CST-350 Activity 4 Dependency Injection, Data Validation and ButtonGrid

Table of Contents

Introduction

This lesson focuses on three essential aspects of modern software development: Dependency Injection, Data Validation, and Dynamic Object Management through a GUI. Each section will build your understanding and skills in creating scalable, maintainable, and user-friendly applications.

Learning Goals

- **Understand and Implement Dependency Injection (DI)**: Learn how to decouple classes from their dependencies using DI, enhancing testability and flexibility.
- **Apply Data Validation**: Ensure data integrity by implementing validation rules in a web application form.
- **Create a Dynamic Button Grid**: Develop a GUI for a game board using ASP.NET MVC, showcasing dynamic object management.

Part 1: Dependency Injection

In the Register and Login app from a previous lesson, we used an interface and applying it to two classes. An interface provides a list of method names that apply to all implementing classes, acting as a "contract." This ensures consistency and interchangeability of classes within the application.

By using Dependency Injection (DI), we will utilize interfaces to decouple the UserController from its dependencies. Instead of the UserController instantiating the UserCollection class directly, we will use an interface to abstract these dependencies.

Part 2: Data Validation

Data validation ensures that the data entered into a form meets specific criteria, such as format and required fields. This is crucial for maintaining data integrity and providing a good user experience. In this section, you will learn how to implement data validation rules in a model class using ASP.NET MVC.

The project involves creating an appointment scheduling form with various data types (string, date, currency, and integer). Some fields will have specific validation rules, such as required fields, number ranges, and specific formats.

By the end of this section, you will be able to create forms with data validation, enhancing the reliability of your web applications.

Part 3: Button Grid - Manage Dynamic Objects on a View

GRAND CANYON UNIVERSITY

In this activity, you will create a dynamic GUI for a game board using .NET MVC. The objective is to display a grid of buttons, each of which can be toggled on or off. This will prepare you for building more complex interactive applications, such as board games.

Part 1 Dependency Injection

Interfaces and Dependency Injection

In the Register and Login app, we applied some **object-oriented programming design concepts** by creating an **interface** and applying it to two classes. Recall that an interface provides a list of method names that apply to all classes that implement the interface. An interface in object-oriented programming is sometimes called a "**contract**" that classes must fulfil.

In this case, we created two classes that manage user accounts. We used the interface to ensure that the method names for the two classes are identical, making them interchangeable within the application.

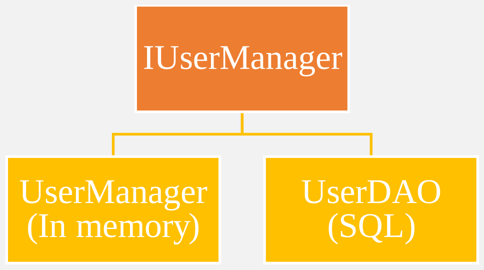


Figure 1 IUserManager is the pattern for two (or more) classes whose responsibility is to manage the list of known users on this system.

We will now see a second advantage to using the interface design pattern, **Dependency Injection**.

By instantiating the **UserCollection** class within the user controller, we have created a **dependency**. The UserController now **depends on** the UserCollection in order to exist. The UserController is **tightly coupled** to the UserCollection class, which can lead to several issues:

1. Hard to Test: When a class directly instantiates its dependencies, it becomes difficult to test the class in isolation.

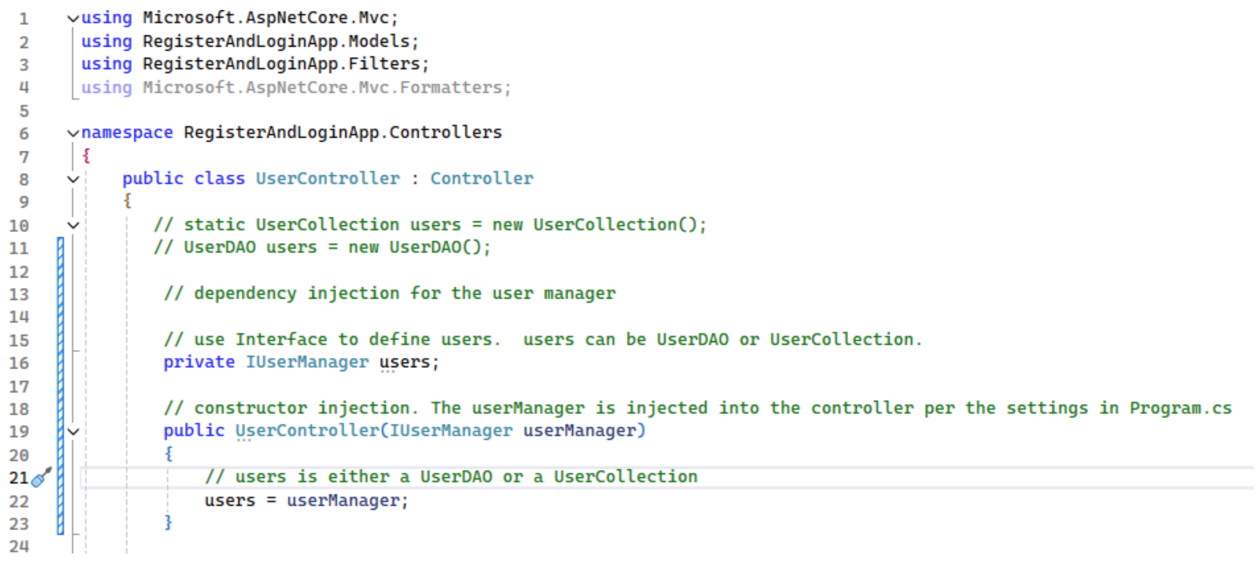
2. Reduced Flexibility: Tight coupling makes it harder to switch out the UserCollection for another implementation without modifying the UserController. If you decide to change how users are managed (e.g., switch to a database-backed UserCollection), you will need to change the UserController as well.

You now will apply Dependency Injection to the Register and Login app. The goal is to decouple the UserController from its direct dependency on the UserCollection class by using interfaces. This will involve:

1. **Removing Direct Instantiation**: Instead of directly creating instances of UserCollection and UserDAO within UserController, you'll use an interface to abstract these dependencies.
2. **Injecting Dependencies**: You'll modify the UserController to accept dependencies through its constructor.
3. **Configuring DI Container**: You'll configure the DI container in the Program.cs file to manage the lifecycle of dependencies and inject the appropriate implementations where needed.

Understanding and applying DI is a fundamental skill in modern software development, especially in frameworks like ASP.NET Core, where DI is a built-in feature that facilitates clean and scalable architecture.

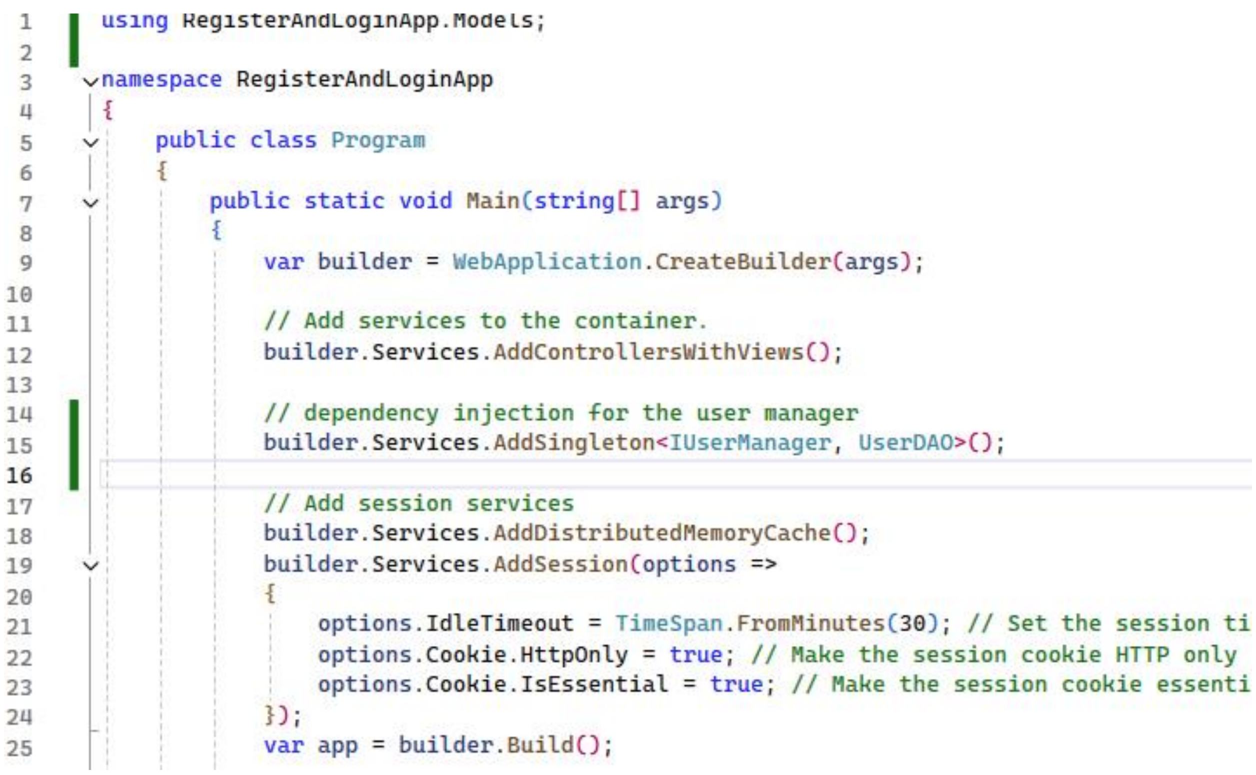
1. Open the UserController and make the following updates
   a. Comment out bother instances of UserCollection and UserDAO. We will no longer instantiate the class directly.
   b. Add a new variable **private IUserManager users**.
   c. Add a constructor with a **IUserManager userManager** parameter
   d. Assign **users** value to be the **userManager** parameter.



```
using Microsoft.AspNetCore.Mvc;
using RegisterAndLoginApp.Models;
using RegisterAndLoginApp.Filters;
using Microsoft.AspNetCore.Mvc.Formatters;

namespace RegisterAndLoginApp.Controllers
{
    public class UserController : Controller
    {
        // static UserCollection users = new UserCollection();
        // UserDAO users = new UserDAO();

         // dependency injection for the user manager

         // use Interface to define users.  users can be UserDAO or UserCollection.
         private IUserManager users;

         // constructor injection. The userManager is injected into the controller per the settings in Program.cs
         public UserController(IUserManager userManager)
         {
             // users is either a UserDAO or a UserCollection
             users = userManager;
         }
```

Figure 2 Using a constructor and dependency injection method to inject the user manager class.

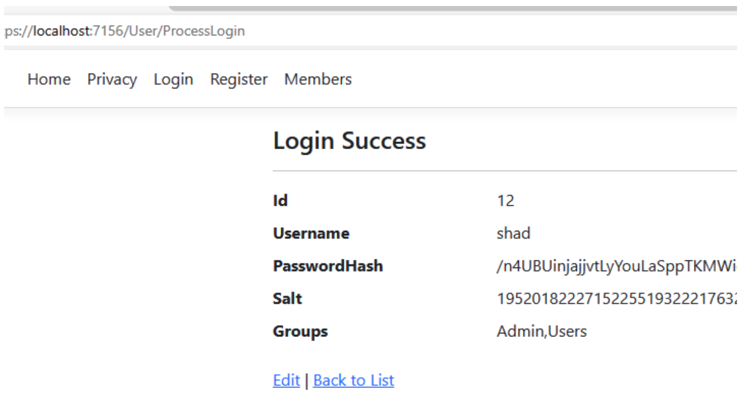
2. Open the Program.cs file to add a setting to associate IUserManager with UserDAO.



```
using RegisterAndLoginApp.Models;

namespace RegisterAndLoginApp
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var builder = WebApplication.CreateBuilder(args);

            // Add services to the container.
            builder.Services.AddControllersWithViews();

            // dependency injection for the user manager
            builder.Services.AddSingleton<IUserManager, UserDAO>();

            // Add session services
            builder.Services.AddDistributedMemoryCache();
            builder.Services.AddSession(options =>
            {
                options.IdleTimeout = TimeSpan.FromMinutes(30); // Set the session ti
                options.Cookie.HttpOnly = true; // Make the session cookie HTTP only
                options.Cookie.IsEssential = true; // Make the session cookie essenti
            });
            var app = builder.Build();
```

Figure 3 New "AddSingleton" command associates the IUserManager interface with the UserDAO class.

3. Run the program. It should function normally if the dependency injection settings are configured correctly.



ps://localhost:7156/User/ProcessLogin

Login Success

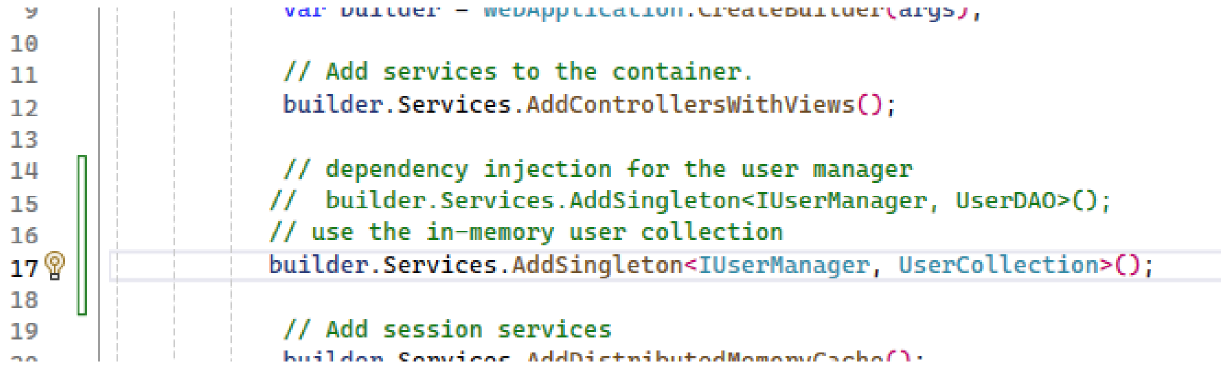
Id	12
Username	shad
PasswordHash	/n4UBUinjajjvtLyYouLaSppTKMWi
Salt	195201822271522551932221763:
Groups	Admin,Users

Edit | Back to List

Figure 4 The app functions normally using the users SQL table as its data source.

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

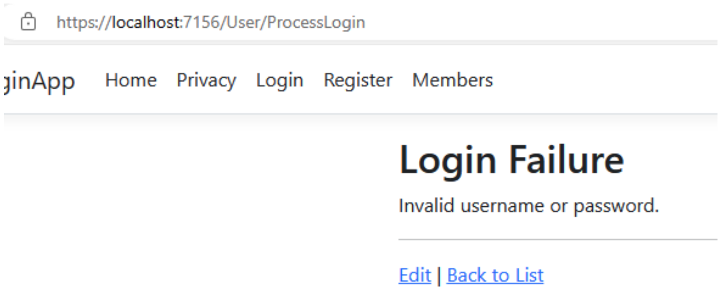
4. For demonstration purposes, change the class that is injected by modifying the Program.cs file



```
        var builder = WebApplication.CreateBuilder(args);

        // Add services to the container.
        builder.Services.AddControllersWithViews();

        // dependency injection for the user manager
       //  builder.Services.AddSingleton<IUserManager, UserDAO>();
       // use the in-memory user collection
       builder.Services.AddSingleton<IUserManager, UserCollection>();

        // Add session services
        builder.Services.AddDistributedMemoryCache();
```

Figure 5 Modified Program.cs now indicates that the UserManager class is to be injected into the controller.

5. Run the program. You should see that the users in the SQL table are no longer being used. Instead, the hard-coded data (Harry and Megan) are in use.

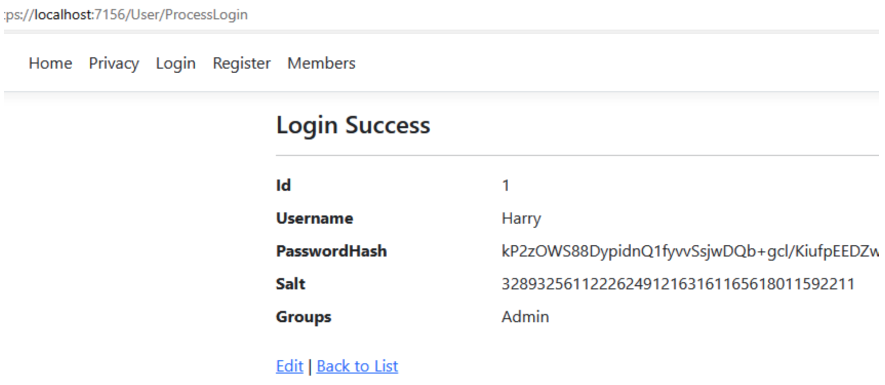


https://localhost:7156/User/ProcessLogin

Login Failure

Invalid username or password.

Figure 6 Login fails for any user stored in the SQL table.



:ps://localhost:7156/User/ProcessLogin

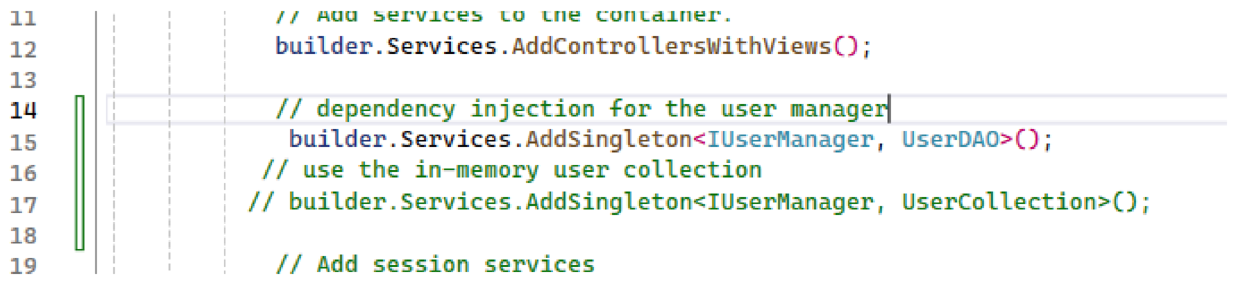
Login Success

Id	1
Username	Harry
PasswordHash	kP2zOWS88DypidnQ1fyvvSsjwDQb+gcl/KiufpEEDZw
Salt	3289325611222624912163161165618011592211
Groups	Admin

Figure 7 User "Harry" is hard-coded into the UserManager class and appears to be the data source for the app.

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

6. Modify Program.cs again to switch the User Manager back to the SQL data source.



```
        // Add services to the container.
        builder.Services.AddControllersWithViews();

        // dependency injection for the user manager
         builder.Services.AddSingleton<IUserManager, UserDAO>();
       // use the in-memory user collection
      // builder.Services.AddSingleton<IUserManager, UserCollection>();

        // Add session services
```

Figure 8 UserDAO is now the injected class.

Summary of Part 1: Dependency Injection

Dependency Injection decouples the UserController from its direct dependency on the UserCollection class by using interfaces. This is achieved by injecting the IUserManager interface into the UserController, allowing different implementations of IUserManager to be swapped without modifying the controller.

Key steps include:

1. **Commenting Out Direct Instantiations**: The direct instantiation of UserCollection and UserDAO in the UserController is commented out.

2. **Introducing an Interface Variable**: A private IUserManager variable is added to the UserController.

3. **Using a Constructor for Dependency Injection**: A constructor is added to the UserController that accepts an IUserManager parameter and assigns it to the interface variable.

4. **Configuring Dependency Injection in Program**.cs: The AddSingleton method is used to associate the IUserManager interface with the UserDAO class in the Program.cs file.

5. **Testing with Different Implementations**: The dependency injection is tested by switching between different implementations of IUserManager, demonstrating how the application behaves with different data sources without modifying the UserController.

Beyond Singletons

In ASP.NET Core, the builder.Services.AddSingleton<IUserManager, UserDAO>() method registers the UserDAO class as the implementation of the IUserManager interface with a **singleton lifetime**. There are other options available for registering services, each with different lifetimes.

1. AddSingleton: Registers a service with a singleton lifetime. This means a single instance of the service is created and shared throughout the application's lifetime. Choose AddSingleton when the service should maintain state and be shared across the entire application. It's ideal for services that are expensive to create or maintain shared state, such as configuration settings, caching, or data access layers that maintain a consistent state.

2. AddScoped:

Definition: Registers a service with a scoped lifetime. A new instance of the service is created once per client request (connection).

Use Case: Choose AddScoped for services that should be created per request but reused within the same request. It's ideal for services that deal with user-specific data or context, like data repositories or business logic handlers that need to maintain state within a single request but not across requests.

3. AddTransient:

Definition: Registers a service with a transient lifetime. A new instance of the service is created each time it is requested.

Use Case: Choose AddTransient for lightweight, stateless services that are cheap to create and do not maintain state. It's ideal for services like logging, utility classes, or services where each operation is independent.

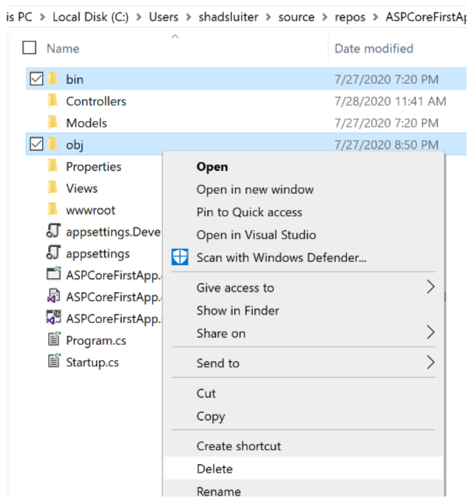
Why Choose AddSingleton for UserDAO?

1. Consistency: Since UserDAO is designed to be a shared service that should provide a consistent view of data or state across the entire application, a singleton is appropriate. This ensures that all parts of the application using IUserManager are interacting with the same instance and state.

Creating instances of services can be costly in terms of resources and time. AddSingleton creates the instance once and reuses it, reducing the overhead of creating and disposing of instances repeatedly.

Singleton services are ideal for caching data or storing configuration settings that need to be accessed globally. This reduces the need to retrieve or compute the same data multiple times.

Deliverables:



1. This activity has multiple parts. Complete all parts before submitting.
2. Create 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. Turn the Word document into a PDF.
5. 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.
6. Attach the PDF separately from the zip file. Multiple files can be uploaded with an assignment.

Part 2 Data Validation:

Goal:

In this part of the 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

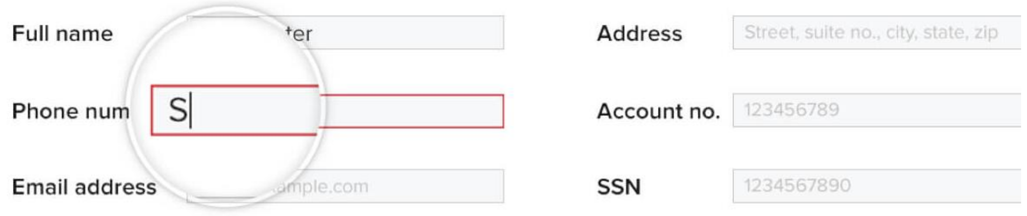


Figure 9 Data validation catches the "S" as an invalid character for a phone number.

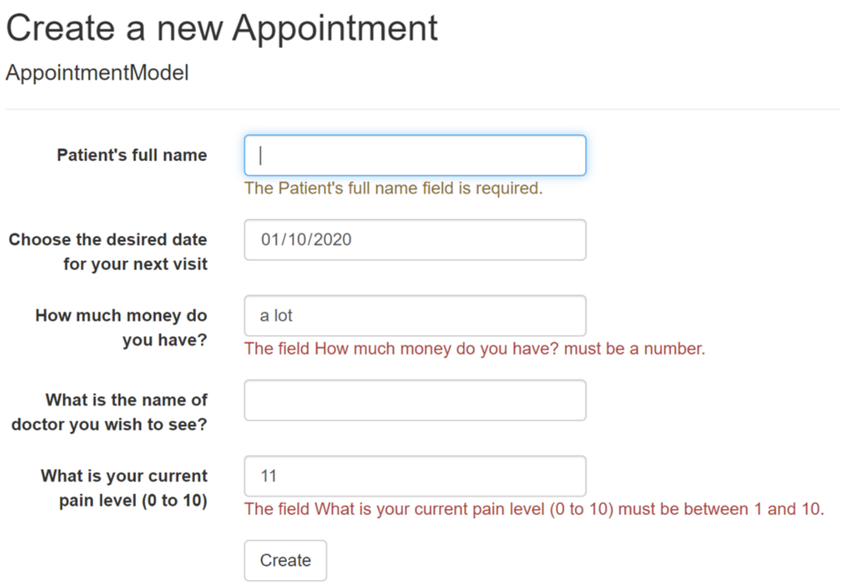
The validation support provided by ASP.NET MVC allows you to specify validation rules in the model class and the rules are enforced *everywhere* in the application.

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.


Create a new Appointment

AppointmentModel

Patient's full name	
	The Patient's full name field is required.
Choose the desired date for your next visit	01/10/2020
How much money do you have?	a lot
	The field How much money do you have? must be a number.
What is the name of doctor you wish to see?	
What is your current pain level (0 to 10)	11
	The field What is your current pain level (0 to 10) must be between 1 and 10.

Figure 10 Preview of the Data Validation example we will create in this lesson.

Create a New Project:

1. Start a new MVC .NET project.
2. I will name the project AppointmentScheduler.
3. 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**.

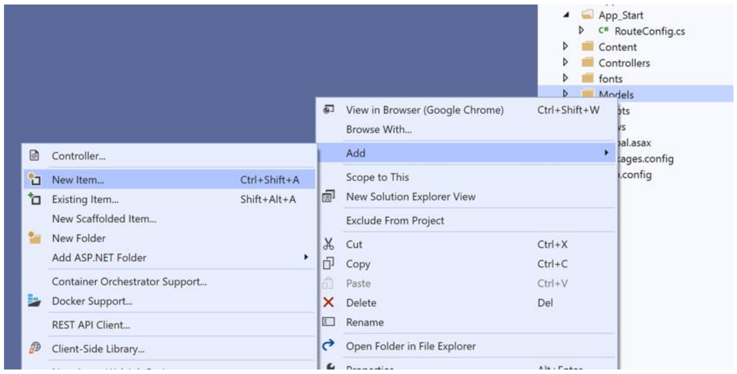
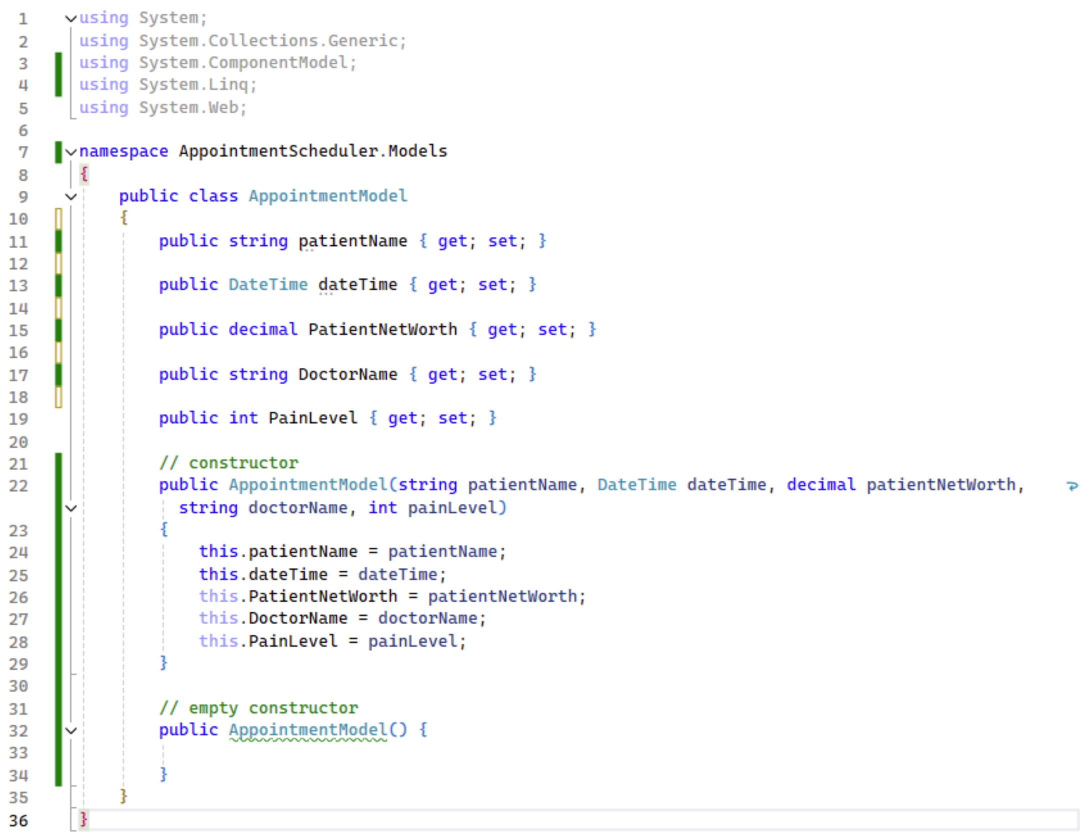


Figure 11 Adding a new Item to the project.

2. Choose **Class**.
3. Name the class **AppointmentModel**.
4. Click the Add button.Create the following properties
    - patientName (string)
    - dateTime (date)
    - PatientNetWorth (decimal)
    - DoctorName (string)
    - PainLevel (int)
5. Add two constructors one parameterized and the other a default (empty) constructor.



```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Linq;
using System.Web;

namespace AppointmentScheduler.Models
{
    public class AppointmentModel
    {
        public string patientName { get; set; }

        public DateTime dateTime { get; set; }

        public decimal PatientNetWorth { get; set; }

        public string DoctorName { get; set; }

        public int PainLevel { get; set; }

        // constructor
        public AppointmentModel(string patientName, DateTime dateTime, decimal patientNetWorth,
          string doctorName, int painLevel)
        {
            this.patientName = patientName;
            this.dateTime = dateTime;
            this.PatientNetWorth = patientNetWorth;
            this.DoctorName = doctorName;
            this.PainLevel = painLevel;
        }

        // empty constructor
        public AppointmentModel() {

        }
    }
}
```

Figure 12 Appointment class has a variety of data types for its properties.

Create a Controller:

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

GRAND CANYON UNIVERSITY™

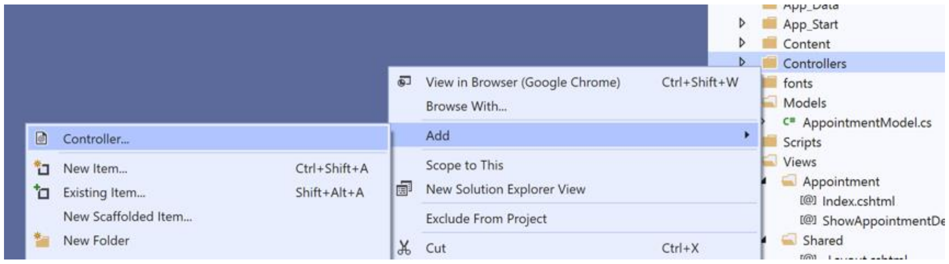
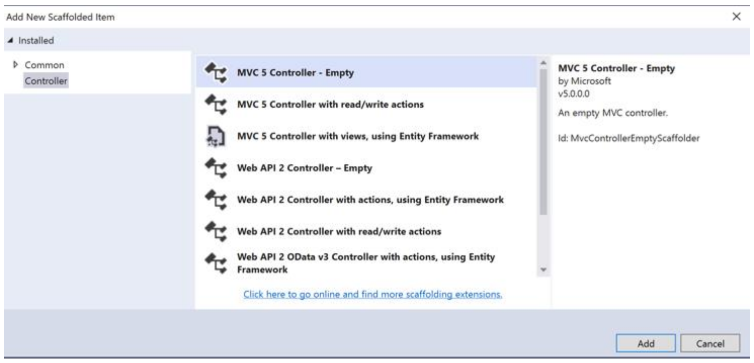


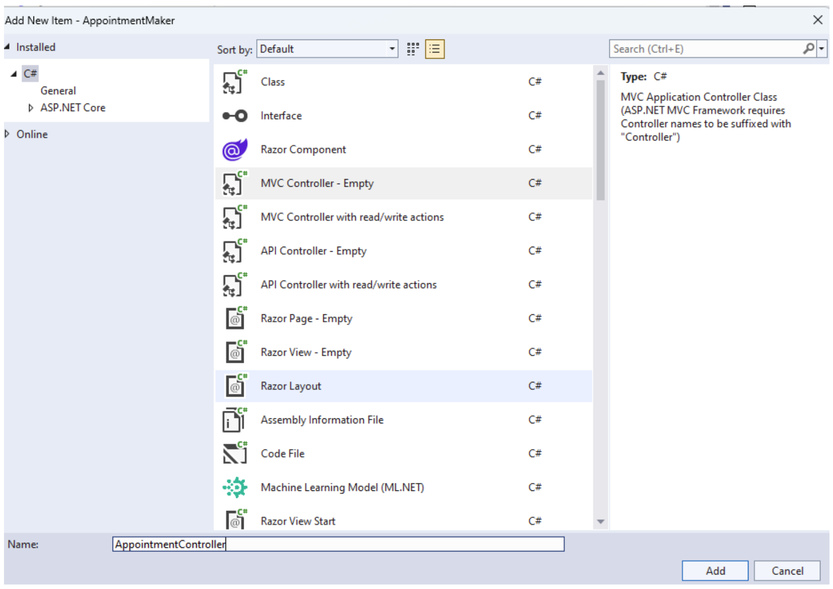
Figure 13 Adding a controller to the project.


Add New Scaffolded Item
Installed
Common
Controller
MVC 5 Controller - Empty
MVC 5 Controller with read/write actions
MVC 5 Controller with views, using Entity Framework
Web API 2 Controller – Empty
Web API 2 Controller with actions, using Entity Framework
Web API 2 Controller with read/write actions
Web API 2 OData v3 Controller with actions, using Entity Framework
Click here to go online and find more scaffolding extensions.
Add
Cancel

MVC 5 Controller - Empty
by Microsoft
v5.0.0.0
An empty MVC controller.
Id: MvcControllerEmptyScaffolder

Figure 14 Choosing the controller template.

2. Name the controller **AppointmentsController**.


Add New Item - AppointmentMaker
Installed
C#
General
ASP.NET Core
Online
Sort by: Default
Search (Ctrl+E)
Class C#
Interface C#
Razor Component C#
MVC Controller - Empty C#
MVC Controller with read/write actions C#
API Controller - Empty C#
API Controller with read/write actions C#
Razor Page - Empty C#
Razor View - Empty C#
Razor Layout C#
Assembly Information File C#
Code File C#
Machine Learning Model (ML.NET) C#
Razor View Start C#
Add
Cancel

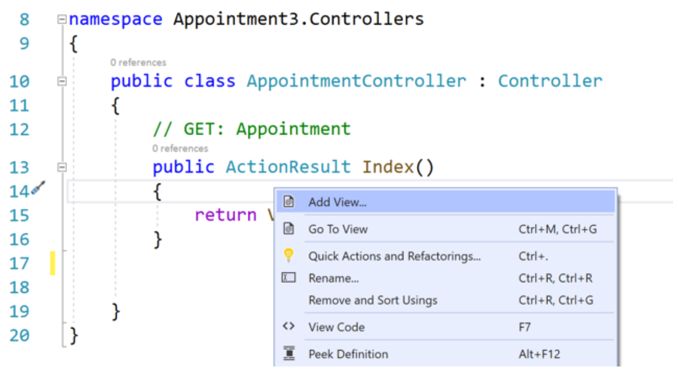
Type: C#
MVC Application Controller Class (ASP.NET MVC Framework requires Controller names to be suffixed with "Controller")

Name: AppointmentController

Figure 15 Creating a new Controller class.

Create an Input Form for the Appointment:

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



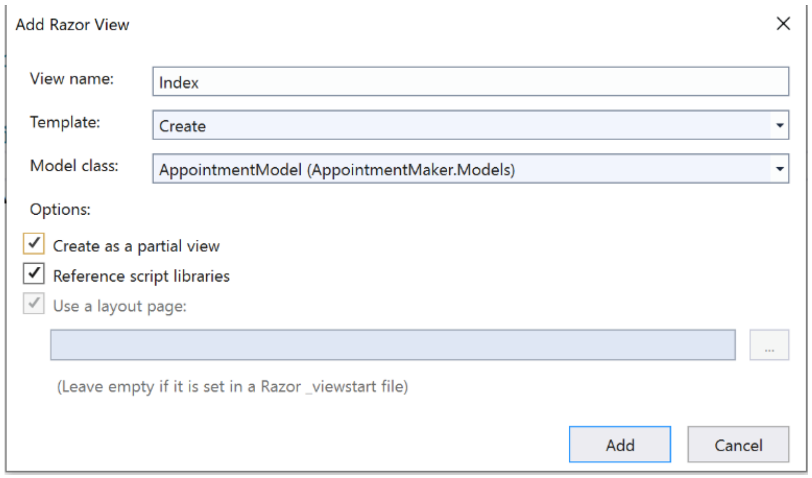
```
namespace Appointment3.Controllers
{
    0 references
    public class AppointmentController : Controller
    {
        // GET: Appointment
        0 references
        public ActionResult Index()
        {
            return V
        }


    }
}
```

Add View...
Go To View Ctrl+M, Ctrl+G
Quick Actions and Refactorings... Ctrl+.
Rename... Ctrl+R, Ctrl+R
Remove and Sort Usings Ctrl+R, Ctrl+G
View Code F7
Peek Definition Alt+F12

Figure 16 Adding a new view for the Index method.

For the View options choose:

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



Add Razor View

View name: Index

Template: Create

Model class: AppointmentModel (AppointmentMaker.Models)

Options:

- [✓] Create as a partial view
- [✓] Reference script libraries
- [✓] Use a layout page:

(Leave empty if it is set in a Razor _viewstart file)

Figure 17 Adding a "Create" razor view using the AppointmentModel for its base.

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

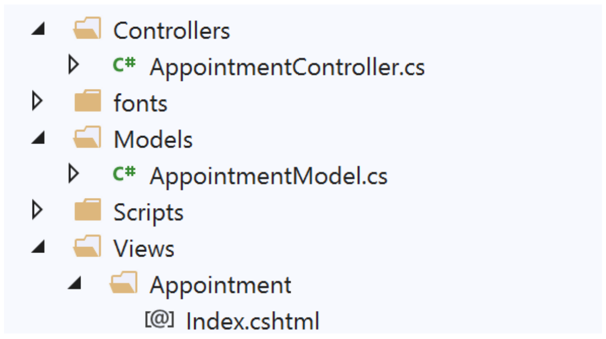
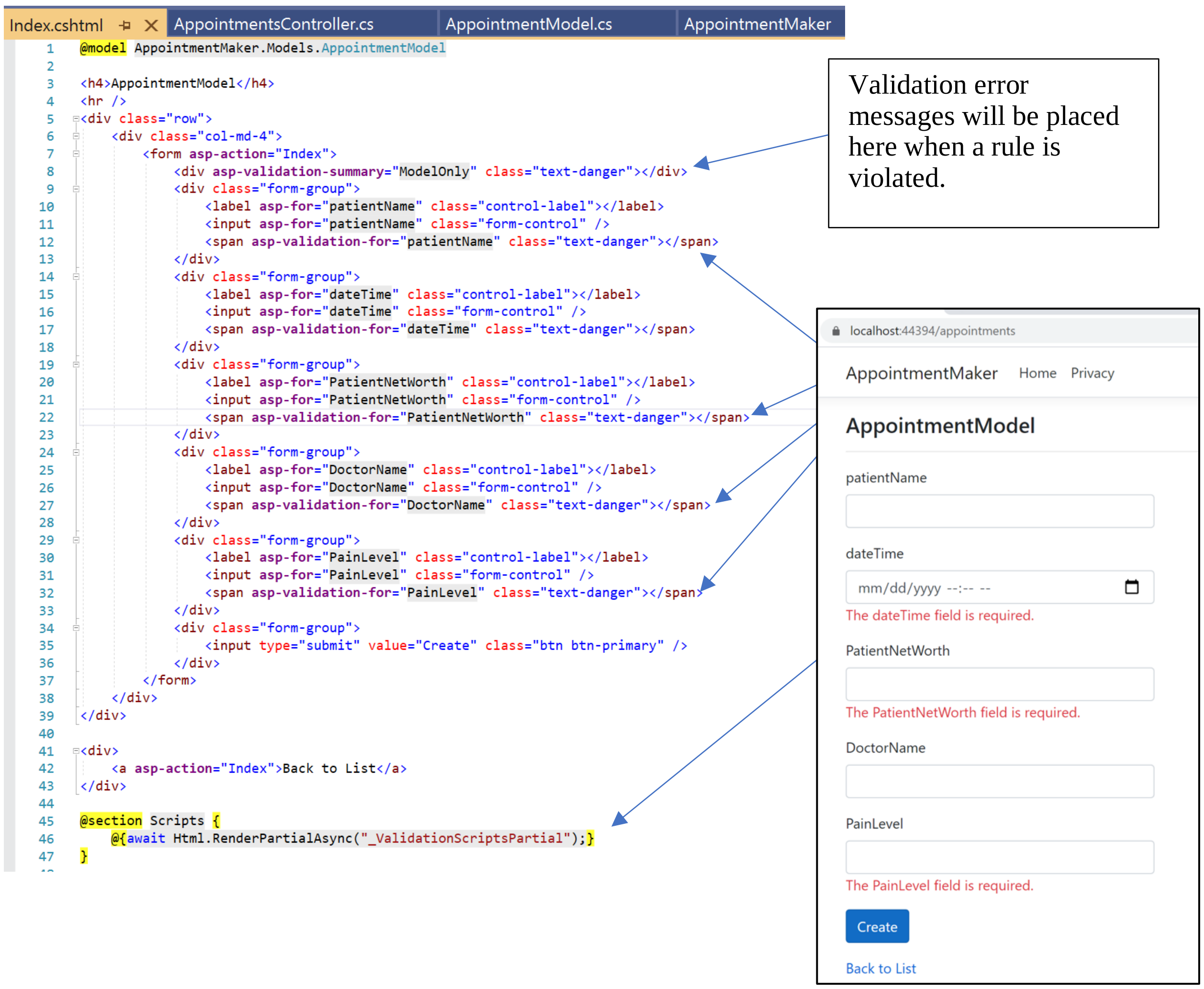


Figure 18 Location of the index view

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.



```
@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.

localhost:44394/appointments

AppointmentMaker Home Privacy

AppointmentModel

patientName

dateTime

mm/dd/yyyy --:-- --

The dateTime field is required.

PatientNetWorth

The PatientNetWorth field is required.

DoctorName

PainLevel

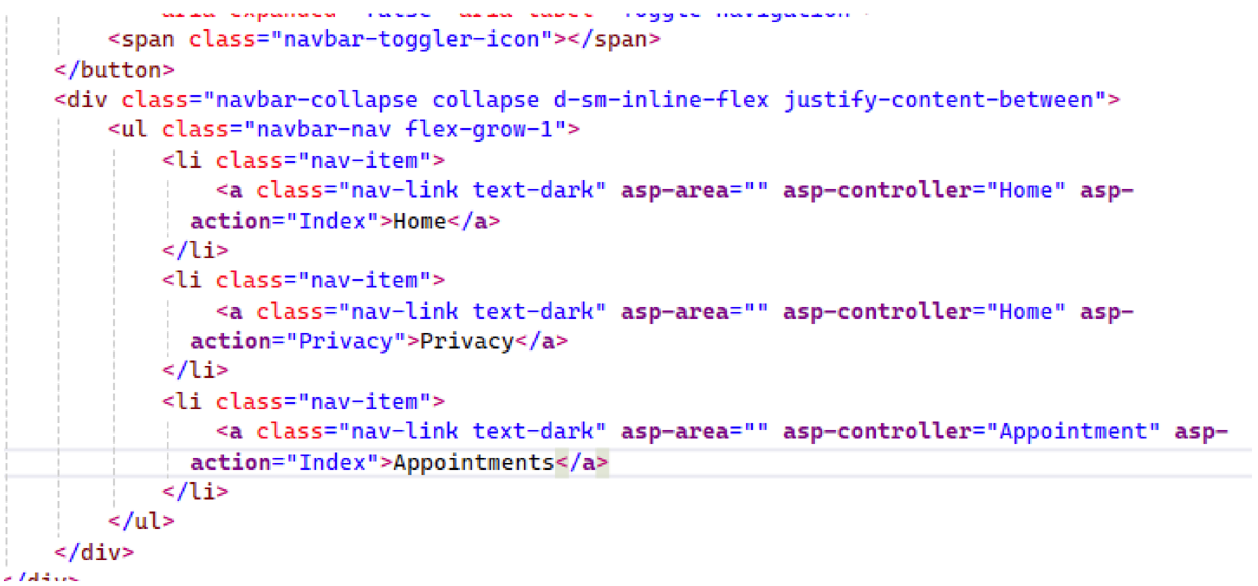
The PainLevel field is required.

Create

Back to List

Figure 19 Validation rules and how they relate the the warning messages on the user interface.

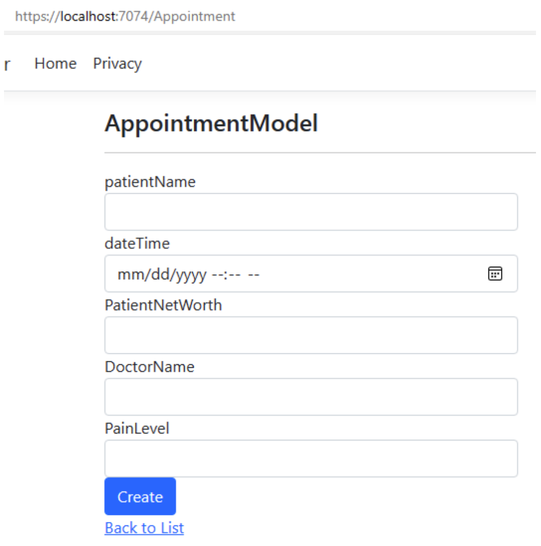
2. Create a new link in the navbar in shaed > _Layout.cshmtl



```
        <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>
            <li class="nav-item">
                <a class="nav-link text-dark" asp-area="" asp-controller="Appointment" asp-
              action="Index">Appointments</a>
            </li>
        </ul>
    </div>
```

Figure 20 New link to the Appointment controller

3. Run the application and navigate to /appointments.



https://localhost:7074/Appointment

r Home Privacy

AppointmentModel

patientName

dateTime

mm/dd/yyyy --:-- --

PatientNetWorth

DoctorName

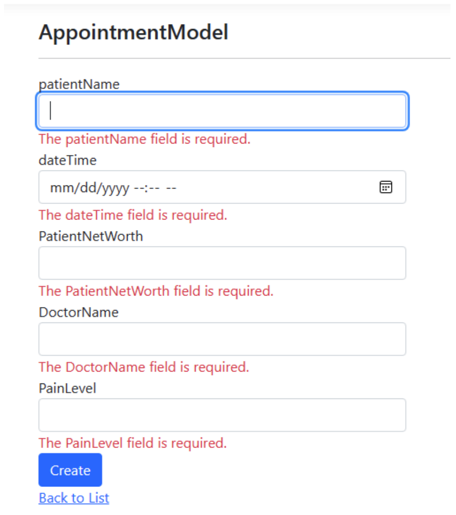
PainLevel

Create

Back to List

Figure 21 Appointment data entry form.

4. Attempt to submit an empty form. You should see validation errors for some of the values.


AppointmentModel

patientName

The patientName field is required.

dateTime

mm/dd/yyyy --:-- --

The dateTime field is required.

PatientNetWorth

The PatientNetWorth field is required.

DoctorName

The DoctorName field is required.

PainLevel

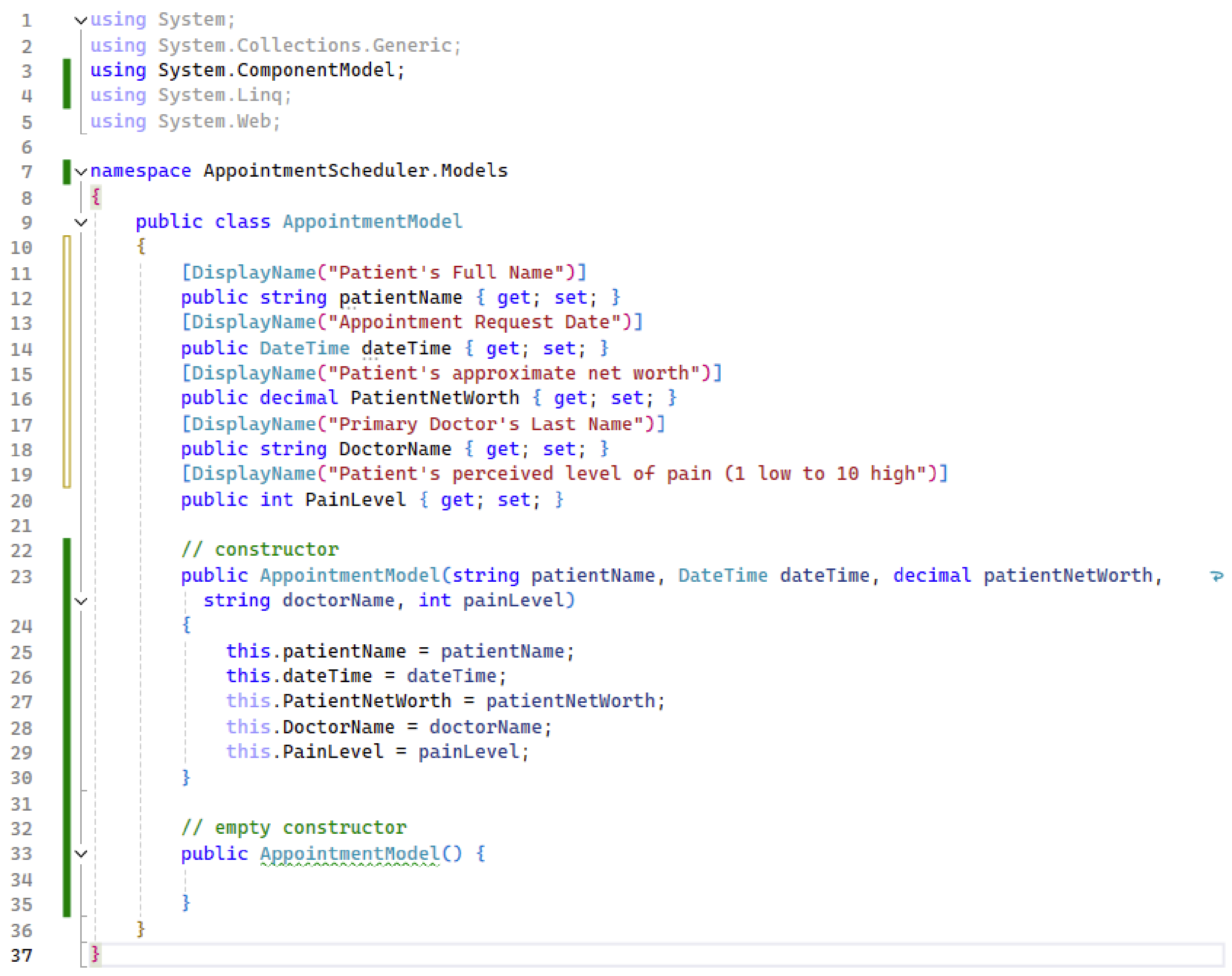
The PainLevel field is required.

Figure 22 Attempting to submit a request with no data in the fields.

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

Add Some Data Validation in the Model:

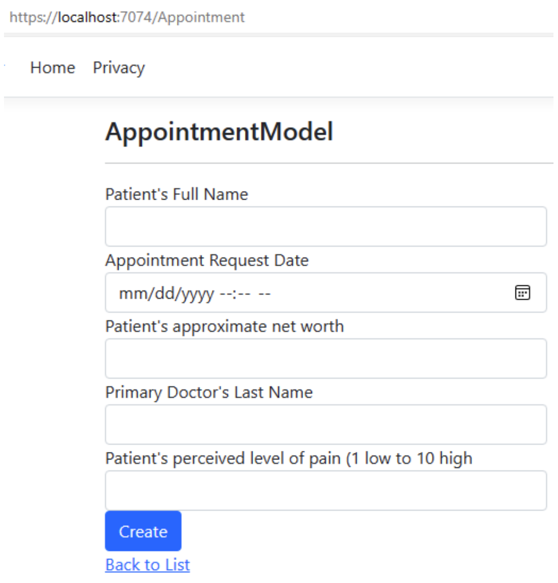
1. Create DisplayName values for each item in the form.



```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Linq;
using System.Web;

namespace AppointmentScheduler.Models
{
    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; }

        // constructor
        public AppointmentModel(string patientName, DateTime dateTime, decimal patientNetWorth,
          string doctorName, int painLevel)
        {
            this.patientName = patientName;
            this.dateTime = dateTime;
            this.PatientNetWorth = patientNetWorth;
            this.DoctorName = doctorName;
            this.PainLevel = painLevel;
        }

        // empty constructor
        public AppointmentModel() {

        }
    }
}
```

Figure 23 Each field now has a custom display name.

2. Run the program and notice each field name is customized.


https://localhost:7074/Appointment

Home Privacy

AppointmentModel

Patient's Full Name

Appointment Request Date

mm/dd/yyyy --:-- --

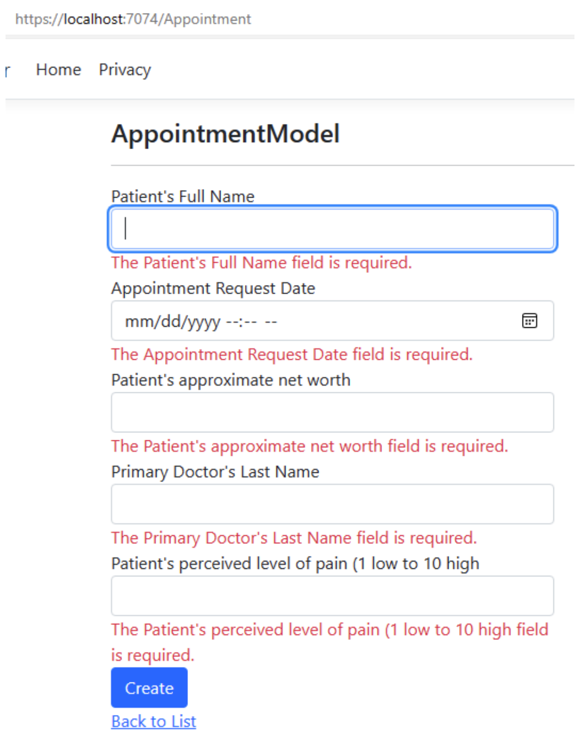
Patient's approximate net worth

Primary Doctor's Last Name

Patient's perceived level of pain (1 low to 10 high

Figure 24 Custom labels appear above each text entry according to the properties defined in the Appointment Model class.

3. You still cannot submit the form unless all of the filed are filled out.


https://localhost:7074/Appointment

AppointmentModel

Patient's Full Name

The Patient's Full Name field is required.

Appointment Request Date

mm/dd/yyyy --:-- --

The Appointment Request Date field is required.

Patient's approximate net worth

The Patient's approximate net worth field is required.

Primary Doctor's Last Name

The Primary Doctor's Last Name field is required.

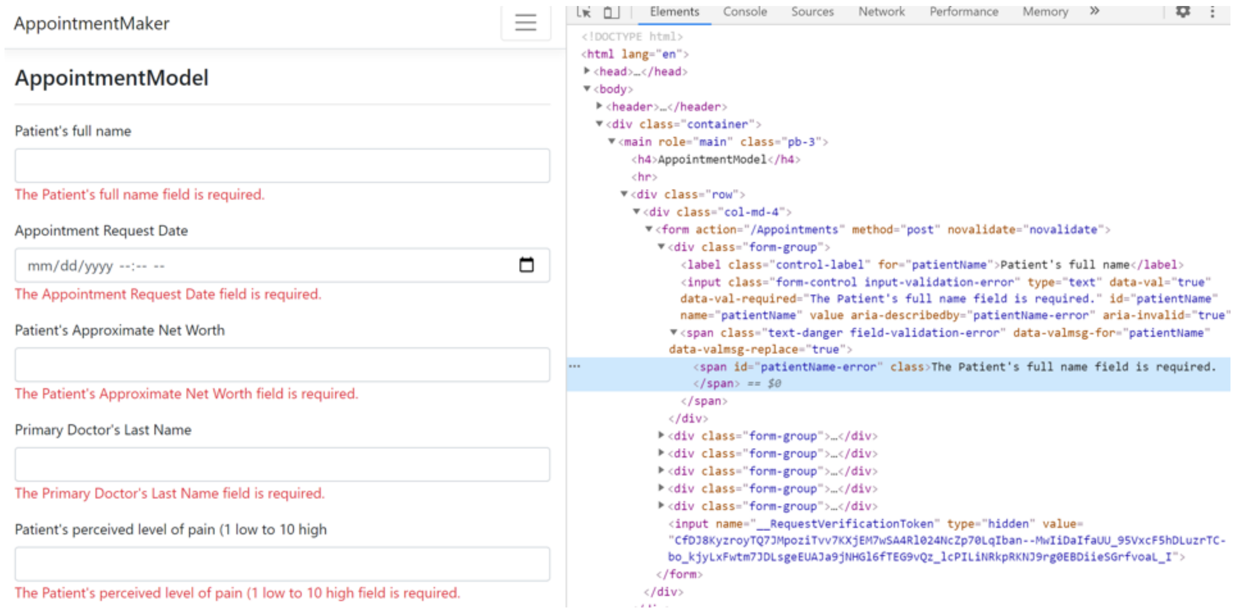
Patient's perceived level of pain (1 low to 10 high

The Patient's perceived level of pain (1 low to 10 high field is required.

Figure 25 Validation rules say values are required.

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

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



AppointmentModel

Patient's full name

The Patient's full name field is required.

Appointment Request Date

mm/dd/yyyy --:-- --

The Appointment Request Date field is required.

Patient's Approximate Net Worth

The Patient's Approximate Net Worth field is required.

Primary Doctor's Last Name

The Primary Doctor's Last Name field is required.

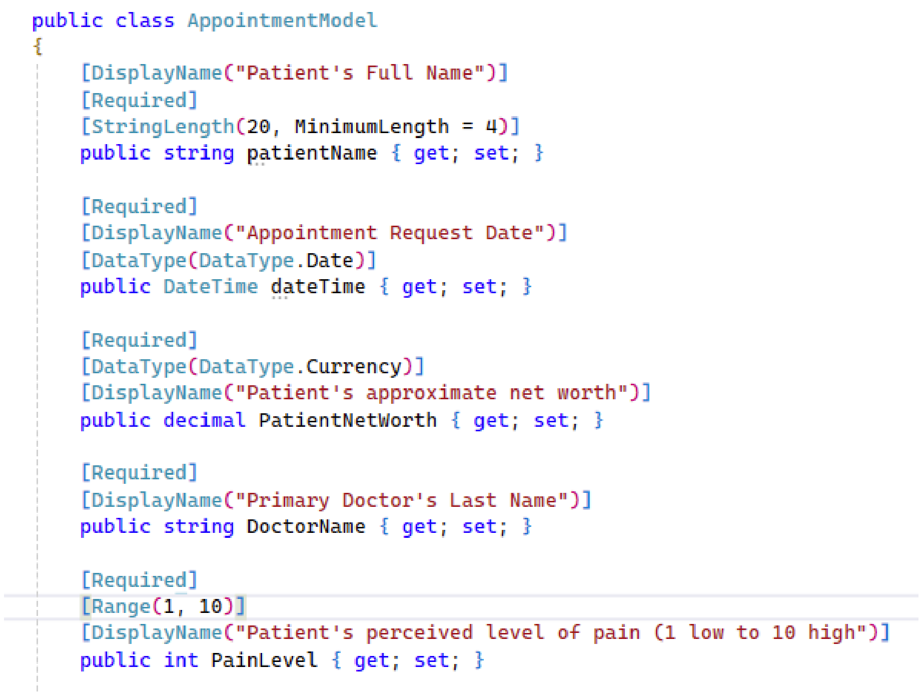
Patient's perceived level of pain (1 low to 10 high

The Patient's perceived level of pain (1 low to 10 high field is required.

```
<!DOCTYPE html>
<html lang="en">
 ▶<head>…</head>
 ▼<body>
   ▶<header>…</header>
   ▼<div class="container">
     ▼<main role="main" class="pb-3">
        <h4>AppointmentModel</h4>
        <hr>
       ▼<div class="row">
         ▼<div class="col-md-4">
           ▼<form action="/Appointments" method="post" novalidate="novalidate">
             ▼<div class="form-group">
                <label class="control-label" for="patientName">Patient's full name</label>
                <input class="form-control input-validation-error" type="text" data-val="true"
                data-val-required="The Patient's full name field is required." id="patientName"
                name="patientName" value aria-describedby="patientName-error" aria-invalid="true"
               ▼<span class="text-danger field-validation-error" data-valmsg-for="patientName"
               data-valmsg-replace="true">
...               <span id="patientName-error" class>The Patient's full name field is required.
                  </span> == $0
                </span>
              </div>
             ▶<div class="form-group">…</div>
             ▶<div class="form-group">…</div>
             ▶<div class="form-group">…</div>
             ▶<div class="form-group">…</div>
             ▶<div class="form-group">…</div>
              <input name="__RequestVerificationToken" type="hidden" value=
              "CfDJ8KyzroyTQ7JMpoziTvv7KXjEM7wSA4Rl024NcZp70LqIban--MwIiDaIfaUU_95VxcF5hDLuzrTC-
              bo_kjyLxFwtm7JDLsgeEUAJa9jNHGl6fTEG9vQz_lcPILiNRkpRKNJ9rg0EBDiieSGrfvoaL_I">
            </form>
          </div>
```

Figure 26 Validation rules are rendered as <span> elements in the html. Css class names are used to make them visible when appropriate.

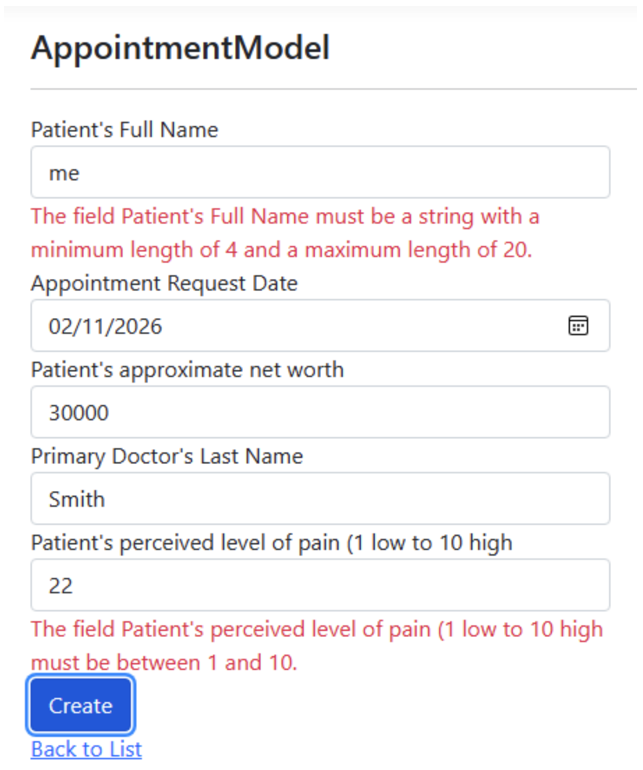
5. Add some additional Data validation rules to the model.
    - The patient's name should be between 4 and 20 characters long.
    - The Appointment Date should be DateTime data format.
    - The Doctor's name should not be required.
    - The PatientNetWorth should be currency data type.
    - The painLevel should be between 1 and 10.



```
public class AppointmentModel
{
    [DisplayName("Patient's Full Name")]
    [Required]
    [StringLength(20, MinimumLength = 4)]
    public string patientName { get; set; }

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

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

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

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

Figure 27 Some arbitrary rules are applied to making a new appointment.

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


AppointmentModel

Patient's Full Name

me

The field Patient's Full Name must be a string with a minimum length of 4 and a maximum length of 20.

Appointment Request Date

02/11/2026

Patient's approximate net worth

30000

Primary Doctor's Last Name

Smith

Patient's perceived level of pain (1 low to 10 high

22

The field Patient's perceived level of pain (1 low to 10 high must be between 1 and 10.

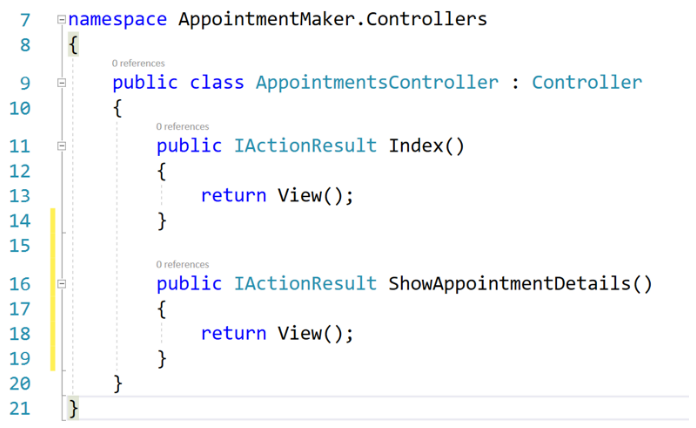
Figure 28 The new rules are not being followed by the patient.

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

Add a Response Page to the Appointment:

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

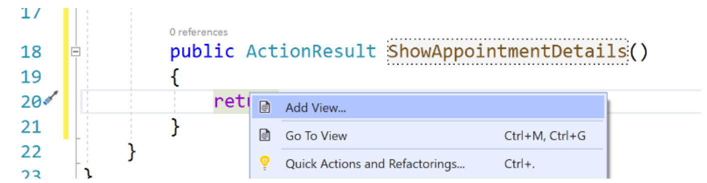
1. In the controller, add a new method called **ShowAppointmentDetails**.



```
namespace AppointmentMaker.Controllers
{
    0 references
    public class AppointmentsController : Controller
    {
        0 references
        public IActionResult Index()
        {
            return View();
        }

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

Figure 29 New method "ShowAppointmentDetails" added to the controller.

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

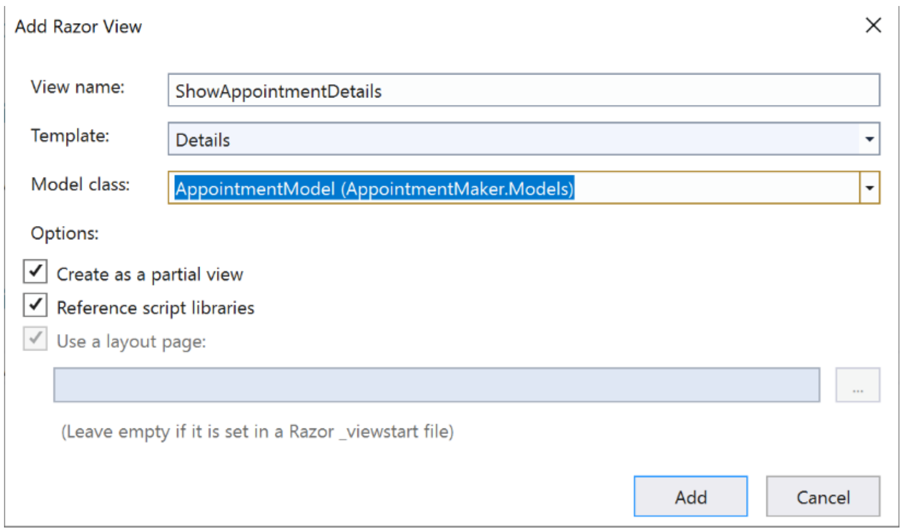


```
        0 references
        public ActionResult ShowAppointmentDetails()
        {
            ret
        }
    }
}
```

Add View...
Go To View Ctrl+M, Ctrl+G
Quick Actions and Refactorings... Ctrl+.

Figure 30 Adding a new view.

3. Choose the following:
   - View Name: **ShowAppointmentDetails**
   - Template: **Details**
   - Model Class: **AppointmentModel**


Add Razor View

View name: ShowAppointmentDetails

Template: Details

Model class: AppointmentModel (AppointmentMaker.Models)

Options:

☑ Create as a partial view

☑ Reference script libraries

☑ Use a layout page:

(Leave empty if it is set in a Razor _viewstart file)

Figure 31 Adding a new view based on the Appointment Model.

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

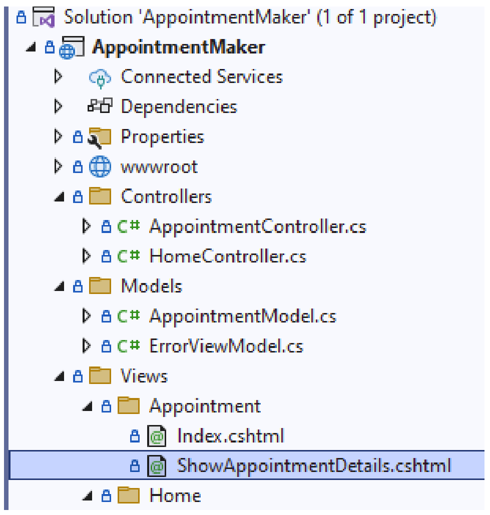
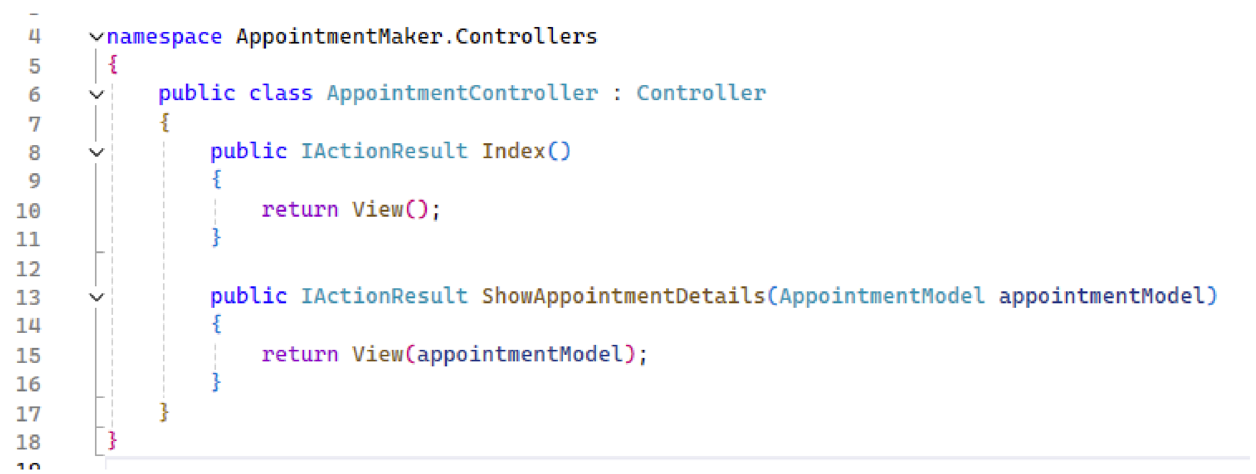

```
Solution 'AppointmentMaker' (1 of 1 project)
  AppointmentMaker
    Connected Services
    Dependencies
    Properties
    wwwroot
    Controllers
      AppointmentController.cs
      HomeController.cs
    Models
      AppointmentModel.cs
      ErrorViewModel.cs
    Views
      Appointment
        Index.cshtml
        ShowAppointmentDetails.cshtml
      Home
```

Figure 32 Location of the new view.

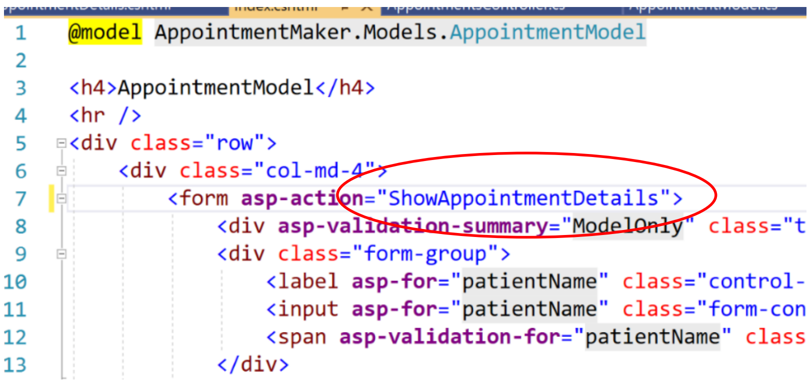
4. In the AppointmentController add a new **AppointmentModel** parameter to the ShowAppointmentDetails method. Pass the appointmentModel to the view.



```
namespace AppointmentMaker.Controllers
{
    public class AppointmentController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }

        public IActionResult ShowAppointmentDetails(AppointmentModel appointmentModel)
        {
            return View(appointmentModel);
        }
    }
}
```

Figure 33 ShowAppointmentDetails has a parameter to pass data to the next view.

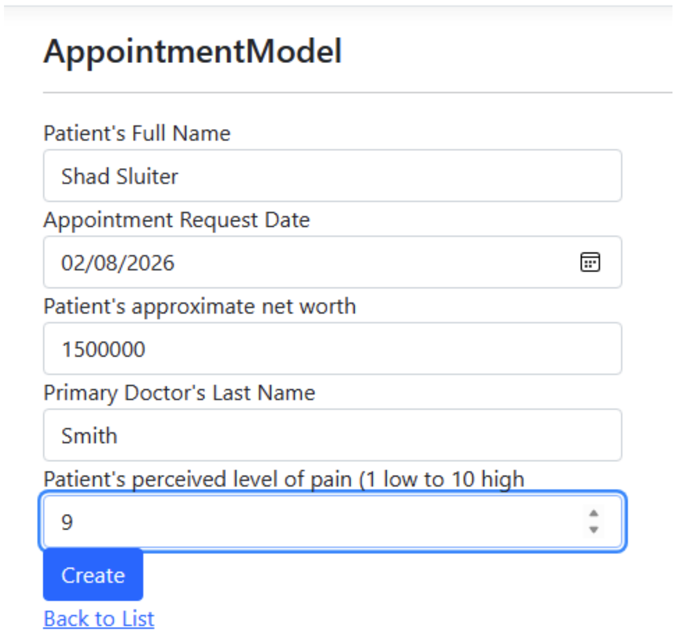
5. Change the form's **action** property in the Index.cshtml page to point to the new controller method.



```
@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>
```

Figure 34 Form action is updated

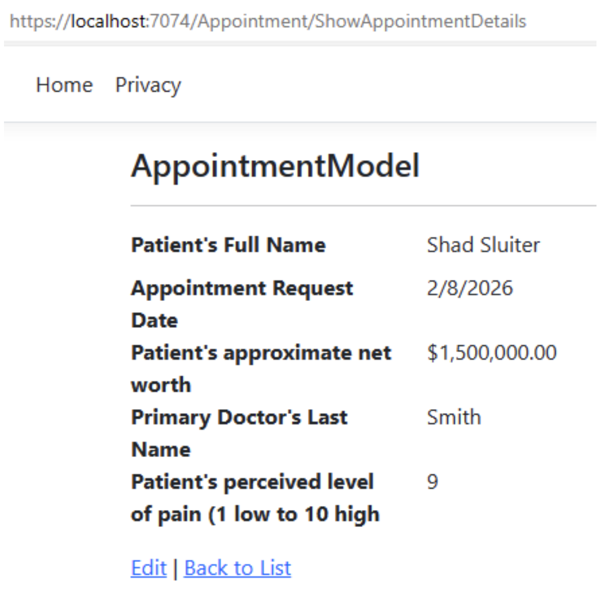
6. Try to run the program.


AppointmentModel

Patient's Full Name	Shad Sluiter
Appointment Request Date	02/08/2026
Patient's approximate net worth	1500000
Primary Doctor's Last Name	Smith
Patient's perceived level of pain (1 low to 10 high	9

Figure 35 Filling out an appointment following all validation rules.

7. Submit a request


https://localhost:7074/Appointment/ShowAppointmentDetails

AppointmentModel

Patient's Full Name	Shad Sluiter
Appointment Request Date	2/8/2026
Patient's approximate net worth	$1,500,000.00
Primary Doctor's Last Name	Smith
Patient's perceived level of pain (1 low to 10 high	9

Figure 36 Successfully submitted an appointment.

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

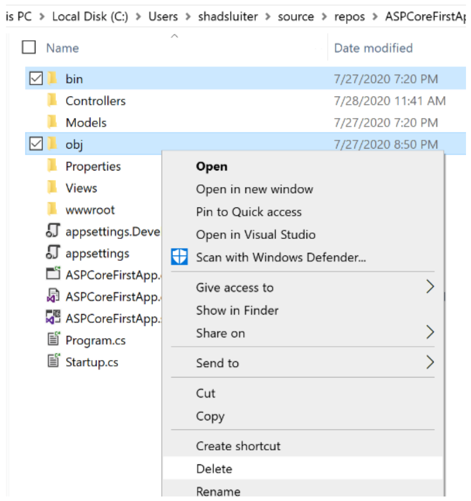
Programming Challenge:

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**".

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

Deliverables:


is PC > Local Disk (C:) > Users > shadsluiter > source > repos > ASPCoreFirstA

Name	Date modified
bin	7/27/2020 7:20 PM
Controllers	7/28/2020 11:41 AM
Models	7/27/2020 7:20 PM
obj	7/27/2020 8:50 PM
Properties	
Views	
wwwroot	
appsettings.Deve	
appsettings	
ASPCoreFirstApp.	
ASPCoreFirstApp.	
ASPCoreFirstApp.	
Program.cs	
Startup.cs	

1. This activity has multiple parts. Complete all parts before submitting.
2. Create 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. Turn the Word document into a PDF.
5. 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.
6. Attach the PDF separately from the zip file. Multiple files can be uploaded with an assignment.

GRAND CANYON UNIVERSITY™

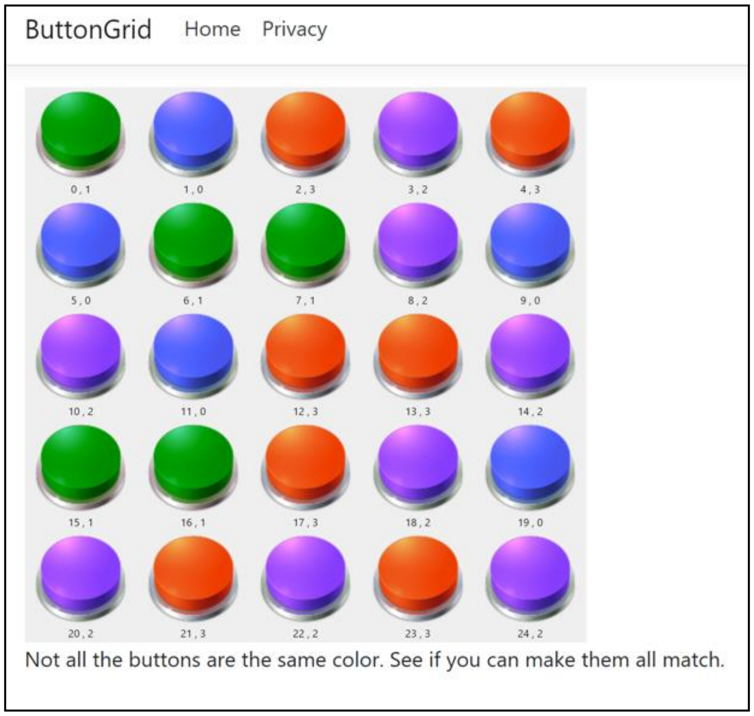
Part 3 Button Grid – Manage Dynamic Objects on a View

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.



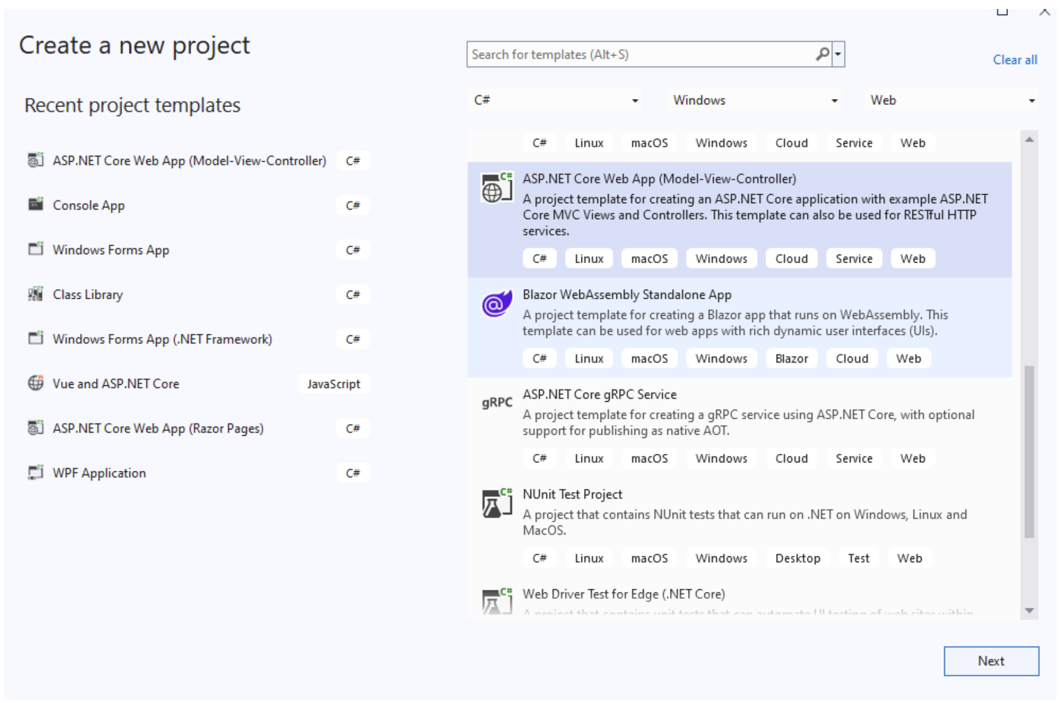
Not all the buttons are the same color. See if you can make them all match.

Figure 37 Button grid game to be created in this lesson.

GRAND CANYON UNIVERSITY™

Instructions:

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



Create a new project

Search for templates (Alt+S)

Clear all

C# | Windows | Web

Recent project templates

ASP.NET Core Web App (Model-View-Controller)	C#
Console App	C#
Windows Forms App	C#
Class Library	C#
Windows Forms App (.NET Framework)	C#
Vue and ASP.NET Core	JavaScript
ASP.NET Core Web App (Razor Pages)	C#
WPF Application	C#

C# Linux macOS Windows Cloud Service Web

ASP.NET Core Web App (Model-View-Controller)
A project template for creating an ASP.NET Core application with example ASP.NET Core MVC Views and Controllers. This template can also be used for RESTful HTTP services.
C# Linux macOS Windows Cloud Service Web

Blazor WebAssembly Standalone App
A project template for creating a Blazor app that runs on WebAssembly. This template can be used for web apps with rich dynamic user interfaces (UIs).
C# Linux macOS Windows Blazor Cloud Web

ASP.NET Core gRPC Service
A project template for creating a gRPC service using ASP.NET Core, with optional support for publishing as native AOT.
C# Linux macOS Windows Cloud Service Web

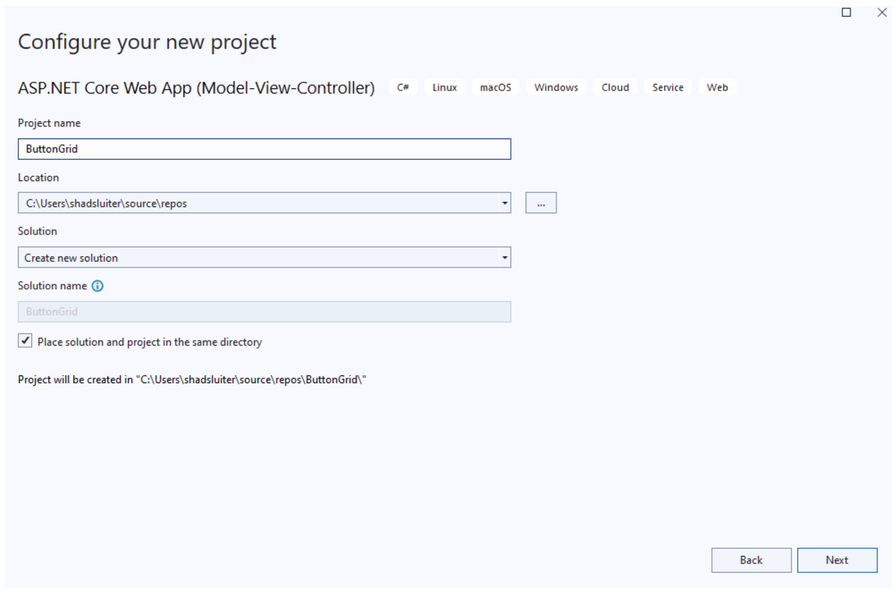
NUnit Test Project
A project that contains NUnit tests that can run on .NET on Windows, Linux and MacOS.
C# Linux macOS Windows Desktop Test Web

Web Driver Test for Edge (.NET Core)

Next

Figure 38 Creating a new project

2. Name is ButtonGrid.



Configure your new project

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

Project name: ButtonGrid

Location: C:\Users\shadsluiter\source\repos

Solution: Create new solution

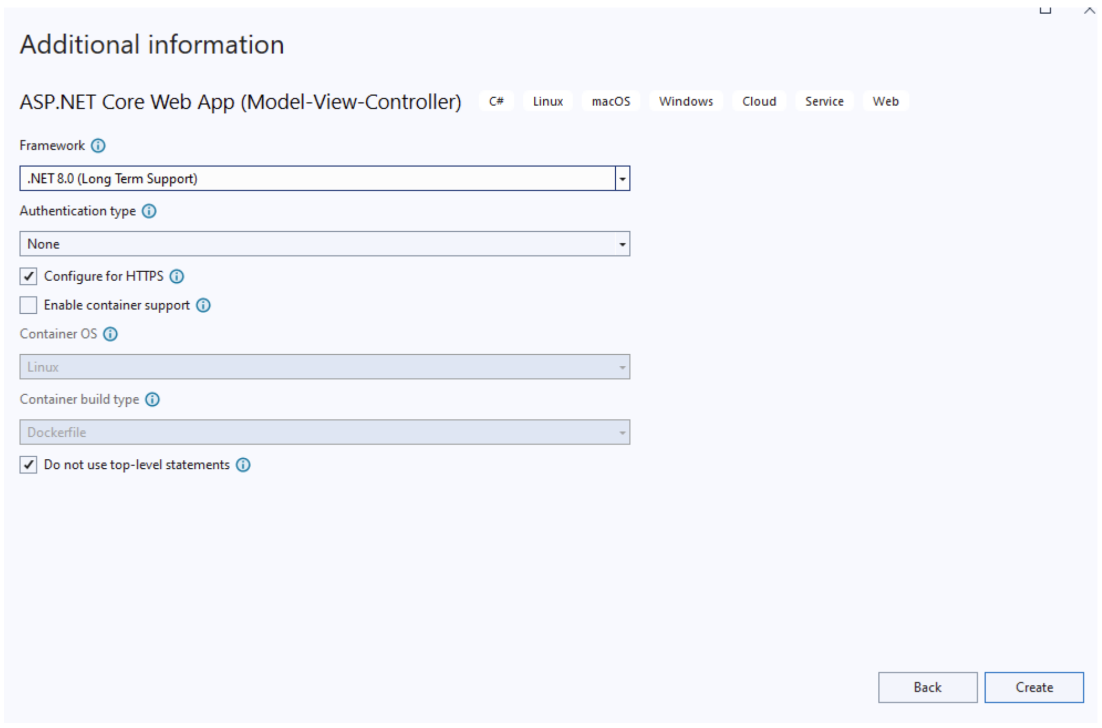
Solution name: ButtonGrid

☑ Place solution and project in the same directory

Project will be created in "C:\Users\shadsluiter\source\repos\ButtonGrid\"

Figure 39 naming the project

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



Additional information

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

Framework: .NET 8.0 (Long Term Support)

Authentication type: None

☑ Configure for HTTPS

☐ Enable container support

Container OS: Linux

Container build type: Dockerfile

☑ Do not use top-level statements

Figure 40 Options for the new project

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

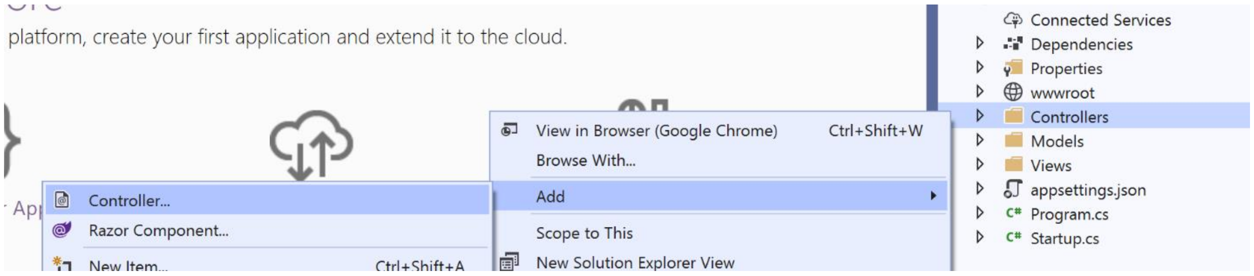


Figure 41 Adding a new controller.

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

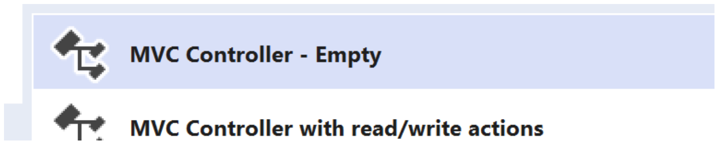
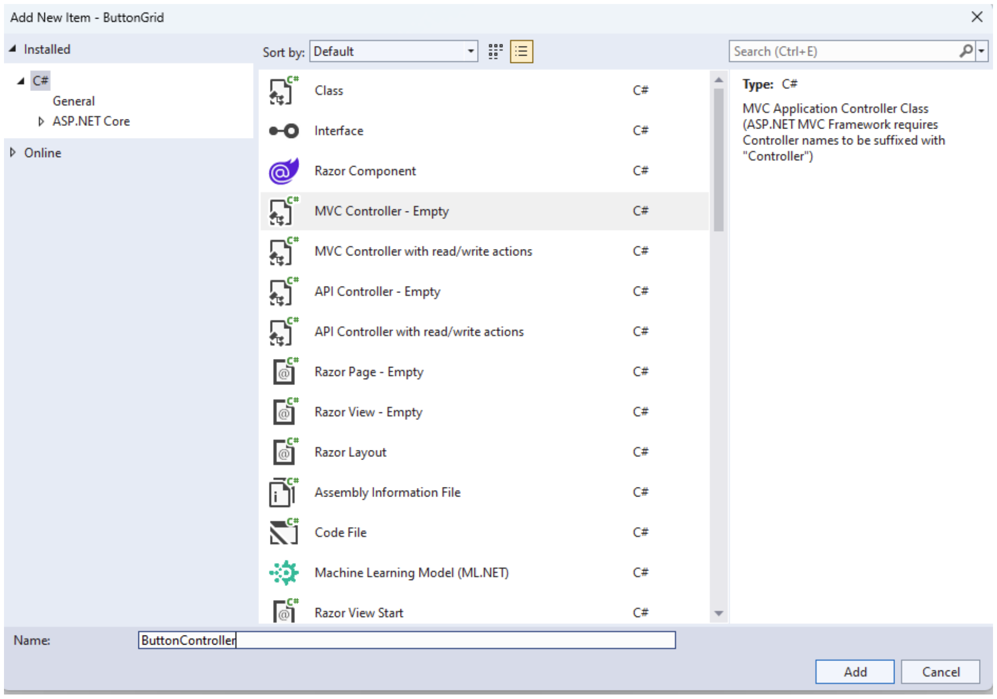


Figure 42 Empty controller option

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

GRAND CANYON UNIVERSITY™


Add New Item - ButtonGrid
Installed
C#
General
ASP.NET Core
Online
Sort by: Default
Search (Ctrl+E)
Class C#
Interface C#
Razor Component C#
MVC Controller - Empty C#
MVC Controller with read/write actions C#
API Controller - Empty C#
API Controller with read/write actions C#
Razor Page - Empty C#
Razor View - Empty C#
Razor Layout C#
Assembly Information File C#
Code File C#
Machine Learning Model (ML.NET) C#
Razor View Start C#
Type: C#
MVC Application Controller Class (ASP.NET MVC Framework requires Controller names to be suffixed with "Controller")
Add
Cancel

Name: ButtonController

Figure 43 Adding a new controller

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.

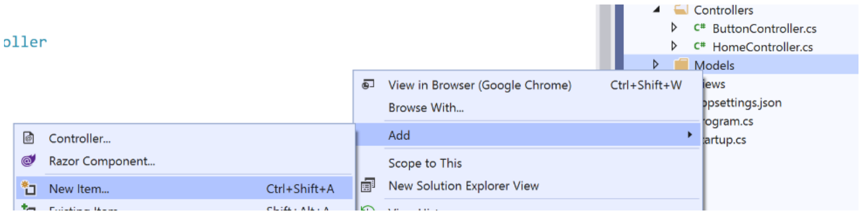
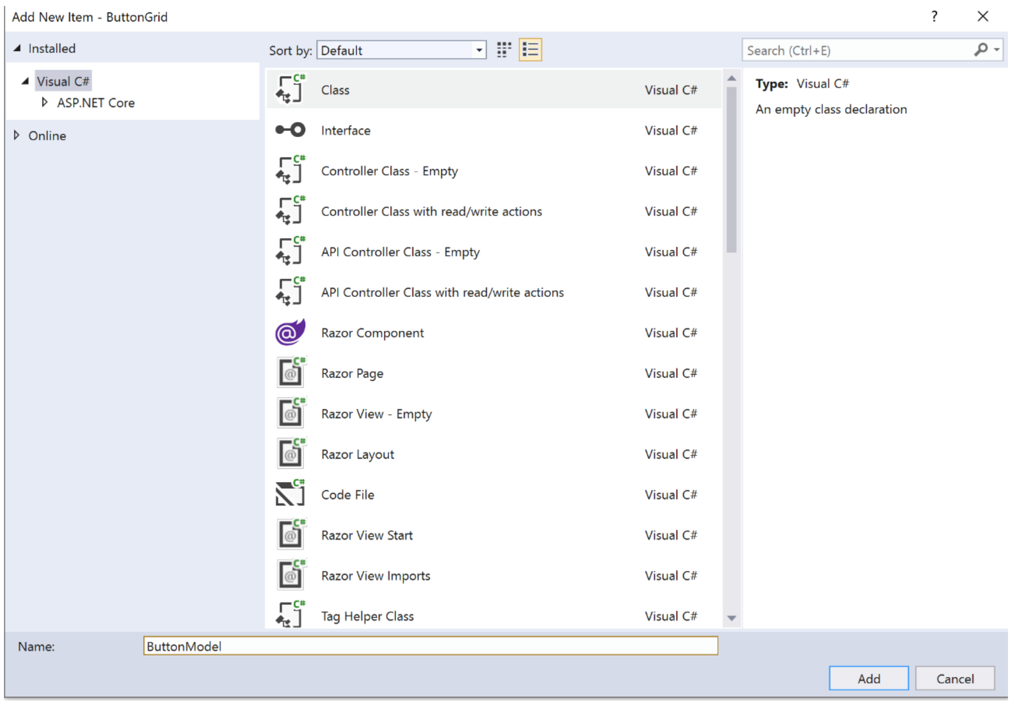


Figure 44 Adding a new item to the models folder

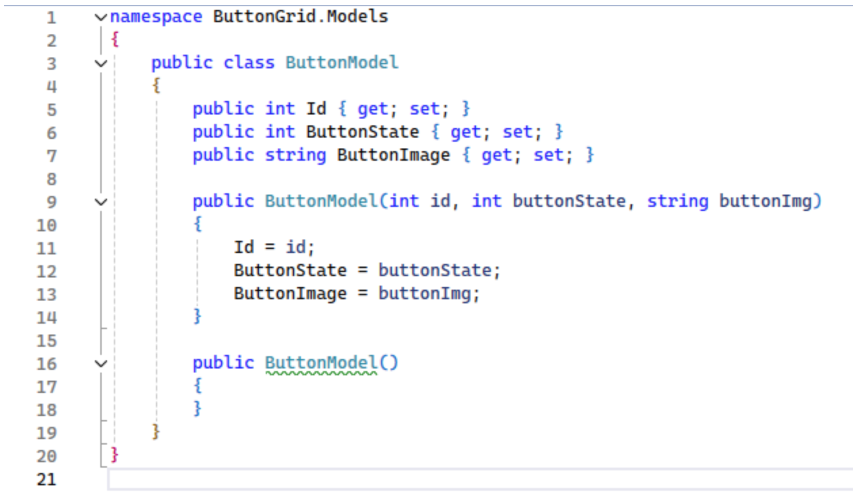
    3. Choose **Class**. Name the new file **ButtonModel**. Click **Add**.


Add New Item - ButtonGrid
Installed
Visual C#
ASP.NET Core
Online
Sort by: Default
Search (Ctrl+E)
Class Visual C#
Interface Visual C#
Controller Class - Empty Visual C#
Controller Class with read/write actions Visual C#
API Controller Class - Empty Visual C#
API Controller Class with read/write actions Visual C#
Razor Component Visual C#
Razor Page Visual C#
Razor View - Empty Visual C#
Razor Layout Visual C#
Code File Visual C#
Razor View Start Visual C#
Razor View Imports Visual C#
Tag Helper Class Visual C#
Type: Visual C#
An empty class declaration
Add
Cancel

Name: ButtonModel

Figure 45 Naming the item ButtonModel

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.



```
namespace ButtonGrid.Models
{
    public class ButtonModel
    {
        public int Id { get; set; }
        public int ButtonState { get; set; }
        public string ButtonImage { get; set; }

        public ButtonModel(int id, int buttonState, string buttonImg)
        {
            Id = id;
            ButtonState = buttonState;
            ButtonImage = buttonImg;
        }

        public ButtonModel()
        {
        }
    }
}
```

Figure 46 Properties for the ButtonModel

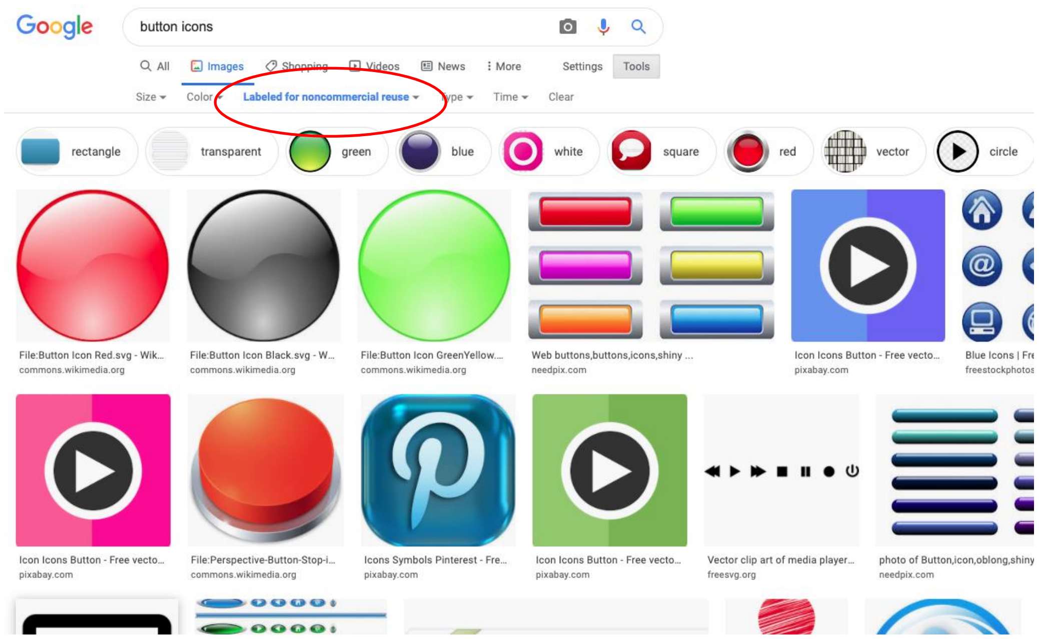

Google
button icons
All
Images
Shopping
Videos
News
More
Settings
Tools
Size
Color
Labeled for noncommercial reuse
Type
Time
Clear
rectangle
transparent
green
blue
white
square
red
vector
circle
File:Button Icon Red.svg - Wik...
commons.wikimedia.org
File:Button Icon Black.svg - W...
commons.wikimedia.org
File:Button Icon GreenYellow...
commons.wikimedia.org
Web buttons,buttons,icons,shiny ...
needpix.com
Icon Icons Button - Free vecto...
pixabay.com
Icon Icons Button - Free vecto...
pixabay.com
File:Perspective-Button-Stop-i...
commons.wikimedia.org
Icons Symbols Pinterest - Fre...
pixabay.com
Icon Icons Button - Free vecto...
pixabay.com
Vector clip art of media player...
freesvg.org
photo of Button,icon,oblong,shiny
needpix.com

Figure 47 Searching Google images for a button icon

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

Here is an image on Wikipedia

Right click and save.

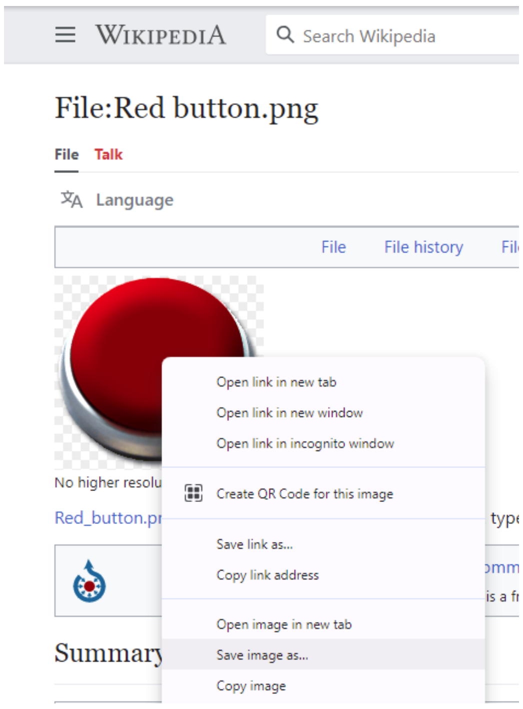
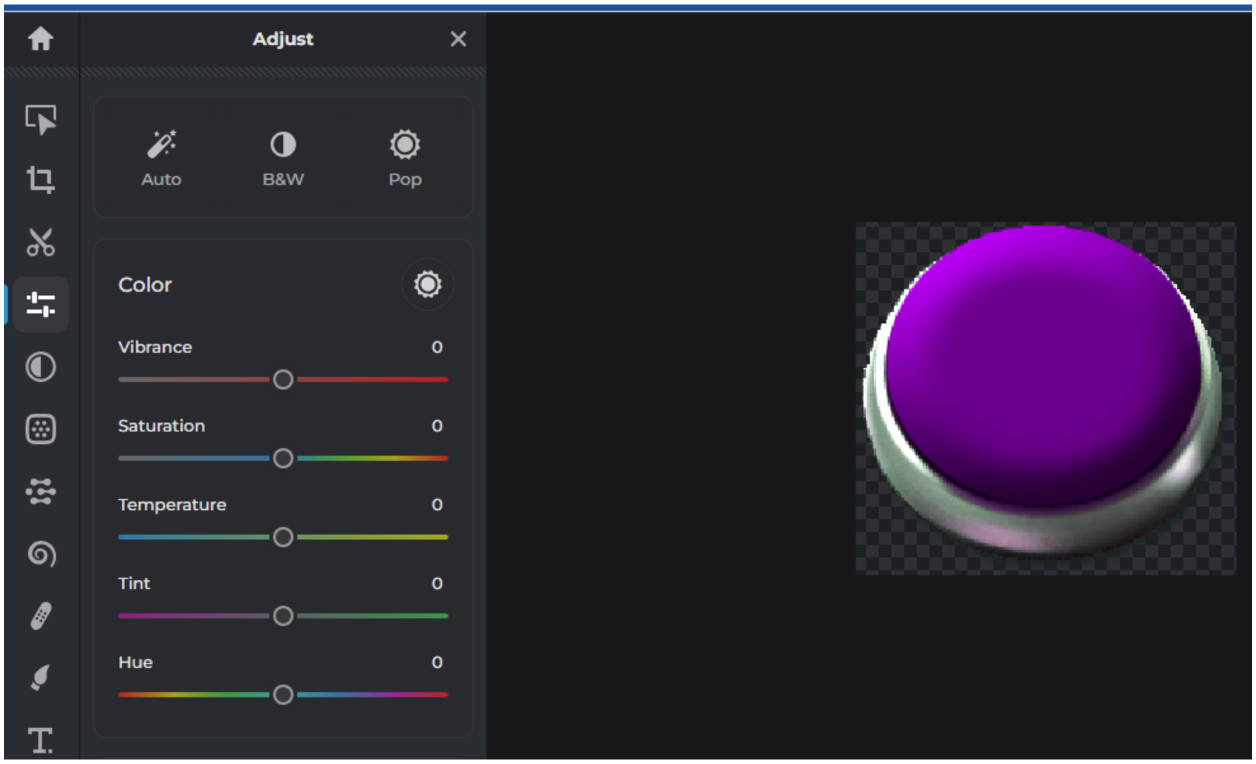


Figure 48 Saving an image to the computer.

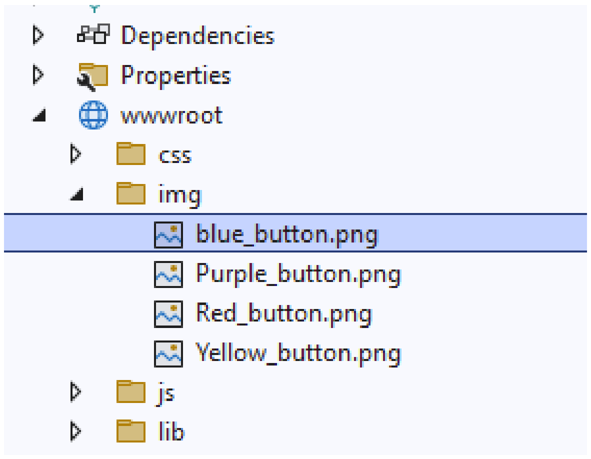
6. Use an image editing program to resize the images to a square size between 50x50 or 100x100 pixels.
    - Use an image editor such as pixlr to resize and recolor the image. You can copy the URL of a photo and paste it directly into the online picture editor.
    - You can also use Photoshop or Microsoft Paint to perform the tasks shown here.
7. Adjust the hue to get a new color



Vibrance 0
Saturation 0
Temperature 0
Tint 0
Hue 0

Figure 49 Adjusting the Hue of an image to change from Green to Purpose.

8. Click the “Save” button to save the current version of the image.
9. Create a folder named **img** inside the **wwwroot** folder in the solution explorer. Copy and paste the new images into the img folder.



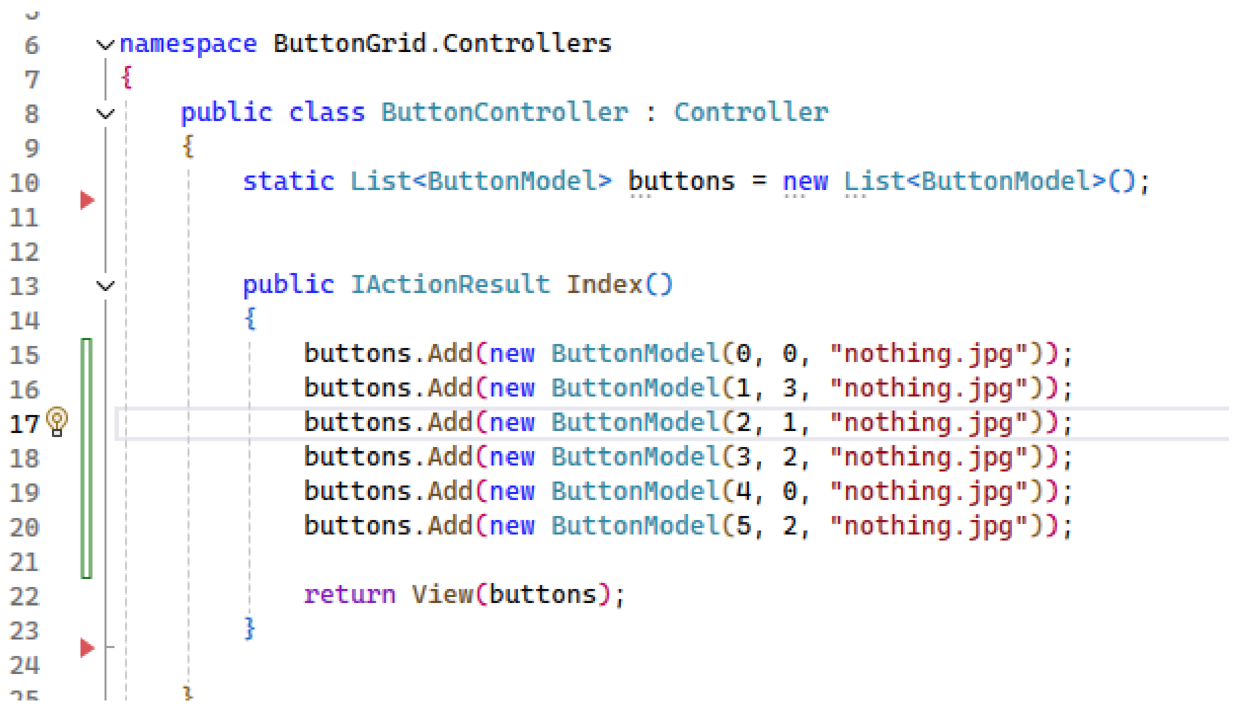
```
Dependencies
Properties
wwwroot
    css
    img
        blue_button.png
        Purple_button.png
        Red_button.png
        Yellow_button.png
    js
    lib
```

Figure 50 Four image files are in 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.

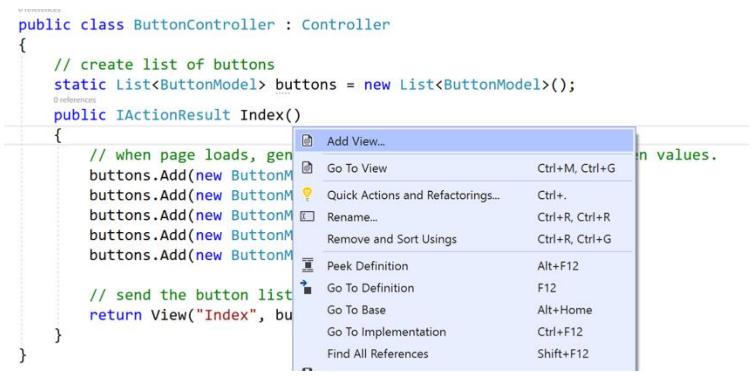
2. In the Index() of the Button Controller, add several ButtonModels to the buttons list. Pass this list as model data to the Button View.



```
namespace ButtonGrid.Controllers
{
    public class ButtonController : Controller
    {
        static List<ButtonModel> buttons = new List<ButtonModel>();


        public IActionResult Index()
        {
            buttons.Add(new ButtonModel(0, 0, "nothing.jpg"));
            buttons.Add(new ButtonModel(1, 3, "nothing.jpg"));
            buttons.Add(new ButtonModel(2, 1, "nothing.jpg"));
            buttons.Add(new ButtonModel(3, 2, "nothing.jpg"));
            buttons.Add(new ButtonModel(4, 0, "nothing.jpg"));
            buttons.Add(new ButtonModel(5, 2, "nothing.jpg"));

            return View(buttons);
        }

    }
```

Figure 51 Button controller now has a list of button models with five elements

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

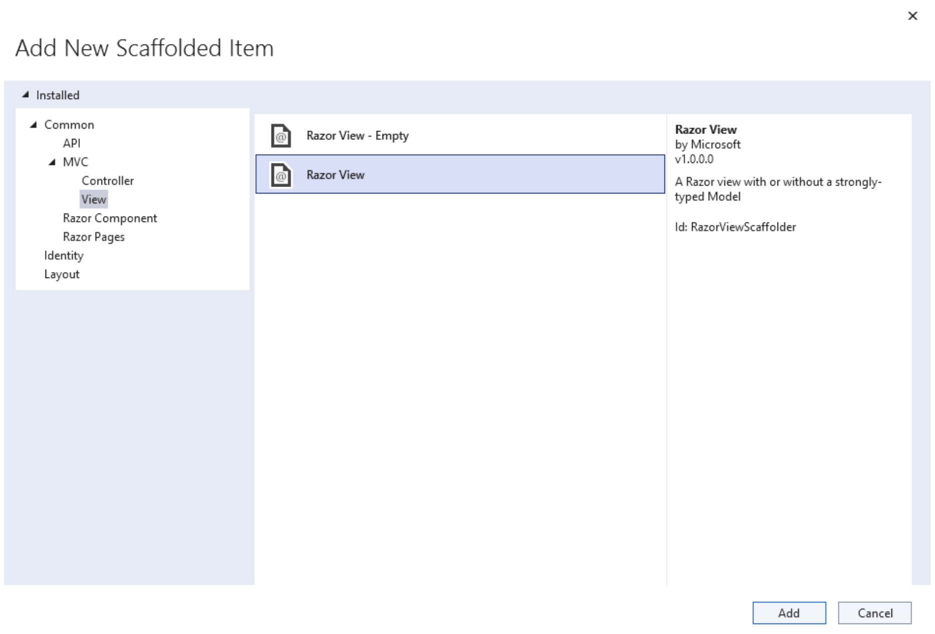


```
public class ButtonController : Controller
{
    // create list of buttons
    static List<ButtonModel> buttons = new List<ButtonModel>();
    public IActionResult Index()
    {
        // when page loads, gen                                    n values.
        buttons.Add(new ButtonM
        buttons.Add(new ButtonM
        buttons.Add(new ButtonM
        buttons.Add(new ButtonM
        buttons.Add(new ButtonM

        // send the button list
        return View("Index", bu
    }
}
```

Add View...
Go To View Ctrl+M, Ctrl+G
Quick Actions and Refactorings... Ctrl+.
Rename... Ctrl+R, Ctrl+R
Remove and Sort Usings Ctrl+R, Ctrl+G
Peek Definition Alt+F12
Go To Definition F12
Go To Base Alt+Home
Go To Implementation Ctrl+F12
Find All References Shift+F12

Figure 52 adding a new view

10. Choose Razor View

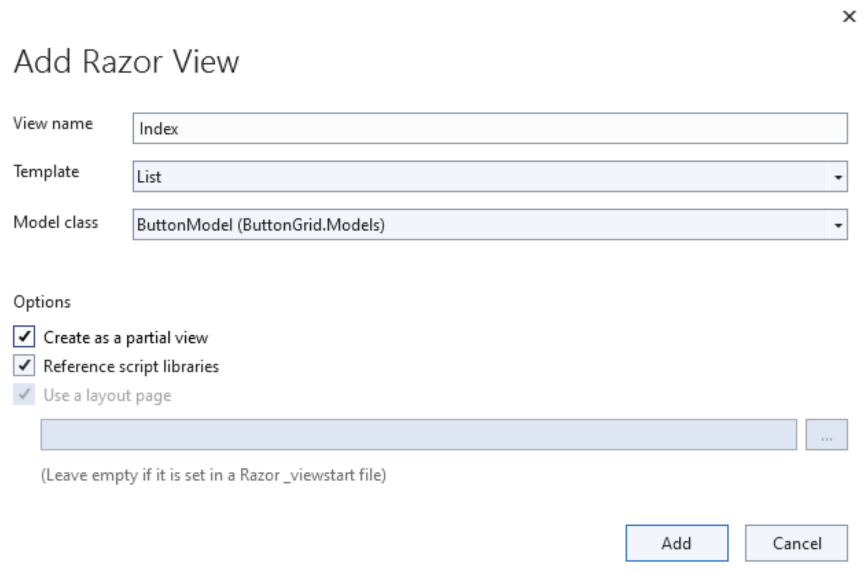


Razor View
by Microsoft
v1.0.0.0
A Razor view with or without a strongly-typed Model

Id: RazorViewScaffolder

Figure 53

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



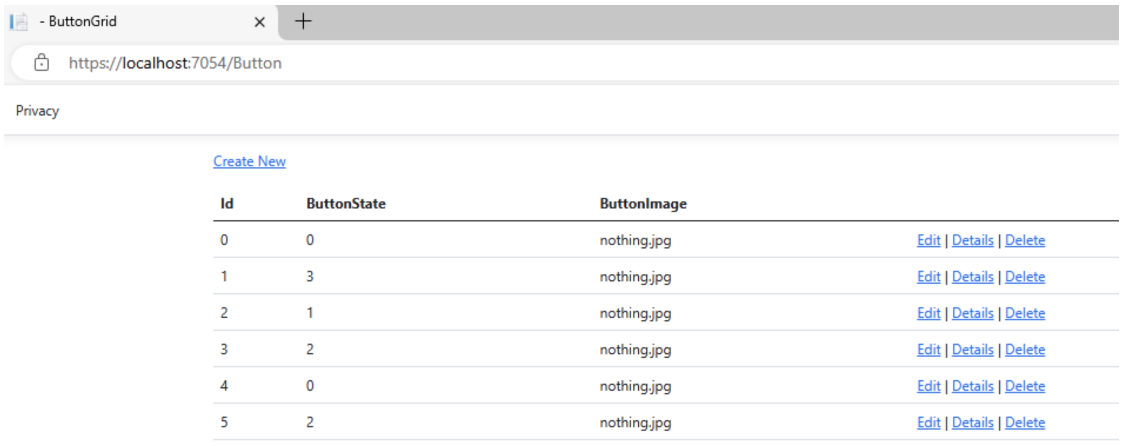
View name: Index
Template: List
Model class: ButtonModel (ButtonGrid.Models)

Options
- [✓] Create as a partial view
- [✓] Reference script libraries
- [✓] Use a layout page

(Leave empty if it is set in a Razor _viewstart file)

Figure 54

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



- ButtonGrid

https://localhost:7054/Button

Privacy

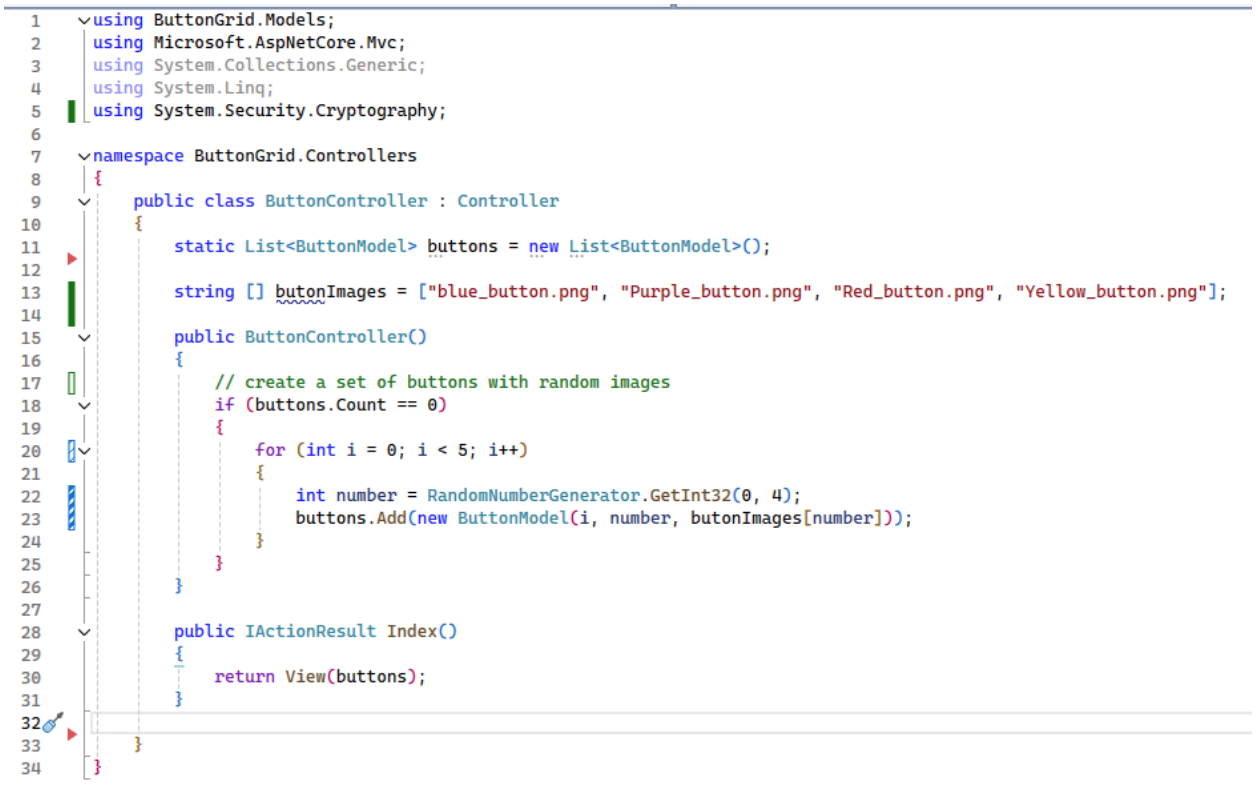
Create New

Id	ButtonState	ButtonImage	
0	0	nothing.jpg	Edit | Details | Delete
1	3	nothing.jpg	Edit | Details | Delete
2	1	nothing.jpg	Edit | Details | Delete
3	2	nothing.jpg	Edit | Details | Delete
4	0	nothing.jpg	Edit | Details | Delete
5	2	nothing.jpg	Edit | Details | Delete

Figure 55

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

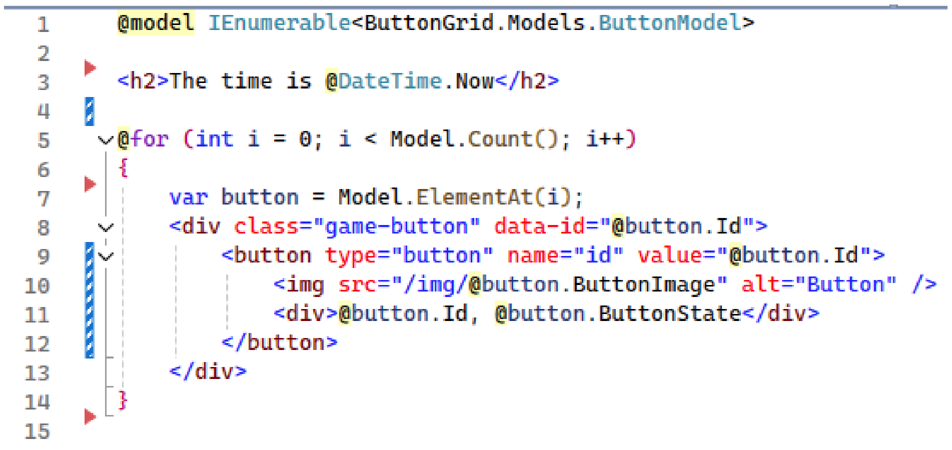
6. In the controller, generate a list of buttons with random state values. Set the image property to one of the strings in the buttonImages array.



```
using ButtonGrid.Models;
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using System.Linq;
using System.Security.Cryptography;

namespace ButtonGrid.Controllers
{
    public class ButtonController : Controller
    {
        static List<ButtonModel> buttons = new List<ButtonModel>();

        string [] butonImages = ["blue_button.png", "Purple_button.png", "Red_button.png", "Yellow_button.png"];

        public ButtonController()
        {
            // create a set of buttons with random images
            if (buttons.Count == 0)
            {
                for (int i = 0; i < 5; i++)
                {
                    int number = RandomNumberGenerator.GetInt32(0, 4);
                    buttons.Add(new ButtonModel(i, number, butonImages[number]));
                }
            }
        }

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

    }
}
```

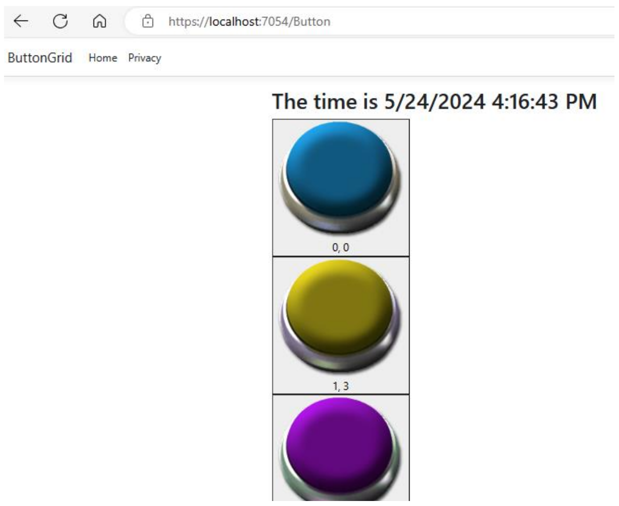
Figure 56

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.



```
@model IEnumerable<ButtonGrid.Models.ButtonModel>

<h2>The time is @DateTime.Now</h2>

@for (int i = 0; i < Model.Count(); i++)
{
    var button = Model.ElementAt(i);
    <div class="game-button" data-id="@button.Id">
        <button type="button" name="id" value="@button.Id">
            <img src="/img/@button.ButtonImage" alt="Button" />
            <div>@button.Id, @button.ButtonState</div>
        </button>
    </div>
}
```

Figure 57



The time is 5/24/2024 4:16:43 PM

Figure 58

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

8. Create a new css file in the css folder. Name it buttons.css

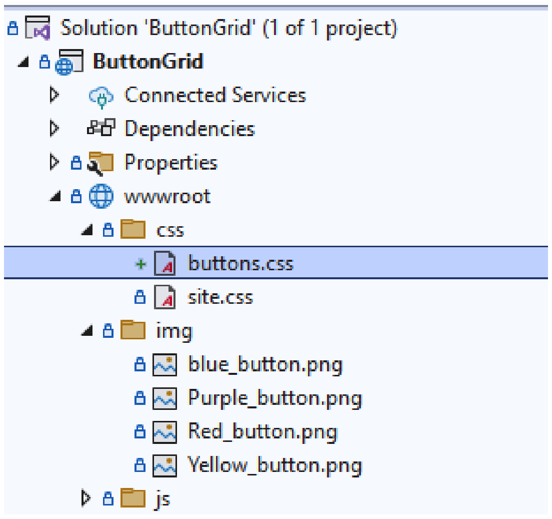
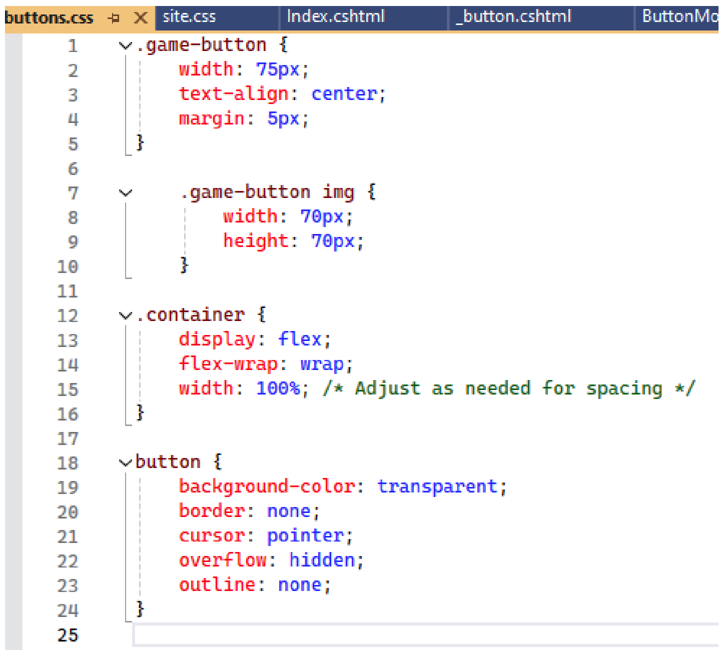


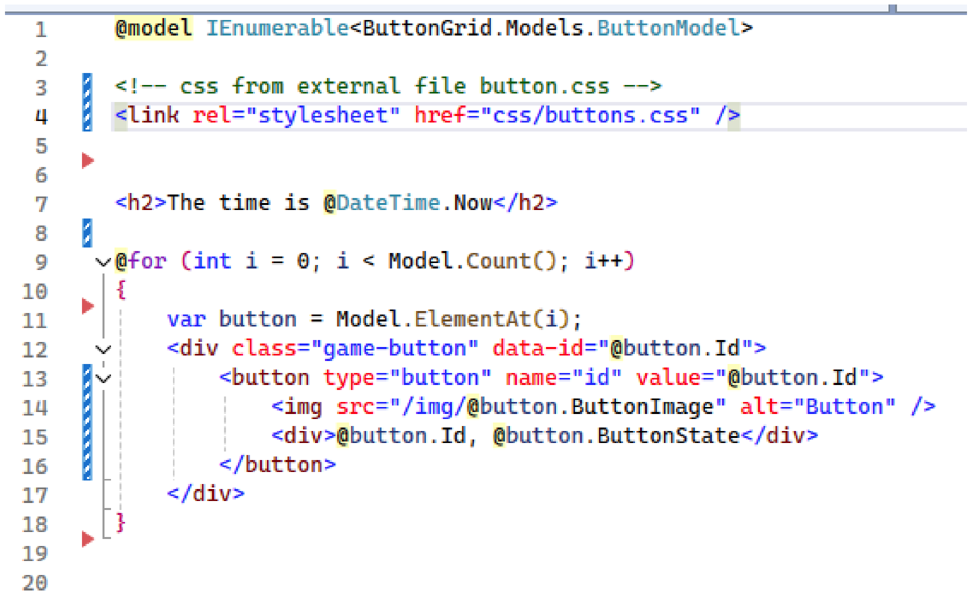
Figure 59

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



```css
.game-button {
    width: 75px;
    text-align: center;
    margin: 5px;
}

    .game-button img {
        width: 70px;
        height: 70px;
    }

.container {
    display: flex;
    flex-wrap: wrap;
    width: 100%; /* Adjust as needed for spacing */
}

button {
    background-color: transparent;
    border: none;
    cursor: pointer;
    overflow: hidden;
    outline: none;
}
```

Figure 60

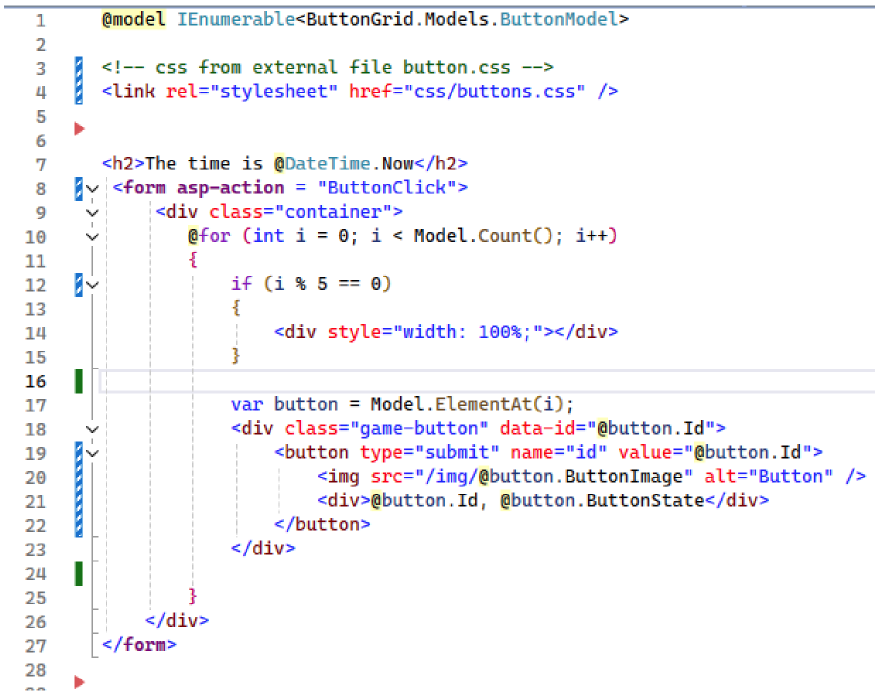


```cshtml
@model IEnumerable<ButtonGrid.Models.ButtonModel>

<!-- css from external file button.css -->
<link rel="stylesheet" href="css/buttons.css" />


<h2>The time is @DateTime.Now</h2>

@for (int i = 0; i < Model.Count(); i++)
{
    var button = Model.ElementAt(i);
    <div class="game-button" data-id="@button.Id">
        <button type="button" name="id" value="@button.Id">
            <img src="/img/@button.ButtonImage" alt="Button" />
            <div>@button.Id, @button.ButtonState</div>
        </button>
    </div>
}
```

Figure 61 CSS and Razor code to generate a 5 x 5 grid of buttons with the button properties printed below.

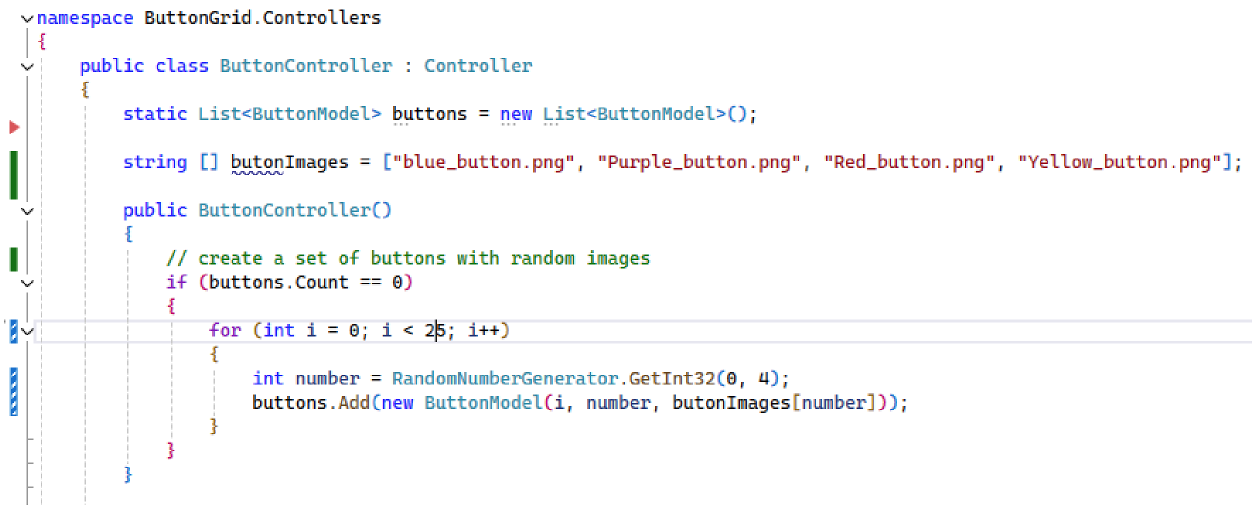
- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

10. Turn this into a square gameboard.



```
@model IEnumerable<ButtonGrid.Models.ButtonModel>

<!-- css from external file button.css -->
<link rel="stylesheet" href="css/buttons.css" />


<h2>The time is @DateTime.Now</h2>
<form asp-action = "ButtonClick">
    <div class="container">
        @for (int i = 0; i < Model.Count(); i++)
        {
            if (i % 5 == 0)
            {
                <div style="width: 100%;"></div>
            }

            var button = Model.ElementAt(i);
            <div class="game-button" data-id="@button.Id">
                <button type="submit" name="id" value="@button.Id">
                    <img src="/img/@button.ButtonImage" alt="Button" />
                    <div>@button.Id, @button.ButtonState</div>
                </button>
            </div>

        }
    </div>
</form>

```

Figure 62



```
namespace ButtonGrid.Controllers
{
    public class ButtonController : Controller
    {
        static List<ButtonModel> buttons = new List<ButtonModel>();

        string [] butonImages = ["blue_button.png", "Purple_button.png", "Red_button.png", "Yellow_button.png"];

        public ButtonController()
        {
            // create a set of buttons with random images
            if (buttons.Count == 0)
            {
                for (int i = 0; i < 25; i++)
                {
                    int number = RandomNumberGenerator.GetInt32(0, 4);
                    buttons.Add(new ButtonModel(i, number, butonImages[number]));
                }
            }
        }
```

Figure 63

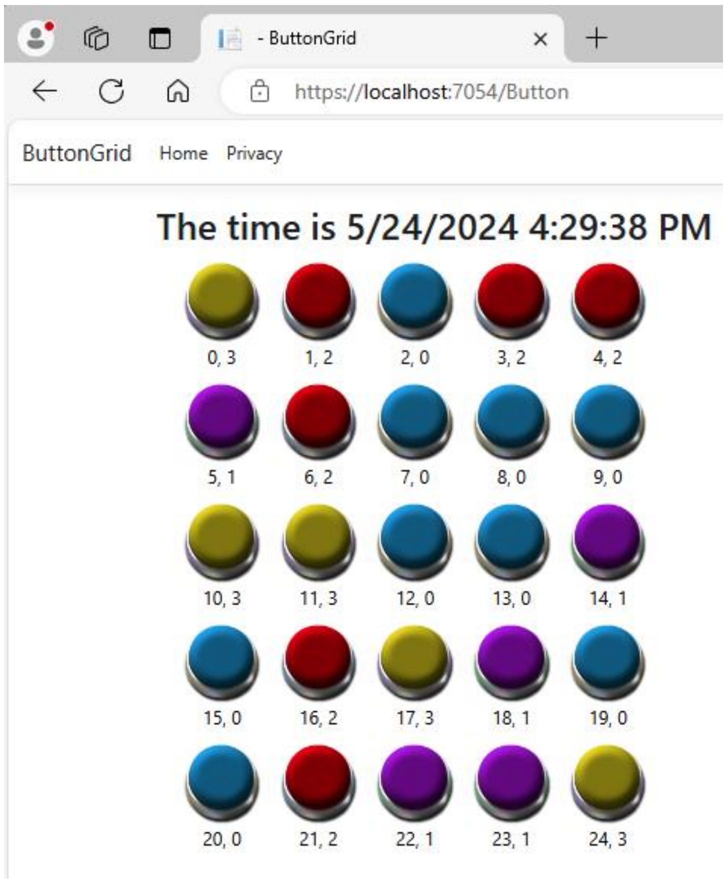
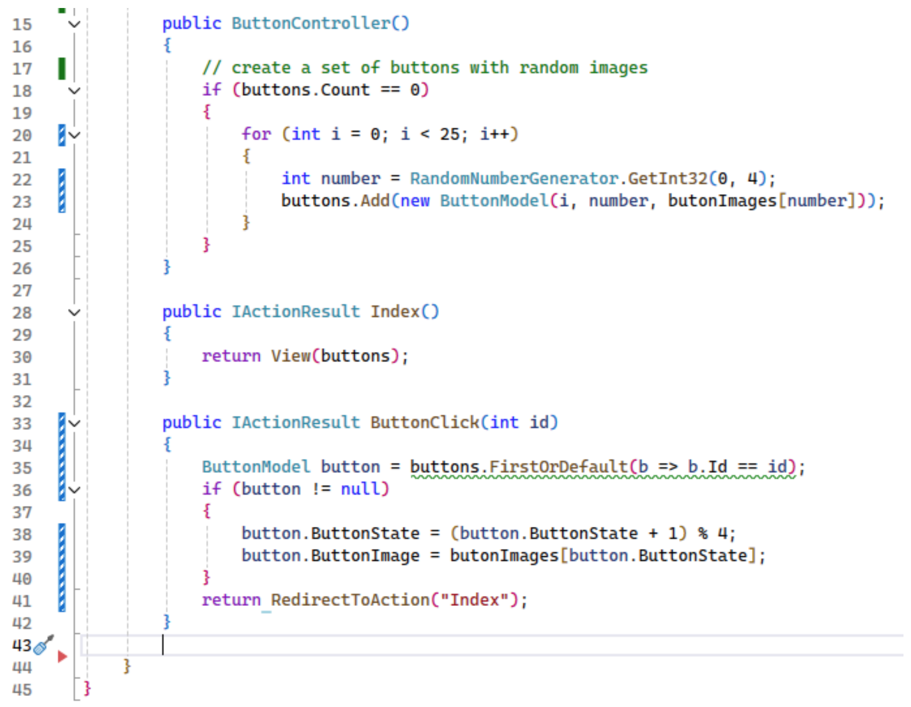


Figure 64

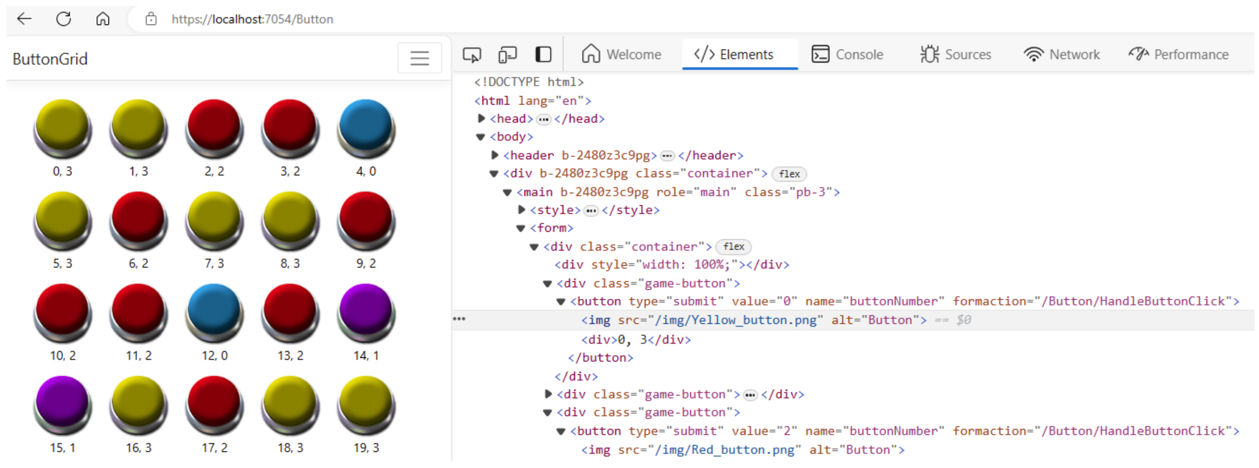
11. The form action is invalid. Let's add a new method to update the board.



```
        public ButtonController()
        {
            // create a set of buttons with random images
            if (buttons.Count == 0)
            {
                for (int i = 0; i < 25; i++)
                {
                    int number = RandomNumberGenerator.GetInt32(0, 4);
                    buttons.Add(new ButtonModel(i, number, butonImages[number]));
                }
            }
        }

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

        public IActionResult ButtonClick(int id)
        {
            ButtonModel button = buttons.FirstOrDefault(b => b.Id == id);
            if (button != null)
            {
                button.ButtonState = (button.ButtonState + 1) % 4;
                button.ButtonImage = butonImages[button.ButtonState];
            }
            return RedirectToAction("Index");
        }

    }
}
```

Figure 65

12. Run the program and inspect the HTML code that is produced.



```
<!DOCTYPE html>
<html lang="en">
 ▶<head> ••• </head>
 ▼<body>
   ▶<header b-2480z3c9pg> ••• </header>
   ▼<div b-2480z3c9pg class="container"> flex
     ▼<main b-2480z3c9pg role="main" class="pb-3">
       ▶<style> ••• </style>
       ▼<form>
         ▼<div class="container"> flex
             <div style="width: 100%;"></div>
           ▼<div class="game-button">
             ▼<button type="submit" value="0" name="buttonNumber" formaction="/Button/HandleButtonClick">
                 <img src="/img/Yellow_button.png" alt="Button"> == $0
                 <div>0, 3</div>
               </button>
             </div>
           ▶<div class="game-button"> ••• </div>
           ▼<div class="game-button">
             ▼<button type="submit" value="2" name="buttonNumber" formaction="/Button/HandleButtonClick">
                 <img src="/img/Red_button.png" alt="Button">
```

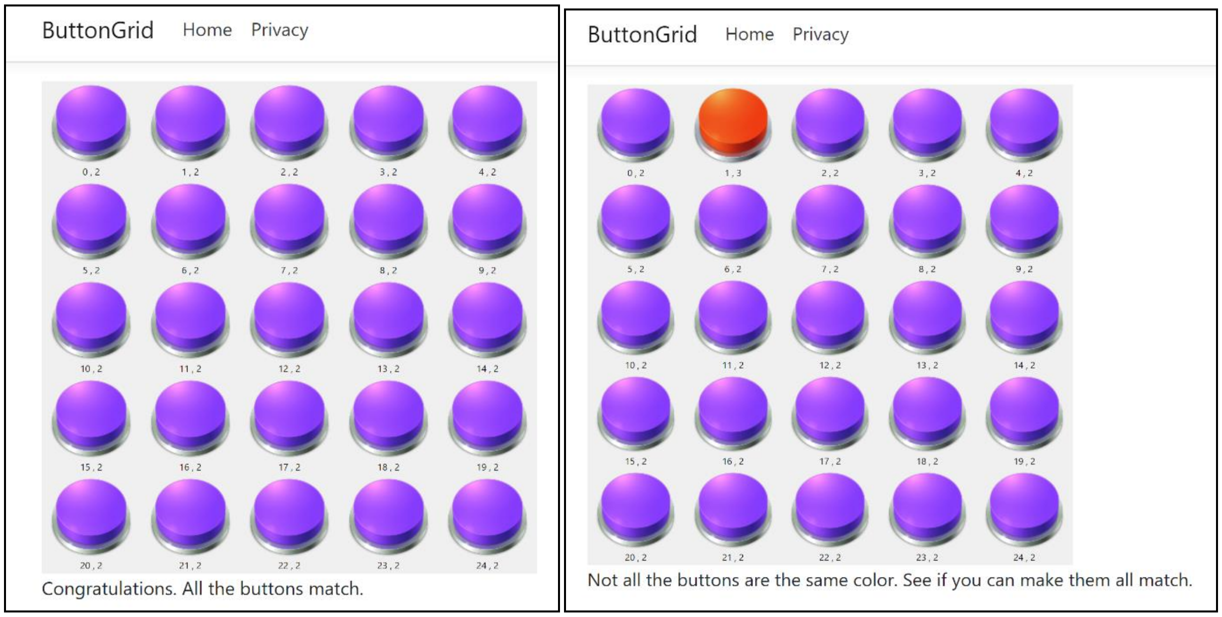
Figure 66

13. 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.
14. Test the app. You should be able to increment each button state with a single mouse click.

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

Programming Challenge

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



Congratulations. All the buttons match.

Not all the buttons are the same color. See if you can make them all match.

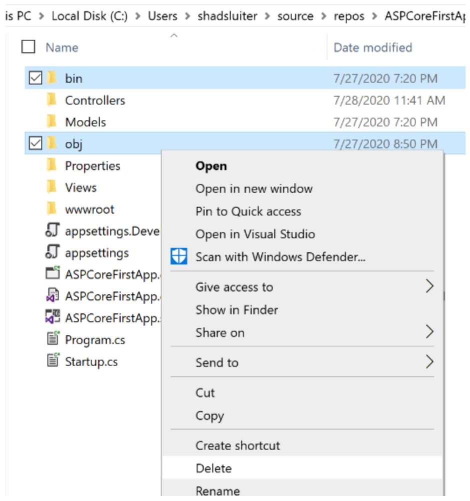
Figure 67

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

GRAND CANYON UNIVERSITY™

Conclusions:

What you learned



In this lesson, you focused on three essential aspects of modern software development: Dependency Injection, Data Validation, and Dynamic Object Management through a GUI. Below is a summary of the main points and specific examples that align with the stated learning objectives:

1. **Dependency Injection (DI)**
    - **Concept:** DI is a design pattern used to implement IoC (Inversion of Control) to decouple classes from their dependencies.
    - **Example:** In the Register and Login app, we replaced direct instantiation of the **UserCollection** and **UserDAO** classes within the **UserController** with interface-based injection (**IUserManager**). This decoupling allows for easier testing and greater flexibility, as different implementations can be injected without modifying the **UserController**.
    - **Implementation Steps:**
        1. Commented out direct instantiations of **UserCollection** and **UserDAO**.
        2. Introduced an interface variable (**IUserManager**) and injected it through the **UserController** constructor.
        3. Configured the DI container in **Program.cs** using the **AddSingleton** method to manage the lifecycle of dependencies.
2. **Data Validation**
    - **Concept:** Data validation ensures that the input data conforms to expected formats and constraints, enhancing data integrity and user experience.
    - **Example:** Implemented data validation in a model class for an appointment scheduling form using ASP.NET MVC. Validation rules included required fields, specific formats, and value ranges for properties such as **patientName**, **dateTime**, **PatientNetWorth**, **DoctorName**, and **PainLevel**.
    - **Implementation Steps:**
        1. Created a new MVC project and defined a **AppointmentModel** class with various properties.
        2. Added data validation annotations to the model properties.
        3. Created a controller and views to handle form input and display validation messages.
        4. Tested the form by submitting various data inputs to verify the validation rules.
3. **Dynamic Button Grid (GUI)**
    - **Concept:** Dynamic object management involves creating and manipulating UI components programmatically to enhance interactivity and user experience.
    - **Example:** Developed a dynamic button grid in a .NET MVC application to represent a game board. Each button in the grid can be toggled on or off, demonstrating dynamic object management.
    - **Implementation Steps:**

1. Created a new MVC project and defined a **ButtonModel** class.
2. Implemented a controller to manage the button grid state and generate the button list.
3. Created views to display the button grid and handle button click events.
4. Applied CSS to style the grid and tested the functionality by toggling button states.

Check for Understanding

These questions are not grade but will give you a picture of upcoming assessments.

1. What is an interface in object-oriented programming often referred to as?
A. A guideline
B. A contract
C. A structure
D. A model

2. Why were interfaces used in the Register and Login app?
A. To simplify the code
B. To ensure method names for the classes are identical, making them interchangeable
C. To improve performance
D. To reduce memory usage

3. What is the main purpose of Dependency Injection in software development?
A. To increase the complexity of code
B. To tightly couple classes and their dependencies
C. To decouple classes from their dependencies
D. To make code harder to read

4. What is a major issue with tightly coupled classes?
A. They are easy to test in isolation
B. They provide better performance
C. They are difficult to switch out with different implementations
D. They reduce the size of the codebase

5. Which method is used in Program.cs to associate the IUserManager interface with a specific implementation?
A. AddTransient
B. AddScoped
C. AddSingleton
D. AddInstance

6. What is the first step in applying Dependency Injection to the UserController?
A. Inject dependencies through its methods
B. Comment out direct instantiations of UserCollection and UserDAO
C. Add a constructor with a parameter for each dependency
D. Add public properties for each dependency

7. What does the AddSingleton<IUserManager, UserDAO>() method do?
  A. Registers a transient service
  B. Registers a scoped service
  C. Registers a singleton service
  D. Registers a temporary service

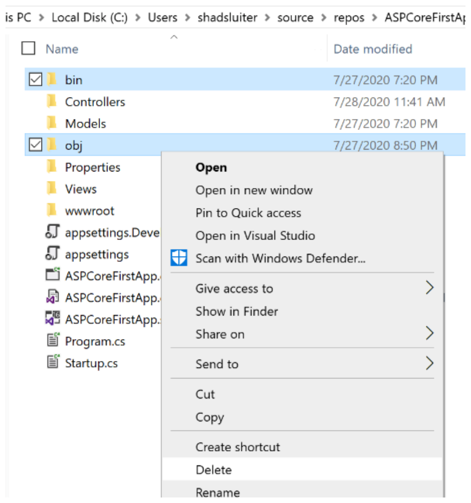
8. Why might you choose AddScoped for a service in ASP.NET Core?
  A. To create a service once per application lifetime
  B. To create a service each time it is requested
  C. To create a service once per client request
  D. To create a service that is shared across multiple applications

9. What is the advantage of using interfaces with Dependency Injection?
  A. It reduces the number of classes needed
  B. It makes the application slower
  C. It allows different implementations to be swapped without modifying the controller
  D. It increases the memory usage

10. What type of dependency injection registration is most suitable for lightweight, stateless services?
   A. AddSingleton
   B. AddScoped
   C. AddTransient
   D. AddPerCall

Deliverables:

1. This activity has multiple parts. Complete all parts before submitting.
2. Create 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. Turn the Word document into a PDF.
5. 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.
6. Attach the PDF separately from the zip file. Multiple files can be uploaded with an assignment.



<u>Check for Understanding Answers:</u>

1. B
2. B
3. C
4. C
5. C
6. B
7. C
8. C
9. C
10. C