-Net-PCL](#) NuGet paquete en su solución, en el proyecto PCL, se puede escribir la siguiente interfaz de ejemplo denominado

IDatabaseConnection que define la firma de un método que debe devolver la ruta de la base de datos:

```
IDatabaseConnection interfaz pública {
```

```
    SQLite.SQLiteConnection DbConnection (); }
```



Consejo: Un recorrido completo de la utilización de bases de datos SQLite locales en Xamarin.Forms está disponible en [MSDN Magazine](#) Del autor de este libro electrónico.

En este punto, en cada proyecto de plataforma, es necesario proporcionar una implementación de esta interfaz, ya que los nombres de archivos, nombres de ruta, y, más en general, el sistema de archivos son específicos de la plataforma. Añadir un nuevo archivo de clase llamada **DatabaseConnection.cs** a los proyectos de iOS, Android y Windows. Listado de Código 28 proporciona la aplicación de iOS, Listado de Código 29 proporciona la implementación de Android, y Listado de Código 30 proporciona la implementación de Windows.

Listado de Código 28

```
utilizando Sistema;  
utilizando SQLite;  
utilizando System.IO;  
utilizando App1.iOS;  
  
[ Asamblea : Xamarin.Forms. Dependencia ( tipo de ( DatabaseConnection ))]  
espacio de nombres App1.iOS {  
  
    público clase DatabaseConnection : IDatabaseConnection
```

```

{
    público SQLiteConnection DbConnection () {

        cuerda dbName = "MyDatabase.db3";
        cuerda personalFolder =
            Sistema. Ambiente . GetFolderPath ( Ambiente . SpecialFolder .Personal);

        cuerda libraryFolder =
            Camino .Combine (personalFolder, "..", "Biblioteca");
        cuerda path = Camino .Combine (libraryFolder, dbName);
        return new SQLiteConnection (camino); }}}
```

Listado de Código 29

```

utilizando Xamarin.Forms;
utilizando App1.Droid;
utilizando SQLite;
utilizando System.IO;

[ Asamblea : Dependencia ( tipo de ( DatabaseConnection ))]
espacio de nombres App1.Droid {

    público clase DatabaseConnection : IDatabaseConnection
    {
        público SQLiteConnection DbConnection () {

            cuerda dbName = "MyDatabase.db3";
            cuerda path = Camino .Combine (Sistema. Ambiente . GetFolderPath
                (Sistema. Ambiente .
                SpecialFolder .Personal), dbName);
            return new SQLiteConnection (camino); }}}
```

Listado de Código 30

```

utilizando SQLite;
utilizando Xamarin.Forms;
utilizando System.IO;
utilizando Windows.Storage;
```

utilizando App1.UWP;

```
[ Asamblea : Dependencia ( tipo de ( DatabaseConnection ))]
espacio de nombres App1.UWP {

    público clase DatabaseConnection : IDatabaseConnection
    {
        público SQLiteConnection DbConnection () {

            {
                cuerda dbName = "MyDatabase.db3" ;
                cuerda path = Camino .Combinar( Datos de la aplicación .
                    Current.LocalFolder.Path, dbName);
                return new SQLiteConnection (camino); }}}
```

Común a cada aplicación específica de la plataforma es decorar el espacio de nombres con el **Dependencia** atribuir, asignado a nivel de montaje, que identifica de forma única la aplicación de la **IDatabaseConnection** interfaz en tiempo de ejecución. En el **DbConnection** cuerpo del método, se puede ver cómo cada plataforma aprovecha sus propias APIs para trabajar con nombres de archivo. En el proyecto PCL, sólo tiene que resolver la correcta aplicación de la **IDatabaseConnection** interfaz como sigue:

```
// Obtener la conexión a la base de datos de
SQLiteConnection
base de datos = DependencyService.Get <IDatabaseConnection> () DbConnection ();
```

los **DependencyService.Get** método genérico recibe la interfaz como el parámetro de tipo y resuelve la aplicación de esa interfaz de acuerdo con la plataforma actual. Con este enfoque, usted no necesita preocuparse por la determinación de la plataforma actual y la invocación de las implementaciones nativas correspondientes, ya que el servicio de dependencia hace el trabajo para usted. Este enfoque se aplica a todas las API nativas que necesita para invocar, y proporciona la opción más potente para acceder a las características específicas de la plataforma en Xamarin.Forms.

Trabajar con plugins

Al tener acceso a las API nativas, la mayoría de las veces es su necesidad real para acceder a funciones que existen varias plataformas, pero con las API que son totalmente diferentes unos de otros. Por ejemplo, los dispositivos iOS, Android y Windows todos tener una cámara, todos ellos tienen un sensor GPS que devuelve la ubicación actual, y así sucesivamente. Para los escenarios en los que es necesario trabajar con las **capacidades que existen multiplataforma**, puede aprovechar [plugins](#). Estas son las bibliotecas que consisten en una implementación abstracta de API nativas que proporcionan capacidades disponibles multiplataforma. También evitan la necesidad de utilizar el servicio de dependencia y escribir código específico de la plataforma en un gran número de

situaciones. Los plugins son de código libre y abierto, y están disponibles como paquetes NuGet. Una lista actualizada de plugins disponibles está en [GitHub](#).

Entre otros, los plugins populares son el complemento de Conectividad (lo que hace que sea fácil de manejar conectividad de red), el plug-in Media (lo que hace que sea sencillo para capturar imágenes y videos desde el proyecto PCL), y el plugin Geolocalizador (que proporciona una abstracción de acceso geolocalización). Por ejemplo, supongamos que desea detectar si una conexión de red está disponible antes de acceder a Internet en su aplicación. Puede utilizar el Administrador de NuGet paquete para descargar e instalar el plugin de conectividad se muestra en la Figura 47. Para cada plugin, hay un enlace a la página de documentación en GitHub, lo que sin duda le recomiendo que visite cuando se utiliza ningún tipo de plugins.

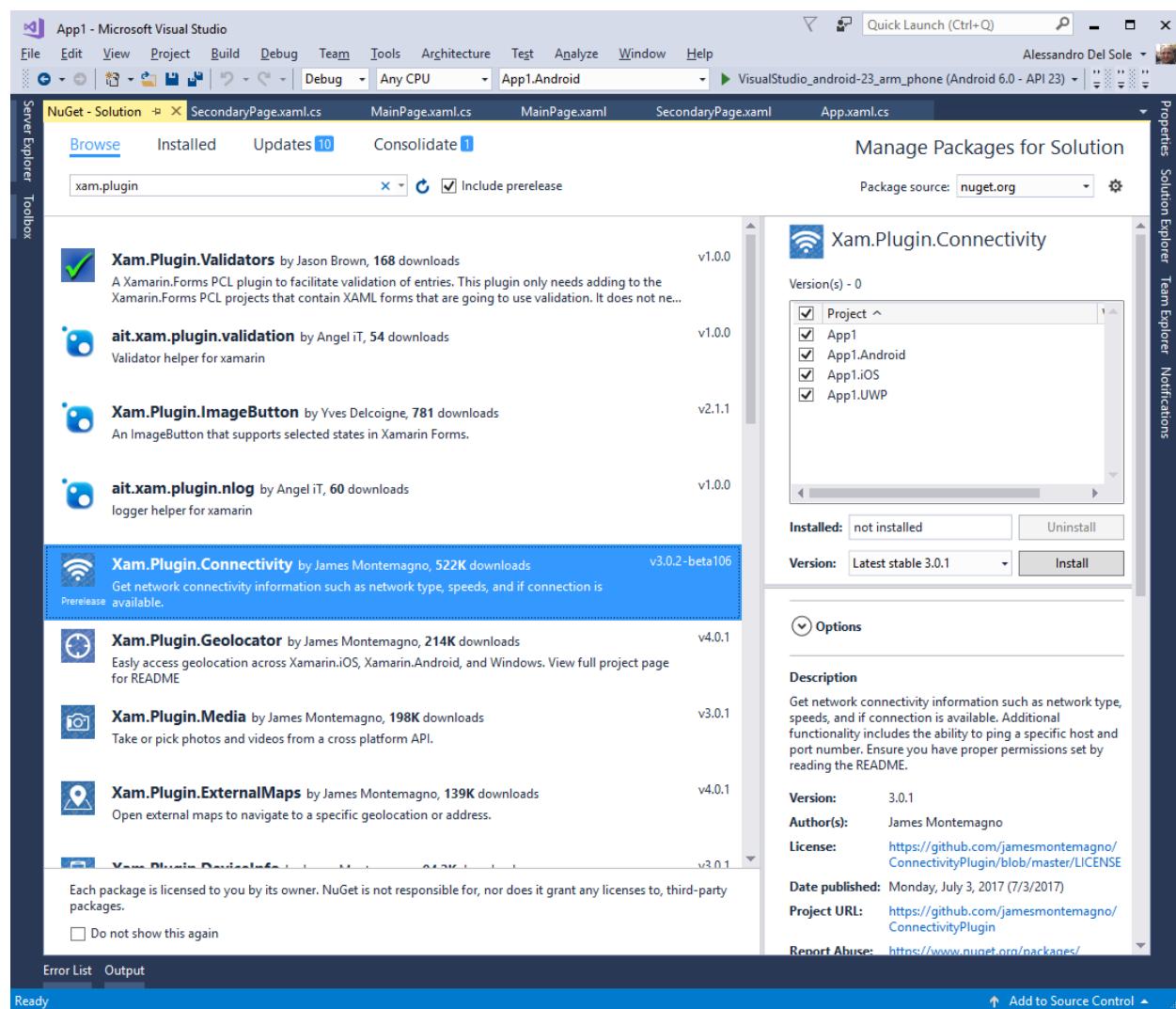


Figura 47: Instalación de plugins

Asegúrese de seleccionar todos los proyectos de la solución (en el cuadro de la derecha), y luego hace clic **Instalar**. No voy a entrar en la arquitectura de plugins' aquí, sólo voy a explicar cómo usarlos. Si está interesado en su arquitectura, se puede leer esto [entrada en el blog](#) Del equipo de Xamarin. Como regla general, el espacio de nombres raíz de un plugin expone una clase singleton que expone la solicitud

característica. Por ejemplo, la raíz **Plugin.Connectivity** expone el espacio de nombres **CrossConnectivity** clase, cuya **Corriente** propiedad representa la instancia singleton que se puede utilizar de la siguiente manera en el código compartido, y por lo tanto sin la necesidad de trabajar con los proyectos de plataforma:

```
si (CrossConnectivity.Current.isConnected) {  
    // conexión está disponible}  
  
CrossConnectivity.Current.ConnectivityChanged +=  
    ((Remitente, e) =>  
    {  
        // Estado de la conexión cambió});
```

Entre otros, esta clase expone la **Está conectado** propiedad, que devuelve verdadero si una conexión de red está disponible, y el **ConnectivityChanged** evento, que se genera cuando la conexión cambia. La clase también expone la **IsRemoteReachable** método que puede utilizar para comprobar si un sitio remoto es accesible, y la **anchos de banda** propiedad, que devuelve una colección de anchos de banda disponibles (no se admite en IOS). Por convención, el nombre de cada clase Singleton expuestos por los plugins comienza con **Cruzar**.

Como se puede ver en el fragmento anterior, que tiene una abstracción multiplataforma que se utiliza en el PCL que no requiere implementaciones complejas, específicas de la plataforma llamada API nativas manualmente. Los complementos pueden ahorrar una enorme cantidad de tiempo, pero, por supuesto, pueden proporcionar una interfaz multiplataforma sólo para aquellas características que están comúnmente disponibles. Por ejemplo, la conectividad Plugin expone características de red que son comunes a iOS, Android y Windows, pero no características nativas que no pueden ser expuestas con una abstracción de plataforma cruzada y en su lugar requieren trabajar con las API nativas directamente. Sin embargo, recomiendo encarecidamente que compruebe si existe un plugin cuando se necesita para acceder a las funciones nativas no incluidos en Xamarin.Forms fuera de la caja; de hecho, en la mayoría de los casos, tendrá características comunes,



Consejo: Otro ejemplo de plugins se proporciona en el capítulo siguiente, cuando se habla del ciclo de vida de la aplicación.

Trabajar con vistas nativos

En las secciones anteriores, se examinó la forma de interactuar con las características nativas de Android, iOS y Windows mediante el acceso a su API directamente en el código C # o a través de plugins. En esta sección, en lugar de ver cómo utilizar vistas nativos en Xamarin.Forms, que es extremadamente útil cuando se necesita para extender vistas proporcionadas por Xamarin.Forms o cuando se desea utilizar vistas nativas que Xamarin.Forms no se ajusta en común objetos fuera de la caja.

Incrustación de vistas nativas en XAML

Xamarin.Forms le permite añadir puntos de vista nativo directamente en el marcado XAML. Esta característica es una adición reciente, y que hace que sea muy fácil de usar elementos visuales nativos. Para entender cómo se trabaja con vistas nativas en XAML funciona, considere Listado de Código 31.

Listado de Código 31

```
<? Xml version = "1.0" encoding = "UTF-8"?> < Pagina de contenido xmlns =
"http://xamarin.com/schemas/2014/forms"
    xmlns : x = "Http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns : ios = "Clr-espacio de nombres: UIKit;
        montaje = Xamarin.iOS; TargetPlatform = iOS"
    xmlns : androidWidget = "Clr-espacio de nombres: Android.Widget;
        montaje = Mono.Android; TargetPlatform = Android"
    xmlns : formsandroid = "clr-espacio de nombres: Xamarin.Forms;
        montaje = Xamarin.Forms.Platform.Android; TargetPlatform =
        Android"
    xmlns : ganar = "clr-espacio de nombres: Windows.UI.Xaml.Controls;
        montaje = Windows, versión = 255.255.255.255, Culture = neutral,
        PublicKeyToken = null, ContentType = WindowsRuntime; TargetPlatform =
        Windows"
    x : Clase = "App1.MainPage" Título = "puntos de vista nativo"> < ContentPage.Content > < StackLayout
> < ios : UILabel Texto = "Texto nativo" View.HorizontalOptions = "Inicio" /

>
< androidWidget : Vista de texto Texto = "Texto nativo"
    x : argumentos = "{ x : Estático formsandroid : formas .Context}"/>
< ganar : Bloque de texto Texto = "Texto nativo" /> </ StackLayout
>
</ ContentPage.Content >
</ Pagina de contenido >
```

En el XAML de la página raíz, primero tiene que añadir espacios de nombres XML que apuntan a los espacios de nombres de plataformas nativas. Los **formsandroid** espacio de nombres es requerido por widgets de Android para obtener el contexto de interfaz de usuario actual. Recuerde que usted puede elegir un nombre diferente para el identificador de nombre. El uso de puntos de vista nativos es entonces muy simple, ya que sólo tiene que declarar el punto de vista específico para cada plataforma en la que desea orientar. En el ejemplo anterior, el marcado XAML incluye una **UILabel** etiqueta nativa en iOS, una **Vista de texto** etiqueta nativa en Android, y una **Bloque de texto**

Ver nativa en Windows. Con vistas Android, debe proporcionar el contexto actual Xamarin.Forms interfaz de usuario, que se realiza con una sintaxis especial que une la estática (**x:estático**) **Forms.Context** propiedad a la vista. Puede interactuar con vistas en código C # como lo haría normalmente, tales como los controladores de eventos, pero la buena noticia es que también se puede asignar propiedades nativas de cada vista directamente en su XAML.

Trabajar con procesadores personalizados

Sencillamente, extracción de grasas son clases que Xamarin.Forms utiliza para acceder y hacer vistas nativos y que unen puntos de vista y diseños discutidos en los capítulos 4 y 5 con sus homólogos nativos de las Xamarin.Forms. Por ejemplo, el **Etiqueta** Ver discutido en el capítulo 4 mapas a una

LabelRenderer Xamarin.Forms clase que utiliza para representar el nativo **UILabel**, **Vista de Texto**, y

Bloque de texto vistas en iOS, Android y Windows, respectivamente. puntos de vista de Xamarin.Forms dependen completamente de extracción de grasas para exponer su aspecto y comportamiento. La buena noticia es que se puede anular los procesadores por defecto con la llamada **procesadores personalizados**, que se puede utilizar para extender o reemplazar características de las vistas Xamarin.Forms. Por consiguiente, un intérprete personalizado es una clase que hereda de la renderizador que mapea la vista nativo y es el lugar donde se puede cambiar el diseño, anular miembros, y cambiar el comportamiento de la vista. Un ejemplo será útil para comprender procesadores personalizados más. Suponga que desea una **Entrada** ver a autoselect su contenido cuando el usuario toca el cuadro de texto. Xamarin.Forms no tiene soporte para este escenario, para que pueda crear un procesador personalizado que funciona en el nivel de la plataforma. En el proyecto PCL, añadir una nueva clase llamada **AutoSelectEntry** que se parece a lo siguiente:

usando Xamarin.Forms; App1

espacio de nombres {

AutoSelectEntry clase pública: Entrada {}

La razón de la creación de una clase que hereda de **Entrada** es que, de lo contrario, el intérprete personalizado que va a crear en breve se aplica a toda la **Entrada** vistas en la interfaz de usuario. Mediante la creación de una vista derivada, puede decidir aplicar la intérprete personalizado sólo para éste. Si por el contrario desea aplicar el intérprete personalizado a todos los puntos de vista en la interfaz de usuario de ese tipo, puede omitir este paso. El siguiente paso es crear una clase que hereda de la renderizador incorporado (el

EntryRenderer en este caso), y proporciona una implementación dentro de cada proyecto de plataforma.



Nota: En los siguientes ejemplos de código, se pueden encontrar muchos objetos nativos y miembros. Sólo voy a destacar los que son estrictamente necesarios para su comprensión. Las descripciones de todos los otros se pueden encontrar en el Xamarin.iOS, Xamarin.Android, y la documentación de Windows Plataforma Universal.

Listado de Código 32 muestra cómo implementar un intérprete personalizado en IOS, Listado de Código 33 muestra la versión de Android, y Listado de Código 34 muestra la versión de Windows.

Listado de Código 32

```
[ Asamblea : ExportRenderer ( tipo de ( AutoSelectEntry ),
    tipo de ( AutoSelectEntryRenderer ))]
espacio de nombres App1.iOS {

    público clase AutoSelectEntryRenderer : EntryRenderer
    {
```

```

    protegido override void OnElementChanged ( ElementChangedEventArgs < Entrada > E) {

        base .OnElementChanged (e);
        var nativeTextField = Control; nativeTextField.EditingDidBegin += ( objeto remitente, EventArgs el
OS) =>
{
    nativeTextField.PerformSelector ( nuevo ObjCRuntime
        . Selector ( "seleccionar todo" ),
        nulo , 0.0f);
}; }}}
```

Listado de Código 33

```

utilizando Xamarin.Forms;
utilizando Xamarin.Forms.Platform.Android;
utilizando NativeAccess;
utilizando NativeAccess.Droid;

[ Asamblea : ExportRenderer ( tipo de ( AutoSelectEntry ),
    tipo de ( AutoSelectEntryRenderer )))
espacio de nombres App1.Droid {

    público clase AutoSelectEntryRenderer : EntryRenderer
    {
        protegido override void OnElementChanged ( ElementChangedEventArgs < Entrada > E) {

            base .OnElementChanged (e);
            Si (E.OldElement == nulo ) {

                var nativeEditText = ( global :: Android.Widget. Editar texto ) Contr
ol;
                nativeEditText.SetSelectAllOnFocus ( cierto );
            }}}}
```

Listado de Código 34

```
utilizando app1;
utilizando App1.UWP;
utilizando Xamarin.Forms;
utilizando Xamarin.Forms.Platform.UWP;

[ Asamblea : ExportRenderer ( tipo de ( AutoSelectEntry ),
    tipo de ( AutoSelectEntryRenderer ))]
espacio de nombres App1.UWP {

    público clase AutoSelectEntryRenderer : EntryRenderer
    {
        protegido override void OnElementChanged ( ElementChangedEventArgs < Entrada > E ) {

            base .OnElementChanged ( e );
            Si ( E.OldElement == nulo ) {

                var nativeEditText = Control;
                nativeEditText.SelectAll (); }}}
}
```

En cada implementación de la plataforma, se anula la **OnElementChanged** método para obtener la instancia de la vista a través de la nativa **Control** propiedad, y luego se invoca el código necesario para seleccionar todo el contenido del cuadro de texto utilizando las API nativas. Es necesario mencionar el

ExportRenderer atribuir a nivel de asamblea que dice Xamarin.Forms para hacer puntos de vista del tipo especificado (**AutoSelectEntry** en este caso) con un objeto de tipo

AutoSelectEntryRenderer, en lugar de la incorporada en el **EntryRenderer**. Una vez que tenga el intérprete personalizado listo, puede utilizar la vista personalizada en XAML como se haría normalmente, como se demuestra en Listado de Código 35.

Listado de Código 35

```
<? Xml version = "1.0" encoding = "UTF-8"?> < Pagina de contenido xmlns =
"http://xamarin.com/schemas/2014/forms"
    xmlns : x = "Http://schemas.microsoft.com/wifx/2009/xaml"
    xmlns : local = "CLR-espacio de nombres: App1"
    Título = "Página Principal"
    x : Clase = "App1.MainPage">

    < StackLayout Orientación = "Vertical" Relleno = "20">
        < Etiqueta Texto = "Introduzca un texto:" />

        < local : AutoSelectEntry x : Nombre = "MyEntry" Texto = "Introducir texto ..."
            HorizontalOptions = "FillAndExpand" />
```

```
</ StackLayout >
```

```
</ Pagina de contenido >
```



Consejo: La local espacio de nombres XML es declarado por defecto, por lo que añadir a su punto de vista es aún más simple. Además, IntelliSense mostrará la vista personalizada en la lista de objetos disponibles desde ese espacio de nombres.

Si ahora se ejecuta este código, se verá que el texto en el **AutoSelectEntry** vista se seleccionará automáticamente cuando el cuadro de texto se toca. procesadores personalizados están muy potente ya que le permite anular completamente el aspecto y el comportamiento de cualquier punto de vista. Sin embargo, a veces sólo tiene algunas personalizaciones menores que en cambio pueden ser proporcionados a través de efectos.

Consejos para los efectos

Los efectos son una adición reciente a la caja de herramientas Xamarin.Forms y pueden ser considerados como procesadores personalizados simplificados, limitado a cambiar algunas propiedades de diseño sin cambiar el comportamiento de una vista. Un efecto es de dos clases: una clase que hereda de **PlatformEffect** y debe ser implementado en todos los proyectos de plataforma; y una clase que hereda de **RoutingEffect** y reside en el proyecto, cuya responsabilidad es la solución de la aplicación específica de la plataforma del efecto personalizado PCL (o compartido). Manejar el **OnAttached** y **OnDetached** eventos para proporcionar la lógica para su efecto. Debido a que su estructura es similar a las estructuras de los recicladores personalizados, que no cubrirá efectos en más detalle aquí, pero es importante que saben que existen. Se puede extraer el funcionario [documentación](#), Que explica cómo consumir efectos incorporados y cómo crear otros personalizados.

Resumen del capítulo

Las aplicaciones móviles a menudo tienen que trabajar con las características que sólo se puede acceder a través de las API nativas. Xamarin.Forms proporciona acceso a todo el conjunto de APIs nativas en iOS, Android y Windows a través de una serie de posibles opciones. Con el **Dispositivo** clase, se puede obtener información sobre el sistema actual de su código compartido. Con el **DependencyService** clase y su **Obtener** método, se puede resolver abstracciones multiplataforma de código específico de la plataforma en su PCL.

Con plugins, tiene abstracciones listas para el uso multiplataforma para los escenarios más comunes, tales como (pero no limitados a) acceder a la cámara, la información de red, configuración o estado de la batería. En cuanto a los elementos visuales nativas, puede incrustar iOS, Android y Windows nativos vistas directamente en su XAML. También puede escribir extracción de grasas o efectos personalizados para cambiar el aspecto y el tacto de sus puntos de vista. En realidad, cada plataforma también gestiona el ciclo de vida de aplicaciones con sus propias APIs. Afortunadamente, Xamarin.Forms tiene una abstracción multiplataforma que hace que sea más simple, como se explica en el siguiente capítulo.

Capítulo 9 Administración del ciclo de vida de aplicaciones

El ciclo de vida de la aplicación consiste en eventos como el inicio, suspensión y reanudación. Cada plataforma gestiona el ciclo de vida de aplicaciones de manera diferente, por lo que la implementación de código específico de la plataforma de iOS, Android y proyectos de Windows requeriría un poco de esfuerzo. Por suerte, Xamarin.Forms le permite gestionar el ciclo de vida de aplicaciones de una manera unificada y se encarga de realizar el trabajo específico de la plataforma en su nombre. En este capítulo se ofrece una breve explicación del ciclo de vida de la aplicación y de cómo se pueden manejar fácilmente el comportamiento de su aplicación.

Presentación de la Aplicación clase

los **Aplicación clase** es una clase que hereda de **Singleton Solicitud** y se define en el archivo `App.xaml.cs`. Puede ser pensado como un objeto que representa su aplicación en ejecución e incluye la infraestructura necesaria para manejar los recursos, la navegación, y el ciclo de vida de la aplicación. Si necesita almacenar algunos datos en las variables que deben estar disponibles para todas las páginas de la aplicación, se puede exponer a los campos estáticos y propiedades de la **Aplicación clase**. En un nivel superior, la **Aplicación clase** expone algunos miembros fundamentales que pueda necesitar a través de todo el ciclo de vida de aplicaciones: la **Página principal** la propiedad se asigna a la página raíz de su aplicación, y el

OnStart, **OnSleep**, y **En resumen** métodos que utiliza para gestionar el ciclo de vida de la aplicación que se describen en la siguiente sección.

La gestión del ciclo de vida de aplicaciones

El ciclo de vida de la aplicación se puede resumir en cuatro eventos: inicio, suspensión, reanudación y apagado. Las plataformas Android, iOS y Windows administran estos eventos de manera diferente, pero Xamarin.Forms proporciona un sistema unificado que permite la gestión de inicio, suspensión de una aplicación, y reanudar desde una única base de código C # compartida. Estos eventos están representados por la **OnStart**, **OnSleep**, y **En resumen** métodos que se pueden ver en el archivo `App.xaml.cs` cuyo cuerpo está vacía. Actualmente, ningún método específico se encarga de la aplicación de apagado, ya que en la mayoría de los casos de suspensión de manipulación es suficiente. Por ejemplo, es posible cargar algunos configuración de la aplicación dentro de

OnStart en el inicio, guardar la configuración cuando la aplicación está suspendida dentro de **OnSleep**, y volver a cargar los ajustes cuando la aplicación vuelve al primer plano dentro de **En resumen**. Para una mejor comprensión de este ejemplo, se puede instalar el [ajustes](#) plugin desde NuGet a todos los proyectos de la solución.



Consejo: Información completa sobre el plugin configuración está disponible en su [GitHub](#) página.

En el **Ayudantes | Settings.cs** archivo, reemplace el código autogenerado con el contenido del Listado de Código 36, que implementa un entorno de tipo **Fecha y hora** que se pueden utilizar para obtener y establecer la fecha y la hora para los eventos del ciclo de vida de aplicaciones.

Listado de Código 36

```
utilizando Plugin.Settings;
utilizando Plugin.Settings.Abstractions;
utilizando Sistema;

espacio de nombres App1.Helpers {

    público clase estática ajustes
    {
        estática privada ISettings Ajustes de Aplicacion {

            obtener
            {
                regreso CrossSettings .Corriente;
            }

            const string privada AccessDateSettings = "fecha de acceso" ;
            privado estático solo lectura Fecha y hora AccessDateDefault = Fecha y hora .Ahora;

            público estático Fecha y hora Fecha de acceso {

                obtener
                {
                    regreso AppSettings.GetValueOrDefault (AccessDateSettings,
                        AccessDateDefault);
                }
                conjunto
                {
                    AppSettings.AddOrUpdateValue (AccessDateSettings, valor );
                }
            }
        }
    }
}

utilizando Plugin.Settings;
utilizando Plugin.Settings.Abstractions;
utilizando Sistema;

espacio de nombres App1.Helpers {

    público clase estática ajustes
    {
        estática privada ISettings Ajustes de Aplicacion {

            obtener
            {
                regreso CrossSettings .Corriente;
            }
        }
    }
}
```

```

    }

const string privada AccessDateSettings = "fecha de acceso";
privado estático solo lectura Fecha y hora AccessDateDefault = Fecha y hora .Ahora;

público estático Fecha y hora Fecha de acceso {

    obtener
    {
        regreso AppSettings.GetValueOrDefault (AccessDateSettings,
AccessDateDefault);
    }
    conjunto
    {
        AppSettings.AddOrUpdateValue (AccessDateSettings, valor );
    }}}
```

Ahora, como se puede ver en el Listado de Código 37, se puede obtener y establecer el valor del ajuste de acuerdo al evento aplicación del ciclo de vida. En este caso, el almacenamiento de la fecha de acceso tiene sentido cuando la aplicación se inicia o reanuda, no cuando está suspendida (en cuyo caso es posible que desee agregar una configuración específica).

Listado de Código 37

```

utilizando App1.Helpers;
utilizando Sistema;

utilizando Xamarin.Forms;

espacio de nombres app1 {

público clase parcial Aplicación : Solicitud
{
    público App () {

        InitializeComponent ();

        MainPage = nuevo App1. Pagina principal (); }

    protegido override void OnStart () {

        // manipulador cuando su aplicación se inicia.
        ajustes .AccessDate = Fecha y hora .Ahora;
    }}
```

```

protegido override void En resumen() {

    // manipulador cuando su aplicación se reanuda.
    ajustes .AccessDate = Fecha y hora .Ahora;
}

protegido override void OnSleep () {

    // manipulador cuando su aplicación duerme.

    // Añadir un nuevo ajuste para almacenar la fecha / hora de OnSleep.
}
}

```

Con la ayuda de los puntos de interrupción y el depurador, usted será capaz de demostrar que la aplicación entra en los métodos apropiados de acuerdo con el evento del ciclo de vida.

Enviar y recibir mensajes

Xamarin.Forms incluye una clase estática interesante llamado **MessagingCenter**. Esta clase puede enviar mensajes de difusión que los suscriptores pueden recibir y tomar acciones, basado en un modelo de editor / suscriptor. En su forma más básica, se utiliza el **MessagingCenter** para enviar un mensaje de la siguiente manera:

MessagingCenter.Send <MainPage> (esto, "MENSAJE");

los **Enviar** los parámetros de tipo de método especifican los tipos de suscriptores deben esperar, y sus argumentos son del remitente (**Página principal** en este caso, como un ejemplo) y el mensaje en forma de una cadena. Se pueden especificar varios parámetros de tipo y, por tanto, múltiples argumentos antes del mensaje.



Consejo: El compilador es capaz de inferir los parámetros de tipo de Enviar, por lo que no es obligatorio especificar explícitamente.

Los suscriptores pueden escuchar los mensajes y tomar las medidas de la siguiente manera:

MessagingCenter.Subscribe <MainPage>
(Esto, "MENSAJE", (emisor) => {

// Hacer algo aquí);

Cuando **MessagingCenter.Send** se invoca algún lugar, los objetos de escuchar un mensaje en particular va a ejecutar la acción especificada dentro de **Subscribe** (esto no tiene por qué ser necesariamente una expresión lambda; que puede ser un delegado expandido). Cuando haya terminado su trabajo, los suscriptores pueden invocar **MessagingCenter.Unsubscribe** dejar de escuchar un mensaje, pasando el remitente como el parámetro de tipo, el objeto actual, y el mensaje, como sigue:

```
MessagingCenter.Unsubscribe <MainPage> (esto, "MENSAJE");
```

los **MessagingCenter** clase puede ser muy útil cuando se tiene lógicas que están desconectados de la interfaz de usuario, e incluso puede ser útil con implementaciones MVVM.

Resumen del capítulo

La gestión del ciclo de vida de aplicaciones puede ser muy importante, especialmente cuando se necesita para obtener y almacenar datos en el inicio de la aplicación o suspensión. Xamarin.Forms evita la necesidad de escribir código específico de la plataforma y ofrece una solución multi-plataforma a través de la **OnStart**, **OnSleep**, y

En resumen métodos que permiten el manejo de la puesta en marcha, suspensión y reanudar eventos, respectivamente, a partir de una única base de código C #, independientemente de la plataforma de la aplicación se está ejecutando. Esto no sólo es una característica de gran alcance, pero realmente simplifica su trabajo como desarrollador. Por último, se ha visto en este capítulo, el **MessagingCenter** clase, un objeto estático que permite para el envío / la suscripción mensajes y es útil con lógicas desacoplados de la interfaz de usuario.

Apéndice: Recursos útiles

Xamarin.Forms es una plataforma de desarrollo con todas las funciones, por lo que este libro no podía cubrir todos los escenarios posibles. En este apéndice, encontrará una lista de recursos que se puede tener en cuenta para el estudio adicional.

Trabajar con bases de datos SQLite

SQLite es una, de código abierto, el motor de base de datos local sin servidor que puede utilizar en sus Xamarin.Forms aplicaciones móviles para almacenar datos estructurados. SQLite está incluido en IOS y Android, y se puede instalar fácilmente en Windows 10 dispositivos. Debido a la necesidad de almacenar datos locales es muy común, el [documentación](#) proporciona información detallada sobre cómo utilizar SQLite en sus aplicaciones móviles. Además, el autor de este libro electrónico ha publicado una introducción fácil de SQLite con Xamarin.Forms en MSDN Magazine en un artículo titulado “[Trabajar con bases de datos locales en Xamarin.Forms utilizando SQLite](#)”.

El consumo de servicios web y servicios en la nube

Otro requisito común para aplicaciones móviles está consumiendo recursos en Internet o, más en general, a través de una red. Esto incluye notificaciones push, servicios web, servicios de WCF y servicios REST, tanto en los establecimientos y en la nube. En términos generales, se consumen recursos en una red a través de la **HttpClient** clase y sus métodos asíncronos. Sin embargo, Microsoft también ofrece bibliotecas para almacenar datos a Azure y para la aplicación en línea de sincronización de datos. Todos estos escenarios, con ejemplos, se describen en una página en la documentación de llamada [Datos y Servicios en nube](#), Que también proporciona la documentación sobre la adición de la inteligencia artificial a través de los servicios de Microsoft cognitivas para sus aplicaciones móviles.

La publicación de aplicaciones

En la mayoría de los casos, tendrá que publicar sus aplicaciones móviles nativas a Google Play, la tienda de aplicaciones de Apple, y la tienda de Windows. En realidad, el proceso de publicación no está relacionada con Xamarin.Forms, sino que implica los proyectos de plataforma. La [documentación oficial Xamarin](#) proporciona orientación para la publicación [Androide](#) y [iOS](#) aplicaciones, mientras que se puede hacer referencia a la plataforma Windows universal [documentación](#) para la publicación de aplicaciones para Windows 10.

Los ejemplos de código y kits de iniciación

A medida que la plataforma se está volviendo más y más popular, es más fácil encontrar código de ejemplo en Internet. Sin embargo, un buen punto de partida es la [repositorio oficial](#) en GitHub, que contiene una serie de aplicaciones de ejemplo que se dirigen a varios escenarios de desarrollo. El autor de este libro electrónico también ha publicado un código abierto [kit de inicio](#), Que muestra cómo obtener los datos de

Internet, almacenar datos en una base de datos local SQLite, poner en práctica la unión y la navegación de datos, y mucho más.

Actualización de herramientas Xamarin

En Visual Studio 2017, la actualización de herramientas Xamarin se logra a través de una extensión llamada [Xamarin Updater](#). Esta extensión añade un nuevo nodo en las extensiones y herramientas de actualizaciones que ya conoce y le permite instalar actualizaciones de forma rápida tanto para las herramientas integradas Xamarin y los SDK. La Figura 48 muestra lo que parece.

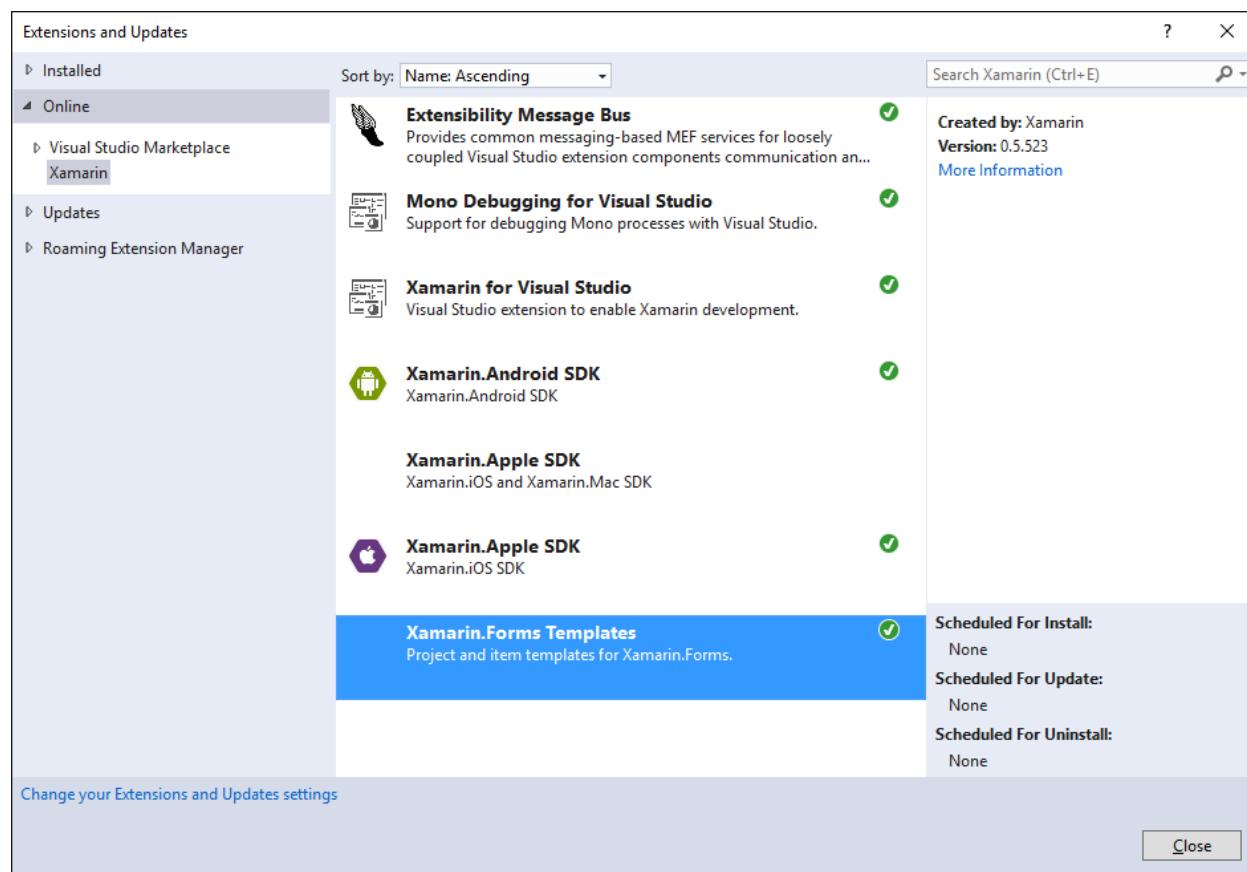


Figura 48: Actualización de herramientas y SDK Xamarin

La extensión muestra las herramientas instaladas y, si hace clic en el **Updates \ Xamarin** nodo, busca versiones estables en Visual Studio 2017 y para versiones preliminares de Visual Studio 2017 Vista previa.