

1: Introduction to WPF

Getting started with WPF using Visual Studio 2019

Windows Presentation Foundation, also known as WPF is a sub-system of .NET family for building a Windows desktop application. WPF uses XAML as its user interface scripting language and C# as its code-behind programming language. In other words, all user interfaces such as windows, pages, controls, menus, and navigations are written in XAML and their code-behind is written using C# language.

This chapter is a swift taste of WPF and its capabilities. In this chapter, you will gain an understanding of how to build your first WPF application using C# and Visual Studio 2019.

Prerequisites

Prerequisites for building WPF applications include the following:

1. Visual Studio 2019 Community or other editions
 2. C# and Object-Oriented Programming concepts
 3. Understanding of XAML language and its syntaxes
-
1. **Visual Studio 2019** Community, developed by Microsoft, is a fully-featured, extensible, free and one of the most popular IDEs for building various software applications for desktop, Web, cloud, libraries, and mobile. You may download Visual Studio 2019 Community from the URL specified in the references section of this chapter or just Google it. Please make sure to download and install Visual Studio 2019 or other editions if you do not have it before you proceed further. Make sure to select the .NET Desktop development option from available packages. Which will install all basic libraries and configurations that you need to develop a WPF application.
 2. **C#** is a programming language developed by Microsoft. If you're not familiar with C#, please find a link for the free C# Programming book in the references section.
Note: We are using C# 9, for all the examples used in the book.
 3. **XAML** stands for Extensible Application Markup Language is a scripting language to build user interfaces for windows client applications. XAML has similar syntaxes as XML.

To learn XAML in-depth, find a reference for an eBook at the end of this chapter.

What is WPF

WPF is a UI framework, designed to create attractive graphical user interfaces for Windows desktop. WPF is a part of Microsoft's .NET ecosystem and gets installed as a part of .NET and Visual Studio. WPF applications are developed using Visual Studio IDE.

WPF is not a replacement for Windows Forms. WPF was developed to replace Windows Forms but some developers and businesses did not want to give up Windows Forms. Today, both Windows Forms and WPF are supported in Visual Studio to build windows client applications. You can also use Windows Forms controls in WPF and vice versa.

WPF is modern, fast, and more dynamic. WPF provides a rich set of 2D and 3D graphics as well as multimedia controls. It is a set of rich controls, advanced layouts, and beautiful styles and themes. WPF also supports advanced features such as navigation controls, upgraded rendering process, and flexible data binding.

Hello, WPF!

Let's create a simple Hello WPF application using Visual Studio 2019 Community.

Create a new project in Visual Studio

Create a new project in Visual Studio by opening Visual Studio and selecting New Project. When you open Visual Studio 2019, you will see on the left side a list of your previous projects. On the right side, you will see various options including **Create a new project**. See Figure 1.

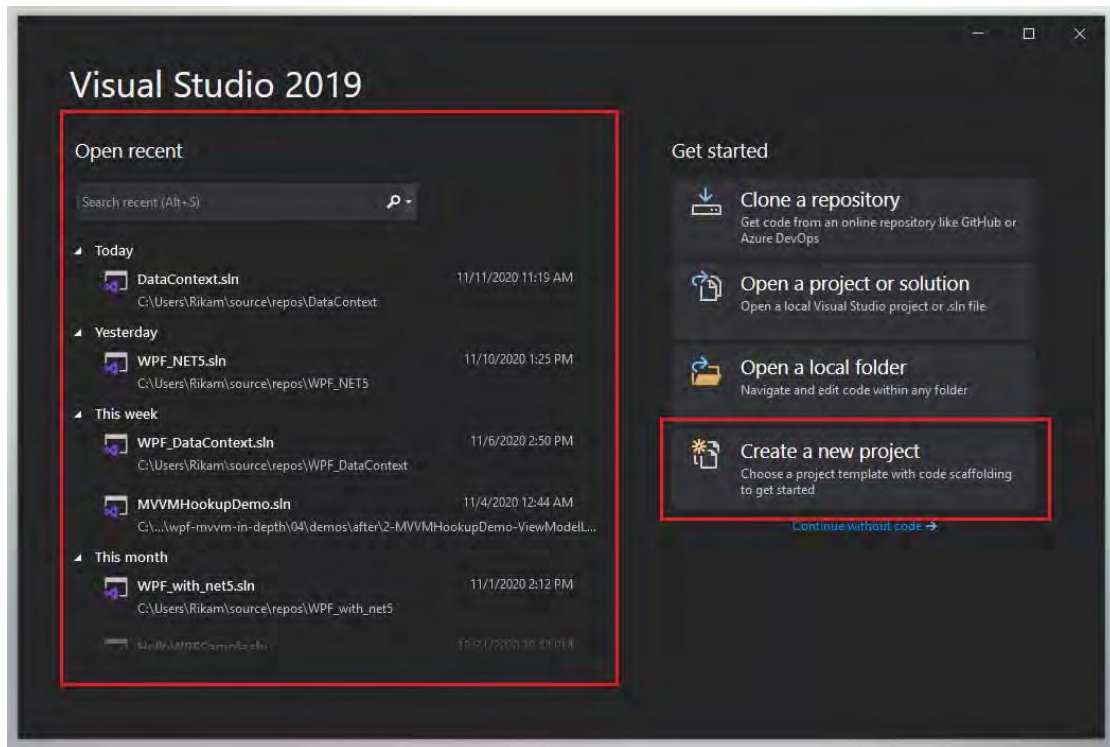


Figure 1

Select Create a new project on the above screen.

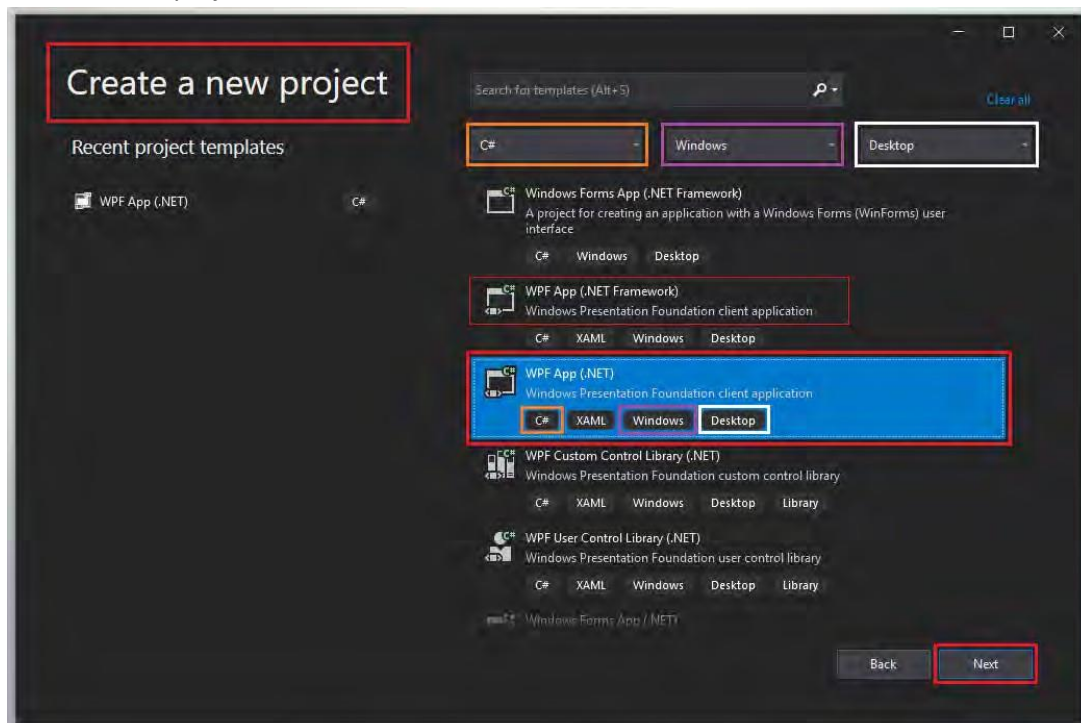


Figure 2

On the next screen, you can filter based on a language, a platform, and a project type.

Select C#, Windows, and Desktop from the dropdowns respectively. See in Figure 2.

In the available WPF templates, you will notice there are two templates for WPF App – WPF App (.NET Framework) and WPF App (.NET).

.NET Framework and .NET are the different versions of the .NET family; .NET Framework is the older so we are ignoring .NET Framework here. We will use .NET only.

Note: .NET 5.0 is the latest version of .NET.

After clicking on a Next Button you will be directed to a configuration screen as shown in Figure 3, It has the following fields.

- In the **Project Name** text box, type the name of your project. We are naming our project **HelloWPFSample**. You can name your project as you desire.
- The **Location** text box lets you browse a folder where you would like to create your new application.
- The **Solution** is a container for one or more projects in Visual Studio, you can even change the solution name by first unchecking the checkbox which says **“Place the solution and project in the same directory”** and which will enable the **Solution Name** text box.

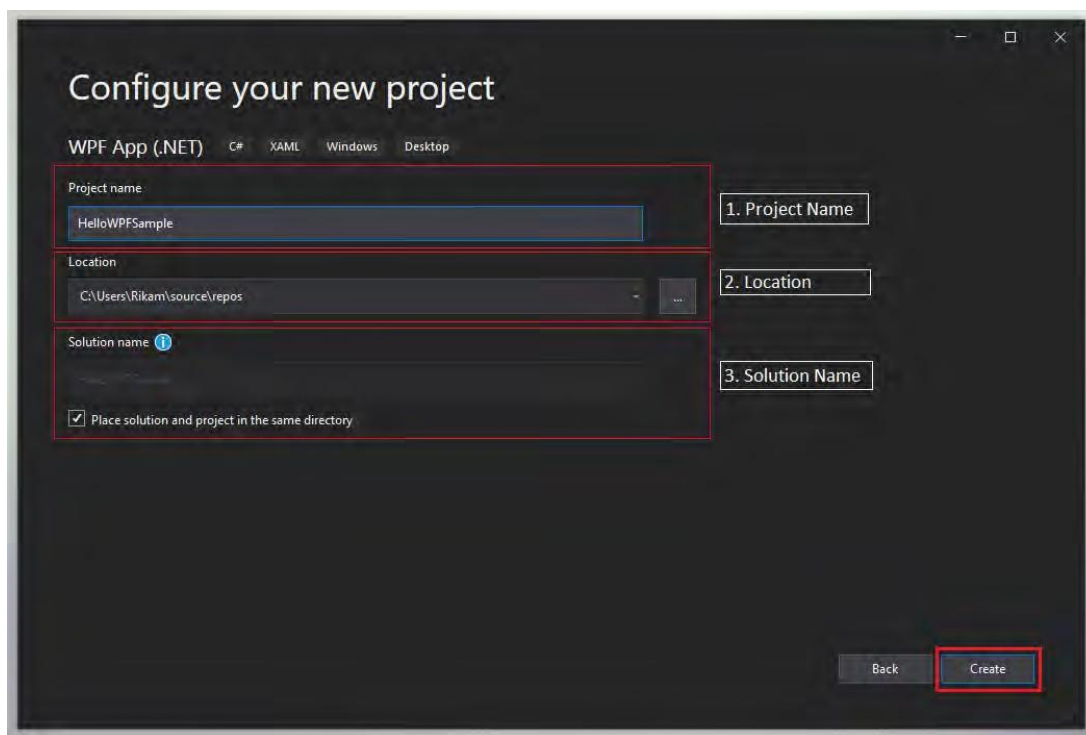


Figure 3

Tip: If you are developing a project in the .NET Framework, then you will have one more field to select the version of .NET Framework. As shown in Figure 4. Select **.NET Framework 4.7.2** or newer versions in the framework version dropdown as per your project's requirements.

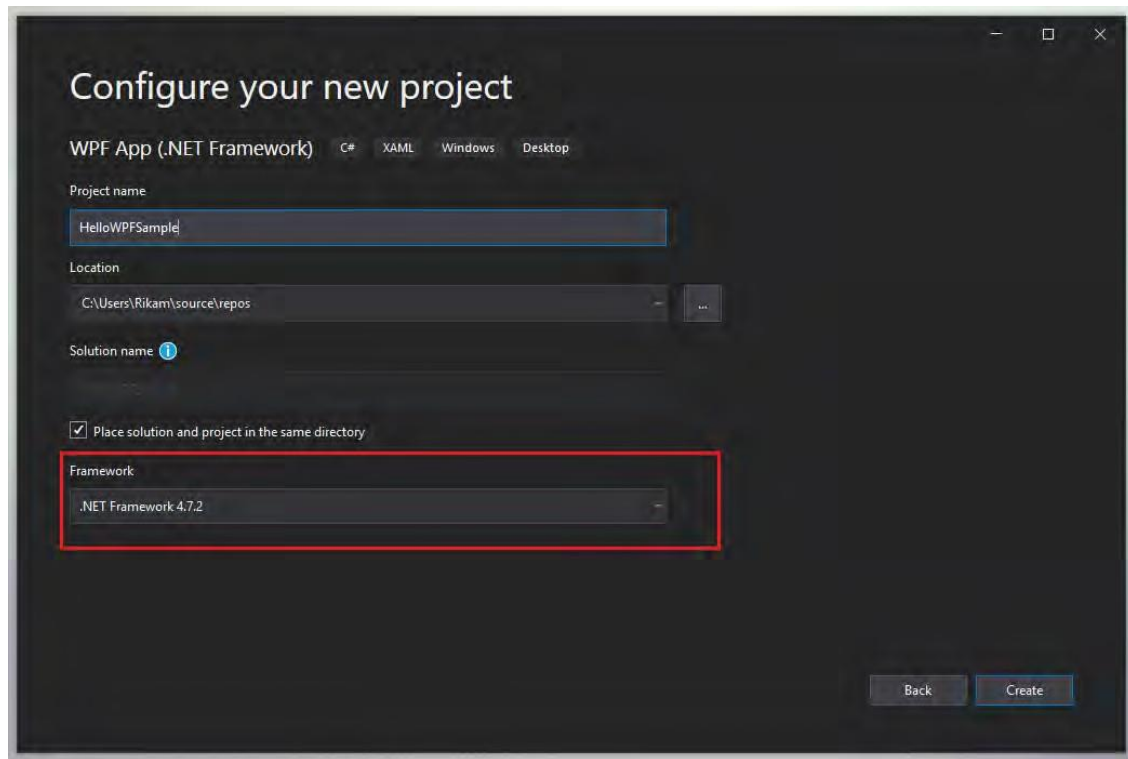


Figure 4

Now click the Create button. This action creates a WPF project with some necessary files already added inside a project. Figure 4 shows the default Visual Studio solution.

Understanding project and files

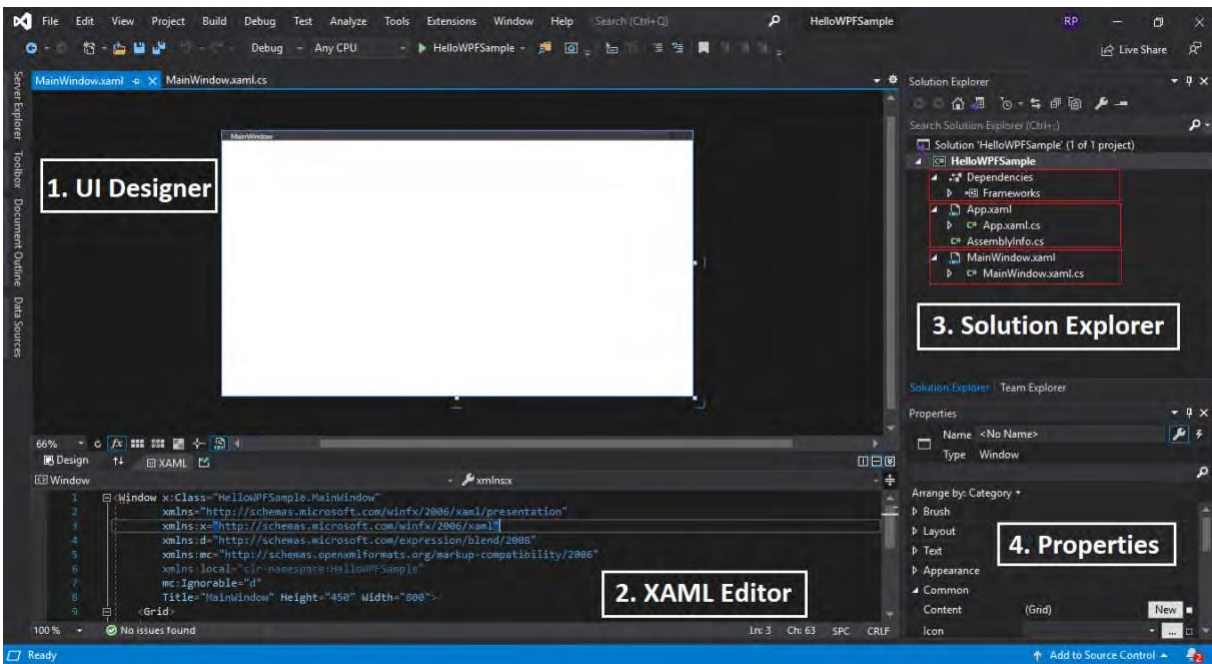


Figure 5

As you can see from Figure 5, there are 4 major sections. Such as 1. UI Designer, 2. XAML Editor, 3. Solution Explorer, and 4. Properties.

1. **The UI Designer** is a window where you can drag and drop elements from a toolbox. We can set layouts such as grid or wrap panel etc. and can arrange other UI elements inside the layout panel. As soon as you drag an element from the toolbox and drop it to the designer, you will notice that the XAML code is automatically written for you by the designer.
2. **XAML Editor**: code for each UI element is present inside a XAML file, using XAML editor we can position, align, style UI elements, add animations etc.
3. **The Solution Explorer** is like a tree view that lists all the projects which are created under the same solution and all the files of those projects. By default, you will see there are 3 nodes created by default for your project – **Dependencies**, **App.xaml**, and **MainWindow.xaml**.
 - **Dependencies** is where we can add references to this project.
 - **The App.xaml** file is an application file which stores resources which are applicable across the entire application. We are going to discuss this in more detail in the following chapters.
 - In this chapter, our focus is on **MainWindow.xaml**. The **MainWindow.xaml** represents the current Window that you see in the designer. As you can see in Figure 4, the XAML editor has a **Window** and a **Grid**. These are UI elements; they have been added to the XAML file by default.

4. **Properties window:** This window has different properties & events related to UI elements. We will learn about Properties window in more detail in coming sub-chapters.

In WPF, each UI element can be represented in two ways.

1. At design-time: By a XAML element, This can be achieved in further 2 ways.
 - a. Drag and drop controls from the Toolbar.
 - b. Manually typing a code for each UI control in XAML editor.
2. At run-time by a code-behind class.

For example, the **XAML editor** represents a window at design-time. If you manually add elements into XAML, they will immediately appear on the window, the developer doesn't have to run the project to see the changes while the code-behind C# class represents a window at run-time, any control added from code-behind won't be available on windows at design time, one has to run the project to see the changes.

1. Designing User Interfaces at compile/design time

a. Drag and drop controls from the Toolbar.

If you click on the **Toolbox** tab in the left side of your screen, you will see a **Toolbox** that lists all the available WPF controls as you can see in Figure 6.

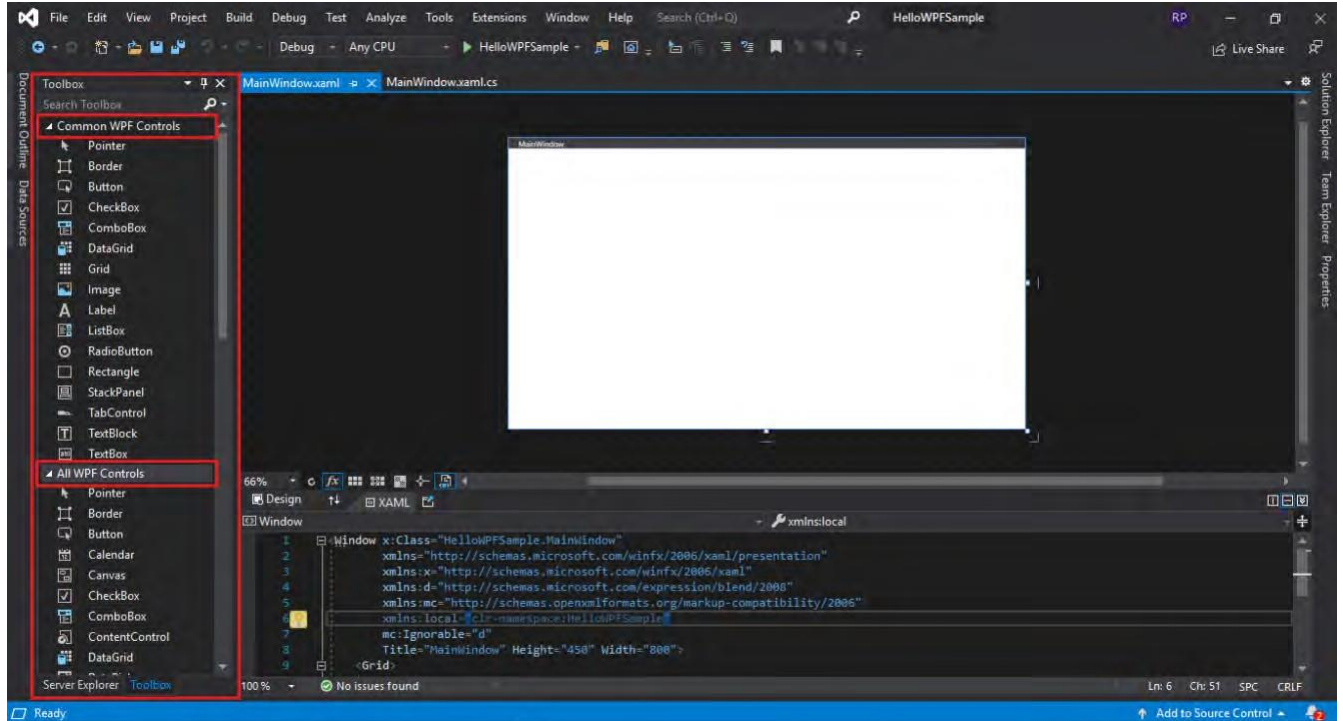


Figure 6

Now to add controls to **MainWindow**, simply drag and drop controls from **Toolbox**. We are going to drag and drop a **Button**, a **TextBox**, and a **ListBox** control from **Toolbox** to **MainWindow**. After moving and placing controls around, my **MainWindow** looks like Figure 7.

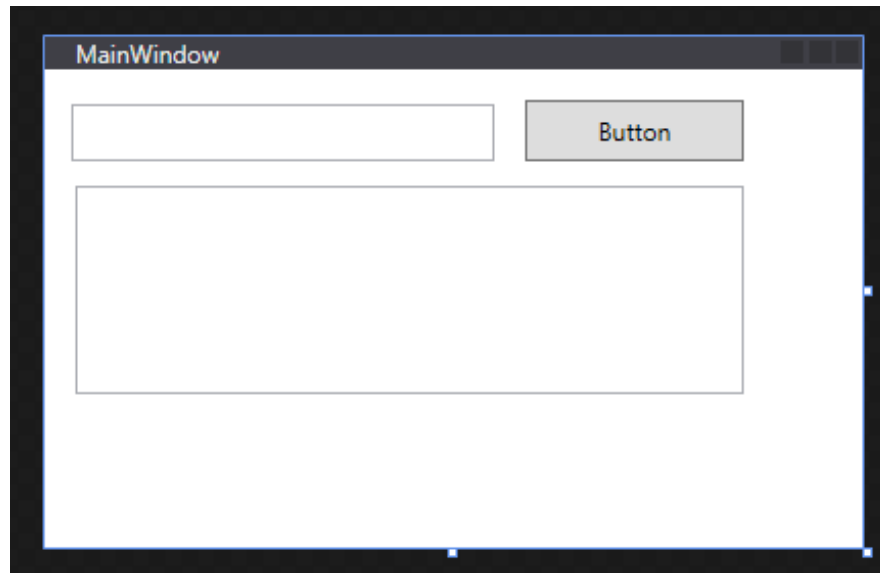


Figure 7

Now one thing you may have noticed when you were dragging and positioning these controls in **MainWindow**. The designer was writing XAML code for you. After placing these controls the same as Figure 7, the XAML code looks like Listing 1.

```
<Window x:Class="HelloWPFSample.MainWindow"
        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"
        xmlns:local="clr-namespace:HelloWPFSample"
        mc:Ignorable="d"
        Height="300" Width="500">
<Grid x:Name="OuterGrid">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <TextBox Name="textBox1"
             Height="30"
             HorizontalAlignment="Left"
             Margin="5 10 0 0"
             Width="207" />
```



```

        <Button Name="button1"
            Content="Button"
            Height="30"
            HorizontalAlignment="Left"
            Margin="0 10 0 0"
            Width="115"
            Grid.Column="1"/>
    <ListBox Name="listBox1"
        Height="102"
        HorizontalAlignment="Left"
        Margin="5 10 0 0"
        Width="360"
        Grid.ColumnSpan="2"
        Grid.Row="1"/>
</Grid>
</Window>

```

Listing 1

As you can see from Listing 1, there is separate XAML element for each of these controls i.e. Button, TextBox, and ListBox. Each element also has Name, Width, Height, Margin, VerticalAlignment, and HorizontalAlignment attributes set for these controls.

Finally, let's just build and run the solution. The output would look like Figure 8.

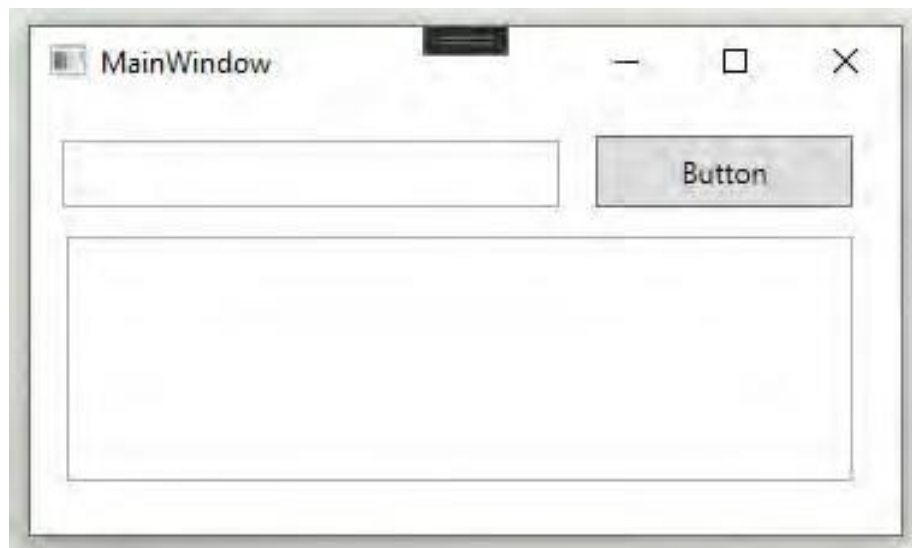


Figure 8

Now here comes the best part. The XAML editor fully supports Intellisense. For example, the moment I started typing <Butt.., you will see Button element is automatically loaded in Intellisense (see Figure 9). So you don't have to rely on the designer completely. You can even manually write your XAML code if you like.

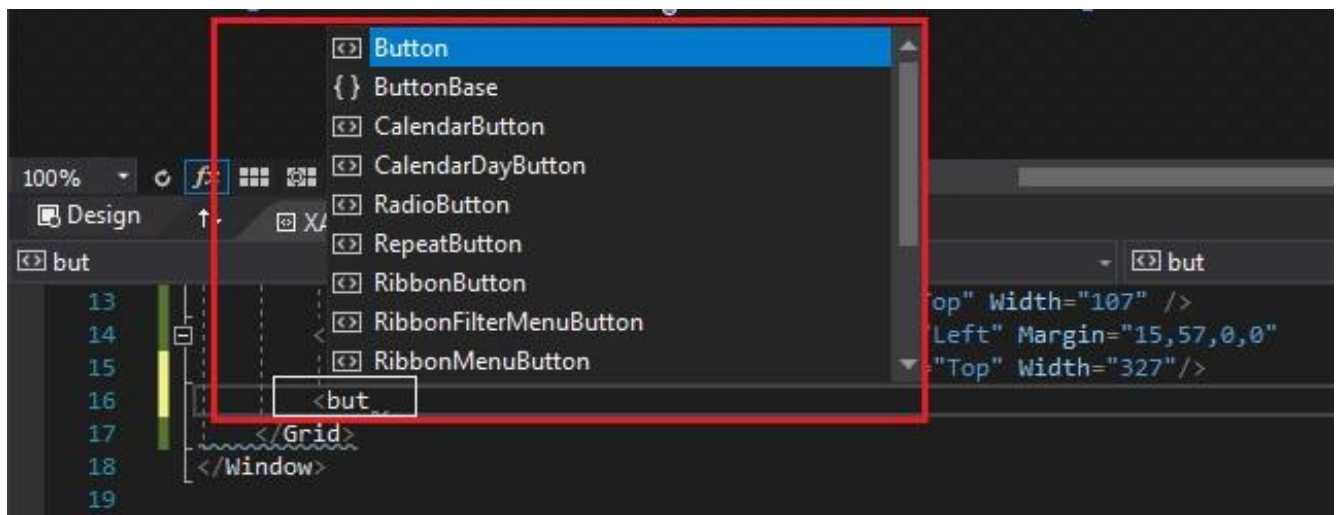


Figure 9

Similarly, we can also edit and change existing XAML manually. You will also notice that as soon you edit XAML, your UI will be updated automatically with your changes.

b. Manually typing a code for each UI control in XAML editor.

Experienced WPF developers leave the habit of dragging and dropping UI controls from Toolbox, and start typing code for each control manually, this gives a better understanding of control and its various properties. You will catch this habit with practice. It is not different from the first approach, the code is the same. The only difference is that you code it with sound knowledge. Next, step is to add properties of these controls at design time. There are two ways to add properties.

1. Using properties window
 2. Manually typing a code for each property in XAML editor
1. Using the properties window.

To see the properties of an element. We need to select that control in the UI designer or a XAML editor and then press the F4 key. This will trigger the properties window to show. As per Figure 10, you can see the red highlighted section are the properties window for a Button.

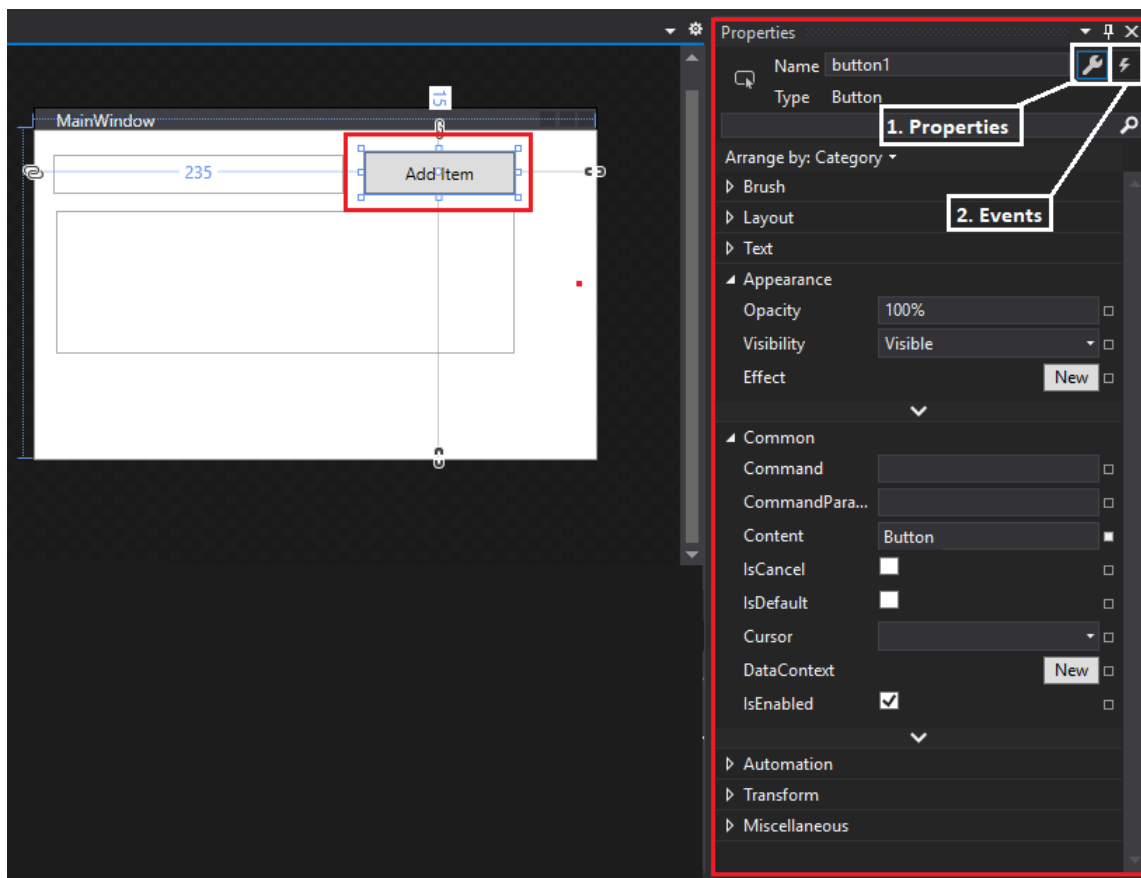


Figure 10

Now there are 2 tabs – **Properties** and **Events**. (Highlighted part in the right-hand corner in Figure 10).

These fields are used to set properties and events for a control respectively. Figure 11 shows how you can change the **Content** property of a **Button** to **Add Item**. And Figure 10 shows how you can add a **Click** event handler to the **Button**.

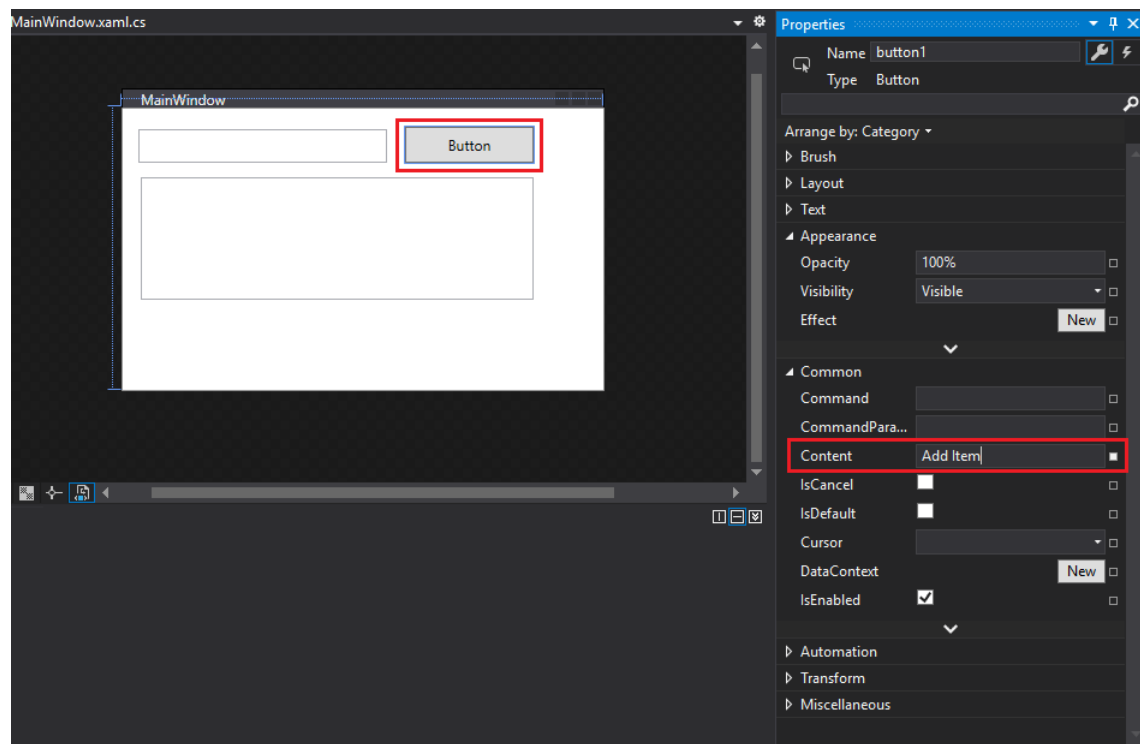


Figure 11

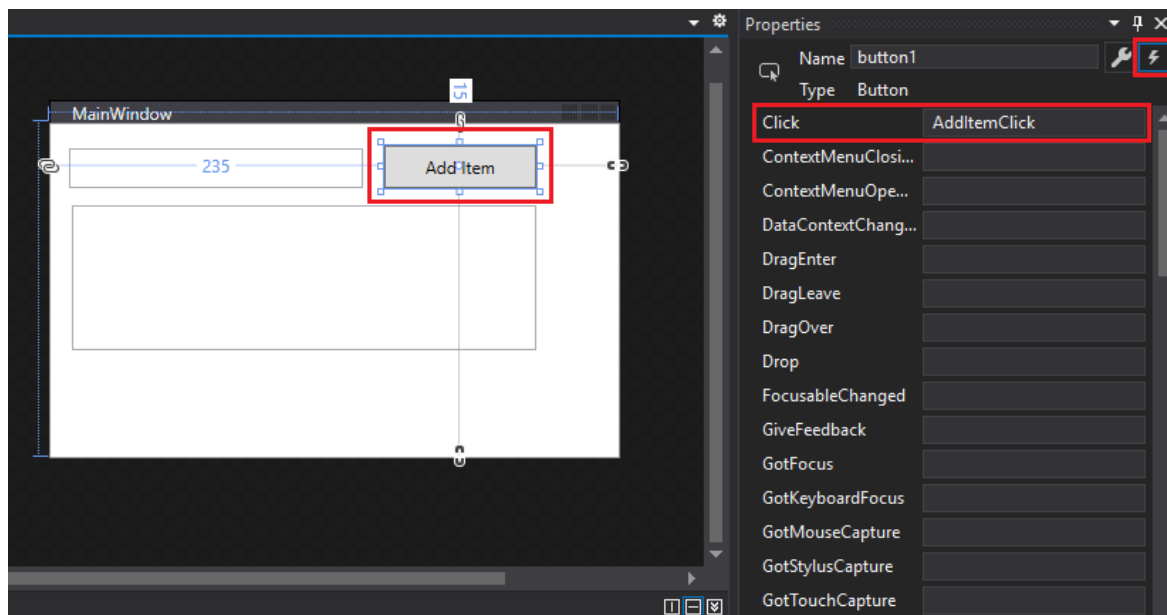


Figure 12

After making these changes in the properties window. If you check the XAML code for Button, you will see the content attribute is changed and a click attribute is added as shown in Listing 2.

```

<Button Name="button1"
        Click="AddItemClick"
        Content="Add Item"
        Height="30"
        HorizontalAlignment="Left"
        Width="107"
        VerticalAlignment="Top"/>

```

Listing 2

Second approach: Manually typing a code for each property in XAML editor

Another way we can set the properties is by coding manually. where you can directly code in XAML for each property. So what if you don't remember the name of each property? no worries we got Intellisense to rescue. It is the same as coding manually for each control. If you know which properties to use you can code them directly and avoid navigating to the Properties window.

We have just created a UI so far. So obviously, our application won't be functional.

Now let's take a look at the *MainWindow.xaml.cs* file that is the code-behind file for *MainWindow.xaml*. The class code is listed in Listing 3 where first some lines are namespace references, followed by a partial class named **MainWindow** who has a default constructor. After that, you will see the *AddItemClick* event handler is added for the Button click event.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace HelloWPFSample
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// partial instance of a MainWindow class
    /// </summary>
    public partial class MainWindow : Window
    {
        /// <summary>
        /// Constructor of a MainWindow
        /// </summary>
        public MainWindow()
        {

```

```

        InitializeComponent();
    }
    /// <summary>
    /// Event-handler for Button click's event
    /// </summary>
    /// <param name="sender"></param>
    /// <param name="e"></param>
    private void AddItemClick(object sender, RoutedEventArgs e)
    {
    }
}

```

Listing 3

Now, let's add an event handler to the Button. This code adds a TextBox's content to the ListBox control when the button is clicked. We do this by adding a single line of code as you can see in Listing 4 where we simply call `ListBox.Items.Add` method and pass `TextBox.Text` as a parameter.

```

private void AddItemClick(object sender, RoutedEventArgs e)
{
    listBox1.Items.Add(textBox1.Text);
}

```

Listing 4

Build and Run

Now let's build and run our application by pressing F5.

If you type any text in the TextBox and click the **Add Item** button, you will see that entered text will be added to the **ListBox** and output will look like Figure 13.

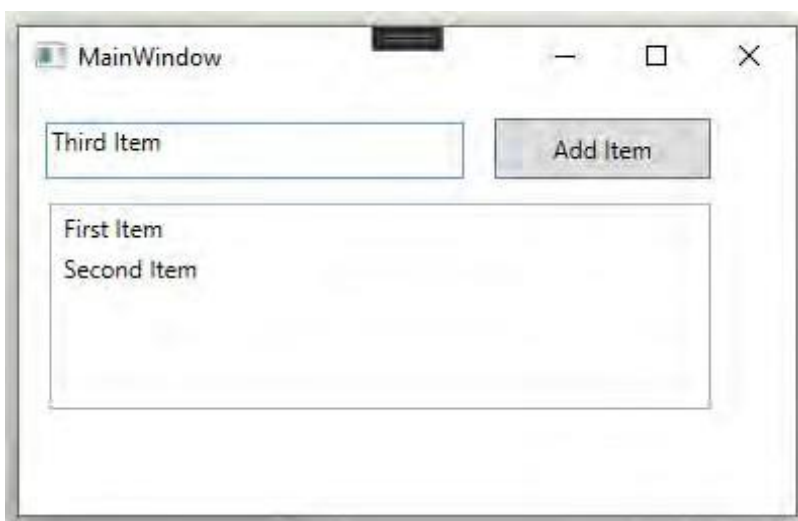


Figure 13

2. Designing User Interfaces at Run-time

As we said earlier in this chapter, we have two choices to design our user interfaces. Using XAML at design-time and using C# classes at run-time. In most of the cases, you will end up mixing both of them. You will end up in designing some interfaces at design-time that would not change no matter what and you would create user interfaces at run-time that will change dynamically.

Now let's create the same UI using WPF code-behind.

First, let's comment out our XAML code that creates TextBox, Button, and ListBox controls. The commented XAML looks like Listing 5. Also, note that we have added the Name attribute of **Grid** to **RootLayout**. This is very important. Because we are going to use this name to add children to the Grid. The code-behind class accesses XAML elements by their name. This confusion will be clear once we understand the code in listing 6.

```
<Window x:Class="HelloWPFSample.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:local="clr-namespace:HelloWPFSample"
  mc:Ignorable="d"
  Title="MainWindow" Height="300" Width="500">
  <Grid x:Name="OuterGrid">
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition/>
      <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <!--<TextBox Name="textBox1"
      Height="30"
      HorizontalAlignment="Left"
      Margin="5 10 0 0"
      Width="207" />
    <Button Name="button1"
      Content="Button"
      Height="30"
      HorizontalAlignment="Left"
      Margin="0 10 0 0"
      Width="115"
      Grid.Column="1"/>
    <ListBox Name="listBox1"
      Height="102"
      HorizontalAlignment="Left"
      Margin="5 10 0 0"
      Width="360"
      Grid.ColumnSpan="2"
      Grid.Row="1"/>-->
  </Grid>
</Window>
```

Listing 5

Basically, we will create a TextBox, Button, and ListBox at run-time and then we need to add these controls to the Grid as children.

First of all, let's add three variables each for TextBox, Button, and ListBox. After that, create a method and name it CreateDynamicUI which is responsible for initializing TextBox, Button, and ListBox controls and setting their properties. As you can see from the code in Listing 6, we can also add event handlers on **button1.Click event** in the code itself by using a RoutedEventHandler and passing a method as a parameter.

```
/// <summary>
/// Interaction logic for MainWindow.xaml
/// </summary>
public partial class MainWindow : Window
{
    #region Constructor
    /// <summary>
    /// Constructor of a MainWindow
    /// </summary>
    public MainWindow()
    {
        InitializeComponent();
    }
    #endregion

    #region UI Properties
    /// <summary>
    /// The TextBox control
    /// </summary>
    protected TextBox textBox1;

    /// <summary>
    /// The Button control
    /// </summary>
    protected Button button1;

    /// <summary>
    /// The ListBox Control
    /// </summary>
    protected ListBox listBox1;
    #endregion

    #region Methods
    /// <summary>
    /// This method creates UI controls and set their properties
    /// </summary>
    private void CreateDynamicUI()
    {
        // Create a TextBox
        textBox1 = new TextBox();
```



```

        textBox1.Name = "textBox1";
        textBox1.Width = 207;
        textBox1.Height = 28;
        textBox1.Margin = new Thickness(13, 17, 0, 0);
        textBox1.HorizontalAlignment =
System.Windows.HorizontalAlignment.Left;
        textBox1.VerticalAlignment =
System.Windows.VerticalAlignment.Top;
        // Add to Grid
        RootLayout.Children.Add(textBox1);

        // Create a Button
        button1 = new Button();
        button1.Name = "textBox1";
        button1.Content = "Add Item";
        button1.Width = 107;
        button1.Height = 30;
        button1.Margin = new Thickness(235, 15, 0, 0);
        button1.HorizontalAlignment =
System.Windows.HorizontalAlignment.Left;
        button1.VerticalAlignment =
System.Windows.VerticalAlignment.Top;
        button1.Click += new RoutedEventHandler(AddItemClick);
        // Add to Grid
        RootLayout.Children.Add(button1);

        // Create a ListBox
        listBox1 = new ListBox();
        listBox1.Name = "listBox1";
        listBox1.Width = 327;
        listBox1.Height = 102;
        listBox1.Margin = new Thickness(15, 57, 0, 0);
        listBox1.HorizontalAlignment =
System.Windows.HorizontalAlignment.Left;
        listBox1.VerticalAlignment =
System.Windows.VerticalAlignment.Top;

        // Add children to the Grid
        RootLayout.Children.Add(listBox1);
    }
#endregion

#region Event-Handlers
    /// <summary>
    /// Event-handler for add button's click event.
    /// </summary>
    /// <param name="sender"></param>
    /// <param name="e"></param>
    private void AddItemClick(object sender, RoutedEventArgs e)
    {
        listBox1.Items.Add(textBox1.Text);
    }
#endregion
}

```

Listing 6

Now, let's call the `CreateDynamicUI` method on `MainWindow`'s constructor as you can see in Listing 7. The output will be the same as Figure 11.

```
public MainWindow()  
{  
    InitializeComponent();  
    CreateDynamicUI();  
}
```

Listing 7

WPF Application Types

WPF allows you to create the following four types of applications.

1. WPF App
2. WPF Browser App
3. WPF User Control Library
4. WPF Custom Control Library

The WPF app template allows you to create a windows standalone application. A WPF Browser App template allows you to create a windows app with browser supports including browser navigation and uses a browser to show the contents. WPF User Control Library is used to create reusable WPF controls and WPF Custom Control Library is used to create third party WPF.

WPF App

Up above we have seen how to create a complete WPF App. When we created the **HelloWPFsample** app in our previous section. A WPF standalone app is a traditional desktop application. The application type is an executable assembly (.exe) that runs on a Windows OS. A user interface in this kind of application is usually a window.

WPF Browser App

A WPF Browser App also known as XBAP application is a traditional desktop application with navigation support by using a browser like navigation buttons. You may build this kind of application to provide back and forward navigation support that will allow users to move back and forth within while navigating through user interfaces. A user interface in this kind of application is usually a page.

Don't let the Browser name in this application type confuses you. This application type still is an executable assembly(.exe) that runs on a local machine. The only difference is how the user interfaces look and how the application keeps track of the history of navigation.

Let's build a WPF Browser App. Create a **New Project** in Visual Studio 2019 and select **WPF Browser App** in the project templates, as per Figure 14.

Note: We are developing this app in .NET Framework because .NET doesn't support WPF Browser App at the time we are writing this book.

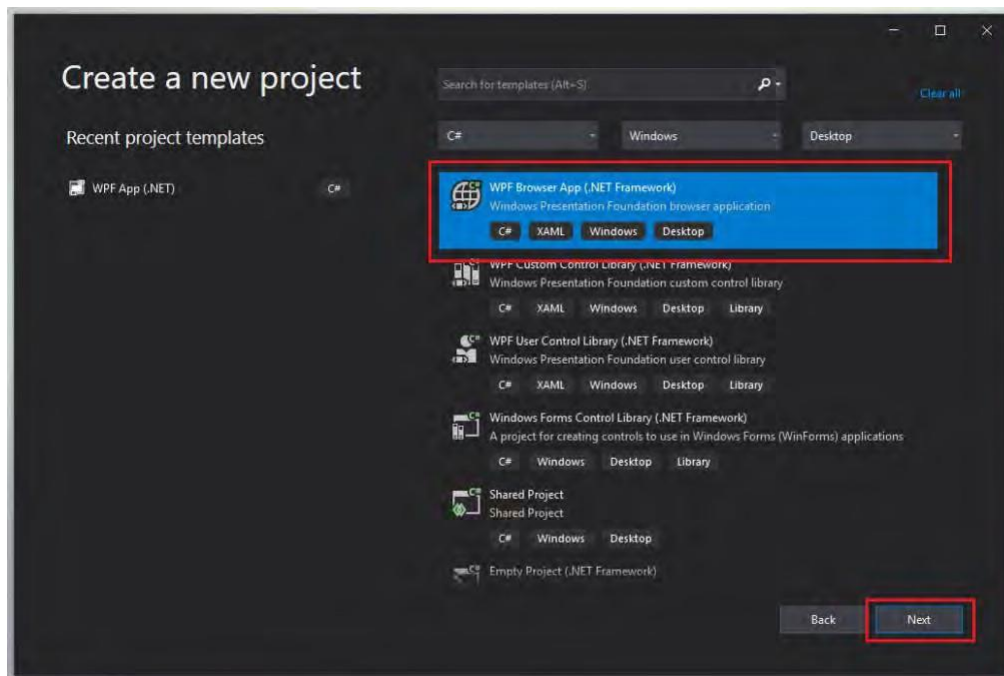


Figure 14

When you build a project. First thing you will notice is that there are no MainWindow.xaml. Instead, you will see Page1.xaml has been added to the project.

```
<Page x:Class="WpfBrowserApp1.Page1"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
      xmlns:local="clr-namespace:WpfBrowserApp1"
      mc:Ignorable="d"
      d:DesignHeight="450" d:DesignWidth="800"
      Title="Page1">
    <Grid>

    </Grid>
</Page>
```

Listing 8

As you can see in Listing 8, unlike Window.XAML, the Page.XAML does not have Width and Height properties. However, there is DesignHeight and DesignWidth of the page. These 2 properties use to maintain this width and height during the design time. When you run the application, it will be a full Browser window anyway.

```
mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="300"
```

Now, let's add a few controls to this application. We have added a Button, a TextBox, and a ListBox control by dragging and dropping them from the toolbox to the page. After adjusting their locations and layout, my page UI looks like Figure 15.

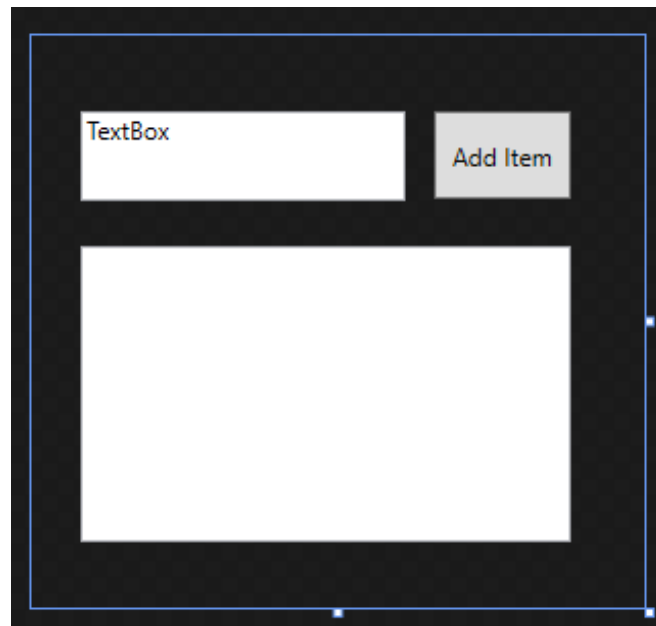


Figure 15

By default, the XAML designer does not specify the name for controls. As we learnt the importance of naming in previous topics i.e. to access UI control's references in code behind, we must set their Name property. Let's set a Name property of Button, TextBox and ListBox controls to Button1, TextBox1, and ListBox1 respectively.

And also change the Content property of **Button** to **Add Item**.

The final XAML code would look like Listing 9.

```
<Grid x:Name="OuterGrid">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <TextBox Name="textBox1"
        Height="30"
        HorizontalAlignment="Left"
        Margin="5 10 0 0"
        Width="207" />
    <Button Name="button1"
        Content="Add Item"
        Height="30"
        Width="100" />
    <ListBox Name="listBox1"
        Height="200"
        Width="500" />
</Grid>
```

```

        HorizontalAlignment="Left"
        Margin="0 10 0 0"
        Width="115"
        Grid.Column="1"/>
<ListBox Name="listBox1"
        Height="102"
        HorizontalAlignment="Left"
        Margin="5 10 0 0"
        Width="360"
        Grid.ColumnSpan="2"
        Grid.Row="1"/>
</Grid>

```

Listing 9

Next, double click on Button to add a click event handler for this button.

Now we need to add the logic to fill ListBox items. Write one line of code that is listed in Listing 10 on the button click event handler. This code simply takes the TextBox.Text property and adds it to a ListBox using ListBox.Items.Add method.

```

private void Button1_Click(object sender, RoutedEventArgs e)
{
    ListBox1.Items.Add(TextBox1.Text);
}

```

Listing 10

When you run the application, you will notice the URL of the Web Browser looks like this:

<file:///C:/Books/Book%20WPF/Chapter%201%20Introducing%20WPF/BrowserAppSample/BrowserAppSample/bin/Debug/BrowserAppSample.xbap>

As you can see in Listing 8, a Page hosts the main window in an XBAP application. When you add some text and click on an Add Item button, you will see the text is being added to the ListBox as shown in Figure 16. You will also notice that this application is running in a Web Browser and has Home, Refresh, Back, Next and other browser functionality.

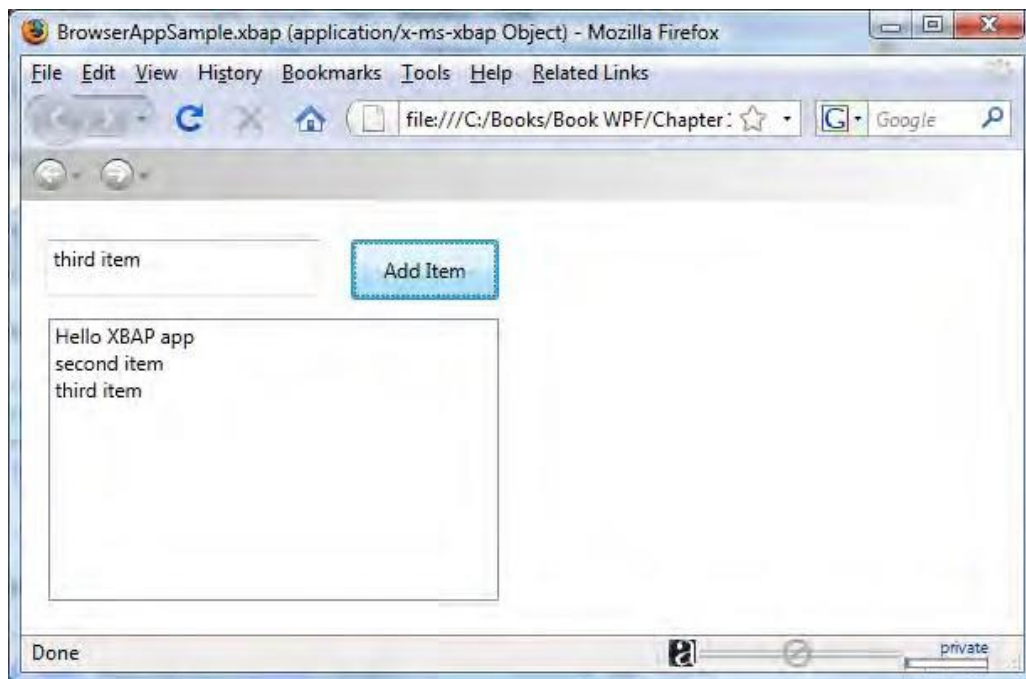


Figure 16

WPF User Control Library

A UserControl is another type of window, the purpose of UserControl is reusability. It compiles into a dynamic linked library (DLL) that can be referenced by other projects. That means the output of UserControl is DLL that can be reused in the same or other projects.

What is DLL?

It stands for **Dynamic-link library**. It is a collection of classes (code and data) that can be shared between Windows applications. Let's say, we need one reusable application to browse files on a system. From the UI perspective, there will be a browse button to select a file from a location and a label to display the selected file name.

One way to do this is to copy and paste the same code every time you need this functionality. Or you can simply build a user control and use that control every time you need that functionality. That means you only once have to develop such an app and can reuse it as per requirements. We can reuse such functionalities within the same application as well as in different applications.

And we can achieve this with the WPF User Control Library. This section has two parts. In the first part, we will build a WPF User Control which will create a DLL file for us. In the second part, we will discuss how to use this DLL in other WPF applications.

The XamlFileBrowser Control

Let's name our application. XamlFileBrowser sounds great. This app allows users to browse files on their machine. Follow the same steps as usual. Create a new project in Visual Studio 2019 and select WPF User Control Library from Installed templates. Click the next button, as shown in Figure 17.

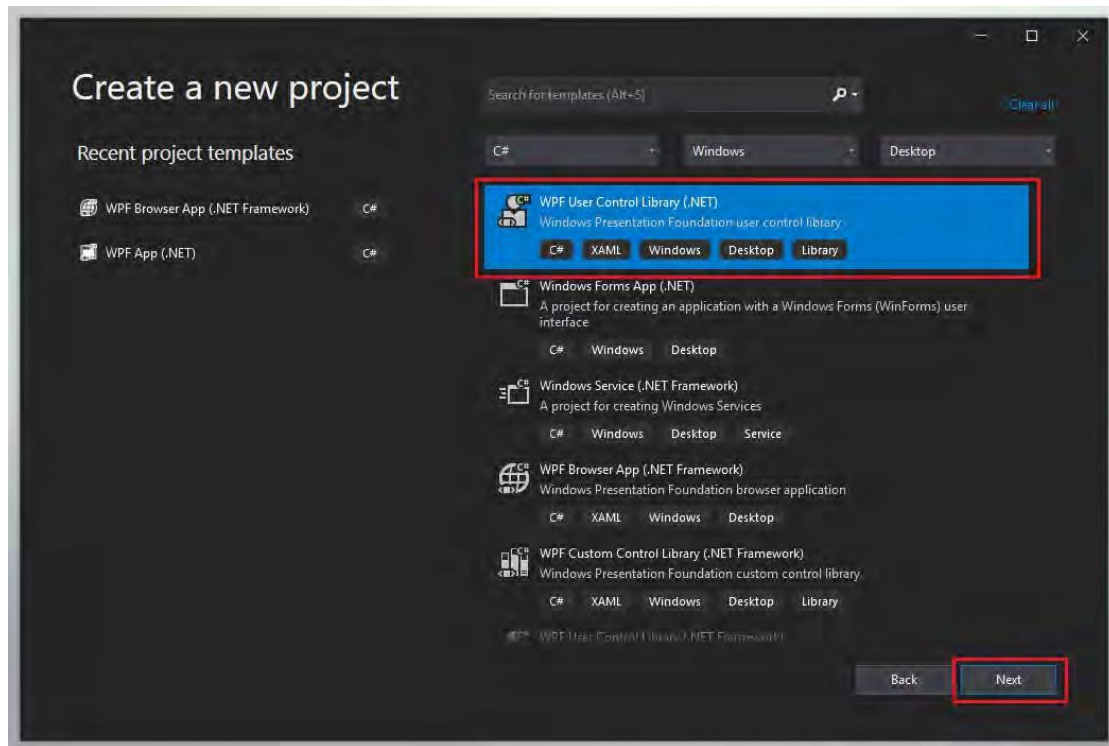


Figure 17

Specify the name for your project. Then click the Create button.

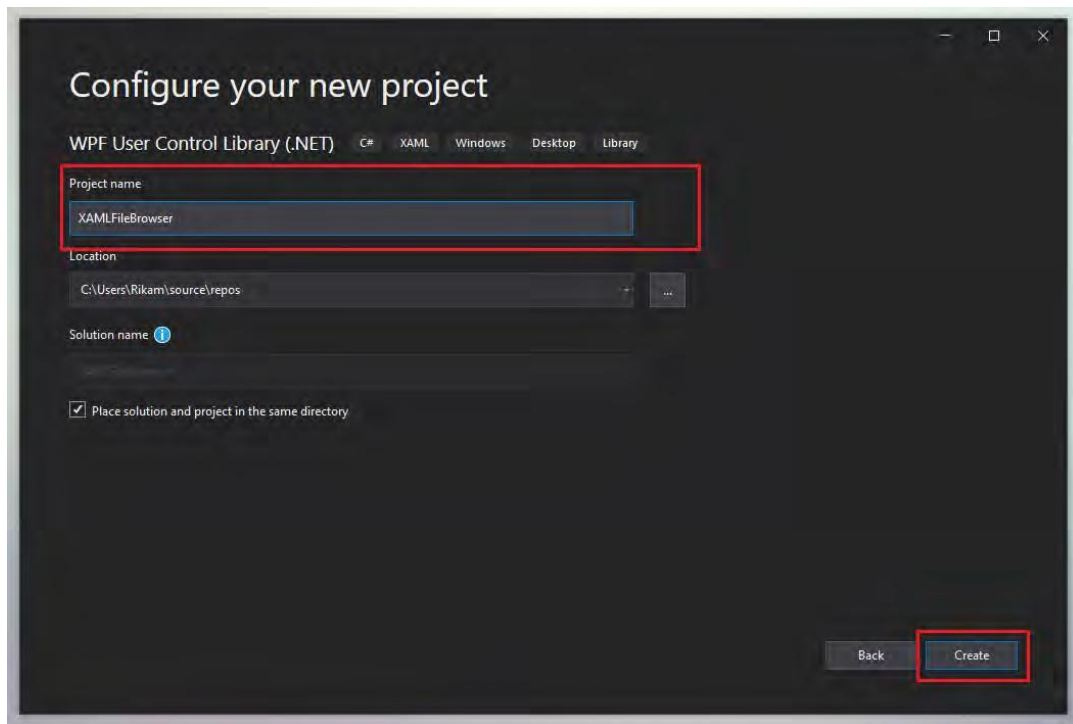


Figure 18

This action creates the WPF User Control Library App.

You will see UserControl1.xaml file has been added to the project. The contents of the control look like Listing 11 where you can see the root element of the control is UserControl instead of a window.

```
<UserControl x:Class="XamlFileBrowser.UserControl1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:local="clr-namespace:XamlFileBrowser"
    mc:Ignorable="d"
    d:DesignHeight="450" d:DesignWidth="800">
    <Grid>

    </Grid>
</UserControl>
```

Listing 11

Designing the XamlFileBrowser Control

Let's add a Button and a TextBox element to this user control by dragging and dropping these controls from Toolbox or you can code manually as well. Then change the names of the TextBox and the Button

controls to *FBCTextBox* and *FBCButton* respectively. After adjusting the size of controls and page, the final design of the user control looks like Figure 19.



Figure 19

You may have noticed from Figure 14; We even have changed the background color of the page to Light-blue by using a **Background** property of user control.

Now double click on Textbox to add TextChanged event-handler for Textbox and double click on a button to add a click event-handler for Button. The final XAML code would look like Listing 12.

```
<UserControl x:Class="XamlFileBrowser.UserControl1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:local="clr-namespace:XamlFileBrowser"
    mc:Ignorable="d"
    d:DesignHeight="35" d:DesignWidth="400" Background="Light Blue">
    <Grid x:Name="OuterGrid">
        <TextBox Name="FBCTextBox"
            Height="50"
            HorizontalAlignment="Left"
            Margin="5,5,0,0"
            Text="TextBox"
            TextChanged="FBCTextBox_TextChanged"
            TextWrapping="Wrap"
            Width="284" />
        <Button Name="FBCButton"
            Click="FBCButton_Click"
            Content="Browse"
            HorizontalAlignment="Left"
            Margin="300,5,0,0"
            Height="50"
            Width="83" />
    </Grid>
</UserControl>
```

Listing 12

Now, let's add logic in the code-behind of user control.

First, Let's add a public property and name it as FileName then add getter and setter which will return and set the text of the FBCTextBox control. The property code is listed in Listing 13:

```
public string FileName
```

```

{
    get => FBCTextBox.Text;
    set => FBCTextBox.Text = value;
}

```

Listing 13

Now, on the FBCButton's click event-handler, we need to add logic to browse files from the system. It can be done with OpenFileDialog control and finally assign FileName property, as per Listing 14.

```

private void FBCButton_Click(object sender, RoutedEventArgs e)
{
    var openFileDialog = new Microsoft.Win32.OpenFileDialog();
    if (openFileDialog.ShowDialog() == true)
        this.FileName = openFileDialog.FileName;
}

```

Listing 14

In the end, we need an event. Let's call it FileNameChanged and map it to the TextChanged event of the TextBox as listed in Listing 15. Notice that we have set e.Handled to true. This indicates that the event-handler has already executed an event, so it wouldn't need to be processed any further.

```

public event EventHandler<EventArgs> FileNameChanged;
private void FBCTextBox_TextChanged(object sender, TextChangedEventArgs e)
{
    e.Handled = true;
    FileNameChanged?.Invoke(this, EventArgs.Empty);
}

```

Listing 15

In this User Control, whenever you type a filename or browse a file from the system, the public FileName property will be updated with the text of the TextBox and we can use this FileName property to display the name of the file.

That's it. Now build the application that will create XamlFileBrowser.dll for us.

Then go to the bin folder of this project to locate this DLL. This is the library we are going to use in our other WPF applications whenever we need a File Browser control.

XAMLFileBrowser User Control's Host Application

Using a WPF User Control in XAML is pretty simple. First, add a reference for a DLL and then update the namespace.

Let's create a WPF application in the same solution.

Right-click on the Solution name in Solution Explorer and select Add >> New Project as you can see in Figure 20.

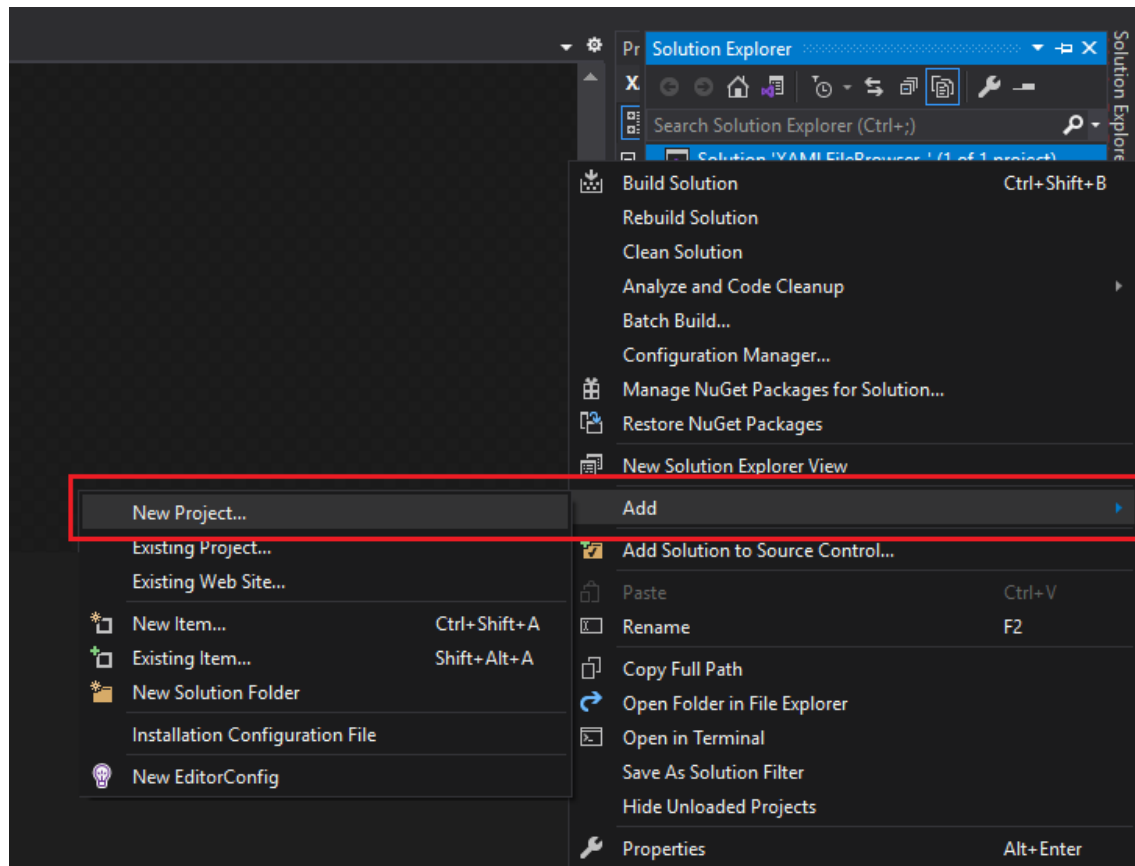


Figure 20

On the next step, select **WPF Application** from the **Project Templates** and give the application a name **XamlFileBrowserHostSample** as you can see in Figure 21.

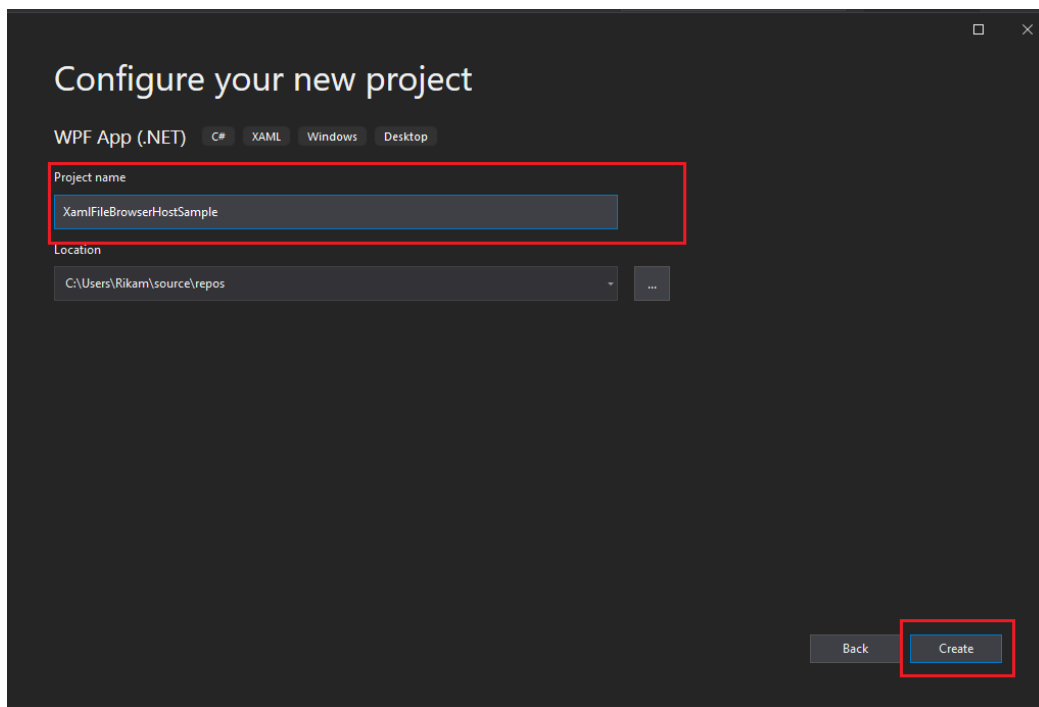


Figure 21

Click on a Create Button to proceed.

First thing you want to do is Right-click on the **XamlFileBrowserHostSample** project and select “**Set as Startup Project**”, as shown in Figure 22. This is important because when the solution has multiple projects, we need to ensure that solution knows which project to execute first when we run the project.

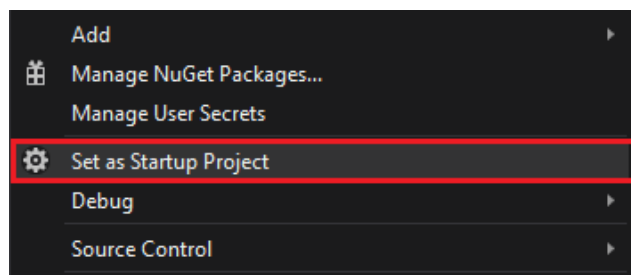


Figure 22

Now we’re going to add a reference to the **XamlFileBrowserHostSample**. Right-click on the “**Dependencies**” and select “**Add Project Reference**” dialogue is shown in Figure 23.

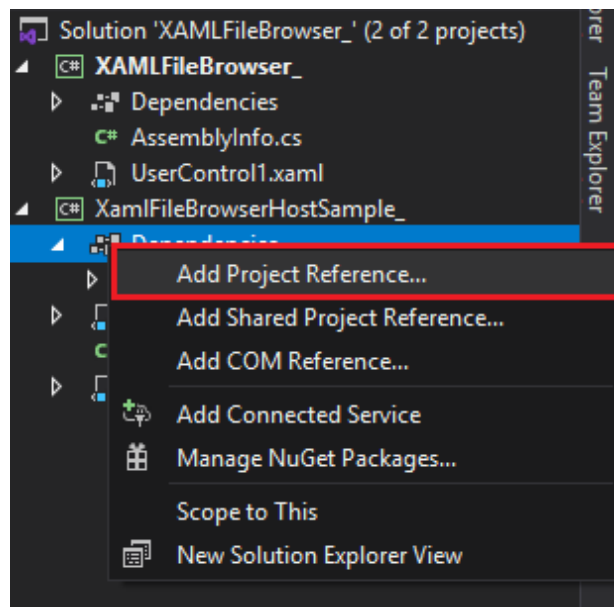


Figure 23

After clicking on **Add Project Reference** Menu, it will open a dialogue box where you will be shown a list of DLLs. simply select a DLL from the list or you can browse the DLL from the system if it is not listed. For our project, we have XamlFileBrowser.dll available on the list. So we need to select the same and click on an OK button. Which will add the DLL to **XamlFileBrowserHostSample** project. Follow instructions as per Figure 24.

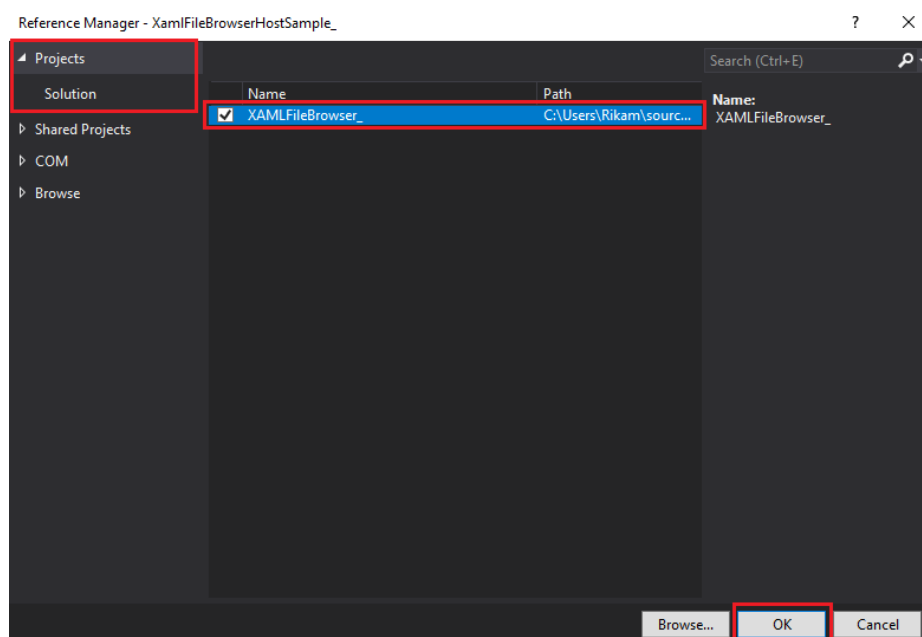


Figure 24

Now **build** the solution by hitting F6.

At last, **XamlFileBrowser** user control will be available to use. In the **MainWindow.XAML** file, type `xmlns:UC=` in the windows root element attributes and at the bottom of the available namespace, you will see *XamlFileBrowser*. Select that from the list, as shown in Figure 25.

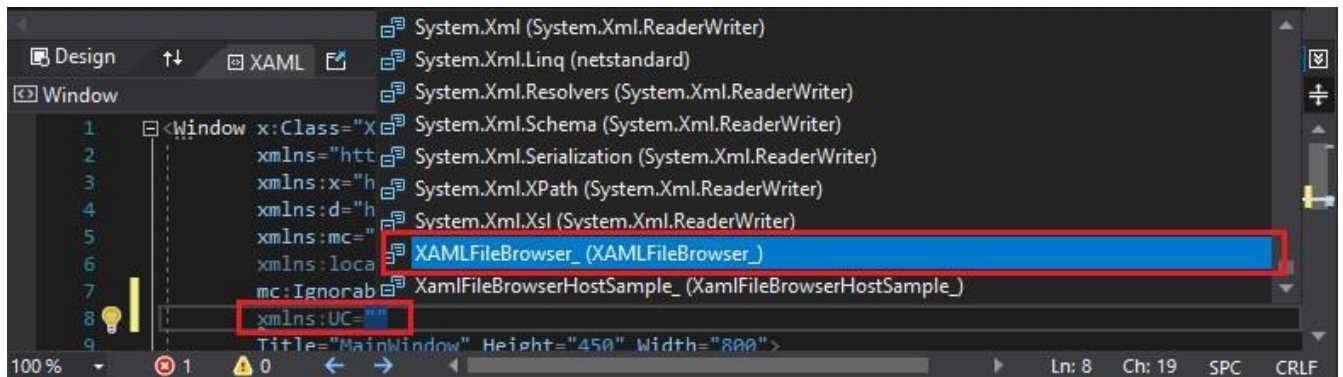


Figure 25

This action will add the following line to the XAML file.

```
xmlns:UC="clr-namespace:XamlFileBrowser;assembly=XamlFileBrowser"
```

Now we can use **namespace UC** to create our user control's instance. As shown in Listing 16 where we have added a User Control, and also sets its Name, Width, Height, and Margin attributes and finally sets its FileNameChanged event-handler. We also need to add a Label control to display the name of the file.

```
<Grid x:Name="OuterGrid">
    <UC:UserControl1 Name="FileBrowserUserControl"
        FileNameChanged="FileBrowserUserControl_FileNameChanged"
        Height="35"
        Margin="26,12,77,264"
        Width="400"/>
    <Label Name="label1"
        Content="Label"
        Height="33"
        HorizontalAlignment="Left"
        Margin="24,67,0,0"
        VerticalAlignment="Top"
        Width="408" />
</Grid>
```

Listing 16

Listing 17 shows how FileNameChanged event-handler used to display FileName in the Label.

```
private void FileBrowserUserControl1_FileNameChanged(object sender, EventArgs e)
{
    label1.Content = FileBrowserUserControl.FileName;
}
```

Listing 17

Now if you Build and Run the application, the output would look like Figure 26. If you click on the Browse button, it opens the *OpenFileDialog* control and lets you browse and select a file. The selected file name is added to the label.



Figure 26

WPF Custom Control Library

WPF allows you to build custom controls. A custom control is developed for third-party developers to customize and then use them in their applications. For example, you could build a cool WPF Chart control and sell that as a control to developers who wish to build similar functionality. Then that chart control can be reused as per developer's requirements and they can customize its style, themes, colors, and other properties.

To create a Custom Control Library, you need to select the WPF Custom Control Library as you can see in Figure 27. We will discuss custom controls in more detail in the coming chapter.

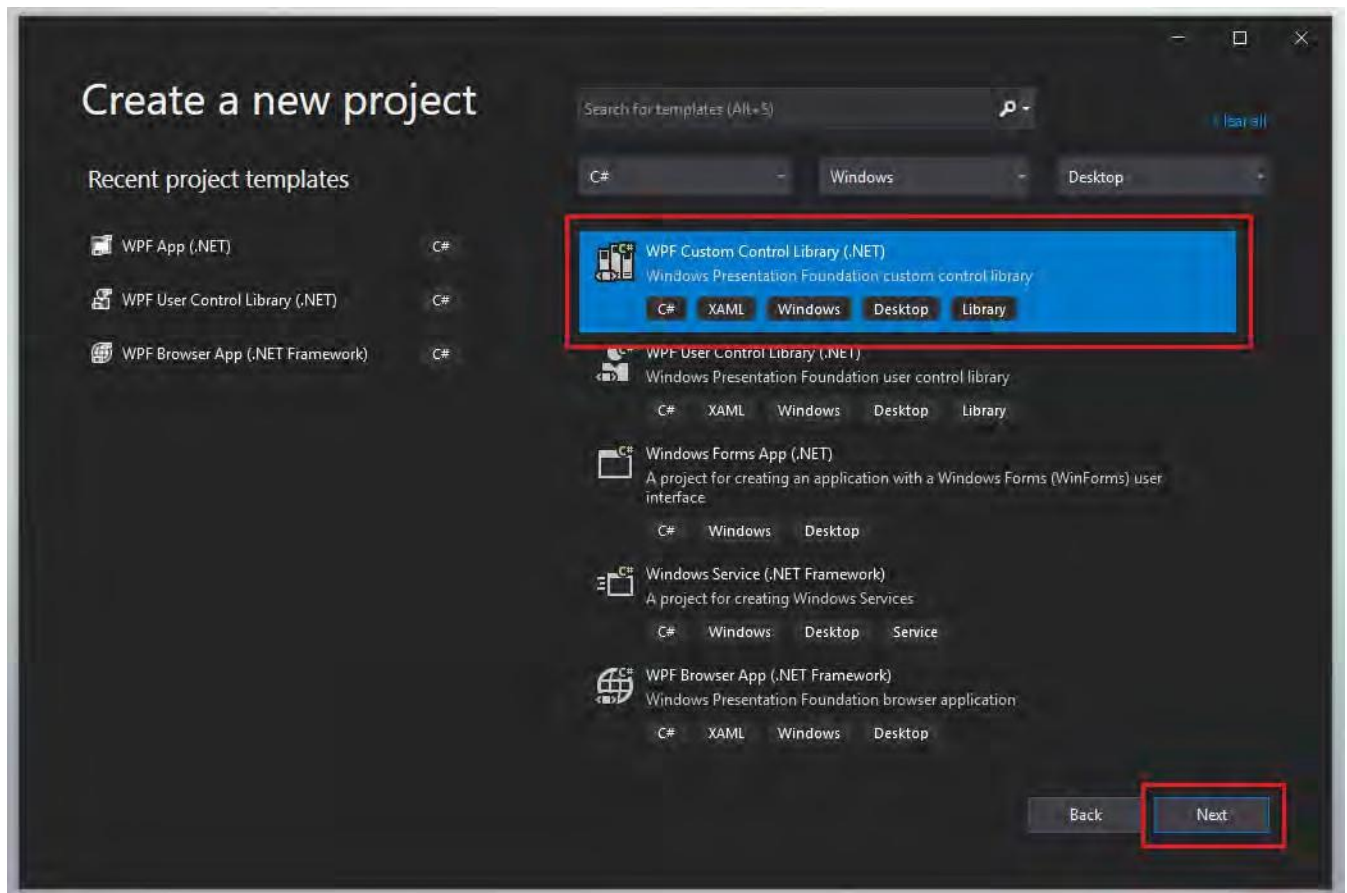


Figure 27

Summary

This chapter is an introduction to WPF. In this chapter, we saw how to create our first WPF application using Visual Studio 2019. We also saw various application types and Visual Studio templates. In the end, we learned how to build simple WPF applications.

In the next chapter, we are going to focus more on the application model supported by WPF and have an in-depth look at various components of a WPF application.

References

Download Visual Studio here: <https://www.visualstudio.com/>

Learn more about the Visual Studio here: <https://visualstudio.microsoft.com/vs/getting-started/>