Diary of a Windows developer
Windows, mobile and much more

Template10: a new template to create Universal Windows apps – The basics

Posted on September 7, 2015 by qmatteoq

When you launch Visual Studio 2015 to create a Universal Windows app for Windows 10 for the first time, the first thing you notice is that the number of available templates is very low. If, when it comes to Windows / Windows Phone 8.1 development, Visual Studio 2013 offered many templates to quickly implement the most used layouts (Hub, Pivot, etc.), in Visual Studio 2015 we have just one template called "Blank app".

The reason of this choice is that, in the past, the standard templates were used by the developers without very good skills in developing the user experience of an application (like me 😊) to make the job easier and to avoid to create the user interface from scratch. The downside of this approach is that, during the course of time, it leads many developers to publish a lot of applications that looked very similar to each other. Consequently, Microsoft has decided, with Windows 10, to leave more freedom to the developers and to avoid forcing them to adopt a specific design patter. It's important to highlight, however, that this doesn't mean that you can do whatever you want when it comes to create the user interface, without following any criteria. Providing a great user experience is still one of the key pillars to create a successful app! As such, I strongly suggest to read the official guidelines to design Universal Windows apps on the Dev Center: https://dev.windows.com/en-us/design

The new blank template provided by Visual Studio 2015 is very simple and it's the perfect starting point for every application. However, sometimes, especially when you need to work on a complex project, it can be even too simple. When we need to handle advanced scenarios (like using the MVVM pattern or managing the state of the page) we need to create all the required infrastructure.

To make the life the developers easier, a team of Microsoft people, lead by Jerry Nixon (a Technical Evangelist very well known for his series of MVA trainings about Windows 10 development) have started to work on an advanced template called **Template10**. It's an open source project, available on GitHub at https://github.com/Windows-XAML/Template10 and, as such, you're more then welcome to give your contribute, by helping with the documentation or by making some enhancements and submitting a pull request.

The project is still evolving, but the goal is very clear: Template10 wants to become the preferred choice for Universal Windows apps developers, independently from the development approach they prefer (code behind, MVVM, etc.) or from the toolkits and libraries they already use in their applications. If you want to learn more about the project, you can read the official FAQ on https://github.com/Windows-XAML/Template10/wiki/Questions. In the near future, Template10 will be released as a NuGet package and as a Visual Studio extension. This way, you'll be able to use it as a starting point for your app directly from the **New project** menu in Visual Studio 2015. Meanwhile, however, even if the main library that defines the template's components is considered stable and ready for production, it's available only on GithHub. Let's see which are the required steps to start creating a new Universal Windows app using Template 10.

UPDATE: Template10 is now available also as NuGet package: http://www.nuget.org/packages/Template10/1.0.2.2-preview. However, to find and install it, you'll need to enable in the NuGet Package Manager UI the **Include prerelease** options, since the package is still in beta.

This post is just the first of a series: in this one we're going to see the basic concepts. In the next ones, we'll see some more advanced topics like the available controls or the MVVM pattern.

Create the first project

The first step is to clone the GitHub repository. We can achieve this goal using the tools provided directly by Visual Studio 2015.

1. Open the window called **Team Explorer**. As default behavior, it's placed on the right side of the screen, in the same section where you have access to the **Solution explorer**'s window.
2. You'll find a section labeled **Local Git Repositories**, which lists all the Git repositories on your computer. Press the **Clone** button.
3. In the first field (the one with the yellow background) you need to specify the URL of the GitHub repository, which is https://github.com/Windows-XAML/Template10
4. In the second field, instead, you need to set the local path on your computer where you want to clone the project.
5. Now press the **Clone** button: Visual Studio will take care of downloading the project and copying it on your computer.

From now on, by using Team Explorer, you'll be able to quickly access to the repository. By using the **Sync** option you'll have the chance to keep the local repository up to date with the latest version of the library.

A good starting point for a new Universal Windows app based on Template10 is to use the **Blank** project, which is stored inside the **Templates (Project)** folder. This project already provides the basic infrastructure required by Template10 to work properly. However, in this post we're going to start with an empty project and to manually configure Template10: this way, it will be easier to understand how it works under the hood.

Let's start by creating a new Universal Windows app with the **Blank app** template provided by Visual Studio. Then, let's right click on the solution in Solution Explorer and choose **Add existing project**. We'll have to look for the folder where we've cloned the GitHub repository and, specifically, we'll have to find the project named **Template10Library**: you'll find in the folder with the same name and it's defined by the **Template10Library.csproj** file. Once you've added the project to your solution, it's time to add a reference to Template10 in our app. Just right click on the project of the application, choose **Add reference** and, in the list, choose the **Template10** project. Now we are ready to start writing some code!

The bootstrapper

The **App** class is the starting point of every Universal Windows app: its goal is to initialize the main window of the application with its frame, which acts as a container of all the pages of the application and provides the necessary infrastructure to handle the navigations. The base structure of the **App** class (stored in the **App.xaml.cs** file), however, can be a little bit confusing, especially for new developers. You'll find a lot of fixed code, which is part of the standard initialization and, as such, it creates just "noise", since you'll probably never change it. Another weak point is the application's lifecycle management. The App class offers different event handlers to manage the different application's state (launching, suspending, resuming, etc.) but, based on the activation's type, you can have multiple entry points. Template10 makes the initialization of the app easier to understand and to use by providing a **bootstrapper,** which is a class that replaces the **App** one and that simplifies the code. The first step to use is to replace the **App** class with the **BootStrapper** one, which is included in the **Template10.Common** namespace. Here is a sample definition:

```
<common:BootStrapper
        x:Class="BasicSample.App"
        xmlns:common="using:Template10.Common"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

</common:BootStrapper>
```

You can notice that the main **App** node has been replaced with the **BootStrapper** one. Consequently, you will have also to update the code-behind class (the **App.xaml.cs** file), so that the **App** class inherits from the **BootStrapper** one, like in the following sample:

```
sealed partial class App : Template10.Common.BootStrapper
{
    public App()
    {
        InitializeComponent();
    }

}
```

Here is, instead, how a complete definition of the **BootStrapper** class looks like:

```
sealed partial class App : Template10.Common.BootStrapper
{
    public App()
    {
        InitializeComponent();
    }

    public override Task OnStartAsync(StartKind startKind, IActivatedEventArgs ar
    {
        NavigationService.Navigate(typeof(MainPage));
 return Task.FromResult<object>(null);
    }
}
```

You can immediately notice how, compared to the standard **App** class, the code is much more simple to understand. The core is the **OnStartAsync()** method, which is the starting point of the application, regardless from the activation's scenario. No matter if the app has been opened using the main tile, a secondary tile or from a toast notification, the **OnStartAsync()** method will always be invoked to let you, as developer, handling the main navigation. In the simplest scenario (your app doesn't have secondary tiles or doesn't support secondary activation entries), the method will look like the one in the sample: it simply takes care of redirecting the user to the main page of the application. The second operation is just a workaround to support asynchronous code inside this method. To properly support async / await operations, the method has been configured to return a **Task** object; however if, like in this case, you don't have the requirement to perform any asynchronous operation, we simply return an empty Task, so that the code can compile.

The main advantage of the **OnStartAsync()** method is that the bootstrapper, under the hood, always takes care of preparing the Frame, which is required to handle the rendering and the navigation of the pages of the application. The standard **App** class, instead, takes care of initializing the Frame only in the main entry point (the **OnLaunched()** event handler), leaving you to do all the "dirty work" in case the app has been opened from a different activation point.

To help you understanding better this approach, here is a sample bootstrapper used to handle the activation from a secondary tile:

```
sealed partial class App : Template10.Common.BootStrapper
{
```

```
    public App()
    {
        InitializeComponent();
    }

    public override Task OnStartAsync(StartKind startKind, IActivatedEventArgs ar
    {
        AdditionalKinds cause = DetermineStartCause(args);
        if (cause == AdditionalKinds.SecondaryTile)
        {
            LaunchActivatedEventArgs eventArgs = args as LaunchActivatedEventArgs
            NavigationService.Navigate(typeof (DetailPage), eventArgs.Arguments);
        }
        else
        {
            NavigationService.Navigate(typeof (MainPage));
        }

        return Task.FromResult<object>(null);
    }
}
```

The code is very simple to understand. Thanks to the internal method **DetermineStartCause()** we are able to determine which is the activation point of the app and to redirect the user to the most appropriate page. In the sample we are handling the scenario where the tap on a secondary tile redirects the user to a specific item of the application (like a category of news): in case the activation's cause is **AdditionalKinds.SecondaryTile**, we retrieve the tile's arguments (thanks to the **Arguments** property) and we pass it to the page called **DetailPage**. In case, instead, the application has been regularly launched using the main tile, the user is simply redirected to the main page.

With the same approach we can handle all the other activation's scenarios: from a toast notification, from a uri, from a sharing contract, etc. In this case, the **DetermineStartCause()** method will return you the **Other** value of the **AdditionalKinds** enumerator. To properly determine what's happened, we need to use the **Kind** property of the activation's parameter which type is **IActivatedEventArgs**.

Inside the bootstrapper class you have also the chance to override three other methods, which can be useful to handle special scenarios.

1. **OnInitializeAsync()** is invoked when the app is initialized, right before calling the **OnStartAsync()** method. It's useful if you have any service or feature that requires to be initialized at startup (for example, analytics services like Application Insights).
2. **OnResuming()** has the same purpose of the method with the same name of the standard **App** class and it's invoked when the application is resumed after it has been suspended. Typically, you're not required to handle this event; the reason is that it's triggered only when the application has been resumed from a suspension state and not from a termination. As such, since the process was kept in memory, you don't have to retrieve the state you may have previously saved, since it's still there. This method can be useful if you need to refresh the data in case, for example, the user has resumed the application after a while.
3. **OnSuspending()** is invoked when the application is suspended. Typically, this method is used to save the application's state, so that it can be restored in the case the app is terminated by the operating system due to low resources. However, as we're going to see in the next posts, with Template10 this operation is not required: the bootstrapper is able to take care of this scenario automatically.

Extended splash screen

The splash screen is a static image (placed in the Assets folder of your project) which is displayed to the user during the app's initialization phase. Once the operation is completed, the splash screen is removed from the screen and the Frame redirects the user to the main page of the app. As best practices, the app's initialization should be as quickest as possible: if the operation isn't completed within 10 seconds, the app will be terminated by the OS. However, it's not uncommon that the app may need more than 10 seconds to be ready, especially if the application's data has to be downloaded from a remote source (like a REST service). To avoid the app to be terminated by the OS, one of the best practices is to avoid loading the data in the App class, but to perform it directly in the main page of the app. This way, the initialization phase will be very quick and the main page will immediately take control of the app. However, since the app is now ready and the initialization operation is completed, the OS won't try to kill the app anymore, even if the main page would need more than 10 seconds to load all the data.

This approach works great, but often it doesn't provide a good user experience: the user is redirected to the main page of the app which, however, will be empty and it will display just a loading message, until all the data is ready to be displayed. To improve the UX many applications have introduced the concept of **extended splash screen**: after that the basic initialization is completed, the user will be redirected to another page of the app which will look exactly like the splash screen. The difference here is that, since we aren't talking anymore about a static image but we're dealing with a real page, we can add additional UI elements to notify to the user that a loading operation is in progress (like a **ProgressBar** or a **ProgressRing** control). If you want to learn more about this topic, the MSDN documentation offers some guidance on how to implement an extended splash screen at https://msdn.microsoft.com/en-us/library/windows/apps/xaml/hh868191.aspx

As you can see in the document, there's a bit of work to do, other than just creating the page that will act as extended splash screen. Template10 simplifies this approach, by offering a class that takes care of handling the transition between splash screen –> extended splash screen –> first page of the app. Let's see how to use it.

The first step is to create the splash screen itself, by adding a new **UserControl** to your application. You have the freedom to add everything you want in this control. However, since the goal of this control is to hide the transition from the static splash screen, typically the first thing you want to add is an **Image** control that displays the same image used as splash screen. The following sample shows a control where the image is placed inside a **Canvas** with the same background color of the image. In addition, we add a **ProgressRing** control to notify to the user that a loading operation is in progress.

```
<UserControl
    x:Class="BasicSample.Views.SplashScreenView"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <Grid>
        <Canvas x:Name="MyCanvas" HorizontalAlignment="Stretch" VerticalAlignment
            <Image x:Name="MyImage" Stretch="None" Source="/Assets/SplashScreen.p
        </Canvas>
        <ProgressRing VerticalAlignment="Bottom" HorizontalAlignment="Center" Wic
                      Margin="0, 0, 0, 100"/>
    </Grid>
</UserControl>
```

In code behind we use the **SplashScreen** class (which returns some important information about the image, like the size and the position) to configure the **Image** control in the same way. This way, the user won't notice the transition from the real splash screen to the extended one.

```
public sealed partial class SplashScreenView : UserControl
{
    public SplashScreenView(SplashScreen splashScreen)
    {
        this.InitializeComponent();
        Action resize = () =>
        {
            MyImage.Height = splashScreen.ImageLocation.Height;
            MyImage.Width = splashScreen.ImageLocation.Width;
            MyImage.SetValue(Canvas.TopProperty, splashScreen.ImageLocation.Top);
            MyImage.SetValue(Canvas.LeftProperty, splashScreen.ImageLocation.Left
        };
        Window.Current.SizeChanged += (s, e) => resize();
        resize();
    }
}
```

The resize and reposition operations are defined into an **Action** which, other than being executed when the user control is initialized, is connected also to the **SizeChanged** event, which is triggered when the size of the app's window changes. It's required since, when the app runs on a desktop, the user is able to resize the window at any time and, consequently, the splash screen needs to readapt itself.

Now that we have created our custom control, we are ready to use it as a splash screen, thanks to the **SplashFactory** class provided by the **BootStrapper:**

```
public App()
{
    InitializeComponent();
    SplashFactory = e => new SplashScreenView(e);
}
```

SplashScrenView is the name of the UserControl we have just created, while **e** is the instance of the **SplashScreen** class that we have passed to the control's constructor (it's the one we've used to calculate the size and the position of the image).

Now we're ready to test it: as soon as the application takes more than a few seconds to initialize, the user will be redirected to the extended splash screen and, only at the end of the loading process, to the main page of the app. If you want to simulate a heavy data loading to test the splash screen, you can add a delay in the **OnStartAsync()** method, before the navigation to the main page is performed, like in the following sample:

```
sealed partial class App : Template10.Common.BootStrapper
{
    public App()
    {
        InitializeComponent();
        SplashFactory = e => new SplashScreenView(e);
    }

    public override async Task OnStartAsync(StartKind startKind, IActivatedEventA
    {
     await Task.Delay(TimeSpan.FromSeconds(5));
        NavigationService.Navigate(typeof(MainPage));
    }
}
```

Once the initialization is completed, the bootstrapper will invoke the **OnStartAsync()** method and, until it's triggered the navigation to the main page of the page, the extended splash screen will be displayed. In this sample, since we've added a 5 second delay, the splash screen will be displayed for 5 seconds before the user will see the main page.

Navigation

Another important feature offered by Template10 is the automatic handling of the page stack. If you already have some experience in developing Windows Store apps for Windows and Windows Phone 8.1, you'll know that navigation management is one of the main differences between the two platforms. On Windows Phone, you don't have to add any visual element in the user interface to allow the user to go back to the previous page of the app, since every device has a hardware Back button. However, the same doesn't apply on Windows, since there is no Back button on a desktop or on a tablet.

Windows 10 has changed this behavior; also the desktop version, in fact, offers a build-in management of the Back button, by providing a virtual back button which can be placed in different positions, based on the scenario:

1. If the app is running in desktop mode, the Back button will be added in the app's chrome, on the top right corner.
2. If the app is running in tablet mode, the Back button will be added in the system bar, between the Start and Cortana buttons.

Typically, in a standard application, it's up to the developer to handle this feature, by using the **SystemNavigationManager** class, which provides the required APIs to handle the visibility of the button or the navigation events. Template10, instead, automatically enables this management on every device family. Consequently:

- When the application is running on the desktop and the page stack isn't empty, on the top right corner the virtual Back button will be visible. If, instead, the stack is empty (for example, because the user is on the first page), the button will be automatically hidden.
- When the application is running on a mobile device, pressing the hardware Back button will automatically redirect the user to the previous page of the app or, if the stack is empty, to the Start screen of the device.

If we want to disable this behavior (for example, because we want to handle the navigation by ourselves, by adding a button in the UI) we can set the **ShowShellButton** property of the **BootStrapper** class to **false**:

```
1  public App()
2  {
3      InitializeComponent();
4      ShowShellBackButton = false;
5  }
```

When this option is disabled, you will notice that the virtual Back button in the app's chrome will never be displayed.

In the next post

In this post we've learned some of the basic concepts of Template10 and which are some of the features that we can start to use, right away, to improve our application. We've just scratched the surfaced and, in the next posts, we're going to see the "true power" of Template10, especially when it's used in combination with the MVVM pattern. If, in the meanwhile, you want to start doing experiments with Template10, you can use the samples that are provided on the GitHub repository. Happy coding!

Related

Template10: a new template to create Universal Windows apps – MVVM
October 5, 2015
In "Universal Apps"

Template10: a new template to create Universal Windows apps – The controls
September 11, 2015
In "Universal Apps"

App Studio: a new tool by Microsoft for Windows Phone developers
August 7, 2013
In "Windows Phone"

This entry was posted in Universal Apps and tagged Template10, Windows 10, wpdev. Bookmark the permalink.

5 Responses to *Template10: a new template to create Universal Windows apps – The basics*

jerry *says:*
February 12, 2016 at 12:12 am

Wow. /jerry

Reply

Selom Ofori *says:*
February 22, 2016 at 3:29 pm

As someone who is doing yet-another-windows-port to UWP, I can say with confidence to avoid Template 10. All this code will render your project unportable if Microsoft makes yet-another-api-update as they been doing since windows phone came out. The best way to write maintainable code is to just stick to the defaults and do it yourself.

Replacing your app.xaml is just a recipie for disaster.

Reply

qmatteoq *says:*
February 22, 2016 at 5:04 pm

I can't really answer to this comment, every developer has the right to find the best approach for him. Personally, I have a completely different opinion. Template10 is very minimal and, except wrapping the App class in a bootstrapper (which, however, faced a minimal change from 8.1 to 10), every thing else is plain standard C# / XAML. In addition, Template10 supports some features that I would however need to implement in some ways in my app and, in case something fundamental changes in the Windows core, I would need to change anyway. Of course, this is my opinion based my experience, I totally respect yours even if is different 🙂

Reply

Robert *says:*
February 9, 2017 at 11:29 am

In a way I agree with Selom and there's a well known downside on using most frameworks: if something goes wrong, and beleve me that's goin to happen sometime, you don't have the development team to support you and you're practically on your own. I did Caliburn, Assisticant, and now Template10 and it cost me more effort with all kinds of issues and unexpected

deep knowledge you need to face problems. After a few days I still have no working Template 10 in my VS2015. Neither do I have got support and advice to solve this problem. So my advice is, stick to the basic tools, Blank UWP template, and build from there your own well known extensively tested layers and tools. I did, and it was worth the effort.

Robert Schuurmans
(About 10 years of development experience VS, C#, WPF enz.)

"Everyone can give support at the birth of a child when all proceed fluently the natural expected way, except when running into complications, then you will need a fully skilled medic."

Reply

qmatteoq *says:*
March 4, 2017 at 2:30 pm

I think that the best thing of being a developer is that you can choose the approach you prefer and that there isn't a "right" or "wrong" choice. Personally, I don't agree with you because I prefer a more "open source and community" approach and to rely on a framework that have a wide support from the community and from other developers that are facing the same challenges rather than having to reinvent the wheel from scratch, but it's just my opinion. I'm not absolutely saying that I'm right and you're wrong 🙂

Reply