CUI

Lab Manual

CSC412-Visual Programming

COMSATS UNIVERSITY ISLAMABAD

Department of Computer Science
Islamabad Campus

Lab Contents:

Topics Include: Introduction to Visual Studio; Arrays; Exception handling & Debuggin; Windows Applications; Introduction to WPF ; Markup Extensions; Deploying & Installing Windows Application; Language Integrated Query; Database Connectivity; Threads; Introduction to ASP.NET Core; Entity Framework; Multiple Views & Controllers; and Code First & DB First Approach.

Student Outcomes (SO)

S.#	Description
2	Identify, formulate, research literature, and solve complex computing problems reaching substantiated conclusions using fundamental principles of mathematics, computing sciences, and relevant domain disciplines
3	Design and evaluate solutions for complex computing problems, and design and evaluate systems, components, or processes that meet specified needs with appropriate consideration for public health and safety, cultural, societal, and environmental considerations
4	Create, select, adapt and apply appropriate techniques, resources, and modern computing tools to complex computing activities, with an understanding of the limitations
5	Function effectively as an individual and as a member or leader in diverse teams and in multi-disciplinary settings.
9	Recognize the need, and have the ability, to engage in independent learning for continual development as a computing professional

Intended Learning Outcomes

Sr.#	Description	Blooms Taxonomy Learning Level	SO
CLO -4	Develop platform independent applications.	*Creating*	2-5,9

Lab Assessment Policy

The lab work done by the student is evaluated using Psycho-motor rubrics defined by the course instructor, viva-voce, project work/performance. Marks distribution is as follows:

Assignments	Lab Mid Term Exam	Lab Terminal Exam	Total
25	25	50	100

Note: Midterm and Final term exams must be computer based.

List of Labs

Lab 01
Introduction to Visual Studio

Objective:

This lab will give you an introduction to Visual Studio environment and developing first Console based application using C# language.

Activity Outcomes:

The activities provide hands - on practice with the following topics

- Installing Visual Studio
- Understanding Visual Studio environment
- Creating Console based application using Visual Studio
- Developing first console based application using C# language
- Understanding structure of console based application

Instructor Note:

In this lab, Visual Studio 2022 is used. Therefore, Download and Install Visual Studio 2022. As pre-lab activity, read Chapter 1 from the text book "Microsoft Visual C# Step by Step (Step by Step Developer), Sharp, J., Microsoft Press, 2018.".

1) Useful Concepts

Visual Studio is a complete set of development tools for building ASP.NET Web applications, XML Web Services, desktop applications, and mobile applications. Visual Basic, Visual C#, and Visual C++ all use the same integrated development environment (IDE), which enables tool sharing and eases the creation of mixed-language solutions. In addition, these languages use the functionality of the .NET Framework, which provides access to key technologies that simplify the development of ASP Web applications and XML Web Services.

Console applications are typically designed without a graphical user interface (GUI) and are compiled into an executable file. You interact with a console application by typing instructions at the command prompt.

The Main method is the entry point of a C# application. (Libraries and services do not require a Main method as an entry point.) When the application is started, the Main method is the first method that is invoked. There can only be one entry point in a C# program. If you have more than one class that has a Main method, you must compile your program with the StartupObject compiler option to specify which Main method to use as the entry point.

2) Solved Lab Activites

Sr. No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
Activity 1	*15 mins*	*Low*	*CLO-4*
Activity 2	*15 mins*	*Low*	*CLO-4*

Activity 1:

Install Visual Studio 2022

Step 1 - Make sure your computer is ready for Visual Studio

Before you begin installing Visual Studio:

1. Check the system requirements. These requirements help you know whether your computer supports Visual Studio 2022.
2. Apply the latest Windows updates. These updates ensure that your computer has both the latest security updates and the required system components for Visual Studio.
3. Reboot. The reboot ensures that any pending installs or updates don't hinder your Visual Studio install.
4. Free up space. Remove unneeded files and applications from your system drive by, for example, running the Disk Cleanup app.

Step 2 - Download Visual Studio

Next, download the Visual Studio bootstrapper file.

To do so, select the following button, choose the edition of Visual Studio that you want, and then save to your **Downloads** folder.

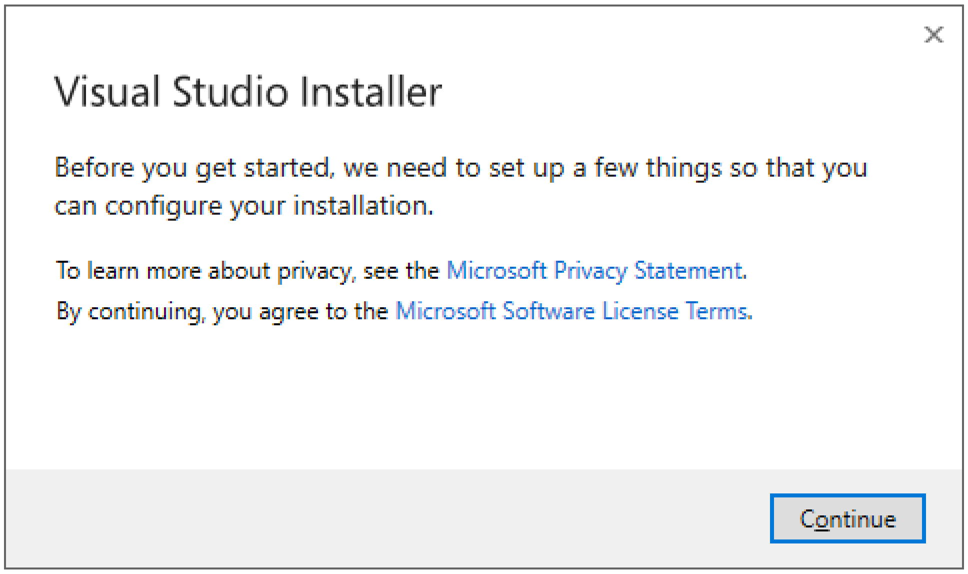
Download Visual Studio

Step 3 - Install the Visual Studio Installer

Run the bootstrapper file to install the Visual Studio Installer. This new lightweight installer includes everything you need to both install and customize Visual Studio.

1. From your **Downloads** folder, double-click the bootstrapper that matches or is similar to the following file:
   - **vs_community.exe** for Visual Studio Community
     If you receive a User Account Control notice, choose **Yes**.

2. We'll ask you to acknowledge the Microsoft License Terms and the Microsoft Privacy Statement. Choose **Continue**.



Visual Studio Installer

Before you get started, we need to set up a few things so that you can configure your installation.

To learn more about privacy, see the Microsoft Privacy Statement.

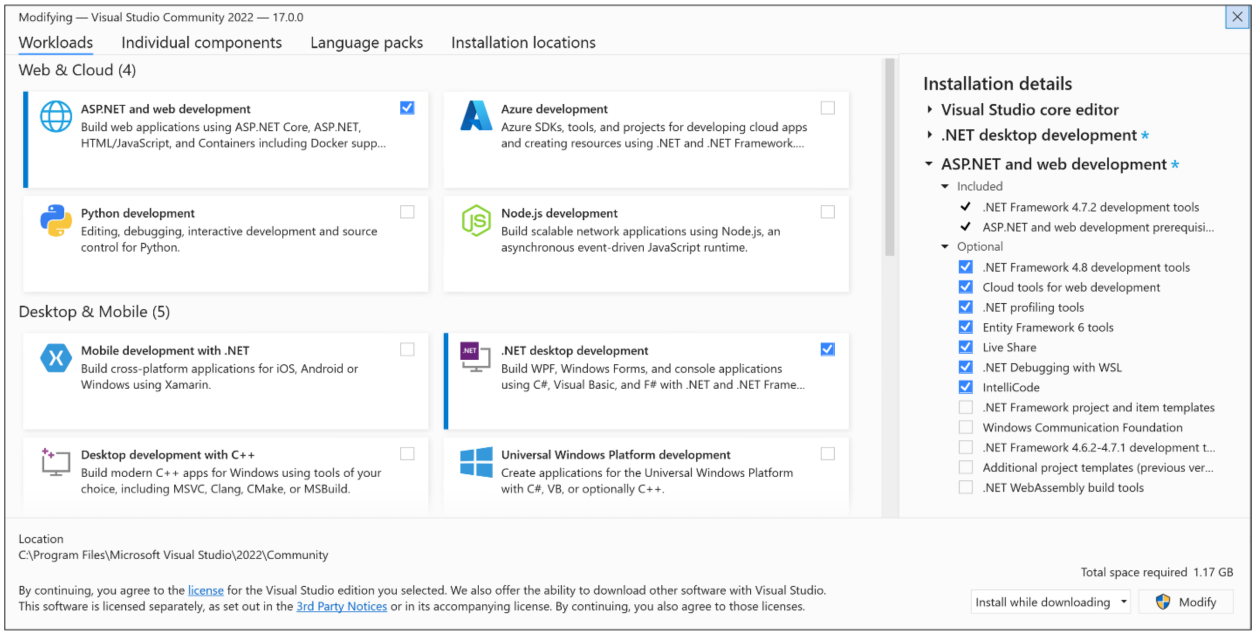
By continuing, you agree to the Microsoft Software License Terms.

Figure 1: Visual Studio Installer

Step 4 - Choose workloads

After the installer is installed, you can use it to customize your installation by selecting the feature sets—or workloads—that you want. Here's how.

1. Select the workload you want in the **Visual Studio Installer**.



Web & Cloud (4)

- [✓] ASP.NET and web development — Build web applications using ASP.NET Core, ASP.NET, HTML/JavaScript, and Containers including Docker supp...
- [ ] Azure development — Azure SDKs, tools, and projects for developing cloud apps and creating resources using .NET and .NET Framework....
- [ ] Python development — Editing, debugging, interactive development and source control for Python.
- [ ] Node.js development — Build scalable network applications using Node.js, an asynchronous event-driven JavaScript runtime.

Desktop & Mobile (5)

- [ ] Mobile development with .NET — Build cross-platform applications for iOS, Android or Windows using Xamarin.
- [✓] .NET desktop development — Build WPF, Windows Forms, and console applications using C#, Visual Basic, and F# with .NET and .NET Frame...
- [ ] Desktop development with C++ — Build modern C++ apps for Windows using tools of your choice, including MSVC, Clang, CMake, or MSBuild.
- [ ] Universal Windows Platform development — Create applications for the Universal Windows Platform with C#, VB, or optionally C++.

Installation details

- Visual Studio core editor
- .NET desktop development *
- ASP.NET and web development *
  - Included
    - ✓ .NET Framework 4.7.2 development tools
    - ✓ ASP.NET and web development prerequisi...
  - Optional
    - [✓] .NET Framework 4.8 development tools
    - [✓] Cloud tools for web development
    - [✓] .NET profiling tools
    - [✓] Entity Framework 6 tools
    - [✓] Live Share
    - [✓] .NET Debugging with WSL
    - [✓] IntelliCode
    - [ ] .NET Framework project and item templates
    - [ ] Windows Communication Foundation
    - [ ] .NET Framework 4.6.2-4.7.1 development t...
    - [ ] Additional project templates (previous ver...
    - [ ] .NET WebAssembly build tools

Location
C:\Program Files\Microsoft Visual Studio\2022\Community

By continuing, you agree to the license for the Visual Studio edition you selected. We also offer the ability to download other software with Visual Studio. This software is licensed separately, as set out in the 3rd Party Notices or in its accompanying license. By continuing, you also agree to those licenses.

Total space required 1.17 GB

Figure 2: Customize your installation

Review the workload summaries to decide which workload supports the features you need. For example, choose the **ASP.NET and web development** workload to edit ASP.NET Web pages with Web Live Preview or build responsive web apps with Blazor, or choose from **Desktop & Mobile** workloads to develop cross-platform apps with C#, or C++ projects that target C++20.

2. After you choose the workload(s) you want, select **Install**.

Next, status screens appear that show the progress of your Visual Studio installation.

Step 5 - Start developing

1. After your Visual Studio installation is complete, select the **Launch** button to get started developing with Visual Studio.
2. On the start window, choose **Create a new project**.
3. In the template search box, enter the type of app you want to create to see a list of available templates. The list of templates depends on the workloads that you chose during installation.

You can also filter your search for a specific programming language by using the **Language** drop-down list. You can filter by using the **Platform** list and the **Project type** list, too.

4. Visual Studio opens your new project, and you're ready to code!

Activity 2:

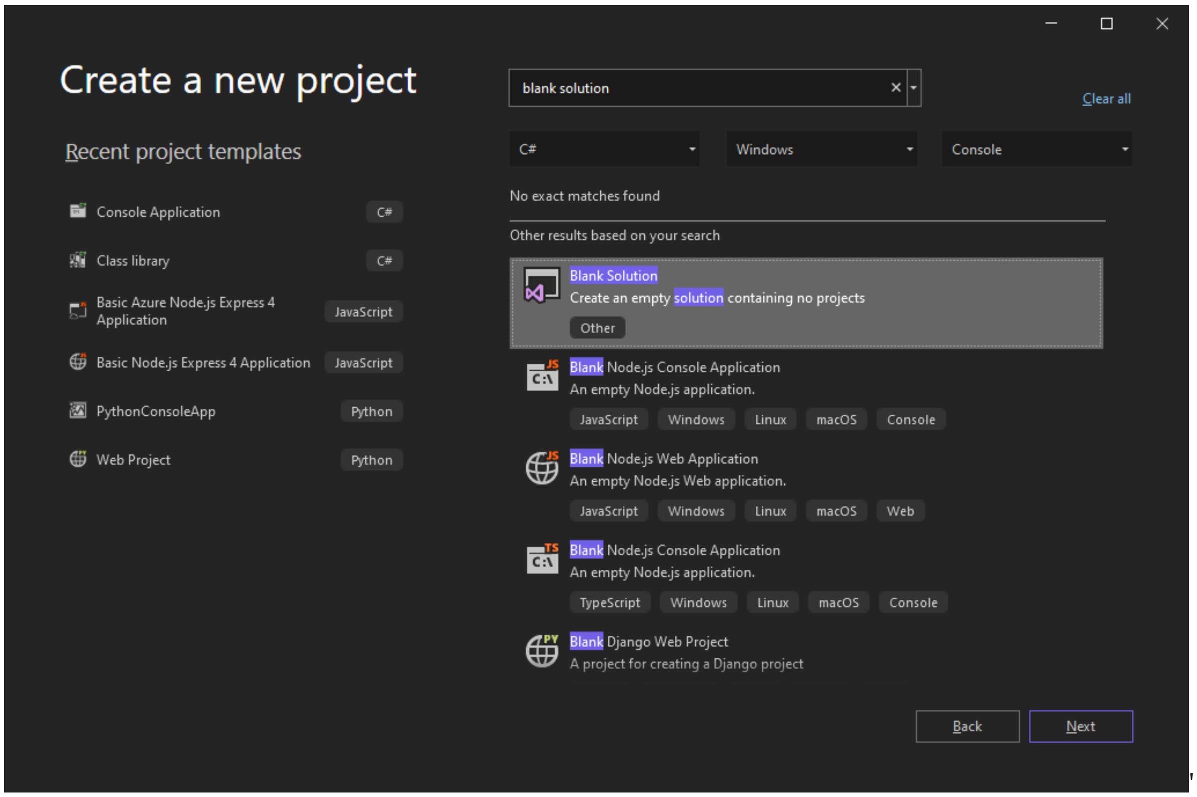
Developing first console based application in Visual Studio using .net framework.

In Visual Studio, a solution isn't an "answer". A solution is simply a container Visual Studio uses to organize one or more related projects. When you open a solution, Visual Studio automatically loads all the projects that the solution contains.

Create a solution

Start your exploration by creating an empty solution. After you get to know Visual Studio, you probably won't create empty solutions very often. When you create a new project, Visual Studio automatically creates a solution for the project unless a solution is already open.

1. Open Visual Studio, and on the start window, select **Create a new project**.
2. On the **Create a new project** page, type *blank solution* into the search box, select the **Blank Solution** template, and then select **Next**.



No exact matches found

Other results based on your search

Blank Solution
Create an empty solution containing no projects
Other

Blank Node.js Console Application
An empty Node.js application.
JavaScript Windows Linux macOS Console

Blank Node.js Web Application
An empty Node.js Web application.
JavaScript Windows Linux macOS Web

Blank Node.js Console Application
An empty Node.js application.
TypeScript Windows Linux macOS Console

Blank Django Web Project
A project for creating a Django project

Figure 3: Create a new project

3. On the **Configure your new project** page, name the solution **QuickSolution**, and then select **Create**.

   The **QuickSolution** solution appears in **Solution Explorer** on the right side of the Visual Studio window. You'll use **Solution Explorer** often to browse the contents of your projects.

Add a project

Now add your first project to the solution. Start with an empty project, and add the items you need.

1. Right-click **Solution 'QuickSolution'** in **Solution Explorer**, and select **Add** > **New Project** from the context menu.
2. On the **Add a new project** page, type *empty* into the search box at the top, and select **C#** under **All languages**.
3. Select the C# **Empty Project (.NET Framework)** template, and then select **Next**.
4. On the **Configure your new project** page, name the project **QuickDate**, and then select **Create**.

The **QuickDate** project appears under the solution in **Solution Explorer**. Currently the project contains a single file called **App.config**.

Add an item to the project

Add a code file to your empty project.

1. From the right-click or context menu of the **QuickDate** project in **Solution Explorer**, select **Add** > **New Item**.

The **Add New Item** dialog box opens.

2. Expand **Visual C# Items**, and then select **Code**. In the middle pane, select the **Class** item template. Under **Name**, type *Calendar*, and then select **Add**.

Visual Studio adds a file named *Calendar.cs* to the project. The *.cs* on the end is the file extension for C# code files. The **Calendar.cs** file appears in the **Solution Explorer** visual project hierarchy, and the file opens in the editor.

3. Replace the contents of the *Calendar.cs* file with the following code:

```
using System;

namespace QuickDate
{
    internal class Calendar
    {
        static void Main(string[] args)
        {
            DateTime now = GetCurrentDate();
            Console.WriteLine($"Today's date is {now}");
            Console.ReadLine();
        }

        internal static DateTime GetCurrentDate()
        {
            return DateTime.Now.Date;
        }
    }
}
```

You don't need to understand everything the code is doing yet. Run the app by pressing **Ctrl+F5**, and see that the app prints today's date to the *console*, or standard output, window. Then, close the console window.

Main Method

- The Main method is the entry point of an executable program; it is where the program control starts and ends.
- Main is declared inside a class or struct. Main must be static and it need not be public. The enclosing class or struct is not required to be static.
- Main can either have a void, int, or, starting with C# 7.1, Task, or Task<int> return type.
- If and only if Main returns a Task or Task<int>, the declaration of Main may include the async modifier. This specifically excludes an async void Main method.
- The Main method can be declared with or without a string[] parameter that contains command-line arguments. When using Visual Studio to create Windows applications, you can add the parameter manually or else use the GetCommandLineArgs() method to obtain the command-line arguments. Parameters are read as zero-indexed command-line arguments. Unlike C and C++, the name of the program is not treated as the first command-line argument in the args array, but it is the first element of the GetCommandLineArgs() method.

The following list shows valid Main signatures:

```
public static void Main() { }
public static int Main() { }
public static void Main(string[] args) { }
public static int Main(string[] args) { }
public static async Task Main() { }
public static async Task<int> Main() { }
public static async Task Main(string[] args) { }
public static async Task<int> Main(string[] args) { }
```

A C# console application must contain a Main method, in which control starts and ends. The Main method is where you create objects and execute other methods.

The Main method is a static (C# Reference) method that resides inside a class or a struct. In the previous example, it resides in a class named Calendar. You can declare the Main method in one of the following ways:
It can return **void**.

```
static void Main()
{
.
.
}
```

It can also return an integer.

```
static int Main()
{      //...

   return 0;

}
```

With either of the return types, it can take arguments.

```
static void Main(string[] args)
{
    //...
}

Or
static int Main(string[] args)
{
    //...
   return 0;
}
```

The parameter of the Main method, args, is a **string** array that contains the command-line arguments used to invoke the program. Unlike in C++, the array does not include the name of the executable (exe) file.

The call to ReadLine at the end of the Main method prevents the console window from closing before you have a chance to read the output when you run your program in debug mode, by pressing F5.

Input and Output

C# programs generally use the input/output services provided by the run-time library of the .NET Framework. The statement Console.WriteLine("…"); uses the WriteLine method. This is one of the output methods of the Console class in the run-time library. It displays its string parameter on the standard output stream followed by a new line. Other Console methods are available for different input and output operations. If you include the using System; directive at the beginning of the program, you can directly use the System classes and methods without fully qualifying them. For example, you can call Console.WriteLine instead of System.Console.WriteLine:

```
using System;
Console.WriteLine("…");
```

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab.

Lab Task 1

Develop a console based application using Visual C# to print your name, registration number and address on console.

Lab 02
Data Types & Arrays

Objective:

This lab will give you an introduction of working with different data types, Arrays (Multi-Dimensional Arrays, Jagged Arrays) and Strings.

Activity Outcomes:

The activities provide hands - on practice with the following topics

- Working with Arrays
- Working with Jagged Arrays
- Using strings

Instructor Note:

As pre-lab activity, read Chapter 2 and 10 from the text book "Microsoft Visual C# Step by Step (Step by Step Developer), Sharp, J., Microsoft Press, 2018.".

1) Useful Concepts

Arrays

You can store multiple variables of the same type in an array data structure. You declare an array by specifying the type of its elements.

```
type[] arrayName;
```

An array has the following properties:

- An array can be Single-Dimensional, Multidimensional or Jagged.
- The number of dimensions and the length of each dimension are established when the array instance is created. These values can't be changed during the lifetime of the instance.
- The default values of numeric array elements are set to zero, and reference elements are set to null.
- A jagged array is an array of arrays, and therefore its elements are reference types and are initialized to null.
- Arrays are zero indexed: an array with n elements is indexed from 0 to n-1.
- Array elements can be of any type, including an array type.
- Array types are reference types derived from the abstract base type Array. Since this type implements IEnumerable and IEnumerable<T>, you can use foreach iteration on all arrays in C#.

Strings

A string is a sequential collection of characters that is used to represent text. A String object is a sequential collection of System.Char objects that represent a string; a System.Char object corresponds to a UTF-16 code unit. The value of the String object is the content of the sequential collection of System.Char objects, and that value is immutable (that is, it is read-only).

2) Solved Lab Activites

Sr. No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
Activity 1	***5 mins***	***Low***	***CLO-4***
Activity 2	***5 mins***	***Low***	***CLO-4***
Activity 3	***5 mins***	***Low***	***CLO-4***
Activity 4	***10 mins***	***Low***	***CLO-4***
Activity 5	***10 mins***	***Low***	***CLO-4***
Activity 6	***10 mins***	***Low***	***CLO-4***
Activity 7	***10 mins***	***Low***	***CLO-4***
Activity 8	***15 mins***	***Low***	***CLO-4***
Activity 9	***15 mins***	***Low***	***CLO-4***

Activity 1:

The Main method is the entry point of a C# console application or windows application. (Libraries and services do not require a Main method as an entry point.). When the application is started, the Main method is the first method that is invoked.

Note: There can only be one entry point in a C# program. If you have more than one class that has a Main method, you must compile your program with the /main compiler option to specify which Main method to use as the entry point This activity is about passing and accessing command line arguments.

Following example prints the number of command line arguments on console.

```
class TestClass
    {
        static void Main(string[] args)
        {
            // Display the number of command line arguments:
            System.Console.WriteLine(args.Length);
        }
    }
```

To pass command line arguments to the program, it can be done while executing the program from command line. Each argument should be separated by a space.

For example, If output file is executed from command line

C:\TestClass\bin\Debug\TestClass.exe arg1 arg2

Then the output of above program would be 2

Command line arguments can also be specified in project properties. To open project properties, go to Project > ProjectName Properties or right click on Project name in

Solution Explorer and click on Properties or press Alt+Enter

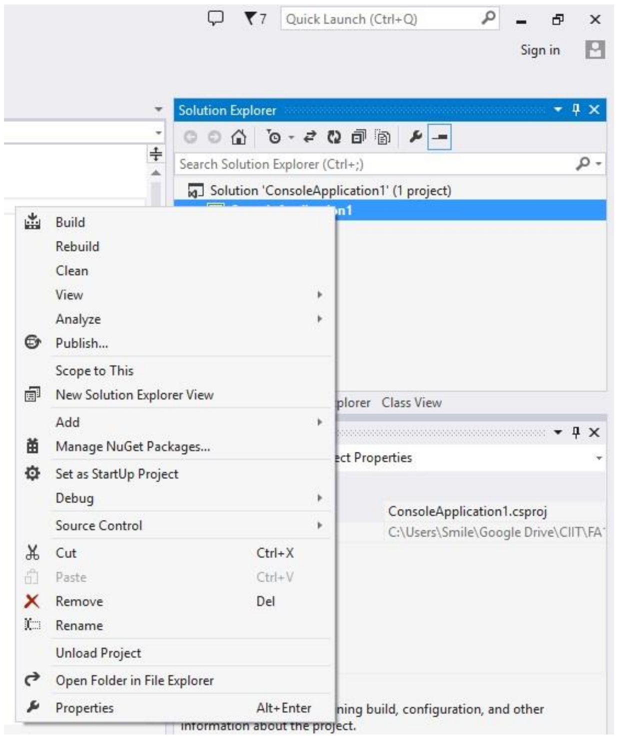
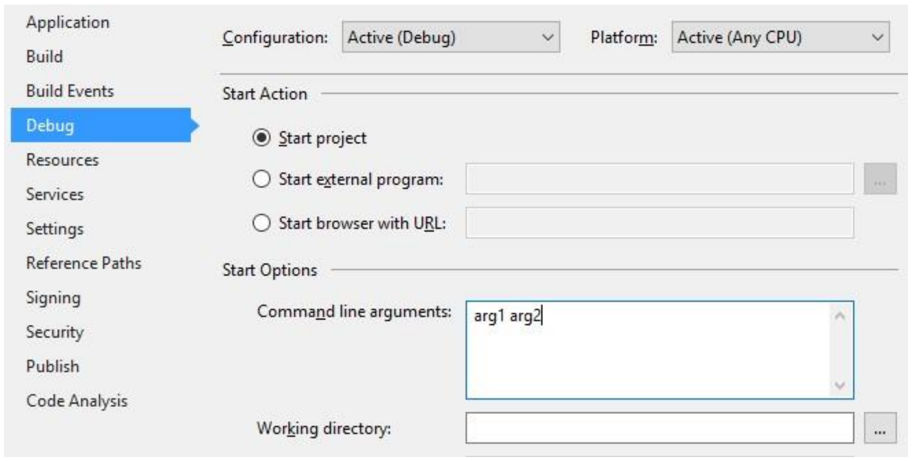


Figure 4: Project properties

In project properties, go to Debug section and specify Command Line Arguments separated with space.



Configuration: Active (Debug)
Platform: Active (Any CPU)

Start Action

- (●) Start project
- ( ) Start external program:
- ( ) Start browser with URL:

Start Options

Command line arguments: arg1 arg2

Working directory:

Figure 5: Debug section

Press F5 to run the program and output will be 2.

Array indexes can be used to access the contents of command line arguments.

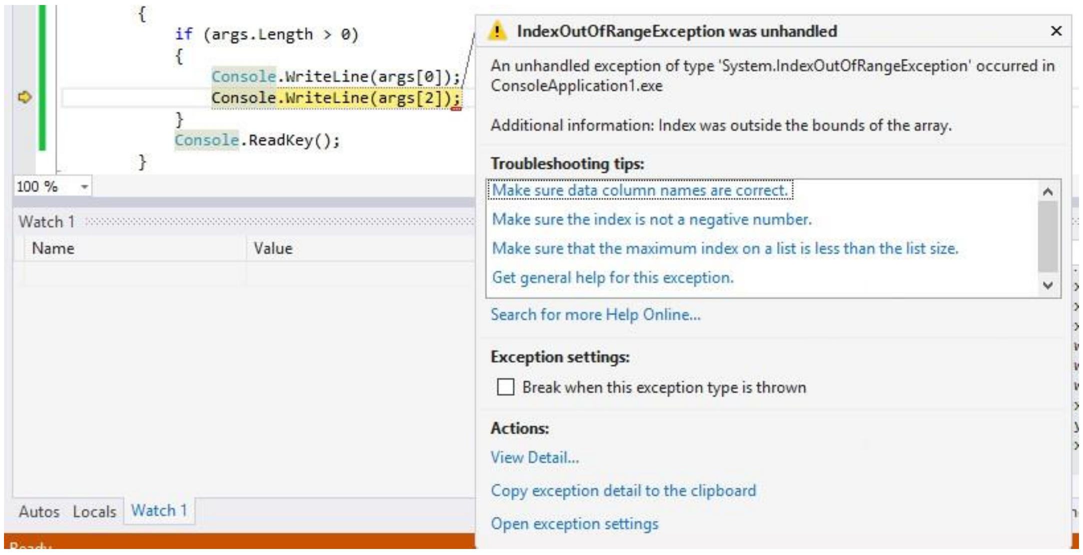
Following example will print the contents of command line arguments on console:

```
    class Program
        {
            static void Main(string[] args)
            {
                if (args.Length > 0)
                {
                    Console.WriteLine(args[0]);
                    Console.WriteLine(args[1]);
                }
                Console.ReadKey();
            }

     }
```

Output of the program is:

arg1 arg2

Note: IndexOutOfRange exception would be thrown if an index does not contain command line argument. For example, in above example, if arg1 or arg2 are not passed as command line argument then the above code would through IndexOutOfRange Exception.



```
        {
            if (args.Length > 0)
            {
                Console.WriteLine(args[0]);
                Console.WriteLine(args[2]);
            }
            Console.ReadKey();
        }
```

IndexOutOfRangeException was unhandled

An unhandled exception of type 'System.IndexOutOfRangeException' occurred in ConsoleApplication1.exe

Additional information: Index was outside the bounds of the array.

Troubleshooting tips:
Make sure data column names are correct.
Make sure the index is not a negative number.
Make sure that the maximum index on a list is less than the list size.
Get general help for this exception.

Search for more Help Online...

Exception settings:
☐ Break when this exception type is thrown

Actions:
View Detail...
Copy exception detail to the clipboard
Open exception settings

Figure 6: IndexOutOfRange Exception

Using foreach loop

Another approach to iterating over the array is to use the foreach statement as shown in this example. The foreach statement can be used to iterate over an array, a .NET Framework collection class, or any class or struct that implements the IEnumerable interface.

```
class CommandLine2
{
    static void Main(string[] args)
    {
      Console.WriteLine("Number of command line parameters = {0}",
 args.Length);

        foreach (string s in args)
        {
            Console.WriteLine(s);
        }
    }
}
```

```
/* Output:
    Number of command line parameters = 3
    John
    Paul
    Mary
*/
```

Using for loop

This example displays the command line arguments passed to a command-line application. The output shown is for the first entry in the table above.

```
   class CommandLine
   {
       static void Main(string[] args)
       {
           // The Length property provides the number of
   array elements
           Console.WriteLine("parameter count = {0}", args.Length);

           for (int i = 0; i < args.Length; i++)
           {
               Console.WriteLine("Arg[{0}] = [{1}]", i, args[i]);
           }
       }
   }
```

```
/* Output (assumes 3 cmd line args):     parameter
count = 3
   Arg[0] = [a]
   Arg[1] = [b]
   Arg[2] = [c]
*/
```

Activity 2:

Create single-dimensional, multidimensional, and jagged arrays.

Solution:

```
class TestArraysClass
   {
       static void Main()
       {
           // Declare a single-dimensional array
           int[] array1 = new int[5];

           // Declare and set array element values
           int[] array2 = new int[] { 1, 3, 5, 7, 9 };
```

```
            // Alternative syntax
            int[] array3 = { 1, 2, 3, 4, 5, 6 };

            // Declare a two dimensional array
            int[,] multiDimensionalArray1 = new int[2, 3];

            // Declare and set array element values
            int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 }
};
            // Declare a jagged rray
            int[][] jaggedArray = new int[6][];

            // Set the values of the first array in the jagged array
structure
            jaggedArray[0] = new int[4] { 1, 2, 3, 4 };
        }
    }
```

Activity 3:

C# also provides the foreach statement. This statement provides a simple, clean way to iterate through the elements of an array or any enumerable collection. The foreach statement processes elements in the order returned by the array or collection type's enumerator, which is usually from the 0th element to the last.

Creates an array called numbers and iterates through it with the foreach statement.

Solution:

Single Dimension Array:

```
int[] numbers = { 4, 5, 6, 1, 2, 3, -2, -1, 0 };

foreach (int i in numbers)
{
    System.Console.Write("{0} ", i);
}
// Output: 4 5 6 1 2 3 -2 -1 0
```

Multi Dimension Array:

```
int[,] numbers2D = new int[3, 2] { { 9, 99 }, { 3, 33 }, { 5, 55 } };
    // Or use the short form:
    // int[,] numbers2D = { { 9, 99 }, { 3, 33 }, { 5, 55 }
```

```
  };

  foreach (int i in numbers2D)
  {
      System.Console.Write("{0} ", i);
  }
  // Output: 9 99 3 33 5 55
```

Activity 4:

Arrays can be passed as arguments to method parameters. Because arrays are reference types, the method can change the value of the elements.

Initialize an array of strings which represents days of week (i.e. Sun, Mon, Tue … Sat) and pass as an argument to a PrintArray. The method displays the elements of the array.

Create methods ChangeArray and ChangeArrayElement change the array elements.

ChangeArray method reverse the array elements so that Sat comes at 0 index.

ChangeArrayElement method changes the first three index of the array so that Sat comes at 0 index, Fri comes at 1st index and Thu comes at 2nd index.

Solution:

```
   class ArrayClass
   {
       static void PrintArray(string[] arr)
       {
           for (int i = 0; i < arr.Length; i++)
           {
               Console.Write(arr[i] + "{0}", i < arr.Length - 1 ? " " :
   "");
           }
           Console.WriteLine();
       }

       static void ChangeArray(string[] arr)
       {
        // The following attempt to reverse the array does not
        // persist when the method returns, because arr is a value
   parameter.
        arr = (arr.Reverse()).ToArray();
           // The following statement displays Sat as the first element
   in the array.
           Console.WriteLine("arr[0] is {0} in ChangeArray.", arr[0]);
   }

       static void ChangeArrayElements(string[] arr)
       {
        // The following assignments change the value of individual
        array elements.
           arr[0] = "Sat";
           arr[1] = "Fri";
           arr[2] = "Thu";
        // The following statement again displays Sat as the first
        element in the array arr, inside the called method.
           Console.WriteLine("arr[0] is {0} in ChangeArrayElements.",
   arr[0]);

   }

       static void Main()
       {
           // Declare and initialize an array.
           string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu",
           "Fri", "Sat" };

           // Pass the array as an argument to PrintArray.
           PrintArray(weekDays);

           // ChangeArray tries to change the array by assigning
   something new to the array in the method.
           ChangeArray(weekDays);

           // Print the array again, to verify that it has not been
   changed.
           Console.WriteLine("Array weekDays after the call to
   ChangeArray:");
           PrintArray(weekDays);
           System.Console.WriteLine();

           // ChangeArrayElements assigns new values to individual
   array elements.
           ChangeArrayElements(weekDays);

           // The changes to individual elements persist after the
```

```
method returns.
        // Print the array, to verify that it has been changed.
        Console.WriteLine("Array weekDays after the call to
ChangeArrayElements:");
        PrintArray(weekDays);
    }
}
// Output:
// Sun Mon Tue Wed Thu Fri Sat
// arr[0] is Sat in ChangeArray.
// Array weekDays after the call to ChangeArray:
// Sun Mon Tue Wed Thu Fri Sat
//
// arr[0] is Sat in ChangeArrayElements.
// Array weekDays after the call to ChangeArrayElements:
// Sat Fri Thu Wed Thu Fri Sat
```

Activity 5:

A jagged array is an array whose elements are arrays. The elements of a jagged array can be of different dimensions and sizes. A jagged array is sometimes called an "array of arrays." The following examples show how to declare, initialize, and access jagged arrays.

The following is a declaration of a single-dimensional array that has three elements, each of which is a single-dimensional array of integers:

```
int[][] jaggedArray = new int[3][];
```

Before you can use jaggedArray, its elements must be initialized. You can initialize the elements like this:

```
jaggedArray[0] = new int[5];
jaggedArray[1] = new int[4];
jaggedArray[2] = new int[2];
```

Each of the elements is a single-dimensional array of integers. The first element is an array of 5 integers, the second is an array of 4 integers, and the third is an array of 2 integers.

It is also possible to use initializers to fill the array elements with values, in which case you do not need the array size. For example:

```
jaggedArray[0] = new int[] { 1, 3, 5, 7, 9 };
 jaggedArray[1] = new int[] { 0, 2, 4, 6 };
 jaggedArray[2] = new int[] { 11, 22 };
```

You can use the following shorthand form. Notice that you cannot omit the new operator from the elements initialization because there is no default initialization for the elements:

```
int[][] jaggedArray3 =
{
  new int[] {1,3,5,7,9},
  new int[] {0,2,4,6},
        new int[] {11,22}
};
```

Activity 6:

Build an array whose elements are themselves arrays (Jagged array). Each one of the array elements has a different size. After creating jagged array, display array elements.

Solution:

```
class ArrayTest
{
    static void Main()
    {
        // Declare the array of two elements:
        int[][] arr = new int[2][];

        // Initialize the elements:
        arr[0] = new int[5] { 1, 3, 5, 7, 9 };
        arr[1] = new int[4] { 2, 4, 6, 8 };

        // Display the array elements:
        for (int i = 0; i < arr.Length; i++)
        {
            Console.Write("Element({0}): ", i);

            for (int j = 0; j < arr[i].Length; j++)
            {
                Console.Write("{0}{1}", arr[i][j], j ==
(arr[i].Length - 1)
? "" : " ");
            }
            Console.WriteLine();
        }
       // Keep the console window open in debug mode.
       Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Element(0): 1 3 5 7 9
    Element(1): 2 4 6 8
*/
```

Activity 7:

You can instantiate a String object in the following ways:

By assigning a string literal to a String variable. This is the most commonly used method for creating a string. The following example uses assignment to create several strings. Note that in C#, because the backslash (\) is an escape character, literal backslashes in a string must be escaped or the entire string must be @-quoted.

Solution:

```
    string string1 = "This is a string created by
    assignment.";
    Console.WriteLine(string1);

    string string2a = "The path is C:\\PublicDocuments\\Report1.doc";
    Console.WriteLine(string2a);

    string string2b = @"The path is C:\PublicDocuments\Report1.doc";
    Console.WriteLine(string2b);

    // The example displays the following output:
    //       This is a string created by assignment.
    //       The path is C:\PublicDocuments\Report1.doc
    //       The path is C:\PublicDocuments\Report1.doc
```

Activity 8:

By retrieving a property or calling a method that returns a string. The following example uses the methods of the String class to extract a substring from a larger string.

Solution:

```
 string sentence = "This sentence has five words.";

 // Extract the second word.
 int startPosition = sentence.IndexOf(" ") + 1;

 string word2 = sentence.Substring(startPosition, sentence.IndexOf(" ",
 startPosition) - startPosition);

 Console.WriteLine("Second word: " + word2);
 // The example displays the following output:
 // Second word: sentence
```

Activity 9:

By calling a formatting method to convert a value or object to its string representation. The following example uses the composite formatting feature to embed the string representation of two objects into a string.

Solution:

```
DateTime dateAndTime = new DateTime(2021, 7, 6, 7, 32, 0);

double temperature = 68.3;

string result = String.Format("At {0:t} on {0:D}, the
temperature was {1:F1} degrees Fahrenheit.",dateAndTime,
temperature);

Console.WriteLine(result);


// The example displays the following output:
// At 7:32 AM on Wednesday, July 06, 2021, the temperature was
68.3 degrees Fahrenheit.
```

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab.

Lab Task 1

Develop a console based application which takes command-line arguments and print them on console.

Test your application on following command line arguments:

Input on Command-line	Array of strings passed to Main
consoleApp.exe a b c	"a" "b" "c"
consoleApp.exe one two	"one" "two"
consoleApp.exe "one two" three	"one two" "three"

Lab Task 2

Console.ReadLine method reads the next line of characters from the standard input stream. Read following article at MSDN and practice using ReadLine method:

https://msdn.microsoft.com/en-us/library/system.console.readline

Lab Task 3

Initialize two-dimensional array of integers and passed to the Print2DArray method. The method displays the elements of the array.

Lab Task 4

Create a method named Merger which takes array of type string as an argument. It combines all elements of string into one string and returns it.

For example:

String which is passed contain following items:

> *string [] s = { "hello ", "and ", "welcome ", "to ", "this ", "demo! " };*
>
> *Merger(s) returns "hello and welcome to this demo!"*

<u>Hint:</u> *Concat method can be used to solve this problem*

Lab Task 5

Write a method which takes string as an argument and extracts all words of length between 4 to 5 and contains vowels in it. Method returns an array of type string containing words which satisfied above criteria.

Lab Task 6

Develop a console-based application which uses all data types given in the chapter 2 of the text book. Read input from user and assigned values to these variables.

Lab 03
Exception handling & Debugging, Properties & Indexers, Delegates

Objective:

The objective of this lab is to learn the students about exception handling & debugging, properties, indexers and delegates in C# with the help of examples and learning tasks. The C# language's exception handling features help you deal with any unexpected or exceptional situations that occur when a program is running.

Activity Outcomes:

The activities provide hands - on practice with the following topics

- Exception Handling & Debugging
- Properties
- Indexers
- Delegates

Instructor Note:

As pre-lab activity, read Chapter 6, 15,16 and 20 from the text book "Microsoft Visual C# Step by Step (Step by Step Developer), Sharp, J., Microsoft Press, 2018.".

1) Useful Concepts

Exception Handling

Exception handling uses the try, catch, and finally keywords to try actions that may not succeed, to handle failures when you decide that it is reasonable to do so, and to clean up resources afterward. Exceptions can be generated by the common language runtime (CLR), by the .NET Framework or any third-party libraries, or by application code. Exceptions are created by using the throw keyword. In many cases, an exception may be thrown not by a method that your code has called directly, but by another method further down in the call stack. When this happens, the CLR will unwind the stack, looking for a method with a catch block for the specific exception type, and it will execute the first such catch block that if finds. If it finds no appropriate catch block anywhere in the call stack, it will terminate the process and display a message to the user.

Debugging

The term debugging can mean a lot of different things, but most literally, it means removing bugs from your code. Now, there are a lot of ways to do this. For example, you might debug by scanning your code looking for typos, or by using a code analyzer. You might debug code by using a performance profiler. Or, you might debug by using a debugger. A debugger is a very specialized developer tool that attaches to your running app and allows you to inspect your code.

Properties

Properties are first class citizens in C#. The language defines syntax that enables developers to write code that accurately expresses their design intent. Properties behave like fields when they are accessed. However, unlike fields, properties are implemented with accessors that define the statements executed when a property is accessed or assigned.

```
public class Person
{
    public string FirstName { get; set; }

    public string LastName { get; set; }

    public string FullName => $"{FirstName} {LastName}";

}
```

Indexers

Indexers are similar to properties. In many ways indexers build on the same language features as properties. Indexers enable indexed properties: properties referenced using one or more arguments. Those arguments provide an index into some collection of values.

Delegates

Delegates provide a late binding mechanism in .NET. Late Binding means that you create an algorithm where the caller also supplies at least one method that implements part of the algorithm. For example, consider sorting a list of stars in an astronomy application. You may choose to sort those stars by their distance from the earth, or the magnitude of the star, or their perceived brightness. In all those cases, the Sort() method does essentially the same thing: arranges the items in the list based on some comparison. The code that compares two stars is different for each of the sort orderings. These kinds of solutions have been used in software for half a century. The C# language delegate concept provides first class language support, and type safety around the concept.

2) Solved Lab Activites

Sr. No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
Activity 1	*5 mins*	*Low*	*CLO-4*
Activity 2	*5 mins*	*Low*	*CLO-4*
Activity 3	*5 mins*	*Low*	*CLO-4*
Activity 4	*10 mins*	*Medium*	*CLO-4*
Activity 5	*10 mins*	*Medium*	*CLO-4*
Activity 6	*15 mins*	*Medium*	*CLO-4*

Activity 1:

In this example, a method tests for division by zero and catches the error. Without the exception handling, this program would terminate with a DivideByZeroException was unhandled error.

Solution:

```
public class ExceptionTest
{
    static double SafeDivision(double x, double y)
    {
        if (y == 0)
            throw new DivideByZeroException();
        return x / y;
    }

    public static void Main()
    {
        // Input for test purposes. Change the values to see
        // exception handling behavior.
        double a = 98, b = 0;
        double result;

        try
        {
            result = SafeDivision(a, b);
            Console.WriteLine("{0} divided by {1} = {2}", a, b,
result);
            }
            catch (DivideByZeroException ex)
            {
                  Console.WriteLine(ex.ToString());
            }
      }
}
```

Activity 2:

Suppose we have a class named TempRecord that represents the temperature in Farenheit as recorded at 10 different times during a 24 hour period. The class contains an array named "temps" of type float to represent the temperatures, and a DateTime that represents the date the temperatures were recorded. By implementing an indexer in this class, clients can access the temperatures in a TempRecord instance as

float temp = tr[4]

instead of as

float temp = tr.temps[4].

The following example shows how to declare a private array field, temps, and an indexer. The indexer enables direct access to the instance tempRecord[i]. The alternative to using the indexer is to declare the array as a public member and access its members, tempRecord.temps[i], directly.

Solution:

```
public class TempRecord
{
      // Array of temperature values
      float[] temps = new float[10]
      {
            56.2F, 56.7F, 56.5F, 56.9F, 58.8F,
            61.3F, 65.9F, 62.1F, 59.2F, 57.5F
      };

      // To enable client code to validate input
      // when accessing your indexer.
      public int Length => temps.Length;

      // Indexer declaration.
      // If index is out of range, the temps array will throw the
exception.
      public float this[int index]
      {
        get => temps[index];
        set => temps[index] = value;
    }
}

class Program
{
    static void Main()
    {
        var tempRecord = new TempRecord();

        // Use the indexer's set accessor
        tempRecord[3] = 58.3F;
        tempRecord[5] = 60.1F;

        // Use the indexer's get accessor
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine($"Element #{i} = {tempRecord[i]}");
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

Output

```
/* Output:
    Element #0 = 56.2
    Element #1 = 56.7
    Element #2 = 56.5
    Element #3 = 58.3
    Element #4 = 58.8
    Element #5 = 60.1
    Element #6 = 65.9
    Element #7 = 62.1
    Element #8 = 59.2
    Element #9 = 57.5
  */
```

Activity 3:

C# doesn't limit the indexer parameter type to integer. For example, it may be useful to use a string with an indexer. Such an indexer might be implemented by searching for the string in the collection, and returning the appropriate value. As accessors can be overloaded, the string and integer versions can coexist.

The following example declares a class that stores the days of the week. A get accessor takes a string, the name of a day, and returns the corresponding integer. For example, "Sunday" returns 0, "Monday" returns 1, and so on.

Solution:

```
using System;

// Using a string as an indexer value
class DayCollection
{
    string[] days = { "Sun", "Mon", "Tues", "Wed", "Thurs", "Fri",
"Sat" };

    // Indexer with only a get accessor with the expression-bodied
definition:
    public int this[string day] => FindDayIndex(day);

    private int FindDayIndex(string day)
    {
        for (int j = 0; j < days.Length; j++)
        {
            if (days[j] == day)
            {
                return j;
            }
        }

        throw new ArgumentOutOfRangeException(
            nameof(day),
            $"Day {day} is not supported.\nDay input must be in the
form \"Sun\", \"Mon\", etc");
    }
}

class Program
{
    static void Main(string[] args)
    {
        var week = new DayCollection();
        Console.WriteLine(week["Fri"]);

        try
```

```
            {
                Console.WriteLine(week["Made-up day"]);
            }
            catch (ArgumentOutOfRangeException e)
            {
                Console.WriteLine($"Not supported input: {e.Message}");
            }
        }
}
```

Output

```
// 5
// Not supported input: Day Made-up day is not supported.
// Day input must be in the form "Sun", "Mon", etc (Parameter 'day')
```

Activity 4:

Indexers allow instances of a class or struct to be indexed just like arrays. The indexed value can be set or retrieved without explicitly specifying a type or instance member. Indexers resemble properties except that their accessors take parameters.

The following example defines a generic class with simple get and set accessor methods to assign and retrieve values. The Program class creates an instance of this class for storing strings.

Solution:

```
using System;

class SampleCollection<T>
{
   // Declare an array to store the data elements.
   private T[] arr = new T[100];

   // Define the indexer to allow client code to use [] notation.
   public T this[int i]
   {
      get { return arr[i]; }
      set { arr[i] = value; }
   }
}

class Program
{
   static void Main()
   {
      var stringCollection = new SampleCollection<string>();
      stringCollection[0] = "Hello, World";
      Console.WriteLine(stringCollection[0]);
   }
}
```

Output

```
//      Hello, World.
```

Activity 5:

Example of single cast delegate.

Solution:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MyFirstConsoleApp
{
    //1. Declaration
    public delegate int MyDelagate(int a, int b);
    //delegates having same signature as method

    public class Example
    {
        //2. Method Definition
        // methods to be assigned and called by delegate
        public int Sum(int a, int b)
        {
            return a + b;
        }

        public int Difference(int a, int b)
        {
            return a - b;
        }
    }
    class Test
    {
        static void Main()
        {
            Example obj = new Example();

            // 3. Instantiation : As a single cast delegate
            MyDelagate sum = new MyDelagate(obj.Sum);
            MyDelagate diff = new MyDelagate(obj.Difference);

            // 4.Invocation
            Console.WriteLine("Sum of two integer is = " + sum(10,
20));
            Console.WriteLine("Difference of two integer is = " +
diff(20, 10));
        }
    }
}
```

Output

```
/*
 Sum of two integer is = 30
 Difference of two integer is = 10
*/
```

Activity 6:

Example of multi-cast delegate.

Solution:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MyFirstConsoleApp
{
    //1. Declaration
    public delegate void MyDelagate(int a, int b);
    public class Example
    {
        // methods to be assigned and called by delegate
        //2. Method definition
        public void Sum(int a, int b)
        {
            Console.WriteLine("Sum of integers is = " + (a + b));
        }
        public void Difference(int a, int b)
   {
    Console.WriteLine("Difference of integer is = " + (a - b));
        }
    }
    class Test
    {
        static void Main()
        {
            Example obj = new Example();
            // 3. Instantiation
            MyDelagate multicastdel = new MyDelagate(obj.Sum);
            multicastdel += new MyDelagate(obj.Difference);

            // 4. Invocation
            multicastdel(50, 20);
        }
    }
}
```

Output

```
/*
 Sum of integers is = 70
 Difference of integer is = 30
*/
```

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab.

Lab Task 1

Write a program which reads following data of 5 students from user:

- *Name (String)*
- *Age (numeric)*
- *GPA (floating point number)*

Implement proper exception handling such that exception will be raised if entered data is not in required format. Error must be shown to user but program should remain in execution.

Lab Task 2

Declare a class named DayCollection that stores the days of the week. Declare a get accessor that takes a string, the name of a day, and returns the corresponding integer.

For example, Sunday will return 0, Monday will return 1, and so on. Create indexer in DayCollection. Following is a demo class.

```
class Program
{
    static void Main(string[] args)
    {
        DayCollection week = new DayCollection();
        System.Console.WriteLine(week["Fri"]);

        // Raises ArgumentOutOfRangeException
        System.Console.WriteLine(week["Made-up Day"]);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to
exit.");
        System.Console.ReadKey();
    }
}
// Output: 5
```

Note: *Employ proper exception handling in the solution.*

Lab Task 3

Implement addition, subtraction, multiplication and division using single cast delegate.

Note: *Employ proper exception handling in the solution.*

Lab Task 4

Implement following functionality using multi cast delegates:

a) Read diameter of a circle from user

b) Calculate radius (diameter / 2) , area (πr^2), and circumference of a circle ($2\pi r$)

Note: *Employ proper exception handling in the solution.*

Lab 04
Introduction to WPF Application Development

Objective:

Purpose of this lab is to familiarize the students with Window Presentation Foundation applications and to introduce them to common controls used in WPF applications.

Activity Outcomes:

The activities provide hands - on practice with the following topics

- Basic windows application and familiarity with xaml
- Common Controls such as Button, Label, ListBox

Instructor Note:

As pre-lab activity, read What is Windows Presentation Foundation - WPF .NET | Microsoft Docs

1) Useful Concepts

Windows Presentation Foundation (WPF)

Welcome to Windows Presentation Foundation (WPF), a UI framework that is resolution-independent and uses a vector-based rendering engine, built to take advantage of modern graphics hardware. WPF provides a comprehensive set of application-development features that include Extensible Application Markup Language (XAML), controls, data binding, layout, 2D and 3D graphics, animation, styles, templates, documents, media, text, and typography. WPF is part of .NET, so you can build applications that incorporate other elements of the .NET API.

Markup and code-behind

WPF lets you develop an application using both *markup* and *code-behind*, an experience with which ASP.NET developers should be familiar. You generally use XAML markup to implement the appearance of an application while using managed programming languages (code-behind) to implement its behavior. This separation of appearance and behavior has the following benefits:

- Development and maintenance costs are reduced because appearance-specific markup isn't tightly coupled with behavior-specific code.
- Development is more efficient because designers can implement an application's appearance simultaneously with developers who are implementing the application's behavior.
- Globalization and localization for WPF applications is simplified.

Markup

XAML is an XML-based markup language that implements an application's appearance declaratively. You typically use it to define windows, dialog boxes, pages, and user controls, and to fill them with controls, shapes, and graphics.

The following example uses XAML to implement the appearance of a window that contains a single button:

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    Title="Window with Button"
    Width="250" Height="100">

  <!-- Add button to window -->
  <Button Name="button">Click Me!</Button>

</Window>
```

Specifically, this XAML defines a window and a button by using the Window and Button elements. Each element is configured with attributes, such as the Window element's Title attribute to specify the window's title-bar text. At run time, WPF converts the elements and attributes that are defined in markup to instances of WPF classes. For example, the Window element is converted to an instance of the Window class whose Title property is the value of the Title attribute.

The following figure shows the user interface (UI) that is defined by the XAML in the previous example:

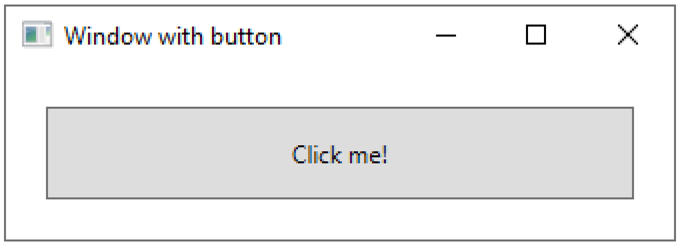


Figure 7: Example output

Code-behind

The main behavior of an application is to implement the functionality that responds to user interactions. For example clicking a menu or button, and calling business logic and data access logic in response. In WPF, this behavior is implemented in code that is associated with markup. This type of code is known as code-behind. The following example shows the updated markup from the previous example and the code-behind:

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SDKSample.AWindow"
    Title="Window with Button"
    Width="250" Height="100">

  <!-- Add button to window -->
  <Button Name="button" Click="button_Click">Click Me!</Button>

</Window>
```

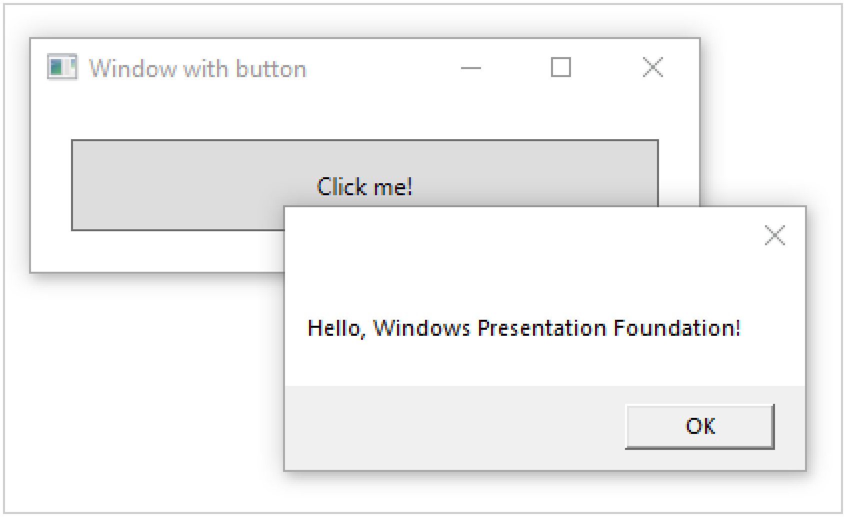
The updated markup defines the xmlns:x namespace and maps it to the schema that adds support for the code-behind types. The x:Class attribute is used to associate a code-behind class to this specific XAML markup. Considering this attribute is declared on the <Window> element, the code-behind class must inherit from the Window class.

```
using System.Windows;

namespace SDKSample
{
    public partial class AWindow : Window
    {
        public AWindow()
        {
            // InitializeComponent call is required to merge the UI
            // that is defined in markup with this class, including
            // setting properties and registering event handlers
            InitializeComponent();
        }

        void button_Click(object sender, RoutedEventArgs e)
        {
            // Show message box when button is clicked.
            MessageBox.Show("Hello, Windows Presentation 
Foundation!");
        }
    }

}
```

InitializeComponent is called from the code-behind class's constructor to merge the UI that is defined in markup with the code-behind class. (InitializeComponent is generated for you when your application is built, which is why you don't need to implement it manually.) The combination of x:Class and InitializeComponent ensure that your implementation is correctly initialized whenever it's created.

Notice that in the markup the <Button> element defined a value of button_click for the Click attribute. With the markup and code-behind initialized and working together, the Click event for the button is automatically mapped to the button_click method. When the button is clicked, the event handler is invoked and a message box is displayed by calling the System.Windows.MessageBox.Show method.

The following figure shows the result when the button is clicked:



Hello, Windows Presentation Foundation!

Figure 8: Dialog box

2) Solved Lab Activites

Sr. No	***Allocated Time***	***Level of Complexity***	***CLO Mapping***
Activity 1	***10 mins***	***Low***	***CLO-4***
Activity 2	***10 mins***	***Low***	***CLO-4***
Activity 3	***10 mins***	***Low***	***CLO-4***
Activity 4	***10 mins***	***Medium***	***CLO-4***
Activity 5	***10 mins***	***Medium***	***CLO-4***

Activity 1:

In this activity, you'll learn how to create a new Windows Presentation Foundation (WPF) app with Visual Studio. Once the initial app has been generated, you'll learn how to add controls and how to handle events. By the end of this activity, you'll have a simple app that adds names to a list box.

Create a WPF app

The first step to creating a new app is opening Visual Studio and generating the app from a template.

1. Open Visual Studio.
2. Select **Create a new project**.

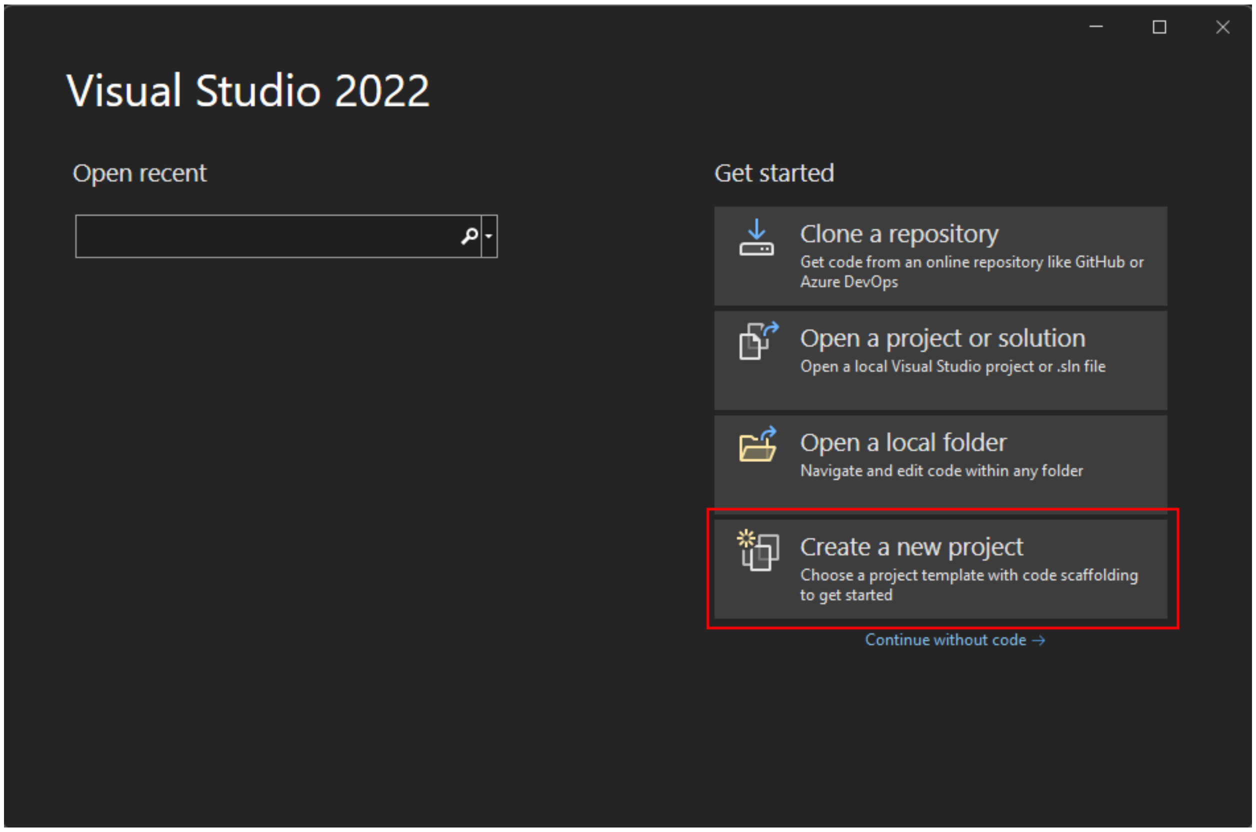
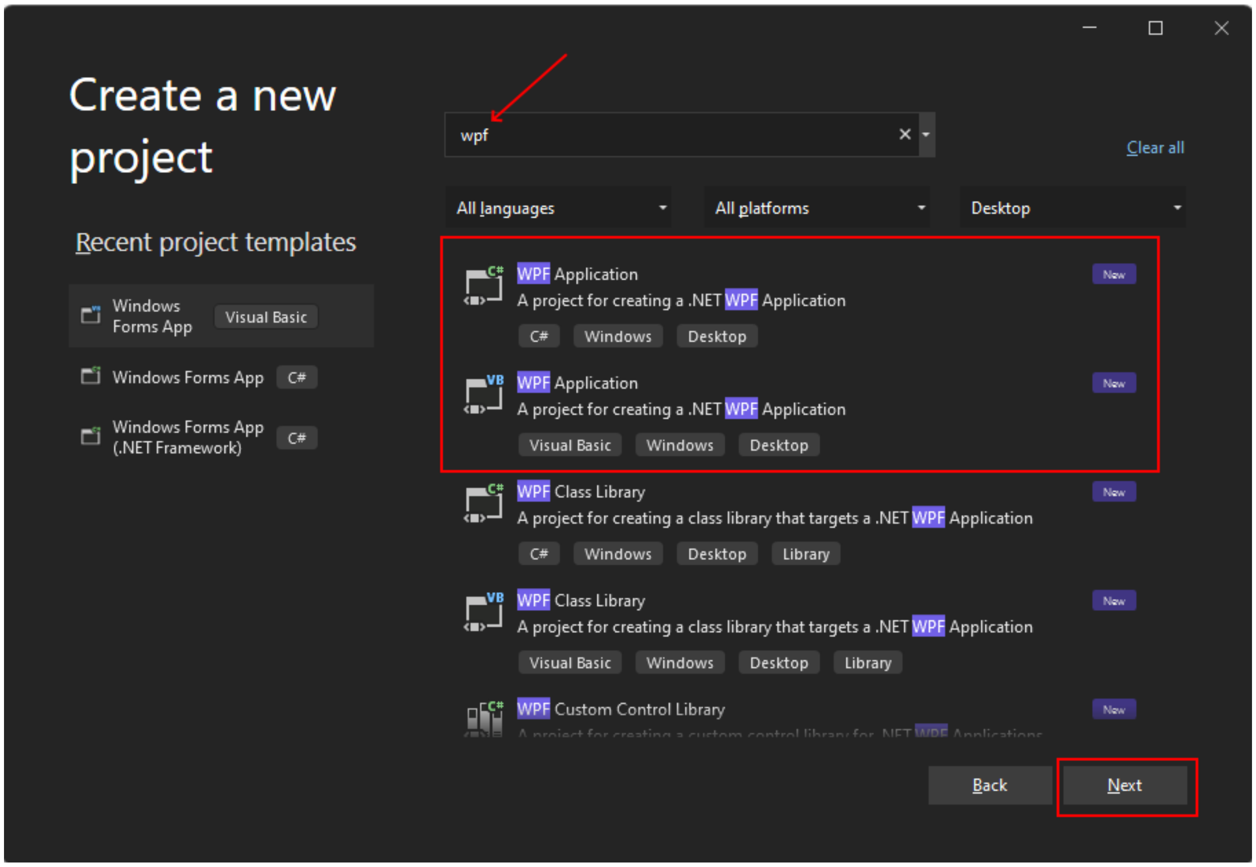

Visual Studio 2022
Open recent
Get started
Clone a repository
Get code from an online repository like GitHub or Azure DevOps
Open a project or solution
Open a local Visual Studio project or .sln file
Open a local folder
Navigate and edit code within any folder
Create a new project
Choose a project template with code scaffolding to get started
Continue without code →

Figure 9: Create a new project

3. In the **Search for templates** box, type *wpf*, and then press Enter.
4. In the **code language** dropdown, choose **C#** or **Visual Basic**.
5. In the templates list, select **WPF Application** and then select **Next**.

The following image shows both C# and Visual Basic .NET project templates. If you applied the **code language** filter, you'll see the corresponding template.


Create a new project

Recent project templates

Windows Forms App · Visual Basic

Windows Forms App · C#

Windows Forms App (.NET Framework) · C#

wpf

Clear all

All languages · All platforms · Desktop

WPF Application · New
A project for creating a .NET WPF Application
C# · Windows · Desktop

WPF Application · New
A project for creating a .NET WPF Application
Visual Basic · Windows · Desktop

WPF Class Library · New
A project for creating a class library that targets a .NET WPF Application
C# · Windows · Desktop · Library

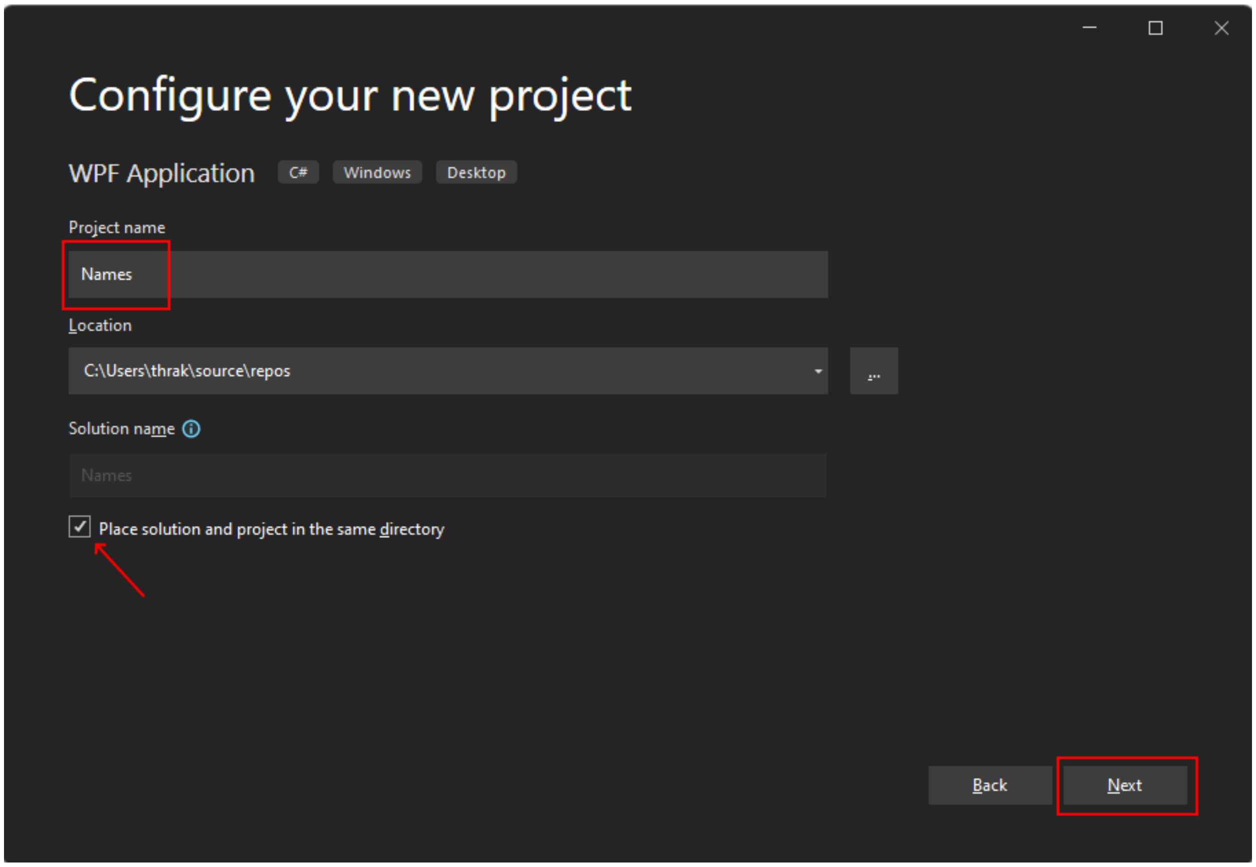
WPF Class Library · New
A project for creating a class library that targets a .NET WPF Application
Visual Basic · Windows · Desktop · Library

WPF Custom Control Library · New

Back · Next

Figure 10: Create a new project

6. In the **Configure your new project** window, do the following:
    1. In the **Project name** box, enter *Names*.
    2. Select the **Place solution and project in the same directory** check box.
    3. Optionally, choose a different **Location** to save your code.
    4. Select the **Next** button.



Configure your new project

WPF Application

Project name: Names

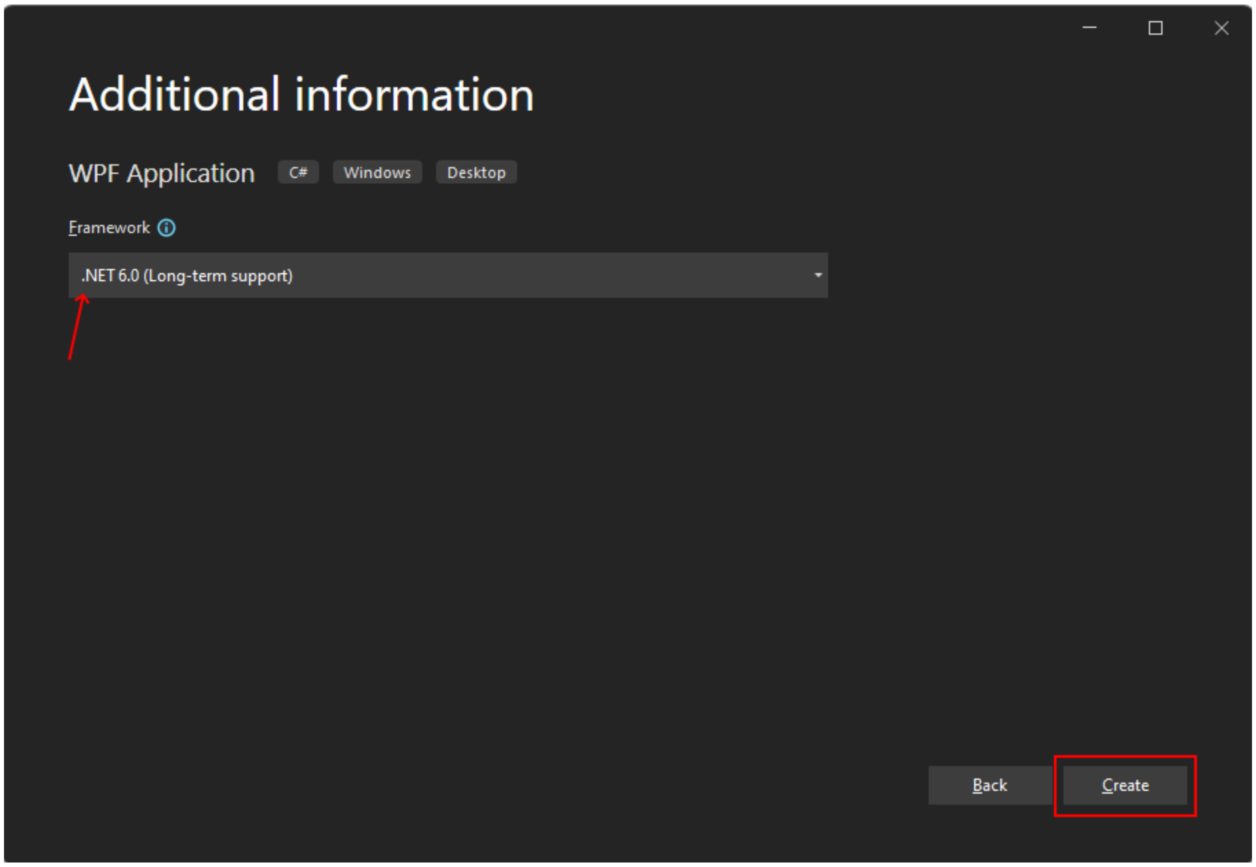
Location: C:\Users\thrak\source\repos

Solution name: Names

✓ Place solution and project in the same directory

Figure 11: Create a new project

7. In the **Additional information** window, select **.NET 6.0 (Long-term support)** for **Target Framework**. Select the **Create** button.


Additional information

WPF Application

Framework

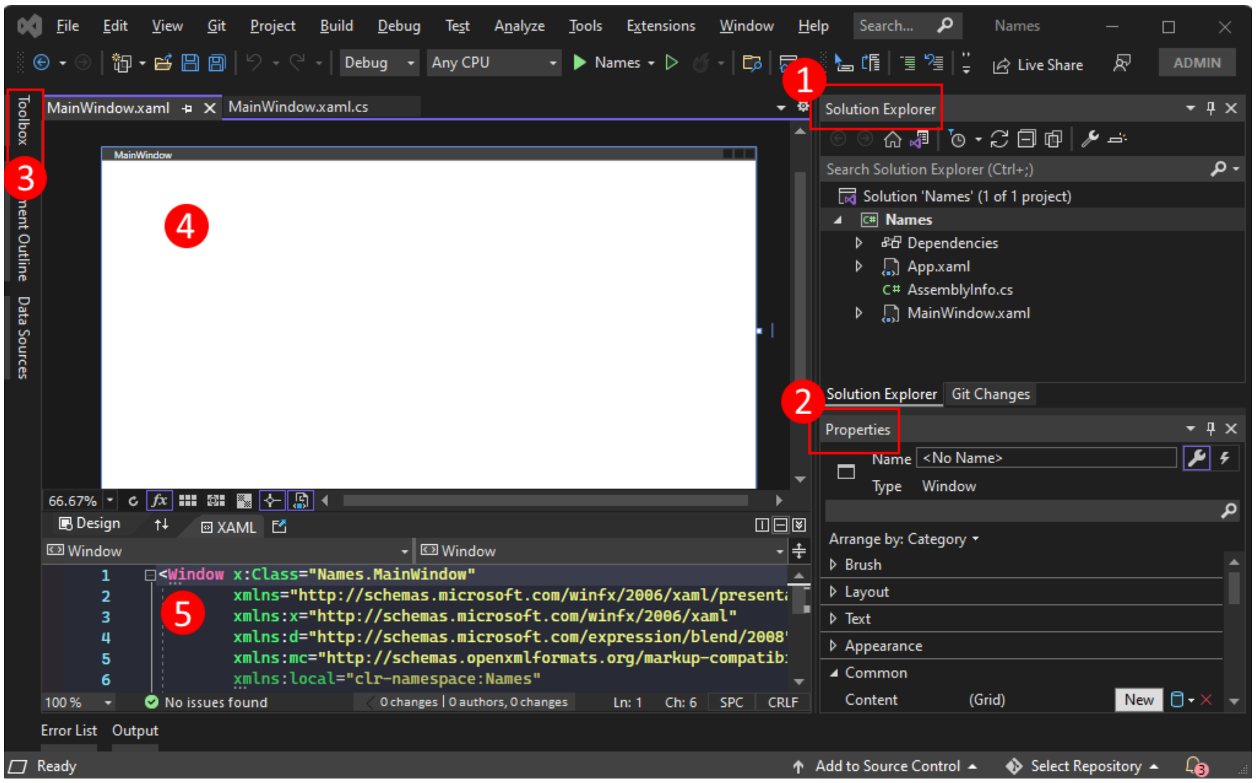
.NET 6.0 (Long-term support)

Figure 12: Additional Information

Once the app is generated, Visual Studio should open the XAML designer pane for the default window, *MainWindow*. If the designer isn't visible, double-click on the *MainWindow.xaml* file in the **Solution Explorer** pane to open the designer.

Important parts of Visual Studio

Support for WPF in Visual Studio has five important components that you'll interact with as you create an app:



```
<Window x:Class="Names.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presenta
        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-compatib
        xmlns:local="clr-namespace:Names"
```

Figure 13: Visual Studio Layout

1. Solution Explorer

   All of your project files, code, windows, resources, will appear in this pane.

2. Properties

   This pane shows property settings you can configure based on the item selected. For example, if you select an item from **Solution Explorer**, you'll see property settings related to the file. If you select an object in the **Designer**, you'll see settings for that item.

3. Toolbox

   The toolbox contains all of the controls you can add to a form. To add a control to the current form, double-click a control or drag-and-drop the control.

4. XAML designer

   This is the designer for a XAML document. It's interactive and you can drag-and-drop objects from the **Toolbox**. By selecting and moving items in the designer, you can visually compose the user interface (UI) for your app. When both the designer and editor are visible, changes to one is reflected in the other. When you select items in the designer, the **Properties** pane displays the properties and attributes about that object.

5. XAML code editor

   This is the XAML code editor for a XAML document. The XAML code editor is a way to craft your UI by hand without a designer. The designer may infer the values of properties on a control when the control is added in the designer. The XAML code editor gives you a lot more control.

Activity 2:

Examine the XAML

After your project is created, the XAML code editor is visible with a minimal amount of XAML code to display the window. If the editor isn't open, double-click the MainWindow.xaml item in the Solution Explorer. You should see XAML similar to the following example:

```
<Window x:Class="Names.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:Names"
        mc:Ignorable="d"
        Title="MainWindow" Height="450" Width="800">
    <Grid>

    </Grid>
</Window>
```

Let's break down this XAML code to understand it better. XAML is simply XML that can be processed by the compilers that WPF uses. It describes the WPF UI and interacts with .NET code. To understand XAML, you should, at a minimum, be familiar with the basics of XML.

The document root <Window> represents the type of object being described by the XAML file. There are eight attributes declared, and they generally belong to three categories:

- Namespaces

An XML namespace provides structure to the XML, determining what XML content can be declared in the file. The main xmlns attribute imports the XML namespace for the entire file, and in this case, maps to the types declared by WPF. The other XML namespaces declare a prefix and import other types and objects for the XAML file. For example, the xmlns:local namespace declares the local prefix and maps to the objects declared by your project, the ones declared in the Names code namespace.

- x:Class attribute

This attribute maps the <Window> to the type defined by your code: the *MainWindow.xaml.cs* or *MainWindow.xaml.vb* file, which is the Names.MainWindow class.

- Title attribute

Any normal attribute declared on the XAML object sets a property of that object. In this case, the Title attribute sets the Window.Title property.

Prepare the layout

WPF provides a powerful layout system with many different layout controls. Layout controls help place and size child controls, and can even do so automatically. The default layout control provided to you in this XAML is the <Grid> control.

The Grid control lets you define rows and columns, much like a table, and place controls within the bounds of a specific row and column combination. You can have any number of child controls or other layout controls added to the Grid. For example, you can place another Grid control in a specific row and column combination, and that new Grid can then define more rows and columns and have its own children.

The <Grid> control defines rows and columns in which your controls will be. A grid always has a single row and column declared, meaning, the grid by default is a single cell. That doesn't really give you much flexibility in placing controls.

Next, define two rows and two columns, dividing the grid into four cells:

```
<Window x:Class="Names.LayoutStep2"

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:Names"
        mc:Ignorable="d"
        Title="Names" Height="180" Width="260">

     <Grid Margin="10">
          <Grid.RowDefinitions>
               <RowDefinition Height="*" />
               <RowDefinition Height="*" />
          </Grid.RowDefinitions>
          <Grid.ColumnDefinitions>
               <ColumnDefinition Width="*" />
```

```
                <ColumnDefinition Width="*" />
            </Grid.ColumnDefinitions>
        </Grid>
</Window>
```

Select the grid in either the XAML code editor or XAML designer, you'll see that the XAML designer shows each row and column:

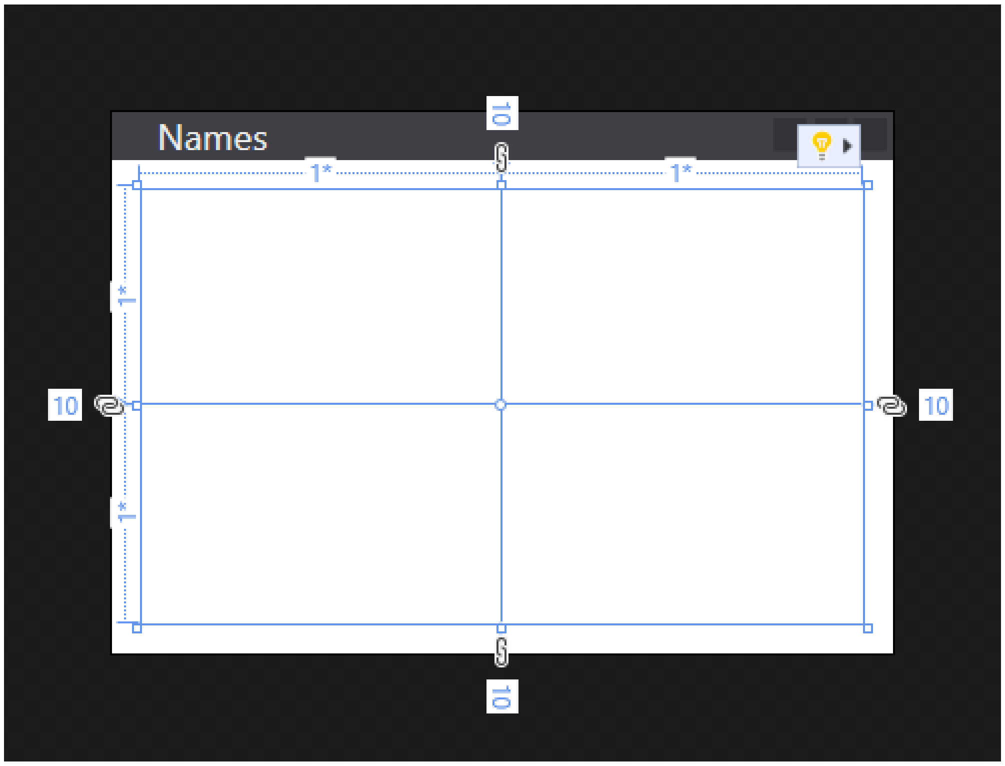


Figure 14: Grid Layout

Add the first control

```
<Grid Margin="10">

    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>
```

```
 <Label>Names</Label>

</Grid>
```

Create the name list box

Now that the grid is correctly sized and the label created, add a list box control on the row below the label. The list box will be in row 1 and column 0. We'll also give this control the name of lstNames. Once a control is named, it can be referenced in the code-behind. The name is assigned to the control with the x:Name attribute.

```
<Grid Margin="10">

    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>

    <Label>Names</Label>
 <ListBox Grid.Row="1" x:Name="lstNames" />

</Grid>
```

Next, create a <TextBox> and <Button> control in the <StackPanel>.

```
<StackPanel Grid.Row="1" Grid.Column="1" Margin="5,0,0,0">
    <TextBox x:Name="txtName" />
    <Button x:Name="btnAdd" Margin="0,5,0,0">Add Name</Button>
</StackPanel>
```

Activity 3:

Add code for the Click event

The <Button> we created has a Click event that is raised when the user presses the button. You can subscribe to this event and add code to add a name to the list box. Just like you set a property on a control by adding a XAML attribute, you can use a XAML attribute to subscribe to an event. Set the Click attribute to ButtonAddName_Click.

```
<StackPanel Grid.Row="1" Grid.Column="1" Margin="5,0,0,0">
    <TextBox x:Name="txtName" />
       <Button x:Name="btnAdd" Margin="0,5,0,0"
Click="ButtonAddName_Click">
     Add Name
        </Button>
</StackPanel>
```

Next, add the following code to do these three steps:

1. Make sure that the text box contains a name.
2. Validate that the name entered in the text box doesn't already exist.
3. Add the name to the list box.

```
private void ButtonAddName_Click(object sender, RoutedEventArgs e)
{
     if (!string.IsNullOrWhiteSpace(txtName.Text) &&
     !lstNames.Items.Contains(txtName.Text))
    {
        lstNames.Items.Add(txtName.Text);
        txtName.Clear();
    }
}
```

Run the app

Now that the event has been coded, you can run the app by pressing the F5 key or by selecting **Debug** > **Start Debugging** from the menu. The window is displayed and you can enter a name in the textbox and then add it by clicking the button.

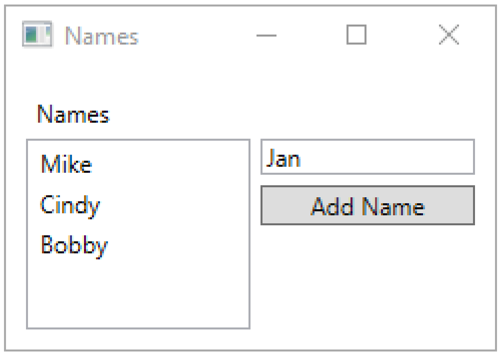


Figure 15: Run the app

Activity 4:

Checkboxes and textboxes

Drag two checkboxes and two textboxes from a toolbox and set the following properties in the properties window.

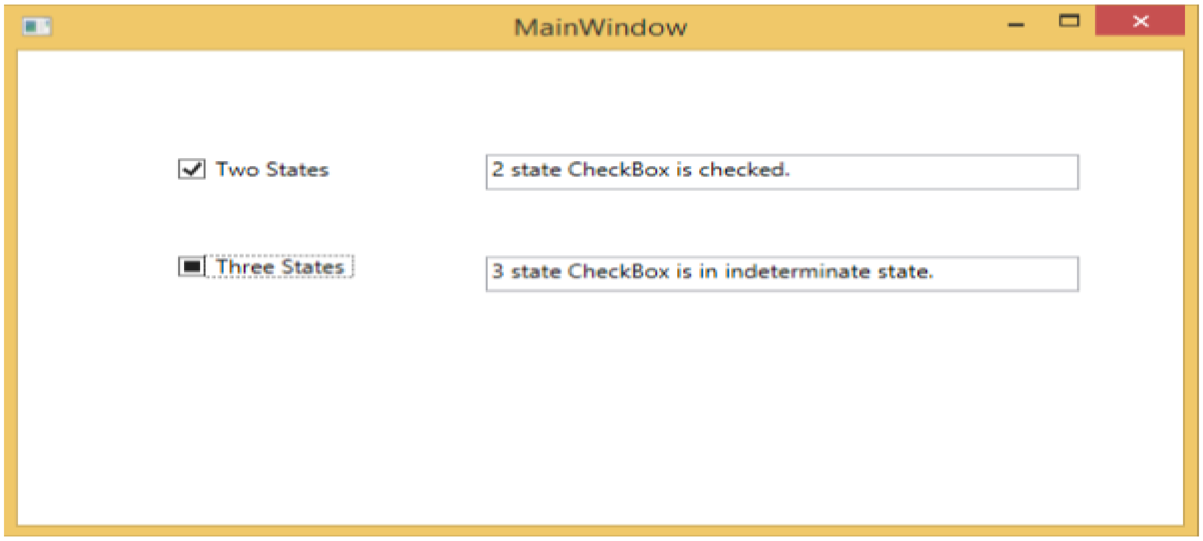
Solution:

```
<Window x:Class="WpfApp2.Window2"
   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:WPFCheckBoxControl"
   mc:Ignorable = "d" Title = "MainWindow" Height = "350" Width =
"604">

    <Grid>
        <CheckBox x:Name = "checkBox1" Content = "Two States"
HorizontalAlignment = "Left"
         Margin = "80,70,0,0" VerticalAlignment = "Top" Checked =
"HandleCheck"
         Unchecked = "HandleUnchecked" Width = "90"/>

        <CheckBox x:Name = "checkBox2" Content = "Three States"
         HorizontalAlignment = "Left" Margin = "80,134,0,0"
VerticalAlignment = "Top"
         Width = "90" IsThreeState = "True" Indeterminate =
"HandleThirdState"
         Checked = "HandleCheck" Unchecked = "HandleUnchecked"/>
        <TextBox x:Name = "textBox1" HorizontalAlignment = "Left"
         Height = "23" Margin = "236,68,0,0" TextWrapping = "Wrap"
         VerticalAlignment = "Top" Width = "300"/>

        <TextBox x:Name = "textBox2" HorizontalAlignment = "Left"
         Height = "23" Margin = "236,135,0,0" TextWrapping = "Wrap"
         VerticalAlignment = "Top" Width = "300"/>
    </Grid>

</Window>
```

Here is the implementation in C# for different events

```
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.Shapes;

namespace WpfApp2
{
    /// <summary>
    /// Interaction logic for Window2.xaml
    /// </summary>
    public partial class Window2 : Window
    {
        public Window2()
        {
            InitializeComponent();
        }

        private void HandleCheck(object sender, RoutedEventArgs e)
        {
            CheckBox cb = sender as CheckBox;
            if (cb.Name == "checkBox1")
                textBox1.Text = "2 state CheckBox is checked.";
            else
                textBox2.Text = "3 state CheckBox is checked.";
        }

        private void HandleUnchecked(object sender, RoutedEventArgs e)
        {
            CheckBox cb = sender as CheckBox;

            if (cb.Name == "checkBox1")
                textBox1.Text = "2 state CheckBox is unchecked.";
            else
                textBox2.Text = "3 state CheckBox is unchecked.";
        }

        private void HandleThirdState(object sender, RoutedEventArgs
e)
        {
            CheckBox cb = sender as CheckBox;
            textBox2.Text = "3 state CheckBox is in indeterminate
state.";
        }
    }
}
```

Output



2 state CheckBox is checked.

3 state CheckBox is in indeterminate state.

Figure 16: Output

Activity 5:

Radio button

Drag five radio buttons and four text blocks from the Toolbox and arrange them as shown in the following XAML code.

Solution:

```
<Window x:Class = "WPFRadioButtonControl.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:WPFRadioButtonControl"
   mc:Ignorable = "d" Title = "MainWindow" Height = "350" Width =
"604">

    <Grid>
        <TextBlock x:Name = "textBlock" HorizontalAlignment = "Left"
         Margin = "23,68,0,0" TextWrapping = "Wrap" Text = "Gender:"
         VerticalAlignment = "Top" Width = "83" />

        <TextBlock x:Name = "textBlock1" HorizontalAlignment = "Left"
         Margin = "23,134,0,0" TextWrapping = "Wrap" Text = "Marital
Status:"
         VerticalAlignment = "Top" Width = "83" />

        <RadioButton x:Name = "rb1" Content = "Male"
HorizontalAlignment = "Left"
         Margin = "126,68,0,0" VerticalAlignment = "Top"
         GroupName = "Gender" Width = "69" Checked = "HandleCheck" />

        <RadioButton x:Name = "rb2" Content = "Female"
HorizontalAlignment = "Left"
         Margin = "201,68,0,0" VerticalAlignment = "Top"
         GroupName = "Gender" Width = "81" Checked = "HandleCheck" />

        <RadioButton x:Name = "rb3" Content = "Single"
HorizontalAlignment = "Left"
         Margin = "126,134,0,0" VerticalAlignment = "Top"
         GroupName = "Status" Width = "69" Checked = "HandleCheck1" />

        <RadioButton x:Name = "radioButton" Content = "Engaged"
```

```
HorizontalAlignment = "Left"
         Margin = "201,134,0,0" VerticalAlignment = "Top"
         GroupName = "Status" Width = "89" Checked = "HandleCheck1" />

        <RadioButton x:Name = "radioButton1" Content = "Married"
         GroupName = "Status" HorizontalAlignment = "Left" Margin =
"302,134,0,0"
         VerticalAlignment = "Top" Width = "95" Checked =
"HandleCheck1" />

        <TextBlock x:Name = "textBlock2" HorizontalAlignment = "Left"
         Margin = "386,68,0,0" TextWrapping = "Wrap"
         VerticalAlignment = "Top" Width = "191" Height = "26" />

        <TextBlock x:Name = "textBlock3" HorizontalAlignment = "Left"
         Margin = "386,134,0,0" TextWrapping = "Wrap"
         VerticalAlignment = "Top" Width = "146" Height = "31" />
    </Grid>

</Window>
```

Here is the implementation in C# for different events

```
using System.Windows;
using System.Windows.Controls;

namespace WPFRadioButtonControl
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>

    public partial class MainWindow : Window
    {

        public MainWindow()
        {
            InitializeComponent();
        }

        private void HandleCheck1(object sender, RoutedEventArgs e)
        {
            RadioButton rb = sender as RadioButton;
            textBlock3.Text = "You are " + rb.Content;
        }
        private void HandleCheck(object sender, RoutedEventArgs e)
        {
            RadioButton rb = sender as RadioButton;
            textBlock2.Text = "You are " + rb.Content;
        }

    }
}
```

Output

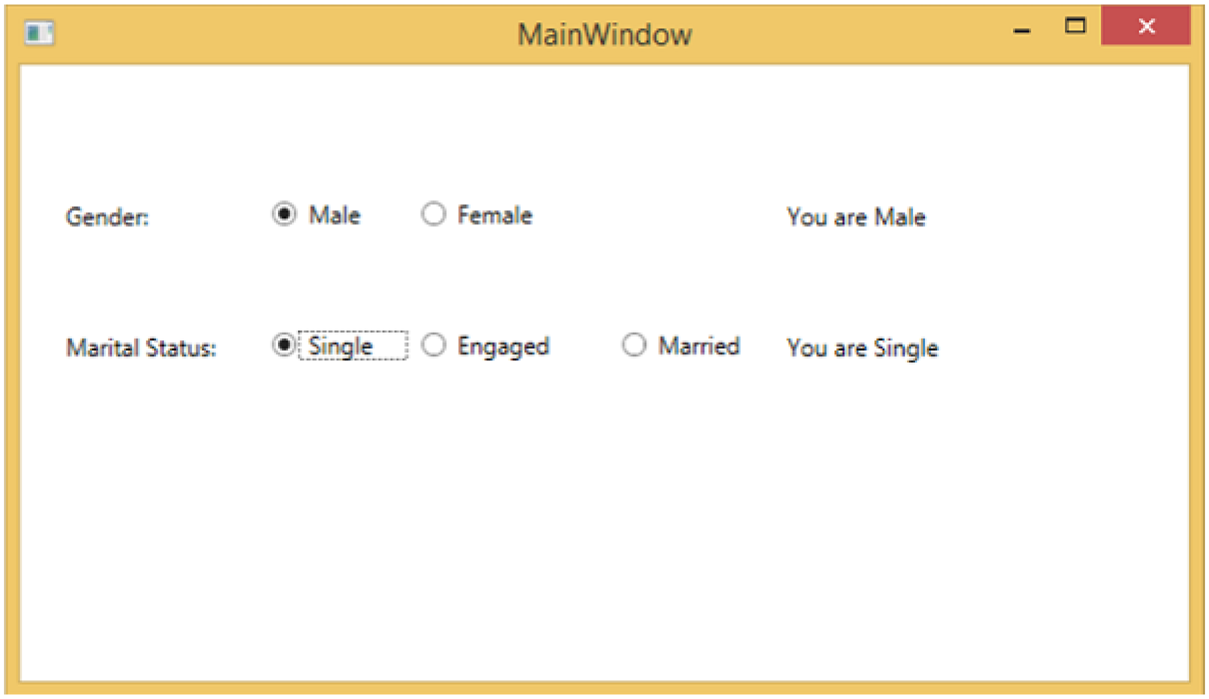


Figure 17: Output

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab.

Lab Task 1

Create a WPF application which reads following data from user:

- *Name (textbox)*
- *Age (texbox)*
- *Known programming languages (checkboxes) e.g. java, c#, python etc.*

Once user enters data and press submit button then the provided information should be displayed in a single textblock.

Lab Task 2

Create a user registration form with following fields:

- *Full name (textbox)*
- *Username (textbox)*
- *Password (password field)*
- *Confirm Password (password field)*
- *Gender (radio button)*
- *Hobbies (checkboxes)*

Once user enters data and press submit button then the provided information should be displayed in a dialog box.

Note: *All fields are mandatory. Password and confirm password must match with each other. If these conditions are not fulfilled then show error message to user in the dialog box.*

Lab 05
WPF Common Controls

Objective:

Purpose of this lab is to familiarize the students with common controls of WPF applications.

Activity Outcomes:

The activities provide hands - on practice with the following topics:

- Common Controls such as Image, Grid, ComboBox, ProgressBar etc.

Instructor Note:

As pre-lab activity, read WPF - Controls (tutorialspoint.com)

1) Useful Concepts

A combobox is a selection control that combines a non-editable textbox and a drop-down listbox that allows users to select an item from a list. It either displays the current selection or is empty if there is no selected item.

A control that displays an image, you can use either the Image object or the ImageBrush object. An Image object display an image, while an ImageBrush object paints another object with an image. The image source is specified by referring to an image file using several supported formats. It can display the following formats −

- Bitmap (BMP)
- Tagged Image File Format (TIFF)
- Icons (ICO)
- Joint Photographic Experts Group (JPEG)
- Graphics Interchange Format (GIF)
- Portable Network Graphics (PNG)
- JPEG XR

A GridView is a control that displays data items in rows and columns. Actually a ListView displays data. By default, it contains a GridView.

ProgressBar is a control that indicates the progress of an operation, where the typical visual appearance is a bar that animates a filled area as the progress continues. It can show the progress in one of the two following styles −

- A bar that displays a repeating pattern, or
- A bar that fills based on a value.

2) Solved Lab Activites

Sr. No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
Activity 1	***10 mins***	***Low***	***CLO-4***
Activity 2	***10 mins***	***Low***	***CLO-4***
Activity 3	***20 mins***	***Low***	***CLO-4***
Activity 4	***20 mins***	***Medium***	***CLO-4***

Activity 1:

Combobox

Drag two comboboxes and two textboxes from a toolbox and set the following properties in the properties window.

Solution:

```
<Window x:Class = "WPFComboBoxControl.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:WPFComboBoxControl"
   mc:Ignorable = "d" Title = "MainWindow" Height = "350" Width =
"604">

    <Grid>

        <ComboBox x:Name = "comboBox" HorizontalAlignment = "Left"
         Margin = "80,53,0,0" VerticalAlignment = "Top" Width = "120"
         SelectionChanged = "Combo_SelectionChanged">

            <ComboBoxItem Content = "Item #1" />
            <ComboBoxItem Content = "Item #2" />
            <ComboBoxItem Content = "Item #3" />
        </ComboBox>

        <ComboBox x:Name = "comboBox1" HorizontalAlignment = "Left"
         Margin = "80,153,0,0" VerticalAlignment = "Top" Width = "120"
         IsEditable = "True"
         SelectionChanged = "Combo1_SelectionChanged">

            <ComboBoxItem Content = "Item #1" />
            <ComboBoxItem Content = "Item #2" />
            <ComboBoxItem Content = "Item #3" />
   </ComboBox>

        <TextBox x:Name = "textBox" HorizontalAlignment = "Left"
         Height = "23" Margin = "253,53,0,0" TextWrapping = "Wrap"
         VerticalAlignment = "Top" Width = "200" />

        <TextBox x:Name = "textBox1" HorizontalAlignment = "Left"
         Height = "23" Margin = "253,152,0,0" TextWrapping = "Wrap"
         VerticalAlignment = "Top" Width = "200" />

    </Grid>

</Window>
```

Here is the implementation in C# for different events

```
using System.Windows;
using System.Windows.Controls;

namespace WPFComboBoxControl
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>

    public partial class MainWindow : Window
    {

        public MainWindow()
        {
            InitializeComponent();
        }

        private void Combo_SelectionChanged(object sender,
SelectionChangedEventArgs e)
        {
            textBox.Text = comboBox.SelectedItem.ToString();
        }

        private void Combo1_SelectionChanged(object sender,
SelectionChangedEventArgs e)
        {
            textBox1.Text = comboBox1.SelectedItem.ToString();
        }

  }
}
```

Output

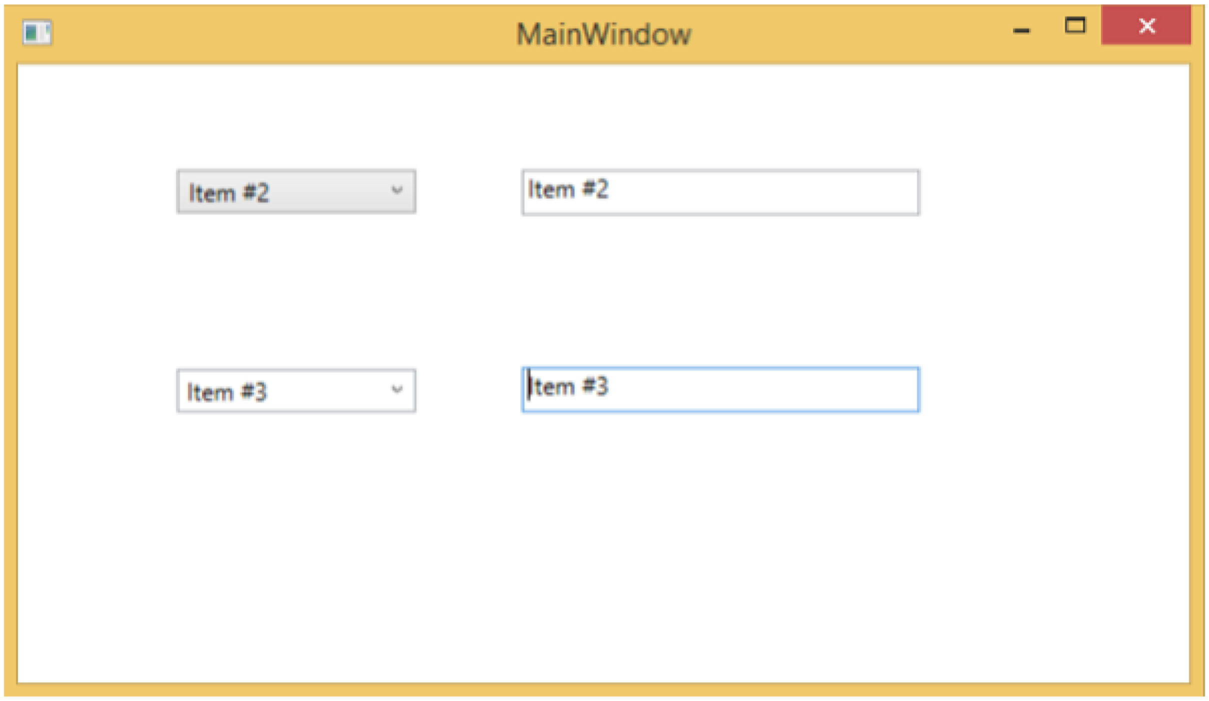


Figure 18: Output

Activity 2:

Image control

First divide the screen into two rows by using Grid.RowDefinition. Drag three Image controls from the Toolbox.

Solution:

```
<Window x:Class = "WPFImageControl.MainWindow"
   xmlns = "http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x = "http://schemas.microsoft.com/winfx/2006/xaml"
   Title = "MainWindow" Height = "500" Width = "604">

  <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height = "1*"/>
            <RowDefinition Height = "1*"/>
        </Grid.RowDefinitions>

        <StackPanel Orientation = "Horizontal">
            <Image Width = "200" Source = "Images\red_rock_01.jpg"
            VerticalAlignment = "Top" Margin = "30"/>
            <Image Width = "200" Source = "Images\red_rock_01.jpg"
VerticalAlignment = "Top"
            Margin = "30" Opacity = "0.5"/>
        </StackPanel>

        <StackPanel Grid.Row = "1">
            <Ellipse Height = "100" Width = "200" HorizontalAlignment
= "Center" Margin = "30">
                <Ellipse.Fill>
                    <ImageBrush ImageSource = "Images\tahoe_01.jpg" />
                </Ellipse.Fill>
            </Ellipse>
        </StackPanel>

    </Grid>

</Window>
```

Output

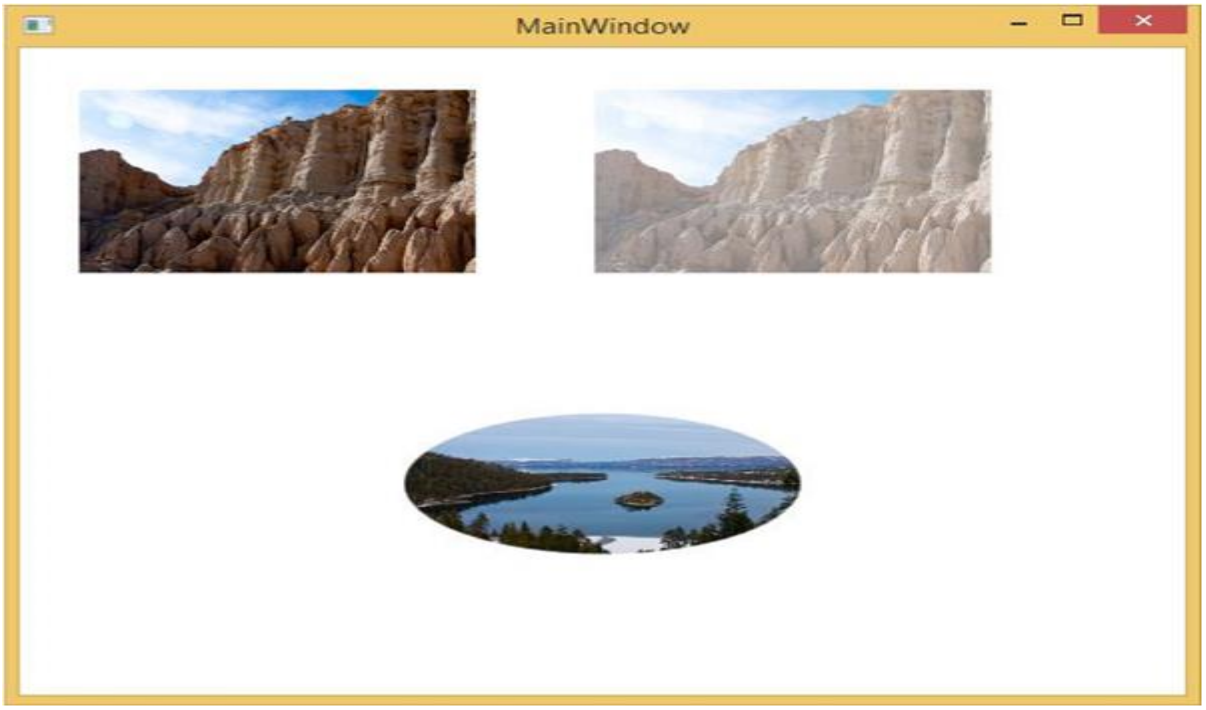


Figure 19: Output

Activity 3:

Gridview

Drag a grid view control from the Toolbox.

Solution:

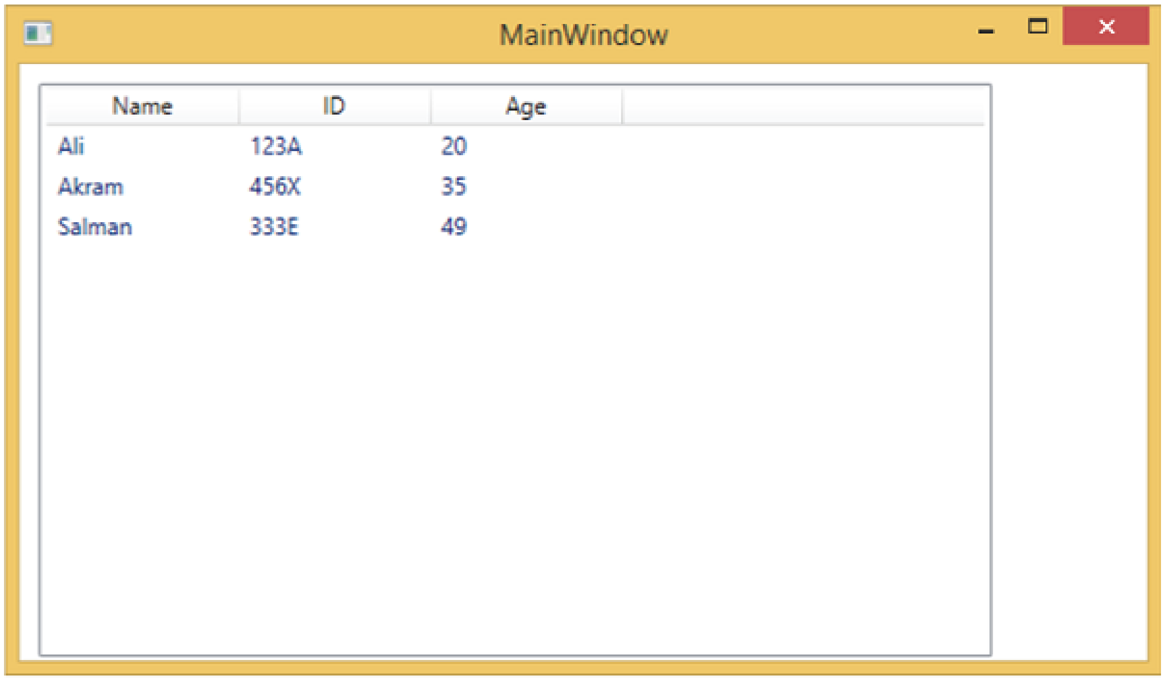
```
<Window x:Class = "WPFGridView.MainWindow"
   xmlns = "http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x = "http://schemas.microsoft.com/winfx/2006/xaml"
   Title = "MainWindow" Height = "350" Width = "525">

    <Grid>
        <ListView HorizontalAlignment = "Left" Height = "299" Margin =
"10,10,0,0"
         VerticalAlignment = "Top" Width = "497"Name = "MenList">

            <ListView.View>
                <GridView>
```

```
                        <GridViewColumn Header = "Name"
DisplayMemberBinding = "{Binding Name}"
                      Width = "100"/>

                        <GridViewColumn Header = "ID" DisplayMemberBinding
= "{Binding ID}"
                      Width = "100"/>

                        <GridViewColumn Header = "Age"
DisplayMemberBinding = "{Binding Age}"
                      Width = "100"/>

                    </GridView>
                </ListView.View>

            </ListView>
        </Grid>

</Window>
```

Here is the implementation in C# for different events

```
using System;
using System.Windows;
using System.Windows.Controls;

namespace WPFGridView
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>

    public partial class MainWindow : Window
    {

        public MainWindow()
        {
            InitializeComponent();

            MenList.Items.Add(new Person() { Name = "Ali", ID =
"123A", Age = 20 });
            MenList.Items.Add(new Person() { Name = "Akram", ID =
"456X", Age = 35 });
            MenList.Items.Add(new Person() { Name = "Salman", ID =
"333E", Age = 49 });
        }
    }

    class Person
    {
        public string Name { get; set; }
        public string ID { get; set; }
   public int Age { get; set; }
    }
}
```

Output



Name	ID	Age
Ali	123A	20
Akram	456X	35
Salman	333E	49

Figure 20: Output

Activity 4:

ProgressBar

The following example shows how to use the ProgressBar control. Here is the XAML code in which two ProgressBar controls are created and initialized.

Solution:

```
<Window x:Class = "WPFProgressBarControl.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:WPFProgressBarControl"
   mc:Ignorable = "d" Title = "MainWindow" Height = "350" Width =
"604">

    <Grid>
        <StackPanel x:Name = "LayoutRoot" Margin = "20">

            <Border BorderThickness = "5" BorderBrush = "Green">
                <StackPanel Background = "White">
                    <TextBlock HorizontalAlignment = "Center" Margin =
"10"
                  Text = "Value-Based Progress Bar" />
                    <ProgressBar x:Name = "pg1" Value = "100"  Margin
= "10" Maximum = "200"
                  Height = "15" IsIndeterminate = "False" />
                </StackPanel>
            </Border>

            <Border BorderThickness = "5" BorderBrush = "Green">
                <StackPanel Background = "White">
                    <TextBlock HorizontalAlignment = "Center"
                  Margin = "10" Text = "Indeterminate Progress Bar" />
                    <ProgressBar x:Name = "pg2" Margin = "10" Height =
"15"
                  IsIndeterminate = "True" />
                </StackPanel>
            </Border>

        </StackPanel>
    </Grid>

</Window>
```

Output

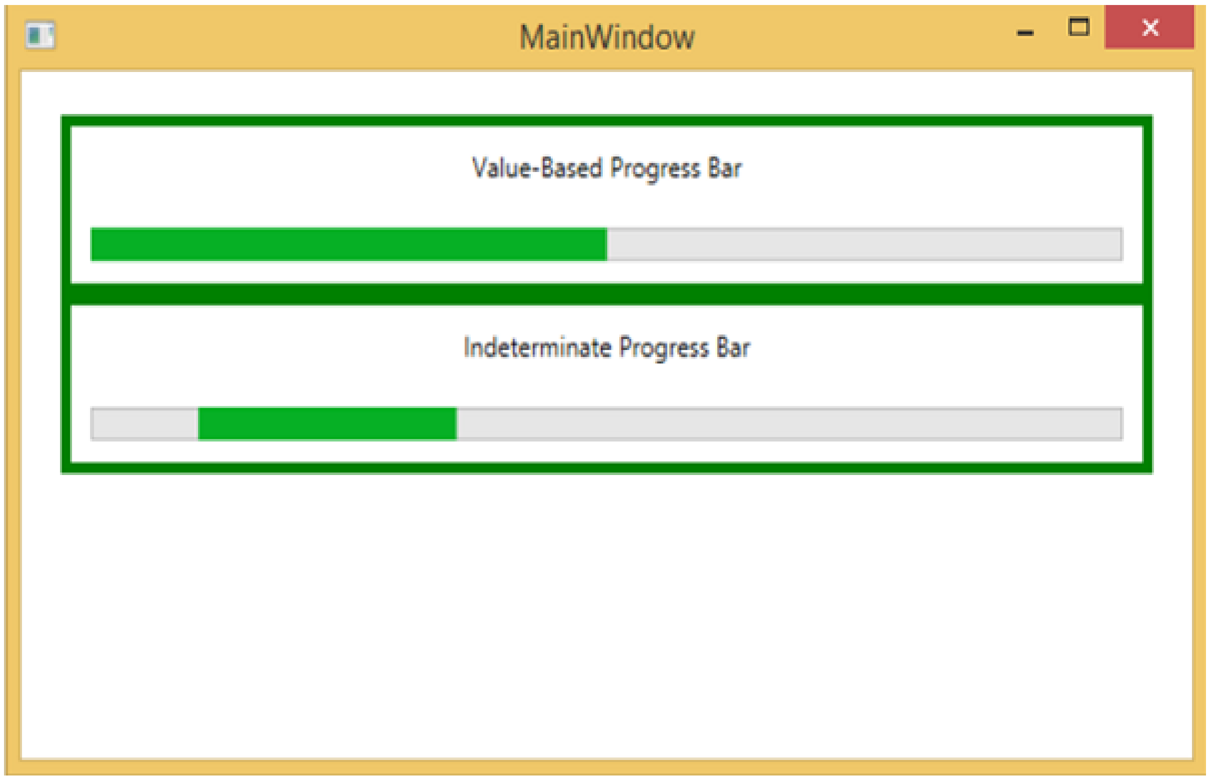


Figure 21: Output

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab.

Lab Task 1

Write a program that visualize which item from combobox is selected.

Lab Task 2

Write a program that shows a ComboBox with various elements added to its Items. For example – add text, ellipse and picture.

Lab Task 3

Create a user registration form with following fields:

- *Username (textbox)*
- *Password (password field)*
- *Role (listbox)*
- *Degree applying for (listbox)*

Once user enters data and press submit button then the provided information should be displayed in a dialog box.

Lab Task 4

Create a new window called BringBackPocoyo, to contain a Border control which in turn contains a horizontal StackPanel control.

Add border and image controls to this window to get the following:

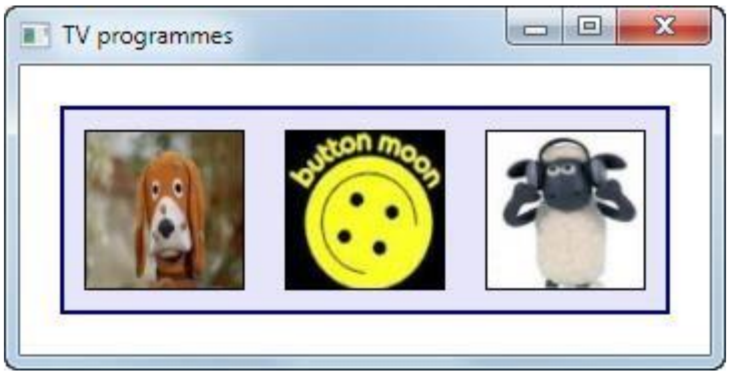


Figure 22: Required Output

Lab 06
Styles and templates

Objective:

Purpose of this lab is to familiarize the students with applying styles and templates to WPF applications.

Activity Outcomes:

The activities provide hands - on practice with the following topics:

- Styles
- Templates

Instructor Note:

As pre-lab activity, read Styles and templates - WPF .NET | Microsoft Docs

1) Useful Concepts

Windows Presentation Foundation (WPF) styling and templating refer to a suite of features that let developers and designers create visually compelling effects and a consistent appearance for their product. When customizing the appearance of an app, you want a strong styling and templating model that enables maintenance and sharing of appearance within and among apps. WPF provides that model.

Another feature of the WPF styling model is the separation of presentation and logic. Designers can work on the appearance of an app by using only XAML at the same time that developers work on the programming logic by using C# or Visual Basic.

2) Solved Lab Activites

Sr. No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
Activity 1	***10 mins***	***Low***	***CLO-4***
Activity 2	***10 mins***	***Low***	***CLO-4***
Activity 3	***10 mins***	***Low***	***CLO-4***
Activity 4	***15 mins***	***Medium***	***CLO-4***
Activity 5	***15 mins***	***Medium***	***CLO-4***

Activity 1:

How to create a style for a control

With Windows Presentation Foundation (WPF), you can customize an existing control's appearance with your own reusable style. Styles can be applied globally to your app, windows and pages, or directly to controls.

Create a style

You can think of a Style as a convenient way to apply a set of property values to one or more elements. You can use a style on any element that derives from FrameworkElement or FrameworkContentElement such as a Window or a Button.

The most common way to declare a style is as a resource in the Resources section in a XAML file. Because styles are resources, they obey the same scoping rules that apply to all resources. Put simply, where you declare a style affects where the style can be applied. For example, if you declare the style in the root element of your app definition XAML file, the style can be used anywhere in your app.

```
<Application x:Class="IntroToStylingAndTemplating.App"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:IntroToStylingAndTemplating"
             StartupUri="WindowExplicitStyle.xaml">
    <Application.Resources>
        <ResourceDictionary>
            <Style x:Key="Header1" TargetType="TextBlock">
                <Setter Property="FontSize" Value="15" />
                <Setter Property="FontWeight" Value="ExtraBold" />
            </Style>
        </ResourceDictionary>
    </Application.Resources>
</Application>
```

If you declare the style in one of the app's XAML files, the style can be used only in that XAML file.

```
<Window x:Class="IntroToStylingAndTemplating.WindowSingleResource"

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:IntroToStylingAndTemplating"
        mc:Ignorable="d"
        Title="WindowSingleResource" Height="450" Width="800">
    <Window.Resources>

        <Style x:Key="Header1" TargetType="TextBlock">
            <Setter Property="FontSize" Value="15" />
            <Setter Property="FontWeight" Value="ExtraBold" />
        </Style>

    </Window.Resources>
    <Grid />
</Window>
```

A style is made up of <Setter> child elements that set properties on the elements the style is applied to. In the example above, notice that the style is set to apply to TextBlock types through the TargetType attribute. The style will set the FontSize to 15 and the FontWeight to ExtraBold. Add a <Setter> for each property the style changes.

Activity 2:

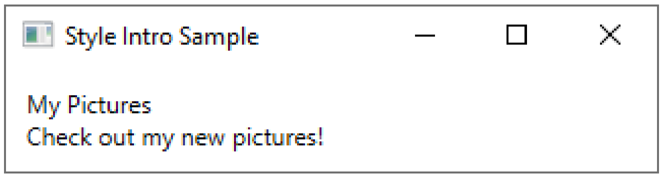
Apply a style implicitly

A Style is a convenient way to apply a set of property values to more than one element. For example, consider the following TextBlock elements and their default appearance in a window.

```
<StackPanel>
    <TextBlock>My Pictures</TextBlock>
    <TextBlock>Check out my new pictures!</TextBlock>
```

```
</StackPanel>
```

Output


Style Intro Sample

My Pictures
Check out my new pictures!

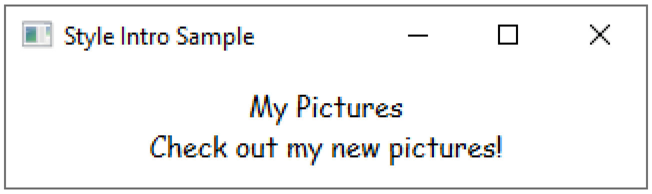
Figure 23: Output

You can change the default appearance by setting properties, such as FontSize and FontFamily, on each TextBlock element directly. However, if you want your TextBlock elements to share some properties, you can create a Style in the Resources section of your XAML file, as shown here.

```
<Window.Resources>
    <!--A Style that affects all TextBlocks-->
    <Style TargetType="TextBlock">
        <Setter Property="HorizontalAlignment" Value="Center" />
        <Setter Property="FontFamily" Value="Comic Sans MS"/>
        <Setter Property="FontSize" Value="14"/>
    </Style>
</Window.Resources>
```

When you set the TargetType of your style to the TextBlock type and omit the x:Key attribute, the style is applied to all the TextBlock elements scoped to the style, which is generally the XAML file itself.

Now the TextBlock elements appear as follows.


Style Intro Sample

My Pictures
Check out my new pictures!

Figure 24: Output

Activity 3:

Apply a style explicitly

If you add an x:Key attribute with value to the style, the style is no longer implicitly applied to all elements of TargetType. Only elements that explicitly reference the style will have the style applied to them.

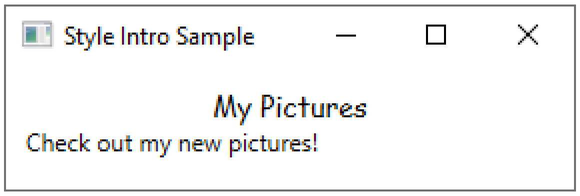
Here is the style from the previous section, but declared with the x:Key attribute.

```
<Window.Resources>
    <Style x:Key="TitleText" TargetType="TextBlock">
        <Setter Property="HorizontalAlignment" Value="Center" />
        <Setter Property="FontFamily" Value="Comic Sans MS"/>
        <Setter Property="FontSize" Value="14"/>
    </Style>
</Window.Resources>
```

To apply the style, set the Style property on the element to the x:Key value, using a StaticResource markup extension, as shown here.

```
<StackPanel>
    <TextBlock Style="{StaticResource TitleText}">My
Pictures</TextBlock>
    <TextBlock>Check out my new pictures!</TextBlock>
</StackPanel>
```

Notice that the first TextBlock element has the style applied to it while the second TextBlock element remains unchanged. The implicit style from the previous section was changed to a style that declared the x:Key attribute, meaning, the only element affected by the style is the one that referenced the style directly.



My Pictures
Check out my new pictures!

Figure 25: Output

Once a style is applied, explicitly or implicitly, it becomes sealed and can't be changed. If you want to change a style that has been applied, create a new style to replace the existing one.

Apply a style programmatically

To assign a named style to an element programmatically, get the style from the resources collection and assign it to the element's Style property. The items in a resources collection are of type Object. Therefore, you must cast the retrieved style to a System.Windows.Style before assigning it to the Style property. For example, the following code sets the style of a TextBlock named textblock1 to the defined style TitleText.

```
textblock1.Style = (Style)Resources["TitleText"];
```

Activity 4:

Extend a style

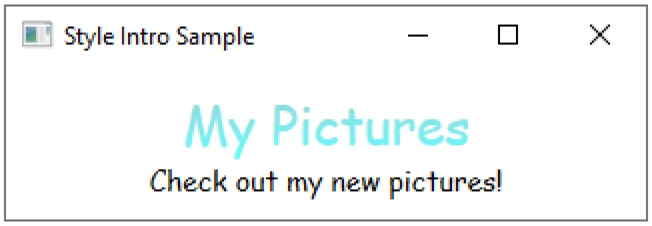
Perhaps you want your two TextBlock elements to share some property values, such as the FontFamily and the centered HorizontalAlignment. But you also want the text My Pictures to have some additional properties. You can do that by creating a new style that is based on the first style, as shown here.

```
<Window.Resources>
    <!-- .... other resources .... -->

    <!--A Style that affects all TextBlocks-->
    <Style TargetType="TextBlock">
        <Setter Property="HorizontalAlignment" Value="Center" />
        <Setter Property="FontFamily" Value="Comic Sans MS"/>
        <Setter Property="FontSize" Value="14"/>
    </Style>

    <!--A Style that extends the previous TextBlock Style with an
x:Key of TitleText-->
    <Style BasedOn="{StaticResource {x:Type TextBlock}}"
           TargetType="TextBlock"
           x:Key="TitleText">
        <Setter Property="FontSize" Value="26"/>
        <Setter Property="Foreground">
            <Setter.Value>
                <LinearGradientBrush StartPoint="0.5,0"
EndPoint="0.5,1">
                    <LinearGradientBrush.GradientStops>
                        <GradientStop Offset="0.0" Color="#90DDDD" />
                        <GradientStop Offset="1.0" Color="#5BFFFF" />
                    </LinearGradientBrush.GradientStops>
                </LinearGradientBrush>
            </Setter.Value>
        </Setter>
    </Style>
</Window.Resources>
```

```
<StackPanel>
    <TextBlock Style="{StaticResource TitleText}" Name="textblock1">My
Pictures</TextBlock>
    <TextBlock>Check out my new pictures!</TextBlock>
</StackPanel>
```

This TextBlock style is now centered, uses a Comic Sans MS font with a size of 26, and the foreground color set to the LinearGradientBrush shown in the example. Notice that it overrides the FontSize value of the base style. If there's more than one Setter pointing to the same property in a Style, the Setter that is declared last takes precedence.

The following shows what the TextBlock elements now look like:



My Pictures

Check out my new pictures!

Figure 26: Output

This TitleText style extends the style that has been created for the TextBlock type, referenced with BasedOn="{StaticResource {x:Type TextBlock}}". You can also extend a style that has an x:Key by using the x:Key of the style. For example, if there was a style named Header1 and you wanted to extend that style, you would use BasedOn="{StaticResource Header1}".

Relationship of the TargetType property and the x:Key attribute

As previously shown, setting the TargetType property to TextBlock without assigning the style an x:Key causes the style to be applied to all TextBlock elements. In this case, the x:Key is implicitly set to {x:Type TextBlock}. This means that if you explicitly set the x:Key value to anything other than {x:Type TextBlock}, the Style isn't applied to all TextBlock elements automatically. Instead, you must apply the style (by using the x:Key value) to the TextBlock elements explicitly. If your style is in the resources section and you don't set the TargetType property on your style, then you must set the x:Key attribute.

In addition to providing a default value for the x:Key, the TargetType property specifies the type to which setter properties apply. If you don't specify a TargetType, you must qualify the properties in your Setter objects with a class name by using the syntax Property="ClassName.Property".

For example, instead of setting Property = "FontSize", you must set Property to "TextBlock.FontSize" or "Control.FontSize".

Also note that many WPF controls consist of a combination of other WPF controls. If you create a style that applies to all controls of a type, you might get unexpected results. For example, if you create a style that targets the TextBlock type in a Window, the style is applied to all TextBlock controls in the window, even if the TextBlock is part of another control, such as a ListBox.

Activity 5:

Control Templates

In WPF, the ControlTemplate of a control defines the appearance of the control. You can change the structure and appearance of a control by defining a new ControlTemplate and assigning it to a control. In many cases, templates give you enough flexibility so that you do not have to write your own custom controls.

Each control has a default template assigned to the Control.Template property. The template connects the visual presentation of the control with the control's capabilities. Because you define a template in XAML, you can change the control's appearance without writing any code. Each template is designed for a specific control, such as a Button.

Control templates are a lot more involved than a style. This is because the control template rewrites the visual appearance of the entire control, while a style simply applies property changes to the existing control. However, since the template of a control is applied by setting the Control.Template property, you can use a style to define or set a template.

With Windows Presentation Foundation (WPF), you can customize an existing control's visual structure and behavior with your own reusable template. Templates can be applied globally to your application, windows and pages, or directly to controls.

When to create a ControlTemplate

Controls have many properties, such as Background, Foreground, and FontFamily. These properties control different aspects of the control's appearance, but the changes that you can make by setting these properties are limited. For example, you can set the Foreground property to blue and FontStyle to italic on a CheckBox. When you want to customize the control's appearance beyond what setting the other properties on the control can do, you create a ControlTemplate.

In most user interfaces, a button has the same general appearance: a rectangle with some text. If you wanted to create a rounded button, you could create a new control that inherits from the button or recreates the functionality of the button. In addition, the new user control would provide the circular visual.

You can avoid creating new controls by customizing the visual layout of an existing control. With a rounded button, you create a ControlTemplate with the desired visual layout.

On the other hand, if you need a control with new functionality, different properties, and new settings, you would create a new UserControl.

```
<Window x:Class="IntroToStylingAndTemplating.Window1"

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:IntroToStylingAndTemplating"
        mc:Ignorable="d"
        Title="Template Intro Sample" SizeToContent="WidthAndHeight"
MinWidth="250">
    <StackPanel Margin="10">
        <Label>Unstyled Button</Label>
        <Button>Button 1</Button>
        <Label>Rounded Button</Label>
        <Button>Button 2</Button>
    </StackPanel>
</Window>
```

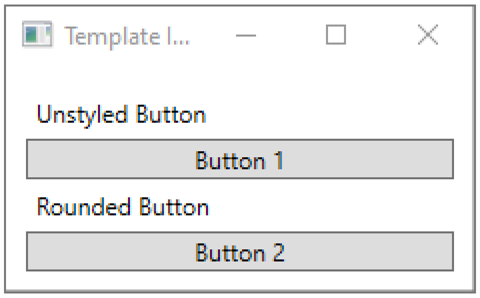


Figure 27: Output

Create a ControlTemplate

The most common way to declare a ControlTemplate is as a resource in the Resources section in a XAML file. Because templates are resources, they obey the same scoping rules that apply to all resources. Put simply, where you declare a template affects where the template can be applied. For example, if you declare the template in the root element of your application definition XAML file, the template can be used anywhere in your application. If you define the template in a window, only the controls in that window can use the template.

To start with, add a Window.Resources element to your *MainWindow.xaml* file:

```
<Window x:Class="IntroToStylingAndTemplating.Window2"

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:IntroToStylingAndTemplating"
        mc:Ignorable="d"
        Title="Template Intro Sample" SizeToContent="WidthAndHeight"
MinWidth="250">
    <Window.Resources>

    </Window.Resources>
    <StackPanel Margin="10">
        <Label>Unstyled Button</Label>
        <Button>Button 1</Button>
        <Label>Rounded Button</Label>
        <Button>Button 2</Button>
    </StackPanel>
</Window>
```

Create a new **<ControlTemplate>** with the following properties set:

```
<ControlTemplate x:Key="roundbutton" TargetType="Button">
    <Grid>
        <Ellipse Fill="{TemplateBinding Background}"
Stroke="{TemplateBinding Foreground}" />
        <ContentPresenter HorizontalAlignment="Center"
VerticalAlignment="Center" />
    </Grid>
</ControlTemplate>
```

TemplateBinding

When you create a new ControlTemplate, you still might want to use the public properties to change the control's appearance. The TemplateBinding markup extension binds a property of an element that is in the ControlTemplate to a public property that is defined by the control. When you use a TemplateBinding, you enable properties on the control to act as parameters to the template. That is, when a property on a control is set, that value is passed on to the element that has the TemplateBinding on it.

Ellipse

Notice that the Fill and Stroke properties of the **<Ellipse>** element are bound to the control's Foreground and Background properties.

ContentPresenter

A <ContentPresenter> element is also added to the template. Because this template is designed for a button, take into consideration that the button inherits from ContentControl. The button presents the content of the element. You can set anything inside of the button, such as plain text or even another control. Both of the following are valid buttons:

```
<Button>My Text</Button>

<!-- and -->

<Button>
    <CheckBox>Checkbox in a button</CheckBox>
</Button>
```

In both of the previous examples, the text and the checkbox are set as the Button.Content property. Whatever is set as the content can be presented through a **<ContentPresenter>**, which is what the template does.

If the ControlTemplate is applied to a ContentControl type, such as a Button, a ContentPresenter is searched for in the element tree. If the ContentPresenter is found, the template automatically binds the control's Content property to the ContentPresenter.

Use the template

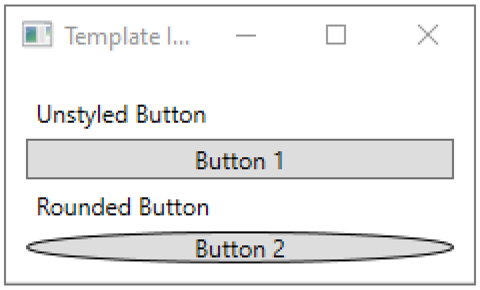
Find the buttons that were declared at the start.

```
<StackPanel Margin="10">
    <Label>Unstyled Button</Label>
    <Button>Button 1</Button>
    <Label>Rounded Button</Label>
    <Button>Button 2</Button>
</StackPanel>
```

Set the second button's Template property to the roundbutton resource:

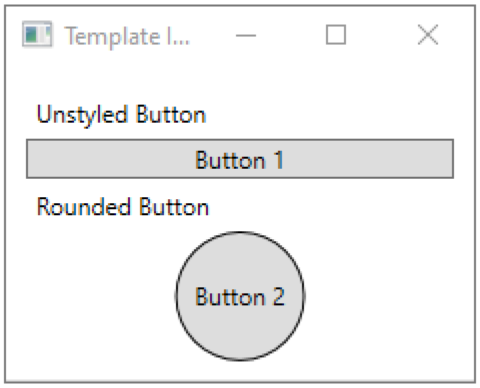
```
<StackPanel Margin="10">
    <Label>Unstyled Button</Label>
    <Button>Button 1</Button>
    <Label>Rounded Button</Label>
    <Button Template="{StaticResource roundbutton}">Button 2</Button>
</StackPanel>
```

If you run the project and look at the result, you'll see that the button has a rounded background.



You may have noticed that the button isn't a circle but is skewed. Because of the way the **<Ellipse>** element works, it always expands to fill the available space. Make the circle uniform by changing the button's width and height properties to the same value:

```
<StackPanel Margin="10">
    <Label>Unstyled Button</Label>
    <Button>Button 1</Button>
    <Label>Rounded Button</Label>
    <Button Template="{StaticResource roundbutton}" Width="65"
Height="65">Button 2</Button>
</StackPanel>
```



3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab.

Lab Task 1

Write a program to change at least 5 different properties of label and button controls using style attribute of XAML. E.g. height, weight, fore ground, back ground, font, margin, alignment, border, corner radius etc.

Lab Task 2

Design a complete user registration form in WPF and apply formatting using style attribute of XAML.

Lab 07
Data templates

Objective:

Purpose of this lab is to familiarize the students with applying data templates to WPF applications.

Activity Outcomes:

The activities provide hands - on practice with the following topics:

- Data Templates

Instructor Note:

As pre-lab activity, read Data Templating Overview - WPF .NET Framework | Microsoft Docs

1) Useful Concepts

The WPF data templating model provides you with great flexibility to define the presentation of your data. WPF controls have built-in functionality to support the customization of data presentation. This topic first demonstrates how to define a DataTemplate and then introduces other data templating features, such as the selection of templates based on custom logic and the support for the display of hierarchical data.

Data binding in Windows Presentation Foundation (WPF) provides a simple and consistent way for apps to present and interact with data. Elements can be bound to data from a variety of data sources in the form of .NET objects and XML. Any ContentControl such as Button and any ItemsControl, such as ListBox and ListView, have built-in functionality to enable flexible styling of single data items or collections of data items. Sort, filter, and group views can be generated on top of the data.

The data binding functionality in WPF has several advantages over traditional models, including inherent support for data binding by a broad range of properties, flexible UI representation of data, and clean separation of business logic from UI.

What is data binding?

Data binding is the process that establishes a connection between the app UI and the data it displays. If the binding has the correct settings and the data provides the proper notifications, when the data changes its value, the elements that are bound to the data reflect changes automatically. Data binding can also mean that if an outer representation of the data in an element changes, then the underlying data can be automatically updated to reflect the change. For example, if the user edits the value in a TextBox element, the underlying data value is automatically updated to reflect that change.

A typical use of data binding is to place server or local configuration data into forms or other UI controls. In WPF, this concept is expanded to include binding a broad range of properties to a variety of data sources. In WPF, dependency properties of elements can be bound to .NET objects (including ADO.NET objects or objects associated with Web Services and Web properties) and XML data.

2) Solved Lab Activites

Sr. No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
Activity 1	***10 mins***	***Low***	***CLO-4***
Activity 2	***10 mins***	***Low***	***CLO-4***
Activity 3	***10 mins***	***Low***	***CLO-4***
Activity 4	***15 mins***	***Medium***	***CLO-4***
Activity 5	***20 mins***	***High***	***CLO-4***

Activity 1:

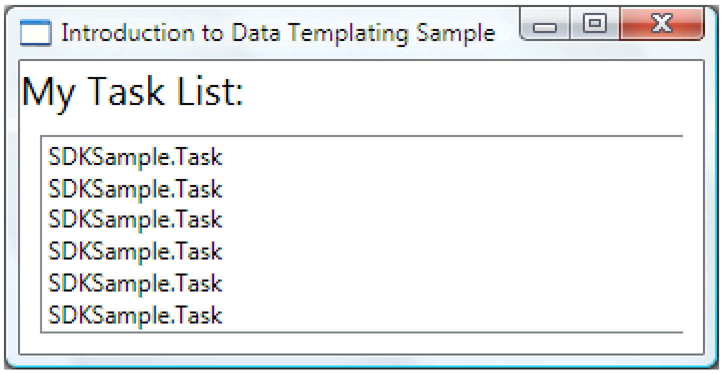
Data binding basics

To demonstrate why DataTemplate is important, let's walk through a data binding example. In this example, we have a ListBox that is bound to a list of Task objects. Each Task object has a TaskName (string), a Description (string), a Priority (int), and a property of type TaskType, which is an Enum with values Home and Work.

```
<Window x:Class="SDKSample.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="clr-namespace:SDKSample"
  Title="Introduction to Data Templating Sample">
  <Window.Resources>
    <local:Tasks x:Key="myTodoList"/>
</Window.Resources>
  <StackPanel>
    <TextBlock Name="blah" FontSize="20" Text="My Task List:"/>
    <ListBox Width="400" Margin="10"
             ItemsSource="{Binding               Source={StaticResource
myTodoList}}"/>
</StackPanel>
</Window>
```

Without a DataTemplate

Without a DataTemplate, our ListBox currently looks like this:



My Task List:

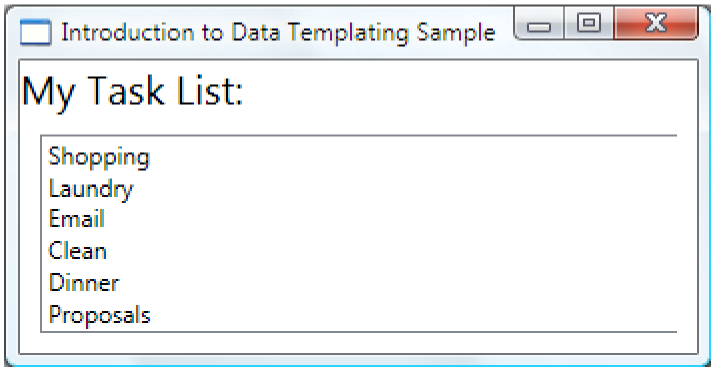
```
SDKSample.Task
SDKSample.Task
SDKSample.Task
SDKSample.Task
SDKSample.Task
SDKSample.Task
```

What's happening is that without any specific instructions, the ListBox by default calls ToString when trying to display the objects in the collection. Therefore, if the Task object overrides the ToString method, then the ListBox displays the string representation of each source object in the underlying collection.

For example, if the Task class overrides the ToString method this way, where name is the field for the TaskName property:

```
public override string ToString()
{
    return name.ToString();
}
```

Then the ListBox looks like the following:



My Task List:

```
Shopping
Laundry
Email
Clean
Dinner
Proposals
```

However, that is limiting and inflexible. Also, if you are binding to XML data, you wouldn't be able to override ToString.

Activity 2:

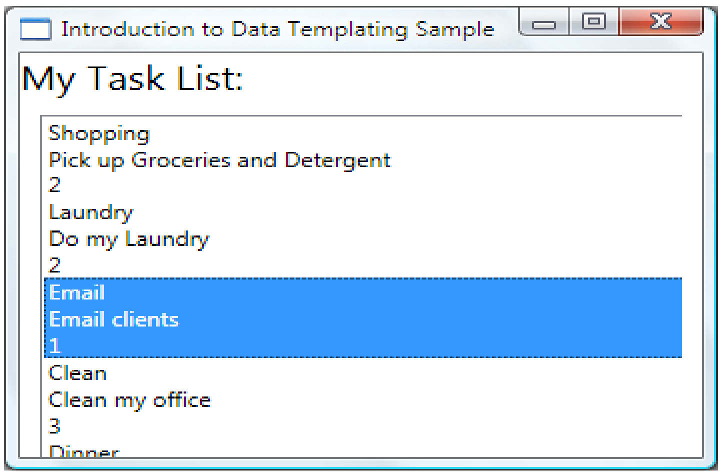
Defining a Simple DataTemplate

The solution is to define a DataTemplate. One way to do that is to set the ItemTemplate property of the ListBox to a DataTemplate. What you specify in your DataTemplate becomes the visual structure of your data object. The following DataTemplate is fairly simple. We are giving instructions that each item appears as three TextBlock elements within a StackPanel. Each TextBlock element is bound to a property of the Task class.

```
<ListBox Width="400" Margin="10"
         ItemsSource="{Binding Source={StaticResource myTodoList}}">
   <ListBox.ItemTemplate>
     <DataTemplate>
       <StackPanel>
         <TextBlock Text="{Binding Path=TaskName}" />
         <TextBlock Text="{Binding Path=Description}"/>
         <TextBlock Text="{Binding Path=Priority}"/>
       </StackPanel>
     </DataTemplate>
   </ListBox.ItemTemplate>
 </ListBox>
```

The underlying data for the examples in this topic is a collection of CLR objects. If you are binding to XML data, the fundamental concepts are the same, but there is a slight syntactic difference. For example, instead of having Path=TaskName, you would set XPath to @TaskName (if TaskName is an attribute of your XML node).

Now our ListBox looks like the following:



My Task List:

```
Shopping
Pick up Groceries and Detergent
2
Laundry
Do my Laundry
2
Email
Email clients
1
Clean
Clean my office
3
Dinner
```

Figure 24: Output

Activity 3:

Creating the DataTemplate as a Resource

In the above example, we defined the DataTemplate inline. It is more common to define it in the resources section so it can be a reusable object, as in the following example:

```
<Window.Resources>
    <DataTemplate x:Key="myTaskTemplate">
       <StackPanel>
            <TextBlock Text="{Binding Path=TaskName}" />
            <TextBlock Text="{Binding Path=Description}"/>
            <TextBlock Text="{Binding Path=Priority}"/>
       </StackPanel>
    </DataTemplate>
</Window.Resources>
```

Now you can use myTaskTemplate as a resource, as in the following example:

```
<ListBox Width="400" Margin="10"
         ItemsSource="{Binding Source={StaticResource myTodoList}}"
         ItemTemplate="{StaticResource myTaskTemplate}"/>
```

Because myTaskTemplate is a resource, you can now use it on other controls that have a property that takes a DataTemplate type. As shown above, for ItemsControl objects, such as the ListBox, it is the ItemTemplate property. For ContentControl objects, it is the ContentTemplate property.

Activity 4:

The DataType Property

The DataTemplate class has a DataType property that is very similar to the TargetType property of the Style class. Therefore, instead of specifying an x:Key for the DataTemplate in the above example, you can do the following:

```
<DataTemplate DataType="{x:Type local:Task}">
  <StackPanel>
    <TextBlock Text="{Binding Path=TaskName}" />
    <TextBlock Text="{Binding Path=Description}"/>
    <TextBlock Text="{Binding Path=Priority}"/>
  </StackPanel>
</DataTemplate>
```

This DataTemplate gets applied automatically to all Task objects. Note that in this case the x:Key is set implicitly. Therefore, if you assign this DataTemplate an x:Key value, you are overriding the implicit x:Key and the DataTemplate would not be applied automatically.

If you are binding a ContentControl to a collection of Task objects, the ContentControl does not use the above DataTemplate automatically. This is because the binding on a ContentControl needs more information to distinguish whether you want to bind to an entire collection or the individual objects. If your ContentControl is tracking the selection of an ItemsControl type, you can set the Path property of the ContentControl binding to "/" to indicate that you are interested in the current item. For an example, see Bind to a Collection and Display Information Based on Selection. Otherwise, you need to specify the DataTemplate explicitly by setting the ContentTemplate property.

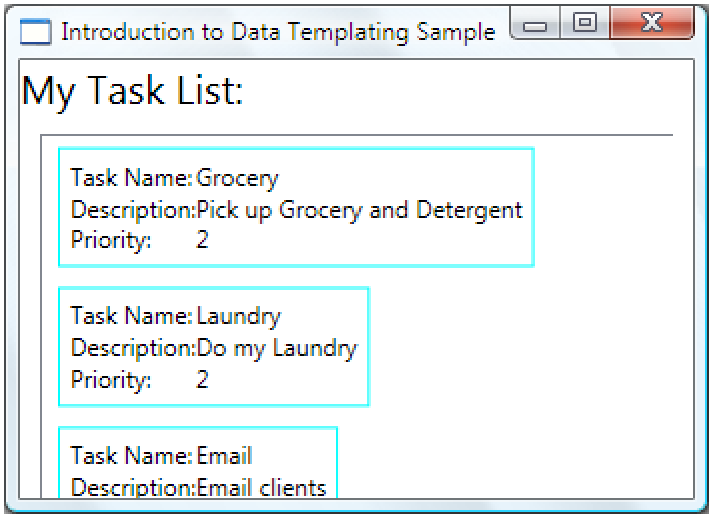
Activity 5:

Adding More to the DataTemplate

Currently the data appears with the necessary information, but there's definitely room for improvement. Let's improve on the presentation by adding a Border, a Grid, and some TextBlock elements that describe the data that is being displayed.

```
<DataTemplate x:Key="myTaskTemplate">
  <Border Name="border" BorderBrush="Aqua" BorderThickness="1"
          Padding="5" Margin="5">
    <Grid>
      <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
      </Grid.RowDefinitions>
      <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
      </Grid.ColumnDefinitions>
      <TextBlock Grid.Row="0" Grid.Column="0" Text="Task Name:"/>
      <TextBlock Grid.Row="0" Grid.Column="1" Text="{Binding Path=TaskName}" />
      <TextBlock Grid.Row="1" Grid.Column="0" Text="Description:"/>
      <TextBlock Grid.Row="1" Grid.Column="1" Text="{Binding Path=Description}"/>
      <TextBlock Grid.Row="2" Grid.Column="0" Text="Priority:"/>
      <TextBlock Grid.Row="2" Grid.Column="1" Text="{Binding Path=Priority}"/>
    </Grid>
  </Border>
</DataTemplate>
```

The following screenshot shows the ListBox with this modified DataTemplate:



My Task List:

Task Name: Grocery
Description:Pick up Grocery and Detergent
Priority: 2

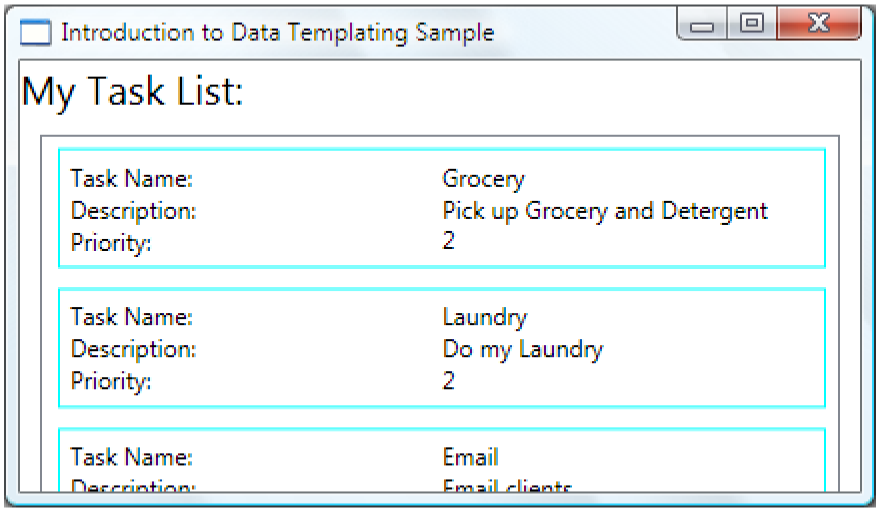
Task Name: Laundry
Description:Do my Laundry
Priority: 2

Task Name: Email
Description:Email clients

We can set HorizontalContentAlignment to Stretch on the ListBox to make sure the width of the items takes up the entire space:

```
<ListBox Width="400" Margin="10"
     ItemsSource="{Binding Source={StaticResource myTodoList}}"
     ItemTemplate="{StaticResource myTaskTemplate}"
     HorizontalContentAlignment="Stretch"/>
```

With the HorizontalContentAlignment property set to Stretch, the ListBox now looks like this:



My Task List:

Task Name:	Grocery
Description:	Pick up Grocery and Detergent
Priority:	2
Task Name:	Laundry
Description:	Do my Laundry
Priority:	2
Task Name:	Email
Description:	Email clients

Use DataTriggers to Apply Property Values

The current presentation does not tell us whether a Task is a home task or an office task. Remember that the Task object has a TaskType property of type TaskType, which is an enumeration with values Home and Work.

In the following example, the DataTrigger sets the BorderBrush of the element named border to Yellow if the TaskType property is TaskType.Home.

```
<DataTemplate x:Key="myTaskTemplate">
<DataTemplate.Triggers>
  <DataTrigger Binding="{Binding Path=TaskType}">
    <DataTrigger.Value>
      <local:TaskType>Home</local:TaskType>
    </DataTrigger.Value>
    <Setter TargetName="border" Property="BorderBrush"
Value="Yellow"/>
  </DataTrigger>
</DataTemplate.Triggers>
</DataTemplate>
```

Our application now looks like the following. Home tasks appear with a yellow border and office tasks appear with an aqua border:

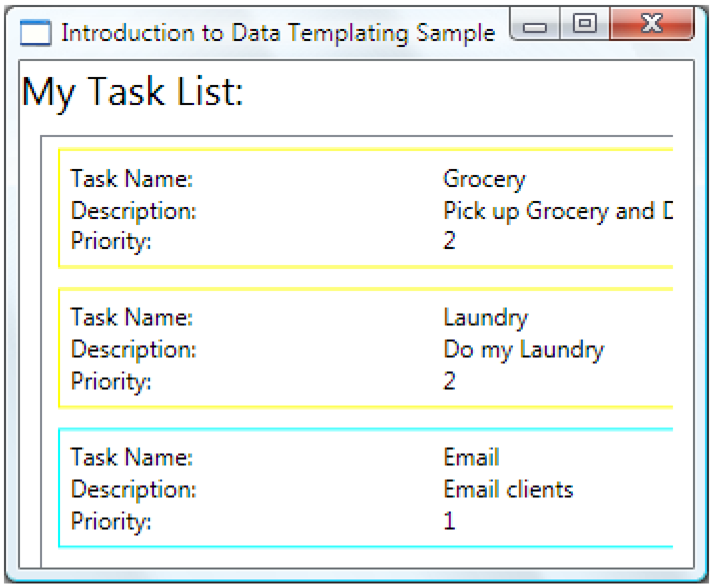

Introduction to Data Templating Sample
My Task List:
Task Name: Grocery
Description: Pick up Grocery and D
Priority: 2
Task Name: Laundry
Description: Do my Laundry
Priority: 2
Task Name: Email
Description: Email clients
Priority: 1

Figure 27: Output

In this example the DataTrigger uses a Setter to set a property value. The trigger classes also have the EnterActions and ExitActions properties that allow you to start a set of actions such as animations. In addition, there is also a MultiDataTrigger class that allows you to apply changes based on multiple data-bound property values.

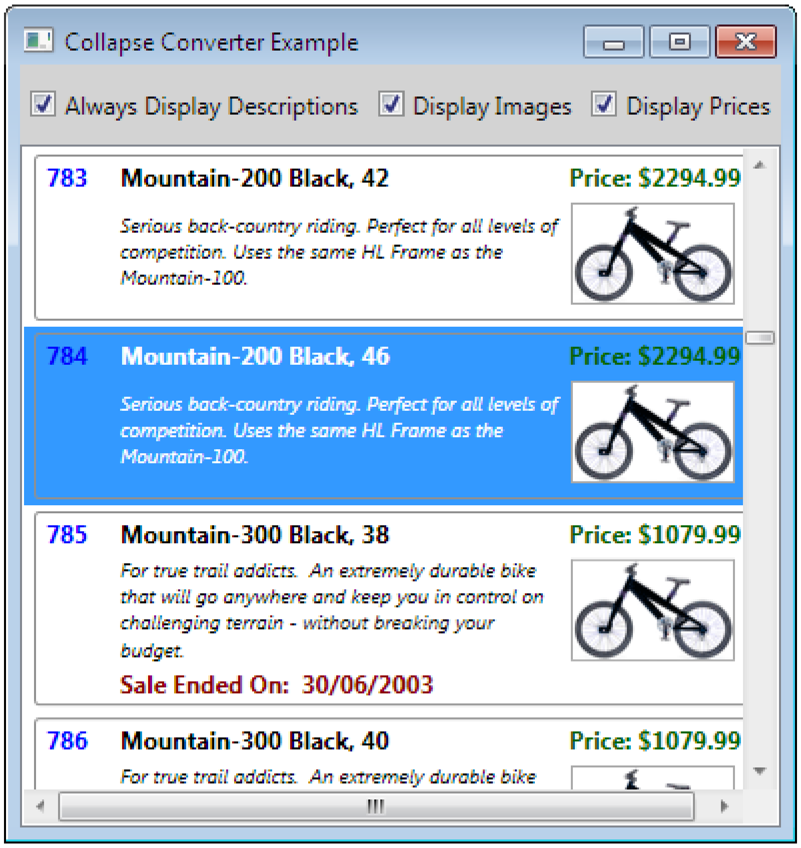
An alternative way to achieve the same effect is to bind the BorderBrush property to the TaskType property and use a value converter to return the color based on the TaskType value. Creating the above effect using a converter is slightly more efficient in terms of performance. Additionally, creating your own converter gives you more flexibility because you are supplying your own logic. Ultimately, which technique you choose depends on your scenario and your preference. For information about how to write a converter, see IValueConverter.

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab.

Lab Task 1

Write a program to change the appearance of data grid using data template as given below:



783 Mountain-200 Black, 42 Price: $2294.99
Serious back-country riding. Perfect for all levels of competition. Uses the same HL Frame as the Mountain-100.

784 Mountain-200 Black, 46 Price: $2294.99
Serious back-country riding. Perfect for all levels of competition. Uses the same HL Frame as the Mountain-100.

785 Mountain-300 Black, 38 Price: $1079.99
For true trail addicts. An extremely durable bike that will go anywhere and keep you in control on challenging terrain - without breaking your budget.
Sale Ended On: 30/06/2003

786 Mountain-300 Black, 40 Price: $1079.99
For true trail addicts. An extremely durable bike

Lab 08
Dependency property
Attached property

Objective:

Purpose of this lab is to familiarize the students with Dependency and Attached Property in WPF Applications.

Activity Outcomes:

The activities provide hands - on practice with the following topics:

- Dependency property
- Attached property

Instructor Note:

As pre-lab activity, read Dependency properties overview - WPF .NET | Microsoft Docs

1) Useful Concepts

Windows Presentation Foundation (WPF) provides a set of services that can be used to extend the functionality of a common language runtime (CLR) property. Collectively, these services are typically referred to as the WPF property system. A property that is backed by the WPF property system is known as a dependency property. This overview describes the WPF property system and the capabilities of a dependency property. This includes how to use existing dependency properties in XAML and in code. This overview also introduces specialized aspects of dependency properties, such as dependency property metadata, and how to create your own dependency property in a custom class.

The main difference is, that the value of a normal .NET property is read directly from a private member in your class, whereas the value of a DependencyProperty is resolved dynamically when calling the GetValue() method that is inherited from DependencyObject.

When you set a value of a dependency property it is not stored in a field of your object, but in a dictionary of keys and values provided by the base class DependencyObject. The key of an entry is the name of the property and the value is the value you want to set.

The advantages of dependency properties are

- Reduced memory footprint

It's a huge dissipation to store a field for each property when you think that over 90% of the properties of a UI control typically stay at its initial values.

Dependency properties solve these problems by only store modified properties in the instance. The default values are stored once within the dependency property.

- Value inheritance

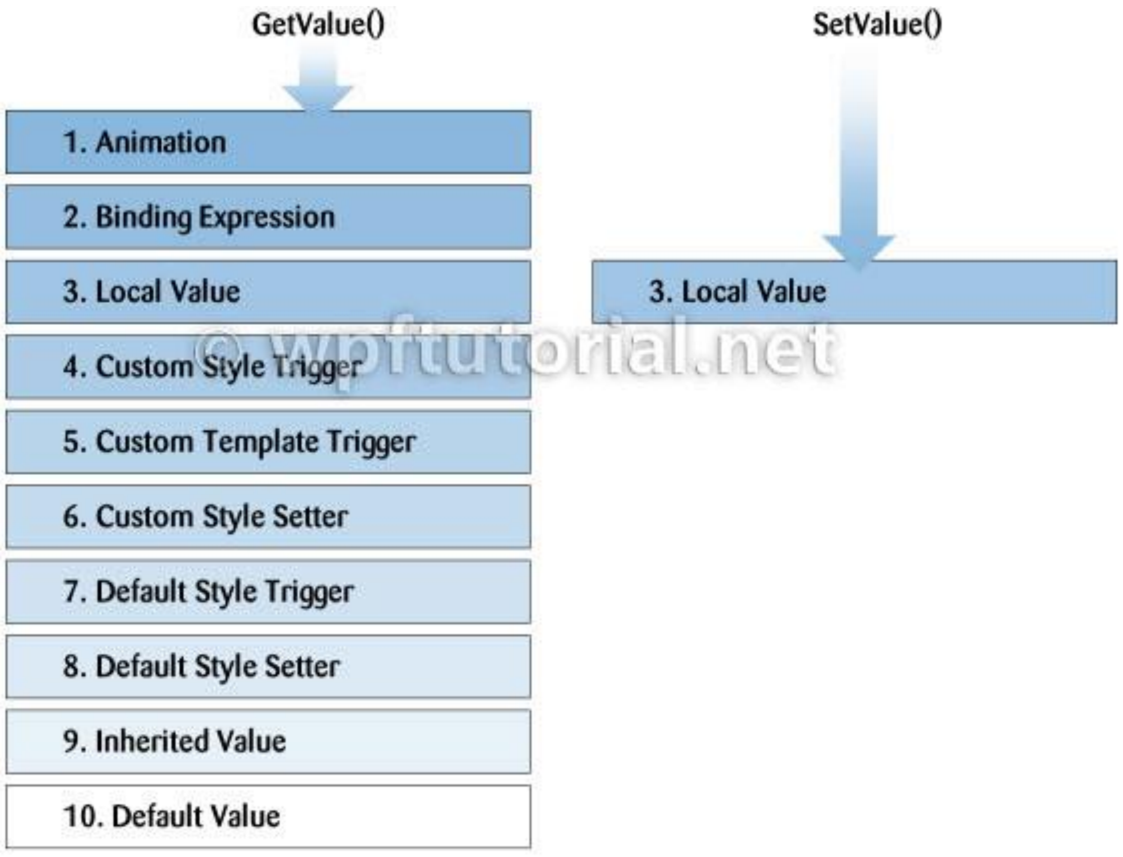
When you access a dependency property the value is resolved by using a value resolution strategy. If no local value is set, the dependency property navigates up the logical tree until it finds a value. When you set the FontSize on the root element it applies to all textblocks below except you override the value.

- Change notification

Dependency properties have a built-in change notification mechanism. By registering a callback in the property metadata you get notified, when the value of the property has been changed. This is also used by the databinding.

Value resolution strategy

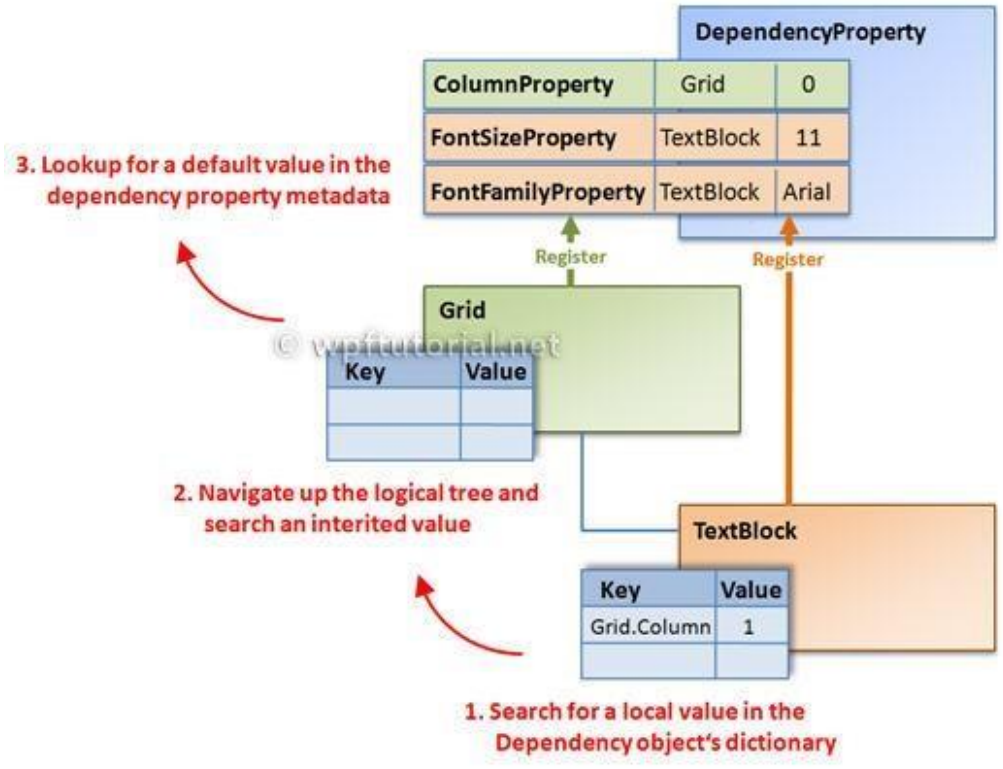
Every time you access a dependency property, it internally resolves the value by following the precedence from high to low. It checks if a local value is available, if not if a custom style trigger is active and continues until it founds a value. At last the default value is always available.



The magic behind it

Each WPF control registers a set of DependencyProperties to the static DependencyProperty class. Each of them consists of a key - that must be unique per type - and a metadata that contain callbacks and a default value.

All types that want to use DependencyProperties must derive from DependencyObject. This baseclass defines a key, value dictionary that contains local values of dependency properties. The key of an entry is the key defined with the dependency property. When you access a dependency property over its .NET property wrapper, it internally calls GetValue(DependencyProperty)to access the value. This method resolves the value by using a value resolution strategy that is explained in detail below. If a local value is available, it reads it directly from the dictionary. If no value is set if goes up the logical tree and searches for an inherited value. If no value is found it takes the default value defined in the property metadata. This sequence is a bit simplified, but it shows the main concept.


DependencyProperty
ColumnProperty
Grid
0
FontSizeProperty
TextBlock
11
FontFamilyProperty
TextBlock
Arial
3. Lookup for a default value in the dependency property metadata
Register
Register
Grid
© wpftutorial.net
Key
Value
2. Navigate up the logical tree and search an interited value
TextBlock
Key
Value
Grid.Column
1
1. Search for a local value in the Dependency object's dictionary

2) Solved Lab Activites

Sr. No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
Activity 1	*15 mins*	*Medium*	*CLO-4*
Activity 2	*15 mins*	*Medium*	*CLO-4*
Activity 3	*15 mins*	*Medium*	*CLO-4*
Activity 4	*15 mins*	*Medium*	*CLO-4*

Activity 1:

How to create a DependencyProperty

To create a DependencyProperty, add a static field of type DepdencyProperty to your type and call DependencyProperty.Register() to create an instance of a dependency property. The name of the DependendyProperty must always end with ...Property. This is a naming convention in WPF.

To make it accessable as a normal .NET property you need to add a property wrapper. This wrapper does nothing else than internally getting and setting the value by using the GetValue() and SetValue() Methods inherited from DependencyObject and passing the DependencyProperty as key.

Important: Do not add any logic to these properties, because they are only called when you set the property from code. If you set the property from XAML the SetValue() method is called directly.

If you are using Visual Studio, you can type propdp and hit 2x tab to create a dependency property.

```
// Dependency Property
public static readonly DependencyProperty CurrentTimeProperty =
     DependencyProperty.Register( "CurrentTime", typeof(DateTime),
     typeof(MyClockControl), new FrameworkPropertyMetadata(DateTime.Now));

// .NET Property wrapper
public DateTime CurrentTime
{
    get { return (DateTime)GetValue(CurrentTimeProperty); }
    set { SetValue(CurrentTimeProperty, value); }
}
```

Each DependencyProperty provides callbacks for change notification, value coercion and validation. These callbacks are registered on the dependency property.

```
new FrameworkPropertyMetadata( DateTime.Now,
                        OnCurrentTimePropertyChanged,
                        OnCoerceCurrentTimeProperty ),
                        OnValidateCurrentTimeProperty );
```

Activity 2:

Value Changed Callback

The change notification callback is a static method, that is called everytime when the value of the TimeProperty changes. The new value is passed in the EventArgs, the object on which the value changed is passed as the source.

```
private static void OnCurrentTimePropertyChanged(DependencyObject source,
        DependencyPropertyChangedEventArgs e)
{
    MyClockControl control = source as MyClockControl;
    DateTime time = (DateTime)e.NewValue;
    // Put some update logic here...
}
```

Coerce Value Callback

The coerce callback allows you to adjust the value if its outside the boundaries without throwing an exception. A good example is a progress bar with a Value set below the Minimum or above the Maximum. In this case we can coerce the value within the allowed boundaries. In the following example we limit the time to be in the past.

```
private static object OnCoerceTimeProperty( DependencyObject sender, object
data )
{
    if ((DateTime)data > DateTime.Now )
    {
        data = DateTime.Now;
    }
    return data;
}
```

Validation Callback

In the validate callback you check if the set value is valid. If you return false, an ArgumentException will be thrown. In our example demand, that the data is an instance of a DateTime.

```
private static bool OnValidateTimeProperty(object data)
{
    return data is DateTime;
}
```

Activity 3:

Readonly DependencyProperties

Some dependency property of WPF controls are readonly. They are often used to report the state of a control, like the IsMouseOver property. Is does not make sense to provide a setter for this value.

Maybe you ask yourself, why not just use a normal .NET property? One important reason is that you cannot set triggers on normal .NET propeties.

Creating a read only property is similar to creating a regular DependencyProperty. Instead of calling DependencyProperty.Register() you call DependencyProperty.RegisterReadonly(). This returns you a DependencyPropertyKey. This key should be stored in a private or protected static readonly field of your class. The key gives you access to set the value from within your class and use it like a normal dependency property.

Second thing to do is registering a public dependency property that is assigned to DependencyPropertyKey.DependencyProperty. This property is the readonly property that can be accessed from external.

```
// Register the private key to set the value
private static readonly DependencyPropertyKey IsMouseOverPropertyKey =
      DependencyProperty.RegisterReadOnly("IsMouseOver",
      typeof(bool), typeof(MyClass),
      new FrameworkPropertyMetadata(false));

// Register the public property to get the value
public static readonly DependencyProperty IsMouseoverProperty =
      IsMouseOverPropertyKey.DependencyProperty;

// .NET Property wrapper
public int IsMouseOver
{
   get { return (bool)GetValue(IsMouseoverProperty); }
   private set { SetValue(IsMouseOverPropertyKey, value); }
}
```

Activity 4:

Attached property

Attached properties are a special kind of DependencyProperties. They allow you to attach a value to an object that does not know anything about this value.

A good example for this concept are layout panels. Each layout panel needs different data to align its child elements. The Canvas needs Top and Left, The DockPanel needs Dock, etc. Since you can write your own layout panel, the list is infinite. So you see, it's not possible to have all those properties on all WPF controls.

The solution are attached properties. They are defined by the control that needs the data from another control in a specific context. For example an element that is aligned by a parent layout panel.

To set the value of an attached property, add an attribute in XAML with a prefix of the element that provides the attached property. To set the the Canvas.Top and Canvas.Left property of a button aligned within a Canvas panel, you write it like this:

```
<Canvas>
    <Button Canvas.Top="20" Canvas.Left="20" Content="Click me!"/>
</Canvas>
```

```
public static readonly DependencyProperty TopProperty =
    DependencyProperty.RegisterAttached("Top",
    typeof(double), typeof(Canvas),
    new FrameworkPropertyMetadata(0d,
        FrameworkPropertyMetadataOptions.Inherits));

public static void SetTop(UIElement element, double value)
{
    element.SetValue(TopProperty, value);
}

public static double GetTop(UIElement element)
{
    return (double)element.GetValue(TopProperty);
}
```

Listen to dependency property changes

If you want to listen to changes of a dependency property, you can subclass the type that defines the property and override the property metadata and pass an PropertyChangedCallback. But an much easier way is to get the DependencyPropertyDescriptor and hookup a callback by calling AddValueChanged()

```
DependencyPropertyDescriptor textDescr = DependencyPropertyDescriptor.
    FromProperty(TextBox.TextProperty, typeof(TextBox));

if (textDescr!= null)
{
    textDescr.AddValueChanged(myTextBox, delegate
    {
        // Add your propery changed logic here...
    });
}
```

How to clear a local value

Because null is also a valid local value, there is the constant DependencyProperty.UnsetValue that describes an unset value.

```
button1.ClearValue( Button.ContentProperty );
```

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab.

Lab Task 1:

Create a WPF Application which contain order information of a purchase. User can choose their existing address from a list. Use dependency and attached property to show the user defined address in shipping address.

Lab Task 2:

When user change the shipping address, ask user to change the existing address (source address) or add new address in the list of addresses.

Lab 10
Language Integrated Query (LINQ) (C#)

Objective:

Purpose of this lab is to familiarize the students with Language Integrated Query (LINQ) in WPF Applications.

Activity Outcomes:

The activities provide hands - on practice with the following topics:

- LINQ Basic
- LINQ with relational database

Instructor Note:

As pre-lab activity, read Language-Integrated Query (LINQ) (C#) | Microsoft Docs

1) Useful Concepts

Language-Integrated Query (LINQ) is the name for a set of technologies based on the integration of query capabilities directly into the C# language. Traditionally, queries against data are expressed as simple strings without type checking at compile time or IntelliSense support. Furthermore, you have to learn a different query language for each type of data source: SQL databases, XML documents, various Web services, and so on. With LINQ, a query is a first-class language construct, just like classes, methods, events. You write queries against strongly typed collections of objects by using language keywords and familiar operators. The LINQ family of technologies provides a consistent query experience for objects (LINQ to Objects), relational databases (LINQ to SQL), and XML (LINQ to XML).

For a developer who writes queries, the most visible "language-integrated" part of LINQ is the query expression. Query expressions are written in a declarative query syntax. By using query syntax, you can perform filtering, ordering, and grouping operations on data sources with a minimum of code. You use the same basic query expression patterns to query and transform data in SQL databases, ADO.NET Datasets, XML documents and streams, and .NET collections.

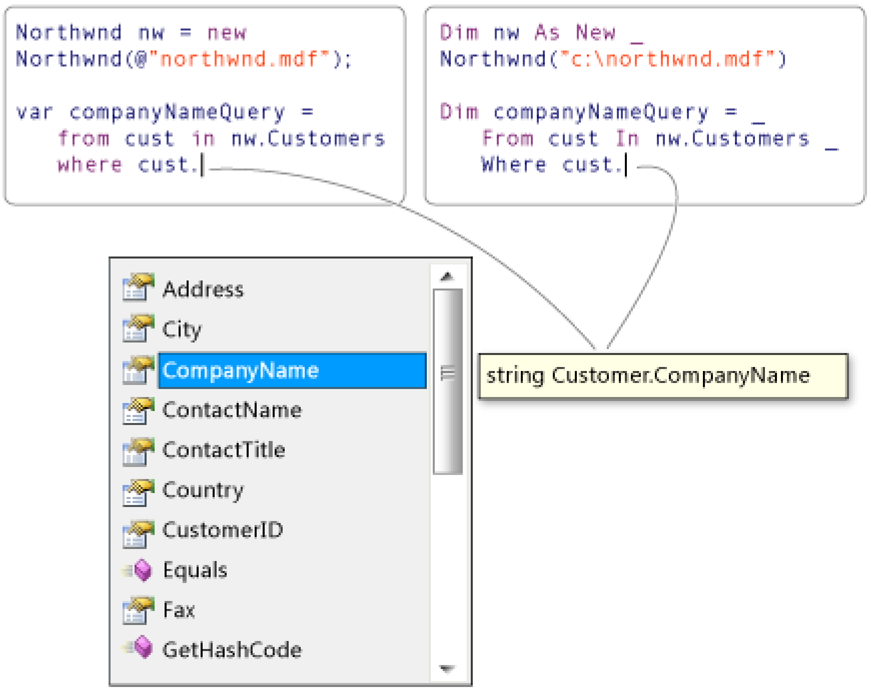
You can write LINQ queries in C# for SQL Server databases, XML documents, ADO.NET Datasets, and any collection of objects that supports IEnumerable or the generic IEnumerable<T> interface. LINQ support is also provided by third parties for many Web services and other database implementations.

The following example shows the complete query operation. The complete operation includes creating a data source, defining the query expression, and executing the query in a foreach statement.

```
// Specify the data source.
int[] scores = { 97, 92, 81, 60 };
```

```
// Define the query expression.
IEnumerable<int> scoreQuery =
    from score in scores
    where score > 80
    select score;

// Execute the query.
foreach (int i in scoreQuery)
{
    Console.Write(i + " ");
}


// Output: 97 92 81
```

The following illustration from Visual Studio shows a partially-completed LINQ query against a SQL Server database in both C# and Visual Basic with full type checking and IntelliSense support:



```
Northwnd nw = new
Northwnd(@"northwnd.mdf");

var companyNameQuery =
    from cust in nw.Customers
    where cust.
```

```
Dim nw As New _
Northwnd("c:\northwnd.mdf")

Dim companyNameQuery = _
    From cust In nw.Customers _
    Where cust.
```

Query expression overview

- Query expressions can be used to query and to transform data from any LINQ-enabled data source. For example, a single query can retrieve data from a SQL database, and produce an XML stream as output.

- The variables in a query expression are all strongly typed, although in many cases you do not have to provide the type explicitly because the compiler can infer it.
- A query is not executed until you iterate over the query variable, for example, in a foreach statement.
- At compile time, query expressions are converted to Standard Query Operator method calls according to the rules set forth in the C# specification. Any query that can be expressed by using query syntax can also be expressed by using method syntax. However, in most cases query syntax is more readable and concise.
- As a rule when you write LINQ queries, we recommend that you use query syntax whenever possible and method syntax whenever necessary. There is no semantic or performance difference between the two different forms. Query expressions are often more readable than equivalent expressions written in method syntax.
- Some query operations, such as Count or Max, have no equivalent query expression clause and must therefore be expressed as a method call. Method syntax can be combined with query syntax in various ways.

2) Solved Lab Activites

Sr. No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
Activity 1	***10 mins***	***Medium***	***CLO-4***
Activity 2	***15 mins***	***Medium***	***CLO-4***
Activity 3	***15 mins***	***Medium***	***CLO-4***

Activity 1:

Three Parts of a Query Operation

All LINQ query operations consist of three distinct actions:

a. Obtain the data source.

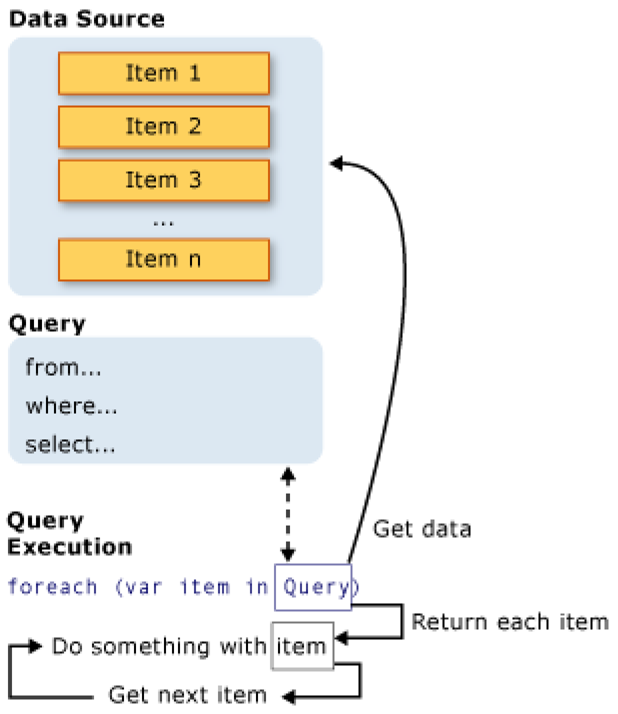
b. Create the query.

c. Execute the query.

The following example shows how the three parts of a query operation are expressed in source code. The example uses an integer array as a data source for convenience; however, the same concepts apply to other data sources also.

```
class IntroToLINQ
{
    static void Main()
    {
        // The Three Parts of a LINQ Query:
        // 1. Data source.
        int[] numbers = new int[7] { 0, 1, 2, 3, 4, 5, 6 };

        // 2. Query creation.
        // numQuery is an IEnumerable<int>
        var numQuery =
            from num in numbers
            where (num % 2) == 0
            select num;

        // 3. Query execution.
        foreach (int num in numQuery)
        {
            Console.Write("{0,1} ", num);
        }
    }
}
```

The following illustration shows the complete query operation. In LINQ, the execution of the query is distinct from the query itself. In other words, you have not retrieved any data just by creating a query variable.



The Data Source

In the previous example, because the data source is an array, it implicitly supports the generic IEnumerable<T> interface. This fact means it can be queried with LINQ. A query is executed in a foreach statement, and foreach requires IEnumerable or IEnumerable<T>. Types that support IEnumerable<T> or a derived interface such as the generic IQueryable<T> are called queryable types.

A queryable type requires no modification or special treatment to serve as a LINQ data source. If the source data is not already in memory as a queryable type, the LINQ provider must represent it as such. For example, LINQ to XML loads an XML document into a queryable XElement type:

```
// Create a data source from an XML document.
// using System.Xml.Linq;
XElement contacts = XElement.Load(@"c:\myContactList.xml");
```

With LINQ to SQL, you first create an object-relational mapping at design time either manually or by using the LINQ to SQL Tools in Visual Studio. You write your queries against the objects, and at run-time LINQ to SQL handles the communication with the database. In the following example, Customers represents a specific table in the database, and the type of the query result, IQueryable<T>, derives from IEnumerable<T>.

```
Northwnd db = new Northwnd(@"c:\northwnd.mdf");
```

```
// Query for customers in London.
IQueryable<Customer> custQuery =
    from cust in db.Customers
    where cust.City == "London"
    select cust;
```

The Query

The query specifies what information to retrieve from the data source or sources. Optionally, a query also specifies how that information should be sorted, grouped, and shaped before it is returned. A query is stored in a query variable and initialized with a query expression. To make it easier to write queries, C# has introduced new query syntax.

The query in the previous example returns all the even numbers from the integer array. The query expression contains three clauses: from, where and select. (If you are familiar with SQL, you will have noticed that the ordering of the clauses is reversed from the order in SQL.) The from clause specifies the data source, the where clause applies the filter, and the select clause specifies the type of the returned elements. For now, the important point is that in LINQ, the query variable itself takes no action and returns no data. It just stores the information that is required to produce the results when the query is executed at some later point.

Query Execution

Deffered Execution

As stated previously, the query variable itself only stores the query commands. The actual execution of the query is deferred until you iterate over the query variable in a foreach statement. This concept is referred to as deferred execution and is demonstrated in the following example:

```
//  Query execution.
foreach (int num in numQuery)
{
    Console.Write("{0,1} ", num);
}
```

The foreach statement is also where the query results are retrieved. For example, in the previous query, the iteration variable num holds each value (one at a time) in the returned sequence.

Because the query variable itself never holds the query results, you can execute it as often as you like. For example, you may have a database that is being updated continually by a separate application. In your application, you could create one query that retrieves the latest data, and you could execute it repeatedly at some interval to retrieve different results every time.

Forcing Immediate Execution

Queries that perform aggregation functions over a range of source elements must first iterate over those elements. Examples of such queries are Count, Max, Average, and First. These execute without an explicit foreach statement because the query itself must use foreach in order to return a result. Note also that these types of queries return a single value, not an IEnumerable collection. The following query returns a count of the even numbers in the source array:

```
var evenNumQuery =
    from num in numbers
    where (num % 2) == 0
    select num;

int evenNumCount = evenNumQuery.Count();
```

To force immediate execution of any query and cache its results, you can call the ToList or ToArray methods.

```

List<int> numQuery2 =
    (from num in numbers
     where (num % 2) == 0
     select num).ToList();

// or like this:
// numQuery3 is still an int[]

var numQuery3 =
    (from num in numbers
     where (num % 2) == 0
     select num).ToArray();
```

You can also force execution by putting the foreach loop immediately after the query expression. However, by calling ToList or ToArray you also cache all the data in a single collection object.

Activity 2:

This example shows how to perform a simple query over a list of Student objects. Each Student object contains some basic information about the student, and a list that represents the student's scores on four examinations.

```
class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int ID { get; set; }
    public GradeLevel? Year { get; set; }
    public List<int> ExamScores { get; set; }

    public Student(string FirstName, string LastName, int ID,
GradeLevel Year, List<int> ExamScores)
    {
        this.FirstName = FirstName;
        this.LastName = LastName;
        this.ID = ID;
        this.Year = Year;
        this.ExamScores = ExamScores;
    }

    public Student(string FirstName, string LastName, int StudentID,
List<int>? ExamScores = null)
    {
        this.FirstName = FirstName;
        this.LastName = LastName;
        ID = StudentID;
        this.ExamScores = ExamScores ??
Enumerable.Empty<int>().ToList();
    }

    public static List<Student> students = new()
    {
        new(
            FirstName: "Terry", LastName: "Adams", ID: 120,
            Year: GradeLevel.SecondYear,
            ExamScores: new() { 99, 82, 81, 79 }
        ),
        new(
            "Fadi", "Fakhouri", 116,
            GradeLevel.ThirdYear,
            new() { 99, 86, 90, 94 }
        ),
        new(
            "Hanying", "Feng", 117,
            GradeLevel.FirstYear,
            new() { 93, 92, 80, 87 }
        ),
        new(
            "Cesar", "Garcia", 114,
                GradeLevel.FourthYear,
                new() { 97, 89, 85, 82 }
            ),
            new(
                "Debra", "Garcia", 115,
                GradeLevel.ThirdYear,
                new() { 35, 72, 91, 70 }
            ),
            new(
                "Hugo", "Garcia", 118,
                GradeLevel.SecondYear,
                new() { 92, 90, 83, 78 }
            ),
            new(
                "Sven", "Mortensen", 113,
                GradeLevel.FirstYear,
                new() { 88, 94, 65, 91 }
            ),
            new(
                "Claire", "O'Donnell", 112,
                GradeLevel.FourthYear,
                new() { 75, 84, 91, 39 }
            ),
            new(
                "Svetlana", "Omelchenko", 111,
                GradeLevel.SecondYear,
                new() { 97, 92, 81, 60 }
            ),
            new(
                "Lance", "Tucker", 119,
                GradeLevel.ThirdYear,
                new() { 68, 79, 88, 92 }
            ),
            new(
                "Michael", "Tucker", 122,
                GradeLevel.FirstYear,
                new() { 94, 92, 91, 91 }
            ),
            new(
                "Eugene", "Zabokritski", 121,
                GradeLevel.FourthYear,
                new() { 96, 85, 91, 60 }
            )
    };
}
```

```
enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};
```

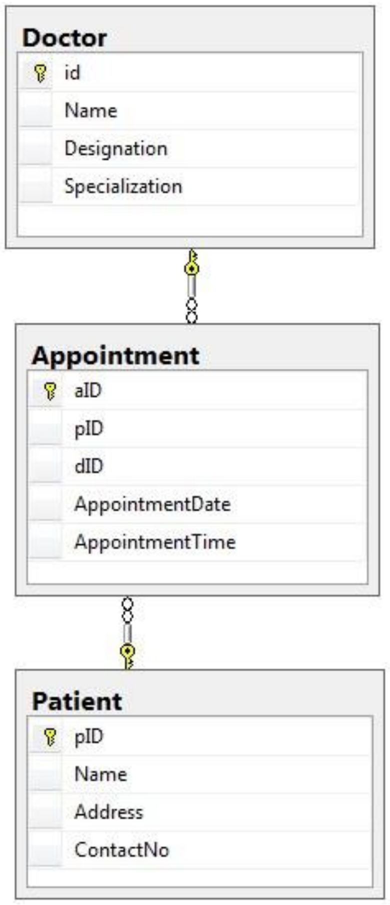
The following query returns the students who received a score of 90 or greater on their first exam.

```
void QueryHighScores(int exam, int score)
{
    var highScores =
        from student in students
        where student.ExamScores[exam] > score
        select new
        {
            Name = student.FirstName,
            Score = student.ExamScores[exam]
        };

    foreach (var item in highScores)
    {
        Console.WriteLine($"{item.Name,-15}{item.Score}");
    }
}

QueryHighScores(1, 90);
```

This query is intentionally simple to enable you to experiment. For example, you can try more conditions in the where clause, or use an orderby clause to sort the results.

Activity 3:

Consider following relational database where Patients get Appointments from Doctors. Patient table contains the information of patients. Doctor table contains the information of doctors. Appointment table is a centre table which has Patient's ID as pID and Doctor's ID as dID.



Write LINQ to show data from Appointments table where match in Doctor and Patient table is found. Using joins and navigation method.

Using Joins

```
from a in Appointments join p
in Patients on a.PID equals
p.PID  join d in Doctors on
a.DID equals d.Id  select new
{
    a.AID,
    Patient = p.Name,
    Doctor = d.Name,
    a.AppointmentDate,
    a.AppointmentTime
}
```

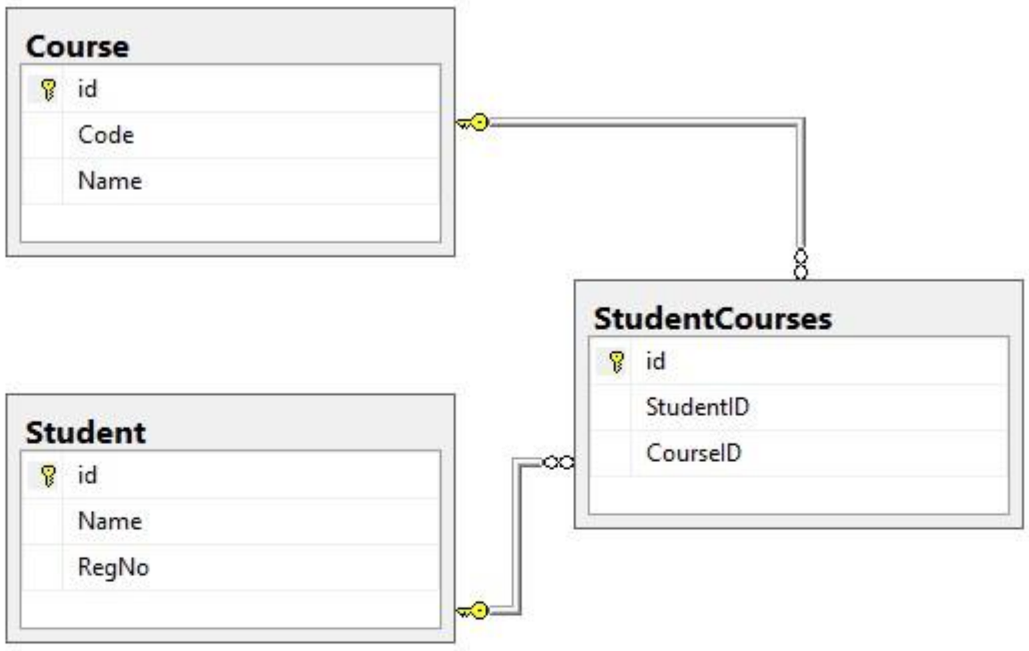
Using Navigation Technique

```
from a in
Appointments
select new {
    a.AID,
    Patient =
    a.Patient.Name,
    Doctor =
    a.Doctor.Name,
    a.AppointmentDate,
    a.AppointmentTime
}
```

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab.

Consider given ERD to perform following activities:



Lab Task 1:

- *Create Tables according to above database diagram and create relations in MS SQL Server or in Database File.*
- *Create WPF forms to display courses and input fields to insert new course record.*

Lab Task 2:

- *Create WPF form to display Students and input fields to insert new student record.*
- *Create WPF form to display Students and their assigned courses. Also, provide with search options where user can search courses or students.*

Lab Task 3:

- *Create WPF form to display all courses which are not assigned to any student.*
- *Create WPF form to display all students which has not been assigned any course*

Lab 11
Developing multithreaded applications
Multithreading using BackgroundWorker, Threadpool

Objective:

In C#, you can write applications that perform multiple tasks at the same time. Tasks with the potential of holding up other tasks can execute on separate threads, a process known as multithreading or free threading.

Applications that use multithreading are more responsive to user input because the user interface stays active as processor-intensive tasks execute on separate threads. Multithreading is also useful when you create scalable applications, because you can add threads as the workload increases.

Activity Outcomes:

The activities provide hands - on practice with the following topics:

- Developing multithreaded applications.
- Multithreading using BackgroundWorker
- Using ThreadPool for multithreading

Instructor Note:

As pre-lab activity, read BackgroundWorker Component Overview - Windows Forms .NET Framework | Microsoft Docs

1) Useful Concepts

The most reliable way to create a multithreaded application is to use the BackgroundWorker component. This class manages a separate thread dedicated to processing the method that you specify.

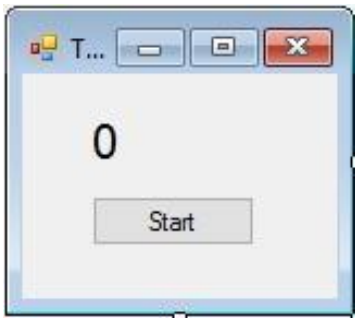
The ThreadPool class provides your application with a pool of worker threads that are managed by the system, allowing you to concentrate on application tasks rather than thread management. If you have short tasks that require background processing, the managed thread pool is an easy way to take advantage of multiple threads. For example, beginning with the .NET Framework 4 you can create Task and Task<TResult> objects, which perform asynchronous tasks on thread pool threads.

2) Solved Lab Activites

Sr. No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
Activity 1	***10 mins***	***Low***	***CLO-4***
Activity 2	***10 mins***	***Low***	***CLO-4***
Activity 3	***10 mins***	***Low***	***CLO-4***
Activity 4	***10 mins***	***Medium***	***CLO-4***
Activity 5	***10 mins***	***Medium***	***CLO-4***

Activity 1:

Create an application containing a label and a button. Label will be used for showing counter from 0 to 100 and increments after each 100 milliseconds. Button will be used to start the counter.



```
private void btnStart_Click(object sender, EventArgs e)
    {
        backgroundWorker1.RunWorkerAsync();
    }

    private void backgroundWorker1_DoWork(object sender,
DoWorkEventArgs e)
    {
        for (int i = 1; i <= 100; i++)
        {
            Console.WriteLine(i);
            System.Threading.Thread.Sleep(100);
backgroundWorker1.ReportProgress(i);
        }
    }
    private void backgroundWorker1_ProgressChanged(object
sender,ProgressChangedEventArgs e)
    {
        this.lblCounter.Text = e.ProgressPercentage.ToString();
    }
```

Activity 2:

In above activity, counter cannot be stopped. It only stopes when counting finishes. Now add feature in the application to start and stop the counter. Button text should be changed based on the counter state (i.e. Start, Stop, Running).

```
private void btnStart_Click(object sender, EventArgs e)
    {
        if (backgroundWorker1.IsBusy)
        {
            backgroundWorker1.CancelAsync(); this.btnStart.Text =
"Start";
        }
        else
        {
            backgroundWorker1.RunWorkerAsync(); this.lblStatus.Text =
"Running"; this.btnStart.Text = "Stop";
        }

    }
    int i = 0; private void backgroundWorker1_DoWork(object sender,
DoWorkEventArgs e)
    {
        for (int i = 0; i <= 100; i++)
        {
            System.Threading.Thread.Sleep(100);
backgroundWorker1.ReportProgress(i); if
(backgroundWorker1.CancellationPending) break;
        }
    }
    private void backgroundWorker1_ProgressChanged(object sender,
ProgressChangedEventArgs e)
    {
        this.lblCounter.Text = e.ProgressPercentage.ToString();
    }
```

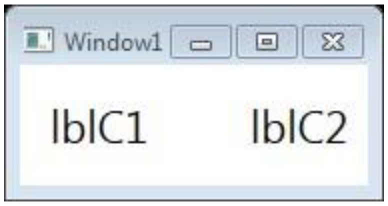
Activity 3:

We can distinguish between thread completed or stopped status. Introduce a label for status which indicates the thread status (i.e. Finished or Stopped).

```
private void btnStart_Click(object sender, EventArgs e)
{
    if (backgroundWorker1.IsBusy)
    {
        backgroundWorker1.CancelAsync(); this.btnStart.Text = "Start";
    }
    else
    {
        backgroundWorker1.RunWorkerAsync(); this.lblStatus.Text =
"Running"; this.btnStart.Text = "Stop";
    }

}
private void backgroundWorker1_DoWork(object sender, DoWorkEventArgs
e)
{
    for (int i = 0; i <= 100; i++)
    {
        System.Threading.Thread.Sleep(100);
backgroundWorker1.ReportProgress(i); if
(backgroundWorker1.CancellationPending)
        {
            e.Cancel = true; break;
        }
    }
}
private void backgroundWorker1_ProgressChanged(object sender,
ProgressChangedEventArgs e)
{
    this.lblCounter.Text = e.ProgressPercentage.ToString();
}

private void backgroundWorker1_RunWorkerCompleted(object sender,
RunWorkerCompletedEventArgs e)
{
    if (e.Cancelled)
        this.lblStatus.Text = "Stopped!";
    else
    {
        this.lblStatus.Text = "Completed!"; this.btnStart.Text =
"Start";
    }
   }
```

Activity 4:

Consider a form contains two labels name lblC1 and lblC2. These labels are used for counter. lblC1 is for counting in ascending order (1 to 100) and lblC2 is for counting in descending order (100 to 1). When form is loaded, both counter starts and display counter increment/decrement after one second. You are required to use BackgroundWorker(s) to achieve above mentioned functionality.



```
BackgroundWorker bwInc, bwDec;

    public Window1()
    {
        InitializeComponent();
        bwInc = new BackgroundWorker();
        bwInc.DoWork += bwInc_DoWork;
        bwInc.ProgressChanged += bwInc_ProgressChanged;
        bwInc.WorkerReportsProgress = true;

        bwDec = new BackgroundWorker();
        bwDec.DoWork += bwDec_DoWork;
        bwDec.ProgressChanged += bwDec_ProgressChanged;
        bwDec.WorkerReportsProgress = true;
    }
    void bwDec_ProgressChanged(object sender, ProgressChangedEventArgs
e)
    {
        this.lblC2.Content = e.ProgressPercentage.ToString();
    }
    void bwDec_DoWork(object sender, DoWorkEventArgs e)
    {
        for (int i = 100; i >= 1; i--)
        {
            System.Threading.Thread.Sleep(1000);
            bwDec.ReportProgress(i);
        }
    }

    void bwInc_ProgressChanged(object sender, ProgressChangedEventArgs
e)
    {
        this.lblC1.Content = e.ProgressPercentage.ToString();
    }
    void bwInc_DoWork(object sender, DoWorkEventArgs e)
    {
        for (int i = 1; i <= 100; i++)
        {
            System.Threading.Thread.Sleep(1000);
            bwInc.ReportProgress(i);
        }
    }
    private void Window_Loaded(object sender, RoutedEventArgs e)
    {
        bwInc.RunWorkerAsync();
        bwDec.RunWorkerAsync();
    }
```

Activity 5:

Use ThreadPool class to display a loop counter between 1 to 10 on console.

```
private void button1_Click(object sender, EventArgs e)
    {
        WaitCallback callBack = new WaitCallback(work);
        ThreadPool.QueueUserWorkItem(callBack, "MyThread");
    }

    private void work(object objName)
    {
        Console.WriteLine("Starting Thread " + objName.ToString());
for (int i = 1; i <= 10; i++)
        {
            Thread.Sleep(100);
            Console.WriteLine(i);
        }
        Console.WriteLine("Ending Thread " + objName.ToString());
    }
```

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab.

Lab Task 1:

Develop a file searching multithreaded application in WPF using BackgroundWorker.

Application should have following options:

- *Choose/Select a file type from checkboxes*
- *Browse/select a directory in which file types should be searched*
- *Search for the files in selected directory and sub-directories*
- *Display the files in a listbox with the complete path*
- *User should be able to stop/start the search anytime*
- *Keep adding the files into the listbox while searching*

Lab 12
Get started with ASP.NET Core MVC

Objective:

Purpose – the purpose of this lab is to learn how to build a MVC application using ASP.net Core. the students shall learn how to handle and add models, views and controllers to their application.

Activity Outcomes:

The activities provide hands - on practice with the following topics:

- Construct a basic ASP.net MVC application
- Maintain model, views and controllers

Instructor Note:

As pre-lab activity, read Get started with ASP.NET Core MVC | Microsoft Docs

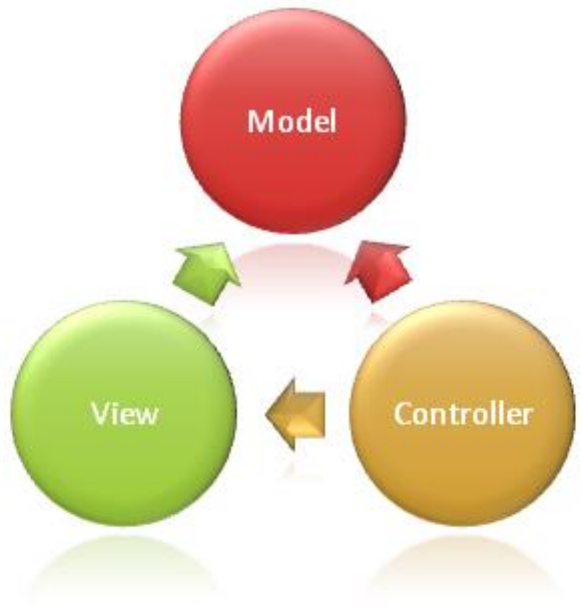
1) Useful Concepts

ASP.NET Core MVC is a rich framework for building web apps and APIs using the Model-View-Controller design pattern.

MVC pattern

The Model-View-Controller (MVC) architectural pattern separates an application into three main groups of components: Models, Views, and Controllers. This pattern helps to achieve separation of concerns. Using this pattern, user requests are routed to a Controller which is responsible for working with the Model to perform user actions and/or retrieve results of queries. The Controller chooses the View to display to the user, and provides it with any Model data it requires.

The following diagram shows the three main components and which ones reference the others:



This delineation of responsibilities helps you scale the application in terms of complexity because it's easier to code, debug, and test something (model, view, or controller) that has a single job. It's more difficult to update, test, and debug code that has dependencies spread across two or more of these three areas. For example, user interface logic tends to change more frequently than business logic. If presentation code and business logic are combined in a single object, an object containing business logic must be modified every time the user interface is changed. This often introduces errors and requires the retesting of business logic after every minimal user interface change.

Note

Both the view and the controller depend on the model. However, the model depends on neither the view nor the controller. This is one of the key benefits of the separation. This separation allows the model to be built and tested independent of the visual presentation.

Model Responsibilities

The Model in an MVC application represents the state of the application and any business logic or operations that should be performed by it. Business logic should be encapsulated in the model, along with any implementation logic for persisting the state of the application. Strongly-typed views typically use ViewModel types designed to contain the data to display on that view. The controller creates and populates these ViewModel instances from the model.

View Responsibilities

Views are responsible for presenting content through the user interface. They use the Razor view engine to embed .NET code in HTML markup. There should be minimal logic within views, and any logic in them should relate to presenting content. If you find the need to perform a great deal of logic in view files in order to display data from a complex model, consider using a View Component, ViewModel, or view template to simplify the view.

Controller Responsibilities

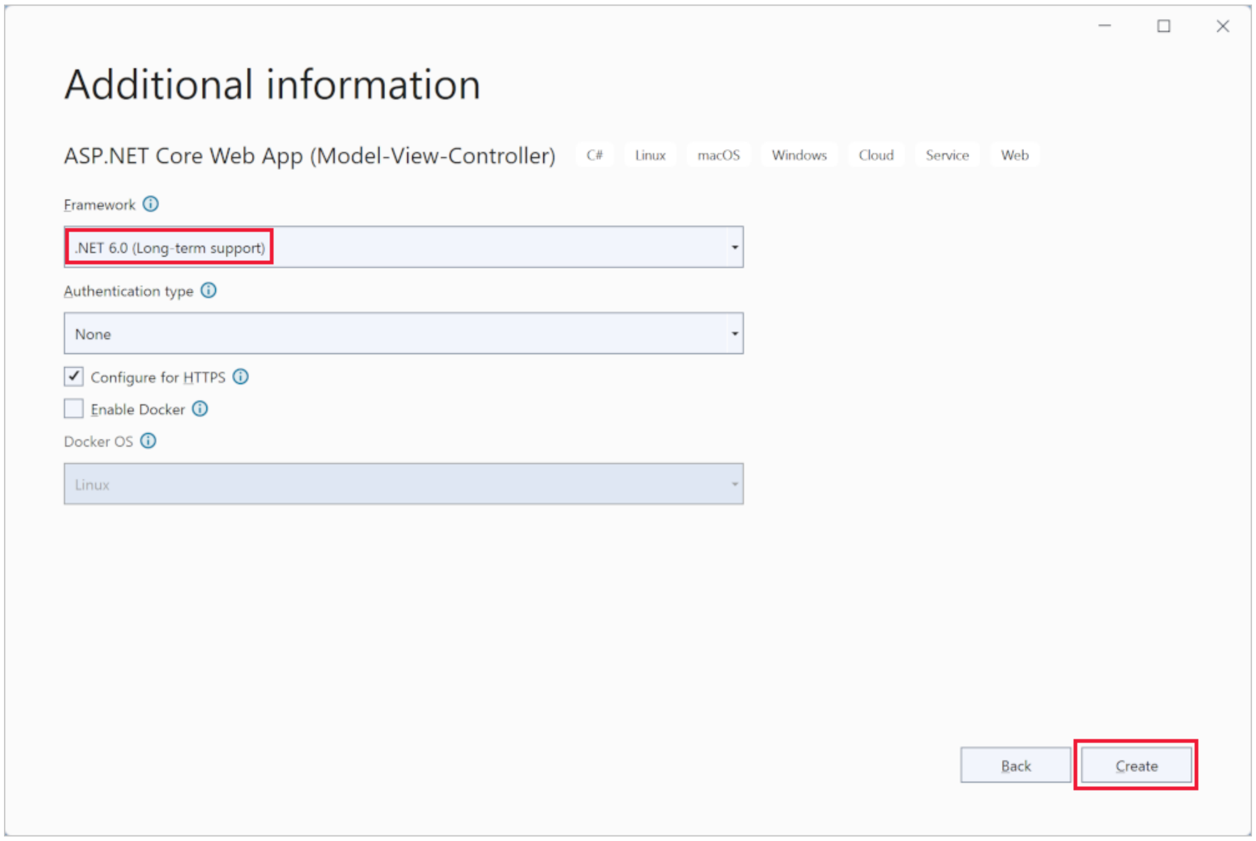
Controllers are the components that handle user interaction, work with the model, and ultimately select a view to render. In an MVC application, the view only displays information; the controller handles and responds to user input and interaction. In the MVC pattern, the controller is the initial entry point, and is responsible for selecting which model types to work with and which view to render (hence its name - it controls how the app responds to a given request).

2) Solved Lab Activites

Sr. No	***Allocated Time***	***Level of Complexity***	***CLO Mapping***
Activity 1	***15 mins***	***Medium***	***CLO-4***
Activity 2	***15 mins***	***Medium***	***CLO-4***
Activity 3	***15 mins***	***Medium***	***CLO-4***
Activity 4	***15 mins***	***Medium***	***CLO-4***

Activity 1: Create a web app

- Start Visual Studio and select **Create a new project**.
- In the **Create a new project** dialog, select **ASP.NET Core Web App (Model-View-Controller)** > **Next**.
- In the **Configure your new project** dialog, enter MvcMovie for **Project name**. It's important to name the project *MvcMovie*. Capitalization needs to match each namespace when code is copied.
- Select **Next**.
- In the **Additional information** dialog, select **.NET 6.0 (Long-term support)**.
- Select **Create**.


Additional information

ASP.NET Core Web App (Model-View-Controller)

Framework: .NET 6.0 (Long-term support)

Authentication type: None

☑ Configure for HTTPS

☐ Enable Docker

Docker OS: Linux

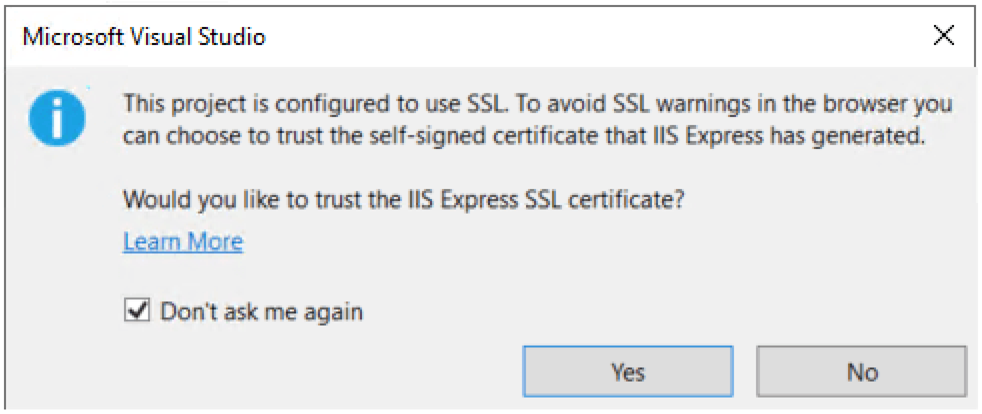
Visual Studio uses the default project template for the created MVC project. The created project:

- Is a working app.
- Is a basic starter project.

Run the app

- Select Ctrl+F5 to run the app without the debugger.

Visual Studio displays the following dialog when a project is not yet configured to use SSL:


Microsoft Visual Studio

This project is configured to use SSL. To avoid SSL warnings in the browser you can choose to trust the self-signed certificate that IIS Express has generated.

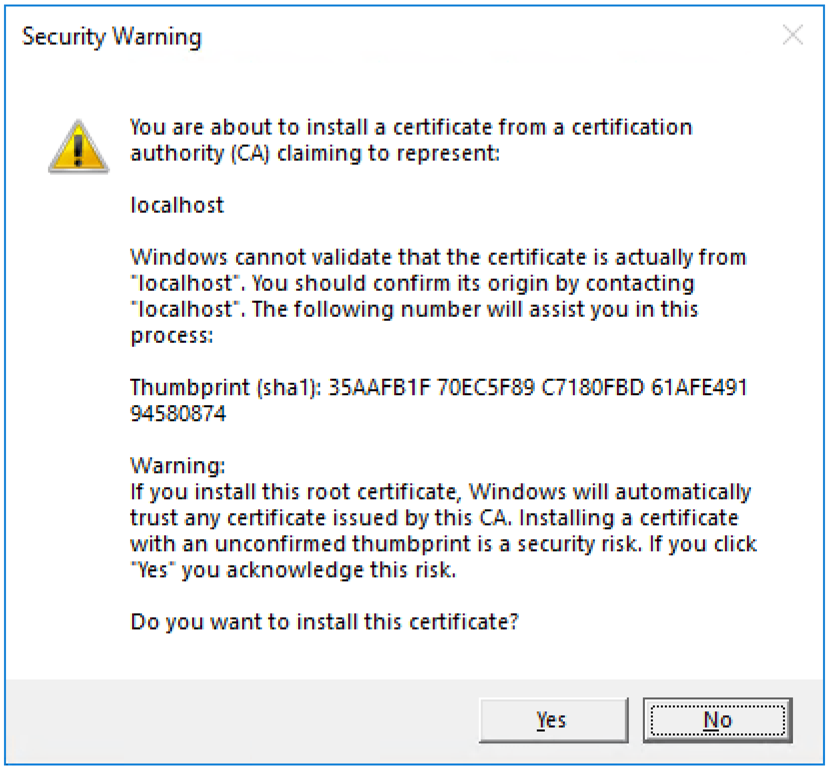
Would you like to trust the IIS Express SSL certificate?

Learn More

☑ Don't ask me again

Select **Yes** if you trust the IIS Express SSL certificate.

The following dialog is displayed:



Security Warning

You are about to install a certificate from a certification authority (CA) claiming to represent:

localhost

Windows cannot validate that the certificate is actually from "localhost". You should confirm its origin by contacting "localhost". The following number will assist you in this process:

Thumbprint (sha1): 35AAFB1F 70EC5F89 C7180FBD 61AFE491 94580874

Warning:
If you install this root certificate, Windows will automatically trust any certificate issued by this CA. Installing a certificate with an unconfirmed thumbprint is a security risk. If you click "Yes" you acknowledge this risk.

Do you want to install this certificate?

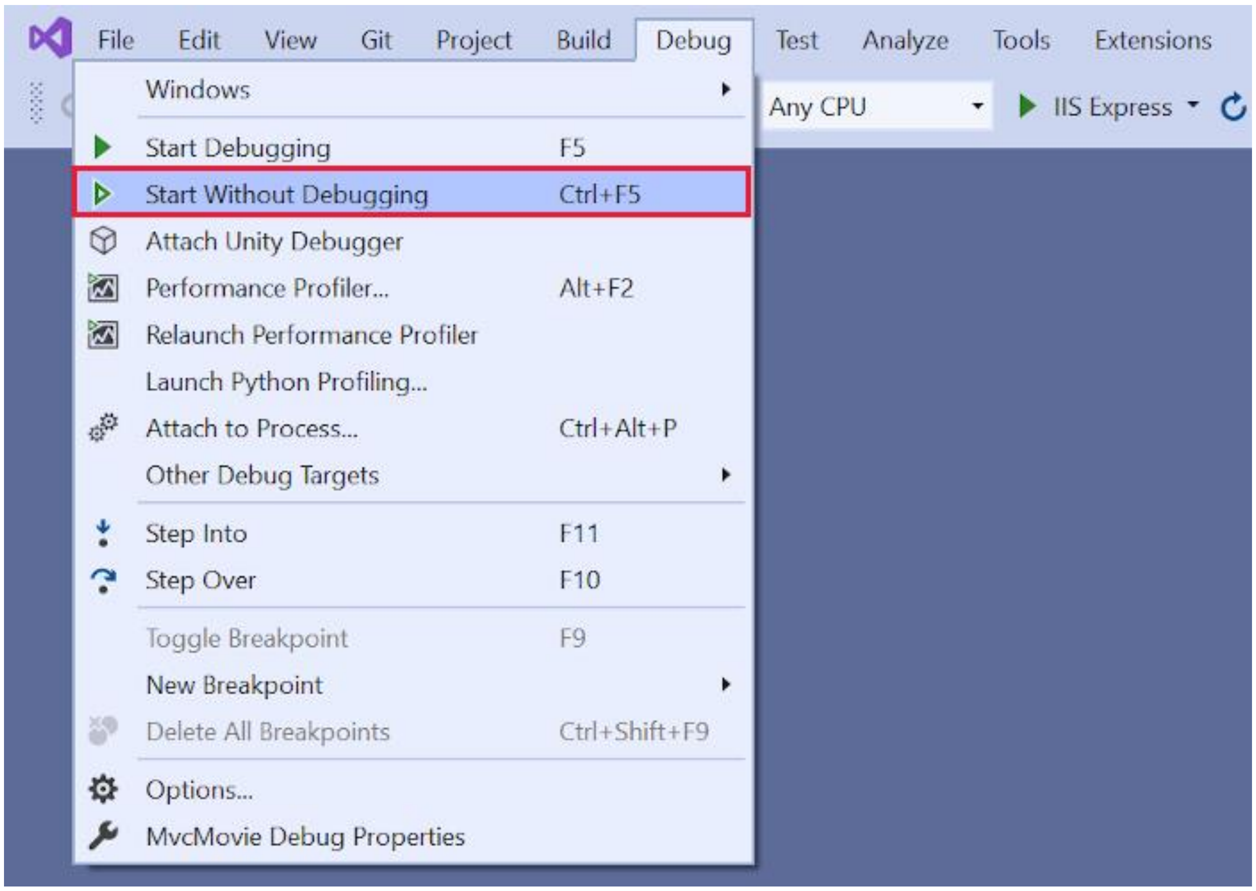
Select **Yes** if you agree to trust the development certificate.

Visual Studio runs the app and opens the default browser.

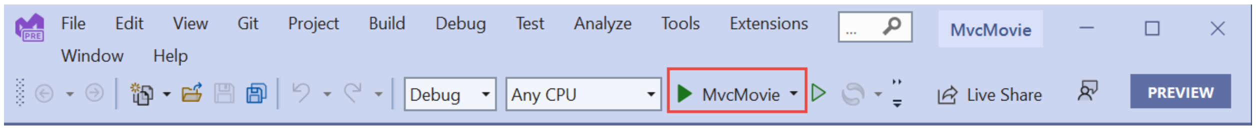
The address bar shows localhost:port# and not something like example.com. The standard hostname for your local computer is localhost. When Visual Studio creates a web project, a random port is used for the web server.
Launching the app without debugging by selecting Ctrl+F5 allows you to:

- Make code changes.
- Save the file.
- Quickly refresh the browser and see the code changes.

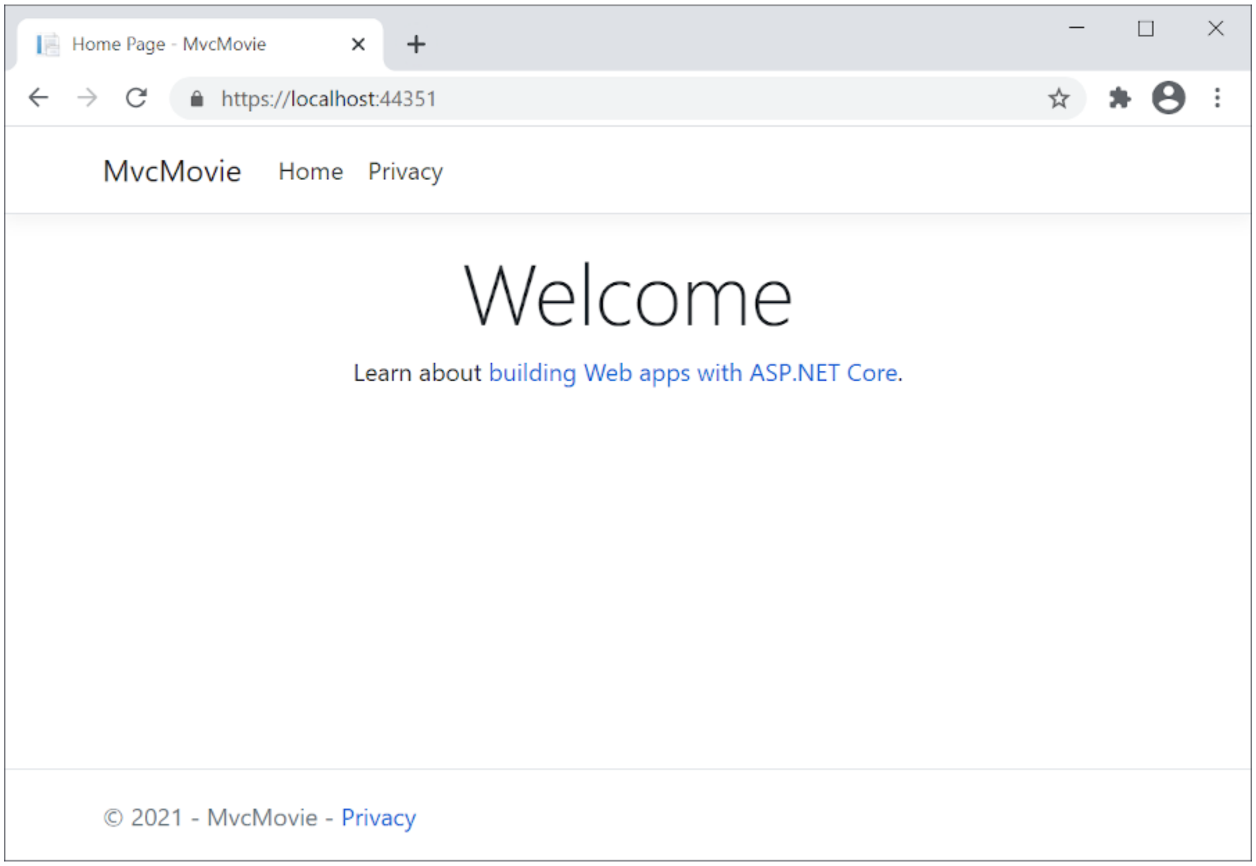
You can launch the app in debug or non-debug mode from the **Debug** menu:



You can debug the app by selecting the **MvcMovie** button in the toolbar:



The following image shows the app:



Welcome

Learn about building Web apps with ASP.NET Core.

© 2021 - MvcMovie - Privacy

Activity 2: add a controller to an ASP.NET Core MVC app

The Model-View-Controller (MVC) architectural pattern separates an app into three main components: **M**odel, **V**iew, and **C**ontroller. The MVC pattern helps you create apps that are more testable and easier to update than traditional monolithic apps.

MVC-based apps contain:

- **M**odels: Classes that represent the data of the app. The model classes use validation logic to enforce business rules for that data. Typically, model objects retrieve and store model state in a database. In this tutorial, a Movie model retrieves movie data from a database, provides it to the view or updates it. Updated data is written to a database.
- **V**iews: Views are the components that display the app's user interface (UI). Generally, this UI displays the model data.
- **C**ontrollers: Classes that:
    - Handle browser requests.
    - Retrieve model data.
    - Call view templates that return a response.

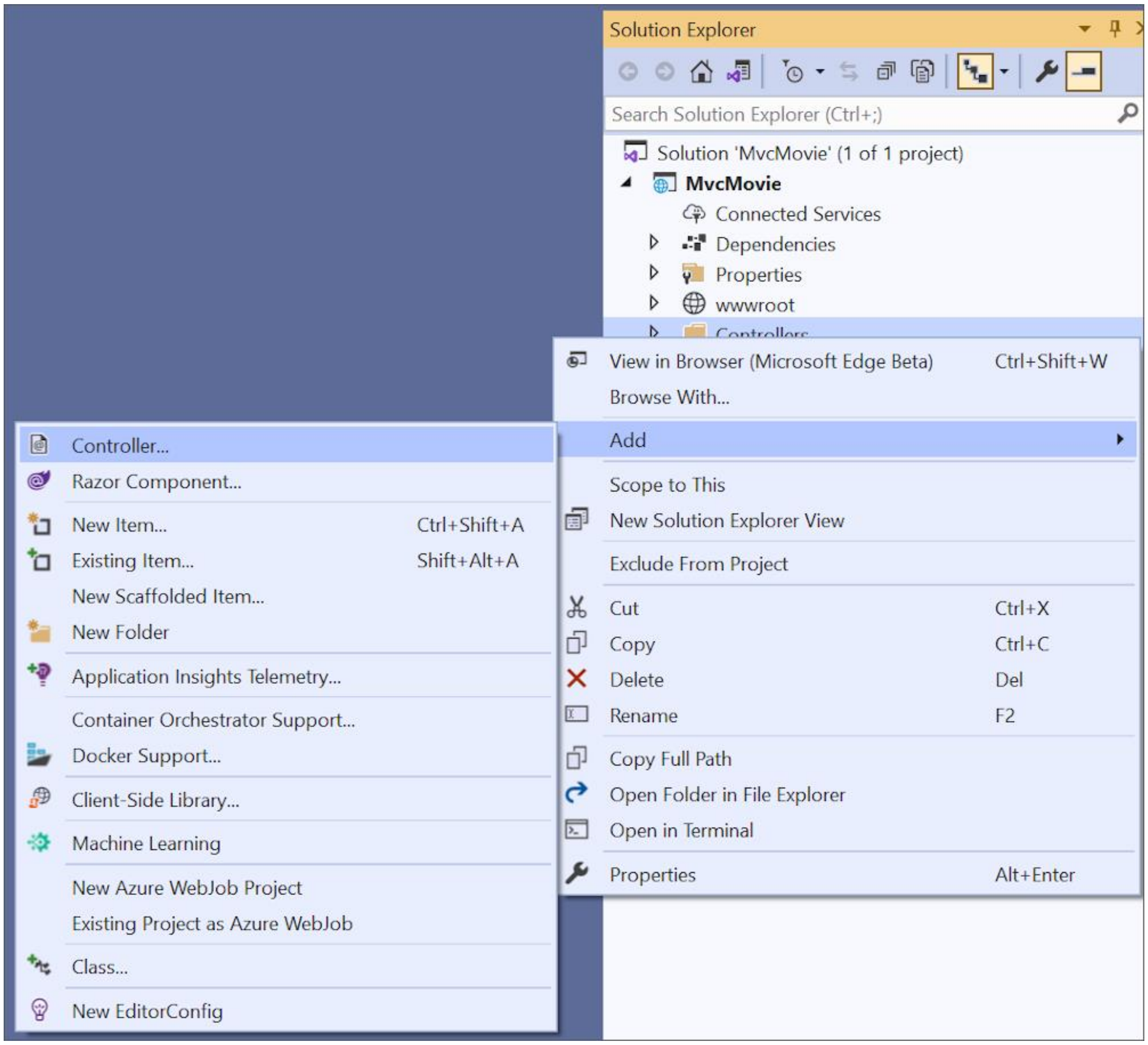
In an MVC app, the view only displays information. The controller handles and responds to user input and interaction. For example, the controller handles URL segments and query-string values, and passes these values to the model. The model might use these values to query the database. For example:

- https://localhost:5001/Home/Privacy: specifies the Home controller and the Privacy action.
- https://localhost:5001/Movies/Edit/5: is a request to edit the movie with ID=5 using the Movies controller and the Edit action, which are detailed later in the tutorial.

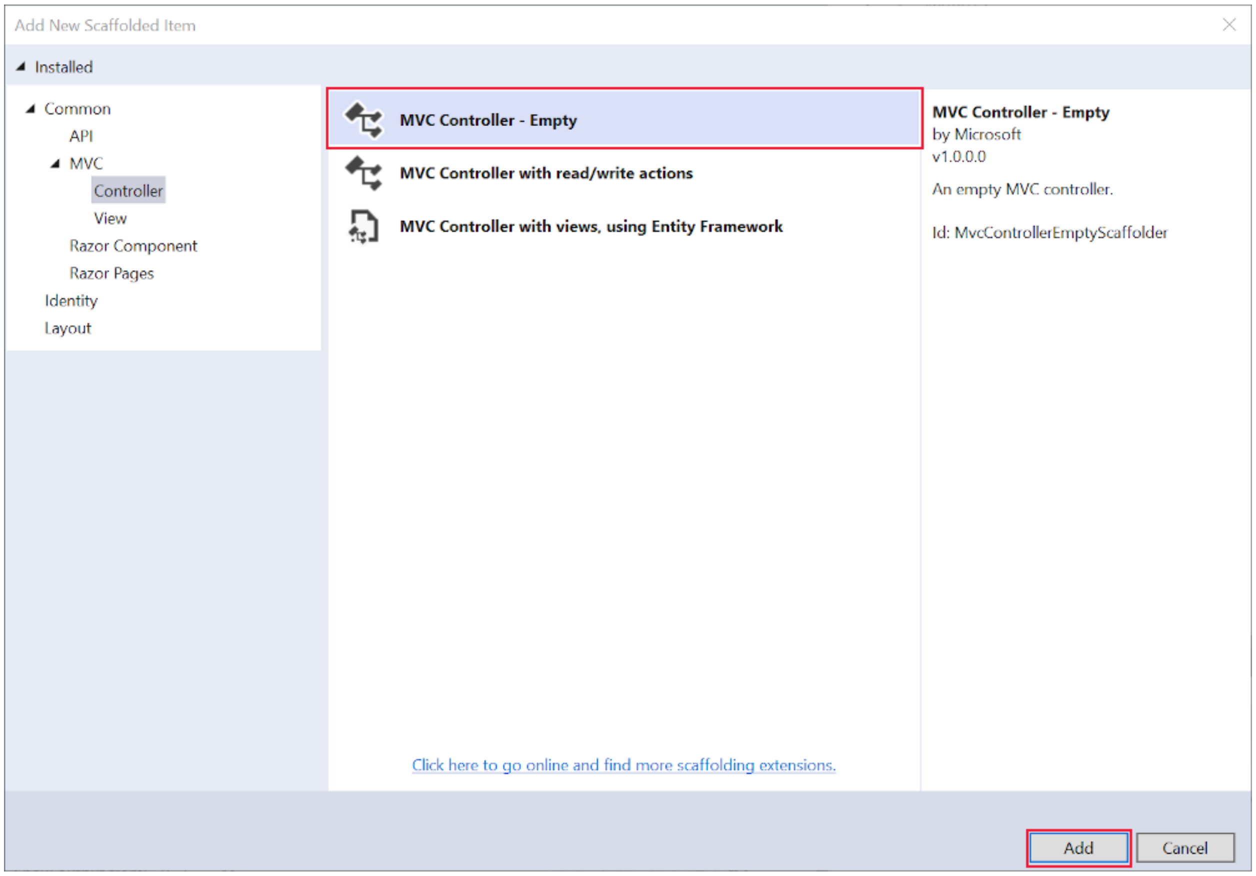
The MVC architectural pattern separates an app into three main groups of components: Models, Views, and Controllers. This pattern helps to achieve separation of concerns: The UI logic belongs in the view. Input logic belongs in the controller. Business logic belongs in the model. This separation helps manage complexity when building an app, because it enables work on one aspect of the implementation at a time without impacting the code of another. For example, you can work on the view code without depending on the business logic code.

Add a controller

In **Solution Explorer**, right-click **Controllers > Add > Controller**.



In the **Add New Scaffolded Item** dialog box, select **MVC Controller - Empty** > **Add**.


Add New Scaffolded Item
Installed
Common
API
MVC
Controller
View
Razor Component
Razor Pages
Identity
Layout
MVC Controller - Empty
MVC Controller with read/write actions
MVC Controller with views, using Entity Framework
Click here to go online and find more scaffolding extensions.
Add
Cancel

MVC Controller - Empty
by Microsoft
v1.0.0.0

An empty MVC controller.

Id: MvcControllerEmptyScaffolder

In the **Add New Item - MvcMovie** dialog, enter HelloWorldController.cs and select **Add**.

Replace the contents of Controllers/HelloWorldController.cs with the following code:

```
using Microsoft.AspNetCore.Mvc;
using System.Text.Encodings.Web;

namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        // 
        // GET: /HelloWorld/

        public string Index()
        {
            return "This is my default action...";
        }

        // 
        // GET: /HelloWorld/Welcome/
        public string Welcome()
        {
            return "This is the Welcome action method...";
        }
    }
}
```

Every public method in a controller is callable as an HTTP endpoint. In the sample above, both methods return a string. Note the comments preceding each method.

An HTTP endpoint:

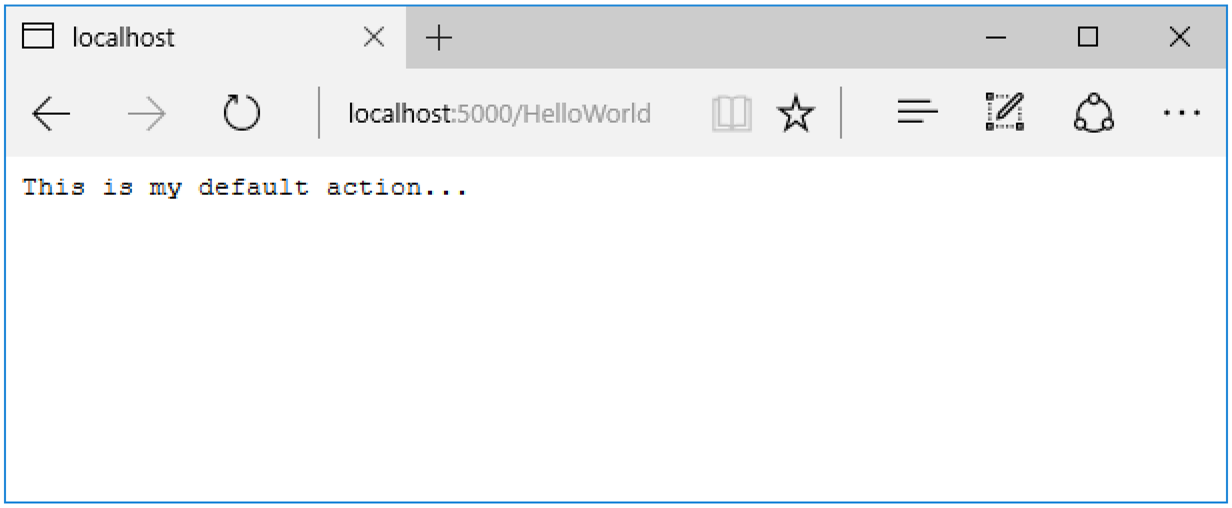
- Is a targetable URL in the web application, such as https://localhost:5001/HelloWorld.
- Combines:
  - The protocol used: HTTPS.
  - The network location of the web server, including the TCP port: localhost:5001.
  - The target URI: HelloWorld.

The first comment states this is an HTTP GET method that's invoked by appending /HelloWorld/ to the base URL.

The second comment specifies an HTTP GET method that's invoked by appending /HelloWorld/Welcome/ to the URL. Later on in the tutorial, the scaffolding engine is used to generate HTTP POST methods, which update data.

Run the app without the debugger.

Append "HelloWorld" to the path in the address bar. The Index method returns a string.



```
This is my default action...
```

MVC invokes controller classes, and the action methods within them, depending on the incoming URL. The default URL routing logic used by MVC, uses a format like this to determine what code to invoke:

/[Controller]/[ActionName]/[Parameters]

The routing format is set in the Program.cs file.

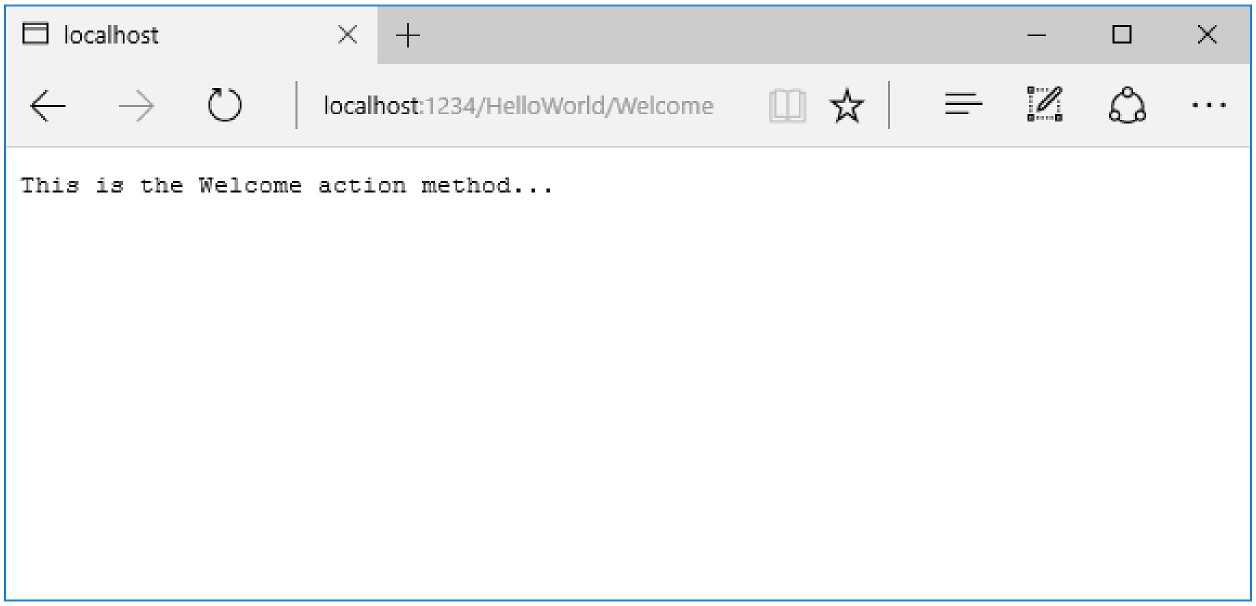
```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

When you browse to the app and don't supply any URL segments, it defaults to the "Home" controller and the "Index" method specified in the template line highlighted above. In the preceding URL segments:

- The first URL segment determines the controller class to run. So localhost:5001/HelloWorld maps to the **HelloWorld** Controller class.
- The second part of the URL segment determines the action method on the class. So localhost:5001/HelloWorld/Index causes the Index method of the HelloWorldController class to run. Notice that you only had to browse to localhost:5001/HelloWorld and the Index method was called by default. Index is the default method that will be called on a controller if a method name isn't explicitly specified.
- The third part of the URL segment (id) is for route data. Route data is explained later in the tutorial.

Browse to: https://localhost:{PORT}/HelloWorld/Welcome. Replace {PORT} with your port number.

The Welcome method runs and returns the string This is the Welcome action method.... For this URL, the controller is HelloWorld and Welcome is the action method. You haven't used the [Parameters] part of the URL yet.



```
This is the Welcome action method...
```

Modify the code to pass some parameter information from the URL to the controller. For example, /HelloWorld/Welcome?name=Rick&numtimes=4.

Change the Welcome method to include two parameters as shown in the following code.

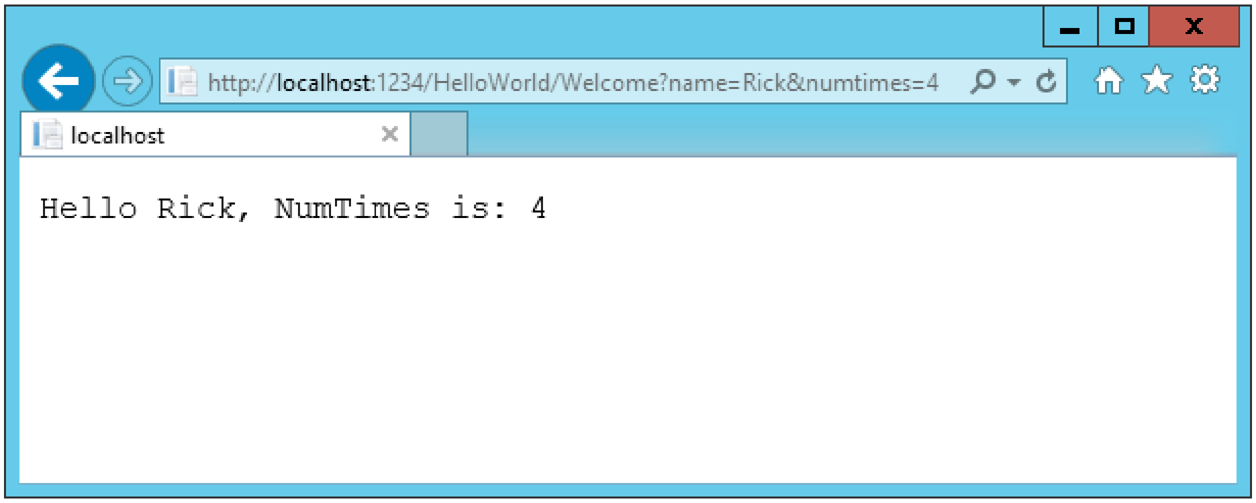
```
// GET: /HelloWorld/Welcome/
// Requires using System.Text.Encodings.Web;
public string Welcome(string name, int numTimes = 1)
{
    return HtmlEncoder.Default.Encode($"Hello {name}, NumTimes is:
{numTimes}");
}
```

The preceding code:

- Uses the C# optional-parameter feature to indicate that the numTimes parameter defaults to 1 if no value is passed for that parameter.
- Uses HtmlEncoder.Default.Encode to protect the app from malicious input, such as through JavaScript.
- Uses Interpolated Strings in $"Hello {name}, NumTimes is: {numTimes}".

Run the app and browse to: https://localhost:{PORT}/HelloWorld/Welcome?name=Rick&numtimes=4. Replace {PORT} with your port number.

Try different values for name and numtimes in the URL. The MVC model binding system automatically maps the named parameters from the query string to parameters in the method.


http://localhost:1234/HelloWorld/Welcome?name=Rick&numtimes=4
localhost

```
Hello Rick, NumTimes is: 4
```

In the previous image:

- The URL segment Parameters isn't used.
- The name and numTimes parameters are passed in the query string.
- The ? (question mark) in the above URL is a separator, and the query string follows.
- The & character separates field-value pairs.

Replace the Welcome method with the following code:

```
public string Welcome(string name, int ID = 1)
{
    return HtmlEncoder.Default.Encode($"Hello {name}, ID: {ID}");
}
```

Run the app and enter the following URL: https://localhost:{PORT}/HelloWorld/Welcome/3?name=Rick

In the preceding URL:

- The third URL segment matched the route parameter id.
- The Welcome method contains a parameter id that matched the URL template in the MapControllerRoute method.
- The trailing ? starts the query string.

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

In the preceding example:

- The third URL segment matched the route parameter id.
- The Welcome method contains a parameter id that matched the URL template in the MapControllerRoute method.
- The trailing ? (in id?) indicates the id parameter is optional.

Activity 3 : add a view to an ASP.NET Core MVC app

In this section, you modify the HelloWorldController class to use Razor view files. This cleanly encapsulates the process of generating HTML responses to a client.

View templates are created using Razor. Razor-based view templates:

- Have a .cshtml file extension.
- Provide an elegant way to create HTML output with C#.

Currently the Index method returns a string with a message in the controller class. In the HelloWorldController class, replace the Index method with the following code:

```
public IActionResult Index()
{
    return View();
}
```

The preceding code:

- Calls the controller's View method.
- Uses a view template to generate an HTML response.

Controller methods:

- Are referred to as *action methods*. For example, the Index action method in the preceding code.
- Generally return an IActionResult or a class derived from ActionResult, not a type like string.

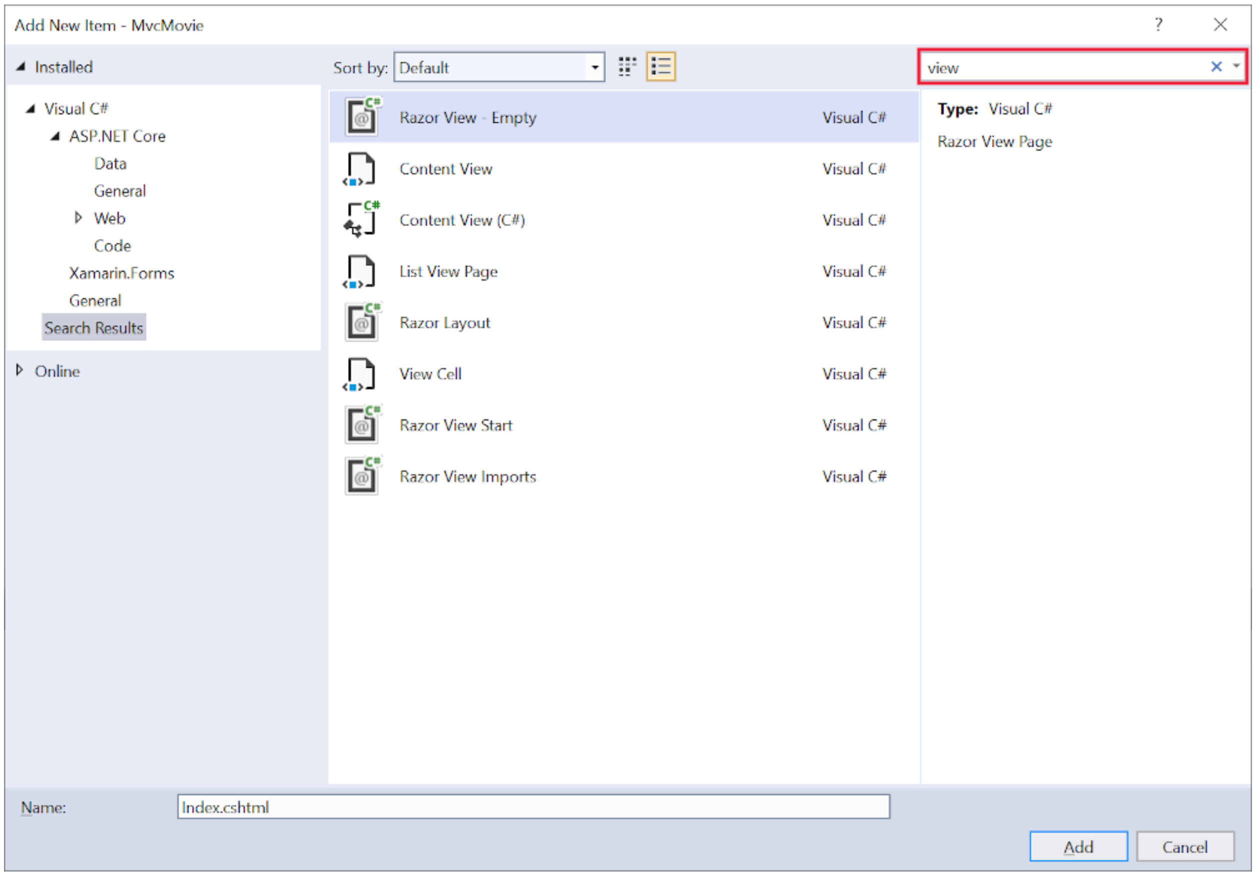
Add a view

Right-click on the *Views* folder, and then **Add** > **New Folder** and name the folder *HelloWorld*.

Right-click on the *Views/HelloWorld* folder, and then **Add > New Item**.

In the **Add New Item - MvcMovie** dialog:

- In the search box in the upper-right, enter *view*
- Select **Razor View - Empty**
- Keep the **Name** box value, Index.cshtml.
- Select **Add**

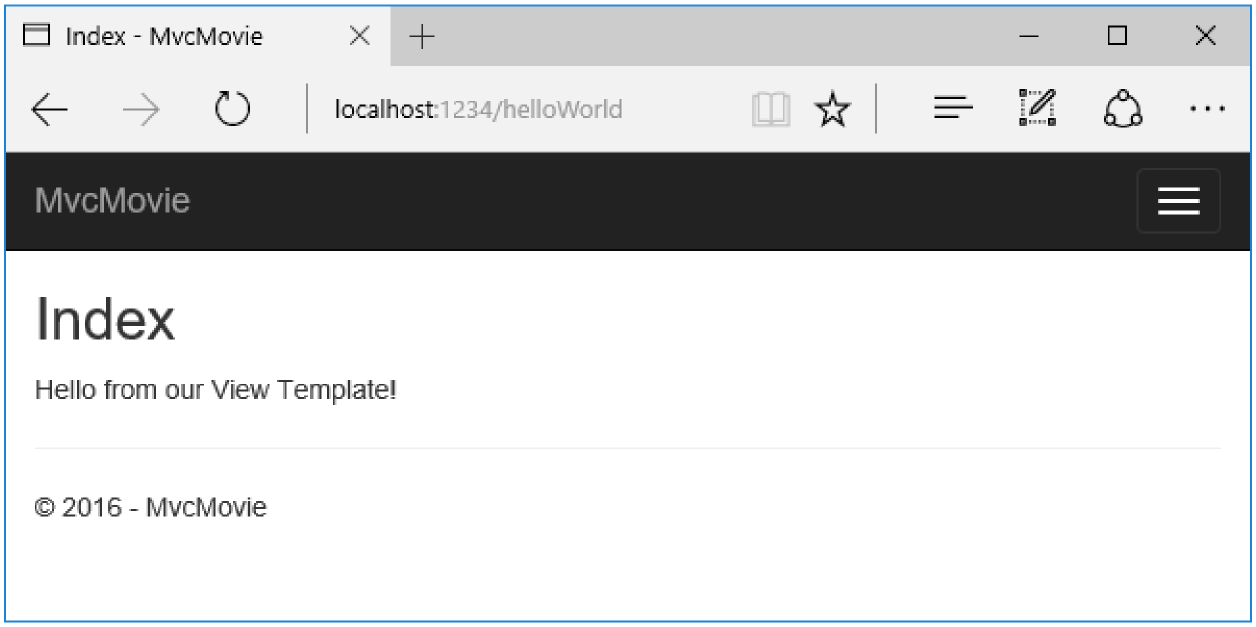


Replace the contents of the Views/HelloWorld/Index.cshtml Razor view file with the following:

```
@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>Hello from our View Template!</p>
```

Navigate to https://localhost:{PORT}/HelloWorld:

- The Index method in the HelloWorldController ran the statement return View();, which specified that the method should use a view template file to render a response to the browser.
- A view template file name wasn't specified, so MVC defaulted to using the default view file. When the view file name isn't specified, the default view is returned. The default view has the same name as the action method, Index in this example. The view template /Views/HelloWorld/Index.cshtml is used.
- The following image shows the string "Hello from our View Template!" hard-coded in the view:



Index

Hello from our View Template!

© 2016 - MvcMovie

Change views and layout pages

Select the menu links **MvcMovie**, **Home**, and **Privacy**. Each page shows the same menu layout. The menu layout is implemented in the Views/Shared/_Layout.cshtml file.

Open the Views/Shared/_Layout.cshtml file.

Layout templates allow:

- Specifying the HTML container layout of a site in one place.
- Applying the HTML container layout across multiple pages in the site.

Find the @RenderBody() line. RenderBody is a placeholder where all the view-specific pages you create show up, *wrapped* in the layout page. For example, if you select the **Privacy** link, the Views/Home/Privacy.cshtml view is rendered inside the RenderBody method.

Change the title, footer, and menu link in the layout file

Replace the content of the Views/Shared/_Layout.cshtml file with the following markup. The changes are highlighted:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-
scale=1.0" />
    <title>@ViewData["Title"] - Movie App</title>
    <link rel="stylesheet"
href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
```

```
    <link rel="stylesheet" href="~/css/site.css" asp-append-
version="true" />
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm
navbar-light bg-white border-bottom box-shadow mb-3">
            <div class="container-fluid">
                <a class="navbar-brand" asp-area="" asp-
controller="Movies" asp-action="Index">Movie App</a>
                <button class="navbar-toggler" type="button" data-bs-
toggle="collapse" data-bs-target=".navbar-collapse" aria-
controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle
navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex
justify-content-between">
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area=""
asp-controller="Home" asp-action="Index">Home</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area=""
asp-controller="Home" asp-action="Privacy">Privacy</a>
                        </li>
                    </ul>
                </div>
            </div>
        </nav>
    </header>
    <div class="container">
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>

    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2021 - Movie App - <a asp-area="" asp-
controller="Home" asp-action="Privacy">Privacy</a>
        </div>
    </footer>
    <script src="~/lib/jquery/dist/jquery.min.js"></script>
    <script
src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>
    @await RenderSectionAsync("Scripts", required: false)
</body>
</html>
```

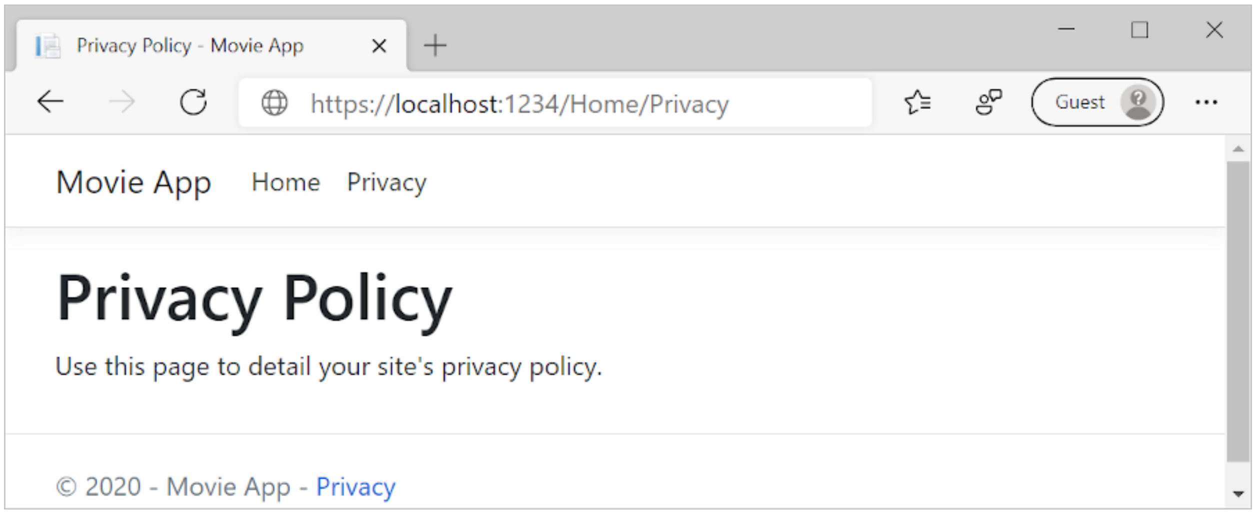
The preceding markup made the following changes:

- Three occurrences of MvcMovie to Movie App.
- The anchor element <a class="navbar-brand" asp-area="" asp-controller="Home" asp-action="Index">MvcMovie</a> to <a class="navbar-brand" asp-controller="Movies" asp-action="Index">Movie App</a>.

In the preceding markup, the asp-area="" anchor Tag Helper attribute and attribute value was omitted because this app isn't using Areas.

Note: The Movies controller hasn't been implemented. At this point, the Movie App link isn't functional.

Save the changes and select the **Privacy** link. Notice how the title on the browser tab displays **Privacy Policy - Movie App** instead of **Privacy Policy - MvcMovie**


Privacy Policy - Movie App
https://localhost:1234/Home/Privacy
Guest
Movie App
Home
Privacy

Privacy Policy

Use this page to detail your site's privacy policy.

© 2020 - Movie App - Privacy

Select the **Home** link.

Notice that the title and anchor text display **Movie App**. The changes were made once in the layout template and all pages on the site reflect the new link text and new title.

Examine the Views/_ViewStart.cshtml file:

```
@{
    Layout = "_Layout";
}
```

The Views/_ViewStart.cshtml file brings in the Views/Shared/_Layout.cshtml file to each view. The Layout property can be used to set a different layout view, or set it to null so no layout file will be used.
Open the Views/HelloWorld/Index.cshtml view file.

Change the title and <h2> element as highlighted in the following:

```
@{
    ViewData["Title"] = "Movie List";
}

<h2>My Movie List</h2>

<p>Hello from our View Template!</p>
```

The title and <h2> element are slightly different so it's clear which part of the code changes the display.

ViewData["Title"] = "Movie List"; in the code above sets the Title property of the ViewData dictionary to "Movie List". The Title property is used in the <title> HTML element in the layout page:

```
<title>@ViewData["Title"] - Movie App</title>
```

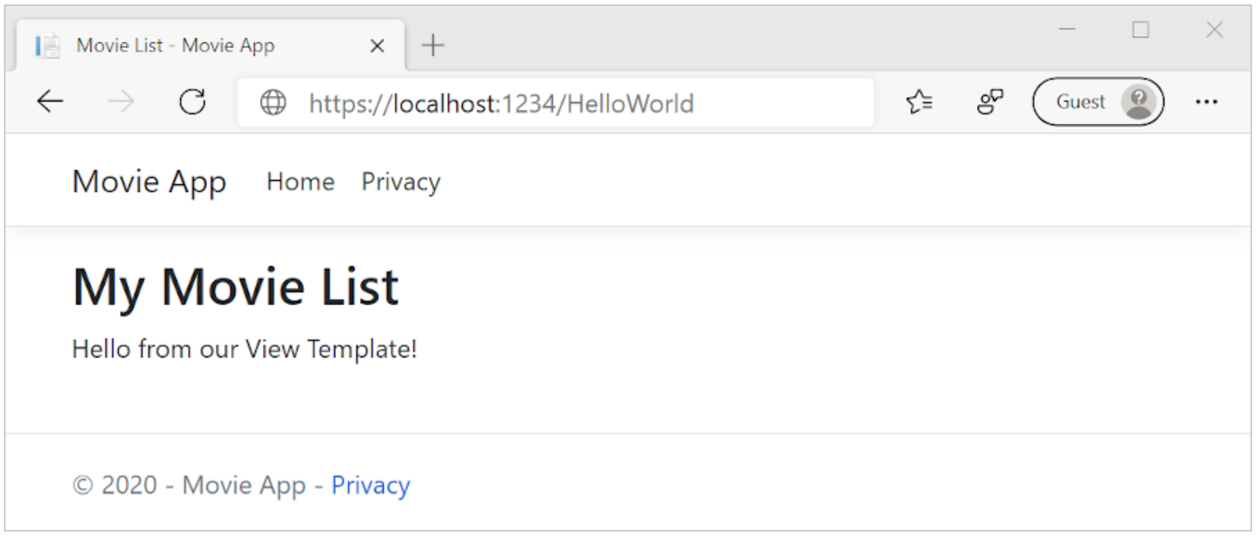
Save the change and navigate to https://localhost:{PORT}/HelloWorld.

Notice that the following have changed:

- Browser title.
- Primary heading.
- Secondary headings.

If there are no changes in the browser, it could be cached content that is being viewed. Press Ctrl+F5 in the browser to force the response from the server to be loaded. The browser title is created with ViewData["Title"] we set in the Index.cshtml view template and the additional "- Movie App" added in the layout file.

The content in the Index.cshtml view template is merged with the Views/Shared/_Layout.cshtml view template. A single HTML response is sent to the browser. Layout templates make it easy to make changes that apply across all of the pages in an app.



My Movie List

Hello from our View Template!

© 2020 - Movie App - Privacy

The small bit of "data", the "Hello from our View Template!" message, is hard-coded however. The MVC application has a "V" (view), a "C" (controller), but no "M" (model) yet.

Passing Data from the Controller to the View

Controller actions are invoked in response to an incoming URL request. A controller class is where the code is written that handles the incoming browser requests. The controller retrieves data from a data source and decides what type of response to send back to the browser. View templates can be used from a controller to generate and format an HTML response to the browser.

Controllers are responsible for providing the data required in order for a view template to render a response.

View templates should **not**:

- Do business logic
- Interact with a database directly.

A view template should work only with the data that's provided to it by the controller. Maintaining this "separation of concerns" helps keep the code:

- Clean.
- Testable.
- Maintainable.

Currently, the Welcome method in the HelloWorldController class takes a name and a ID parameter and then outputs the values directly to the browser.

Rather than have the controller render this response as a string, change the controller to use a view template instead. The view template generates a dynamic response, which means that appropriate data must be passed from the controller to the view to generate the response. Do this by having the controller put the dynamic data (parameters) that the view template needs in a ViewData dictionary. The view template can then access the dynamic data.

In HelloWorldController.cs, change the Welcome method to add a Message and NumTimes value to the ViewData dictionary.

The ViewData dictionary is a dynamic object, which means any type can be used. The ViewData object has no defined properties until something is added. The MVC model binding system automatically maps the named parameters name and numTimes from the query string to parameters in the method. The complete HelloWorldController:

```
using Microsoft.AspNetCore.Mvc;
using System.Text.Encodings.Web;

namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }

        public IActionResult Welcome(string name, int numTimes = 1)
        {
            ViewData["Message"] = "Hello " + name;
            ViewData["NumTimes"] = numTimes;

            return View();
        }
    }
}
```

The ViewData dictionary object contains data that will be passed to the view.

Create a Welcome view template named Views/HelloWorld/Welcome.cshtml.

You'll create a loop in the Welcome.cshtml view template that displays "Hello" NumTimes. Replace the contents of Views/HelloWorld/Welcome.cshtml with the following:

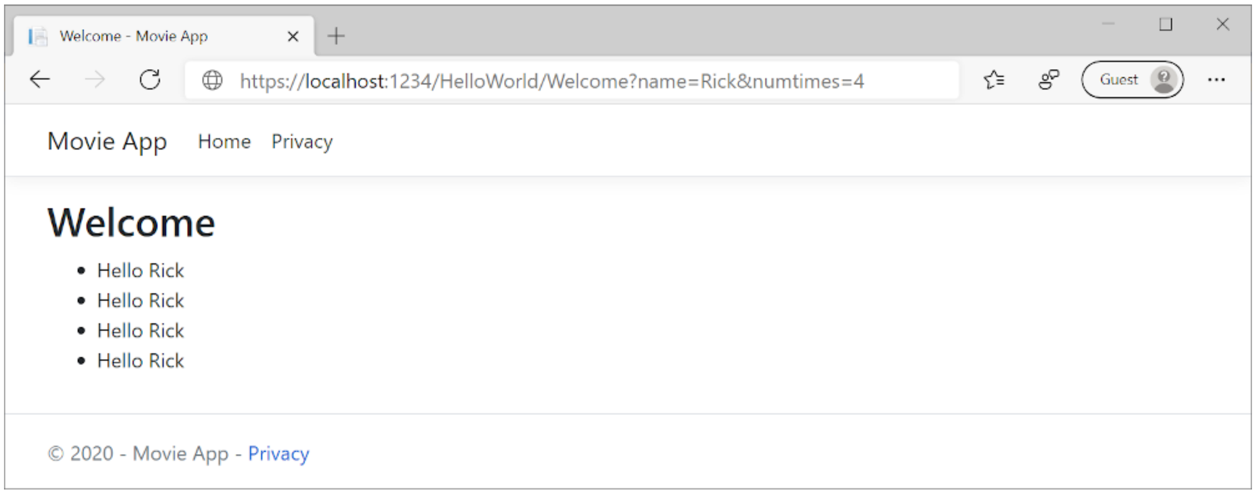
```
@{
    ViewData["Title"] = "Welcome";
}
```

```
<h2>Welcome</h2>

<ul>
    @for (int i = 0; i < (int)ViewData["NumTimes"]!; i++)
    {
        <li>@ViewData["Message"]</li>
    }
</ul>
```

Save your changes and browse to the following URL:

https://localhost:{PORT}/HelloWorld/Welcome?name=Rick&numtimes=4

Data is taken from the URL and passed to the controller using the MVC model binder. The controller packages the data into a ViewData dictionary and passes that object to the view. The view then renders the data as HTML to the browser.


Welcome - Movie App
https://localhost:1234/HelloWorld/Welcome?name=Rick&numtimes=4
Guest
Movie App
Home
Privacy

Welcome

- Hello Rick
- Hello Rick
- Hello Rick
- Hello Rick

© 2020 - Movie App - Privacy

In the preceding sample, the ViewData dictionary was used to pass data from the controller to a view. Later in the tutorial, a view model is used to pass data from a controller to a view. The view model approach to passing data is preferred over the ViewData dictionary approach.

Activity 4 : add a model to an ASP.NET Core MVC app

In this tutorial, classes are added for managing movies in a database. These classes are the "**M**odel" part of the **M**VC app.

These model classes are used with Entity Framework Core (EF Core) to work with a database. EF Core is an object-relational mapping (ORM) framework that simplifies the data access code that you have to write.

The model classes created are known as **POCO** classes, from **P**lain **O**ld **C**LR **O**bjects. POCO classes don't have any dependency on EF Core. They only define the properties of the data to be stored in the database.

In this tutorial, model classes are created first, and EF Core creates the database.

Add a data model class

Right-click the *Models* folder > **Add** > **Class**. Name the file Movie.cs.

Update the Models/Movie.cs file with the following code:

```
using System.ComponentModel.DataAnnotations;

namespace MvcMovie.Models
{
    public class Movie
    {
        public int Id { get; set; }
        public string? Title { get; set; }

        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string? Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

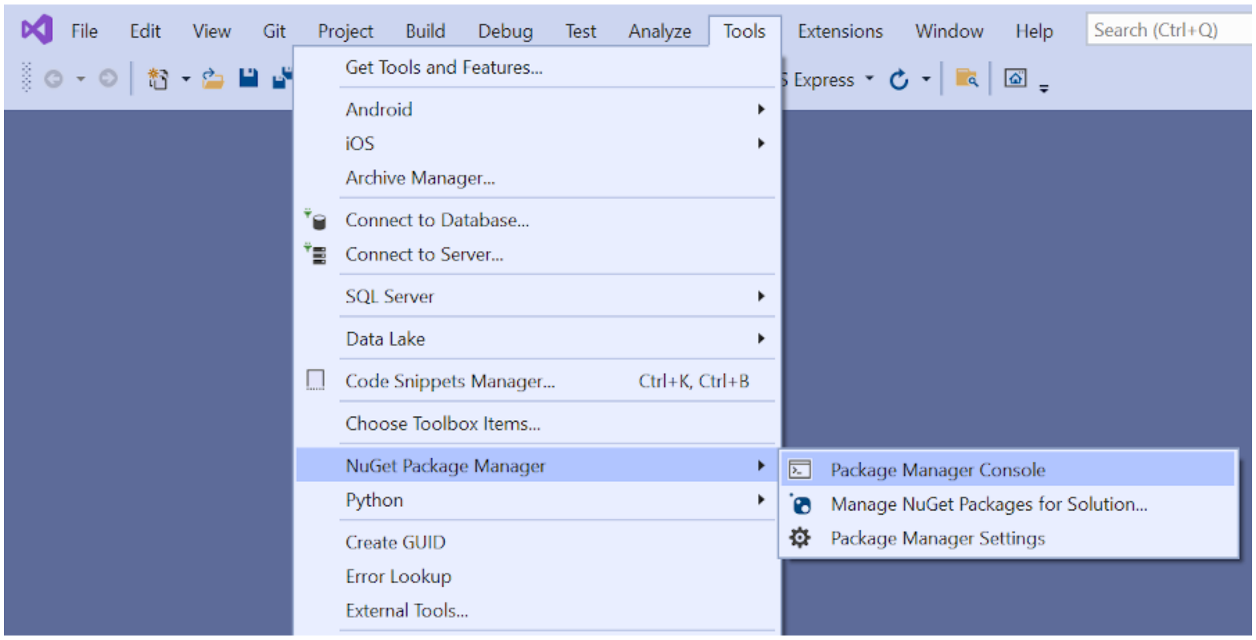
The Movie class contains an Id field, which is required by the database for the primary key.

The DataType attribute on ReleaseDate specifies the type of the data (Date). With this attribute:

- The user isn't required to enter time information in the date field.
- Only the date is displayed, not time information.

Add NuGet packages

From the **Tools** menu, select **NuGet Package Manager** > **Package Manager Console** (PMC).



In the PMC, run the following command:

PowerShellCopy

```
Install-Package Microsoft.EntityFrameworkCore.Design
Install-Package Microsoft.EntityFrameworkCore.SqlServer
```

The preceding commands add:

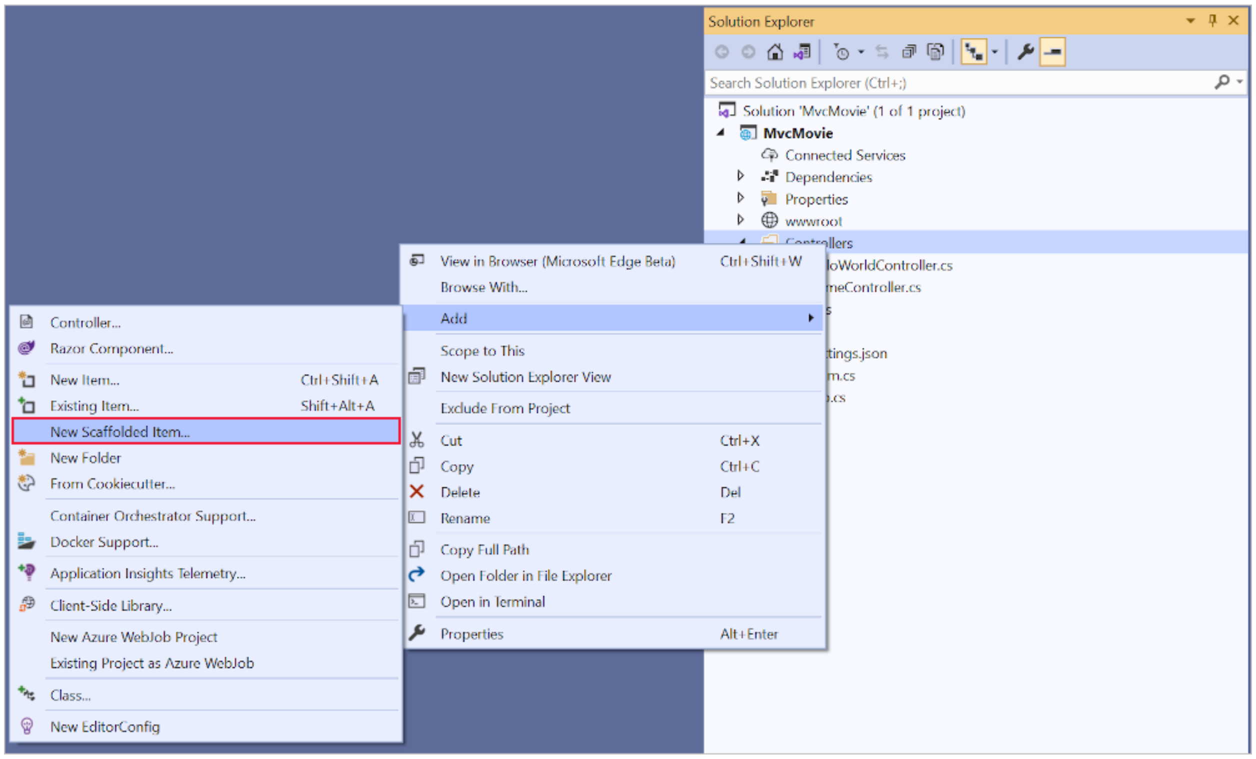
- The EF Core SQL Server provider. The provider package installs the EF Core package as a dependency.
- The utilities used by the packages installed automatically in the scaffolding step, later in the tutorial.

Build the project as a check for compiler errors.

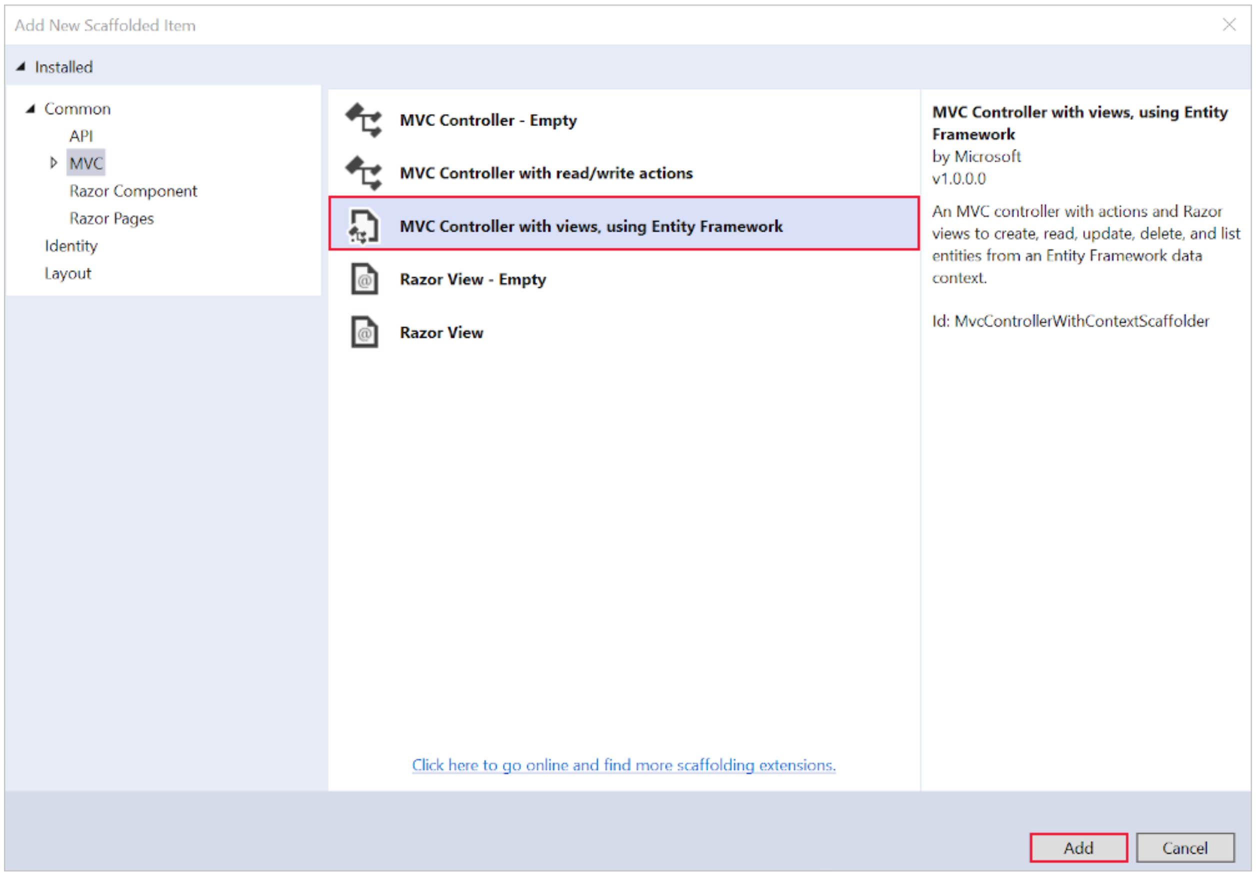
Scaffold movie pages

Use the scaffolding tool to produce Create, Read, Update, and Delete (CRUD) pages for the movie model.

In **Solution Explorer**, right-click the *Controllers* folder and select **Add > New Scaffolded Item**.



In the **Add Scaffold** dialog, select **MVC Controller with views, using Entity Framework** > **Add**.

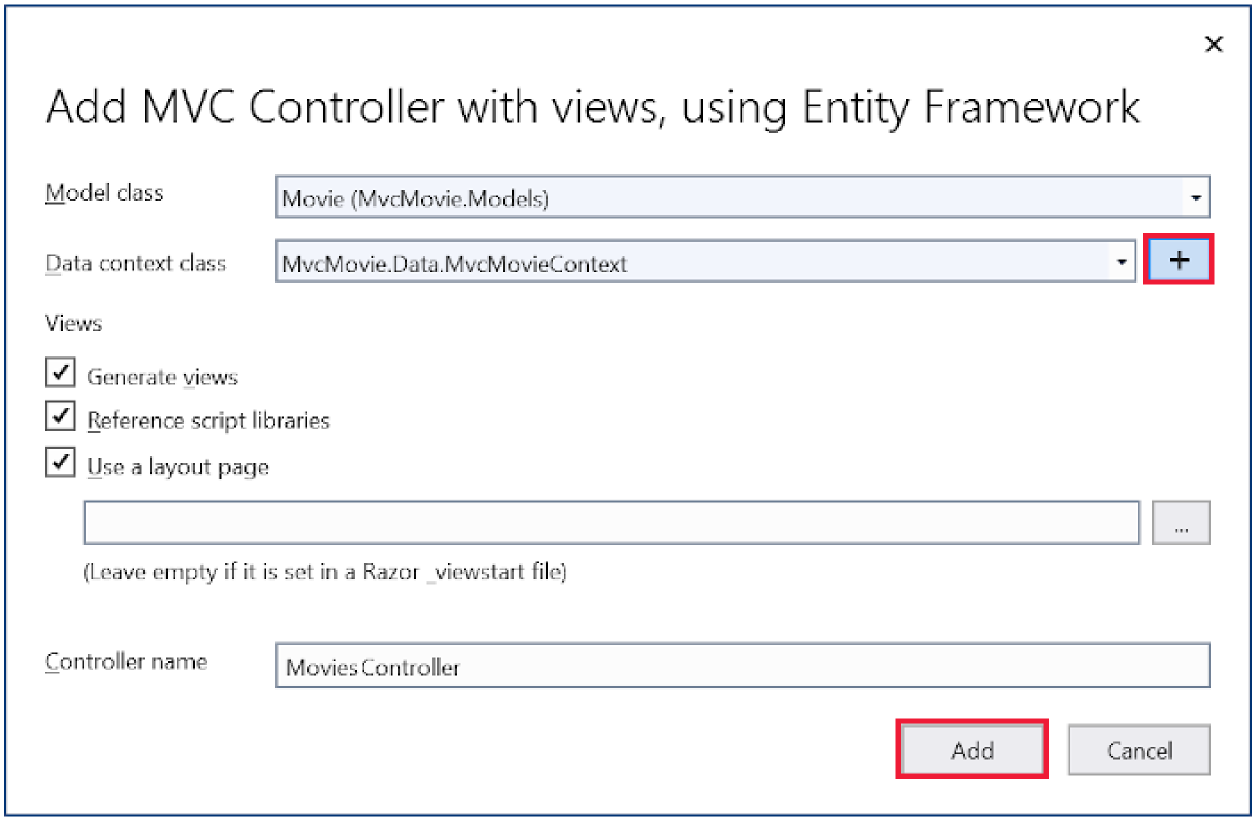

Add New Scaffolded Item
Installed
Common
API
MVC
Razor Component
Razor Pages
Identity
Layout
MVC Controller - Empty
MVC Controller with read/write actions
MVC Controller with views, using Entity Framework
Razor View - Empty
Razor View
Click here to go online and find more scaffolding extensions.
Add
Cancel

MVC Controller with views, using Entity Framework
by Microsoft
v1.0.0.0
An MVC controller with actions and Razor views to create, read, update, delete, and list entities from an Entity Framework data context.

Id: MvcControllerWithContextScaffolder

Complete the **Add MVC Controller with views, using Entity Framework** dialog:

- In the **Model class** drop down, select **Movie (MvcMovie.Models)**.
- In the **Data context class** row, select the **+** (plus) sign.
    - In the **Add Data Context** dialog, the class name *MvcMovie.Data.MvcMovieContext* is generated.
    - Select **Add**.
- **Views** and **Controller name**: Keep the default.
- Select **Add**.



Add MVC Controller with views, using Entity Framework

Model class: Movie (MvcMovie.Models)

Data context class: MvcMovie.Data.MvcMovieContext

Views

✓ Generate views

✓ Reference script libraries

✓ Use a layout page

(Leave empty if it is set in a Razor _viewstart file)

Controller name: MoviesController

If you get an error message, select **Add** a second time to try it again.

Scaffolding updates the following:

- Inserts required package references in the MvcMovie.csproj project file.
- Registers the database context in the Program.cs file.
- Adds a database connection string to the appsettings.json file.

Scaffolding creates the following:

- A movies controller: Controllers/MoviesController.cs
- Razor view files for **Create**, **Delete**, **Details**, **Edit**, and **Index** pages: Views/Movies/*.cshtml
- A database context class: Data/MvcMovieContext.cs

The automatic creation of these files and file updates is known as *scaffolding*.

The scaffolded pages can't be used yet because the database doesn't exist. Running the app and selecting the **Movie App** link results in a *Cannot open database* or *no such table: Movie* error message.

Build the app

Build the app. The compiler generates several warnings about how null values are handled. See this GitHub issue and Nullable reference types for more information.

To eliminate the warnings from nullable reference types, remove the following line from the MvcMovie.csproj file:

<Nullable>enable</Nullable>

Initial migration

Use the EF Core Migrations feature to create the database. *Migrations* is a set of tools that create and update a database to match the data model.

From the **Tools** menu, select **NuGet Package Manager** > **Package Manager Console** .

In the Package Manager Console (PMC), enter the following commands:

```
Add-Migration InitialCreate
Update-Database
```

- Add-Migration InitialCreate: Generates a Migrations/{timestamp}_InitialCreate.cs migration file. The InitialCreate argument is the migration name. Any name can be used, but by convention, a name is selected that describes the migration. Because this is the first migration, the generated class contains code to create the database schema. The database schema is based on the model specified in the MvcMovieContext class.
- Update-Database: Updates the database to the latest migration, which the previous command created. This command runs the Up method in the Migrations/{time-stamp}_InitialCreate.cs file, which creates the database.

The Update-Database command generates the following warning:
No type was specified for the decimal column 'Price' on entity type 'Movie'. This will cause values to be silently truncated if they do not fit in the default precision and scale. Explicitly specify the SQL server column type that can accommodate all the values using 'HasColumnType()'.

Test the app

Run the app and select the **Movie App** link.

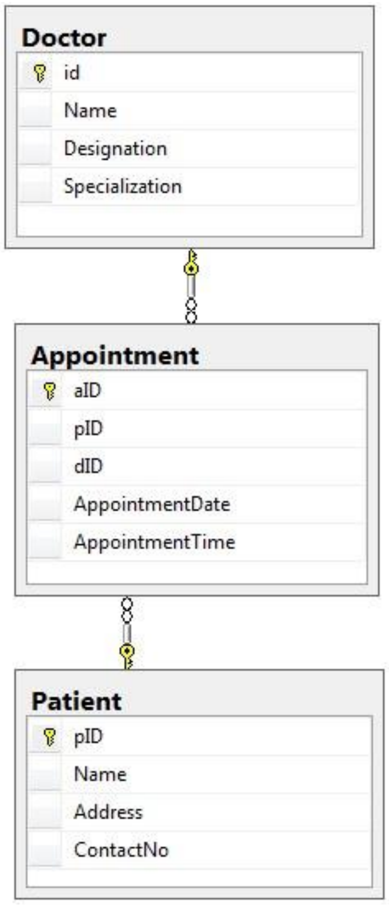
3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab.

Lab Task 1:

Build a MVC application that contains model for patient doctor information system. The application should contain views for patients, doctors that can manage their personal information and for receptionist that can manage the appointments for patients and doctors.

For this exercise Consider following relational database where Patients get Appointments from Doctors. Patient table contains the information of patients. Doctor table contains the information of doctors. Appointment table is a centre table which has Patient's ID as pID and Doctor's ID as dID



Lab 13
Working with a Database in ASP.NET Core MVC

Objective:

Students will learn to use Entity framework to access and query data from model classes and database.

Activity Outcomes:

The activities provide hands - on practice with the following topic:

- Build an MVC application and use entity framework

Instructor Note:

As pre-lab activity, read Part 5, work with a database in an ASP.NET Core MVC app | Microsoft Docs

1) Useful Concepts

Traditionally we have been designing and developing data centric applications(Data Driven Design). What this means is that we used to think about what data is required to fulfill our business needs and then we build our software bottom up from the database schema. This approach is still being followed for many applications and for such applications we should use the Entity framework database first approach.

The alternative way of designing or architecturing our application is by using Domain centric approach(Domain Driven Design). In this approach we think in terms of entities and models that we needed to solve a particular business problem. Now if some of these models need persistence we can keep them in a database or any data store. In this approach we design our models in such a way that they can be stored/persisted anywhere. In other words we create persistent ignorant models and write persistence logic separately. Entity framework code first approach is for creating application's models using Domain centric approach and then they can be persisted later.

So Entity framework code first approach enables us to write Plain Old CLR Objects(POCOs) for our models and then let us persist them in a data store by defining a DbContext class for our model classes. Few of the benefits of using this approach are:

- Ability to support domain driven design.

- Ability to start development faster(without waiting for the database and to be ready and mature).

- Model classes are lot cleaner since there is no(or very minimal) persistence related code in the models.

- The persistence layer can be changed without having any impact on the models.

2) Solved Lab Activites

Sr. No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
Activity 1	***25 mins***	***High***	***CLO-4***
Activity 2	***25 mins***	***High***	***CLO-4***

Activity 1:

Work with a database in an ASP.NET Core MVC app

The MvcMovieContext object handles the task of connecting to the database and mapping Movie objects to database records. The database context is registered with the Dependency Injection container in the Program.cs file:

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<MvcMovieContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("MvcMov
ieContext")));
```

The ASP.NET Core Configuration system reads the ConnectionString key. For local development, it gets the connection string from the appsettings.json file:

```
"ConnectionStrings": {
  "MvcMovieContext":
"Server=(localdb)\\mssqllocaldb;Database=MvcMovieContext-7dc5b790-
765f-4381-988c-
5167405bb107;Trusted_Connection=True;MultipleActiveResultSets=true"
}
```

SQL Server Express LocalDB

LocalDB:

- Is a lightweight version of the SQL Server Express Database Engine, installed by default with Visual Studio.
- Starts on demand by using a connection string.
- Is targeted for program development. It runs in user mode, so there's no complex configuration.
- By default creates *.mdf* files in the *C:/Users/{user}* directory.

Seed the database

Create a new class named SeedData in the *Models* folder. Replace the generated code with the following:

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using MvcMovie.Data;
using System;
using System.Linq;

namespace MvcMovie.Models
{
    public static class SeedData
    {
        public static void Initialize(IServiceProvider
serviceProvider)
        {
            using (var context = new MvcMovieContext(
                serviceProvider.GetRequiredService<
                    DbContextOptions<MvcMovieContext>>()))
            {
                // Look for any movies.
                if (context.Movie.Any())
                {
                    return;   // DB has been seeded
                }

                context.Movie.AddRange(
                    new Movie
                    {
                        Title = "When Harry Met Sally",
                        ReleaseDate = DateTime.Parse("1989-2-12"),
                        Genre = "Romantic Comedy",
                        Price = 7.99M
                    },

                    new Movie
                    {
                        Title = "Ghostbusters ",
                        ReleaseDate = DateTime.Parse("1984-3-13"),
                        Genre = "Comedy",
                        Price = 8.99M
                    },

                    new Movie
                    {
                        Title = "Ghostbusters 2",
                        ReleaseDate = DateTime.Parse("1986-2-23"),
                        Genre = "Comedy",
                        Price = 9.99M
                    },
                    new Movie
                    {
                        Title = "Rio Bravo",
                        ReleaseDate = DateTime.Parse("1959-4-15"),
                        Genre = "Western",
                        Price = 3.99M
                    }
                );
                context.SaveChanges();
            }
        }
    }
}
```

If there are any movies in the database, the seed initializer returns and no movies are added.

```
if (context.Movie.Any())
{
    return;   // DB has been seeded.
}
```

Add the seed initializer

Replace the contents of Program.cs with the following code. The new code is highlighted.

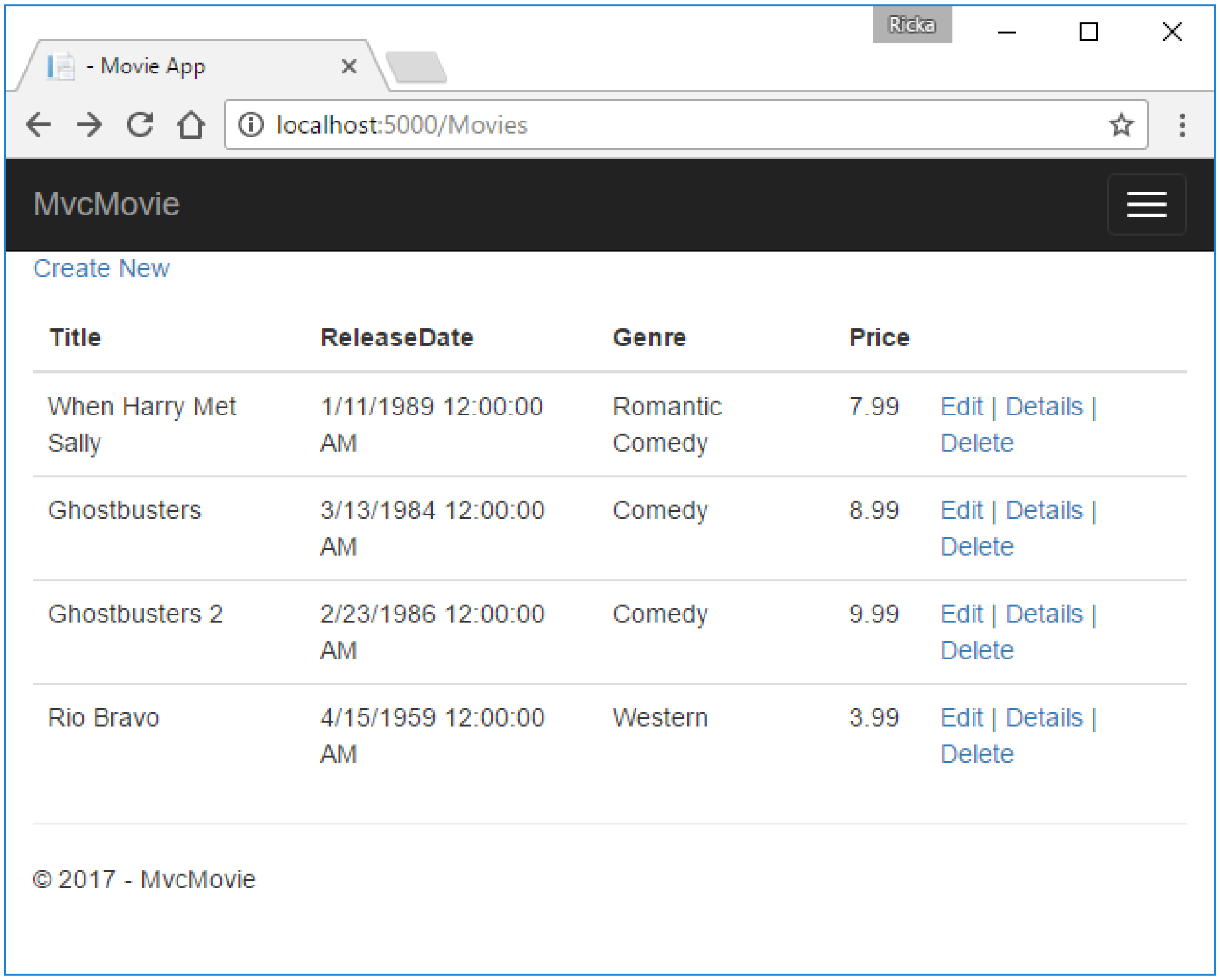
```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using MvcMovie.Data;
using MvcMovie.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<MvcMovieContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("MvcMov
ieContext")));

// Add services to the container.
builder.Services.AddControllersWithViews();

var app = builder.Build();

using (var scope = app.Services.CreateScope())
{
    var services = scope.ServiceProvider;
    SeedData.Initialize(services);
}

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    // The default HSTS value is 30 days. You may want to change this
for production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();
```

Delete all the records in the database. You can do this with the delete links in the browser or from SSOX.

Test the app. Force the app to initialize, calling the code in the Program.cs file, so the seed method runs. To force initialization, close the command prompt window that Visual Studio opened, and restart by pressing Ctrl+F5.

The app shows the seeded data.

ss



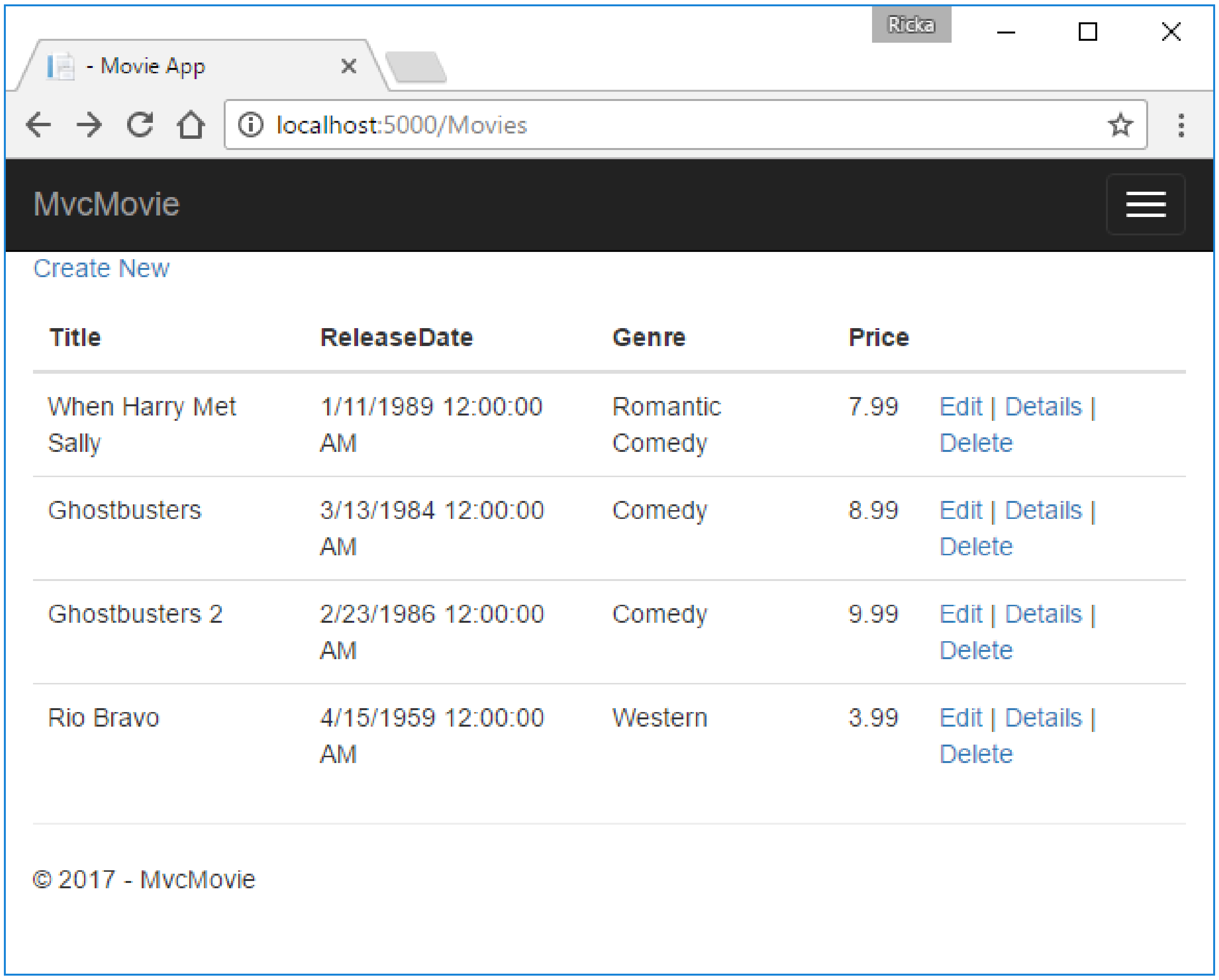
Title	ReleaseDate	Genre	Price	
When Harry Met Sally	1/11/1989 12:00:00 AM	Romantic Comedy	7.99	Edit | Details | Delete
Ghostbusters	3/13/1984 12:00:00 AM	Comedy	8.99	Edit | Details | Delete
Ghostbusters 2	2/23/1986 12:00:00 AM	Comedy	9.99	Edit | Details | Delete
Rio Bravo	4/15/1959 12:00:00 AM	Western	3.99	Edit | Details | Delete

© 2017 - MvcMovie

Activity 2:

Controller methods and views in ASP.NET Core

We have a good start to the movie app, but the presentation isn't ideal, for example, **ReleaseDate** should be two words.



Title	ReleaseDate	Genre	Price	
When Harry Met Sally	1/11/1989 12:00:00 AM	Romantic Comedy	7.99	Edit | Details | Delete
Ghostbusters	3/13/1984 12:00:00 AM	Comedy	8.99	Edit | Details | Delete
Ghostbusters 2	2/23/1986 12:00:00 AM	Comedy	9.99	Edit | Details | Delete
Rio Bravo	4/15/1959 12:00:00 AM	Western	3.99	Edit | Details | Delete

© 2017 - MvcMovie

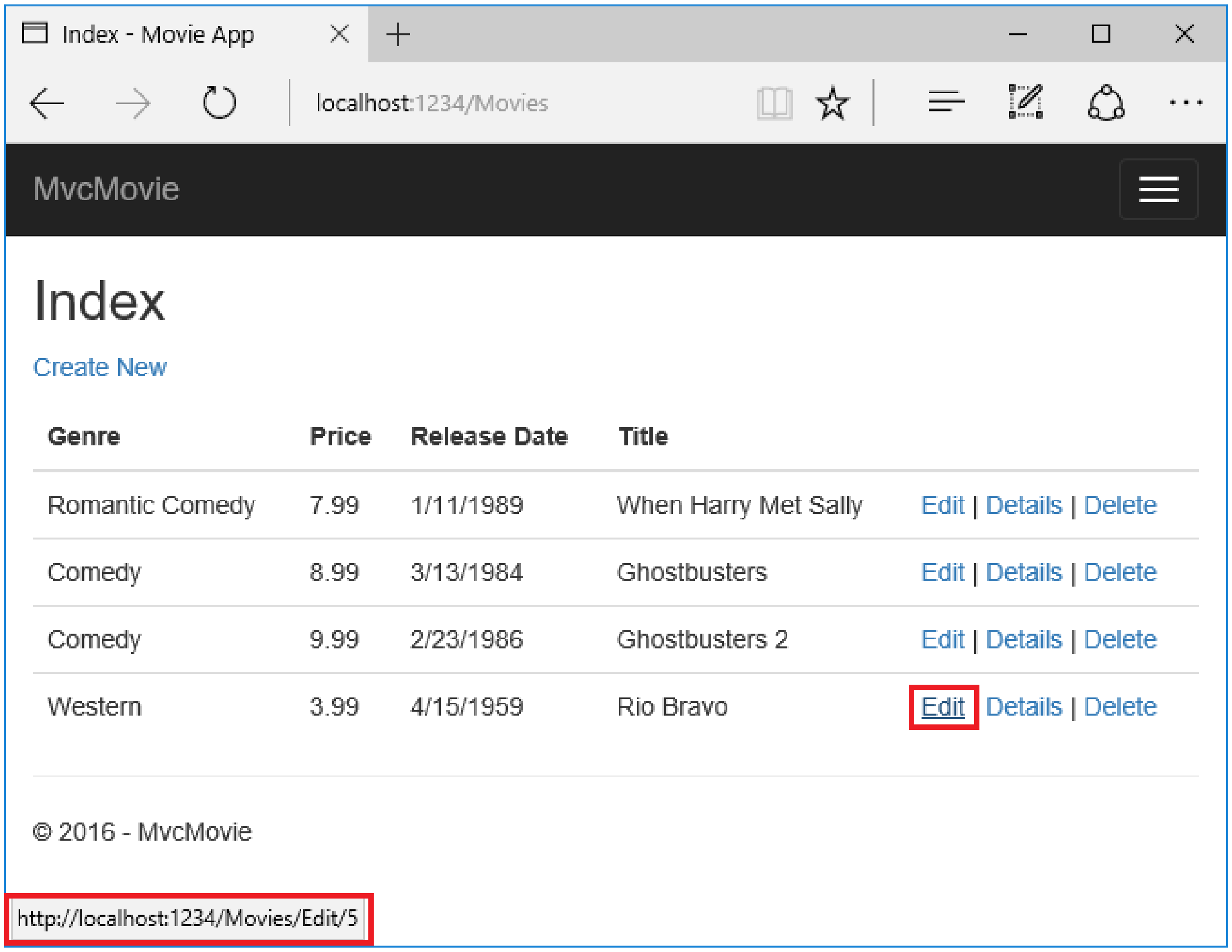
Open the Models/Movie.cs file and add the highlighted lines shown below:

```
using System;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace MvcMovie.Models
{
    public class Movie
    {
        public int Id { get; set; }
        public string? Title { get; set; }

        [Display(Name = "Release Date")]
        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string? Genre { get; set; }

        [Column(TypeName = "decimal(18, 2)")]
        public decimal Price { get; set; }
        }
}
```

DataAnnotations are explained in the next tutorial. The Display attribute specifies what to display for the name of a field (in this case "Release Date" instead of "ReleaseDate"). The DataType attribute specifies the type of the data (Date), so the time information stored in the field isn't displayed.

The [Column(TypeName = "decimal(18, 2)")] data annotation is required so Entity Framework Core can correctly map Price to currency in the database. For more information, see Data Types.

Browse to the Movies controller and hold the mouse pointer over an **Edit** link to see the target URL.



Index

Create New

Genre	Price	Release Date	Title	
Romantic Comedy	7.99	1/11/1989	When Harry Met Sally	Edit | Details | Delete
Comedy	8.99	3/13/1984	Ghostbusters	Edit | Details | Delete
Comedy	9.99	2/23/1986	Ghostbusters 2	Edit | Details | Delete
Western	3.99	4/15/1959	Rio Bravo	Edit | Details | Delete

© 2016 - MvcMovie

http://localhost:1234/Movies/Edit/5

The **Edit**, **Details**, and **Delete** links are generated by the Core MVC Anchor Tag Helper in the Views/Movies/Index.cshtml file.

```
        <a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |
        <a asp-action="Details" asp-route-id="@item.Id">Details</a> |
        <a asp-action="Delete" asp-route-id="@item.Id">Delete</a>
    </td>
```

```
</tr>
```

Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files. In the code above, the AnchorTagHelper dynamically generates the HTML href attribute value from the controller action method and route id. You use **View Source** from your favorite browser or use the developer tools to examine the generated markup. A portion of the generated HTML is shown below:

```
<td>
    <a href="/Movies/Edit/4"> Edit </a> |
    <a href="/Movies/Details/4"> Details </a> |
    <a href="/Movies/Delete/4"> Delete </a>
</td>
```

Recall the format for routing set in the Program.cs file:

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

ASP.NET Core translates https://localhost:5001/Movies/Edit/4 into a request to the Edit action method of the Movies controller with the parameter Id of 4. (Controller methods are also known as action methods.)

Tag Helpers are one of the most popular new features in ASP.NET Core. For more information, see Additional resources.

Open the Movies controller and examine the two Edit action methods. The following code shows the HTTP GET Edit method, which fetches the movie and populates the edit form generated by the Edit.cshtml Razor file.

```
// GET: Movies/Edit/5
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie.FindAsync(id);
    if (movie == null)
    {
        return NotFound();
    }
    return View(movie);
}
```

The following code shows the HTTP POST Edit method, which processes the posted movie values:

```
// POST: Movies/Edit/5
// To protect from overposting attacks, enable the specific properties
you want to bind to.
// For more details, see
http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id,
[Bind("Id,Title,ReleaseDate,Genre,Price,Rating")] Movie movie)
{
    if (id != movie.Id)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.Id))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction(nameof(Index));
    }
    return View(movie);
}
```

The [Bind] attribute is one way to protect against over-posting. You should only include properties in the [Bind] attribute that you want to change. For more information, see Protect your controller from over-posting. ViewModels provide an alternative approach to prevent over-posting.

Notice the second Edit action method is preceded by the [HttpPost] attribute.

```
// POST: Movies/Edit/5
// To protect from overposting attacks, enable the specific properties
you want to bind to.
```

```
// For more details, see
http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id,
[Bind("Id,Title,ReleaseDate,Genre,Price,Rating")] Movie movie)
{
    if (id != movie.Id)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.Id))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction(nameof(Index));
    }
    return View(movie);
}
```

The HttpPost attribute specifies that this Edit method can be invoked *only* for POST requests. You could apply the [HttpGet] attribute to the first edit method, but that's not necessary because [HttpGet] is the default.

The ValidateAntiForgeryToken attribute is used to prevent forgery of a request and is paired up with an anti-forgery token generated in the edit view file (Views/Movies/Edit.cshtml). The edit view file generates the anti-forgery token with the Form Tag Helper.

```
<form asp-action="Edit">
```

The Form Tag Helper generates a hidden anti-forgery token that must match the [ValidateAntiForgeryToken] generated anti-forgery token in the Edit method of the Movies controller.

The HttpGet Edit method takes the movie ID parameter, looks up the movie using the Entity Framework FindAsync method, and returns the selected movie to the Edit view. If a movie cannot be found, NotFound (HTTP 404) is returned.

```
// GET: Movies/Edit/5
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie.FindAsync(id);
    if (movie == null)
    {
        return NotFound();
    }
    return View(movie);
}
```

When the scaffolding system created the Edit view, it examined the Movie class and created code to render <label> and <input> elements for each property of the class. The following example shows the Edit view that was generated by the Visual Studio scaffolding system:

```
@model MvcMovie.Models.Movie

@{
    ViewData["Title"] = "Edit";
}

<h1>Edit</h1>

<h4>Movie</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Edit">
            <div asp-validation-summary="ModelOnly" class="text-
danger"></div>
            <input type="hidden" asp-for="Id" />
            <div class="form-group">
                <label asp-for="Title" class="control-label"></label>
                <input asp-for="Title" class="form-control" />
                <span asp-validation-for="Title" class="text-
danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="ReleaseDate" class="control-
label"></label>
```

```
                <input asp-for="ReleaseDate" class="form-control" />
                <span asp-validation-for="ReleaseDate" class="text-
danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Genre" class="control-label"></label>
                <input asp-for="Genre" class="form-control" />
                <span asp-validation-for="Genre" class="text-
danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Price" class="control-label"></label>
                <input asp-for="Price" class="form-control" />
                <span asp-validation-for="Price" class="text-
danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-
primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Notice how the view template has a @model MvcMovie.Models.Movie statement at the top of the file. @model MvcMovie.Models.Movie specifies that the view expects the model for the view template to be of type Movie.

The scaffolded code uses several Tag Helper methods to streamline the HTML markup. The Label Tag Helper displays the name of the field ("Title", "ReleaseDate", "Genre", or "Price"). The Input Tag Helper renders an HTML <input> element. The Validation Tag Helper displays any validation messages associated with that property.

Run the application and navigate to the /Movies URL. Click an **Edit** link. In the browser, view the source for the page. The generated HTML for the <form> element is shown below.

```
<form action="/Movies/Edit/7" method="post">
    <div class="form-horizontal">
        <h4>Movie</h4>
        <hr />
        <div class="text-danger" />
        <input type="hidden" data-val="true" data-val-required="The ID 
field is required." id="ID" name="ID" value="7" />
        <div class="form-group">
            <label class="control-label col-md-2" for="Genre" />
            <div class="col-md-10">
                <input class="form-control" type="text" id="Genre" 
name="Genre" value="Western" />
                <span class="text-danger field-validation-valid" data-
valmsg-for="Genre" data-valmsg-replace="true"></span>
            </div>
        </div>
        <div class="form-group">
            <label class="control-label col-md-2" for="Price" />
            <div class="col-md-10">
                <input class="form-control" type="text" data-
val="true" data-val-number="The field Price must be a number." data-
val-required="The Price field is required." id="Price" name="Price" 
value="3.99" />
                <span class="text-danger field-validation-valid" data-
valmsg-for="Price" data-valmsg-replace="true"></span>
            </div>
        </div>
        <!-- Markup removed for brevity -->
        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Save" class="btn btn-
default" />
            </div>
        </div>
    </div>
    <input name="__RequestVerificationToken" type="hidden" 
value="CfDJ8Inyxgp63fRFqUePGvuI5jGZsloJu1L7X9le1gy7NCIlSduCRx9jDQClrV9
pOTTmqUyXnJBXhmrjcUVDJyDUMm7-
MF_9rK8aAZdRdlOri7FmKVkRe_2v5LIHGKFcTjPrWPYnc9AdSbomkiOSaTEg7RU" />
</form>
```

The <input> elements are in an HTML <form> element whose action attribute is set to post to the /Movies/Edit/id URL. The form data will be posted to the server when the Save button is clicked. The last line before the closing </form> element shows the hidden XSRF token generated by the Form Tag Helper.

Processing the POST Request

The following listing shows the [HttpPost] version of the Edit action method.

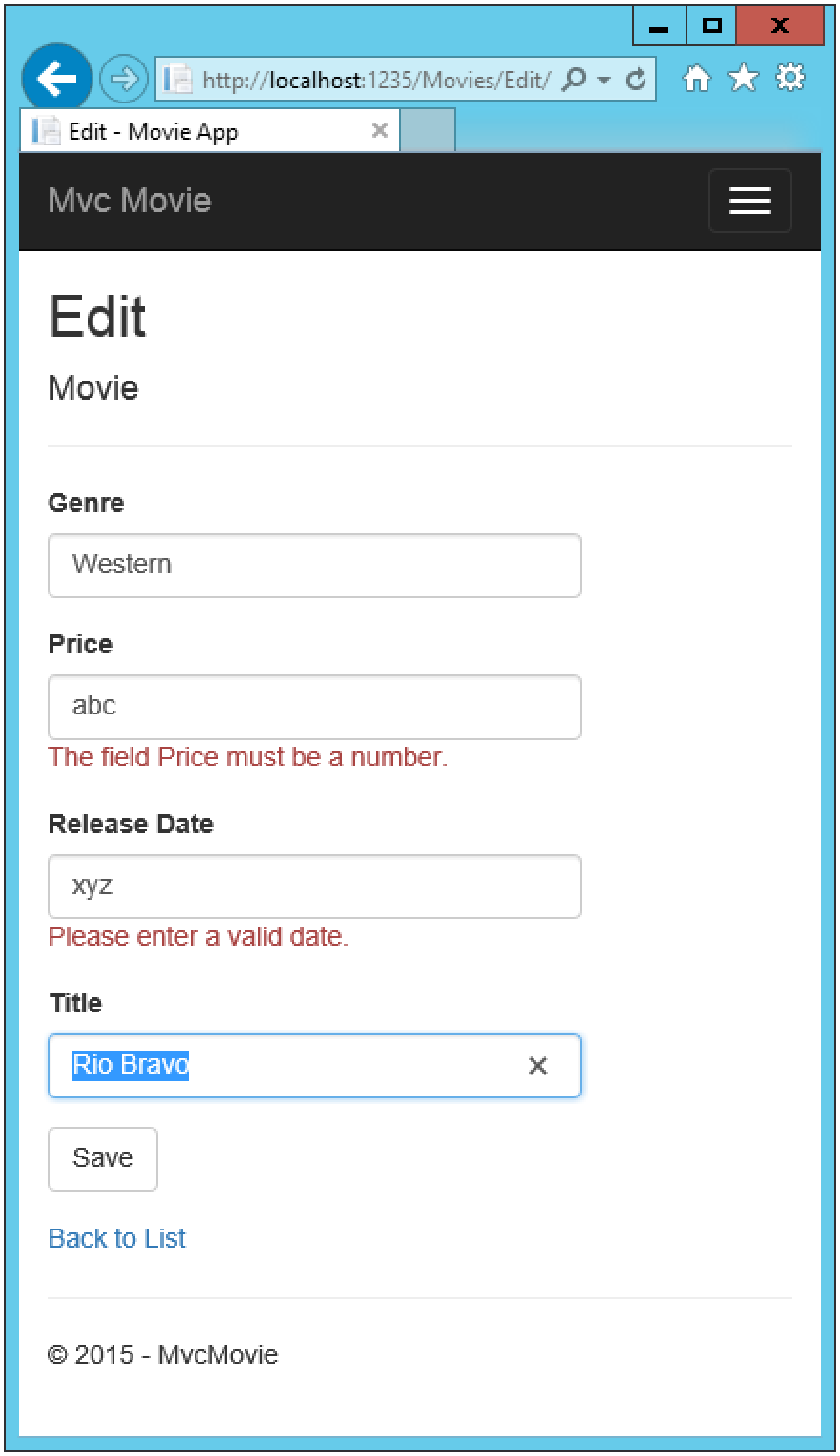
```
// POST: Movies/Edit/5
// To protect from overposting attacks, enable the specific properties 
you want to bind to.
```

```
// For more details, see
http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id,
[Bind("Id,Title,ReleaseDate,Genre,Price,Rating")] Movie movie)
{
    if (id != movie.Id)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.Id))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction(nameof(Index));
    }
    return View(movie);
}
```

The [ValidateAntiForgeryToken] attribute validates the hidden XSRF token generated by the anti-forgery token generator in the Form Tag Helper

The model binding system takes the posted form values and creates a Movie object that's passed as the movie parameter. The ModelState.IsValid property verifies that the data submitted in the form can be used to modify (edit or update) a Movie object. If the data is valid, it's saved. The updated (edited) movie data is saved to the database by calling the SaveChangesAsync method of database context. After saving the data, the code redirects the user to the Index action method of the MoviesController class, which displays the movie collection, including the changes just made.

Before the form is posted to the server, client-side validation checks any validation rules on the fields. If there are any validation errors, an error message is displayed and the form isn't posted. If JavaScript is disabled, you won't have client-side validation but the server will detect the posted values that are not valid, and the form values will be redisplayed with error messages. Later in the tutorial we examine Model Validation in more detail. The Validation Tag Helper in the Views/Movies/Edit.cshtml view template takes care of displaying appropriate error messages.



Edit

Movie

Genre

Western

Price

abc

The field Price must be a number.

Release Date

xyz

Please enter a valid date.

Title

Rio Bravo

Save

Back to List

© 2015 - MvcMovie

All the HttpGet methods in the movie controller follow a similar pattern. They get a movie object (or list of objects, in the case of Index), and pass the object (model) to the view. The Create method passes an empty movie object to the Create view. All the methods that create, edit, delete, or otherwise modify data do so in the [HttpPost] overload of the method. Modifying data in an HTTP GET method is a security risk. Modifying data in an HTTP GET method also violates HTTP best practices and the architectural REST pattern, which specifies that GET requests shouldn't change the state of your application. In other words, performing a GET operation should be a safe operation that has no side effects and doesn't modify your persisted data.

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab.

Lab Task 1:

Create an application similar to the one made in lab according to our own university.

Lab 14
Working with a Database (continued)

Objective:

Students will learn to use Entity framework to access and query data from model classes and database.

Activity Outcomes:

The activities provide hands - on practice with the following topic:

- Build an MVC application and use entity framework

Instructor Note:

As pre-lab activity, read Part 7, add search to an ASP.NET Core MVC app | Microsoft Docs

1) Useful Concepts

Traditionally we have been designing and developing data centric applications (Data Driven Design). What this means is that we used to think about what data is required to fulfill our business needs and then we build our software bottom up from the database schema. This approach is still being followed for many applications and for such applications we should use the Entity framework database first approach.

The alternative way of designing or architecturing our application is by using Domain centric approach (Domain Driven Design). In this approach we think in terms of entities and models that we needed to solve a particular business problem. Now if some of these models need persistence we can keep them in a database or any data store. In this approach we design our models in such a way that they can be stored/persisted anywhere. In other words we create persistent ignorant models and write persistence logic separately. Entity framework code first approach is for creating application's models using Domain centric approach and then they can be persisted later.

2) Solved Lab Activites

Sr. No	***Allocated Time***	***Level of Complexity***	***CLO Mapping***
Activity 1	***15 mins***	***Medium***	***CLO-4***
Activity 2	***15 mins***	***Medium***	***CLO-4***
Activity 3	***15 mins***	***Medium***	***CLO-4***
Activity 4	***15 mins***	***Medium***	***CLO-4***

Activity 1:

Add search to an ASP.NET Core MVC app

In this section, we will add search capability to the Index action method that lets you search movies by *genre* or *name*.

Update the Index method found inside Controllers/MoviesController.cs with the following code:

```
public async Task<IActionResult> Index(string searchString)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title!.Contains(searchString));
    }

    return View(await movies.ToListAsync());
}
```

The first line of the Index action method creates a LINQ query to select the movies:

```
var movies = from m in _context.Movie
             select m;
```

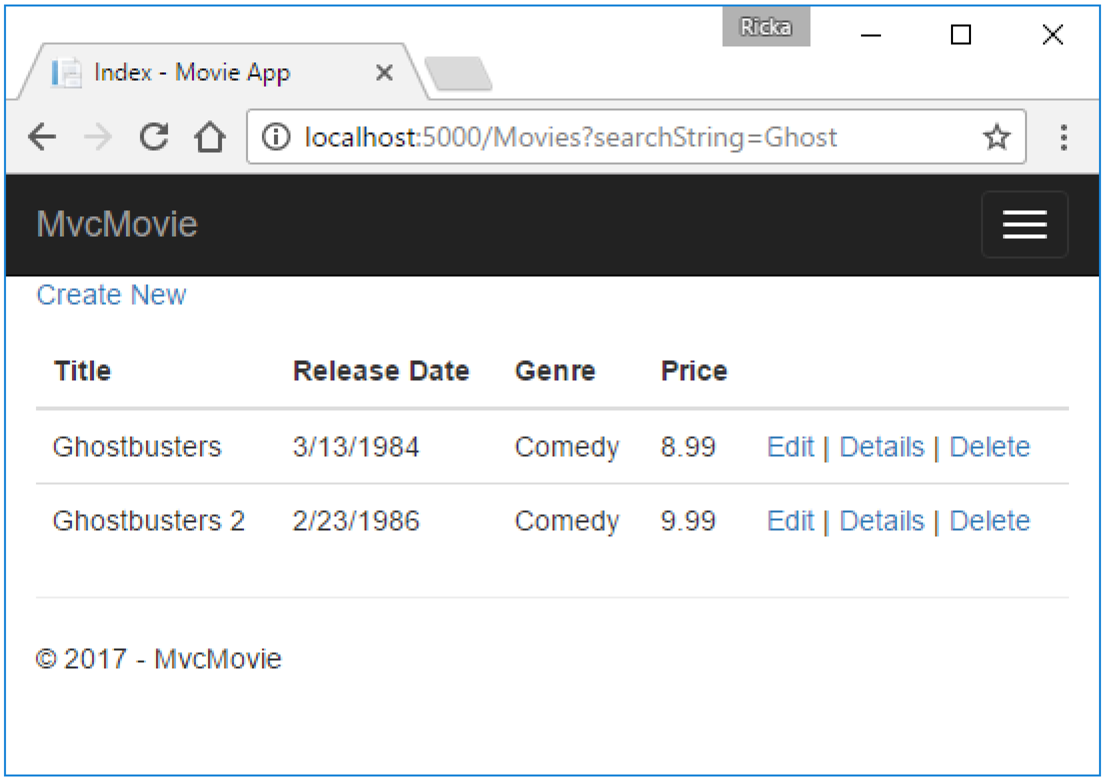
The query is *only defined* at this point, it has **not** been run against the database.

If the searchString parameter contains a string, the movies query is modified to filter on the value of the search string:

```
if (!String.IsNullOrEmpty(searchString))
{
    movies = movies.Where(s => s.Title!.Contains(searchString));
}
```

The s => s.Title!.Contains(searchString) code above is a Lambda Expression. Lambdas are used in method-based LINQ queries as arguments to standard query operator methods such as the Where method or Contains (used in the code above). LINQ queries are not executed when they're defined or when they're modified by calling a method such as Where, Contains, or OrderBy. Rather, query execution is deferred. That means that the evaluation of an expression is delayed until its realized value is actually iterated over or the ToListAsync method is called. For more information about deferred query execution, see Query Execution.

Navigate to /Movies/Index. Append a query string such as ?searchString=Ghost to the URL. The filtered movies are displayed.



Create New

Title	Release Date	Genre	Price	
Ghostbusters	3/13/1984	Comedy	8.99	Edit | Details | Delete
Ghostbusters 2	2/23/1986	Comedy	9.99	Edit | Details | Delete

© 2017 - MvcMovie

If you change the signature of the Index method to have a parameter named id, the id parameter will match the optional {id} placeholder for the default routes set in Program.cs.

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

Change the parameter to id and change all occurrences of searchString to id.

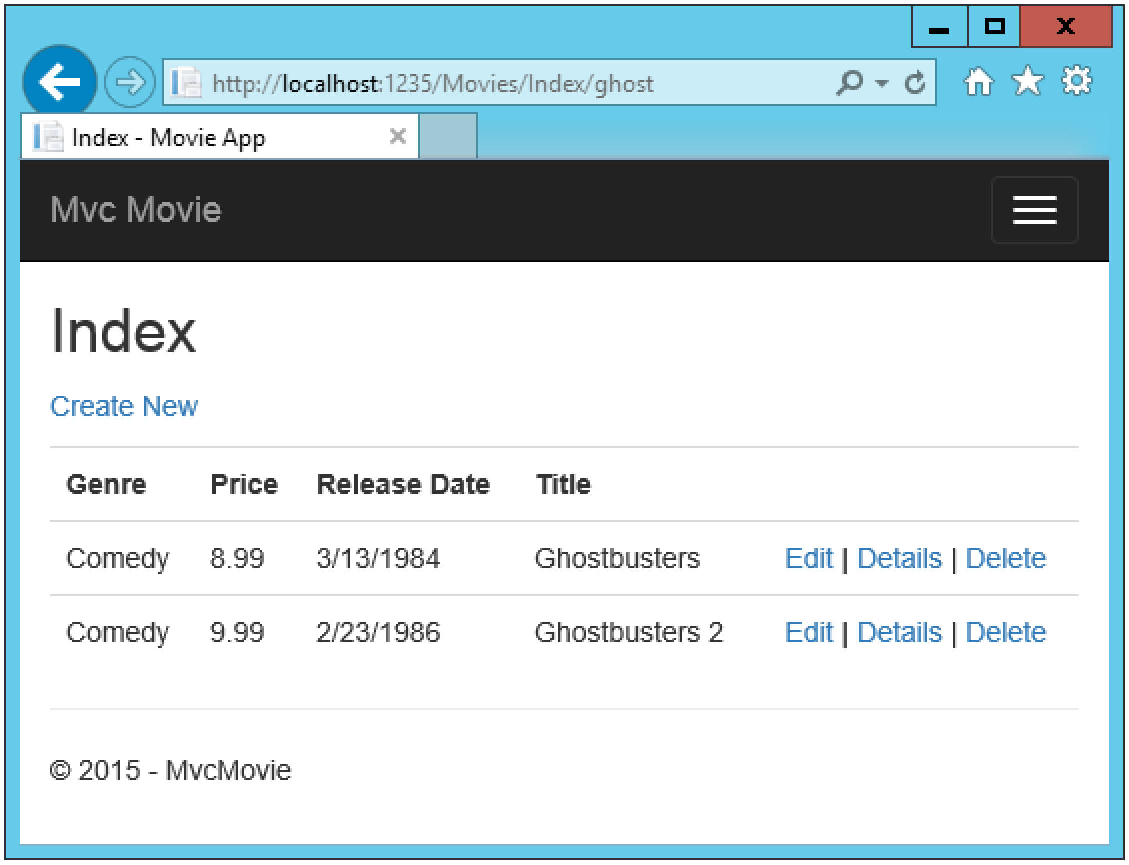
The previous Index method:

```
public async Task<IActionResult> Index(string searchString)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title!.Contains(searchString));
    }

    return View(await movies.ToListAsync());
}
```

The updated Index method with id parameter:

```
public async Task<IActionResult> Index(string id)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(id))
    {
        movies = movies.Where(s => s.Title!.Contains(id));
    }

    return View(await movies.ToListAsync());
}
```

You can now pass the search title as route data (a URL segment) instead of as a query string value.



Index

Create New

Genre	Price	Release Date	Title	
Comedy	8.99	3/13/1984	Ghostbusters	Edit | Details | Delete
Comedy	9.99	2/23/1986	Ghostbusters 2	Edit | Details | Delete

© 2015 - MvcMovie

However, you can't expect users to modify the URL every time they want to search for a movie. So now you'll add UI elements to help them filter movies. If you changed the signature of the Index method to test how to pass the route-bound ID parameter, change it back so that it takes a parameter named searchString:

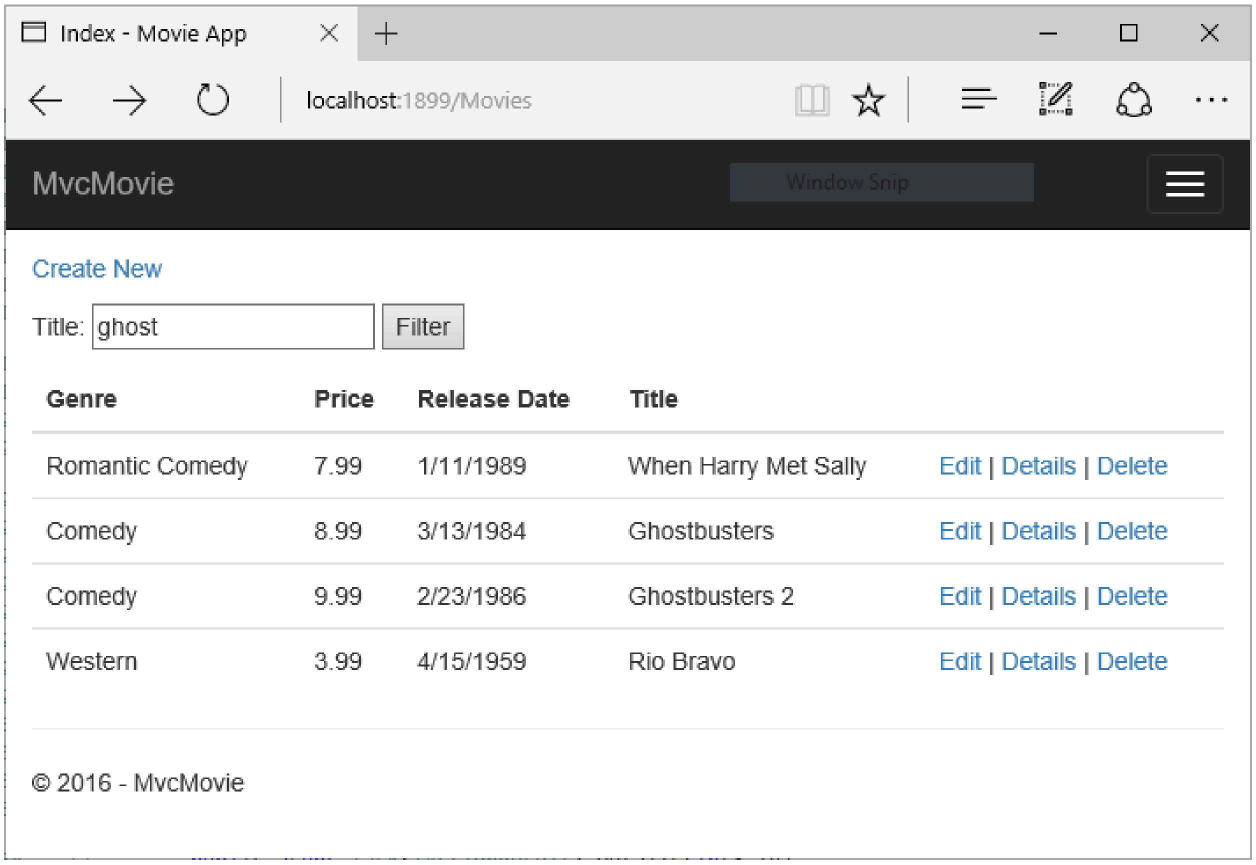
```
public async Task<IActionResult> Index(string searchString)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title!.Contains(searchString));
    }

    return View(await movies.ToListAsync());
}
```

Open the Views/Movies/Index.cshtml file, and add the <form> markup highlighted below:

```
@model IEnumerable<MvcMovie.Models.Movie>
```

```
@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>

<form asp-controller="Movies" asp-action="Index">
    <p>
        Title: <input type="text" name="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>
```

The HTML <form> tag uses the Form Tag Helper, so when you submit the form, the filter string is posted to the Index action of the movies controller. Save your changes and then test the filter.



Genre	Price	Release Date	Title	
Romantic Comedy	7.99	1/11/1989	When Harry Met Sally	Edit | Details | Delete
Comedy	8.99	3/13/1984	Ghostbusters	Edit | Details | Delete
Comedy	9.99	2/23/1986	Ghostbusters 2	Edit | Details | Delete
Western	3.99	4/15/1959	Rio Bravo	Edit | Details | Delete

© 2016 - MvcMovie

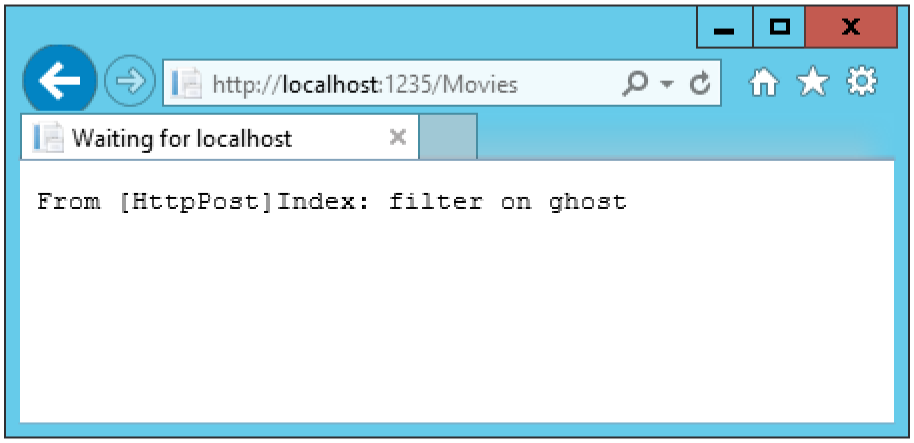
There's no [HttpPost] overload of the Index method as you might expect. You don't need it, because the method isn't changing the state of the app, just filtering data.

You could add the following [HttpPost] Index method.

```
[HttpPost]
public string Index(string searchString, bool notUsed)
{
    return "From [HttpPost]Index: filter on " + searchString;
}
```

The notUsed parameter is used to create an overload for the Index method. We'll talk about that later in the tutorial.

If you add this method, the action invoker would match the [HttpPost] Index method, and the [HttpPost] Index method would run as shown in the image below.



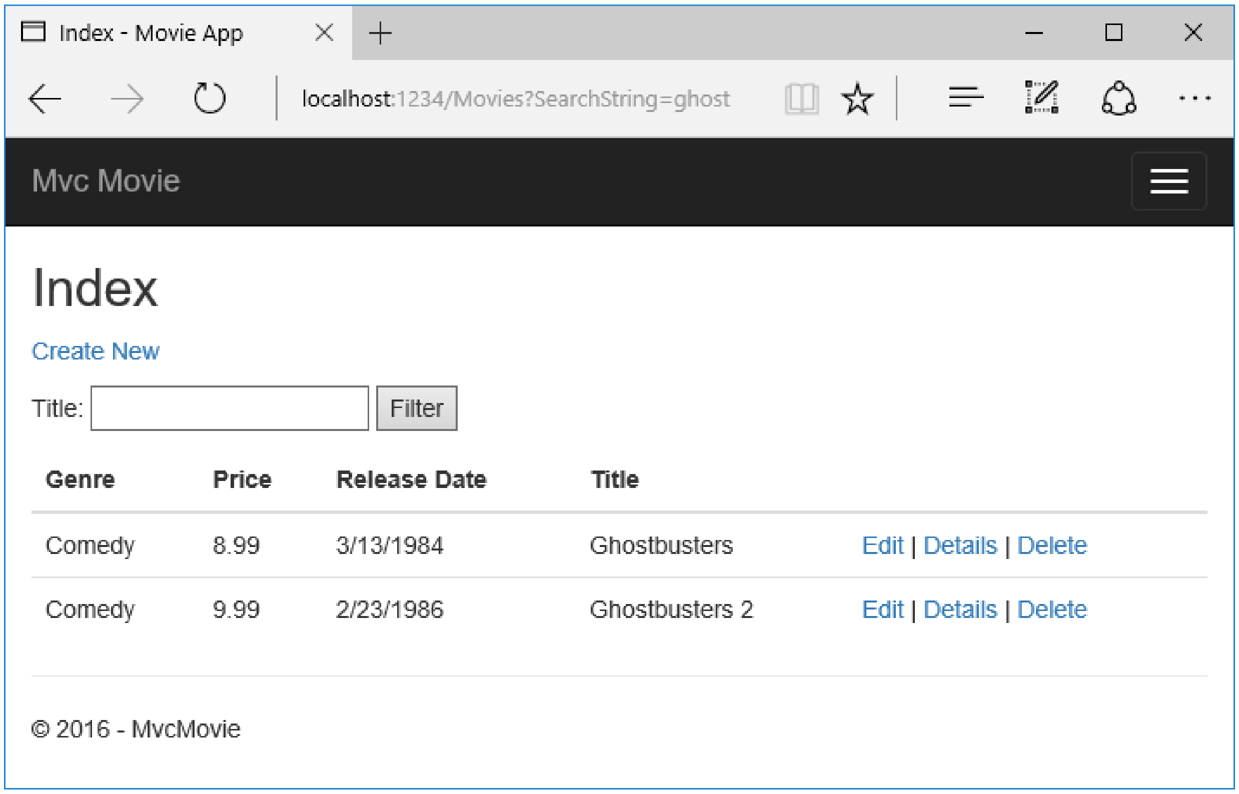
```
From [HttpPost]Index: filter on ghost
```

However, even if you add this [HttpPost] version of the Index method, there's a limitation in how this has all been implemented. Imagine that you want to bookmark a particular search or you want to send a link to friends that they can click in order to see the same filtered list of movies. Notice that the URL for the HTTP POST request is the same as the URL for the GET request (localhost:{PORT}/Movies/Index) -- there's no search information in the URL. The search string information is sent to the server as a form field value.

Because the search parameter is in the request body and not the URL, you can't capture that search information to bookmark or share with others. Fix this by specifying the request should be HTTP GET found in the Views/Movies/Index.cshtml file.

```
@model IEnumerable<MvcMovie.Models.Movie>

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>

<form asp-controller="Movies" asp-action="Index" method="get">
    <p>
        Title: <input type="text" name="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>
<table class="table">
```

Now when you submit a search, the URL contains the search query string. Searching will also go to the HttpGet Index action method, even if you have a HttpPost Index method.



Genre	Price	Release Date	Title	
Comedy	8.99	3/13/1984	Ghostbusters	Edit | Details | Delete
Comedy	9.99	2/23/1986	Ghostbusters 2	Edit | Details | Delete

© 2016 - MvcMovie

The following markup shows the change to the form tag:

```
<form asp-controller="Movies" asp-action="Index" method="get">
```

Add Search by genre

Add the following MovieGenreViewModel class to the *Models* folder:

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace MvcMovie.Models
{
    public class MovieGenreViewModel
    {
        public List<Movie>? Movies { get; set; }
        public SelectList? Genres { get; set; }
        public string? MovieGenre { get; set; }
        public string? SearchString { get; set; }
    }
}
```

The movie-genre view model will contain:

- A list of movies.
- A SelectList containing the list of genres. This allows the user to select a genre from the list.
- MovieGenre, which contains the selected genre.
- SearchString, which contains the text users enter in the search text box.

Replace the Index method in MoviesController.cs with the following code:

```
// GET: Movies
public async Task<IActionResult> Index(string movieGenre, string
searchString)
{
    // Use LINQ to get list of genres.
    IQueryable<string> genreQuery = from m in _context.Movie
                                    orderby m.Genre
                                    select m.Genre;
    var movies = from m in _context.Movie
                 select m;

    if (!string.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title!.Contains(searchString));
    }

    if (!string.IsNullOrEmpty(movieGenre))
    {
        movies = movies.Where(x => x.Genre == movieGenre);
    }

    var movieGenreVM = new MovieGenreViewModel
    {
        Genres = new SelectList(await
genreQuery.Distinct().ToListAsync()),
        Movies = await movies.ToListAsync()
    };

    return View(movieGenreVM);
}
```

The following code is a LINQ query that retrieves all the genres from the database.

```
// Use LINQ to get list of genres.
IQueryable<string> genreQuery = from m in _context.Movie
                                orderby m.Genre
                                select m.Genre;
```

The SelectList of genres is created by projecting the distinct genres (we don't want our select list to have duplicate genres).

When the user searches for the item, the search value is retained in the search box.

Add search by genre to the Index view

Update Index.cshtml found in *Views/Movies/* as follows:

```
@model MvcMovie.Models.MovieGenreViewModel

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>
```

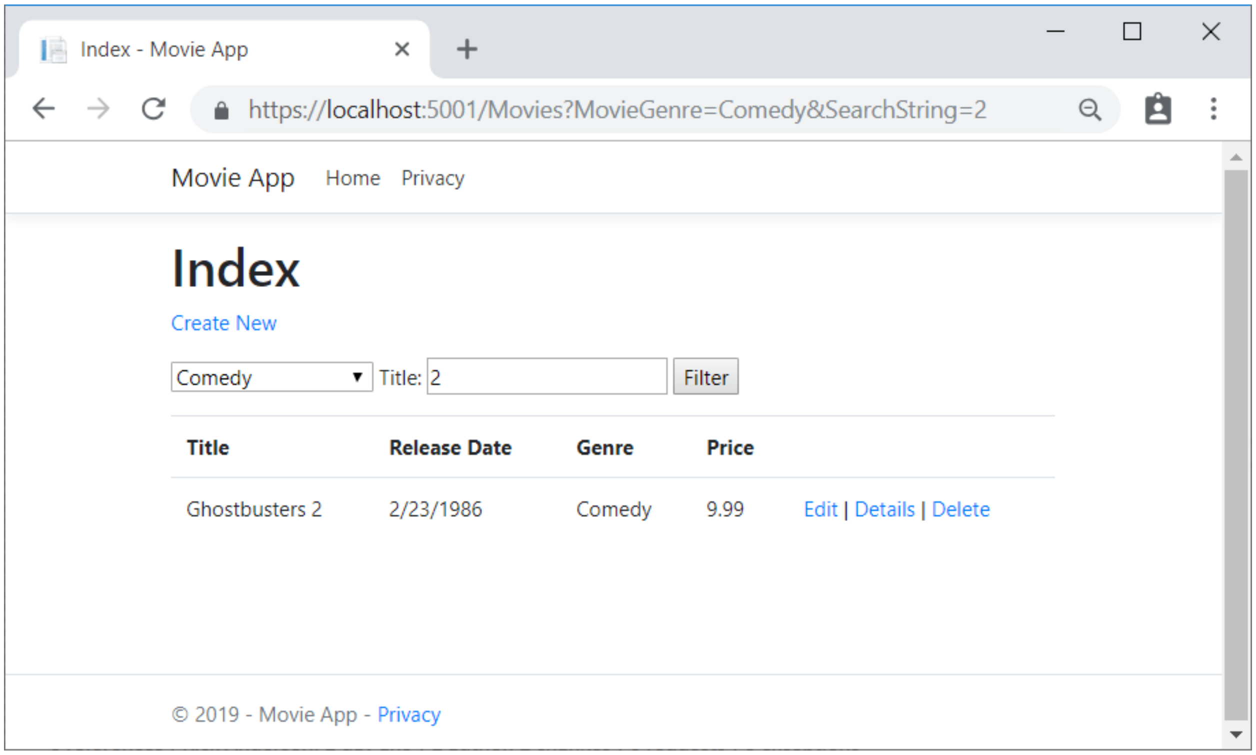
```
<p>
    <a asp-action="Create">Create New</a>
</p>
<form asp-controller="Movies" asp-action="Index" method="get">
    <p>

        <select asp-for="MovieGenre" asp-items="Model.Genres">
            <option value="">All</option>
        </select>

        Title: <input type="text" asp-for="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Movies[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model =>
model.Movies[0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movies[0].Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movies[0].Price)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Movies)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.ReleaseDate)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Genre)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Price)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-
id="@item.Id">Edit</a> |
                    <a asp-action="Details" asp-route-
id="@item.Id">Details</a> |
                    <a asp-action="Delete" asp-route-
id="@item.Id">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

Examine the lambda expression used in the following HTML Helper:

@Html.DisplayNameFor(model => model.Movies[0].Title)

In the preceding code, the DisplayNameFor HTML Helper inspects the Title property referenced in the lambda expression to determine the display name. Since the lambda expression is inspected rather than evaluated, you don't receive an access violation when model, model.Movies, or model.Movies[0] are null or empty. When the lambda expression is evaluated (for example, @Html.DisplayFor(modelItem => item.Title)), the model's property values are evaluated.

Test the app by searching by genre, by movie title, and by both:


Index - Movie App
https://localhost:5001/Movies?MovieGenre=Comedy&SearchString=2
Movie App
Home
Privacy

Index

Create New

Comedy ▾ Title: 2 Filter

Title	Release Date	Genre	Price	
Ghostbusters 2	2/23/1986	Comedy	9.99	Edit | Details | Delete

© 2019 - Movie App - Privacy

Activity 2:

Add a new field to an ASP.NET Core MVC app

In this activity Entity Framework Code First Migrations is used to:

- Add a new field to the model.
- Migrate the new field to the database.

When EF Code First is used to automatically create a database, Code First:

- Adds a table to the database to track the schema of the database.
- Verifies the database is in sync with the model classes it was generated from. If they aren't in sync, EF throws an exception. This makes it easier to find inconsistent database/code issues.

Add a Rating Property to the Movie Model

Add a Rating property to Models/Movie.cs:

```
using System;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace MvcMovie.Models
```

```
{
    public class Movie
    {
        public int Id { get; set; }
        public string? Title { get; set; }

        [Display(Name = "Release Date")]
        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string? Genre { get; set; }

        [Column(TypeName = "decimal(18, 2)")]
        public decimal Price { get; set; }
        public string? Rating {  get; set; }
    }
}
```

Build the app

Ctrl+Shift+B

Because you've added a new field to the Movie class, you need to update the property binding list so this new property will be included. In MoviesController.cs, update the [Bind] attribute for both the Create and Edit action methods to include the Rating property:

```
[Bind("Id,Title,ReleaseDate,Genre,Price,Rating")]
```

Update the view templates in order to display, create, and edit the new Rating property in the browser view.

Edit the /Views/Movies/Index.cshtml file and add a Rating field:

```
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Movies[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model =>
model.Movies[0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movies[0].Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movies[0].Price)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movies[0].Rating)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Movies)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.ReleaseDate)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Genre)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Price)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Rating)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-
id="@item.Id">Edit</a> |
                    <a asp-action="Details" asp-route-
id="@item.Id">Details</a> |
                    <a asp-action="Delete" asp-route-
id="@item.Id">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

Update the /Views/Movies/Create.cshtml with a Rating field.

You can copy/paste the previous "form group" and let intelliSense help you update the fields. IntelliSense works with Tag Helpers.

Update the remaining templates.

Update the SeedData class so that it provides a value for the new column. A sample change is shown below, but you'll want to make this change for each new Movie.

```
new Movie
{
    Title = "When Harry Met Sally",
    ReleaseDate = DateTime.Parse("1989-1-11"),
    Genre = "Romantic Comedy",
    Rating = "R",
    Price = 7.99M
},
```

The app won't work until the DB is updated to include the new field. If it's run now, the following SqlException is thrown:

SqlException: Invalid column name 'Rating'.

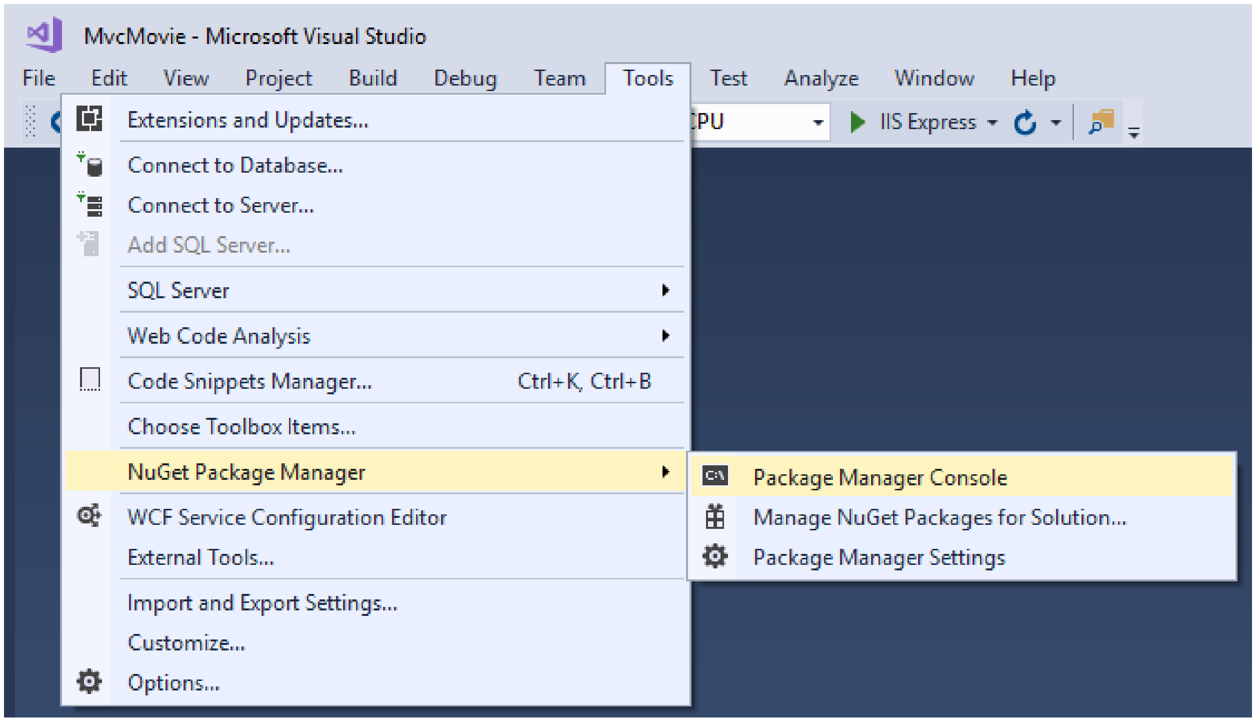
This error occurs because the updated Movie model class is different than the schema of the Movie table of the existing database. (There's no Rating column in the database table.)

There are a few approaches to resolving the error:

- Have the Entity Framework automatically drop and re-create the database based on the new model class schema. This approach is very convenient early in the development cycle when you're doing active development on a test database; it allows you to quickly evolve the model and database schema together. The downside, though, is that you lose existing data in the database — so you don't want to use this approach on a production database! Using an initializer to automatically seed a database with test data is often a productive way to develop an application. This is a good approach for early development and when using SQLite.
- Explicitly modify the schema of the existing database so that it matches the model classes. The advantage of this approach is that you keep your data. You can make this change either manually or by creating a database change script.
- Use Code First Migrations to update the database schema.

For this tutorial, Code First Migrations is used.

From the **Tools** menu, select **NuGet Package Manager** > **Package Manager Console**.



In the PMC, enter the following commands:

```
Add-Migration Rating
Update-Database
```

The Add-Migration command tells the migration framework to examine the current Movie model with the current Movie DB schema and create the necessary code to migrate the DB to the new model.

The name "Rating" is arbitrary and is used to name the migration file. It's helpful to use a meaningful name for the migration file.

If all the records in the DB are deleted, the initialize method will seed the DB and include the Rating field.

Run the app and verify you can create, edit, and display movies with a Rating field.

Actvity 3:

Add validation to an ASP.NET Core MVC app

The DataAnnotations namespace provides a set of built-in validation attributes that are applied declaratively to a class or property. DataAnnotations also contains formatting attributes like DataType that help with formatting and don't provide any validation.

Update the Movie class to take advantage of the built-in Required, StringLength, RegularExpression, and Range validation attributes.

```
using System;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace MvcMovie.Models
{
    public class Movie
    {
        public int Id { get; set; }

        [StringLength(60, MinimumLength = 3)]
        [Required]
        public string? Title { get; set; }

        [Display(Name = "Release Date")]
        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }

        [Range(1, 100)]
        [DataType(DataType.Currency)]
        [Column(TypeName = "decimal(18, 2)")]
        public decimal Price { get; set; }

        [RegularExpression(@"^[A-Z]+[a-zA-Z\s]*$")]
        [Required]
        [StringLength(30)]
        public string? Genre { get; set; }

        [RegularExpression(@"^[A-Z]+[a-zA-Z0-9""'\s-]*$")]
        [StringLength(5)]
        [Required]
        public string? Rating { get; set; }
    }
}
```

The validation attributes specify behavior that you want to enforce on the model properties they're applied to:

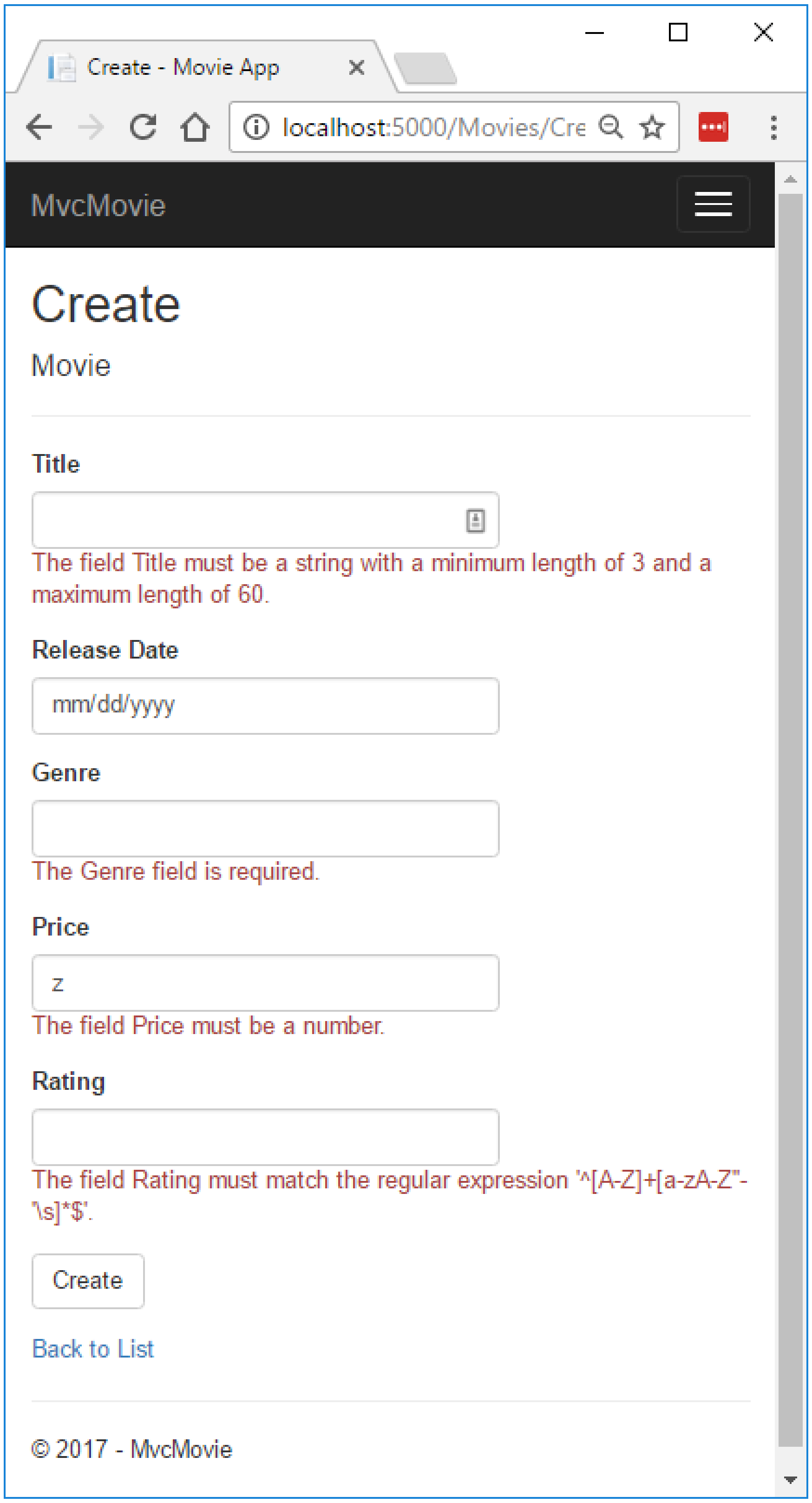
- The Required and MinimumLength attributes indicate that a property must have a value; but nothing prevents a user from entering white space to satisfy this validation.
- The RegularExpression attribute is used to limit what characters can be input. In the preceding code, "Genre":
  - Must only use letters.
  - The first letter is required to be uppercase. White spaces are allowed while numbers, and special characters are not allowed.
- The RegularExpression "Rating":
  - Requires that the first character be an uppercase letter.
  - Allows special characters and numbers in subsequent spaces. "PG-13" is valid for a rating, but fails for a "Genre".
- The Range attribute constrains a value to within a specified range.
- The StringLength attribute lets you set the maximum length of a string property, and optionally its minimum length.
- Value types (such as decimal, int, float, DateTime) are inherently required and don't need the [Required] attribute.

Having validation rules automatically enforced by ASP.NET Core helps make your app more robust. It also ensures that you can't forget to validate something and inadvertently let bad data into the database.

Validation Error UI

Run the app and navigate to the Movies controller.

Select the **Create New** link to add a new movie. Fill out the form with some invalid values. As soon as jQuery client side validation detects the error, it displays an error message.



Create

Movie

Title

The field Title must be a string with a minimum length of 3 and a maximum length of 60.

Release Date

mm/dd/yyyy

Genre

The Genre field is required.

Price

z

The field Price must be a number.

Rating

The field Rating must match the regular expression '^[A-Z]+[a-zA-Z"'-\s]*$'.

© 2017 - MvcMovie

A significant benefit is that you didn't need to change a single line of code in the MoviesController class or in the Create.cshtml view in order to enable this validation UI. The controller and views you created earlier in this tutorial automatically picked up the validation rules that you specified by using validation attributes on the properties of the Movie model class. Test validation using the Edit action method, and the same validation is applied.

Using DataType Attributes

Open the Movie.cs file and examine the Movie class.

The System.ComponentModel.DataAnnotations namespace provides formatting attributes in addition to the built-in set of validation attributes. We've already applied a DataType enumeration value to the release date and to the price fields. The following code shows the ReleaseDate and Price properties with the appropriate DataType attribute.

```
[Display(Name = "Release Date")]
[DataType(DataType.Date)]
public DateTime ReleaseDate { get; set; }

[Range(1, 100)]
[DataType(DataType.Currency)]
[Column(TypeName = "decimal(18, 2)")]
public decimal Price { get; set; }
```

The DataType attributes only provide hints for the view engine to format the data (and supplies elements/attributes such as <a> for URL's and <a href="mailto:EmailAddress.com"> for email. You can use the RegularExpression attribute to validate the format of the data. The DataType attribute is used to specify a data type that's more specific than the database intrinsic type, they're not validation attributes. In this case we only want to keep track of the date, not the time. The DataType Enumeration provides for many data types, such as Date, Time, PhoneNumber, Currency, EmailAddress and more. The DataType attribute can also enable the application to automatically provide type-specific features. For example, a mailto: link can be created for DataType.EmailAddress, and a date selector can be provided for DataType.Date in browsers that support HTML5. The DataType attributes emit HTML 5 data- (pronounced data dash) attributes that HTML 5 browsers can understand. The DataType attributes do **not** provide any validation.

DataType.Date doesn't specify the format of the date that's displayed. By default, the data field is displayed according to the default formats based on the server's CultureInfo.

The DisplayFormat attribute is used to explicitly specify the date format:

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
public DateTime ReleaseDate { get; set; }
```

The ApplyFormatInEditMode setting specifies that the formatting should also be applied when the value is displayed in a text box for editing. (You might not want that for some fields — for example, for currency values, you probably don't want the currency symbol in the text box for editing.)

You can use the DisplayFormat attribute by itself, but it's generally a good idea to use the DataType attribute. The DataType attribute conveys the semantics of the data as opposed to how to render it on a screen, and provides the following benefits that you don't get with DisplayFormat:

- The browser can enable HTML5 features (for example to show a calendar control, the locale-appropriate currency symbol, email links, etc.)
- By default, the browser will render data using the correct format based on your locale.
- The DataType attribute can enable MVC to choose the right field template to render the data (the DisplayFormat if used by itself uses the string template).

You will need to disable jQuery date validation to use the Range attribute with DateTime. It's generally not a good practice to compile hard dates in your models, so using the Range attribute and DateTime is discouraged.

The following code shows combining attributes on one line:

```
using System;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace MvcMovie.Models
{
    public class Movie
    {
        public int Id { get; set; }

        [StringLength(60, MinimumLength = 3)]
        public string Title { get; set; }

        [Display(Name = "Release Date"), DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }

        [RegularExpression(@"^[A-Z]+[a-zA-Z\s]*$"), Required,
StringLength(30)]
        public string Genre { get; set; }

        [Range(1, 100), DataType(DataType.Currency)]
        [Column(TypeName = "decimal(18, 2)")]
        public decimal Price { get; set; }
        [RegularExpression(@"^[A-Z]+[a-zA-Z0-9""'\s-]*$"),
StringLength(5)]
        public string Rating { get; set; }
    }
}
```

Activity 4:

Examine the Details and Delete methods of an ASP.NET Core app

Open the Movie controller and examine the Details method:

```
// GET: Movies/Details/5
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie
        .FirstOrDefaultAsync(m => m.Id == id);
    if (movie == null)
    {
        return NotFound();
    }

    return View(movie);

}
```

The MVC scaffolding engine that created this action method adds a comment showing an HTTP request that invokes the method. In this case it's a GET request with three URL segments, the Movies controller, the Details method, and an id value. Recall these segments are defined in Program.cs.

```
app.MapControllerRoute(
    name: "default",

 pattern: "{controller=Home}/{action=Index}/{id?}");
```

EF makes it easy to search for data using the FirstOrDefaultAsync method. An important security feature built into the method is that the code verifies that the search method has found a movie before it tries to do anything with it. For example, a hacker could introduce errors into the site by changing the URL created by the links from http://localhost:{PORT}/Movies/Details/1 to something like http://localhost:{PORT}/Movies/Details/12345 (or some other value that doesn't represent an actual movie). If you didn't check for a null movie, the app would throw an exception.

Examine the Delete and DeleteConfirmed methods.

```
// GET: Movies/Delete/5
public async Task<IActionResult> Delete(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie
        .FirstOrDefaultAsync(m => m.Id == id);
    if (movie == null)
    {
        return NotFound();
    }

    return View(movie);
}

// POST: Movies/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    var movie = await _context.Movie.FindAsync(id);
    _context.Movie.Remove(movie);
    await _context.SaveChangesAsync();
    return RedirectToAction(nameof(Index));

}
```

Note that the HTTP GET Delete method doesn't delete the specified movie, it returns a view of the movie where you can submit (HttpPost) the deletion. Performing a delete operation in response to a GET request (or for that matter, performing an edit operation, create operation, or any other operation that changes data) opens up a security hole.

The [HttpPost] method that deletes the data is named DeleteConfirmed to give the HTTP POST method a unique signature or name. The two method signatures are shown below:

```
// GET: Movies/Delete/5
```

```
public async Task<IActionResult> Delete(int? id)

{
```

```
// POST: Movies/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)

{
```

The common language runtime (CLR) requires overloaded methods to have a unique parameter signature (same method name but different list of parameters). However, here you need two Delete methods -- one for GET and one for POST -- that both have the same parameter signature. (They both need to accept a single integer as a parameter.)

There are two approaches to this problem, one is to give the methods different names. That's what the scaffolding mechanism did in the preceding example. However, this introduces a small problem: ASP.NET maps segments of a URL to action methods by name, and if you rename a method, routing normally wouldn't be able to find that method. The solution is what you see in the example, which is to add the ActionName("Delete") attribute to the DeleteConfirmed method. That attribute performs mapping for the routing system so that a URL that includes /Delete/ for a POST request will find the DeleteConfirmed method.

Another common work around for methods that have identical names and signatures is to artificially change the signature of the POST method to include an extra (unused) parameter. That's what we did in a previous post when we added the notUsed parameter. You could do the same thing here for the [HttpPost] Delete method:

```
// POST: Movies/Delete/6
[HttpPost]
[ValidateAntiForgeryToken]

public async Task<IActionResult> Delete(int id, bool notUsed)
```

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab.

Lab Task 1:

Create an application similar to the one made in lab according to our own university.