

Hand­s-on lab

Lab 1: Introducing Template 10

May 2015

Contents

[Overview 3](#_Toc420057385)

[Exercise 1: Getting to Know Template 10 4](#_Toc420057386)

[Task 1 – Installing the Template 10 templates 4](#_Toc420057387)

[Task 2 – Navigating the folders 6](#_Toc420057388)

[Task 3 – MVVM 11](#_Toc420057389)

[Exercise 2: Classic and Compiled Bindings 12](#_Toc420057390)

[Task 1 – Open the starter project 13](#_Toc420057391)

[Task 2 – Create the model 13](#_Toc420057392)

[Task 3 – Create the view 15](#_Toc420057393)

[Task 4 – Add classic and compiled bindings 16](#_Toc420057394)

[Exercise 3: Binding to the Property of an Element 18](#_Toc420057395)

[Task 1 – Open the starter project 19](#_Toc420057396)

[Task 2 – Create the ElementName binding 19](#_Toc420057397)

[Summary 25](#_Toc420057398)

Overview

* 1. Welcome to a series of hands-on labs designed to immerse you in Universal Windows Platform app development. As you work though the labs, you will create a functional, real-world app that makes use of some of the key new features available in Windows 10. By the end of the series, you will know how to build an app that incorporates many of the key characteristics of a great app for the Windows Store, including:
* A user experience that employs the signature Windows 10 controls.
* A user experience that responds across large and small displays.
* Handling of lifecycle and app-model events to properly save and restore state and to roam settings so that users can seamlessly move among tasks and even devices.
* Seamless integration with modern hardware to implement features such as photo and video capture.
* Secondary tile pinning, notifications, and badges to keep your app’s content alive and ever-present to users.
* Integration with the Windows Store APIs for trial and in-app purchasing.

In this first lab in the series, we’ll get started by examining the different parts of the Blank template and getting to know the basics of our Windows App.

# Objectives

* 1. This lab will provide:
  + An introduction to the template 10 blank template.
  + An introduction to MVVM

# System requirements

* 1. You must have the following to complete this lab:
  + Microsoft Windows 10
  + Microsoft Visual Studio 2015

# Setup

* 1. You must perform the following steps to prepare your computer for this lab:
  2. Install Microsoft Windows 10.
  3. Install Microsoft Visual Studio 2015.
  4. Install the Universal Windows Platform SDK.

# Exercises

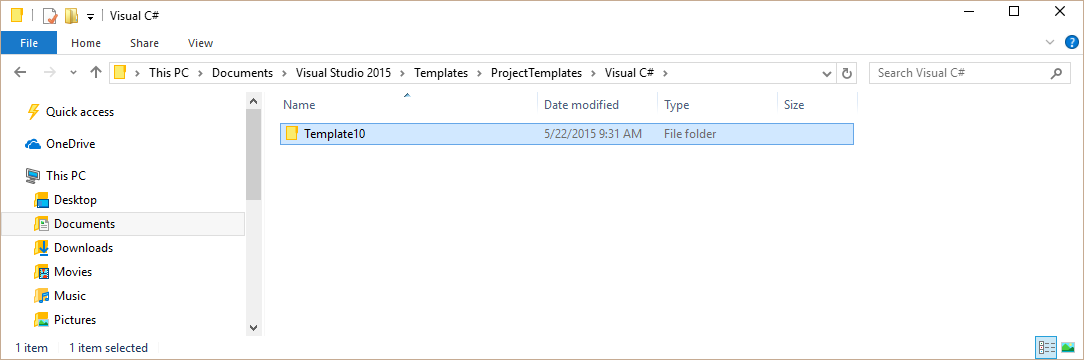
* 1. This Hands-on lab includes the following exercise:
  2. Getting to Know Template 10
  3. Estimated time to complete this lab:  **30 to 45 minutes**.

Exercise 1: Getting to Know Template 10

1. In this exercise, we will cover the basics of the Template 10 blank template.
2. Task 1 – Installing the Template 10 templates
   1. Before we can explore the blank Template 10 templates, let us first add them to Visual Studio 2015.
3. Open the **Assets** folder in the Hands on Labs directory and locate two files: **Blank.zip** and **ToDo.zip**.
   1. **Note:** **Blank.zip** and **ToDo.zip** are exported project templates, and they form the foundation for many of the labs and exercises. Each exercise will contain a zip archive of the project source for the exercise starting point, as well as the source for the completed exercise. During the early exercises, these projects will be based upon a modified version of the Blank template, and we will modify the code to demonstrate features. Later exercises will leverage the ToDo template, which is a more fully featured sample, Contoso Stack, that we will build upon.

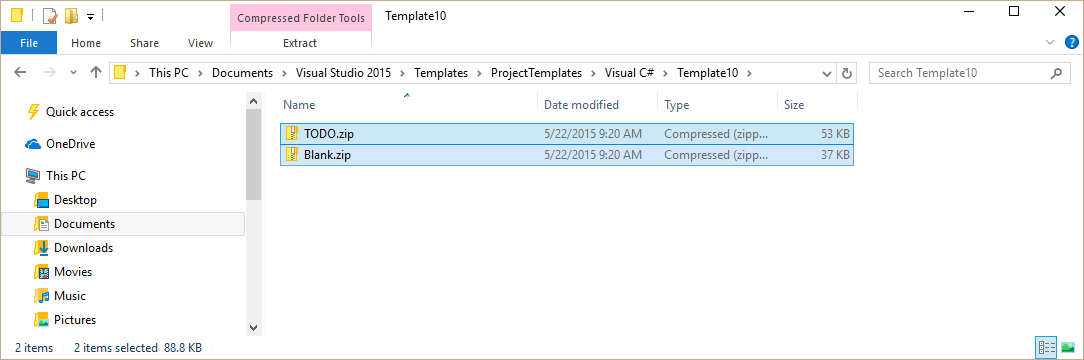
We recommend using the Blank template as the foundation for your apps as it provides strong implementations of many of the patterns necessary to build maintainable and rich Universal Windows Platform (UWP) applications.

1. Open another File Explorer window and navigate to **This PC > Documents > Visual Studio 2015 > Templates > ProjectTemplates > Visual C#**.
2. Create a new folder and name it Template10.

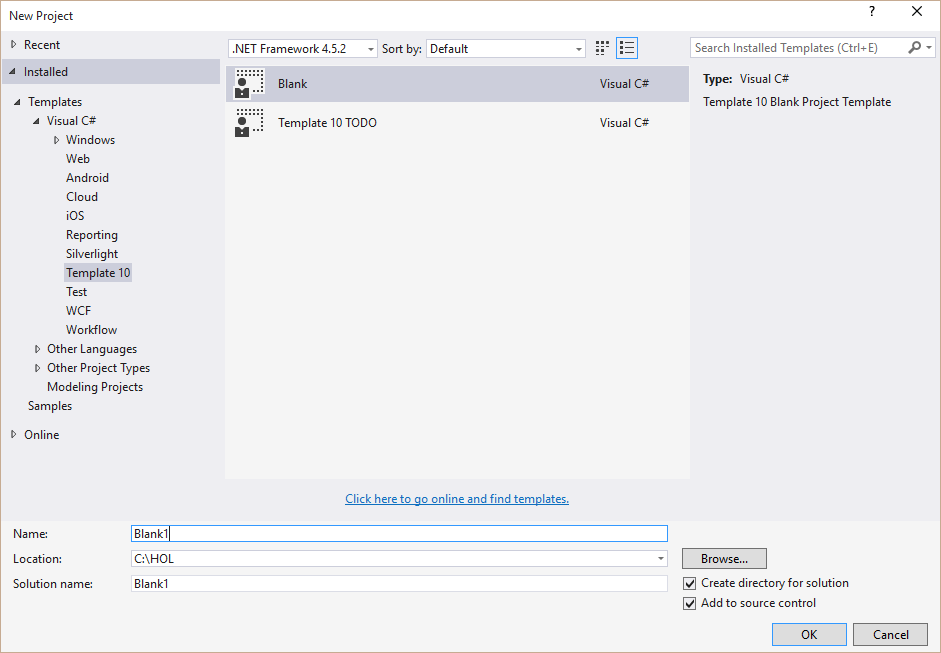


* + 1. **Figure 1**
    2. *Create the Template10 folder.*

1. Open the Template10 folder and copy the Blank.zip and ToDo.zip folders to that location.

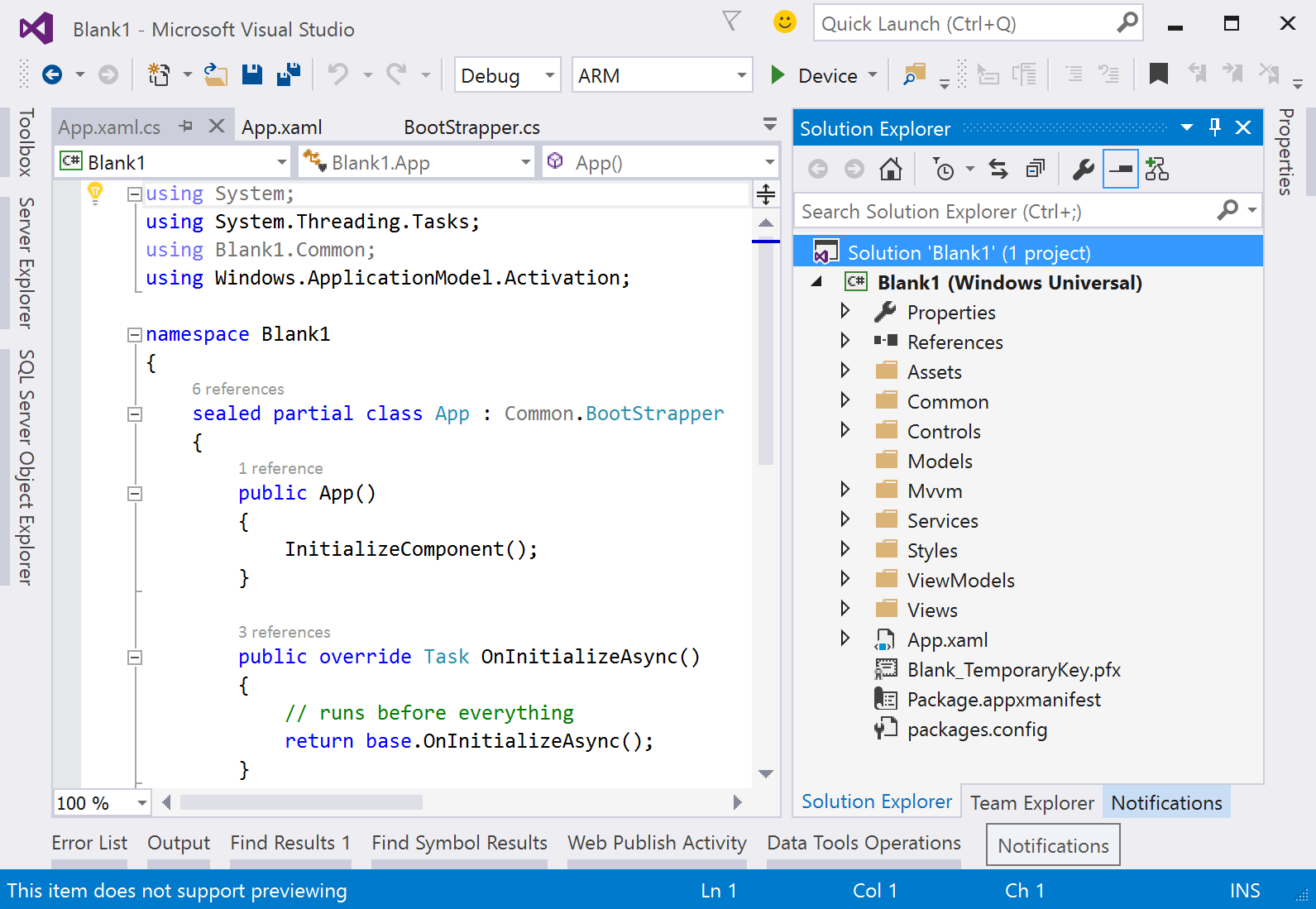


* + 1. **Figure 2**
    2. *The template zip files copied to the Template10 folder.*

1. To verify the templates have installed correctly, open Visual Studio 2015, select **File > New > Project**. Expand the **Installed > Visual C#** nodes of the template tree and look for **Template 10**. Select that node and you will see two options – Blank and Template 10 TODO.
   * 1. 
     2. **Figure 3**
     3. *The installed templates in the New Project dialog.*
2. You have successfully installed the templates. In the next task, you will create a new project using Template 10.
3. Task 2 – Navigating the folders

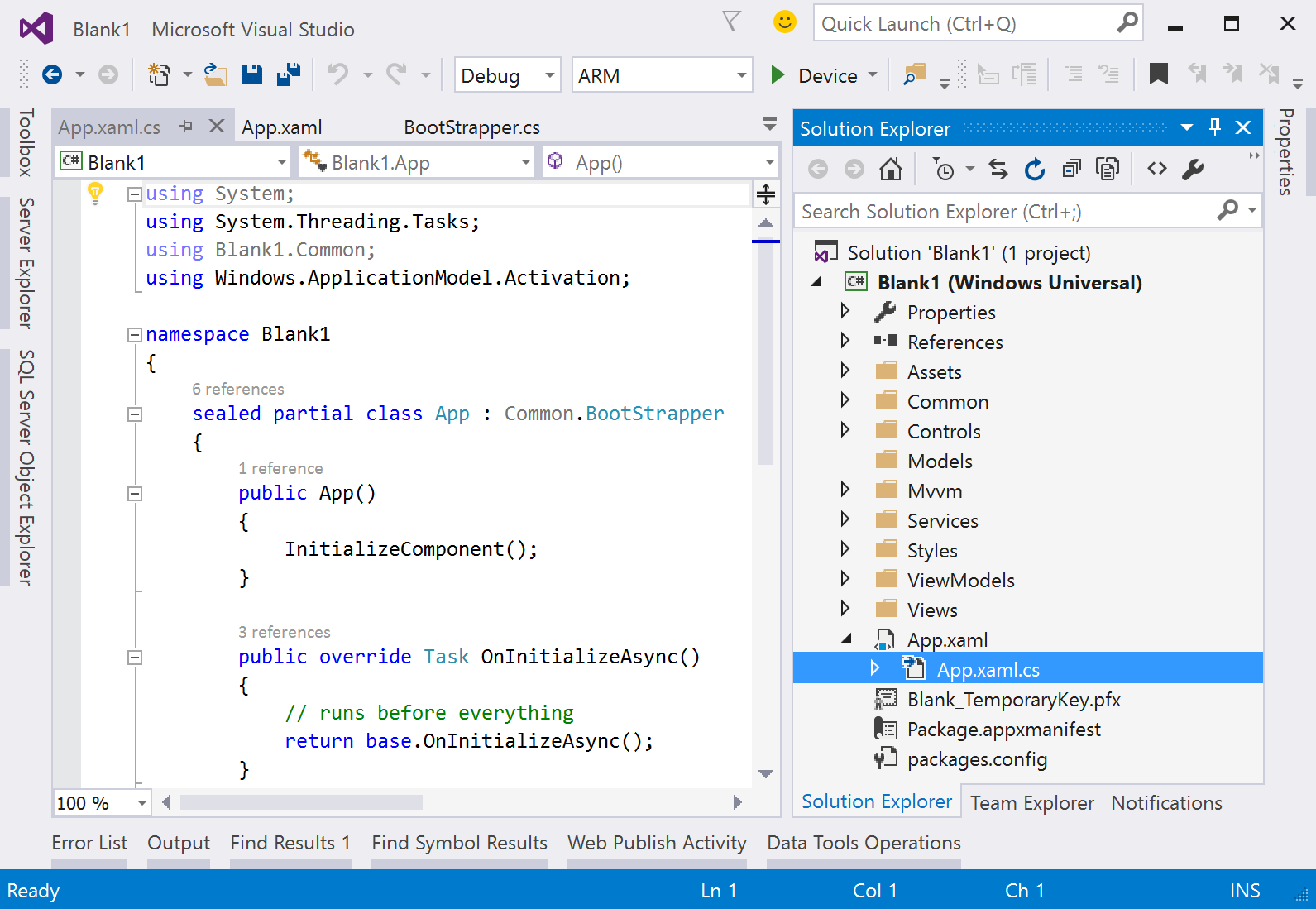
Now that you have installed the template, let’s create a new project and walk through the folders and files to discover how it works.

1. Open Visual Studio 2015 and create a new project using **Visual C# > Template 10 > Blank**
2. Once the project has opened, open the **Solution Explorer** pane. Here you will see a listing of the files in your project.



* + 1. **Figure 4**
    2. *The Template 10 Blank Template in the Solution Explorer*

1. Let’s begin by examining the first bit of code that runs whenever you start a Windows Universal Platform app. Expand the arrow next to **App.xaml** to see its associated files. Here you will see **App.xaml.cs**. Double click this file to open it.



* + 1. **Figure 5**
    2. *App.xaml.cs*

1. **App.xaml.cs** is called as soon as your application starts. It defines what *View* to show on start and how to deal with things like Tile activation and File association activation. Here you can see that our **App.xaml.cs** is quite small with only three methods - the majority of the startup sequence has been abstracted away into a base class called **BootStrapper.cs**.
   * 1. **C#**
   1. sealed partial class App : Common.BootStrapper
   2. {
   3. public App()
   4. {
   5. InitializeComponent();
   6. }
   7. public override Task OnInitializeAsync()
   8. {
   9. // runs before everything
   10. return base.OnInitializeAsync();
   11. }
   12. public override Task OnStartAsync(StartKind startKind, IActivatedEventArgs args)
   13. {
   14. // start the user experience
   15. NavigationService.Navigate(typeof(Views.MainPage));
   16. return Task.FromResult<object>(null);
   17. }
   18. }
2. Let’s take a look at this class, which you will find in **Common > BootStrapper.cs**. This class is does a lot of the heavy lifting for us, allowing the developer to focus on building the app instead of implementing boilerplate functionality.
3. The **Bootstrapper** class handles the initial setup of two primary features of any app:
   * 1. App Lifecycle
     2. Navigation
4. The **Bootstrapper** class consolidates the many lifecycle events, such as *Suspending, Resuming* and *Activation* (via File, Search or Share), into a single consolidated function **InternalLaunchAsync**. This function is invoked when your application is starting and sets up many aspects of your application such as the root *Frame*, *Services* and *Navigation Handlers.* Take a look at these lines:

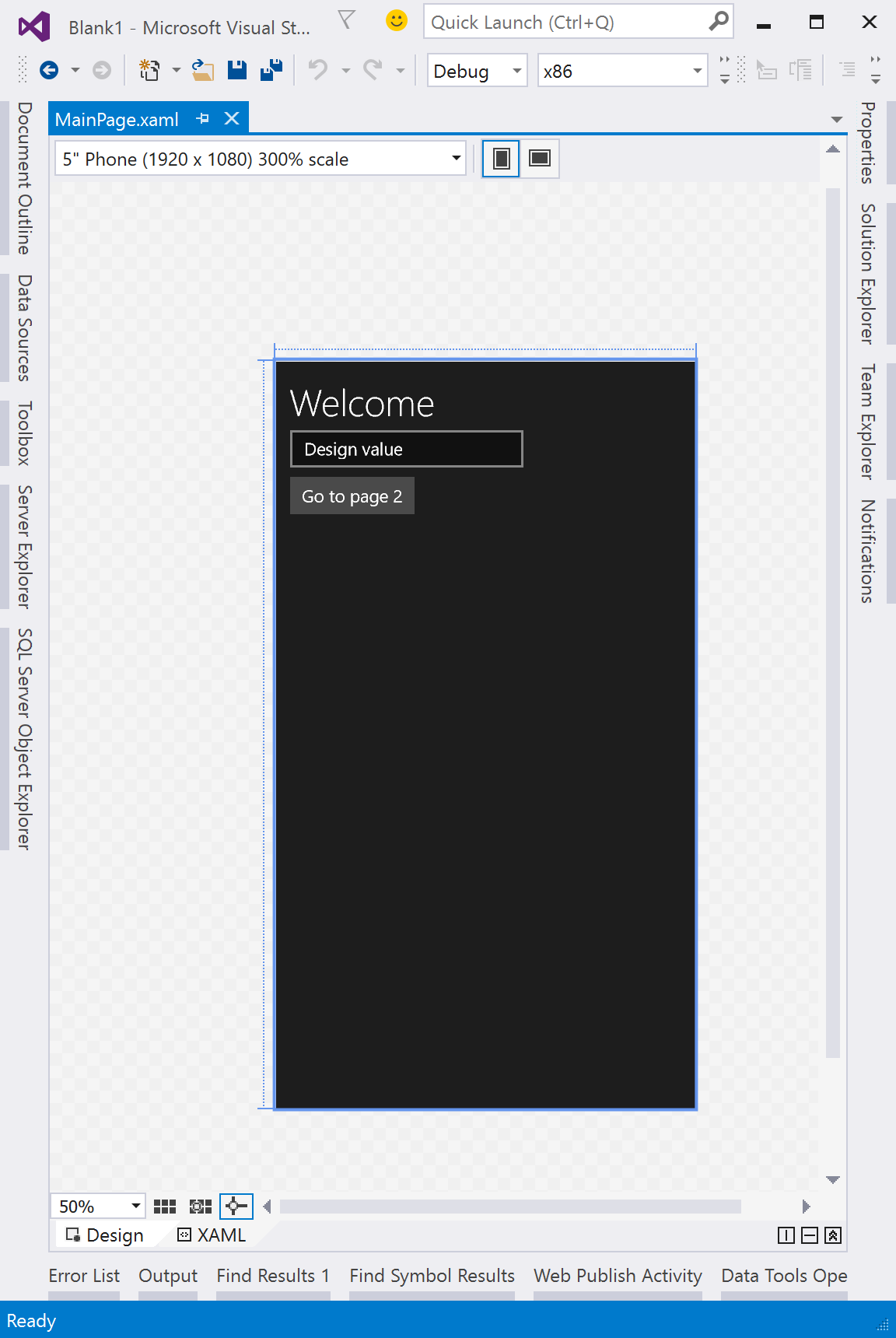
**C#**

* 1. // setup frame
  2. RootFrame = RootFrame ?? new Frame();
  3. RootFrame.Language = Windows.Globalization.ApplicationLanguages.Languages[0];
  4. NavigationService = NavigationService ?? new Services.NavigationService.NavigationService(RootFrame);

1. These lines create what is called a **Frame**. A **Frame** is a navigation concept that we will cover in depth in a future lab, but for now think of it as a container for your application’s UI. The UI is commonly a collection of controls called a Page. You can navigate from one Page to another in your Frame using the **NavigationService**. In this code, you can see we create a new Frame, set the default language of the Frame, and create a new **NavigationService** passing in the Frame the service will switch Pages in. We make this frame the first thing our users see by setting it as the content of our window.

**C#**

* 1. Window.Current.Content = RootFrame;

1. So how can we specify which Page will show first on startup? Move back to **App.xaml.cs** and look at the method **OnStartAsync**. In this method, we use the **NavigationService** we defined in the previous step to navigate our Frame to our first page **MainPage**.
   * 1. **C#**
   1. public override Task OnLaunchedAsync(ILaunchActivatedEventArgs e)
   2. {
   3. this.NavigationService.Navigate(typeof(Views.MainPage));
   4. return Task.FromResult<object>(null);
   5. }
2. Next, take a look at the **MainPage** view, which you will find in **Views > MainPage.xaml**. Double click on this file to open it. Here you can see the Page in a designer that allows you to modify the look and functionality of elements in the Page. Future labs will walk you through XAML and Views in more detail.
   * 1. 
     2. **Figure 6**
     3. *MainPage.xaml*
3. Let us continue exploring other areas of our project:
   * 1. In the **Assets** folder, you will find the images used for the logos and splash screen.
     2. In the **Controls** folder, you’ll see a file called **BackButton.xaml**. This file defines a *user control* that can be implemented in a *Page*. In this case, it defines a button with a left pointing arrow indicating ‘Back’.
     3. The **Services** folder contains the code for the **NavigationService**. This is another complex set of code. If you’re interested, take a look at the classes inside to see if you can understand how it works.
     4. Also in the **Services** folder is another service called **KeyboardService*.*** This service helps process the input received from different keyboard events to assist with navigation such as the left, right, top, bottom or backspace keys.
     5. The **Styles** folder has one file in it call **Custom.xaml**. This file defines a look and feel for a **UserControl** that can be applied to different pages. In-depth use of styles will be covered in a future lab.
4. To effectively explain the remaining two folders, **Mvvm** and **ViewModels,** we will need to examine a concept called Model-View-View Model or *MVVM.*
5. Task 3 – Understanding MVVM

In this task, we will discuss the basic principles of MVVM and how it relates to building Universal Windows Applications.

* 1. When building any kind of non-trivial application, you will quickly find that making a change in one part of your application will break functionality in another. In order to mitigate these breaking changes, you can use a pattern called Model-View-View Model or MVVM. The basic principle of this pattern splits your application into three parts in order to decouple it, thus making it easier to update parts of your app without breaking the others.
     1. **Model** - The model represents your business objects (whatever your app is about). The model should also include any business rules related to the data.
     2. **View** - The view is the graphical representation of the model expressed in XAML. For example, if your model is a *student* the view for that model could be a simple grid showing the student’s name, grade level and student ID, or something more complex, like a map showing the student’s path from home to school. The view also receives any user input such as taps, clicks or swipes.
     3. **View Model** - The view model acts as the mediator between the view and the view model. This intermediary alleviates the need for models to know how their data will display, which allows them to be loosely coupled. The view model may contain references to one or more instances of one or more models and can interact with these models directly via their properties, methods and events. Interactions with the view are typically handled through data bindings. These bindings allow for two-way communication between the view and the view model to update values on the view to and from the model. Bindings are the only method through which views and view models should interact; there should never be a direct reference to a view in a view model.

View

Presentation (.xaml)

View Model

Presentation Logic (.cs)

Model

Data and Business Logic (.cs)

Data Bindings

Commands

Methods and Properties

Indirect Data Bindings thru VM

Indirect Data Bindings thru VM

Event Handlers

Raise Event / Event Listener

Raise Event / Event Listener

* + 1. **Figure 7**
    2. *MVVM*
  1. Note: To find out more about MVVM you can find a detailed walkthrough of it [here](https://msdn.microsoft.com/en-us/magazine/dd419663.aspx). If you prefer videos, there is a good tutorial [here](http://channel9.msdn.com/Shows/Visual-Studio-Toolbox/Getting-Started-with-MVVM).
  2. Now that we’ve covered the basics of MVVM, let’s discuss how they relate to our application. The **MVVM** folder contains some base classes that help implement the MVVM pattern. The **BindableBase** class is the base class for your models. It implements **INotifyPropertyChanged** to enable binding.
  3. The **ViewModelBase.cs** class is the base class for your view models. In addition to supporting binding (which it does this by inheriting from **BindableBase**), it provides some additional properties in the **NavigationService** to make navigation easier.
  4. Lastly, we have the **Command.cs** class. This class implements the **ICommand** interface, which allows you to take a UI event such as a button click and let it perform an action on your ViewModel. You will find this ability is very useful in future labs.
  5. In the **ViewModels** folder, you will find the view model for the **MainPage.xaml** view. Inside the class, you can see how the view model handles navigation events raised by the **NavigationService***.*

Task 4 – Compile and execute our blank project

In this task, we will compile and run our blank project on both the Local Machine.

* 1. Open Visual Studio and open the project you created in Task 2 above (or create a new project from the Blank Template 10).
  2. To prepare to build and run your app, use the Solution Configurations dropdown to choose the **Debug** configuration and use the Solution Platforms dropdown to target **x86 (Figure 8)**. To run the app on the local machine, select **Local Machine** from the drop-down list next to the Start Debugging button on the debugger Standard toolbar.
  3. Note:  is the Start Debugging button.

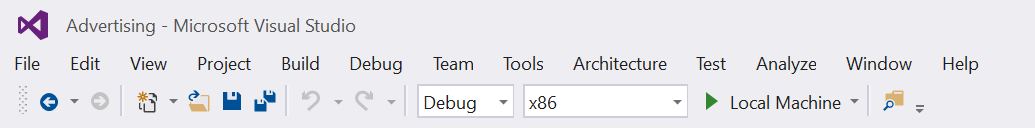


Figure 8

Select options for building and running the app.

* 1. Click the Start Debugging button to build and run the app. Alternatively, you may use the **F5** shortcut to run and build the app once your configurations are set. A window will open that shows the Blank app.
     1. 

1. Figure 9
2. The running Blank template.
3. Return to Visual Studio and stop debugging.
   1. **Note:** Other choices for running the app include the Visual Studio simulator for Windows Apps and Windows Phone emulator.
   2. The Visual Studio simulator for Windows Store apps is a desktop application that simulates a Windows Store app. It lets a developer on a single machine run applications and simulate common touch and rotation events. You can also choose the physical screen size and resolution that you want to emulate, simulate the geo-location co-ordinates of user input events, and simulate network connection properties.
   3. Windows Phone emulator is a desktop application that simulates a Windows Phone. The emulator provides a virtualized environment in which you can debug and test Windows Phone apps on your computer without a physical device. You can simulate common touch and rotation events and choose the physical screen size and resolution that you want to emulate. You can also test many commonly used features, such as location, networking, notifications, sensors, the accelerometer, and the optional SD card.

Summary

* 1. In this lab, we took a look into the template that you will use often in the following labs. We also introduced the MVVM pattern and its uses in our applications.
  2. Now that we have walked through the basics of the template, it’s time to get coding! Let’s dive into some of the features of the Windows 10 platform.