Uno Platform tutorial

# Introduction

Uno Platform is a framework that allows us develop applications for multiple devices and platforms with the minimal effort.

Currently, it can deploy to the following systems:

* Android
* iOS
* Windows
* Web Assembly

## Fundamentals

The goal of Uno Platform is to maximize our productivity and the number of systems our applications can be delivered. To do it, Uno takes the Universal Windows Platform (UWP) as the reference to generate working code for the other systems.

# First steps

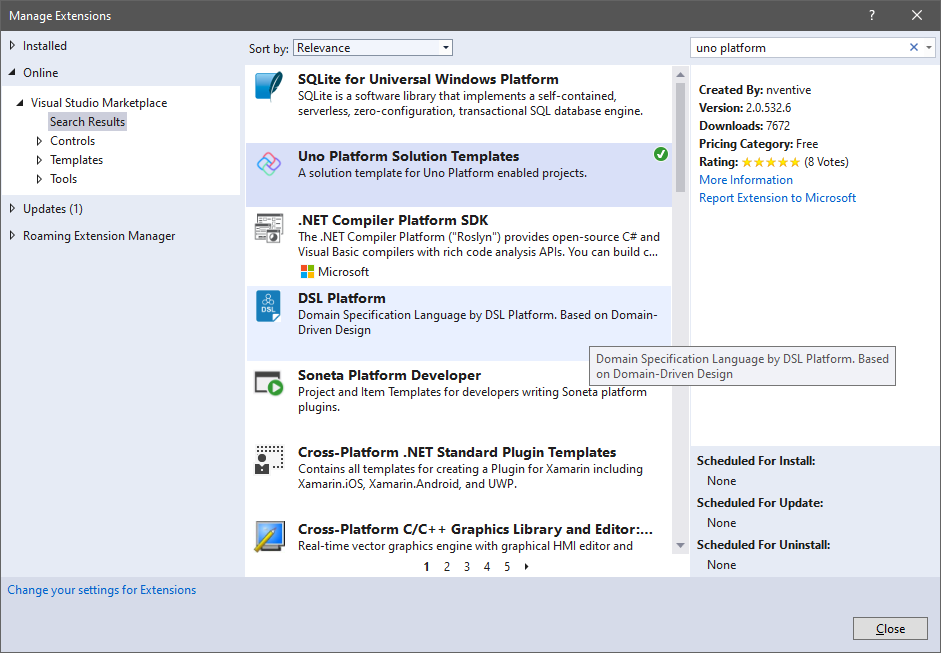
Developing with Uno Platform is very easy. However, you should have some previous knowledge.

* You should have developed applications in at least one of the following technologies:
  + WPF
  + UWP
  + Xamarin Forms
* You should be familiar with the MVVM pattern.
* Knowing the basics of Reactive UI is recommendable.

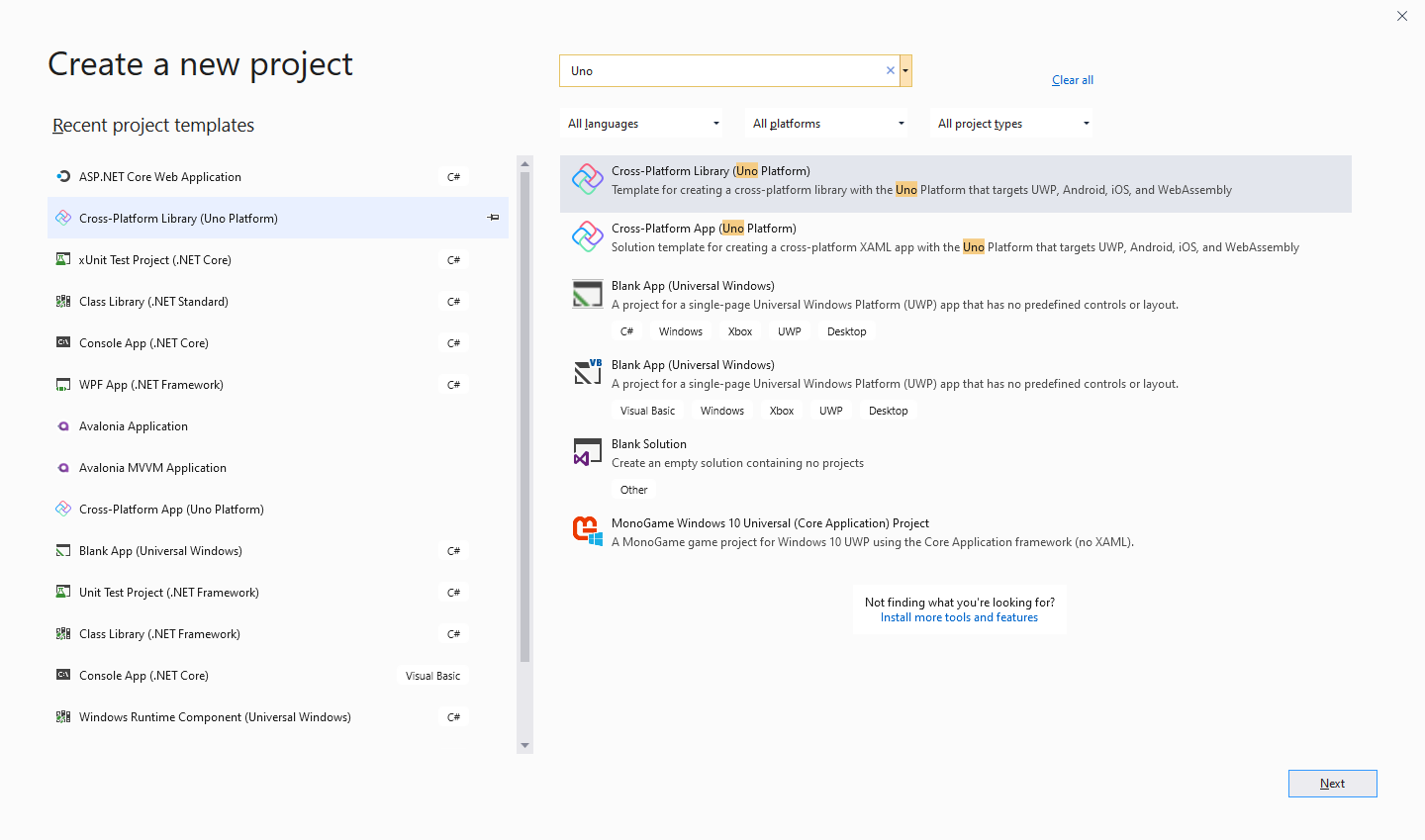
## First contact with Uno Platform

In order to start developing, the first thing we must do is to install the **Uno Platform Solution Templates**. These templates will help us start coding quickly, because they create both the solution structure and the bootstrap code for us. Let’s do it.

Go to the Extensions menu in Visual Studio and click on “Manage Extensions”.



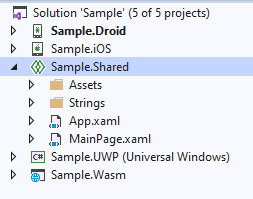
Once they’re installed, we will see them in the Visual Studio startup page or going to “Create new project”.



Since we’re creating a new application (not a library), we will use the **Cross-Platform App (Uno Platform)** template.

After choosing the template and giving a name to our project, we will be presented with a “hello world” application ready to run that consists of 5 projects. Let’s examine its structure a bit.

## Uno Platform solution structure



This is the how a standard Uno Platform solution looks like.

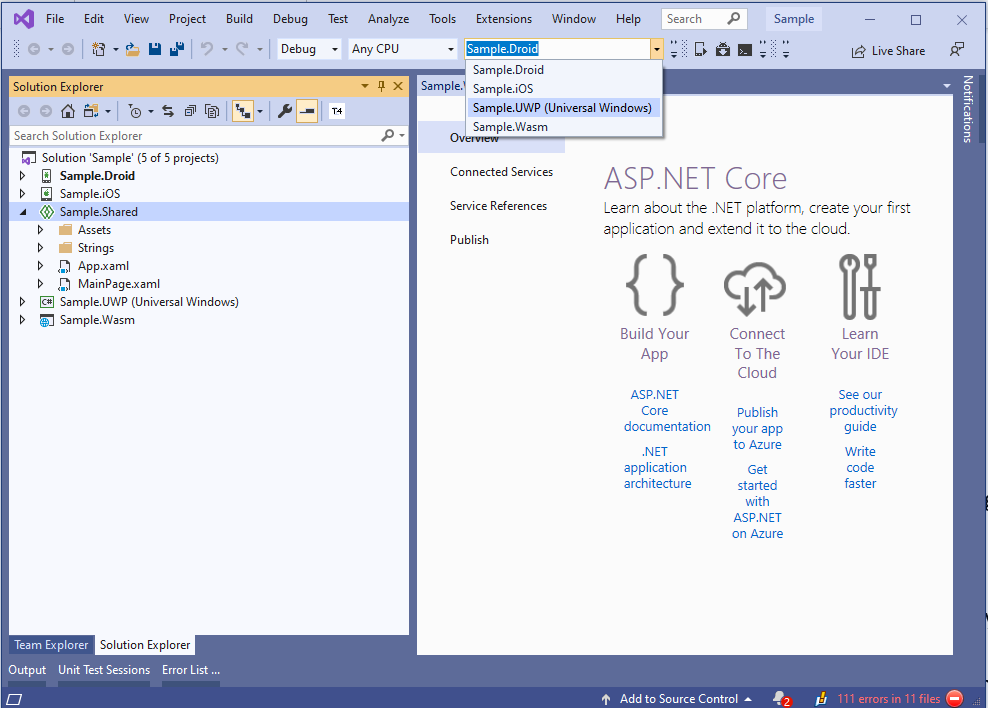
In short there are 2 kinds of project:

* Heads: “Head projects” are bootstrap projects for each platform target. We can see one for of the supported systems (iOS, Android, Windows and Web Assembly).
* Shared Project: it’s Shared with all the heads and contains all the shared code. This project will also contain all our views (XAML)

## First run

The start, we’ll run the Windows head to check that everything was created correctly.

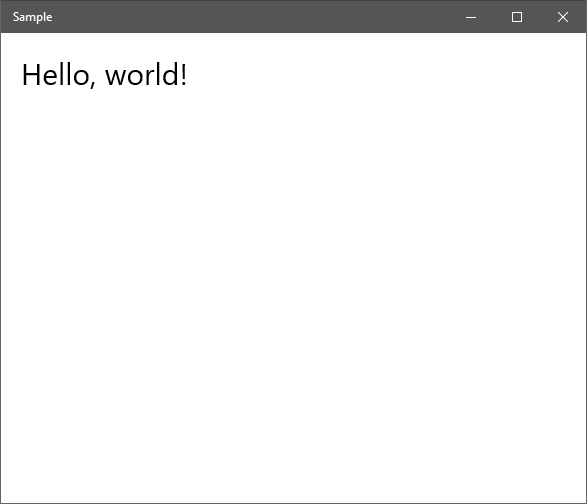
Go to the startup project dropdown



Choose the UWP project (Universal Windows)



Once it’s selected, run it pressing F5 or clicking the “Play” button (), as usual.



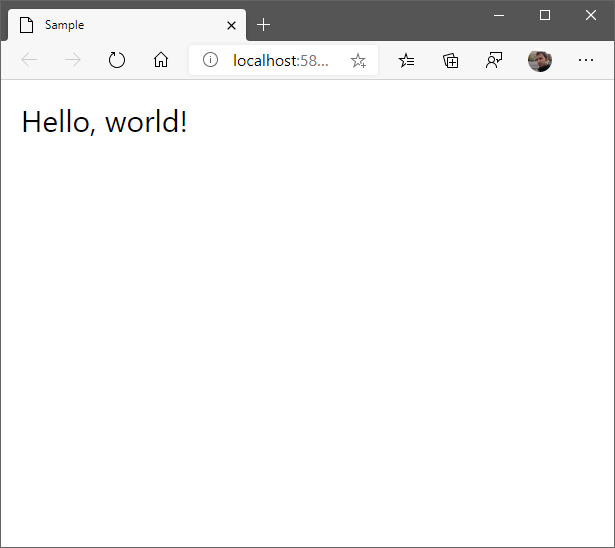
After the build is complete (it can take a bit the first time) a window should pop up in front of us saying “Hello, world!”. Good 😊

Now that we’ve checked that the Windows head is working, we can try to do the same with the Web Assembly head. This is much more exciting because it will make our application run inside the web browser.

Let’s go to the Startup project dropdown again and choose the Web Assembly head (WASM, for short).



Click the Play button or F5 and a the browser will launch a new tab with our application running!

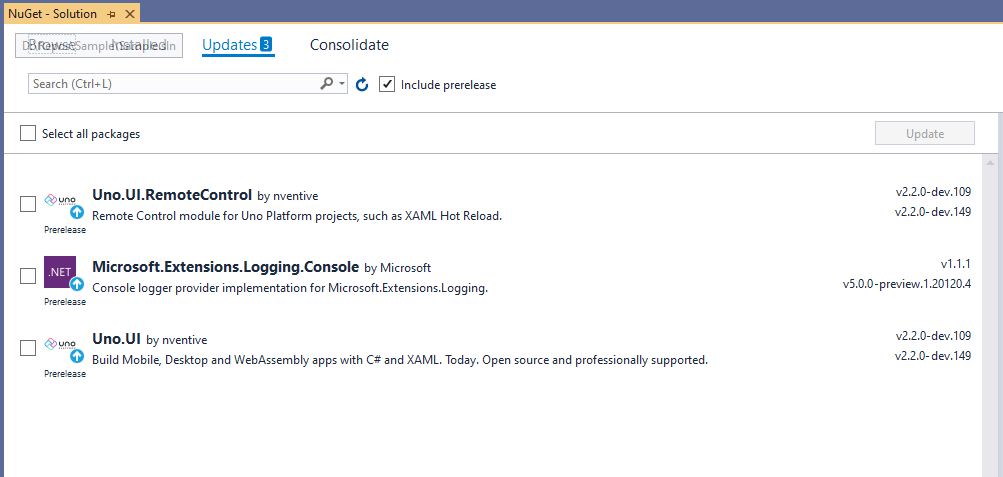


### Actualización de paquetes NuGet

Una práctica común muy recomendable después de crear un proyecto mediante una plantilla es actualizar los paquetes NuGet que la misma pudiera utilizar. Esto es porque las plantillas se actualizan con mucha menos frecuencia y las versiones de los paquetes que utilizan suelen ser antiguas.

Con Uno Platform esto es especialmente cierto, porque el desarrollo es muy rápido y se introducen mejorar y arreglos constantemente. Tanto es así, que los paquetes más recomendables son aquellos que están aún etiquetados como “prerelease”.

Para actualizar iremos a Administrar los paquetes NuGet de la solución, marcaremos la casilla de “prerelease” y actualizamos todos los paquetes excepto Microsoft.Extensions.Logging.Console [[1]](#footnote-1)



## Guía de referencia

Para modelar la interfaz de usuario usaremos el lenguaje de marcado XAML, como es habitual en otros frameworks.

El dialecto de XAML que usaremos es el propio de la Plataforma Universal de Windows.

Los elementos básicos son muy sencillos de utilizar y son similares a los usados en otros dialectos de XAML, pero conviene saber que hay algunas diferencias notables, sobre todo con Xamarin Forms.

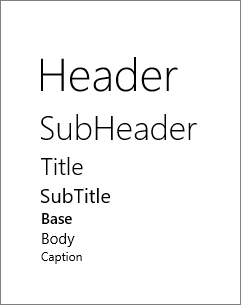
## Elementos Básicos

### Controles

#### TextBlock

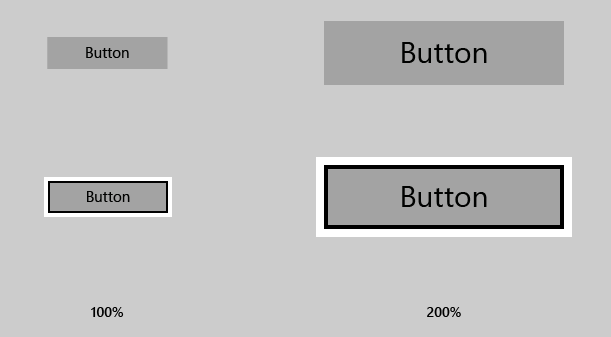
Muestra un texto. Se suele usar la propiedad Text con un Binding en modo OneWay (unidireccional) o TwoWay (bidireccional).

a una propiedad string del ViewModel



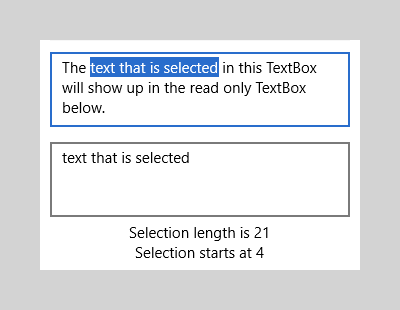
#### Button

Permite la ejecución de un comando mediante la propiedad Command. El contenido del botón puede ser cualquier cosa, incluidos otros controles. Normalmente, como Content se pone una cadena.



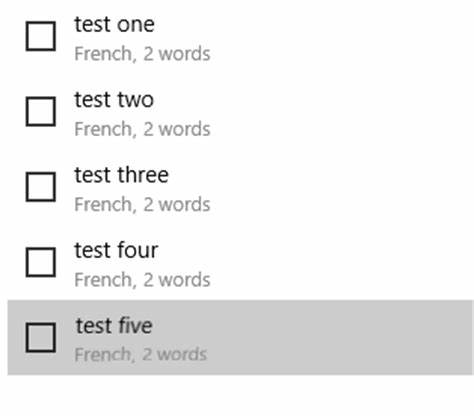
#### TextBox

Es el control de entrada de texto. Se suele usar la propiedad Text con un Binding en modo TwoWay (bidireccional)



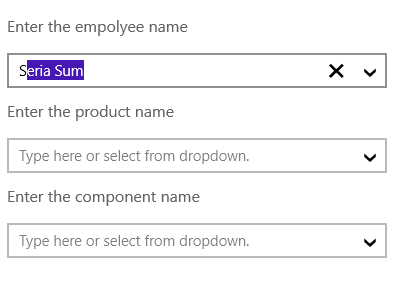
#### ListView

El control para representar listas. Siempre se suele usar con su propiedad ItemsSource (el enlace de dato desde donde se obtendrán los elementos) e ItemTemplate, que es la plantilla de cada elemento



#### ComboBox

Un combo desplegable



#### Border

Un contenedor que adorna el contenido con un borde. Suele usarse para mejorar el aspecto y delimitar controles



#### Image



Control para representar imágenes

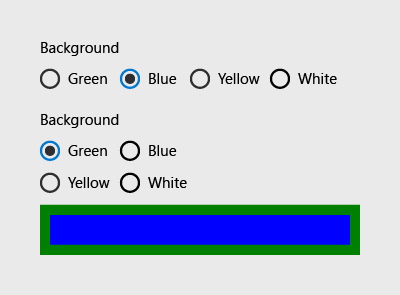
#### CheckBox

Cuadro con capacidad para marcar, desmarcar y estado intermedio



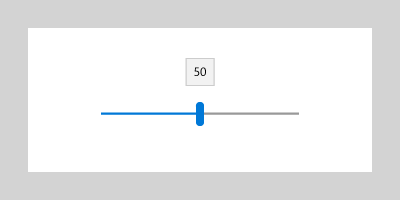
#### RadioButton

Botón de opción exclusiva.



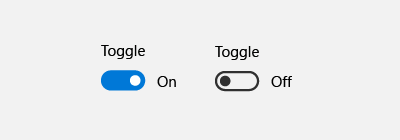
#### Slider

Deslizador. Su valor es numérico.



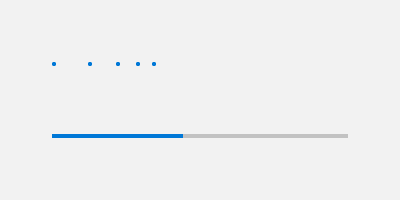
#### ToggleSwitch

Encendido y apagado



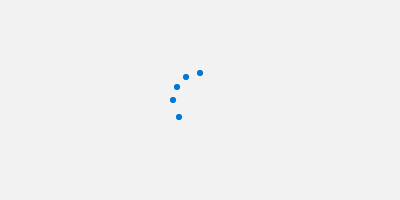
#### ProgressBar

Barra de progreso, puede ser indeterminada, lo cual indica que se desconoce el progreso. En ese caso, aparece una animación de puntos.



#### ProgressRing

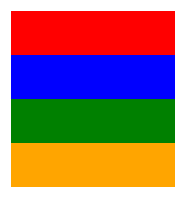
Similar a ProgressBar, pero circular. No tienen ningún valor para indicar el porcentaje.



### Diseño (disposición de elementos)

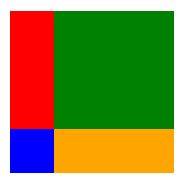
#### StackPanel

Es un panel que apila elementos vertical u horizontalmente.



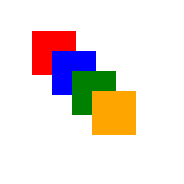
#### Grid

Rejilla que permite situar elementos en filas y columnas



#### Canvas

Panel que permite situar lo elementos con coordenadas (relativas al panel)



### Bindings

Los bindings son un elemento básico de todas las plataformas basadas en XAML, sirven para enlazar 2 propiedades. Una se llama “origen” y la otra “destino”. El enlace puede ser unidireccional o bidireccional.

Ejemplo

* Unidireccional <Button Content=”{Binding ButtonText}” />
* Bidireccional <TextBox Text=”{Binding Text, Mode=TwoWay}” />

También existen los bindings en tiempo de compilación, x:Bind, pero no los recomiendo por las fuertes restricciones que imponen en la mayoría de los casos.

### Data Templates

Las plantillas (Data Templates) también son un elemento clave y esencial de todas las plataformas XAML. Las más habituales son las de ListView (ItemTemplate), pero todos los controles ofrecen plantillas que nos permiten elegir cómo se presenta el contenido de estos.

### Styles

Los estilos son una manera de reutilizar conjuntos de valores de las propiedades de manera que puedan reutilizarse y mantenerse de una manera más sencilla. Los estilos también son un elemento común en los framework XAML. Tienen un TargetType que es el tipo de elemento para el que está destinado, y una lista de Setters, que aplican los valores a cada propiedad que elijamos. Ejemplo:

<Style x:Key=”MyButtonStyle” TargetType=”Button”>

<Setter Property=”Background” Value=”Red />

</Style TargetType=”Button”>

Los estilos se suelen definir en **diccionarios de recursos,** que son contendores donde los recursos tienen una clave. Dicha clave puede referenciarse luego mediante una sintaxis especial (la extensión de marcado “StaticResource”).

Ejemplo de asignación de estilo:

<Button Style=”{StaticResource MyButtonStyle}“ />

# App1. Creación de aplicación sencilla.

Vamos a crear una aplicación sencilla como entrenamiento. En esta aplicación pondremos a prueba nuestra experiencia previa e iremos adaptándola a la forma de desarrollar con Uno Platform.

Vamos a centrarnos en aspectos muy esenciales que nos permitan evolucionar y crear aplicaciones más complejas más tarde.

Para empezar:

Clonaremos el siguiente repositorio: <https://github.com/SuperJMN/UnoPlatformTutorial-App1>

En el historial vamos a poder ver cada paso para que entendamos fácilmente lo que se va haciendo.

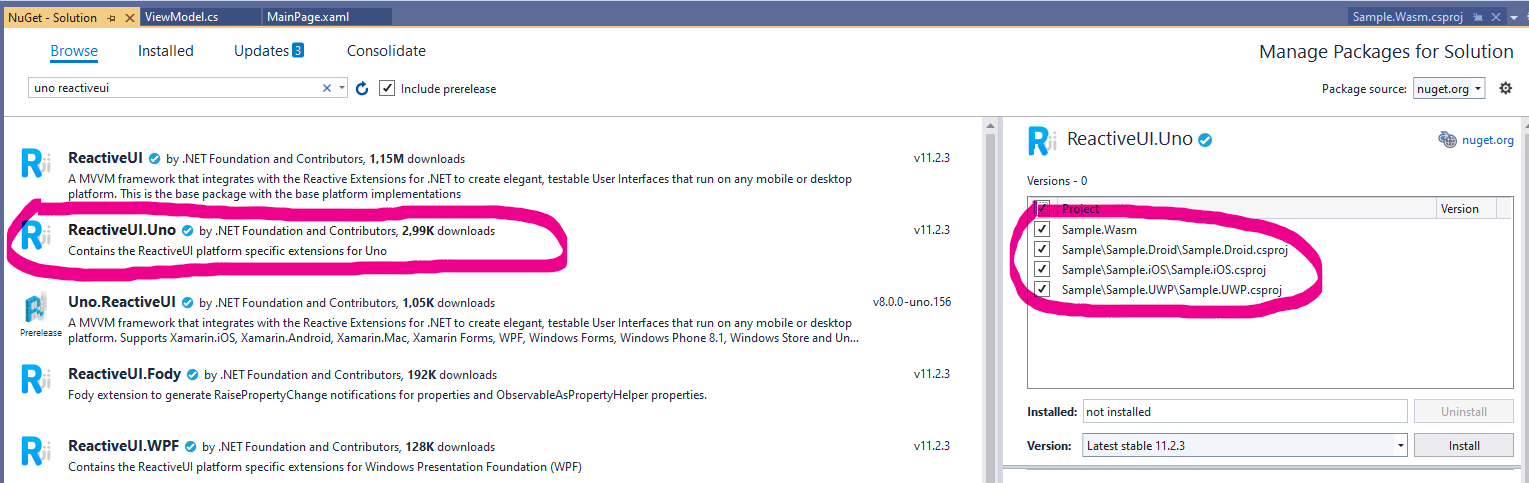
1. Agregaremos un ViewModel [[commit](https://github.com/SuperJMN/UnoPlatformTutorial-App1/commit/ff35be8c0e76865e64497970fcda3e038a9752b8)]

Lo crearemos en el proyecto Shared, para simplificar.

1. Nos apoyaremos en ReactiveUI [[commit](https://github.com/SuperJMN/UnoPlatformTutorial-App1/commit/3d26fe0292b3b559bf9632515c86de2244e15cb3)]

Es conveniente que nos acostumbremos a usar programación reactiva 😊 ReactiveUI es el framework recomendado para ello.

Para instalar ReactiveUI **no podemos instalar directamente la versión oficial**, sino una especialmente adaptada para Uno Platform. Esto una excepción: la mayoría de los paquetes pueden usarse directamente.



**IMPORTANTE:** Cuando instalemos o añadamos una referencia debemos hacerlo en todas las heads. Si no lo hacemos, el proyecto en cuestión no compilará.

1. Añadimos unas propiedades básicas al ViewModel y creamos un XAML sencillo [[commit]](https://github.com/SuperJMN/UnoPlatformTutorial-App1/commit/b75c63b655dcec96b116479204203441ecc77f82)
2. Retocamos el XAML para que luzca mejor [[commit](https://github.com/SuperJMN/UnoPlatformTutorial-App1/commit/695516e3b8b6da1e86b3b2b59721730286fe7e11)]
3. Creamos un UserControl (Control de Usuario) para encapsular el XAML de cada tarjeta de contacto [[commit](https://github.com/SuperJMN/UnoPlatformTutorial-App1/commit/b718e3b397521a7371dee416d1cb869675d0e192)]
4. Renombramos el ViewModel: Pasará a ser ContractViewModel [[commit](https://github.com/SuperJMN/UnoPlatformTutorial-App1/commit/bafa158ad55e09a4ea8ada82c4be3c80415e46e5)]
5. Añadimos un ListView con un listado fijo de tarjetas de contacto [[commit](https://github.com/SuperJMN/UnoPlatformTutorial-App1/commit/346a7f54274fa5dc763207b049d1d0cbecb20acf)]
6. Introducimos un ViewModel principal con una lista de contactos enlazada vía Binding [[commit](https://github.com/SuperJMN/UnoPlatformTutorial-App1/commit/346a7f54274fa5dc763207b049d1d0cbecb20acf)]
7. Añadimos la funcionalidad de agregar contactos [[commit](https://github.com/SuperJMN/UnoPlatformTutorial-App1/commit/4dc48f6bbd461a545d4edbefa206f55329e28bf1)]
8. Añadimos la funcionalidad de eliminar contactos [[commit](https://github.com/SuperJMN/UnoPlatformTutorial-App1/commit/4a671f25ce1ed9d5765a0eec35cc9cc61dcf6710)]

# App2. Aplicación más compleja.

La aplicación compleja que forma parte de este tutorial puede obtenerse clonando el siguiente repositorio de GitHub: <https://github.com/SuperJMN/UnoPlatformTutorial-App2>

Por ser más compleja, no la desarrollaremos paso a paso, sino que explicaremos todas sus partes para despejar todas las dudas sobre cómo está hecha.

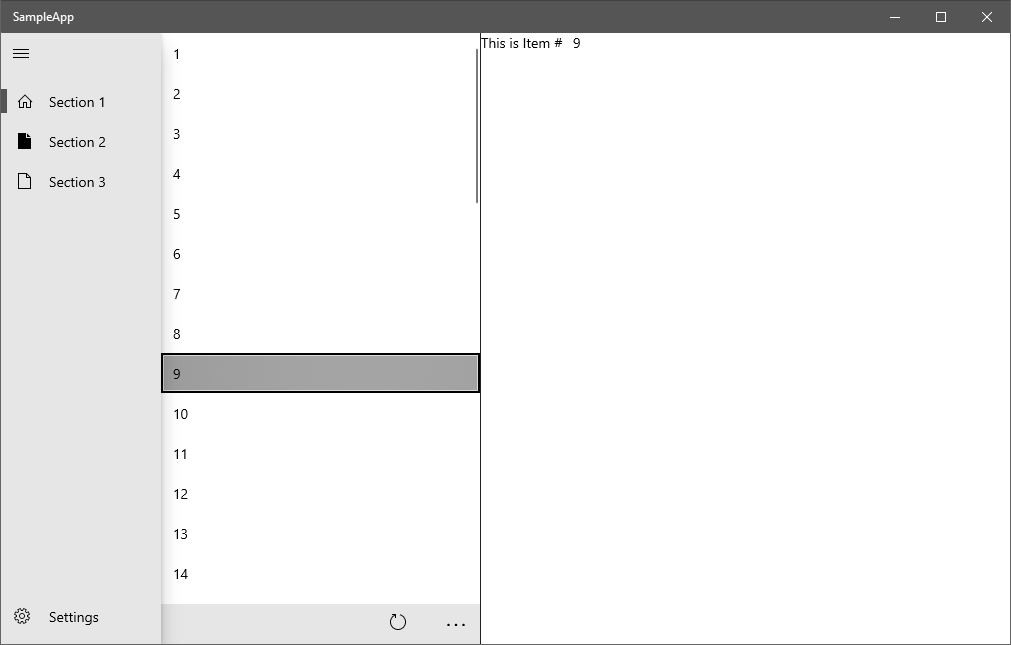
IMPORTANTE: esta aplicación no se ha creado desde 0 completamente, sino que hemos usado una plantilla de aplicación disponible en GitHub en este repositorio: <https://github.com/SuperJMN/UnoPlatform.AppTemplate>

Con esta plantilla obtendremos el esqueleto básico de la una aplicación genérica de acceso a datos podremos ir completando. De la plantilla podemos destacar:

* Inyección de dependencias preconfigurada con Grace, un contenedor ID IoC potente y sencillo de usar.
* Servicio distribuido funcional en proyecto de servicio usando EasyRPC, que provee una manera especialmente fácil de invocar métodos de nuestros servicios sin tener que crear controladores.
* ReactiveUI, para aplicar programación reactiva con MVVM, el método más recomendable para trabajar con XAML.
* Interfaz con menú lateral de navegación.
* Navegación basada en ViewModels.

## Análisis de la plantilla

La plantilla, sin tocar nada, nos ofrece la siguiente aplicación:



En la barra la lateral tenemos 3 secciones de ejemplo. Estas secciones que pueden personalizarse accediendo al proyecto SampleApp.Shared\Views.

Encontraremos las 3 vistas con sus respectivos UserControls

En la carpeta SampleApp.Shared\ViewModels están sus respectivos ViewModels. Los modificaremos para adaptarlos a nuestra lógica.

### CompositionRoot

La plantilla tiene un archivo llamado Composition.cs. Este archivo es la raíz de composición, un concepto común en las aplicaciones que usan inyección de dependencias. En él se configuran los servicios y los mapeos entre Vistas y sus ViewModels, entre otras cosas.

#### ConfigureServices

En este método se registran los servicios del contenedor Grace (nuestro contenedor de DI IoC). Este contenedor resolverá todos los servicios de nuestra aplicación, incluidos los remotos, usando un “Proxy Namespace”, esto es, un mapeo entre un namespace del CLR y una URL, de manera que, al resolver un servicio remoto, se entregará un proxy que permitirá hacer las llamadas como si fuera un servicio local. Esto lo aporta Grace + EasyRPC.

#### GetServiceUri

Este método devuelve la URL correcta para conectar a los servicios remotos. Tiene compilación condicional para que el emulador de Android resuelva correctamente la URL.

#### ConfigureViewModelToViewMaps

En este método se asocian los ViewModels a cada vista correspondiente.

#### ConfigureSections

En este método se configuran las secciones. Se tiene que devolver una lista de una abstracción de tipo “Section”, que define el tipo de ViewModel de cada sección y su icono.

## Análisis de la aplicación

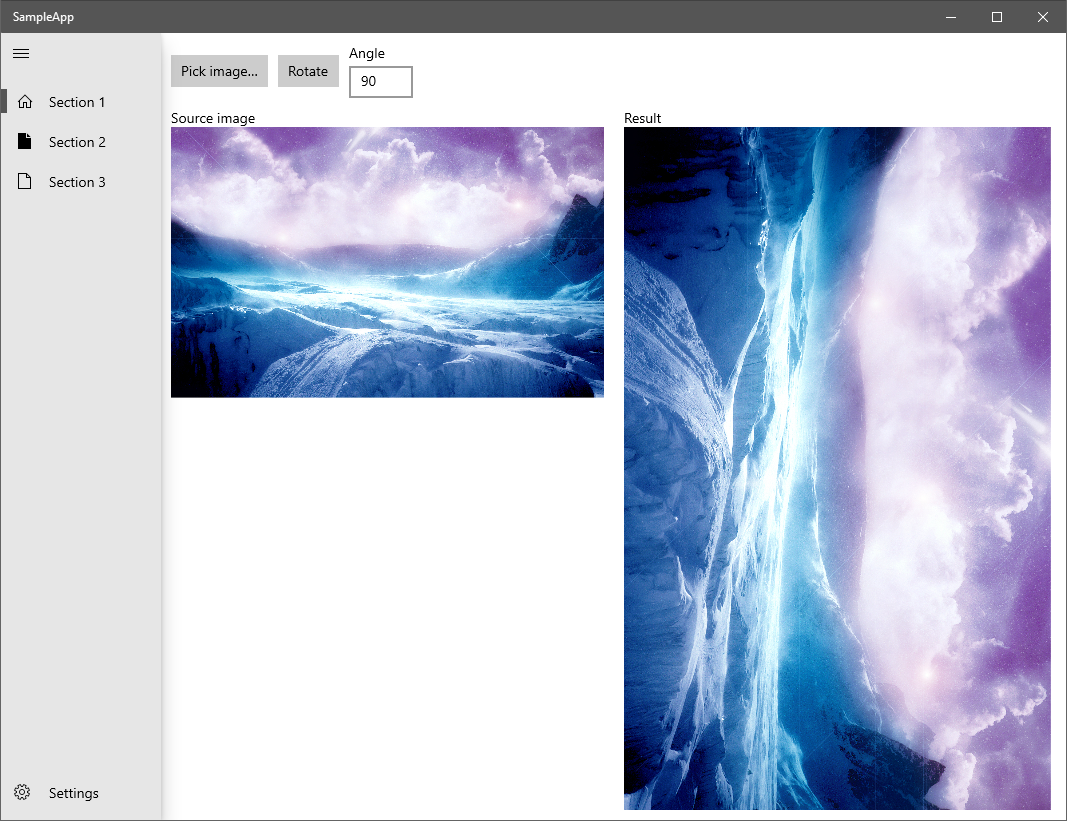
Como decíamos arriba, la aplicación se ha creado clonando la plantilla **UnoPlatform.AppTemplate**.

**Para complicarla hemos decidido hacer que la primera sección nos permita manipular imágenes.**

El funcionamiento es sencillo:

1. Indicamos una imagen (selección de archivo)
2. Elegimos un ángulo
3. Pulsar un botón “Rotar”
4. La imagen es rotada de acuerdo con el ángulo indicado

Para hacerlo más divertido, la imagen no la rotamos nosotros, sino que la enviamos a un servicio remoto para que la rote y nos la devuelve ya rotada. Nosotros simplemente mostramos el antes y el después.



# Consejos para la creación de aplicaciones

Pendiente

# Consejos para migración de aplicaciones existentes

Pendiente

# Anexo. ReactiveUI

ReactiveUI (RxUI) es un framework que nos facilita el diseño de nuestros ViewModels para hacer que su definición sea declarativa y más sencilla de entender. Se basa en System.Reactive (anteriormente conocido como Reactive Extensions o Rx)

Aunque no vamos a entrar en detalle, unas nociones básicas nos serán de gran ayuda para entender el funcionamiento de las aplicaciones expuestas.

## ReactiveObject

Todo modelo que queremos que use RxUI debe heredar de la clase ReactiveObject. Al hacerlo, obtendremos inmediatamente toda la funcionalidad.

## ViewModels reactivos

Nuestros ViewModels son los candidatos perfectos para lo dicho en el punto anterior, por lo que heredaremos todos nuestros ViewModels heredarán de ReactiveObject.

Lo primero que debemos saber es que usando RxUI, los ViewModels normalmente tienen su casi toda su lógica en el constructor. El motivo es que es precisamente en el constructor donde se va a definir el comportamiento del ViewModel de forma “reactiva”. Observemos un ejemplo:

class ViewModel : ReactiveObject

{

public ViewModel()

{

var ageChanged = this.WhenAnyValue(vm => vm.Age);

ageChanged.Subscribe(age => Debug.Write(age));

}

private int age;

public int Age

{

get => age;

set => this.RaiseAndSetIfChanged(ref age, value);

}

}

Vamos a analizar el código

En este ViewModel se define una propiedad “Age”. Sería una definición típica si no fuera por el hecho de que está llamando a un método RaiseSetIfChanged. Este método nos los brinda RxUI y se encarga de notificar el cambio del valor. Es necesario para que el resto de las partes del sistema se enteren. Es como una señal. Siempre se usa en las propiedades que son de lectura y escritura en la parte de escritura (setter)

La otra parte interesante es el constructor.

Veamos la parte de this.WhenAnyValue(…):

WhenAnyValue: El método WhenAnyValue es importantísimo. Su misión es crear una secuencia a partir de una propiedad. Una vez creada la secuencia podemos suscribirnos a ella y observar sus cambios.

En la última parte vemos la suscripción a la secuencia.

ageChanged.Subscribe(age => Debug.Write(age));

Aquí, simplemente estamos diciendo que cada vez que la edad cambie, se imprima en la ventana de depuración el valor de age. Siempre que cambie la propiedad Age, un nuevo valor vendrá por la secuencia y se ejecutará Debug.Write(age)

## Propiedades derivadas

Un caso de uso muy habitual es tener el valor de una propiedad compuesto de 2 o más propiedades.

Un ejemplo clásico es

Nombre completo = nombre + apellidos

Esto se consigue en RxUI haciendo lo siguiente

var fullNameSequence = this.WhenAnyValue(vm => vm.FirstName, vm => vm.LastName, (first, last) => firstName + " " + lastName);

en la variable fullNameSequence obtenemos una sequencia que combina FirstName y LastName siempre que alguno de ellos cambie y nos entrega el nombre completo.

Pero, atención, fullNameSequence es una secuencia, ¿cómo “proyectamos” esa secuencia a una propiedad del ViewModel?

Para eso tenemos el método ToProperty

### Método ToProperty

Se usa proyectar el último valor obtenido de una secuencia observable en una propiedad “normal” del ViewModel.

Dada una secuencia como la anterior (fullNameSequence), la llamada sería la siguiente

fullName = fullNameSequence.ToProperty(this, vm => vm.FullName);

Con ello estamos diciéndole, proyectar fullNameSequence a la propiedad FullName.

Lo que no es tan obvio es que este método devuelve un objeto del tipo ObservableAsPropertyHelper<string>. En realidad, el método ToProperty no proyecta sin más el valor, sino que crea un objeto “helper” que es una especie de almacén donde se va a almacenar el valor de FullName. Para que realmente se use el valor tenemos que definir la propiedad en el ViewModel de la siguiente manera:

public string FullName => fullName.Value;

Estamos obteniendo el valor desde el helper que hemos creado en el constructor. Es importante darse cuenta de que no hay setter. Las propiedades derivadas no cambian “desde fuera” sino que cambian como consecuencia de los cambios en las propiedades origen.

Para ilustrar mejor el mecanismo, veamos el ViewModel completo

class ViewModel : ReactiveObject

{

private readonly ObservableAsPropertyHelper<string> fullName = new ObservableAsPropertyHelper<string>();

public ViewModel()

{

var ageChanged = this.WhenAnyValue(vm => vm.Age);

ageChanged.Subscribe(age => Debug.Write(age));

var fullNameSequence = this.WhenAnyValue(vm => vm.FirstName, vm => vm.LastName, (first, last) => firstName + " " + lastName);

fullName = fullNameSequence.ToProperty(this, vm => vm.FullName);

}

public string FullName => fullName.Value;

private int age;

public int Age

{

get => age;

set => this.RaiseAndSetIfChanged(ref age, value);

}

private string firstName;

public string FirstName

{

get => firstName;

set => this.RaiseAndSetIfChanged(ref firstName, value);

}

private string lastName;

public string LastName

{

get => lastName;

set => this.RaiseAndSetIfChanged(ref lastName, value);

}

}

1. El motivo de mantener esta versión es que las versiones actuales tienen cambios incompatibles con el código de la plantilla de Uno Platform. Esto podría solucionarse en el futuro próximo [↑](#footnote-ref-1)