Uno Platform tutorial

# Introduction

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

Currently, it can deploy to the following systems:

* Android
* iOS
* Windows
* Web Assembly (web browser)

## Fundamentals

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

# First steps

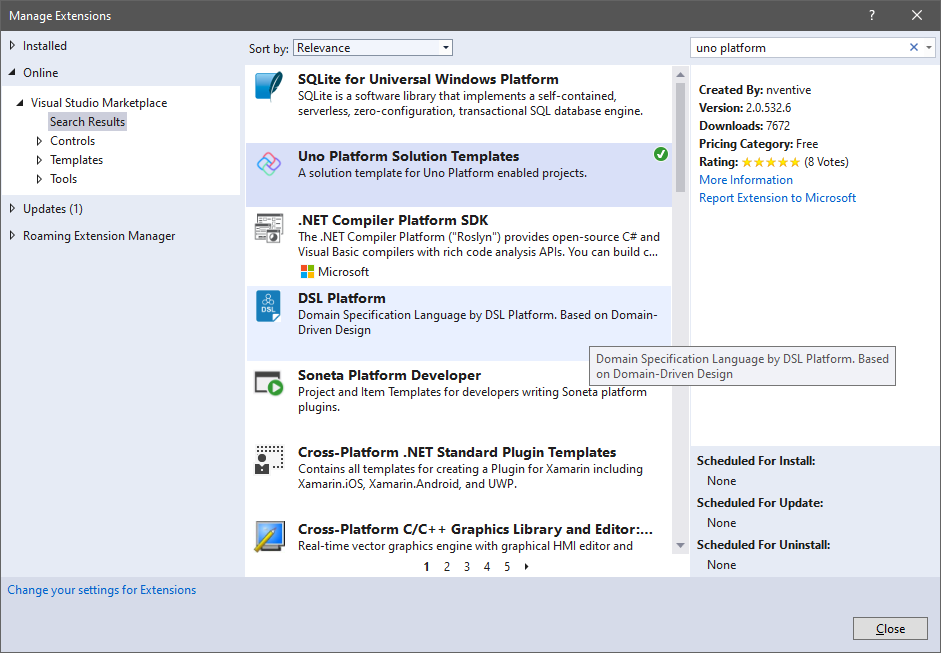
Developing with Uno Platform is very easy. However, you should have some previous knowledge.

* 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.

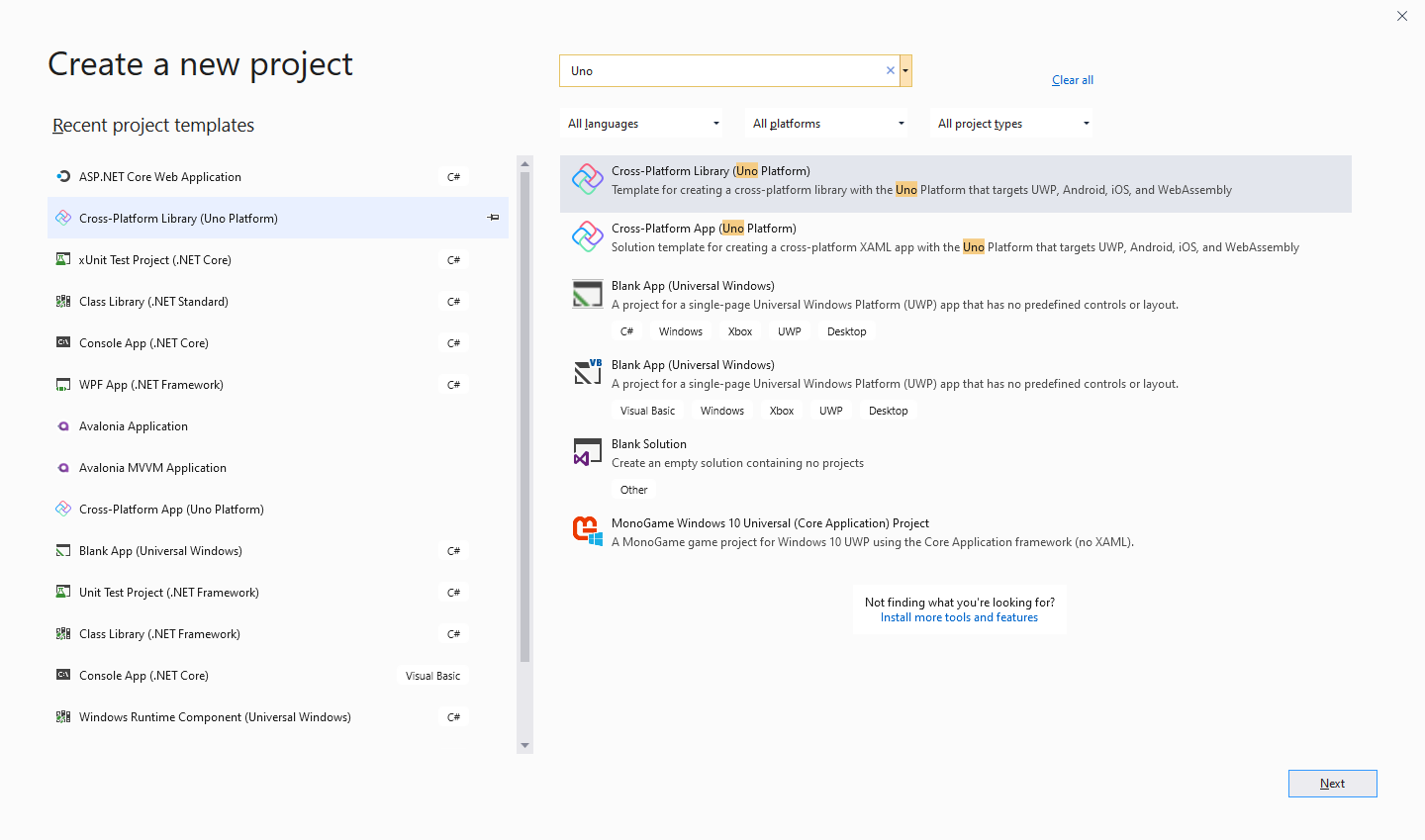
## First contact with Uno Platform

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”.



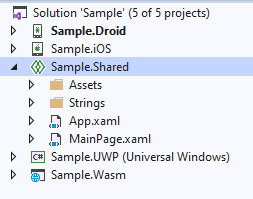
Once they’re installed, we will see them in the Visual Studio startup page or going to “Create new project”.



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 solution, 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.

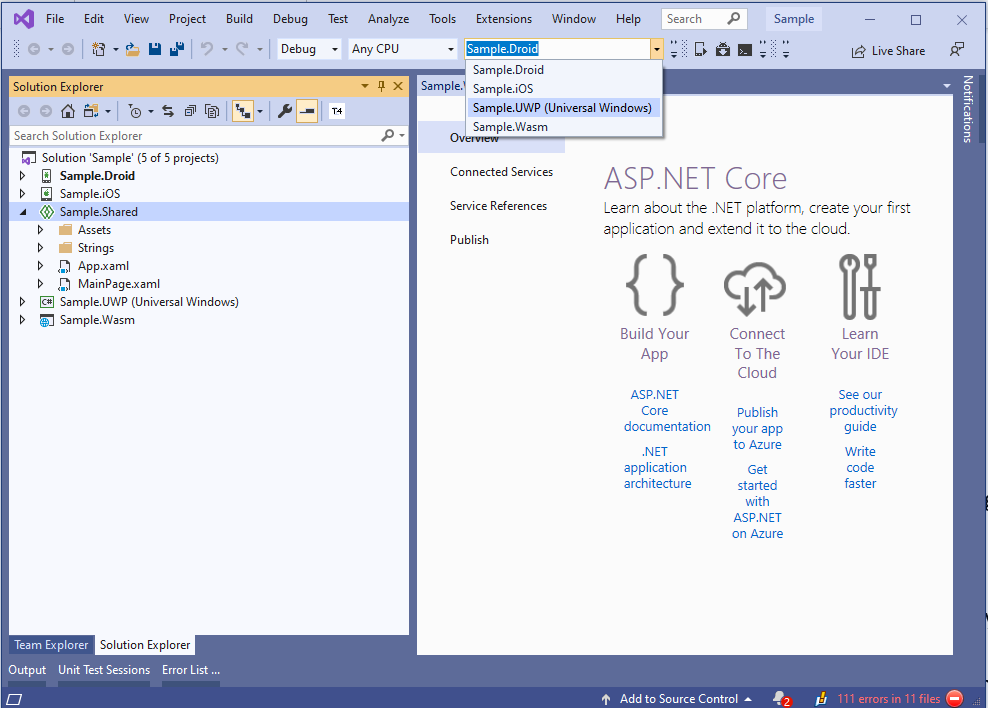
In short there are 2 kinds of project:

* 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.

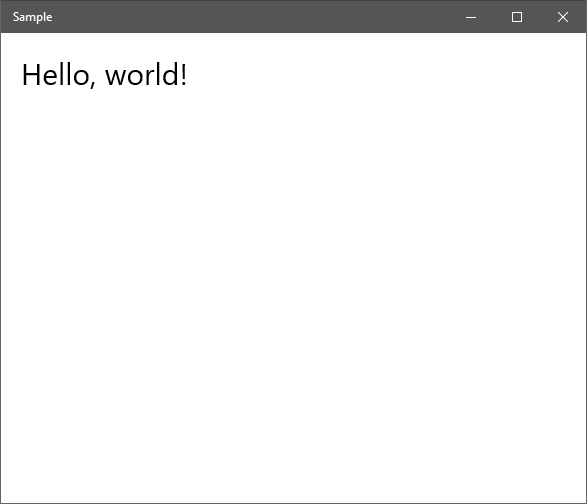
Go to the startup project dropdown



Choose the UWP project (Universal Windows)



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



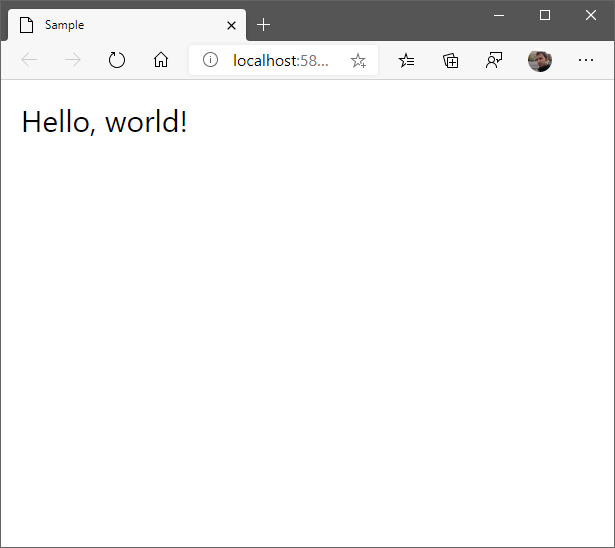
After the build is complete (it can take a bit the first time) a window should pop up in front of us saying “Hello, world!”. Good 😊

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 the browser will launch a new tab with our application running!

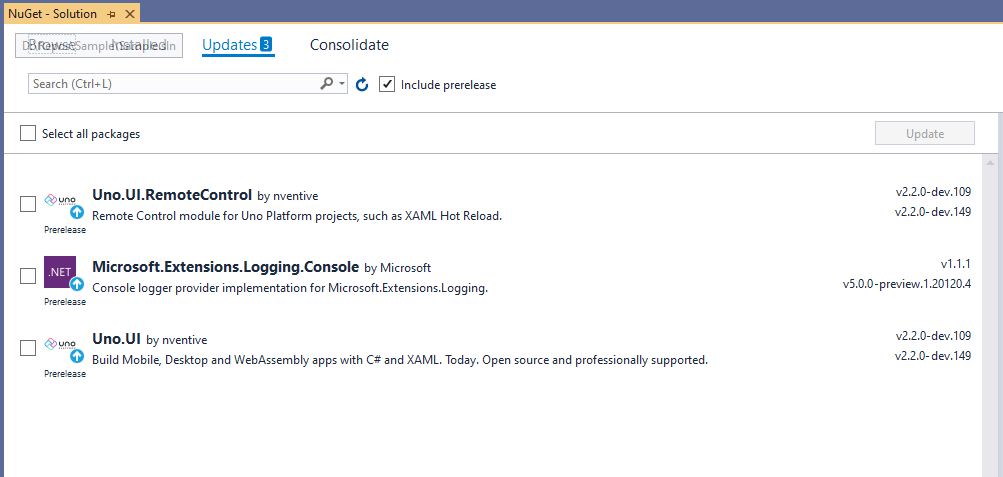


### Update the NuGet Packages

Please, remember that the development on Uno is quite fast and there are a lot of improvements and fixes coming up. Therefore, **it’s important to update the NuGet packages of Uno Platform** when you create your solution using a template. This is because the templates use a fixed set of packages and they are updated much less often that Uno Platform itself.

In addition, most of the current packages are being published as “prerelease”. To get the latest goodies, you definitely have to keep them updated as much as you can 😊

To update them, we will go to the Manage NuGet Packages for Solution (right-click the solution in the Solution Explorer) and go to Updates. Then, check everything except for the one called Microsoft.Extensions.Logging.Console[[1]](#footnote-1)



## Reference

We’ll design our user interface using the XAML markup language, concretely, the dialect that Universal Windows Applications use.

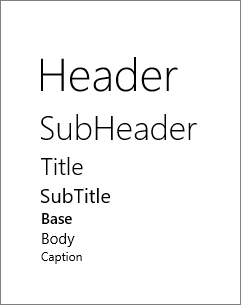
The basic elements inside the XAML are quite easy to use and are like the ones used in other XAML dialects (like Xamarin Forms or WPF), but there are some notable differences that we should consider.

## Basic Elements

### Controls

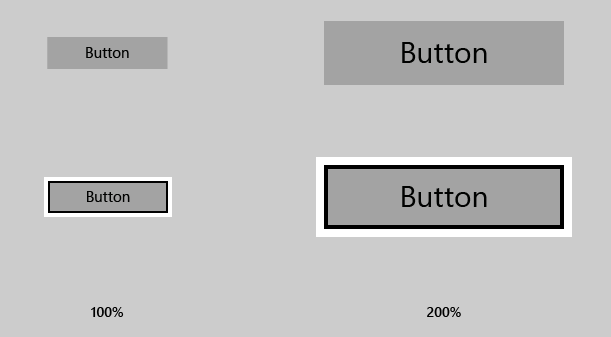
#### TextBlock

Shows a text. The **Text** property is commonly used with either a fixed text or with a Binding to a string property in a ViewModel.



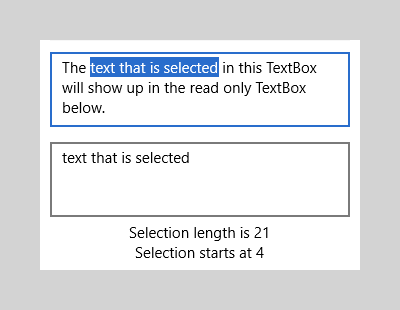
#### Button

It’s a normal button that performs an action. The action it does is usually encapsulated into a Command set via the **Command** property. Its text is set using the **Content** property that can be anything (even Images or other elements), but it’s usually set to a text.



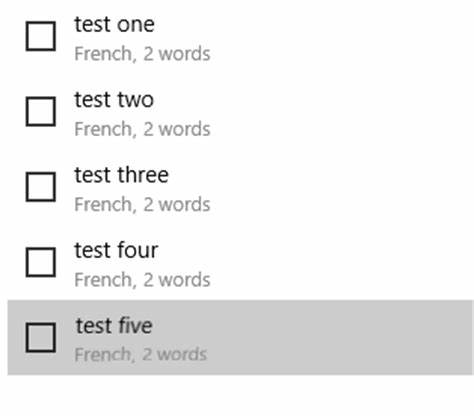
#### TextBox

It’s the control that allows the user write text on it. It’s commonly called “Input” or “Entry”. The **Text** property is commonly used with a Binding in TwoWay mode.



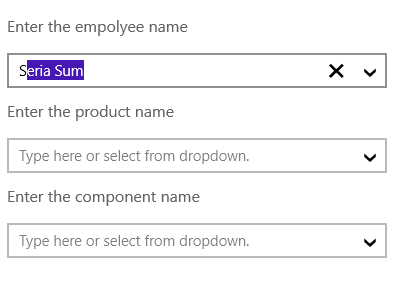
#### ListView

This is the control to represent lists of items. It’s usually used along with its **ItemsSource** property) that is a link to the items that we want to show. It’s also important to mention the **ItemTemplate** property, that controls the appearance of each item.



#### ComboBox

It’s the regular combined list control with a list of items that can be expanded using the arrow button.



#### Border

It’s used to control the appearance. It can contain other controls inside, so it’s more a box than just a border.



#### Image

The control to show images. The image we want to show is set via the **Source** property, that is usually a URI to the image file.



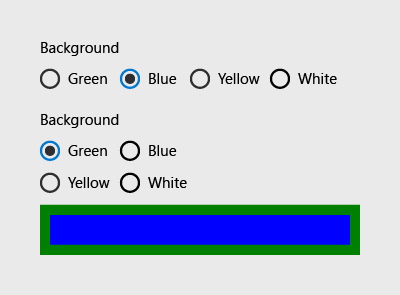
#### CheckBox

A box with the ability to check, uncheck and that has an intermediate state



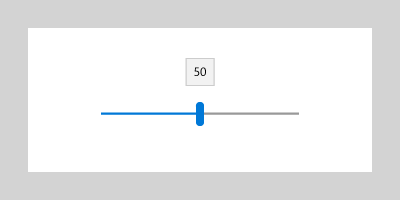
#### RadioButton

A control used to select a single exclusive option out of a group of available options.



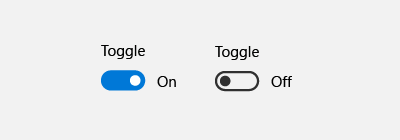
#### Slider

A control with sliding button that the user can slide to modify a numeric value within a range.



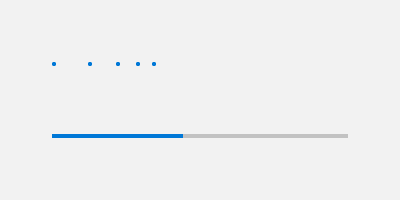
#### ToggleSwitch

To represent binary values



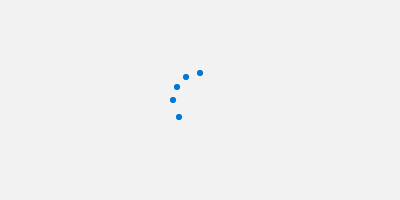
#### ProgressBar

A progress bar that can be either undetermined or determined. When the progress is undetermined, a set of dots with an animation will play. If it’s determined, the actual value will be shown.



#### ProgressRing

It’s similar to the ProgressBar but using a circle of dots. It’s used only for undetermined progress.

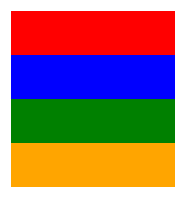


### Layout

Layout control how elements are place on the screen. Here I will show you the most common of them.

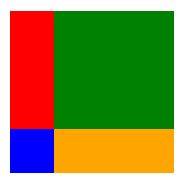
#### StackPanel

It’s a panel that stacks items horizontally or vertically.



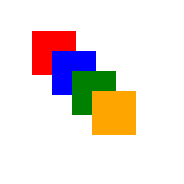
#### Grid

It’s a panel that lets us place controls like a table, using rows and columns.



#### Canvas

A panel we can use to place items at absolute locations (X and Y). Items ara placed using the **Canvas.Left** and **Canvas.Top** properties.



### Bindings

Bindings are a common and well-known mechanism to synchronize data between 2 objects.

One is the source and the other is the target. To each change in a bound property in the the source, the target is updated accordingly.

There are also Two-Way Bindings in which any change in the target also updates the source.

Example

* Standard (one-way) <Button Content=”{Binding ButtonText}” />
* Bidirectional (two-way) <TextBox Text=”{Binding Text, Mode=TwoWay}” />

We have another kind of bindings, x:Bind. These bindings are compile-time, but I don’t recommend using them unless is strictly necessary or you feel more comfortable with them, since there are basic scenarios that they don’t support.

### Data Templates

Templates (Data Templates) are a key element in all the XAML platforms. The most common are the ones in ListView (ItemTemplate), but every control offers their own. We can customize them to our needs, changing how the content is presented.

### Styles

Styles are a way to reuse sets of property values, so they are maintainable and centralized. They are also a common element XAML. The have a TargetType, that is the kind of element which the style targets to, and a list of Setters, that apply the desired values to each property.

Example:

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

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

</Style TargetType=”Button”>

Styles are normally defined in **resource dictionaries**, that are containers where resources are stored using a key to look them up later, using a special syntax (the “StaticResource” markup extension).

We can apply a style to a control like this:

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

# Let’s build a simple application. App1.

We’re going to a simple application for training. In it, we’ll put to practice our previous experience with other XAML frameworks, and we’ll evolve and adapt it to the development lifecycle of Uno Platform.

If you’re a Universal Windows developer, you’ll soon find out that most things you already know are exactly the same in Uno.

In the first place, we need to clone the following repository:

<https://github.com/SuperJMN-Tutorials/UnoPlatformTutorial-App1>

You can check the history to understand each step. For convenience, I’ve added a link to each commit, so you see the actual changes I’ve made into the source.

Let’s start with the “Hello world” application we did in the beginning of the tutorial. Remember that it’s already a functional application.

1. We’ve added a ViewModel [[commit](https://github.com/SuperJMN-Tutorials/UnoPlatformTutorial-App1/commit/ff35be8c0e76865e64497970fcda3e038a9752b8)]

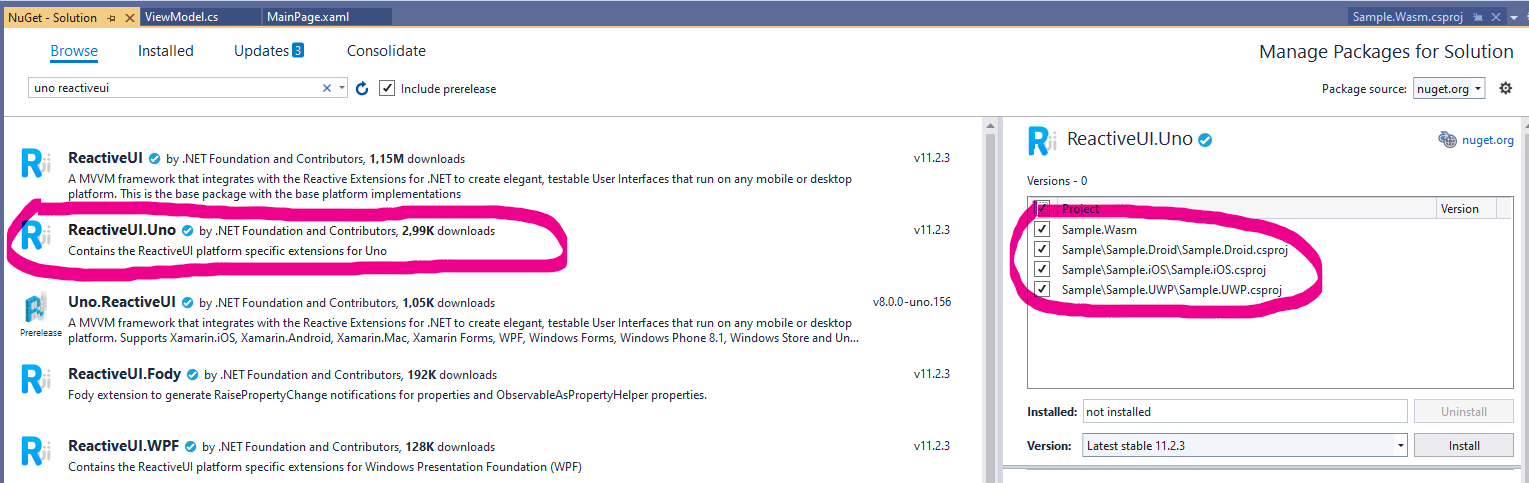
We’ve added into the Share project, for simplicity. We’re not adding new projects, but as the code grows, we can decide otherwise.

1. We’ve added ReactiveUI [[commit](https://github.com/SuperJMN-Tutorials/UnoPlatformTutorial-App1/commit/3d26fe0292b3b559bf9632515c86de2244e15cb3)]

We’re using it to design our View Models.

It’s very convenient to start familiarizing with reactive programming 😊 ReactiveUI is the framework I suggest, no matter what XAML framework you use.

To install ReactiveUI, we cannot install the official version, but one that is specially adapted to work with Uno Platform. This is, however, an exception: most Packages can be use directly without problems.



**IMPORTANT:** Whenever you need to install a NuGet Package or add a project reference, don’t forget to do it to ALL THE HEADS. Otherwise, the project won’t build.

1. We added basic properties to our View Model and we added a simple XAML to our view [[commit]](https://github.com/SuperJMN-Tutorials/UnoPlatformTutorial-App1/commit/b75c63b655dcec96b116479204203441ecc77f82)
2. We improved the look and feel a bit [[commit](https://github.com/SuperJMN-Tutorials/UnoPlatformTutorial-App1/commit/695516e3b8b6da1e86b3b2b59721730286fe7e11)]
3. We added a UserControl to encapsulate the XAML of each contact card [[commit](https://github.com/SuperJMN-Tutorials/UnoPlatformTutorial-App1/commit/b718e3b397521a7371dee416d1cb869675d0e192)]
4. We only had a ViewModel, but as we added functionality, it evolved to become a new entity: The ContactViewModel [[commit](https://github.com/SuperJMN-Tutorials/UnoPlatformTutorial-App1/commit/bafa158ad55e09a4ea8ada82c4be3c80415e46e5)]
5. We added a ListView with a fixed list of contact cards [[commit](https://github.com/SuperJMN-Tutorials/UnoPlatformTutorial-App1/commit/346a7f54274fa5dc763207b049d1d0cbecb20acf)]
6. We introduced a new main ViewModel bound to the contact list [[commit](https://github.com/SuperJMN-Tutorials/UnoPlatformTutorial-App1/commit/346a7f54274fa5dc763207b049d1d0cbecb20acf)]
7. We added the feature to add new contacts [[commit](https://github.com/SuperJMN-Tutorials/UnoPlatformTutorial-App1/commit/4dc48f6bbd461a545d4edbefa206f55329e28bf1)]
8. We added the feature to delete contacts [[commit](https://github.com/SuperJMN-Tutorials/UnoPlatformTutorial-App1/commit/4a671f25ce1ed9d5765a0eec35cc9cc61dcf6710)]

# Let’s build a more complex application. App2.

The next application that is part of this tutorial can be obtained by cloning the following repository:

[https://github.com/SuperJMN-Tutorials/UnoPlatformTutorial-App2](https://github.com/SuperJMN/UnoPlatformTutorial-App2)

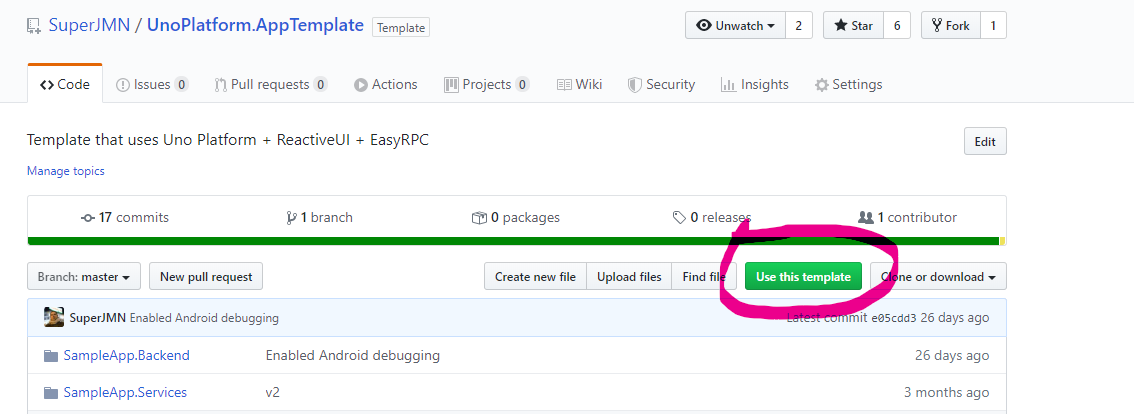
This time, I won’t give so much detail like in the previous application. It will be enough with an explanation of each of its parts.

**NOTICE:** this application wasn’t built from completely scratch, but from an “application template” I created for you all, to make speed up your developments.

The template is here.

<https://github.com/SuperJMN/UnoPlatform.AppTemplate>

**NOTICE:** You can click the “Use this template” button in GitHub to base your project in it

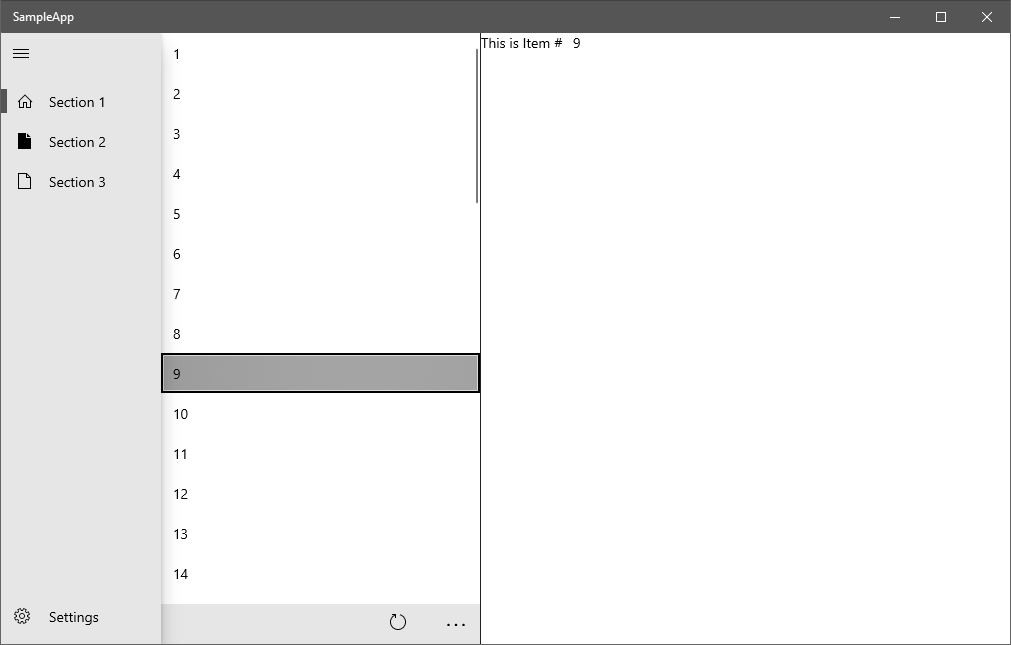


With this template we’ll get the basic skeleton of a generic application with data access that you can customize to your needs easily. We can remark the following:

* Preconfigured dependency injection using Grace, a power and easy to use DI IoC container.
* Distributed services layer using EasyRPC, a framework that provides a very easy way to invoke methods remotely from our services with zero boilerplate code.
* ReactiveUI, to apply reactive programming and MVVM.
* Left sidebar navigation.
* ViewModel-based navigation

## Analysis of the application template

When you first use the template and run the application, you’ll see something like this:



In the sidebar, we’ll have 3 sample sections. These sections can be customized by modifying the views inside SampleApp.Shared\Views.

We’ll find the 3 views with their respective UserControls. We will also find their ViewModels inside SampleApp.Shared\ViewModels. We can adapt them to our logic.

### CompositionRoot

There is a file called Composition.cs.

This is the composition root of the application, a common concept in the applications that use dependency injection. Among other things, it contains the service configuration, in addition to the mappings between the Views and their ViewModels.

#### ConfigureServices

In this method, the services are registered with our container (Grace). This container will resolve all the services in our application, including the remote ones. The remote services are located using a “Proxy Namespace”, that is a mapping between a CLR-namespace and a URL, so upon a request to resolve a remote service, the container will provide a proxy that will transparently invoke the remote methods like they were local. This is provided by Grace + EasyRPC.

#### GetServiceUri

This method should return the URL to connect to the remote service. It uses conditional symbols so the Android emulator can resolve the correct URL.

#### ConfigureViewModelToViewMaps

This is where the association between ViewModels are associated to a concrete view type.

#### ConfigureSections

Here you configure the sections of the application (the ones that appear on the left sidebar). A section is related to the ViewModel it uses and its icon.

## Analysis of the application

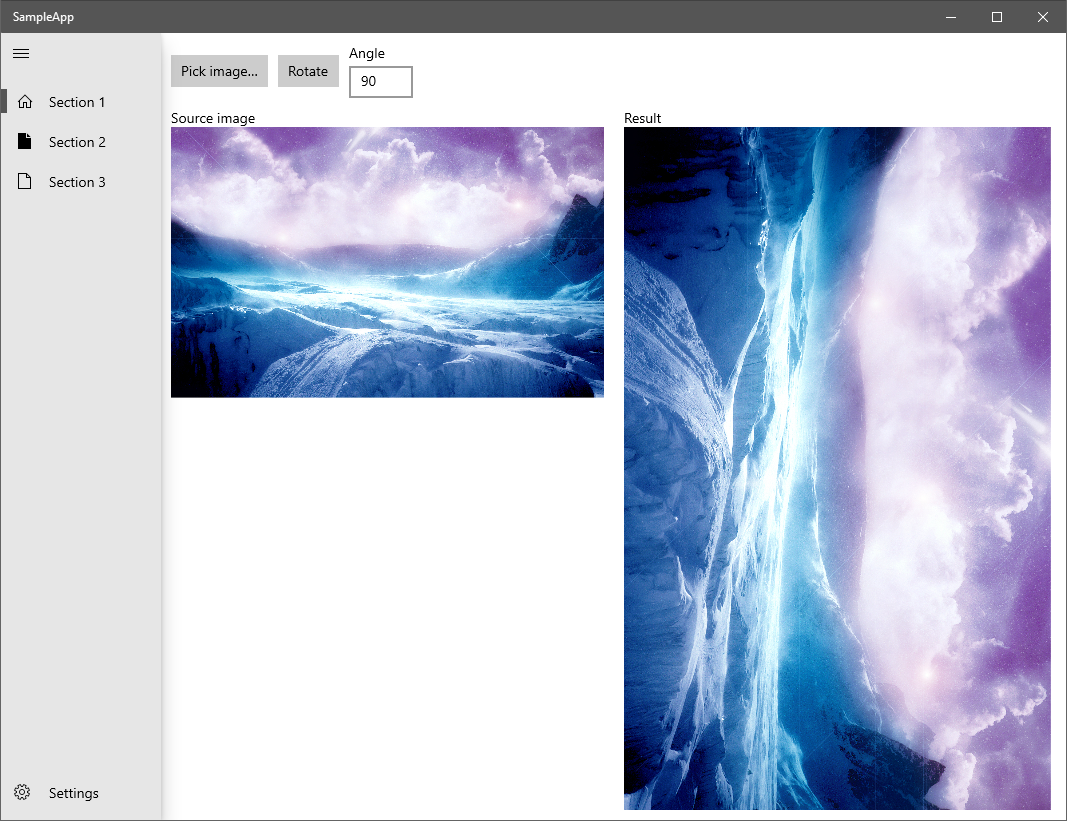
Like we already mentioned, the application has been created using the template **UnoPlatform.AppTemplate.**

**To make it a bit more complex, I decided to turn the first section into an image manipulation section.**

It works like this:

1. The user picks an image (via file section)
2. They set an angle in degrees
3. They click a button that says “Rotate”
4. The image is rotated in the given angle

To make it more fun, I decided that we won’t be rotating the image ourselves, but instead sending it to the remote service to do it. This way, the image will go though the wires in a round-trip travel. First, we send the image unrotated, and we’ll receive it from the service with the desired rotation applied. We just have to show the “before” and “after”.



# Tips for building apps with Uno

We should be aware that not everything that works in UWP will work in other heads. There are some shortcomings.

The recommended cycle of development would be the following:

* Develop a small part of the UI
* Validate the appearance and the behavior in the other platforms
* Repeat

If you find a problem, try to simplify the layout to use controls that are well supported.

For example, if you want to use a header in a ListBox, you would normally use the **Header** property. If this property isn’t supported, you can still get the desired look using a TextBlock on top of your ListView.

Also, don’t forget that WASM is the most limited of the heads. If it’s important for you, I would test the WASM head, first. If it works for WASM, it’s very probable that it will work in the other heads.

If something isn’t working after a workaround, you can go to the official support chat <https://gitter.im/uno-platform/Lobby> and ask. The Uno Team is there, along with a lot of committed and friendly people, me included 😊

# Appendix. ReactiveUI

ReactiveUI (RxUI for short) is a framework that makes it easier to design our applications in a reactive fashion. It’s based on System.Reactive, AKA “Reactive Extensions” or Rx. The framework has a lot of features, but we will focus on how ViewModels are implemented using RxUI.

Although we’re not get into the details, a basic knowledge about RxUI will be of great help to understand the applications in this tutorial.

## ReactiveObject

If we want a model to be reactive, we need to make it inherit from the ReactiveObject class. Doing so will provide it the full list of reactive features, some of them via extension methods, some others via inheritance.

## Reactive ViewModels

Our ViewModels are the perfect candidates to be reactive models. Thus, our ViewModels should derive inherit from ReactiveObject.

The first thing we should know is that **ViewModels using RxUI hold almost all their logic inside the constructor**. The reason is that “reactive things” are normally defined in a declarative way. Those declarations, that are only set once make the constructor the perfect place to put them.

Let’s see an example:

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);

}

}

We can see that the ViewModel defines an “Age” property. It would be a typical definition of a regular property if it wasn’t for the call to “RaiseSetIfChanged”. This method is provided by RxUI and its goal is to notify a value change. It’s needed for the rest of the system to know when the value is changed. It’s a way to ping them. This method is used for mutable (modifiable) properties, that is, properties that have a getter and a setter.

The other interesting part is the constructor, as I mentioned before.

Let’s take a look to **this.WhenAnyValue(…).** This method is very important. Its mission is to create a sequence from a property in the ViewModel. Once the sequence is created, we can subscribe to it and observe its changes. Each time the Age property changes, the new value is pushed to the observable (the ageChanged sequence).

In the last part, we see how to subscribe to the sequence.

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

The statement above means that we want to write each of the elements pushed to the ageChanged sequence to the Debug window.

## Derived properties

A very common case is to have a value composed by the values of 2 or more properties. A classic example of this is a a “full name”. A full name is composed of both “first name” and “last name”.

Formally, we would define: Full name = first name + last name

This is possible in RxUI doing the following

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

In the fullNameSequence, we will have a sequence that combines FirstName and LastName whenever any of them change, and the full name be calculated accordingly.

But, beware! fullNameSeqence is a sequence and sequences cannot be represented directly as a property. How do we “project” that sequence to a property in our ViewModel?

That’s why we have the **ToProperty** method.

### ToProperty

It’s used to project the most recent value of an sequence to a regular property in our ViewModel.

Given a sequence like the previous (fullNameSequence), the call would be the following

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

With it, we’re telling RxUI to project **fullNameSequence** to the **FullName** property

What’s not so obvious is that this method returns an instance of type ObservableAsPropertyHelper<string>. What the ToProperty method does is to create a “helper” object that is something like a value store in which it will put the value of FullName. To use the value inside it we will have to define the property like this:

public string FullName => fullName.Value;

We’re retrieving the value from the helper. It’s important to realize that there’s no setter for derived properties because they can’t be set directly: their value changes when the originating properties change.

To illustrate the mechanism better, let’s see the complete code of the ViewModel

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. The reason to keep this version is that newer versions have incompatible changes with Uno Platform. This could change in future releases. [↑](#footnote-ref-1)