ASP.NET Core Activity #2

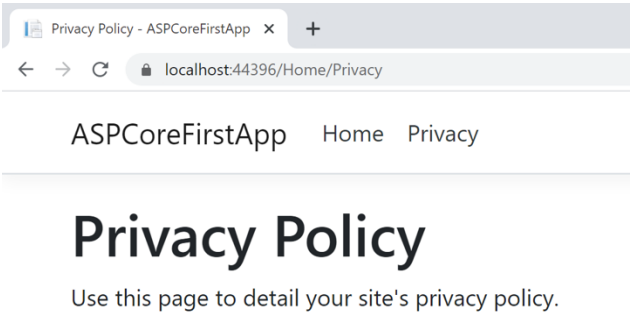
Part 1 Controllers and Views

Introduction to Controllers and Views

Goal and Directions:

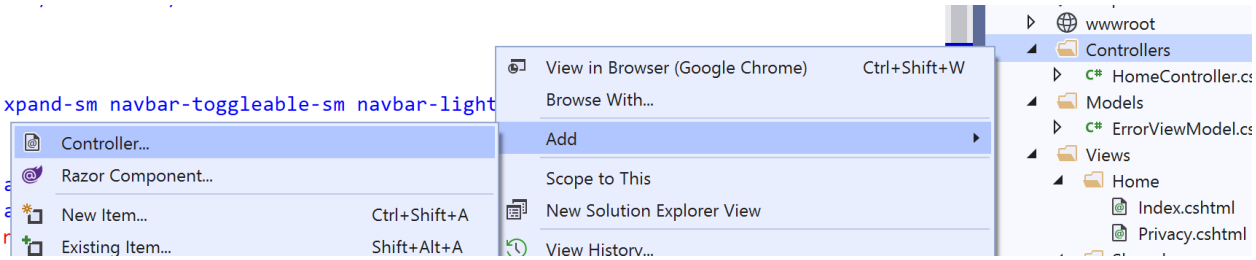
In this lesson, we will display some data of objects and their properties in the Views of the application.

We will continue working with the default, two-page app created in the previous instructions.

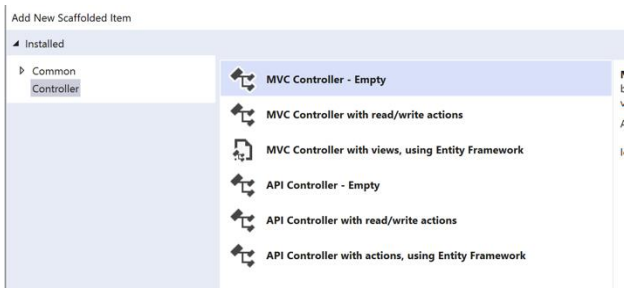


How to create and utilize a controller

1. Right-click on the Controllers Folder. Select the Add → Controller… menu items.

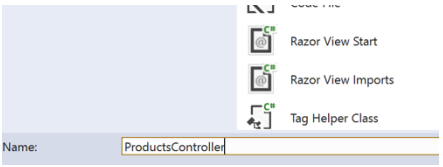


2. Select the 'MVC Controller - Empty' type.



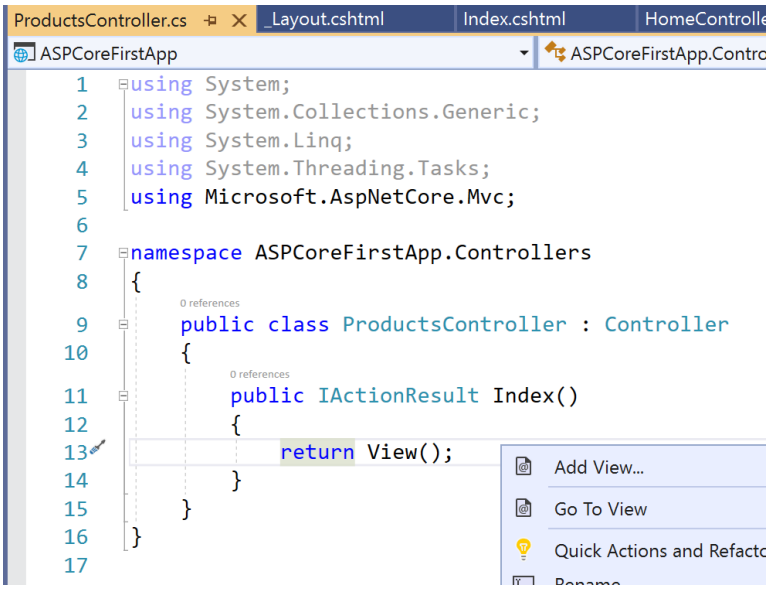
3. Click the Add button.
4. Name your Controller **ProductsController** and click the Add button.

Controllers are commonly named after the type of object they are meant to manage; Products, People, Tickets, Appointments, Orders, etc. are all examples of things that controllers typically manage.

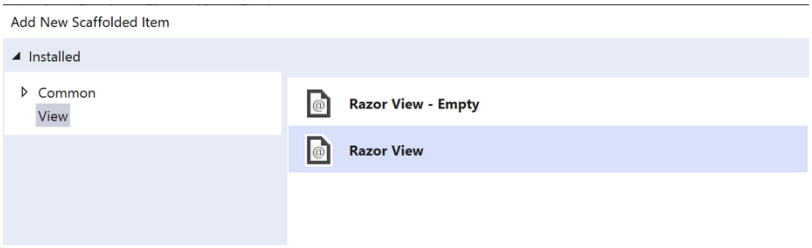


You should see a new Controller in the Controllers folder.

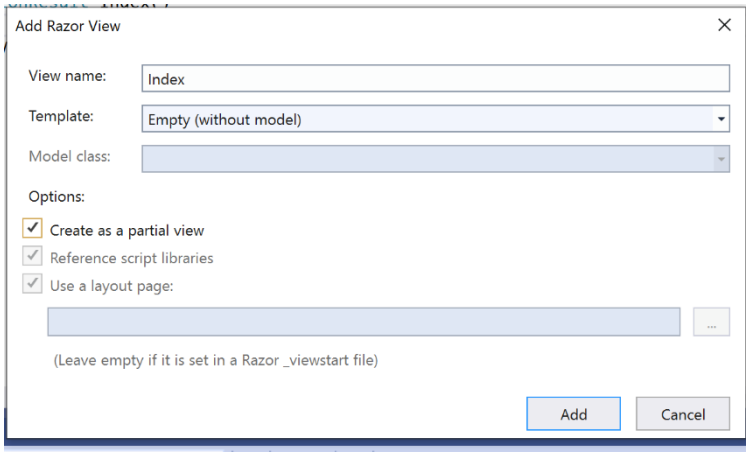
5. **Right-click** inside the Index() method off the controller and choosing **Add View…**



6. Choose Razor View.



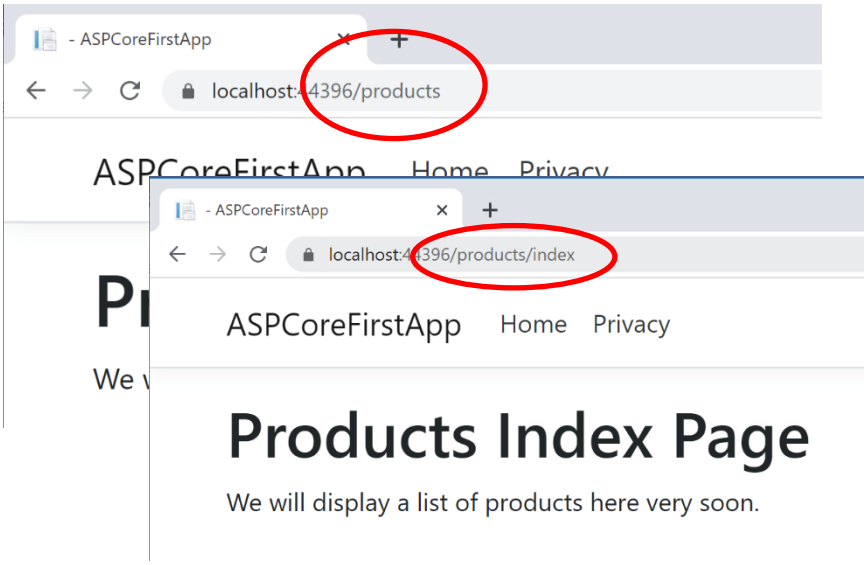
7. Use "Index" as a View Name, empty for Template and "Create as a partial view."



8. Add some HTML text to the page.

```
Index.cshtml*  ProductsController.cs  _Layout.cshtml  Index.cshtml  HomeCont

1
2   @*
3       For more information on enabling MVC for empty projects, vis
4   *@
5
6   <h1>Products Index Page</h1>
7   <p>We will display a list of products here very soon.</p>
```

9. You should be able to run the app. Add a **/products** or **/products/index** to the page URL and the new View will be displayed. The index page is assumed if it is omitted.
10. Capture a screenshot of the app at this stage. Put the image into a Microsoft Word document with a caption explaining what you have just demonstrated.



Routing Format Explanation

The trailing part of the URL after the first slash character is called the **route**. The route determines which controller and method will be executed in the app. The pattern of the route is shown here.

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

You can set the format for routing in the ***Startup.cs*** file of the solution. However, no changes should be necessary for the examples in this lesson.

```
48
49              app.UseEndpoints(endpoints =>
50              {
51                  endpoints.MapControllerRoute(
52                      name: "default",
53                      pattern: "{controller=Home}/{action=Index}/{id?}");
54              });
55          }
56      }
57  }
```

When you run the application and don't supply any URL segments, it defaults to the "Home" controller and the "Index" action method specified in the defaults section of the code above.

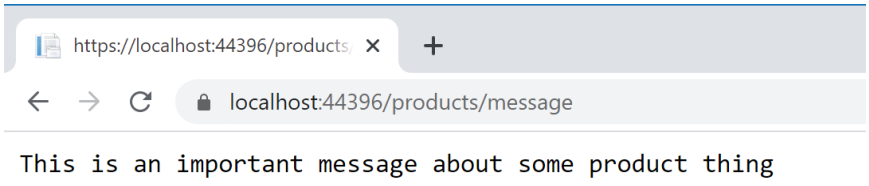
The first part of the URL determines the controller class to execute. So ***/products*** maps to the ProductsController class. The word "controller" is assumed. The second part of the URL determines the action method on the class to execute.

---

1. Update the **ProductsController** file to include a new method, Message. This method will send a **string** to the browser instead of a complete page.

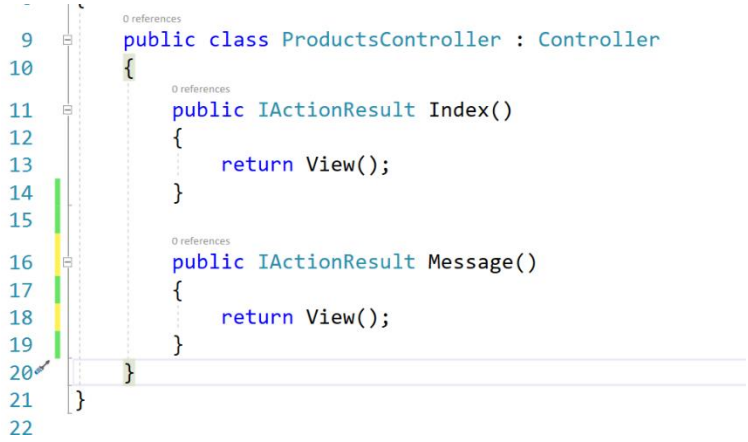
```
7   namespace ASPCoreFirstApp.Controllers
8   {
        0 references
9       public class ProductsController : Controller
10      {
            0 references
11          public IActionResult Index()
12          {
13              return View();
14          }
15
            0 references
16          public string Message()
17          {
18              return "This is an important message about some product thing";
19          }
20      }
21  }
```

2. Run the app and manually type the URL as shown here. localhost:[port]/Products/Message. You should see the html string appear in the browser window.



3. Capture a screenshot of the app at this stage. Put the image into a Microsoft Word document with a caption explaining what you have just demonstrated.

4. Change the controller back to its original code as shown here.



```
0 references
public class ProductsController : Controller
{
    0 references
    public IActionResult Index()
    {
        return View();
    }

    0 references
    public IActionResult Message()
    {
        return View();
    }
}
}
```

IActionResult Side Note

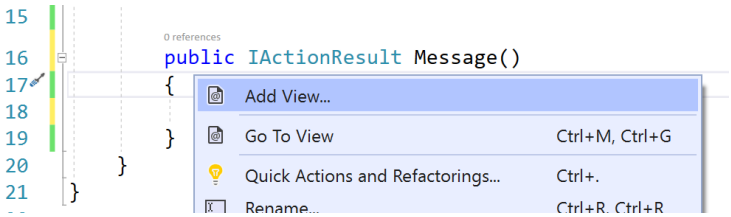
The Interface type IActionResult is used to define several types of responses that a method can produce, including the most common result, showing a view.

The full list of IActionResult implementations is listed below. The return **View( ) (**or **ViewResult**) statement is the most commonly seen type of IActionResult, but it is just one of several options for a controller response.

Microsoft.AspNetCore.Mvc.ActionResult
Microsoft.AspNetCore.Mvc.AntiforgeryValidationFailedResult
Microsoft.AspNetCore.Mvc.ContentResult
Microsoft.AspNetCore.Mvc.JsonResult
Microsoft.AspNetCore.Mvc.ObjectResult
Microsoft.AspNetCore.Mvc.PartialViewResult
Microsoft.AspNetCore.Mvc.RedirectResult
Microsoft.AspNetCore.Mvc.RedirectToActionResult
Microsoft.AspNetCore.Mvc.RedirectToPageResult
Microsoft.AspNetCore.Mvc.RedirectToRouteResult
Microsoft.AspNetCore.Mvc.StatusCodeResult
Microsoft.AspNetCore.Mvc.ViewComponentResult
Microsoft.AspNetCore.Mvc.ViewResult
Microsoft.AspNetCore.Mvc.Core.Infrastructure.IAntiforgeryValidationFailedResult
Microsoft.AspNetCore.Mvc.Infrastructure.IClientErrorActionResult
Microsoft.AspNetCore.Mvc.Infrastructure.IStatusCodeActionResult
Microsoft.AspNetCore.Mvc.ViewFeatures.IKeepTempDataResult

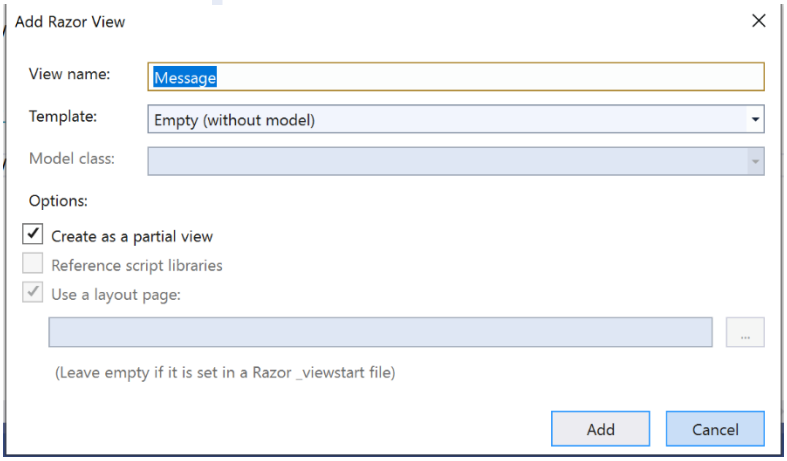
Refer to the readings in the syllabus for this lesson for more extensive information.

5. Right-click inside the Message method and choose Add View…

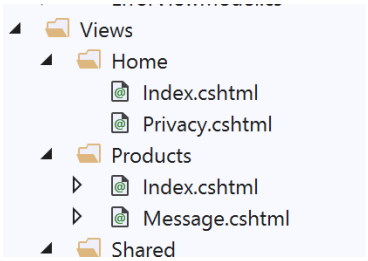


6. Choose Razor View.

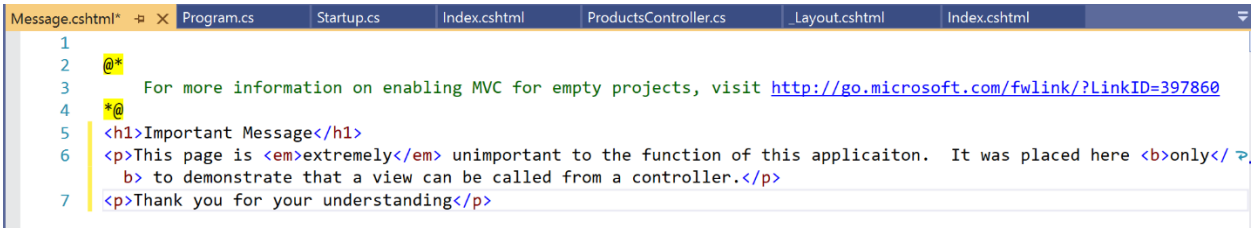
7. Leave the default choices as shown here: Message, Empty, and Partial View.



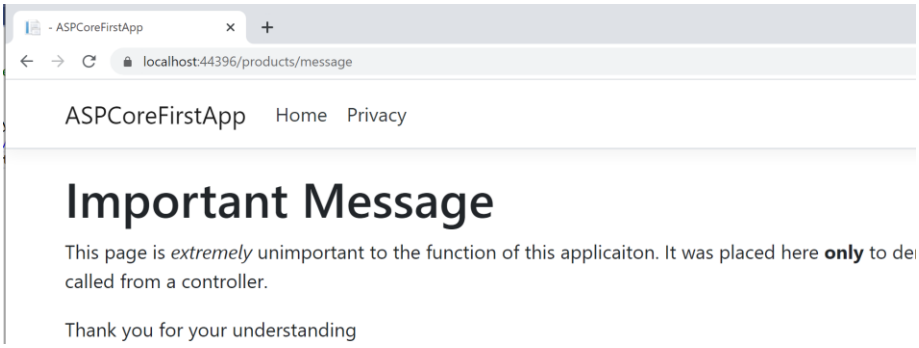
You should see the new View inside the Views > Products folder.



8. Add some HTML content to the **message.cshtml** page. Write something that demonstrates a few HTML or CSS tags.



9. You should be able to navigate to this new page with the following url: https://localhost:44396/products/message



10. Capture a screenshot of the app at this stage. Put the image into a Microsoft Word document with a caption explaining what you have just demonstrated.

Route Parameters

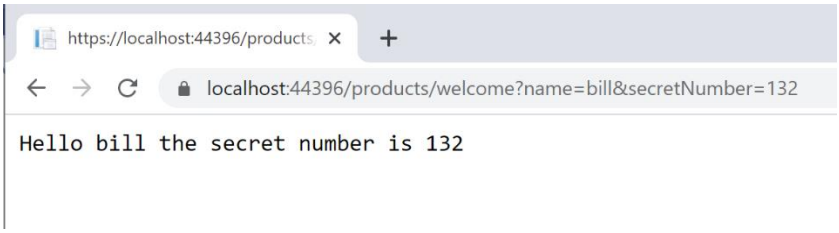
Let's modify the controller so that you can pass some parameter information from the URL to the controller. For example, */Products/Welcome?name=Bill&secretNumber=4* will be the URL with two parameters.

1. Change your ProductsController to add a Welcome method that includes two parameters as shown below. Note that the code uses the C# optional-parameter feature to indicate that the secretNumber parameter should default to 13 if no value is passed for that parameter.

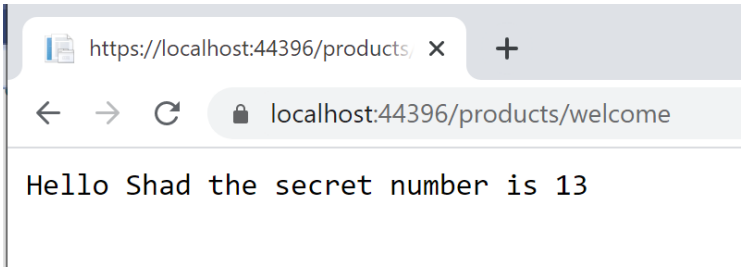
```
21
        0 references
22      public string Welcome(string name, int secretNumber = 13)
23      {
24          return HttpUtility.HtmlEncode("Hello " + name + " the secret number is " + secretNumber);
25      }
26  }
27  }
```

Security Note: The code above uses **HttpUtility.HtmlEncode** to protect the application from malicious input (namely JavaScript). The code would work equally well as string without the extra method call but would be less secure.

2. If you type the following URL, you should see the new parameters in the webpage.



3. If you omit the parameters, the default values are used.



4. Modify the Welcome method to include some data in the ViewBag.

```
16
                0 references
17              public IActionResult Welcome()
18              {
19                  ViewBag.name = "Shad";
20                  ViewBag.secretNumber = 13;
21                  return View();
22              }
```

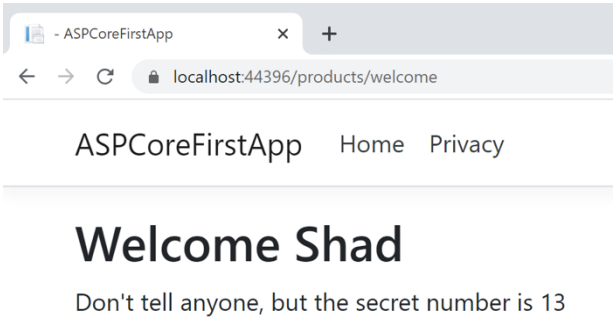
ViewBag is a loosely typed collection of data. Unlike strong types, loose types means that you don't explicitly declare the type of data you're adding to the object. You can use the collection of weakly typed data for passing small amounts of data in and out of controllers and views. In most current applications, ViewBag is no longer used to pass data, but you will see examples of ViewBag in applications if you are an ASP.NET developer. Usually, a strongly typed class is passed, which we will get to in future lessons.

5. Create a view called **Welcome.cshtml** and display the contents of the **ViewBag**.

Welcome.cshtml | Message.cshtml | Program.cs | Startup.cs | Index.cshtml | ProductsCont

```
1   <h2>Welcome @ViewBag.name</h2>
2   <p>Don't tell anyone, but the secret number is @ViewBag.secretNumber</p>
```

You should be able to see similar results with the same URL as before. However, this page has HTML and CSS formatting because we are seeing a View page.



6. Capture a screenshot of the app at this stage. Put the image into a Microsoft Word document with a caption explaining what you have just demonstrated.

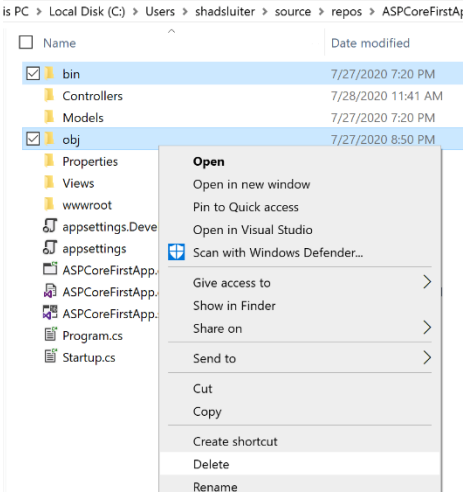
Key Learnings

So far in this course, you have seen Controllers and Views. That is 2/3 of the pieces of the MVC pattern! How hard can it get from here?

Here are some key concepts you should have learned in this activity:

- A route is a URL pattern that is mapped to a handler called a controller.
- A controller is responsible for returning the output to the browser.
- A controller can have many methods called actions.
- A controller can return a string, return a View which is a separate html page or can return a number of other messages, such as JSON formatted data.
- A View is a file with HTML and other code that is displayed as part of the application webpage when called upon by the controller.

Deliverables:



1. This activity has multiple parts. Complete all parts before submitting.
2. Submit a Microsoft Word document with screenshots of the application being run. Show each screen of the output and put a caption under each picture explaining what is being demonstrated.
3. In the same document, in one paragraph, write a summary of the key concepts that were demonstrated in this lesson. Be sure to explain the key words introduced in this lesson.
4. Submit a ZIP file of the project file. In order to save space, you can delete the bin and the obj folders of the project. These folders contain the compiled version of the application and are automatically regenerated each time the build or run commands are executed.

Part 2 Login

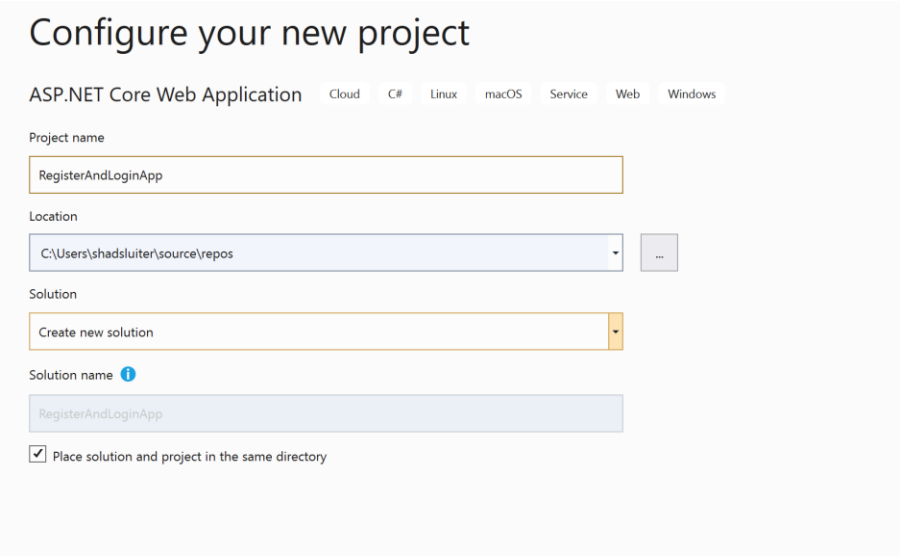
Goals:

In this activity, you use Routes, Controllers, Models, and Views to build a simple Login Page using a N-Layer Web Application Architecture. Complete the following tasks for this activity:

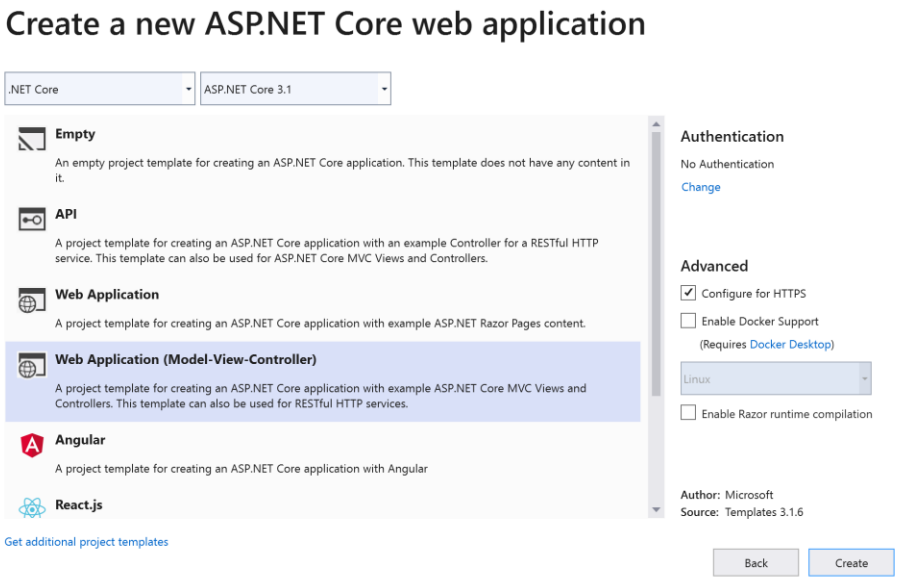
1. Create a Login view, route and controller.
2. Create a user Model class.
3. Bind the login model to the view.
4. Create a data security class and integrate it with the login page.

Step-by-Step Instructions

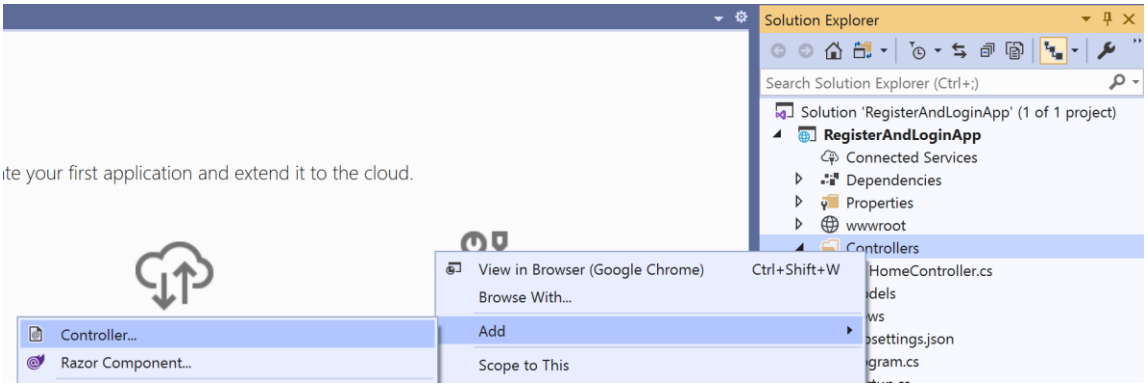
1. Create a new .NET MVC Application:
    a. Select File → New Project. Choose the ASP.NET Web Application. Name your application **RegisterAndLoginApp**.
    b. Click Create.



    c. Select the **Web Application (Model-View-Controller)** template. Click Create.

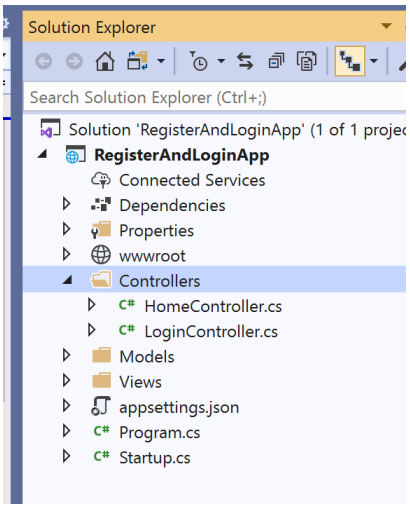


d. In the Controllers folder, right-click and choose Add → Controller… menu items.



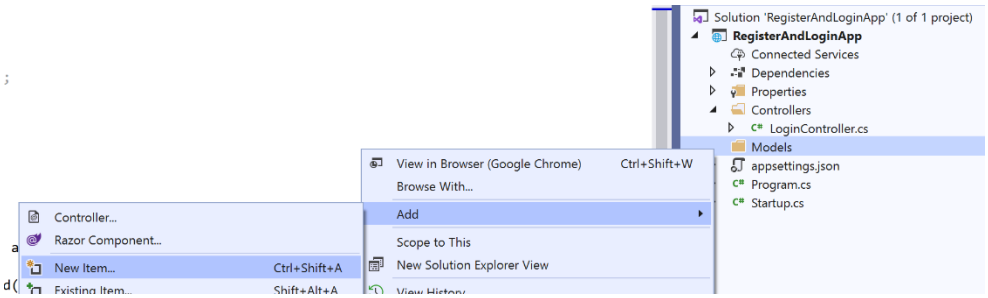
e. Select the 'MVC Controller – Empty' type. Click the Add button. Name your Controller 'Login' and click the Add button.

f. You should see the Login Controller file as a LoginController.cs class in the Controllers folder.

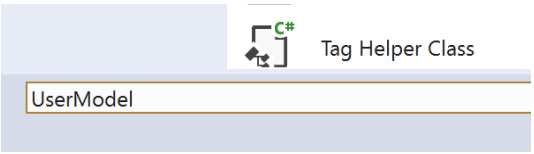


Add a User Model

1. Add a new Item to the Models folder. Right-click on the Models folder. Choose Add > New Item.

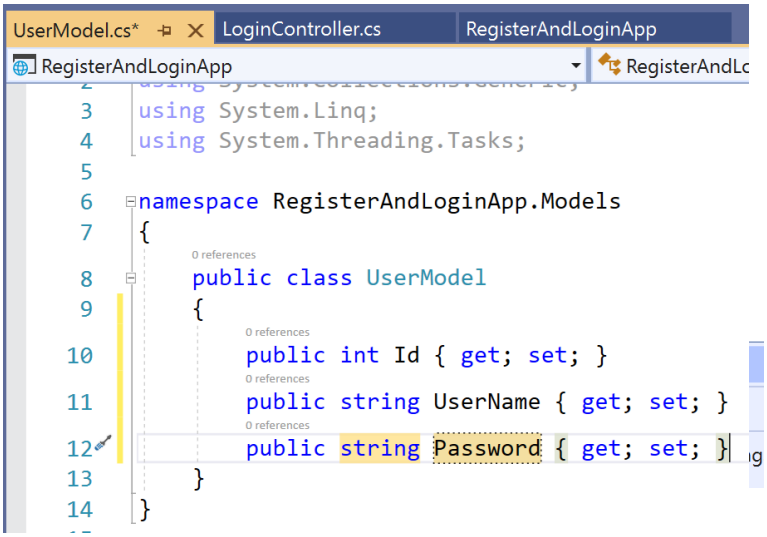


2. Select Class and enter **UserModel** for the name. Click Add.

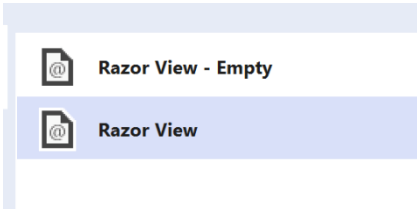


3. Add 3 properties, **Id**, **Username**, and **Password**, to the UserModel class with both setter and getter methods.

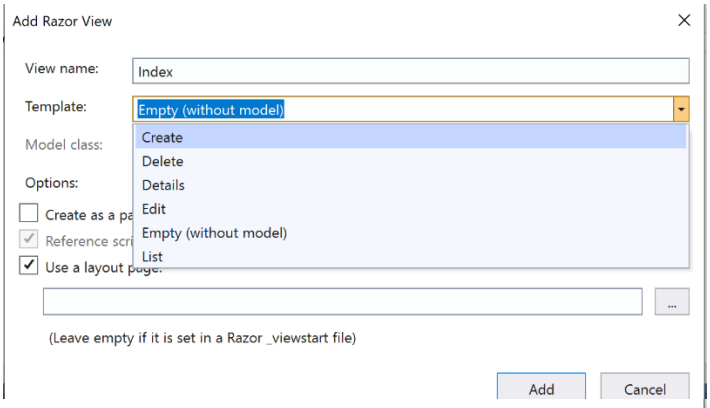
4. Create a Login View:
    a. Right-click on any part of the code body for the Login Controller Index(). Select the **Add View** menu option.



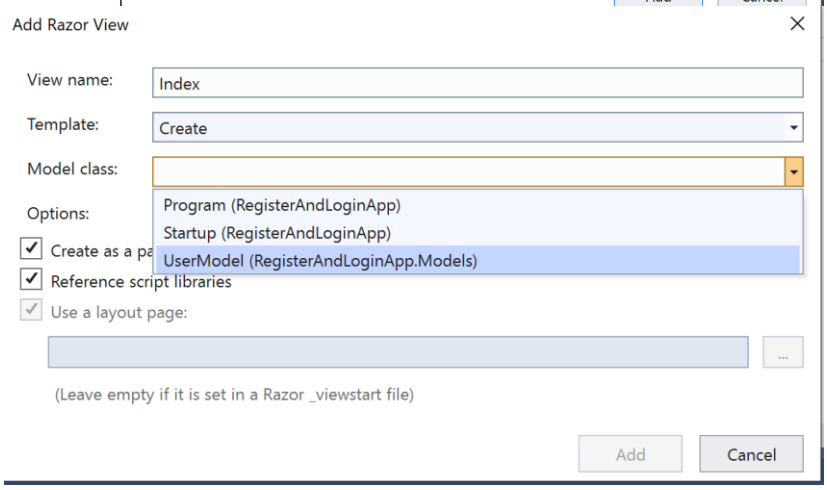
5. Select **Razor View**.



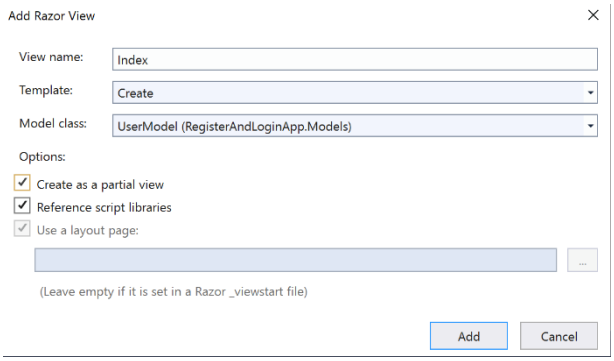
6. Leave the View name as **Index**.
7. Choose the **Create** template.



9. From the **Model** class dropdown, select the **UserModel** created in step 3.



10. Choose the following options:



11. Click Add.

Inspect the Login Page (as Index.cshtml) within the Views/Login folder. You should see a lot of code that is automatically generated for you based on the properties of the UserModel class. This code will display a data entry form.

12. Modify the page to make it into a login form instead of a "Create" form.
    a. Change the **title**.
    b. Delete the section for the **Id property**.
    c. Change the password input **to type password**.

d. Change the text of the submit button to **Login**.

13. Validate the Login Page by running the solution using localhost:[port]/Login as the URL.

14. Capture a screenshot of the app at this stage. Put the image into a Microsoft Word document with a caption explaining what you have just demonstrated.

Bind the Login View and Login Controller with a Controller method

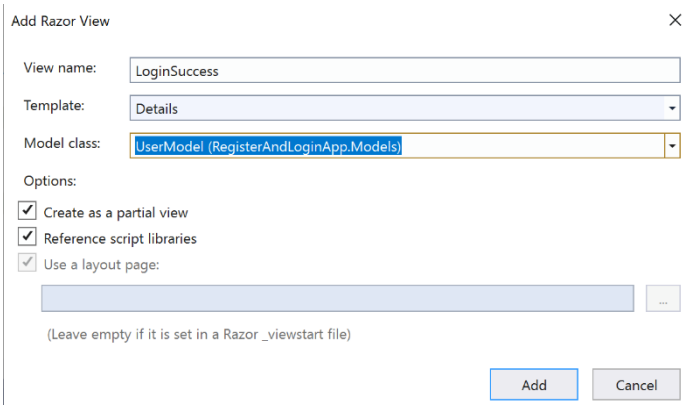
1. Add a **ProcessLogin**() method to the **Login** controller that handles the Login View Form Post. This version of the login handler displays the username and password and validates only one user. In future code, we will authenticate the user with a database connection.

Notice that in this method, we pass two parameters: the first parameter is the view's name (LoginSuccess), and the second is the data object (UserModel) that the form will display.

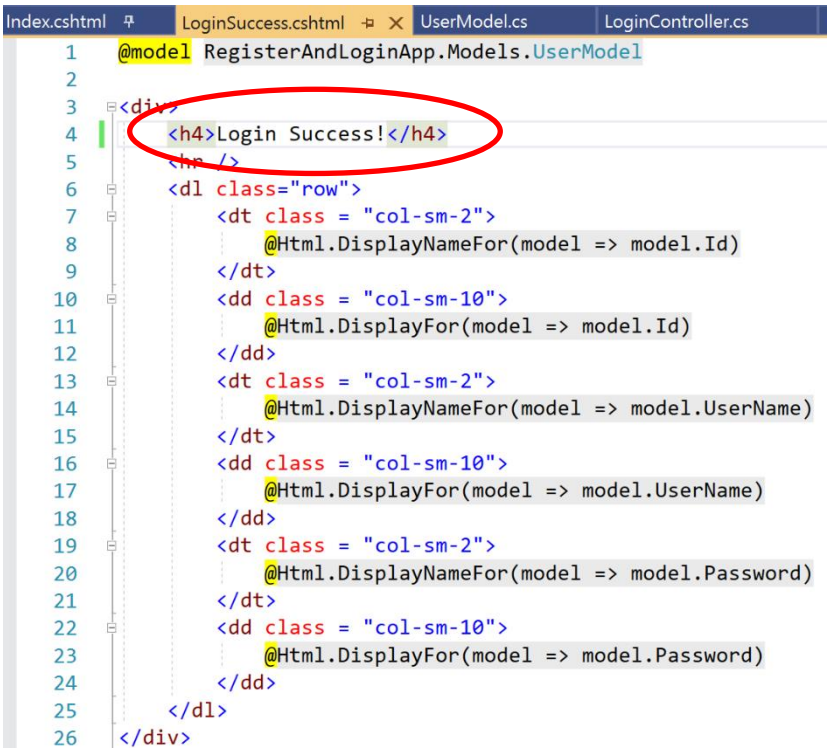
```
x.cshtml | LoginSuccess.cshtml | UserModel.cs | LoginController.cs | RegisterAndLoginApp
egisterAndLoginApp  |  RegisterAndLoginApp.Controllers.LoginController
```

```
8   namespace RegisterAndLoginApp.Controllers
9   {
        0 references
10      public class LoginController : Controller
11      {
            0 references
12          public IActionResult Index()
13          {
14              return View();
15          }
16
            0 references
17          public IActionResult ProcessLogin(UserModel user)
18          {
19              if (user.UserName == "BillGates" && user.Password == "bigbucks")
20              {
21                  return View("LoginSuccess", user);
22              }
23              else
24              {
25                  return View("LoginFailure", user);
26              }
27          }
28      }
29  }
```

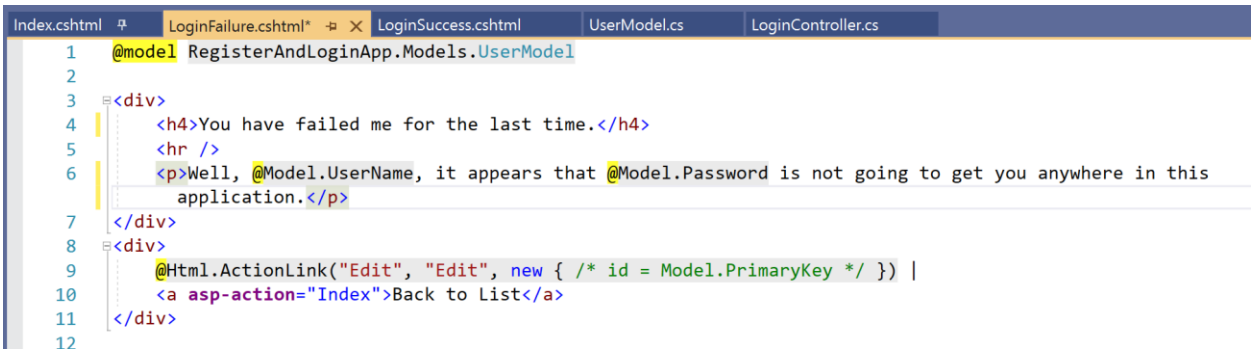
2. Right-click inside the ProcessLogin method to create a new View:
    a. Name the view LoginSuccess.
    b. Choose the Details template.
    c. Select the UserModel.



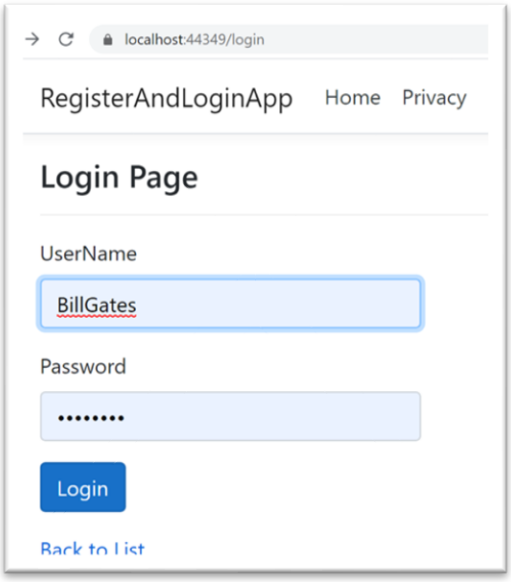
3. Modify the HTML to tell the user that the login was successful.

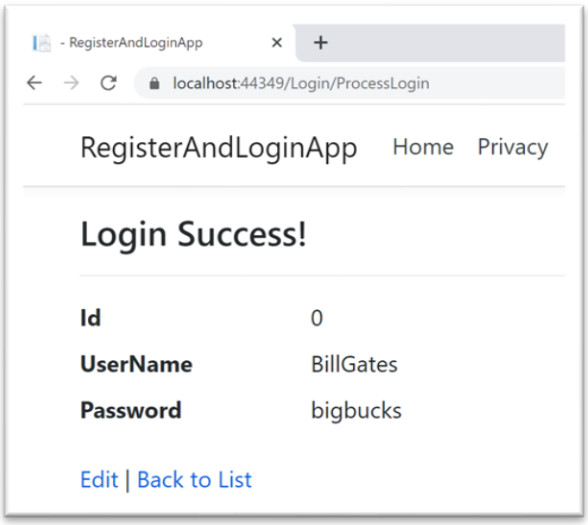


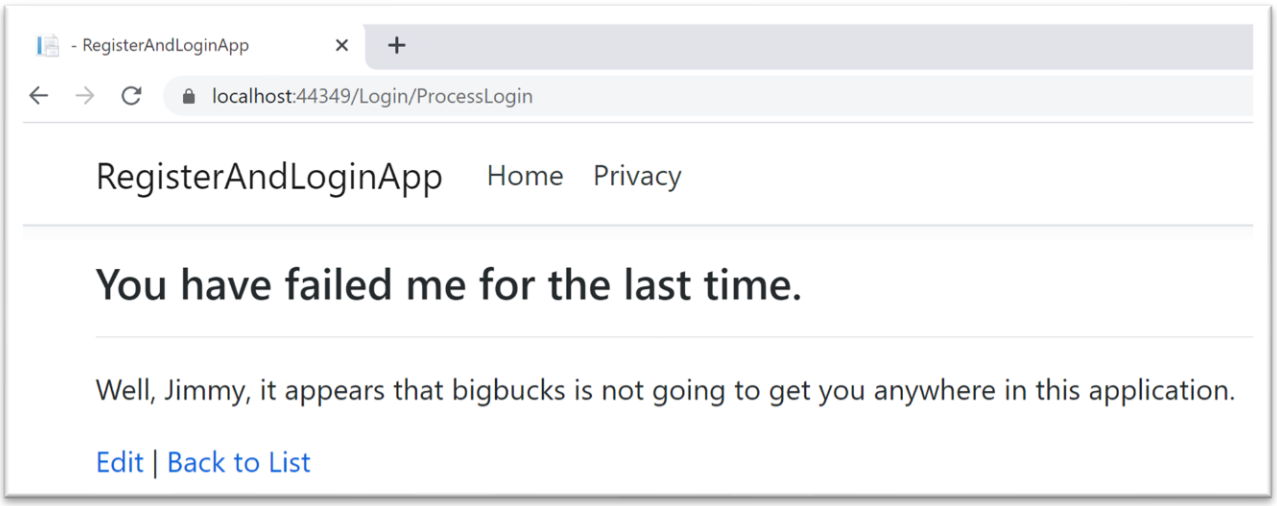
4. Add another View called "LoginFailure" that is used to tell the user that the login was not successful. I decided to simplify the form to show only two lines of text…



You should be able to test the login page with both a success and failure:







5. Capture a screenshot of the app at this stage. Put the image into a Microsoft Word document with a caption explaining what you have just demonstrated.

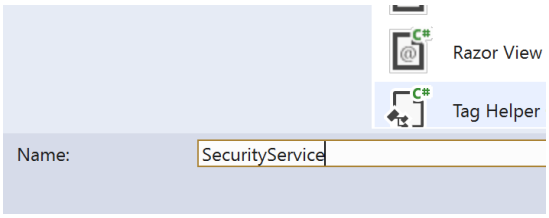
Authenticate Multiple User Login

In this stage of the process, we will create a class that contains the correct username and password. This code is for demonstration. Ideally, we should connect to a database, lookup the usernames and passwords, and determine if the correct credentials were supplied. However, for simplicity, we will continue to have user data hardcoded in the program.

1. Right-click on the RegisterAndLogin project name.
2. Select Add → New Folder.

3. Name the folder **Services**.

4. Right-click on the Services folder and select Add → New Class.
5. Name the class SecurityService.



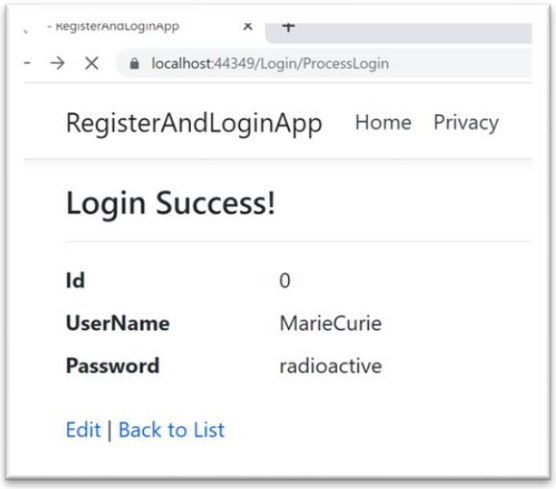
6. Add the following code to the new class. Create your own usernames and passwords.

```
namespace RegisterAndLoginApp.Services
{


    1 reference
    public class SecurityService
    {
        List<UserModel> knownUsers = new List<UserModel>();

        0 references
        public SecurityService()
        {
            knownUsers.Add(new UserModel { Id = 0, UserName = "BillGates", Password = "bigbucks" });
            knownUsers.Add(new UserModel { Id = 1, UserName = "MarieCurie", Password = "radioactive" });
            knownUsers.Add(new UserModel { Id = 2, UserName = "WatsonCrick", Password = "dna" });
            knownUsers.Add(new UserModel { Id = 3, UserName = "AlexanderFlemming", Password = "penicillin" });
        }

        0 references
        public bool IsValid(UserModel user)
        {
            return knownUsers.Any(x => x.UserName == user.UserName && x.Password == user.Password);
        }
    }
}

```

7. Modify the Login controller to utilize the new service.

```
11  public class LoginController : Controller
12  {
13      public IActionResult Index()
14      {
15          return View();
16      }
17
18      public IActionResult ProcessLogin(UserModel user)
19      {
20          SecurityService securityService = new SecurityService();
21
22          if (securityService.IsValid(user))
23              return View("LoginSuccess", user);
24          else
25              return View("LoginFailure", user);
26      }
27  }
28  }
```

8. Test out the login page.

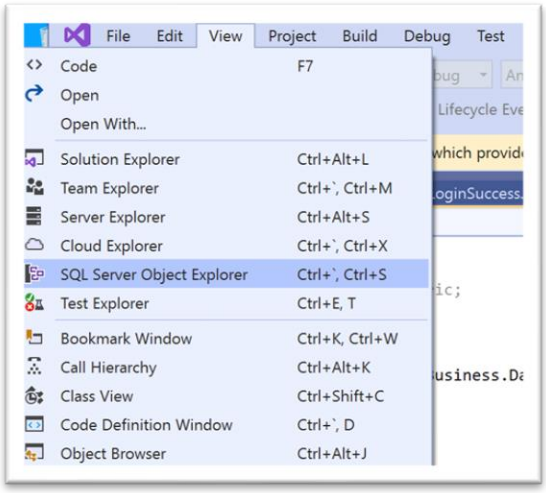


9. Capture a screenshot of the app at this stage. Put the image into a Microsoft Word document with a caption explaining what you have just demonstrated.

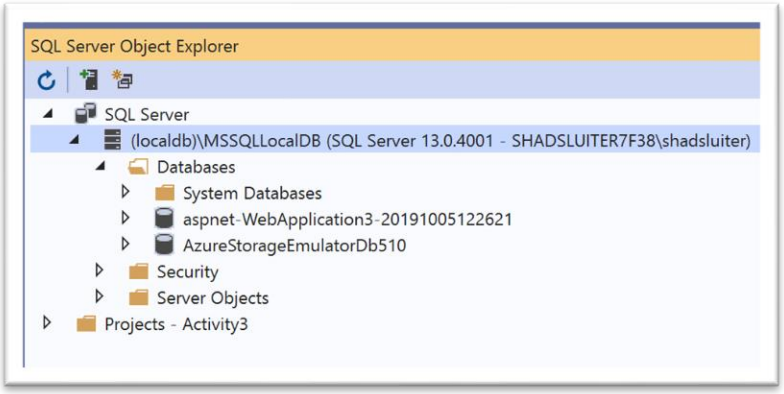
Create a Database Schema and Table

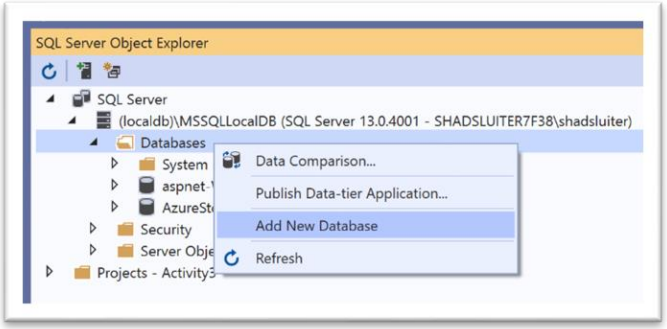
In this section, we will add an actual database that will store usernames and password.

1. Open SQL Server Object Explorer by selecting the **View → SQL Server Object Explorer** menu items. By default, under the SQL Servers you should see a local DB called MSSQLLocalDB listed. Expand this database server.

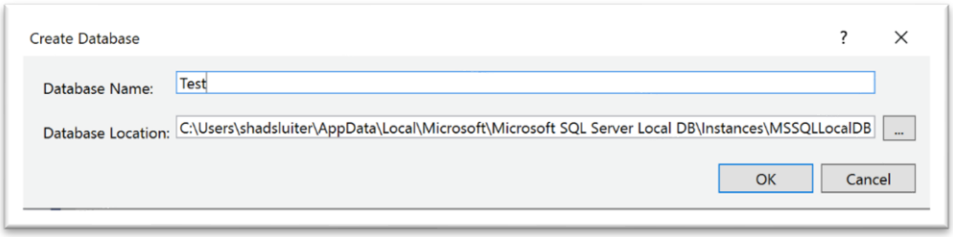


2. If this server is not listed:
   a) Click the Add SQL Server icon.
   b) Expand the Local tree.
   c) Select the MSSQLLocalDB.
   d) Select Windows Authentication.
   e) Click the Connect button to add the local database server to the list of database servers in SQL Server Object Explorer.

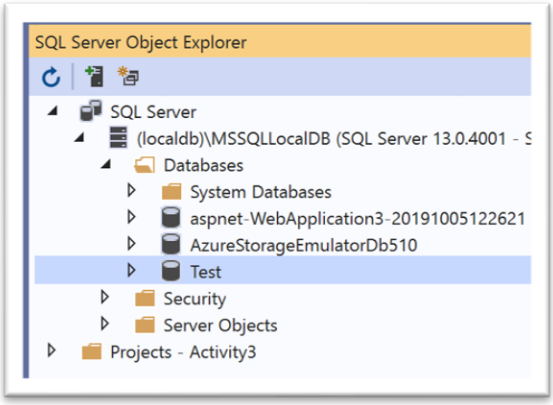




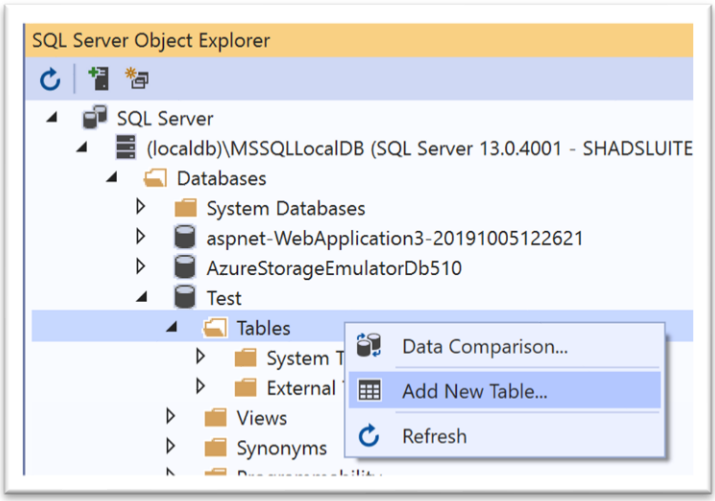
3. Right-click on the Databases folder and select Add New Database. Enter a Database Name of **Test**.



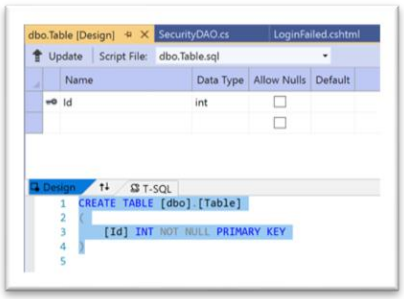
4. A new Database should be now available under the MSSQLLocalDB databases.



5. Expand the Test database to display the Tables folder.
   a) Right-click on the **Tables** folder and select **Add New** table menu option.



You should see the **Table designer**.



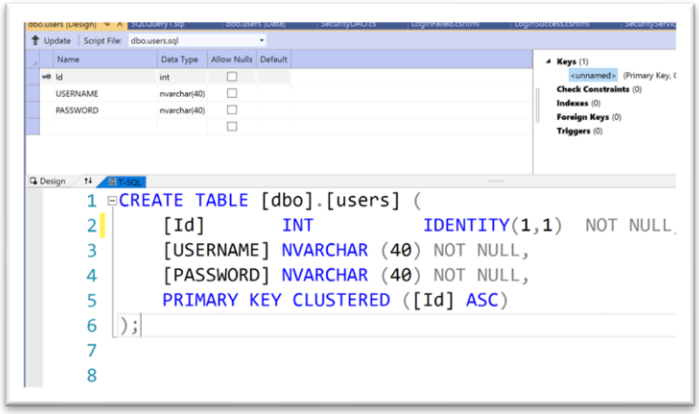
b) In the T-SQL window, change the name of the Table in the Create statement to Users.

c) Add the following columns:

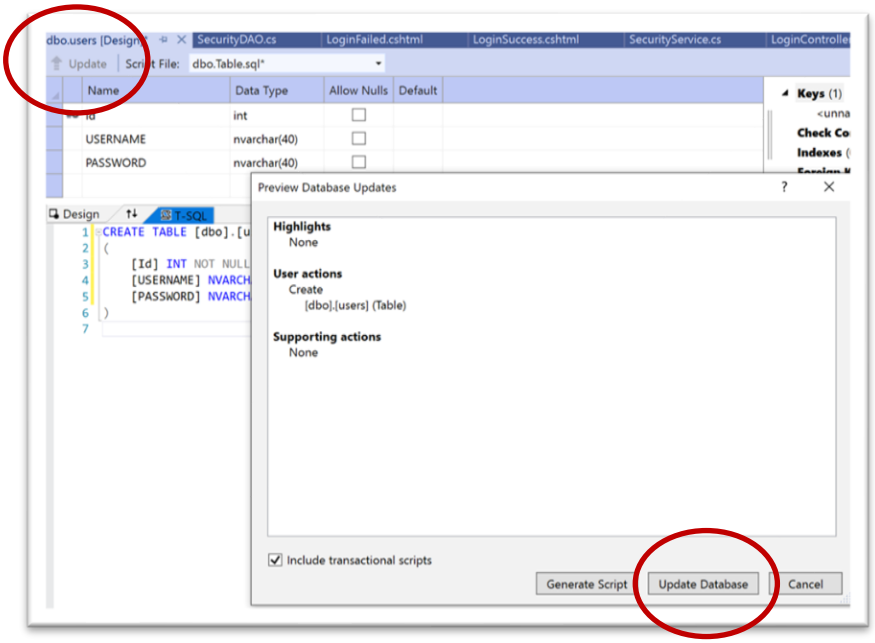
i. ID, primary key, type int, no NULL's, with auto increment (in the T-SQL window add IDENTITY(1,1) to the script.

ii. USERNAME, type nvarchar(50), no NULL's

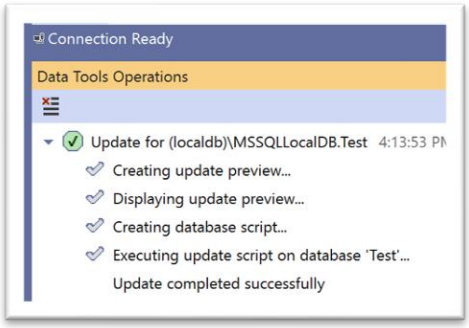
iii. PASSWORD, type nvarchar(50), no NULL's



d) Click the **Update** button from the designer.



e) In the Preview Database Updates dialog, click the **Update** Database button.



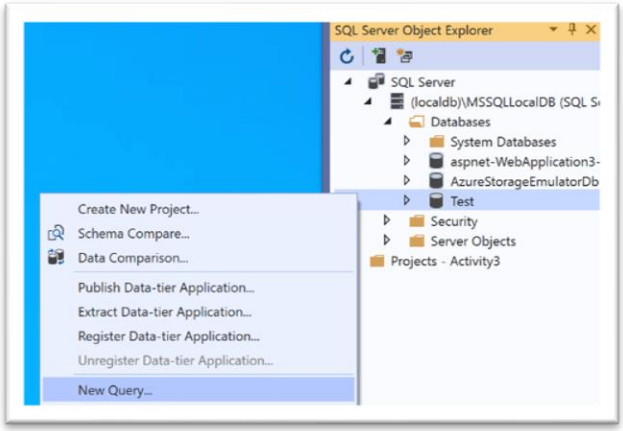
f) In the Output window, validate that the table and columns were created without errors.

Validate your Test Database

1. Open SQL Server Object Explorer by selecting the View → SQL Server Object Explorer menu items. By default, under the SQL Servers you should see a local DB called MSSQLLocalDB listed.
   a) If this database is NOT listed, click the 'Add SQL Server' icon.
   b) Expand the Local tree.
   c) Select the MSSQLLocalDB.
   d) Select Windows Authentication.
   e) Click the Connect button to add the local database to the list of databases in SQL Server Object Explorer.

2. Expand the MSSQLLocalDB database server and under the Databases folder select the Test database.

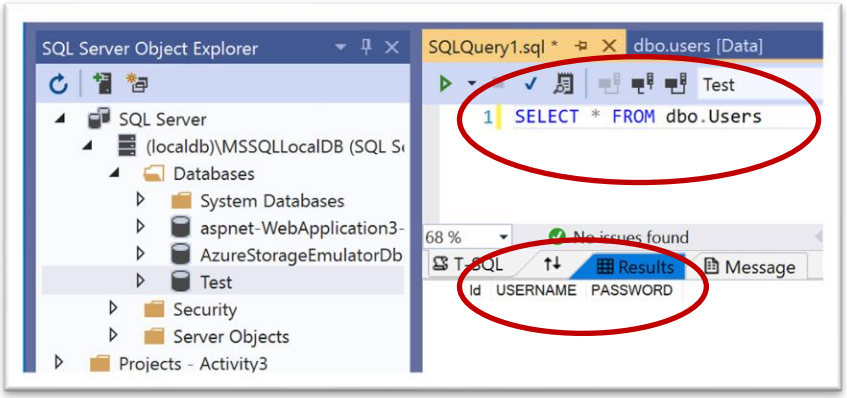
3. Right-click on the Test database and select the **New Query** menu option.



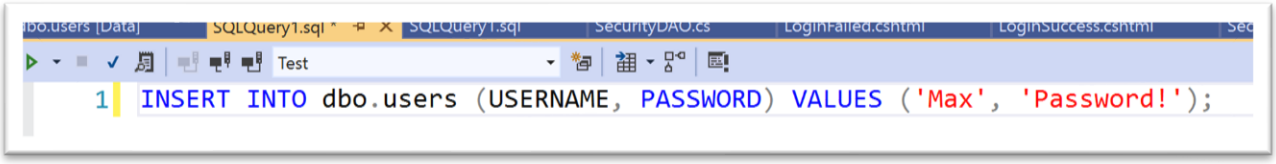
4. In the query run execute the following **SQL Statement:**

SELECT * FROM dbo.Users

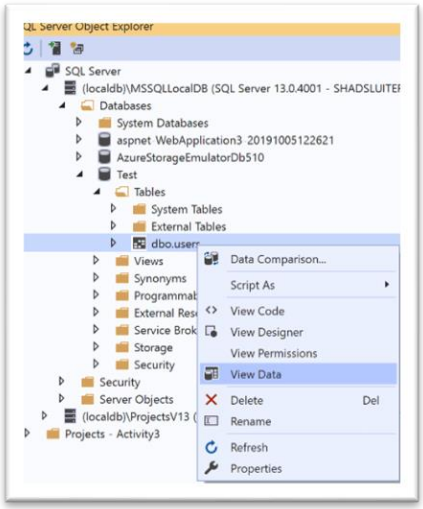
The Message window should display the 3 columns of the Users table with no data.



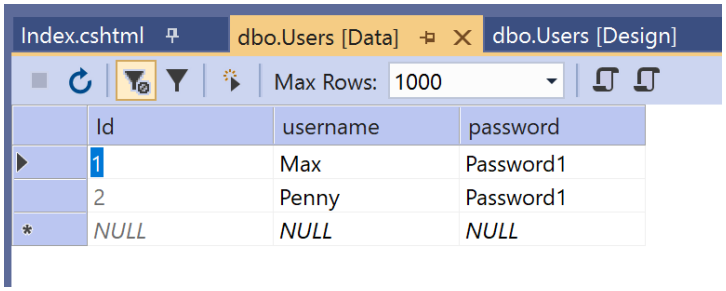
5. Insert a few new rows in the table by supplying a username and password. Use the following SQL Statement as an example.



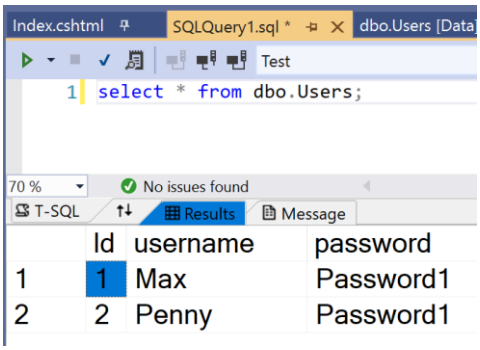
6. Confirm that the new data is saved in the table.



You can view the data with one of two methods. First you can right-click on the dbo.users table and select View Data.



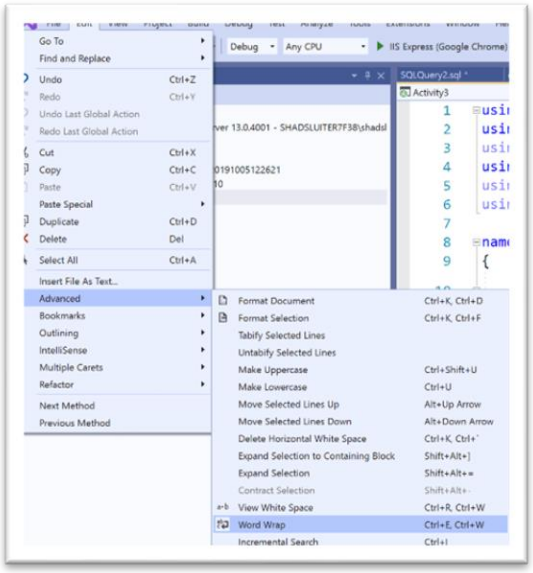
The second method to show all data is to run a SQL query to select all columns from the Users table.



Add a Security DAO using ADO.NET:

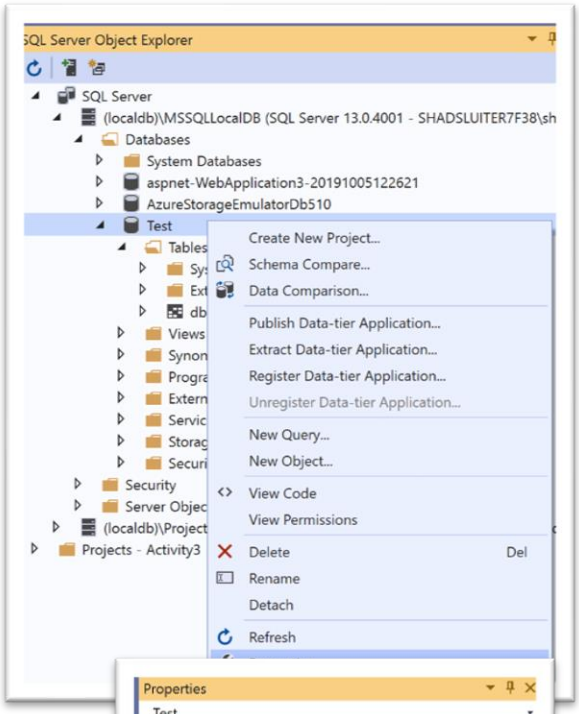
In this section, we will connect to the database from the C# code. In the application design that we are using, the class that is supposed to handle all SQL queries is the SecurityDAO. Recall that DAO is "Data Access Object," meaning that its job is to perform data access work.

1. Create a new class in the Services folder called SecurityDAO.

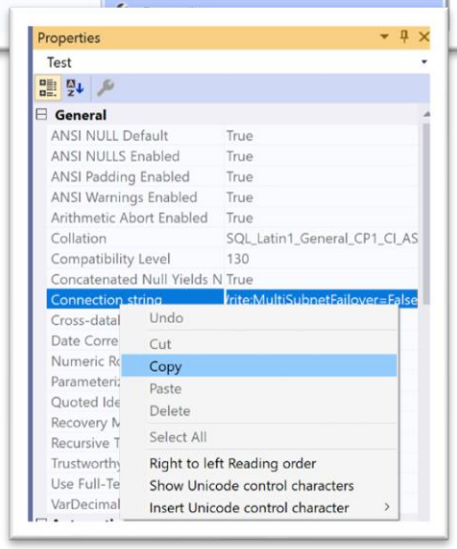


You may wish to start by turning on **Word Wrap** in the code editor. The connection string and SQL queries can easily become wider than the width of the text editing window. Choose **Edit > Advanced > Word Wrap** to toggle the feature on or off.

a. Get the properties for the Test database. Right-click the Test database and choose Properties.



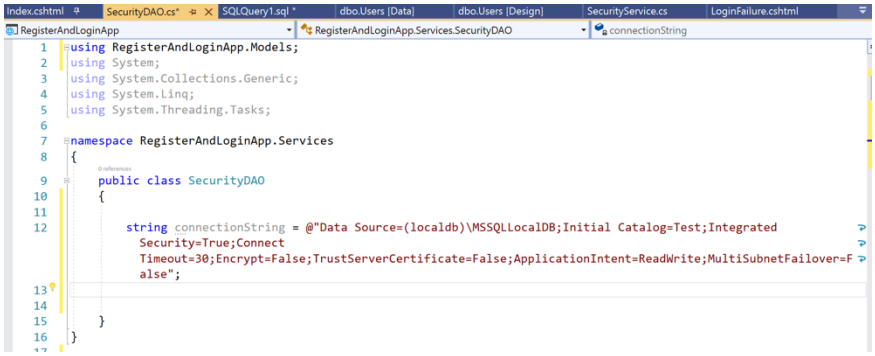
b. In the properties window you can right-click and copy the Connection String property.



c. Save the Connection string in a variable in the code for SecurityDAO as shown here.

Here is a picture of the database connection string used in my application.

Now this class can run SQL queries against the Test database in any of the class methods.



2. Create a SQL query in the FindUserByNameAndPassword() method to search for a record by username.

```
Index.cshtml | SecurityDAO.cs | SQLQuery1.sql * | dbo.Users [Data] | dbo.Users [Design] | SecurityService.cs | LoginFailure.cshtml
RegisterAndLoginApp | RegisterAndLoginApp.Services.SecurityDAO | FindUserByNameAndPassword(UserMo
```

```
using System.Linq;
using System.Threading.Tasks;

namespace RegisterAndLoginApp.Services
{
    2 references
    public class SecurityDAO
    {

        string connectionString = @"Data Source=(localdb)\MSSQLLocalDB;Initial Catalog=Test;Integrated Security=True;Connect
          Timeout=30;Encrypt=False;TrustServerCertificate=False;ApplicationIntent=ReadWrite;MultiSubnetFailover=False";

        1 reference
        public bool FindUserByNameAndPassword(UserModel user)
        {
            // assume nothing is found
            bool success = false;

            // uses prepared statements for security. @username @password are defined below
            string sqlStatement = "SELECT * FROM dbo.Users WHERE username = @username and password = @password";

            using (SqlConnection connection = new SqlConnection(connectionString))
            {
                SqlCommand command = new SqlCommand(sqlStatement, connection);

                // define the values of the two placeholders in the sqlStatement string
                command.Parameters.Add("@USERNAME", System.Data.SqlDbType.VarChar, 50).Value = user.UserName;
                command.Parameters.Add("@PASSWORD", System.Data.SqlDbType.VarChar, 50).Value = user.Password;

                try
                {
                    connection.Open();
                    SqlDataReader reader = command.ExecuteReader();

                    if (reader.HasRows)
                        success = true;
                }
                catch (Exception ex)
                {
                    Console.WriteLine(ex.Message);
                };
            }
            return success;
        }
    }
}
```

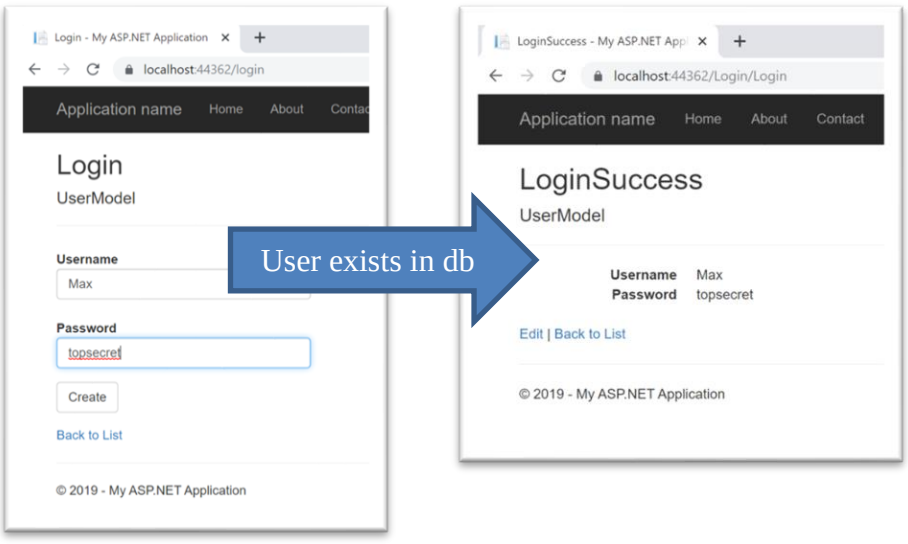
The code above is derived from the example given at https://docs.microsoft.com/en-us/dotnet/api/system.data.sqlclient.sqlcommand.parameters?view=netframework-4.8

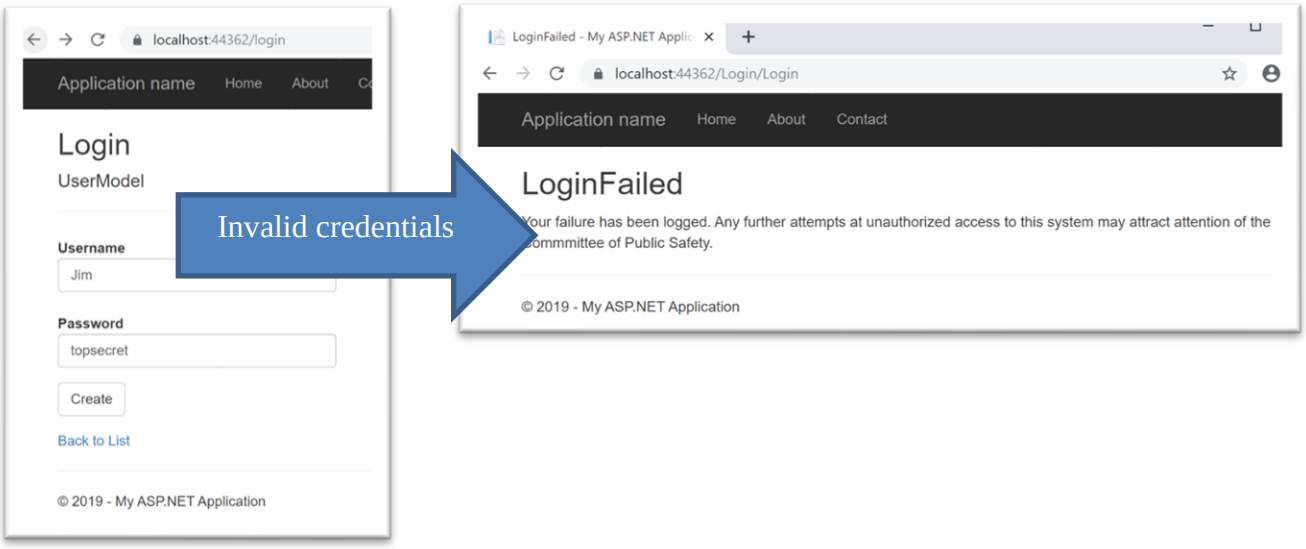
3. Update the SecurityService to utilize the database lookup.

```
using System.Threading.Tasks;

namespace RegisterAndLoginApp.Services
{


    2 references
    public class SecurityService
    {
        SecurityDAO securityDAO = new SecurityDAO();

        1 reference
        public bool IsValid(UserModel user)
        {
            return securityDAO.FindUserByNameAndPassword(user);
        }
    }
}
```

a) Test the Controller and Views with the browser by navigating to /Login. Try to login with the username and password data you provided earlier. Attempt to login with an invalid username or an invalid password.

Capture a screenshot of the app at this stage. Put the image into a Microsoft Word document with a caption explaining what you have just demonstrated.





Here are more resources for how to connect a C# .NET app to a database.

1. Channel 9 is Microsoft's Official video channel: https://channel9.msdn.com/
2. T-SQL Tutorial shows all of the possible SQL commands available to your app: https://www.tutorialspoint.com/t_sql/t_sql_quick_guide.htm

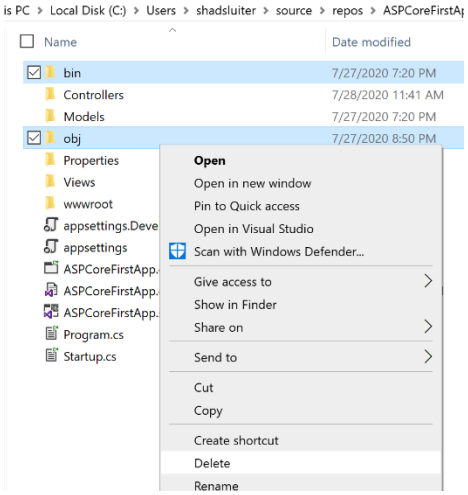
3. ADO.NET Documentation:
    a. Follow links and references to SQL Server or SqlClient.
    b. For API help:
        i. https://docs.microsoft.com/en-us/dotnet/api/
        ii. For ADO.NET search for System.Data.SqlClient
    c. For ADO.NET Tutorial:
        i. https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/retrieving-and-modifying-data
        ii. http://csharp-station.com/Tutorial/AdoDotNet
        iii. https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ado-net-code-examples.

Key Learnings

1. Controllers and Views usually share data in the form of Model objects. In this example we used UserModel since we were creating a login service. In other cases, you will use other class types such as products, tickets, customers, invoices, orders etc.
2. A view frequently has @Model on the very first line of the page to indicate what type of data is being displayed.
3. Services in an nLayered design are encapsulated in classes. Each layer communicates with the items one level up and one level below.
4. SQL commands are required to access a database.
5. The SecurityDAO class should have more methods than the one items we created. As we develop new features in the application, we will add more data access calls.

Deliverables:

6. Submit a Microsoft Word document with screenshots of the application being run. Show each screen of the output and put a caption under each picture explaining what is being demonstrated.
7. In the same document, in one paragraph, write a summary of the key concepts that were demonstrated in this lesson.
8. Submit a ZIP file of the project file. In order to save space, you can delete the bin and the obj folders of the project. These folders contain the compiled version of the application and are automatically regenerated each time the build or run commands are executed.



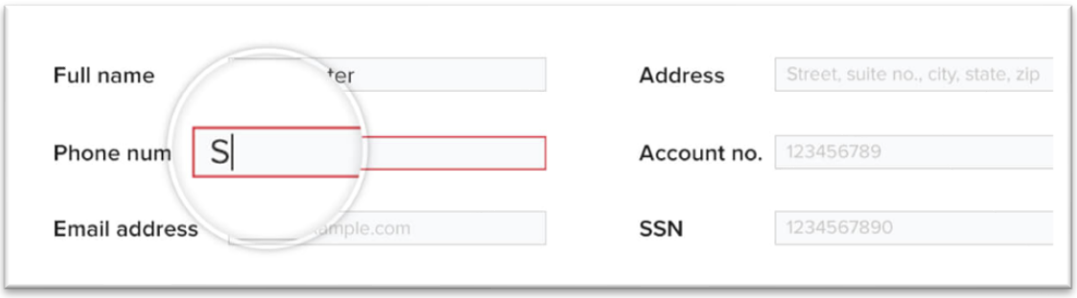
Part 3 Data Validation

Goal:

In this activity, you will validate the format of data entered into a form.

Introducing Data Validation

Most data entry forms should have rules for what type of data the user can enter. For example, a phone number should only accept digits or an email address should have the format name@domain.com.



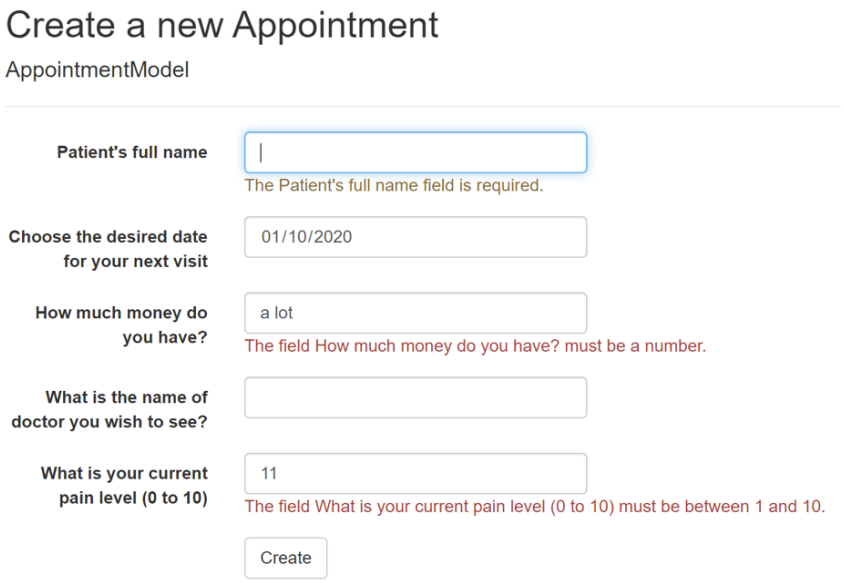
The validation support provided by ASP.NET MVC allows you to specify validation rules in the model class and the rules are enforced *everywhere* in the application.

Project Preview

The goal of this project is to demonstrate the use of data validation rules in a data entry form.

We will see a variety of data types (string, date, currency, and integer) in the data entry form. Some fields will not be permitted to be left blank. Some will require a specific type of number or number range.

This form is used to make an appointment at a medical clinic.



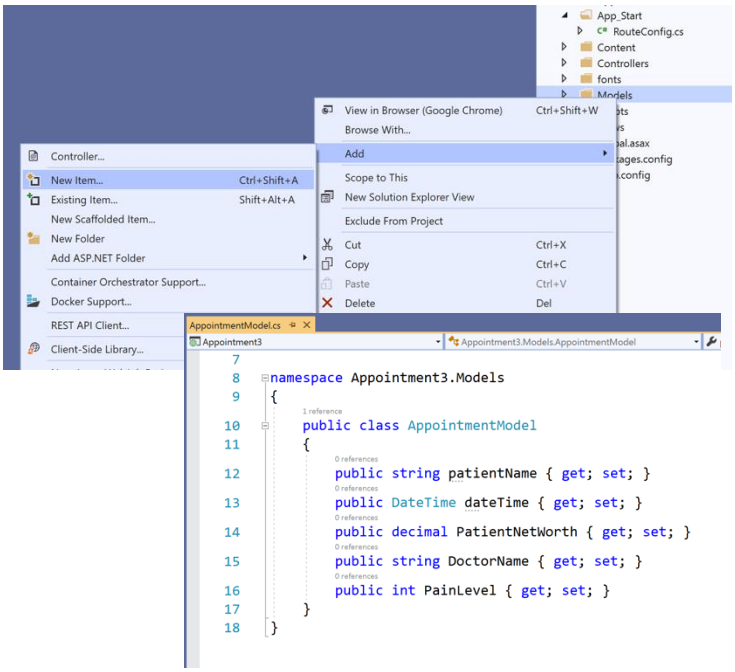
Step-by-Step Instructions

Create a new Project

2. Start a new MVC .NET project.
3. I will name the project AppointmentMaker.
4. Choose the following options:
    - Web Application (Model-View-Controller) project
    - No authentication

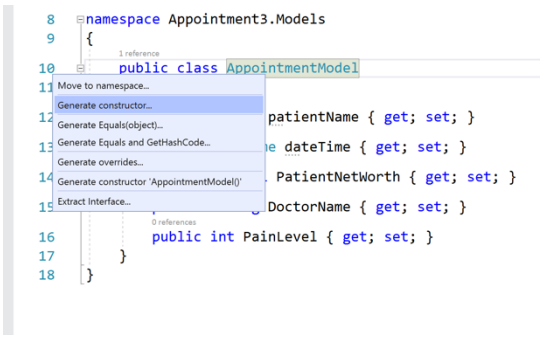
Create a Model for an Appointment

1. Right-click in the **Models** folder and choose **Add → New Item**.
2. Choose **Class**.
3. Name the class **AppointmentModel**.
4. Click the Add button.

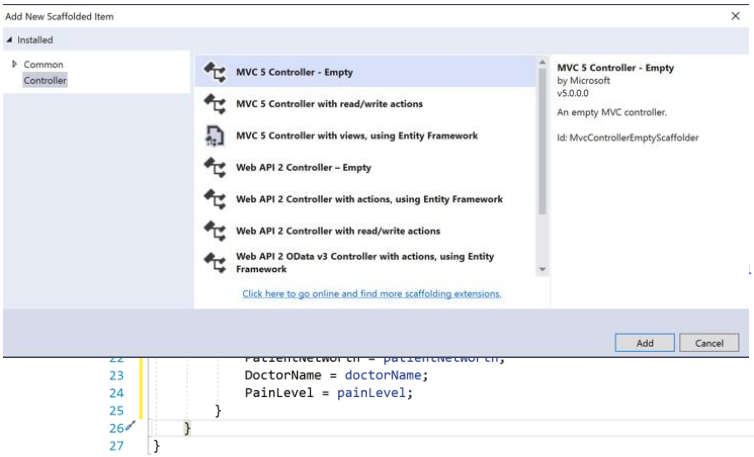


5. Create the following properties
    - patientName (string)
    - dateTime (date)
    - PatientNetWorth (decimal)
    - DoctorName (string)
    - PainLevel (int)

6. Create a constructor.
    a) Right-click inside the **AppointmentModel** and choose **Quick Actions and Refactoring** (or press **CTRL + .**).
    b) Choose **Generate Constructor**.
    c) Select all properties and click **OK**.

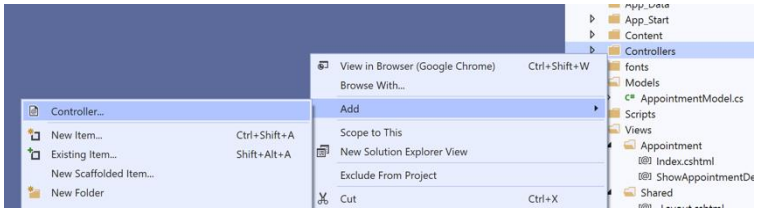


You should see a new constructor in the code.



Create a controller

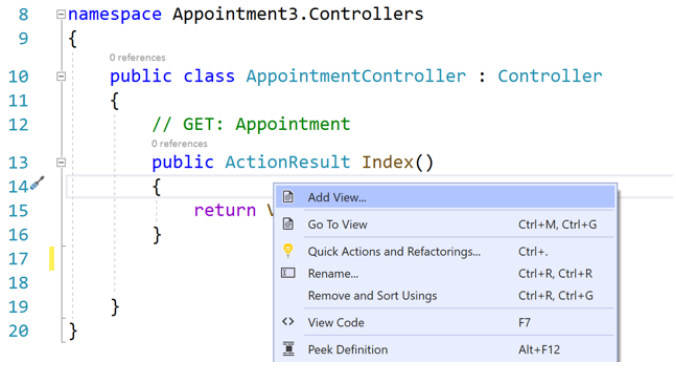
1. Right-click in the controller's folder and create a new empty controller.



2. Name the controller **AppointmentsController**.

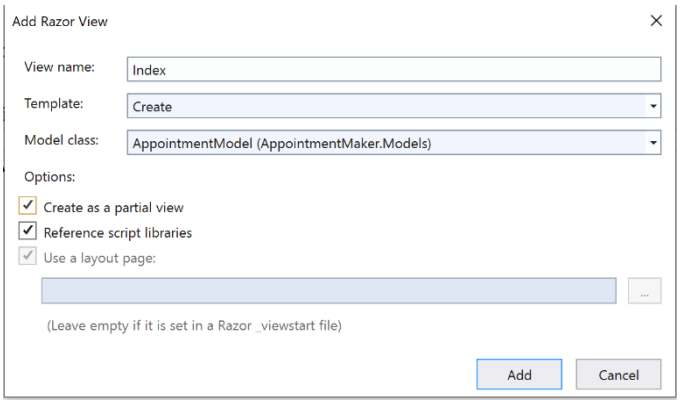
Create an input form for the appointment

1. In the AppointmentController, right-click inside the index method. Choose Add View.

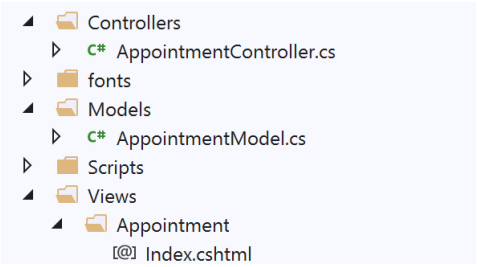


2. For the View options choose:

- **Index** for the view name
- **Create** for the template
- **AppointmentModel** for the Model class
- Reference script libraries checked



You should also notice that the index.cshtml file was placed in the Views folder.



You should see the index.cshtml file has a lot of HTML and Razor code that is automatically generated based on the Appointment Model. In this lesson, we will utilize the validation sections indicated here.

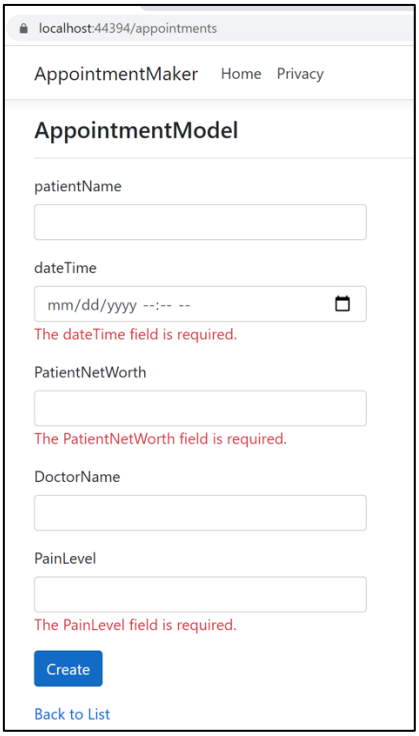
Index.cshtml | AppointmentsController.cs | AppointmentModel.cs | AppointmentMaker

```
@model AppointmentMaker.Models.AppointmentModel

<h4>AppointmentModel</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Index">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="patientName" class="control-label"></label>
                <input asp-for="patientName" class="form-control" />
                <span asp-validation-for="patientName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="dateTime" class="control-label"></label>
                <input asp-for="dateTime" class="form-control" />
                <span asp-validation-for="dateTime" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="PatientNetWorth" class="control-label"></label>
                <input asp-for="PatientNetWorth" class="form-control" />
                <span asp-validation-for="PatientNetWorth" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="DoctorName" class="control-label"></label>
                <input asp-for="DoctorName" class="form-control" />
                <span asp-validation-for="DoctorName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="PainLevel" class="control-label"></label>
                <input asp-for="PainLevel" class="form-control" />
                <span asp-validation-for="PainLevel" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Validation error messages will be placed here when a rule is violated.

Validation error messages will be placed here when a rule is violated.

JavaScript validation code is in this file inside the **views** > **shared** folder.



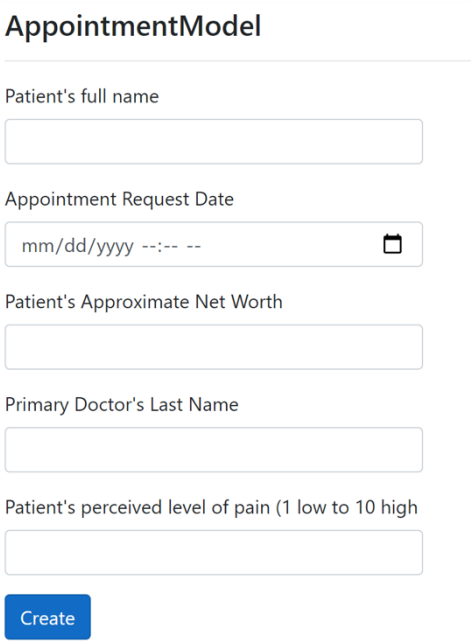
3. Run the application and navigate to localhost:44394/appointments.
4. Attempt to submit an empty form. You should see validation errors for some of the values.

Add Some Data Validation in the Model

1. Add some tags to modify the display value of each property in the **AppointmentModel**. The display name is displayed on the labels on the input form.

```
public class AppointmentModel
{
    [DisplayName("Patient's Full Name")]
    public string patientName { get; set; }
    [DisplayName("Appointment Request Date")]
    public DateTime dateTime { get; set; }
    [DisplayName("Patient's approximate net worth")]
    public decimal PatientNetWorth { get; set; }
    [DisplayName("Primary Doctor's Last Name")]
    public string DoctorName { get; set; }
    [DisplayName("Patient's perceived level of pain (1 low to 10 high)")]
    public int PainLevel { get; set; }
```

2. Run the program to see your changes.

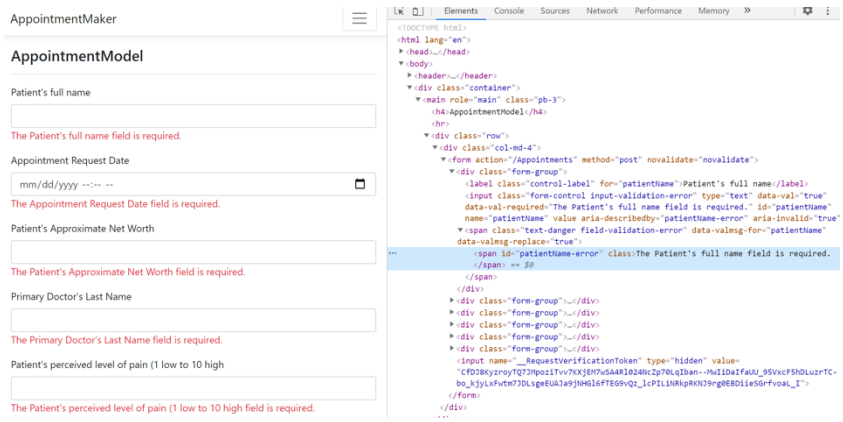


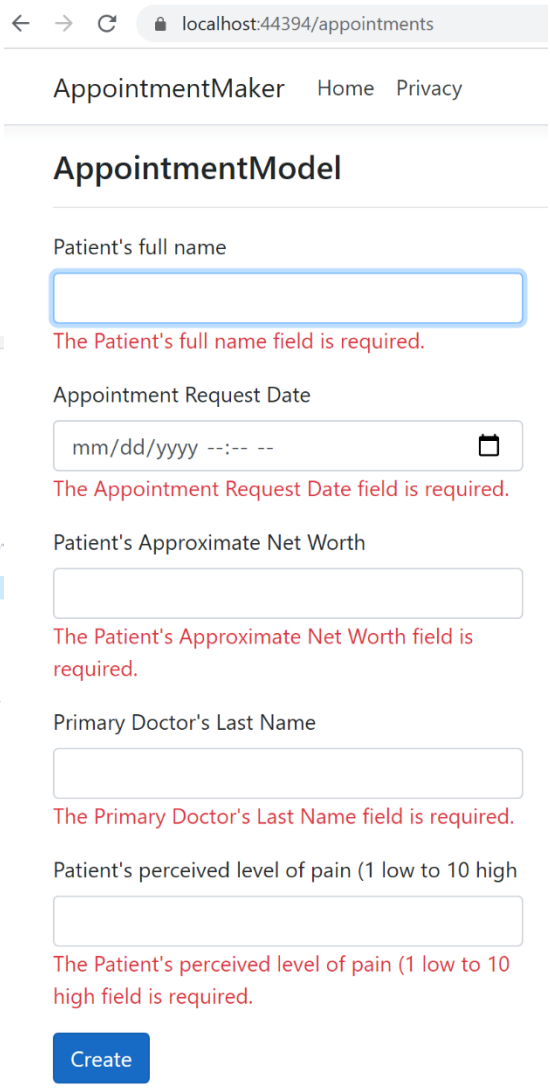
3. Add the [**Required**] tag to each property in the AppointmentModel.

```
public class AppointmentModel
{
    [Required]
    [DisplayName("Patient's Full Name")]
    public string patientName { get; set; }

    [Required]
    [DisplayName("Appointment Request Date")]
    public DateTime dateTime { get; set; }

    [Required]
    [DisplayName("Patient's approximate net worth")]
    public decimal PatientNetWorth { get; set; }

    [Required]
    [DisplayName("Primary Doctor's Last Name")]
    public string DoctorName { get; set; }

    [Required]
    [DisplayName("Patient's perceived level of pain (1 low to 10 high)")]
    public int PainLevel { get; set; }
```

Run the program and notice that you cannot submit the form unless all of the filed are filled out.

You should inspect the HTML code in the web browser (right-click the page and choose Inspect). You will see the various tags are included in the design of the form to show error messages when needed.





4. Add some additional Data validation rules to the model.

The patient's name should be between 4 and 20 characters long.

The Appointment Date should be DateTime data format.

The Doctor's name should not be required.

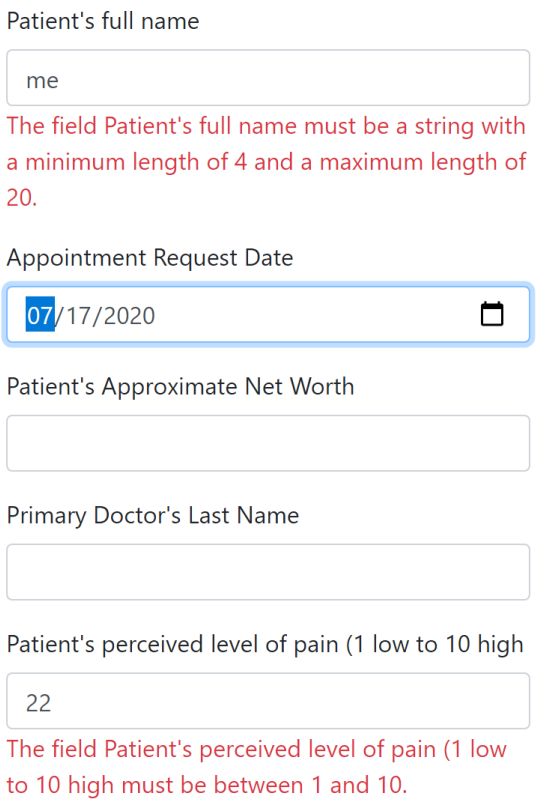
The PatientNetWorth should be currency data type.

The painLevel should be between 1 and 10.

```
10  public class AppointmentModel
11  {
12      [Required]
13      [DisplayName("Patient's Full Name")]
14      [StringLength(20, MinimumLength = 4)]
        1 reference
15      public string patientName { get; set; }
16
17      [Required]
18      [DataType(DataType.Date)]
19      [DisplayName("Appointment Request Date")]
        1 reference
20      public DateTime dateTime { get; set; }
21
22      [DisplayName("Patient's approximate net worth")]
23      [DataType(DataType.Currency)]
        1 reference
24      public decimal PatientNetWorth { get; set; }
25
26      [DisplayName("Primary Doctor's Last Name")]
        1 reference
27      public string DoctorName { get; set; }
28      [Required]
29      [Range(1, 10)]
30      [DisplayName("Patient's perceived level of pain (1 low to 10 high)")]
        1 reference
31      public int PainLevel { get; set; }
```

5. Run the program again and test the validation rules.

6. Take a screenshot of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.



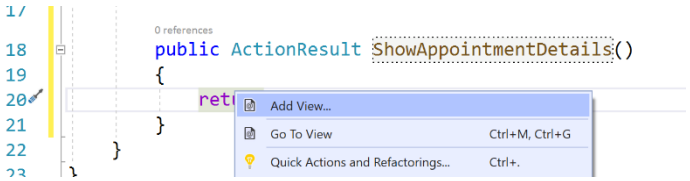
Add a response page to the appointment

Next, we will create a new view to show the results of a new appointment.

1. In the controller, add a new method called **ShowAppointmentDetails**.

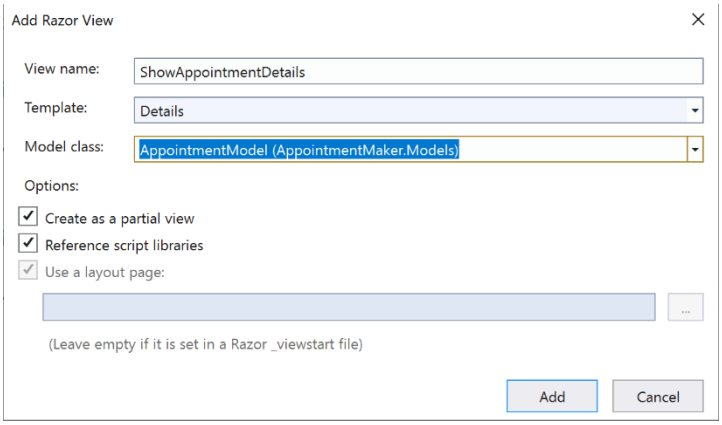
```
7   namespace AppointmentMaker.Controllers
8   {
        0 references
9       public class AppointmentsController : Controller
10      {
            0 references
11          public IActionResult Index()
12          {
13              return View();
14          }
15
            0 references
16          public IActionResult ShowAppointmentDetails()
17          {
18              return View();
19          }
20      }
21  }
```

2. Right-click in the new method and choose **Add View**.



3. Choose the following:

- View Name: **ShowAppointmentDetails**
- Template: **Details**
- Model Class: **AppointmentModel**



You should see a new View in the Views > Appointments folder.

4. In the controller, add a new **AppointmentModel** parameter to both the method and the View statements as shown here.

5. Change the form's **action** property in the Index.cshtml page to point to the new controller method.

```
@model AppointmentMaker.Models.AppointmentModel

<h4>AppointmentModel</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="ShowAppointmentDetails">
            <div asp-validation-summary="ModelOnly" class="t
            <div class="form-group">
                <label asp-for="patientName" class="control-
                <input asp-for="patientName" class="form-con
                <span asp-validation-for="patientName" class
            </div>
```

6. Try to run the program. You should see an error. The program demands a **parameterless** constructor in the model.

An unhandled exception occurred while processing the request.

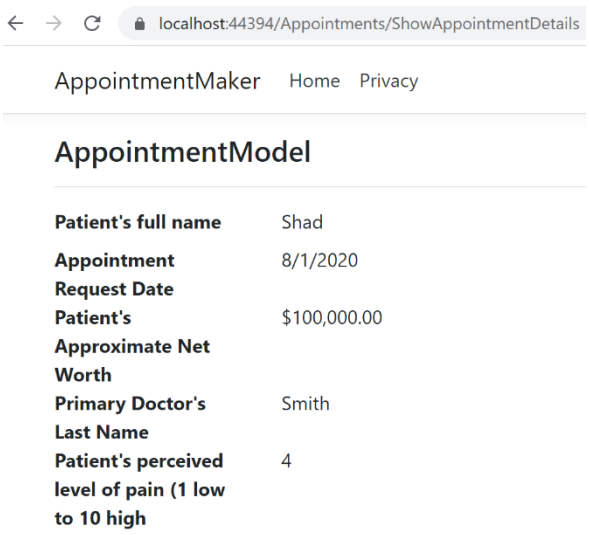
InvalidOperationException: Could not create an instance of type 'AppointmentMaker.Models.AppointmentModel'. Model bound complex types must not be abstract or value types and must have a parameterless constructor. Alternatively, give the 'appointment' parameter a non-null default value.

Microsoft.AspNetCore.Mvc.ModelBinding.Binders.ComplexTypeModelBinder.CreateModel(ModelBindingContext bindingContext)

7. Add a new constructor to the model that has no parameters.

```
0 references
public AppointmentModel(string patientName, DateTime dateTime, decimal
  patientNetWorth, CreditCardAttribute creditCard, string doctorName,
  int painLevel)
{
    this.patientName = patientName;
    this.dateTime = dateTime;
    PatientNetWorth = patientNetWorth;
    DoctorName = doctorName;
    PainLevel = painLevel;
}

0 references
public AppointmentModel()
{
}
}
}
```

You should be able to run the application and receive a response when a new appointment is created.



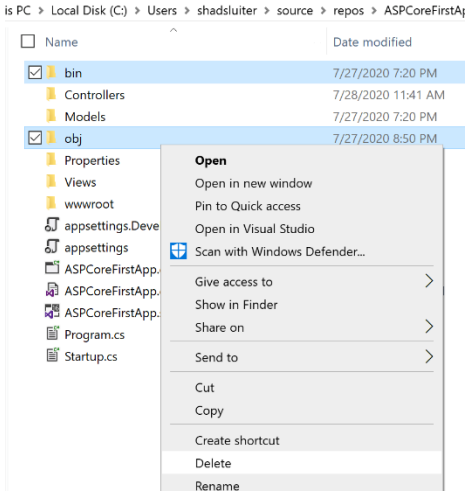
Programming Challenge

Modify the data entry form to include the following rules:

1. Add some **address fields** to the CustomerModel: **Street, City, ZIP code, email, and phone**.
2. Update (or recreate) the **ShowAppointmentDetails** view to include Street, City, ZIP code, email, and phone.
3. Validate that each of these items is in the **correct format**.
4. Add another rule: "**Doctors refuse to see patients unless their net worth is more than $90,000**"
5. Add another rule: "**Doctors refuse to see patients unless their pain level is above a 5**".
6. Take a screenshot of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.

Deliverables:

7. Submit a Microsoft Word document with screenshots of the application being run. Show each screen of the output and put a caption under each picture explaining what is being demonstrated.
8. In the same document, in one paragraph, write a summary of the key concepts that were demonstrated in this lesson.
9. Submit a ZIP file of the project file. In order to save space, you can delete the bin and the obj folders of the project. These folders contain the compiled version of the application and are automatically regenerated each time the build or run commands are executed.



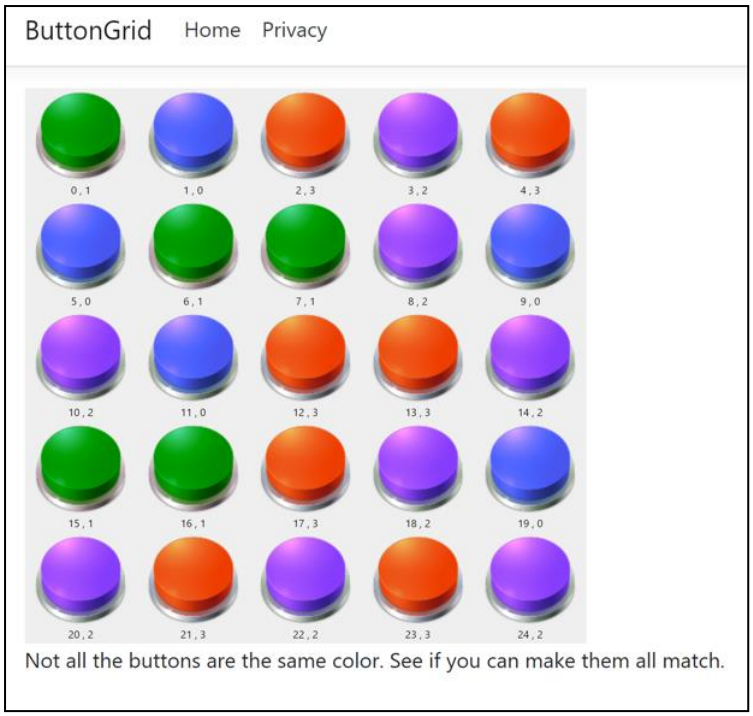
Part 4 Button Grid – Manage Dynamic Objects on a View

Goal and Directions:

In this activity, you use .NET MVC Views to:

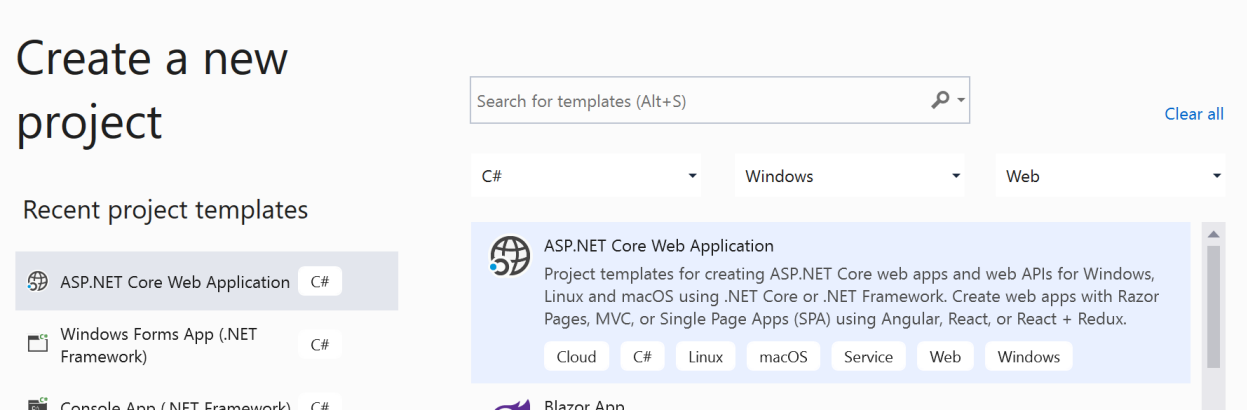
- Display a list of images on a webpage.
- Create a data structure to represent a game board.
- Create some methods to modify the game pieces.
- Challenge – program a winning game condition.

In this example, we will create a page that displays buttons. Each time you click a button it will turn on or off. This example will prepare you for building a board game in ASP.NET.

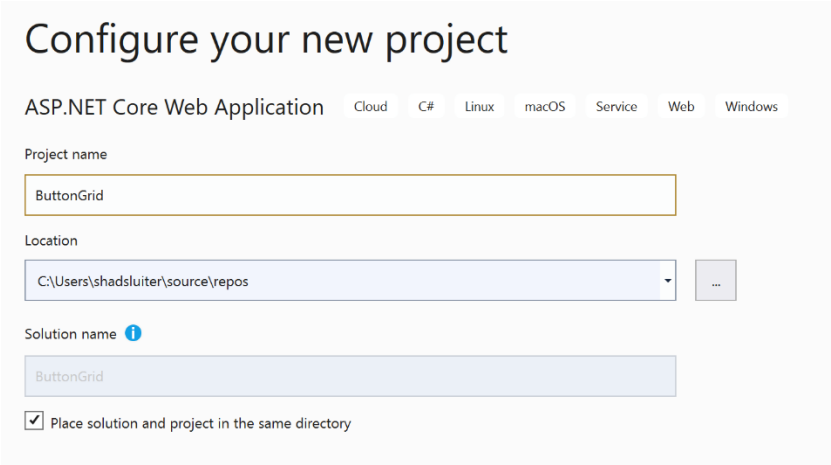


Instructions

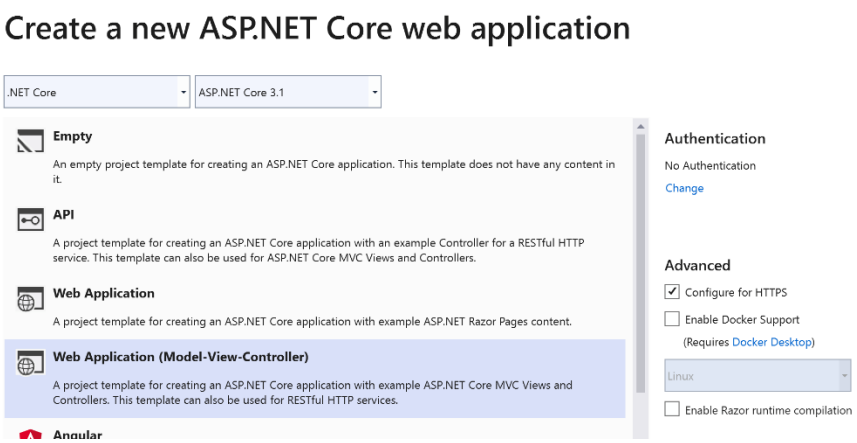
1. Create a new solution in ASP.NET Core Web Application.



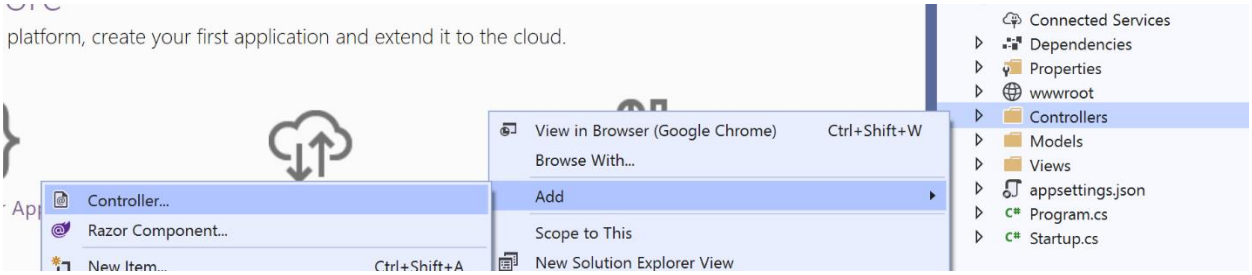
2. I will name mine ButtonGrid.



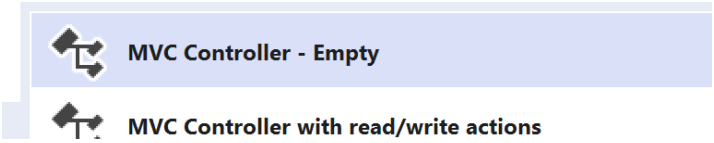
3. Set the options to Web Application (Model-View-Controller).



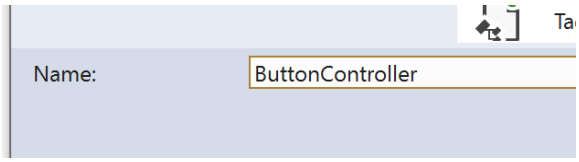
4. Create a Button Controller
    a. Right-click on the Controllers Folder.
    b. Select the Add > Controller… menu items.



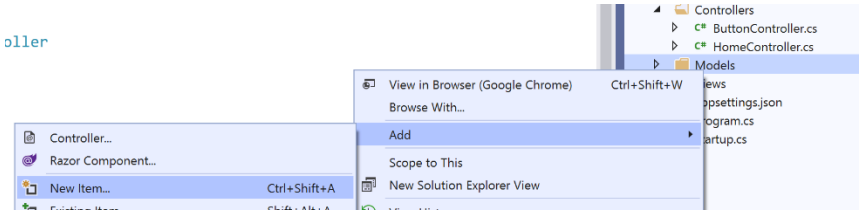
1. Then select the "MVC Controller – Empty." Click the Add button.



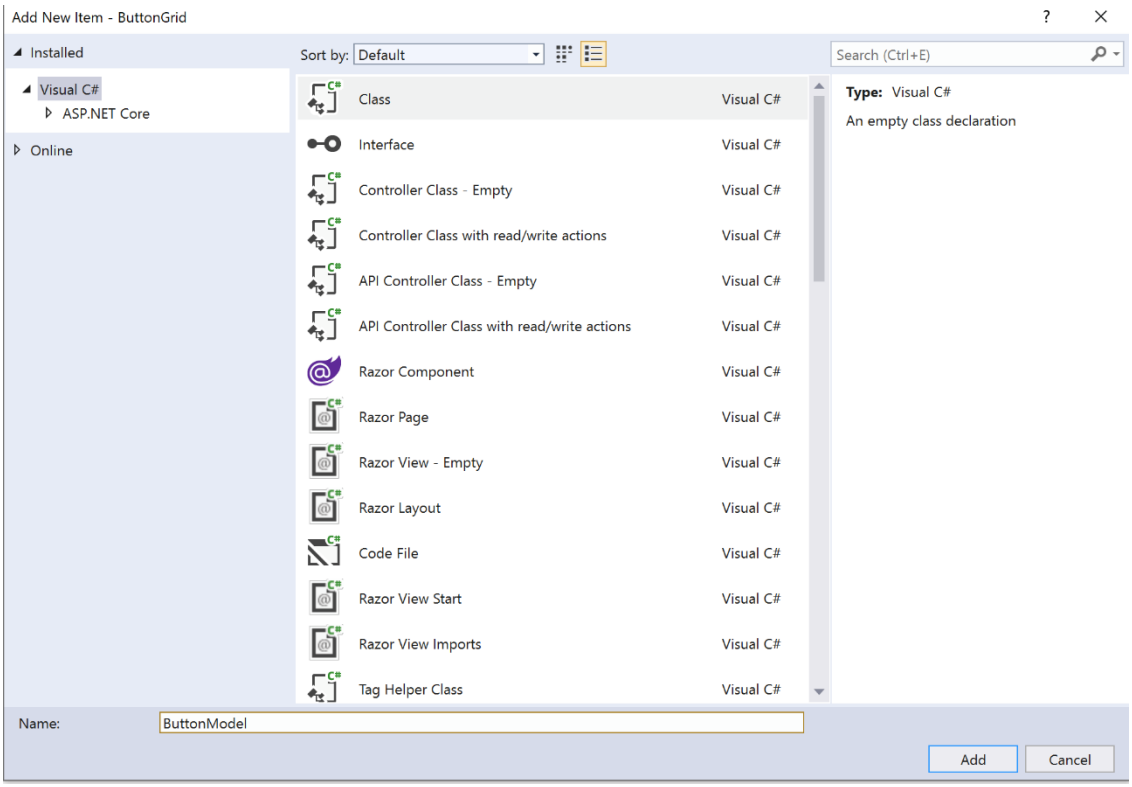
2. Name your Controller 'Button' and click the Add button. Inspect the Button Controller (as ButtonController.cs class) with in the Controllers folder.



5. Create a Button Model class.
    1. Right-click on the Models Folder.
    2. Create a Model C# class by selecting Add New Item menu.



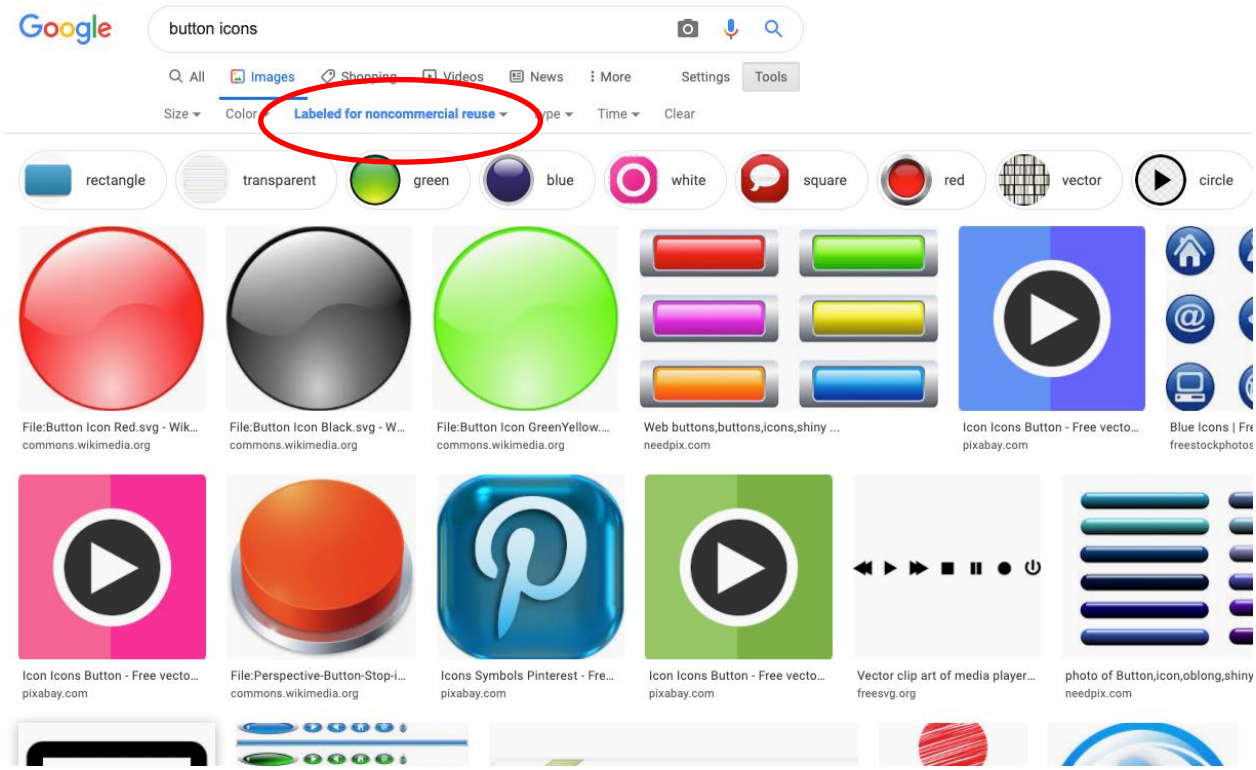
    3. Choose **Class**. Name the new file **ButtonModel**. Click **Add**.



4. Add 3 properties named Id and ButtonState in the ButtonModel class with both setter and getter methods. Also add two constructors that will initialize the state property.

```
5
6   namespace ButtonGrid.Models
7   {
        2 references
8       public class ButtonModel
9       {
            1 reference
10          public int Id { get; set; }
            1 reference
11          public int ButtonState { get; set; }
12
            0 references
13          public ButtonModel()
14          {
15
16          }
            0 references
17          public ButtonModel(int id, int buttonState)
18          {
19              Id = id;
20              ButtonState = buttonState;
21          }
22      }
23  }
```

5. Download several button icons. The picture below shows how to search for free images that are free for noncommercial use.



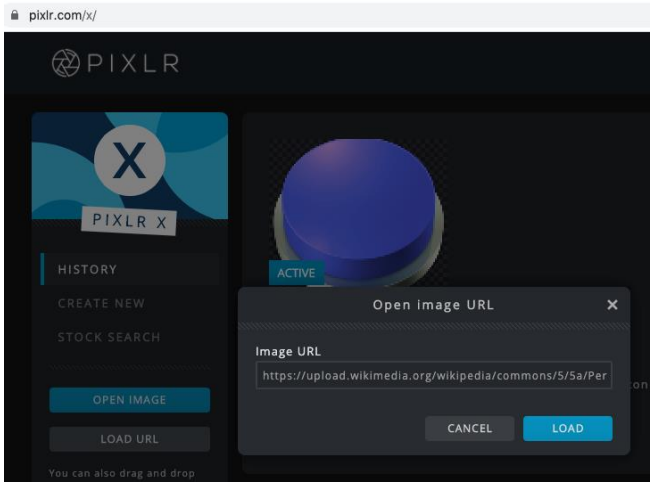
I chose this red button found at Wikipedia.

6. Use an image editing program to resize the images to a square size between 50x50 or 100x100 pixels.

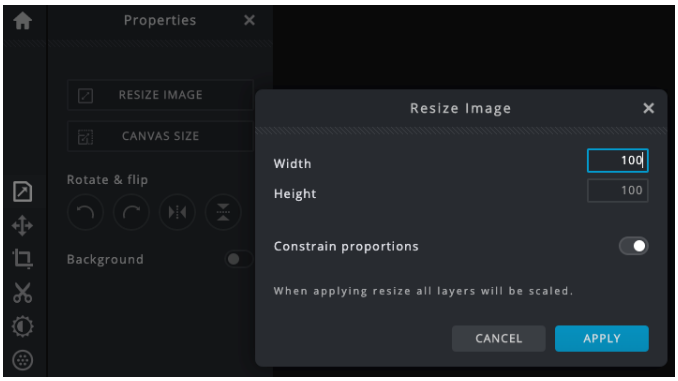
I am using pixlr to resize and recolor the image. You can copy the URL of a photo and paste it directly into the online picture editor.

You can also use Photoshop or Microsoft Paint to perform the tasks shown here.

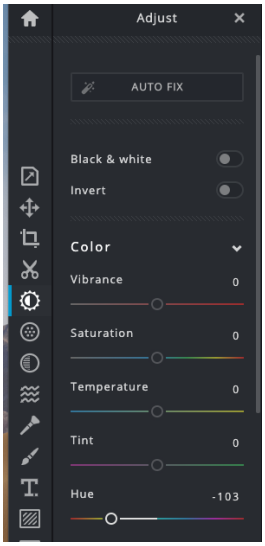
a. Choose open URL in Pixlr to open the image directly from the Wikipedia location.



b. Select Properties and Resize the Image

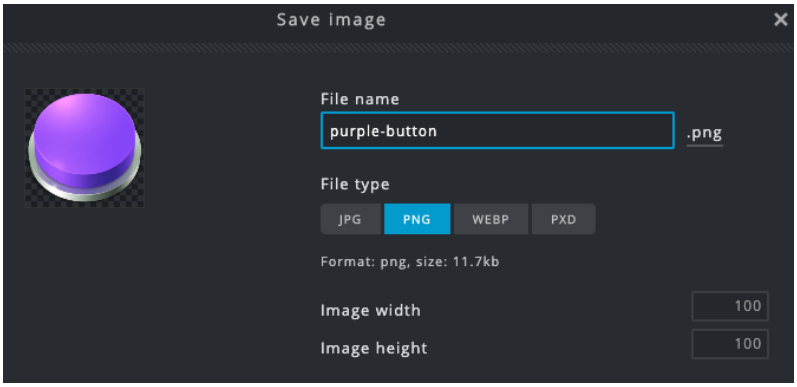


c. Choose Adjust > Hue to modify the color of the image.

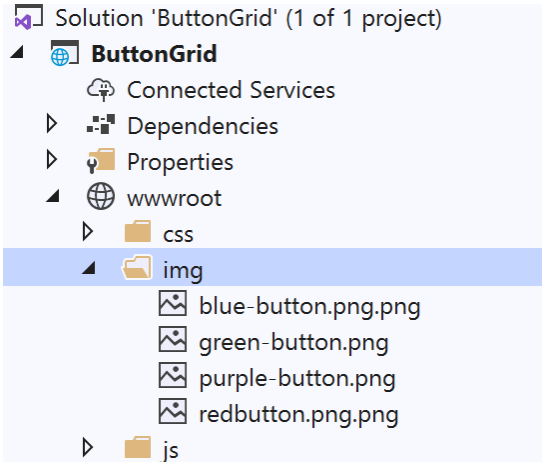


d. Choose Save to download the image and give it a name.

e. Repeat several times with different colors.

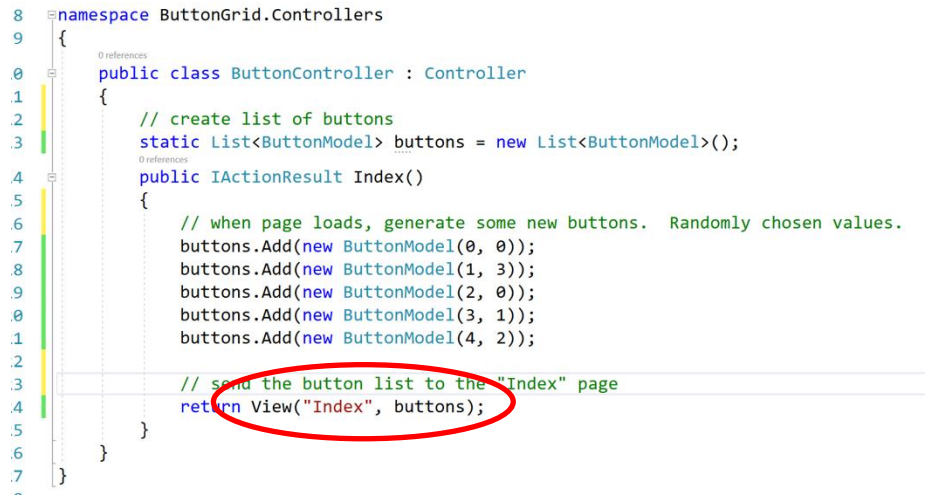


f. Create a folder named img inside the wwwroot folder in the solution explorer. Copy and paste the new images into the img folder.



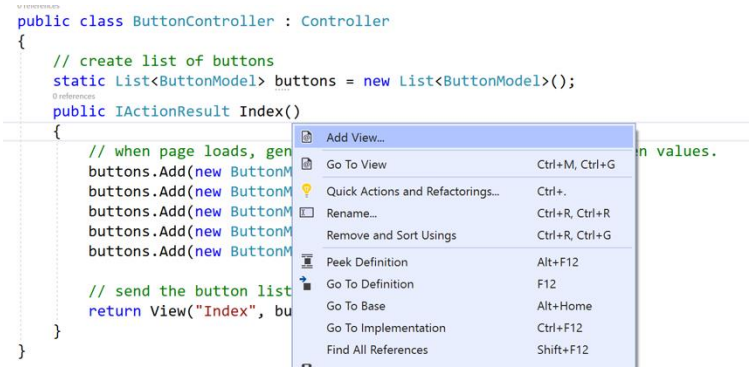
How to Create a List of Type ButtonModel in the Controller

1. Create a static class scoped member variable named buttons as a List of type ButtonModel. Creating the list using the static property means that the list values will remain the same each time the controller is invoked.

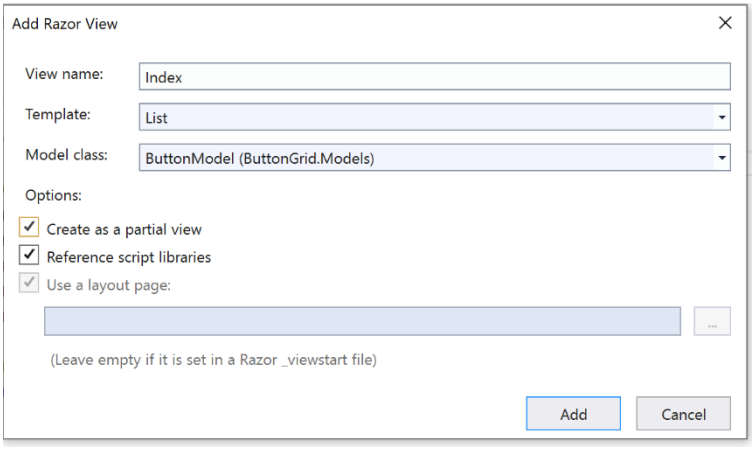


2. In the Index() of the Button Controller, add several ButtonModels to the buttons list. Pass this list as model data to the Button View.

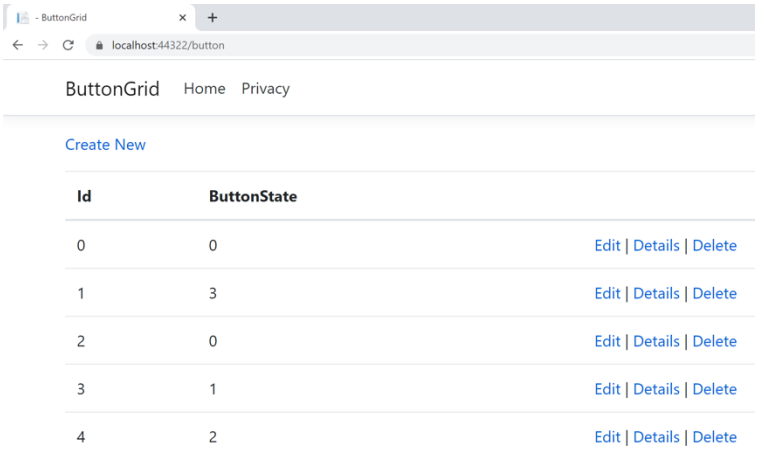
3. Create a new View for the Index method. Right-click inside the method, and choose Add View.



4. Choose the List template. View name Index. Model class ButtonModel. Partial view.

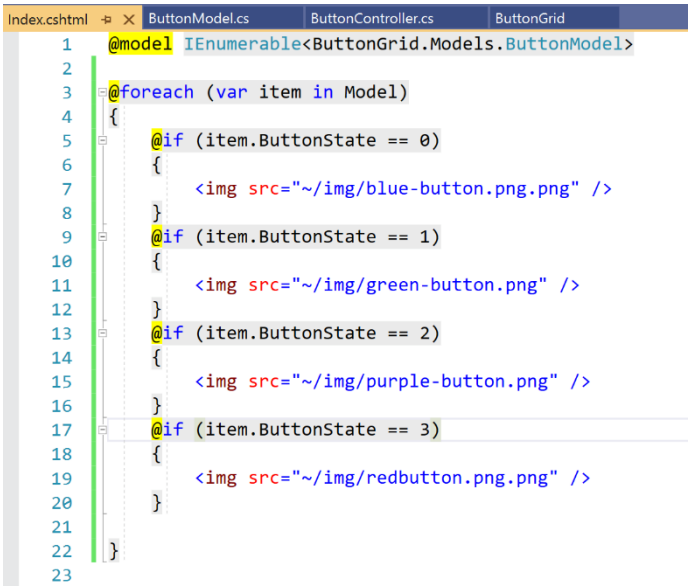


5. Test the app. The resulting button index page shows the data we generated but does not yet show the images.



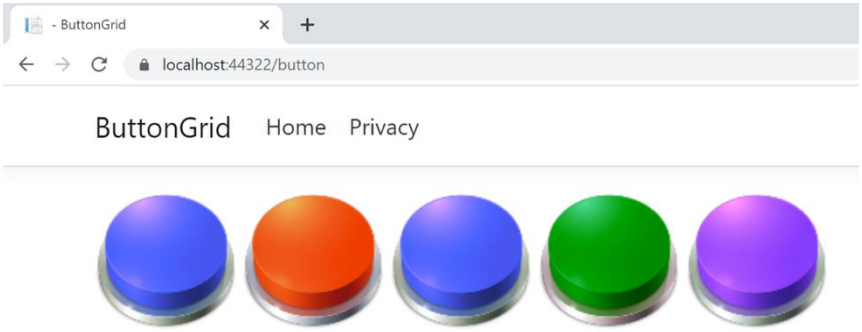
6. Take a screenshot of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.

7. Delete most of the content of the View and replace it with a for loop that displays an image for each button based on its ButtonState property.



The resulting page should show some images.

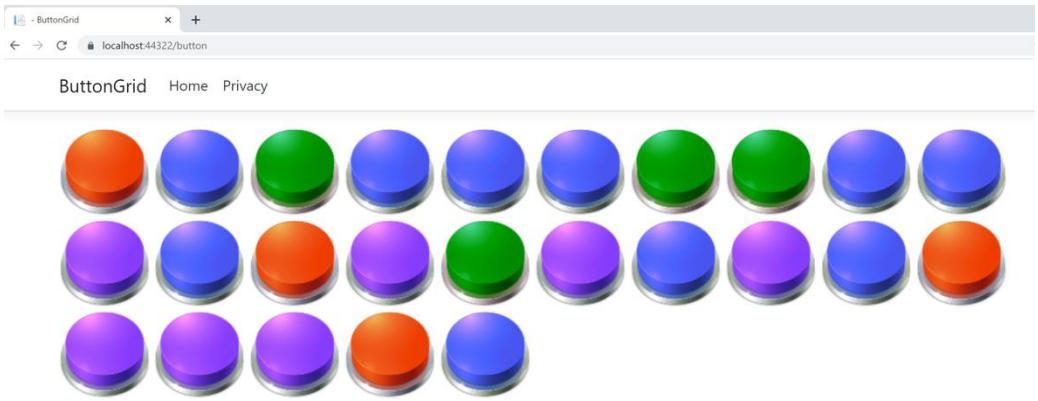
Take a screenshot of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.



8. Use a loop to add 25 buttons to the list in the controller. Notice that I am demonstrating the use of a constant value called GRID_SIZE.

```
8   namespace ButtonGrid.Controllers
9   {
        0 references
10      public class ButtonController : Controller
11      {
12          // create list of buttons
13          static List<ButtonModel> buttons = new List<ButtonModel>();
14          Random random = new Random();
15          const int GRID_SIZE = 25;
16
            0 references
17          public IActionResult Index()
18          {
19              // empty the liste when the page loads.
20              buttons = new List<ButtonModel>();
21
22              // Generate some new buttons.  Randomly chosen color values.
23              for (int i = 0; i < GRID_SIZE; i++)
24              {
25                  buttons.Add(new ButtonModel(i, random.Next(4)));
26              }
27
28              // send the button list to the "Index" page
29              return View("Index", buttons);
30          }
31      }
32  }
33
```

9. The resulting page should show 25 randomly chosen button colors. Take a screenshot of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.



10. Update the Index View to show all of the buttons using a loop. Add some CSS to shrink the button size. Start a new line every 5 buttons using the mod operator. Refactor the code to shorten the code using an array of image names.

Index.cshtml* | ButtonModel.cs | ButtonController.cs | ButtonGrid

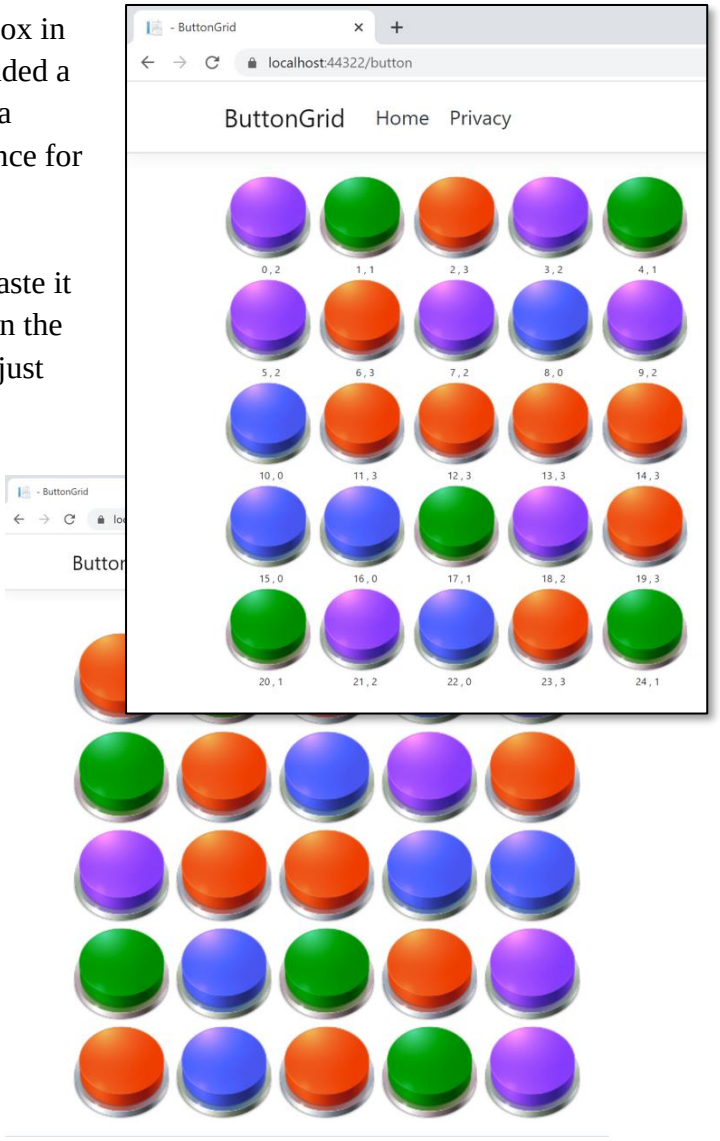
```
1   @model IEnumerable<ButtonGrid.Models.ButtonModel>
2   <style>
3       .game-button {
4           width: 75px;
5       }
6   </style>
7
8
9   @{
10      // store image names in an array for more efficient code.
11      string[] imageNames = { "blue-button.png", "green-button.png", "purple-button.png",
          "redbutton.png" };
12  }
13
14  @for (int i = 0; i < Model.Count(); i++)
15  {
16      // start a new line every five elements.
17      if (@i % 5 == 0)
18      {
19          <br />
20      }
21      // show the proper image according to the buttonState property
22      <img class="game-button" src="~/img/@imageNames[Model.ElementAt(i).ButtonState]" />
23  }
24
```

11. The resulting page should show a 5x5 grid of randomly colored buttons. Take a screenshot of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.

---

1. Apply some CSS modifications using Flexbox in order to insert labels under each button. I added a new tag surrounding the buttons and a tag under each button. A good reference for how to use flexbox is found here.

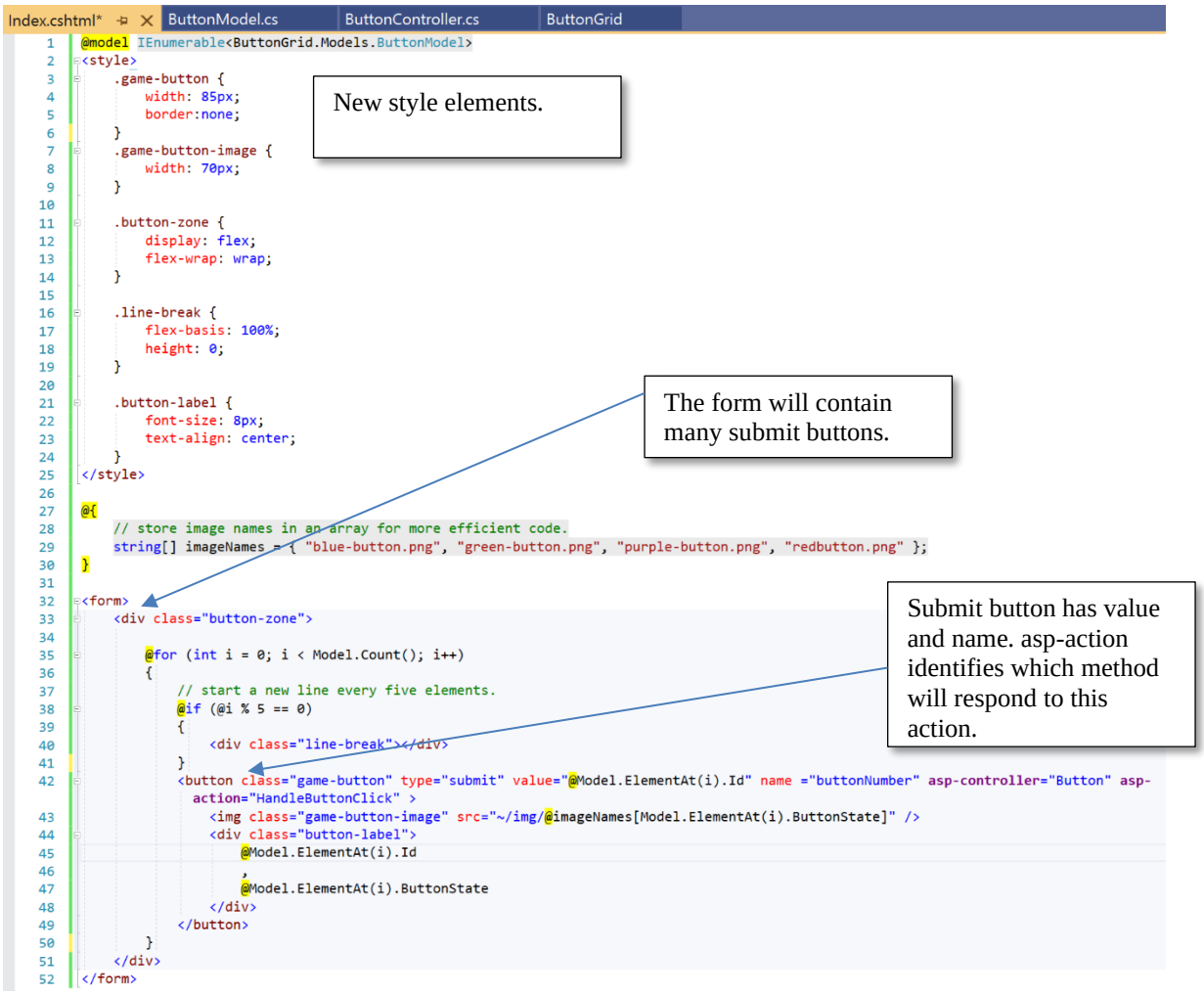
2. Take a screenshot of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.



Index.cshtml | ButtonModel.cs | ButtonController.cs | ButtonGrid

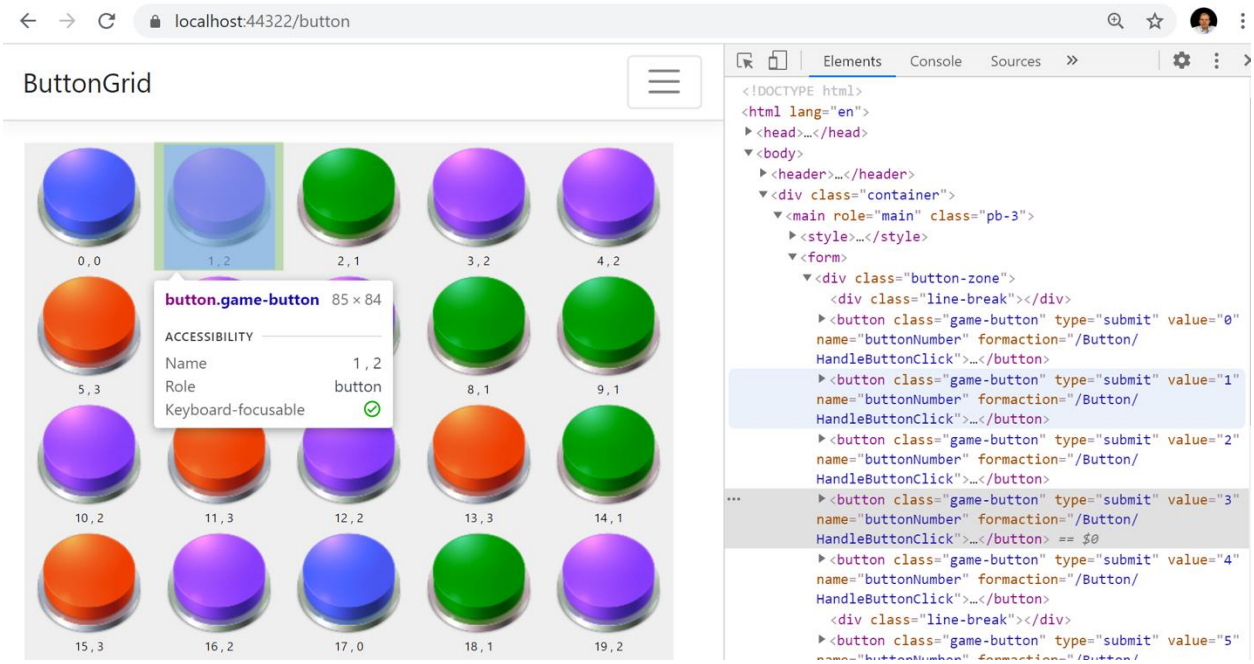
```
@model IEnumerable<ButtonGrid.Models.ButtonModel>
<style>
    .game-button {
        width: 75px;
    }
    .game-button-image{
        width:70px;
    }
    .button-zone{
        display:flex;
        flex-wrap:wrap;
    }
    .line-break{
        flex-basis:100%;
        height:0;
    }
    .button-label {
        font-size: 8px;
        text-align:center;
    }
</style>

@{
    // store image names in an array for more efficient code.
    string[] imageNames = { "blue-button.png", "green-button.png", "purple-button.png", "redbutton.png" };
}

<div class="button-zone">

@for (int i = 0; i < Model.Count(); i++)
{
    // start a new line every five elements.
    @if (@i % 5 == 0)
    {
        <div class="line-break"></div>
    }

    <div class="game-button">
        <img class="game-button-image" src="~/img/@imageNames[Model.ElementAt(i).ButtonState]" />
        <div class="button-label">
            @Model.ElementAt(i).Id
            ,
            @Model.ElementAt(i).ButtonState
        </div>
    </div>
}

</div>
```

12. Create a form and surround each img with a submit button with the asp-action property set to "HandleButtonClick." We will have to create the HandleButtonClick event handler in the controller in order for this to work correctly.




```
@model IEnumerable<ButtonGrid.Models.ButtonModel>
<style>
    .game-button {
        width: 85px;
        border:none;
    }
    .game-button-image {
        width: 70px;
    }

    .button-zone {
        display: flex;
        flex-wrap: wrap;
    }

    .line-break {
        flex-basis: 100%;
        height: 0;
    }

    .button-label {
        font-size: 8px;
        text-align: center;
    }
</style>

@{
    // store image names in an array for more efficient code.
    string[] imageNames = { "blue-button.png", "green-button.png", "purple-button.png", "redbutton.png" };
}

<form>
    <div class="button-zone">

        @for (int i = 0; i < Model.Count(); i++)
        {
            // start a new line every five elements.
            @if (@i % 5 == 0)
            {
                <div class="line-break"></div>
            }
            <button class="game-button" type="submit" value="@Model.ElementAt(i).Id" name ="buttonNumber" asp-controller="Button" asp-
              action="HandleButtonClick" >
                <img class="game-button-image" src="~/img/@imageNames[Model.ElementAt(i).ButtonState]" />
                <div class="button-label">
                    @Model.ElementAt(i).Id
                    ,
                    @Model.ElementAt(i).ButtonState
                </div>
            </button>
        }
    </div>
</form>
```

13. Run the program and inspect the HTML code that is produced. You should see form has an action equal to "/Button/HandleButtonClick." You should also notice that there are many submit buttons in the form, each with its own value from 0 to 24. Take a screenshot of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.



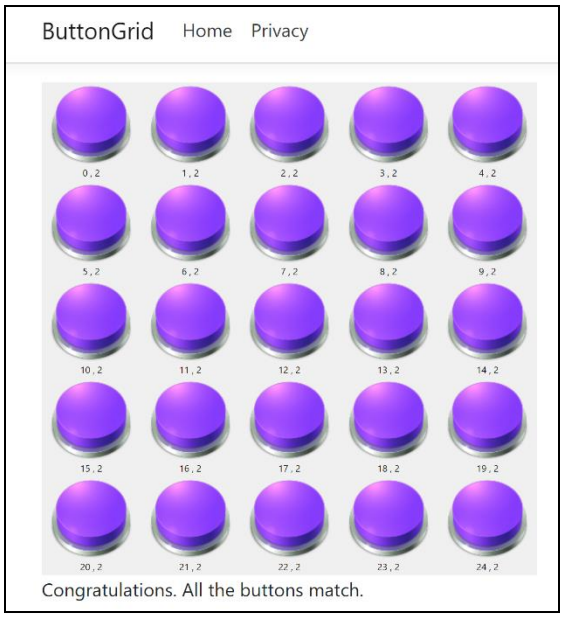
14. Update the Button Controller:
    1. Add a new public method **HandleButtonClick** that returns an IActionResult and takes a single method argument named **buttonNumber** of type string.
    2. In the **HandleButtonClick** method implementation gets the integer value of the button that was clicked. Use this number to increment the value of the corresponding button. The mod 4 operator keeps the value within the acceptable range of button properties. The method passes the buttons list model to the Index view.

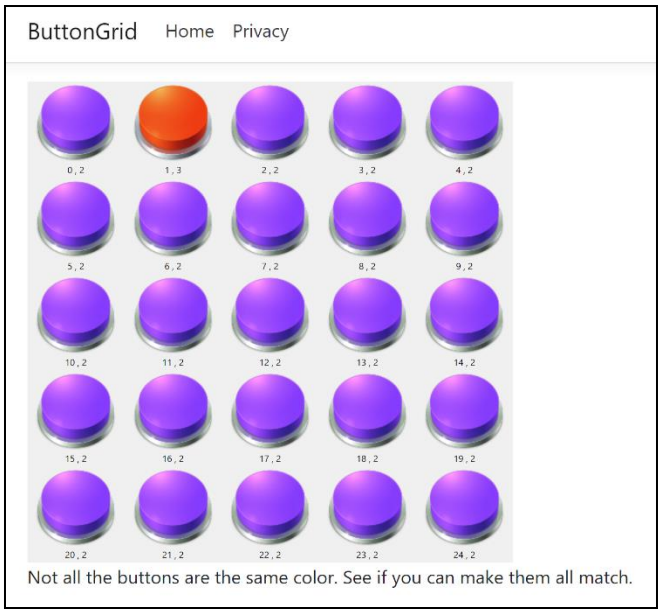
```
17          0 references
            public IActionResult Index()
18          {
19              // empty the liste when the page loads.
20              buttons = new List<ButtonModel>();
21
22              // Generate some new buttons.  Randomly chosen color values.
23              for (int i = 0; i < GRID_SIZE; i++)
24              {
25                  buttons.Add(new ButtonModel(i, random.Next(4)));
26              }
27
28              // send the button list to the "Index" page
29              return View("Index", buttons);
30          }
31
            0 references
32          public IActionResult HandleButtonClick(string buttonNumber)
33          {
34              // convert from string to int.
35              int bN = int.Parse(buttonNumber);
36
37              // add one to the button state. If greater than 4, reset to 0.
38              buttons.ElementAt(bN).ButtonState = ( buttons.ElementAt(bN).ButtonState + 1 ) % 4;
39
40              // re-display the buttons
41              return View("Index", buttons);
42          }
43      }
44  }
```

Test the app. You should be able to increment each button state with a single mouse click. Toggle the buttons and take at least 2 screenshots showing different states of your button images. Take a screenshot of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.

Programming Challenge

Add a game element to the grid. Give the player a success message when all the buttons are the same color.





Deliverables:

1. Submit a Microsoft Word document with screenshots of the application being run at each stage of development. Show each screen of the output and put a caption under each picture explaining what is being demonstrated.
2. In the same document, in one paragraph, write a summary of the key concepts that were demonstrated in this lesson.
3. Submit a ZIP file of the project file. In order to save space, you can delete the bin and the obj folders of the project. These folders contain the compiled version of the application and are automatically regenerated each time the build or run commands are executed.