Cihan University Sulaimaniya
Computer Science Department

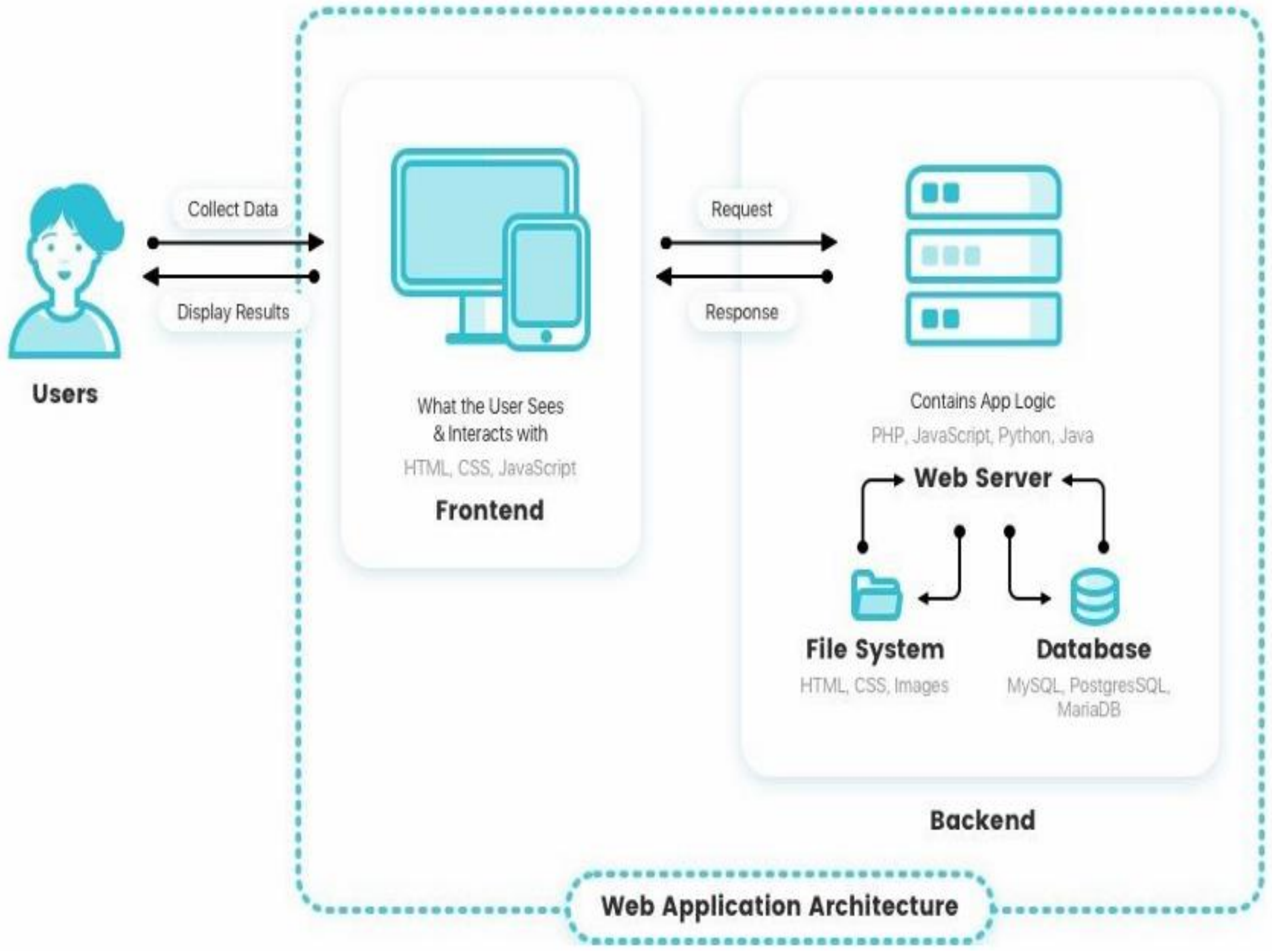
# WEB PROGRAMMING

3rd Stage
Lecturer: Asan Baker

## Introduction to Web Programming

- **Definition**: Web programming involves the development of applications and services that run over the web, typically using client-server architecture.
- **Key Technologies**: HTML, CSS, JavaScript, server-side languages, and databases.
- **Applications**: Websites, web applications, e-commerce platforms, social media, etc.

## Web Programming



## Importance of Web Programming

- **Global Connectivity**: Powers the internet, enabling users to access content from anywhere.
- **User Experience**: Helps create interactive, responsive, and user-friendly interfaces.
- **Business Growth**: Supports digital transformation, e-commerce, and online services.
- **Automation & Integration**: Facilitates process automation and seamless integration with other systems.

## Introduction to Front-End Web Programming

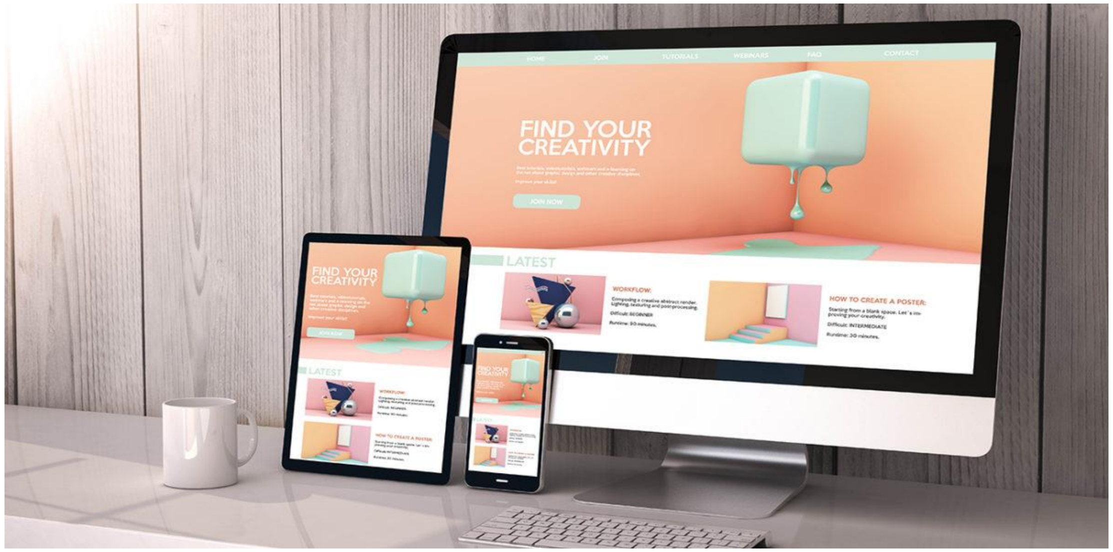
- **Definition**: Front-end web programming involves developing the part of a website or web application that users interact with directly.
- **Role**: Focuses on creating an engaging user experience by combining design with programming.
- **Core Technologies**: HTML, CSS, and JavaScript.

## Components of Front-End Development (Core Technologies of Front-End Development)

- **HTML (Hypertext Markup Language)**: Provides the structure of the webpage.
- **CSS (Cascading Style Sheets)**: Handles the presentation and layout of the webpage.
- **JavaScript**: Adds interactivity and dynamic behavior to web pages.

## Importance of Front-End Development

- **User Engagement**: First point of interaction between users and a website.
- **Accessibility**: Ensures content is accessible to all users, including those with disabilities.
- **Responsiveness**: Adapts web content to various devices (mobile, tablet, desktop).



- **User Interface (UI)**: Responsible for the look and feel of websites.
- **User Experience (UX)**: Ensures seamless interaction between users and web applications.

- **First Impressions**: A well-designed front-end determines the user’s first impression of a website or app.

## Static Pages in Front-End Development

- **Definition**: A static web page displays fixed content, and the same information is shown to all users every time they visit. i.e. Web pages delivered to the user's browser exactly as stored, without any server-side processing.

- **Characteristics**:
  - No dynamic content.
  - No interaction with a server or database.
  - Usually fast to load due to minimal processing.

## Pros and Cons of Static Pages

- **Pros**:
  - Fast loading speed.
  - Easy to develop and maintain.
  - Low cost to host.
- **Cons**:
  - No real-time data updates.
  - Requires manual updating for changes.
  - Limited user interaction.

## Limitations of Static Pages

- **Scalability**: Difficult to manage large websites with numerous pages.
- **Interactivity**: Limited functionality; cannot handle user input dynamically.
- **Maintenance**: Requires manual updates for content changes.

## When to Use Static Pages

- **Use Cases**:
  - Personal portfolios or blogs.
  - Small business websites with minimal content updates.
  - Landing pages for marketing campaigns.
- **Suitability**: Ideal for websites where content is not frequently updated.

# Static Pages vs. Dynamic Webpages

| Feature | Static Pages | Dynamic Websites |
|---|---|---|
| **Content** | Fixed content, the same for all users | Content changes based on user interaction or database |
| **Technology** | HTML, CSS only | HTML, CSS, JavaScript + server-side languages (PHP, ASP.NET, etc.) |
| **Interactivity** | Minimal or none | High, with real-time interaction (forms, data retrieval) |
| **Data Handling** | No interaction with databases | Pulls data from databases or APIs based on user input |
| **Performance** | Fast to load since there is no server processing | May have slower load times due to server-side processing |
| **Maintenance** | Requires manual updates for content changes | Content can be updated dynamically, often with CMS |
| **Use Cases** | Simple sites: personal blogs, landing pages | Complex sites: social media, e-commerce, dashboard |

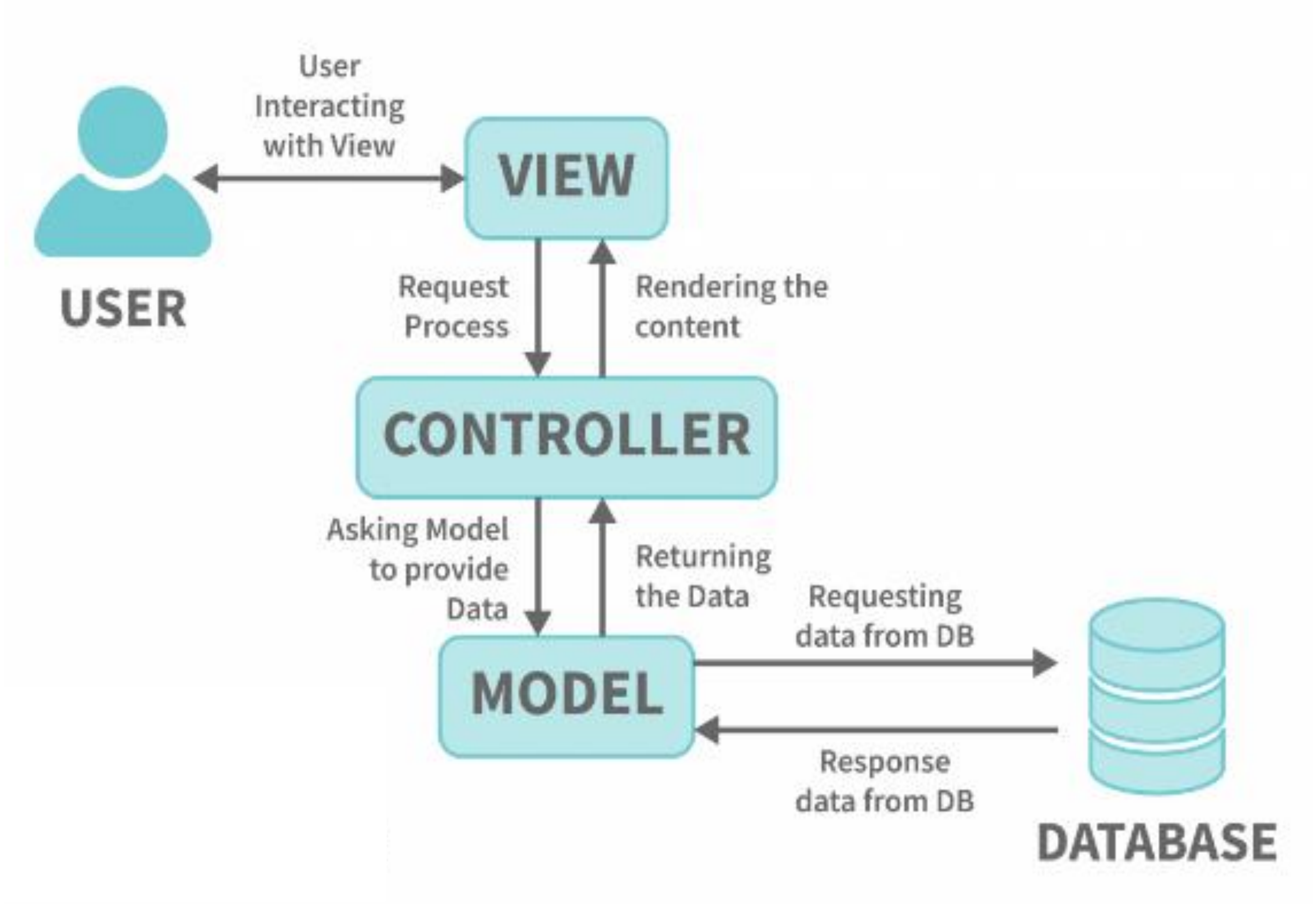
## Introduction to ASP.NET Core MVC

- **Definition**: ASP.NET Core MVC is a framework for building web applications using the Model-View-Controller (MVC) architectural pattern.
- **Key Features**:
  - Separation of concerns (model, view, controller).
  - Built on top of ASP.NET Core, supporting cross-platform development.
  - Built-in dependency injection, security features, and middleware pipeline.

## MVC Architecture Overview

- **Model**: Represents the application's data and business logic.
- **View**: The user interface that displays the model's data.
- **Controller**: Handles user input and interacts with the model and view to fulfill the user's request.

# basic MVC diagram



## Example: Basic MVC Project Structure

- **Project Structure**:
  - **Models**: Contains data classes.
  - **Views**: Contains UI elements (Razor views).
  - **Controllers**: Contains logic for handling user requests.

## Basic structure

```
|MyMvcApp/
|
|---Models/
|     └─── Product.cs
|
|---Views/
|        └─── Home/
|              └─── Index.cshtml
|
|---Controllers/
|            └─── HomeController.cs
|
└─── Startup.cs
```

## Example: Controller in ASP.NET Core MVC

```
using Microsoft.AspNetCore.Mvc;

public class HomeController : Controller
{
    public IActionResult Index()
    {
        return View();
    }
}
```

**Explanation**:

- The HomeController class extends Controller.
- The Index method returns a view (HTML page) when a user navigates to the Home page.

## Example: Razor View

- **What is Razor?**: Razor is a syntax for combining C# code with HTML in ASP.NET Core MVC.

```
<h1>Welcome to My MVC App</h1>
<p>@DateTime.Now</p>
```

## Example: Model in ASP.NET Core MVC

```
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
}
```

**Explanation**:

This Product class represents the model containing data about a product.

## Passing Data from Controller to View

```
public IActionResult ProductList()
{
    var products = new List<Product>
    {
        new Product { Id = 1, Name = "Laptop", Price = 799.99M },
        new Product { Id = 2, Name = "Phone", Price = 499.99M }
    };
    return View(products);
}
```

**Explanation**:

The ProductList action method creates a list of products and passes it to the view.

## Example: Displaying Model Data in the View

```
@model List<Product>

<h1>Product List</h1>

<ul>
@foreach (var product in Model)
{
    <li>@product.Name - $@product.Price</li>
}
</ul>
```

**Explanation**:

- The @model directive specifies the data type passed from the controller.
- A loop is used to display each product's name and price in the view.

## Introduction to HTML in ASP.NET Core MVC

- **HTML**: The backbone of web pages, provides structure.
- **ASP.NET Core MVC**: Uses Razor Pages to embed dynamic content into HTML.
- **Key Point**: HTML forms the foundation of MVC Views.

- In an ASP.NET Core MVC project, you can use **HTML** and **CSS** to build the user interface while taking advantage of the server-side capabilities of the MVC framework.

## Structure of an HTML Document

- **Components** of an HTML Document:
  - **<!DOCTYPE html>**: Declaration for HTML5.
  - **<html>**: Root element.
  - **<head>**: Meta-information (title, CSS links, etc.).
  - **<body>**: Contains the visible content.

## Example of a Basic HTML Structure:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>My MVC Webpage</title>
  </head>
  <body>
    <h1>Welcome to My MVC Webpage</h1>
    <p>This content is served through MVC.</p>
  </body>
</html>
```

## HTML Elements in ASP.NET Core Views

- **HTML Elements in Razor Pages**:
  - **Headings (<h1>, <h2>)**
  - **Paragraphs (<p>)**
  - **Links (<a href="url">Link</a>)**
  - **Forms (<form>)** for data submission.

## **Example**: Displaying Data in Razor View (HTML with C# Code):

- <h1>Hello, @Model.UserName!</h1>
- <p>Your role is: @Model.UserRole</p>

## HTML Fundamentals in ASP.NET Core MVC

- HTML is used to structure the content of a web page. In an ASP.NET Core MVC application, HTML tags are written in **Razor Views** (.cshtml files) along with server-side Razor syntax.

## Example: HTML in ASP.NET Core MVC

Consider the following Razor view that displays a form to collect user data:

```
@model UserModel

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>User Form</title>
</head>
<body>

    <h1>User Registration</h1>

    <form asp-action="Register" method="post">
        <label for="name">Name:</label>
        <input type="text" id="name" name="Name" value="@Model.Name" required><br>

        <label for="email">Email:</label>
        <input type="email" id="email" name="Email" value="@Model.Email" required><br>

        <button type="submit">Submit</button>
    </form>

</body>
</html>
```

## Explanation of the example:

- The HTML tags (<form>, <input>, <button>) are used to create the structure of the form.
- The @Model directive allows server-side data to be injected into the HTML.
- For example, @Model.Name and @Model.Email display the values of the Name and Email properties of the UserModel class.

## Server-side action in MVC

- The form posts the data to an action method (asp-action="Register"), which is defined in the controller. Here's an example of the corresponding controller action:

```
[HttpPost]
public IActionResult Register(UserModel model)
{
    if (ModelState.IsValid)
    {
        // Save user data to database (example)
        return RedirectToAction("Success");
    }
    return View(model);
}
```

## CSS Fundamentals in ASP.NET Core MVC

- CSS is used to style the HTML content in Razor views. In ASP.NET Core MVC, you can use internal, inline, or external CSS files to style your web pages.
- Example: CSS in ASP.NET Core MVC Let’s extend the previous example by adding some CSS for styling.
- **Inline CSS**:
- You can directly add CSS styles to individual elements in the HTML.

```
<h1 style="color: blue; font-family: Arial;">User Registration</h1>
```

- **Internal CSS:** You can add CSS within the <style> tag in the <head> section of the Razor view.

```
<head>
    <style>
        body {
            font-family: 'Arial', sans-serif;
        }
        h1 {
            color: blue;
        }
        label {
            font-weight: bold;
        }
        input {
            margin-bottom: 10px;
        }
    </style>
</head>
```

- **External CSS:** In real-world applications, it is common to separate CSS into a dedicated file. In ASP.NET Core MVC, you can add your CSS file in the wwwroot folder and reference it in the view.Create a CSS file, e.g., site.css, in wwwroot/css/.

```
/* site.css */
body {
    font-family: 'Arial', sans-serif;
}

h1 {
    color: blue;
}

label {
    font-weight: bold;
}

input {
    margin-bottom: 10px;
}
```

- Reference the CSS file in your Razor view:

```
<head>
    <link rel="stylesheet" href="~/css/site.css">
</head>
```

# Combining HTML and CSS in ASP.NET Core MVC

- The complete Razor view with external CSS and HTML would look like this:

```
@model UserModel

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>User Form</title>
    <link rel="stylesheet" href="~/css/site.css">
</head>
<body>

    <h1>User Registration</h1>

    <form asp-action="Register" method="post">
        <label for="name">Name:</label>
        <input type="text" id="name" name="Name" value="@Model.Name" required><br>

        <label for="email">Email:</label>
        <input type="email" id="email" name="Email" value="@Model.Email" required><br>

        <button type="submit">Submit</button>
    </form>

</body>
</html>
```

## ASP.NET Core Tag Helpers

- ASP.NET Core MVC also provides Tag Helpers, which allow you to work with HTML elements more easily by extending them with server-side features. An example is the asp-action and asp-for attributes.

```
<form asp-action="Register" method="post">
    <div>
        <label asp-for="Name"></label>
        <input asp-for="Name" />
    </div>

    <div>
        <label asp-for="Email"></label>
        <input asp-for="Email" type="email" />
    </div>

    <button type="submit">Submit</button>
</form>
```

## Explanation for example:

- Explanation:asp-for="Name" automatically generates a label and input field for the Name property in the UserModel.
- asp-action="Register" links the form submission to the Register action method on the server.

- This combination of HTML, CSS, and Tag Helpers makes it easy to create well-structured and styled web pages in an ASP.NET Core MVC application.

## Responsive Design Basics

- **Definition**: Making websites adaptable to different screen sizes (mobile, tablet, desktop).
- **Techniques**:
  - **Fluid layouts**: Use percentage-based widths instead of fixed widths.
  - **Media Queries**: Apply CSS based on device screen size.
  - **Flexible images**: Ensure images scale with the screen.

## Responsive Design using CSS

- CSS allows you to make your pages responsive so they look good on all devices. You can use media queries for this.
- Example: Responsive CSS with Media Queries

```
/* Basic form styling */
.form-control {
    border: 1px solid #ccc;
    padding: 10px;
    width: 100%;
    max-width: 300px;
}

.submit-button {
    background-color: green;
    color: white;
    padding: 10px 20px;
    border: none;
}

/* Responsive styling for mobile devices */
@media (max-width: 600px) {
    .form-control {
        width: 100%;
    }

    .submit-button {
        width: 100%;
    }
}
```

## Key Concepts of Responsive Design

1. Fluid Layouts (Percentage-based widths):Instead of using fixed pixel values, responsive designs use percentages or other flexible units (like em, rem, or vw/vh for viewport units) for element widths. This ensures the layout scales fluidly on different screen sizes.Example:

```
.container {
    width: 80%;  /* 80% of the viewport width */
    margin: 0 auto;  /* Center the container */
}
```

2. Media Queries: Media queries allow you to apply CSS rules conditionally, based on the width, height, orientation, or resolution of the device.The most common use is to change the layout when the screen width reaches certain "breakpoints," like for mobile devices

Example:

```
/* Styles for desktop devices */
body {
  font-size: 16px;
}

/* Styles for screens narrower than 768px (tablets and mobile) */
@media (max-width: 768px) {
  body {
    font-size: 14px;
  }
}

/* Styles for screens narrower than 480px (mobile phones) */
@media (max-width: 480px) {
  body {
    font-size: 12px;
  }
}
```

In this example, the font size changes as the screen size decreases, making the content easier to read on smaller devices.

3. Flexible Images and Media: Images and other media (like videos) should scale according to the size of their containing elements to avoid overflowing or being too large on small screens.

Example:

```
img {
    max-width: 100%;  /* Image never exceeds 100% of its container's width */
    height: auto;     /* Keep the image's aspect ratio */
}
```

4. Viewport Meta Tag: This tag ensures that the web page scales correctly on mobile devices. Without it, mobile browsers typically render the page at a fixed width (e.g., 980px) and scale it down, which can make it hard to read or interact with.

- Example:

```
<meta name="viewport" content="width=device-width,
initial-scale=1.0">
```

This tag sets the width of the viewport to the device’s width and prevents initial zooming.

5. Grid Layouts: Responsive web design often relies on grid systems (e.g., CSS Grid or Flexbox) to create flexible and adaptable layouts. Grid systems allow for complex layouts that can easily shift or resize based on screen size.

Example (CSS Grid):

```
.grid-container {
    display: grid;
    grid-template-columns: repeat(3, 1fr);  /* 3 equal columns */
    gap: 10px;
}

@media (max-width: 768px) {
    .grid-container {
        grid-template-columns: repeat(2, 1fr);  /* 2 equal columns on tablets */
    }
}

@media (max-width: 480px) {
    .grid-container {
        grid-template-columns: 1fr;  /* 1 column on mobile phones */
    }
}
```

- 6. Responsive Typography: Use relative units like em or rem for font sizes so that they scale based on the parent or root element. This ensures that text is readable on any screen size.
- **Example:**

```
body {
    font-size: 16px;  /* base size */
}

h1 {
    font-size: 2rem;  /* 2 times the base size (32px) */
}

@media (max-width: 768px) {
    body {
        font-size: 14px;
    }
}
```

7. Responsive Navigation Menu A navigation menu should be easy to use on all screen sizes. For mobile, you might switch from a horizontal menu to a collapsible or hamburger menu.Example:

```
<nav>
    <ul class="menu">
        <li><a href="#">Home</a></li>
        <li><a href="#">About</a></li>
        <li><a href="#">Services</a></li>
        <li><a href="#">Contact</a></li>
    </ul>
</nav>

<style>
.menu {
    list-style: none;
    display: flex;
    justify-content: space-around;
    padding: 0;
}

.menu li {
    margin: 10px;
}

@media (max-width: 600px) {
    .menu {
        flex-direction: column;
    }
}
</style>
```

- In this example:On larger screens, the menu items are displayed horizontally using flexbox.On smaller screens
-  (max-width: 600px), the menu switches to a vertical layout.

## Example: A Basic Responsive Page

- Here’s a simple example of how these concepts work together in practice:

## html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Responsive Design Example</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <header>
        <h1>Responsive Design Basics</h1>
    </header>
    <main class="container">
        <section class="grid-container">
            <article class="box">Box 1</article>
            <article class="box">Box 2</article>
            <article class="box">Box 3</article>
        </section>
    </main>
</body>
</html>
```

# CSS (styles.css):

```css
/* Basic styles */
body {
    font-family: Arial, sans-serif;
    line-height: 1.6;
}

h1 {
    text-align: center;
}

.container {
    width: 80%;
    margin: 0 auto;
}

.grid-container {
    display: grid;
    grid-template-columns: repeat(3, 1fr);  /* 3 equal columns */
    gap: 20px;
}

.box {
    background-color: lightblue;
    padding: 20px;
    text-align: center;
    font-size: 1.5rem;
}
```

```css
/* Responsive changes for tablets (max-width: 768px) */
@media (max-width: 768px) {
    .grid-container {
        grid-template-columns: repeat(2, 1fr);  /* 2 columns */
    }
}


/* Responsive changes for mobile phones (max-width:
480px) */
@media (max-width: 480px) {
    .grid-container {
        grid-template-columns: 1fr;  /* 1 column */
    }


    .box {
        font-size: 1.2rem;  /* Adjust text size */
    }
}
```

## How It Works:

- Desktop: The grid has 3 columns with equal width.
- Tablets (max-width: 768px): The grid adjusts to 2 columns, providing a better fit for medium-sized screens.
- Mobile (max-width: 480px): The grid collapses to a single column, ensuring all content is readable without zooming or horizontal scrolling.

## Best Practices for Responsive Design

- Mobile-First Design: Start by designing for small screens and progressively enhance the layout for larger devices. This keeps the design lean and functional on mobile devices by default.
- Example:

```
/* Mobile-first styles */
.container {
    width: 100%;
}

@media (min-width: 768px) {
    /* Styles for larger screens (tablets and above) */
    .container {
        width: 80%;
    }
}
```

- Minimize Fixed Widths: Avoid using fixed pixel widths for containers or elements; instead, use percentages, em, or rem units to make elements more fluid.

- Use BreakpointsThoughtfully: Define breakpoints based on content, not just device sizes. Breakpoints should be introduced where the layout breaks, not just at standard resolutions.

- Real Devices: Ensure you test the design on multiple devices, including desktops, tablets, and smartphones, to see how the content adapts.
- Responsive Media (Images and Videos): Always ensure that media like images and videos are responsive and do not exceed their container's width.

## Integrating HTML and CSS in ASP.NET Core MVC

- **HTML as Razor Pages**:
  - Render dynamic content with @Model syntax.
- **CSS in ASP.NET Core**:
  - Use internal/external CSS or Bootstrap for consistent styling.

    (**Bootstrap**: A popular CSS framework integrated with ASP.NET Core for responsive design.)
- **Responsive Design**:
  - Implement media queries in CSS for adaptable layouts.

## Introduction to JavaScript

- **Definition**: JavaScript is a programming language that enables interactive web pages.
- **Role in Web Development**:
  - Client-side scripting.
  - Used for enhancing the user experience with dynamic content.

## **Example**: Simple JavaScript alert:

```
<script>
  alert('Hello, World!');
</script>
```

- **How JavaScript Works in ASP.NET Core MVC**:
- JavaScript is used to enhance the interactivity of MVC views.

## JavaScript in the Browser

- **Where JavaScript Runs**: JavaScript executes in the browser.
- **How to Include JavaScript**:
  - **Inline JavaScript**: Directly in the HTML <script> tag.
  - **External JavaScript**: Linked through an external file.
- **Example**:

```
<script src="scripts/app.js"></script>
```

## JavaScript Example in ASP.NET Core MVC

- **Example of Adding JavaScript in Razor View**:

```
<script>
document.getElementById("btnClick").addEventListener("click", function() {
    alert('Button clicked!');
  });
</script>
```

**Button in Razor View**:

```
<button id="btnClick">Click Me</button>
```

## Manipulating the DOM

- **DOM (Document Object Model)**: A tree structure that represents the HTML document.
- **JavaScript DOM Manipulation**:
  - JavaScript can dynamically change HTML elements, attributes, and styles.

- **Example**: Changing the content of a <div>:

```
<div id="content">Original Content</div>
<script>
  document.getElementById("content").innerHTML =
"Updated Content!";
</script>
```

## Common DOM Manipulation Methods

- **Selecting Elements**:
  - getElementById()
  - getElementsByClassName()
  - querySelector()
- **Example**:

```
<p id="text">Hello</p>
<script>
  document.getElementById("text").innerText =
"Hello, JavaScript!";
</script>
```

## DOM Manipulation Example in ASP.NET Core MVC

- **Example in Razor View**: Updating content dynamically.

```
<div id="message">Welcome, user!</div>
<script>
  document.getElementById("message").innerHTML = "Welcome to the ASP.NET MVC app!";
</script>
```

**Interactive form**: Modifying form elements based on user input.

## Basic Event Handling

- **Event Handling in JavaScript**: Detecting user interactions (clicks, keypresses, etc.).
- **Example**: Handling a button click event.

```
<button id="submitBtn">Submit</button>
<script>

document.getElementById("submitBtn").addEventListener("
click", function() {
    alert("Button clicked!");
  });
</script>
```

## Types of JavaScript Events

- **Common Events**:
  - **Click**: When an element is clicked.
  - **Keypress**: When a key is pressed.
  - **Load**: When the webpage is fully loaded.
- **Example**:

```
<button id="btn">Click Me</button>
<script>
  document.getElementById("btn").onclick = function() {
    alert("You clicked the button!");
  };
</script>
```

## Event Handling Example in ASP.NET Core MVC

- **Example in Razor View**: Submitting a form via button click:

```
<form>
  <input type="text" id="nameInput" placeholder="Enter your
name">
  <button id="submitBtn">Submit</button>
</form>

<script>

document.getElementById("submitBtn").addEventListener("click"
, function() {
    var name = document.getElementById("nameInput").value;
    alert("Submitted: " + name);
  });
</script>
```

## Integrating JavaScript with ASP.NET Core MVC

- **JavaScript in Razor Views**:
  - Use <script> tags to include both inline and external scripts.
- **Event Handling**:
  - Capture user interactions for dynamic updates without reloading the page.
- **DOM Manipulation**:
  - Modify HTML elements to improve user experience.

# MVC-based apps contain:

- Models: Classes that represent the data of the app. The model classes use validation logic to enforce business rules for that data. Typically, model objects retrieve and store model state in a database
- Views: Views are the components that display the app's user interface (UI). Generally, this UI displays the model data.
- Controllers: Classes that:
  - Handle browser requests.
  - Retrieve model data.
  - Call view templates that return a response.

## What are the advantages of MVC

- **Multiple view support**
- Due to the separation of the model from the view, the user interface can display multiple views of the same data at the same time.
- **Change Accommodation**
- User interfaces tend to change more frequently than business rules (different colors, fonts, screen layouts, and levels of support for new devices such as cell phones or PDAs) because the model does not depend on the views, adding new types of views to the system generally does not affect the model. As a result, the scope of change is confined to the view.
- **SoC – Separation of Concerns**
- Separation of Concerns is one of the core advantages of ASP.NET MVC. The MVC framework provides a clean separation of the UI, Business Logic, Model or Data.

- **More Control**
- The ASP.NET MVC framework provides more control over HTML, JavaScript, and CSS than the traditional Web Forms.
- **Testability**
- ASP.NET MVC framework provides better testability of the Web Application and good support for test driven development too.
- **Lightweight**
- ASP.NET MVC framework doesn’t use View State and thus reduces the bandwidth of the requests to an extent.
- **Full features of ASP.NET**
- One of the key advantages of using ASP.NET MVC is that it is built on top of the ASP.NET framework and hence most of the features of the ASP.NET like membership providers, roles, etc can still be used.

## Explain MVC application life cycle?

- Any web application has two main execution steps, first understanding the request and depending on the type of the request sending out an appropriate response. MVC application life cycle is not different it has two main phases, first creating the request object and second sending our response to the browser.

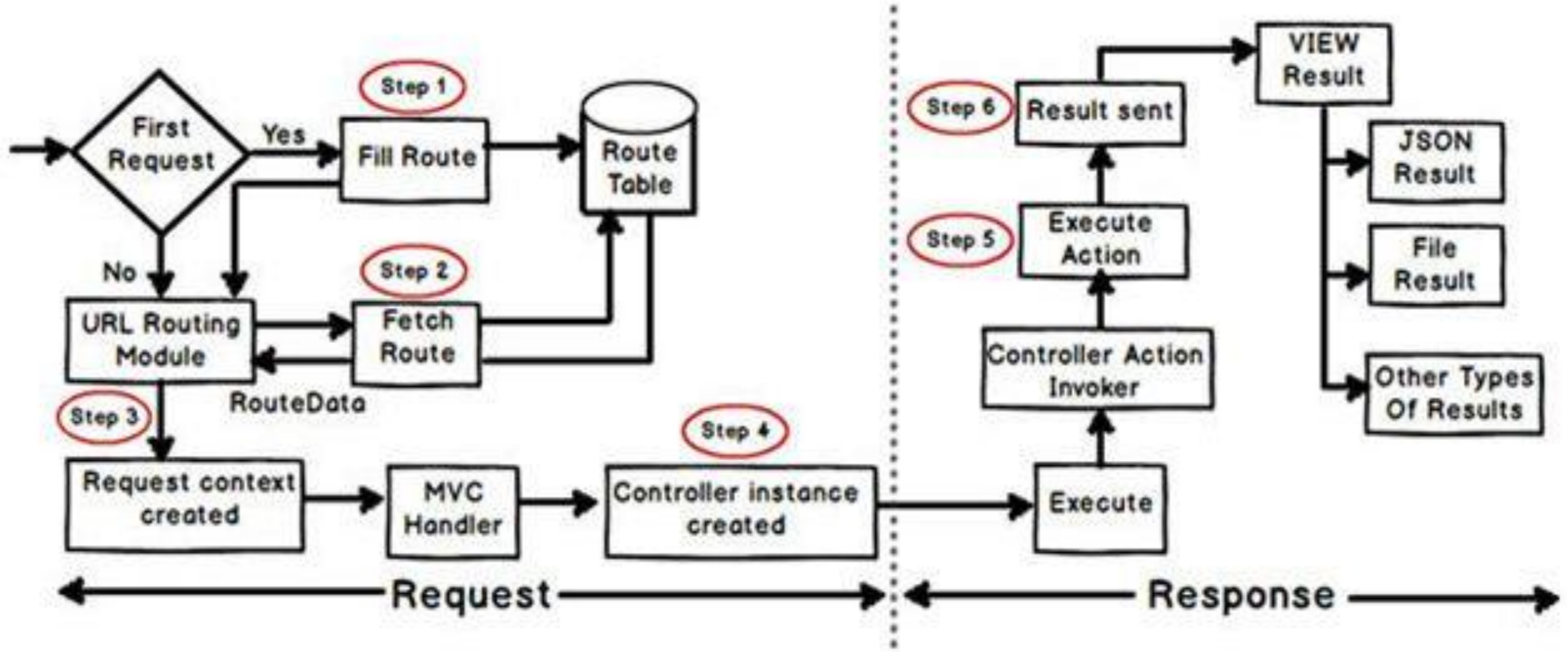
## Creating the request object,

- The request object creation has four major steps. The following is a detailed explanation of the same.
- **Step 1 - Fill route**
- MVC requests are mapped to route tables which in turn specify which controller and action to be invoked. So if the request is the first request the first thing is to fill the rout table with routes collection. This filling of the route table happens the global.asax file
- **Step 2 - Fetch route**
- Depending on the URL sent “UrlRoutingModule” searches the route table to create “RouteData” object which has the details of which controller and action to invoke.

- **Step 3 - Request context created**
- The “RouteData” object is used to create the “RequestContext” object.
- **Step 4 - Controller instance created**
- This request object is sent to “MvcHandler” instance to create the controller class instance. Once the controller class object is created it calls the “Execute” method of the controller class.

# Creating a Response object

- This phase has two steps executing the action and finally sending the response as a result to the view.



## List out different return types of a controller action method

- There are total of nine return types we can use to return results from the controller to view.
- The base type of all these result types is ActionResult.
- *ViewResult (View)*

  This return type is used to return a webpage from an action method.
- *PartialviewResult (Partialview)*

  This return type is used to send a part of a view that will be rendered in another view.
- *RedirectResult (Redirect)*

  This return type is used to redirect to any other controller and action method depending on the URL.

- *RedirectToRouteResult (RedirectToAction, RedirectToRoute)*

  This return type is used when we want to redirect to any other action method.
- *ContentResult (Content)*

  This return type is used to return HTTP content type like text/plain as the result of the action.
- *jsonResult (json)*

  This return type is used when we want to return a JSON message.
- *javascriptResult (javascript)*

  This return type is used to return JavaScript code that will run in the browser.
- *FileResult (File)*

  This return type is used to send binary output in response.
- *EmptyResult*

  This return type is used to return nothing (void) in the result.

## what is routing in MVC? What are the three segments for routing important?

- Routing is a mechanism to process the incoming URL that is more descriptive and gives the desired response. In this case, URL is not mapped to specific files or folder as was the case of earlier days web sites.
- There are two types of routing:
  - Convention-based routing - to define this type of routing, we call MapRoute method and set its unique name, URL pattern and specify some default values.

  - Attribute-based routing - to define this type of routing, we specify the Route attribute in the action method of the controller.

- In other ways let us say routing help you to define a URL structure and map the URL with controller. There are three segments for routing that are important,
  - ControllerName
  - ActionMethodName
  - Parammeter

## What is Partial View in MVC

- A partial view is a chunk of HTML that can be safely inserted into an existing DOM. Most commonly, partial views are used to componentize Razor views and make them easier to build and update. Partial views can also be returned directly from controller methods.

## Explain what is the difference between View and Partial View?

- **View**
- It contains the layout page.
- Before any view is rendered, viewstart page is rendered.
- A view might have markup tags like body, HTML, head, title, meta etc.
- The view is not lightweight as compare to Partial View.

- **Partial View**
- It does not contain the layout page.
- Partial view does not verify for a viewstart.cshtml.We cannot put common code for a partial view within the viewStart.cshtml.page.
- Partial view is designed specially to render within the view and just because of that it does not consist any mark up.
- We can pass a regular view to the RenderPartial method.

## What are the Main Razor Syntax Rules?

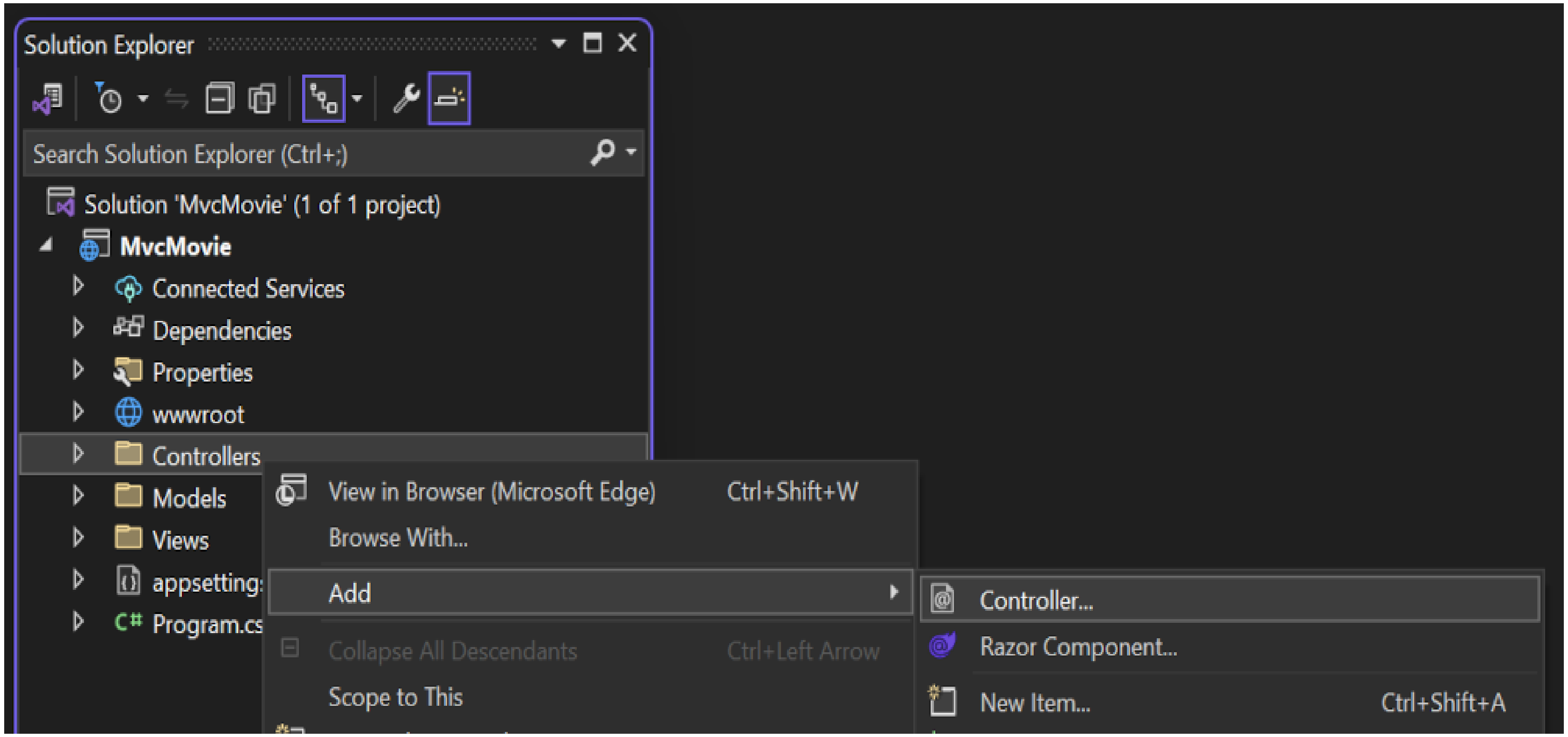
- Razor code blocks are enclosed in @{ ... }
- Inline expressions (variables and functions) start with @
- Code statements end with semicolon
- Variables are declared with the var keyword
- Strings are enclosed with quotation marks
- C# code is case sensitive
- C# files have the extension .cshtml

## Example

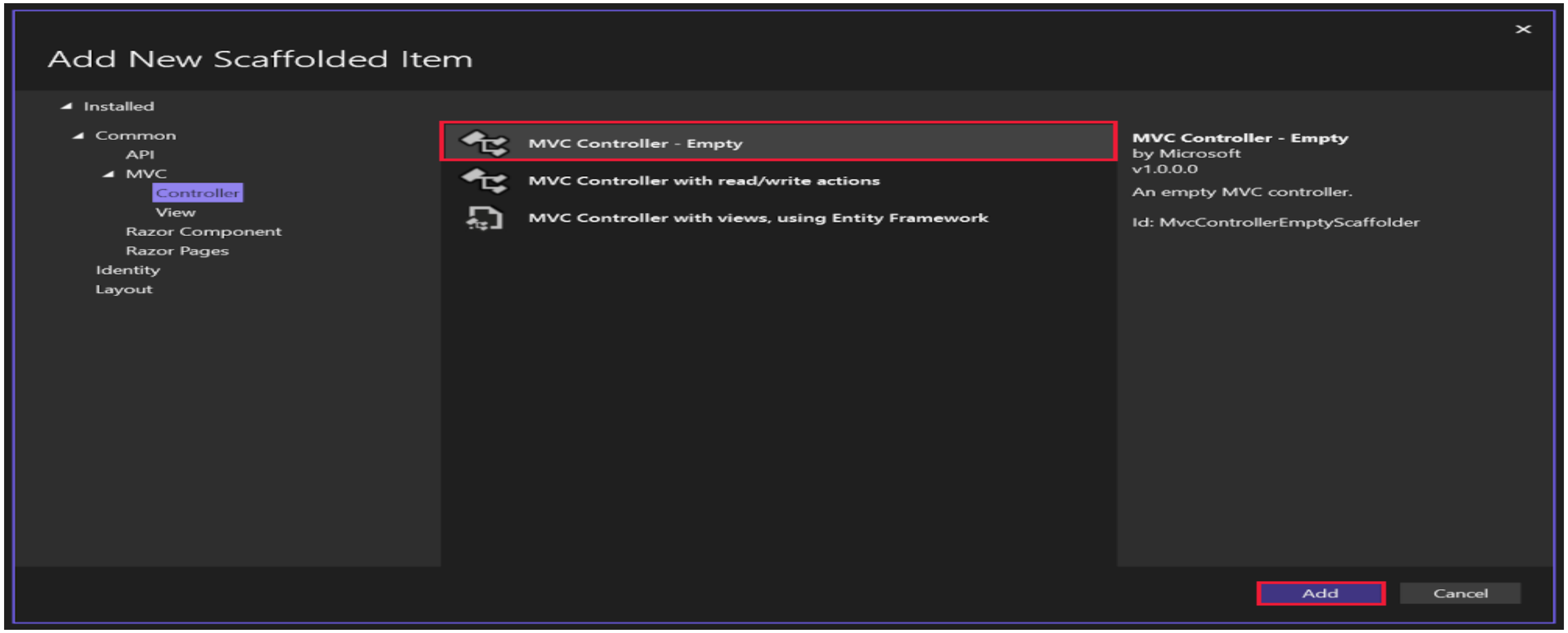
- <!-- Single statement block -->
- @ {
-     varmyMessage = "Hello World";
- }
- <!-- Inline expression or variable -->
- < p > The value of myMessage is: @myMessage < /p>
-     <!-- Multi-statement block -->
- @ {
-     var greeting = "Welcome to our site!";
-     varweekDay = DateTime.Now.DayOfWeek;
-     vargreetingMessage = greeting + " Here in Huston it is: " + weekDay;
- } < p > The greeting is: @greetingMessage < /p>

## Add a controller

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



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



MVC Controller - Empty
by Microsoft
v1.0.0.0

An empty MVC controller.

Id: MvcControllerEmptyScaffolder

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

```
using Microsoft.AspNetCore.Mvc;
namespace Addcontroller.Controllers;

public class HelloWorldController : Controller
{
    //
    // GET: /HelloWorld/
    public string Index()
    {
        return "This is my default action...";
    }
    //
    // GET: /HelloWorld/Welcome/
    public string Welcome()
    {
        return "This is the Welcome action method...";
    }
}
```

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

## How it works:

- **HelloWorldController**: The controller class is named HelloWorldController. The framework automatically drops the "Controller" part when constructing the route.

- **Index() method**: The default route for an action method is /[Controller]/[Action]/[id]?, where [Controller] is the controller name, [Action] is the action method, and [id] is an optional parameter.

## So in this case:

- HelloWorldController:The class is named HelloWorldController, so the controller name for routing purposes is just HelloWorld.
- Index() action method: When no specific action is provided in the URL, the framework looks for the Index() action as the default method. If you access /HelloWorld/, it will call the Index() method and return the string "This is my default action...".
- Welcome() action method:Accessing /HelloWorld/Welcome/ will call the Welcome() action and return the string "This is the Welcome action method...".

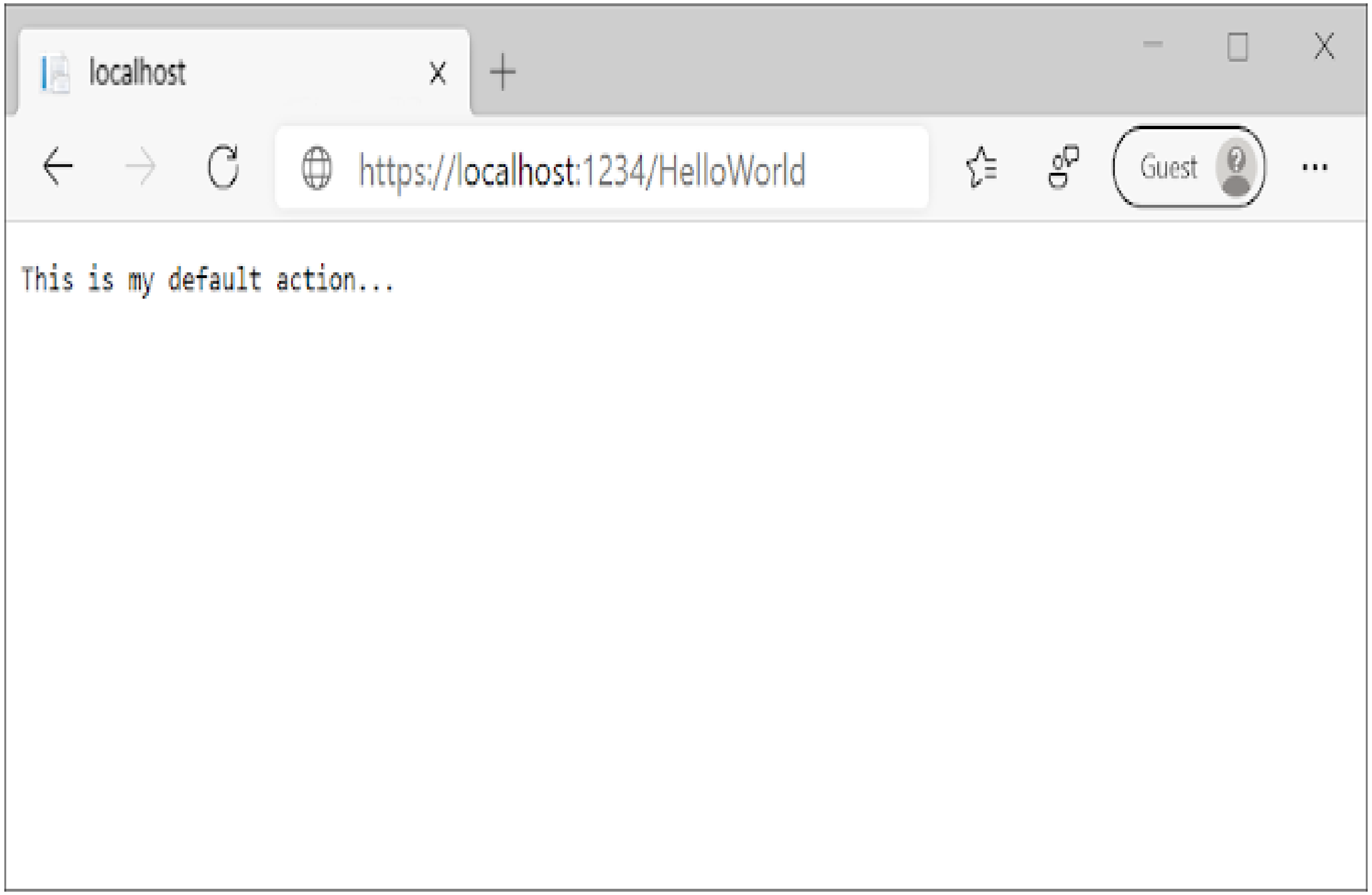
- Every public method in a controller is callable as an HTTP endpoint. In the sample above, both methods return a string. Note the comments preceding each method.
- An HTTP endpoint:
- Is a targetable URL in the web application, such as https://localhost:5001/HelloWorld.
- Combines:
  - The protocol used: HTTPS.
  - The network location of the web server, including the TCP port: localhost:5001.
  - The target URI: HelloWorld.

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

- /[Controller]/[ActionName]/[Parameters]
- The routing format is set in the Program.cs file.

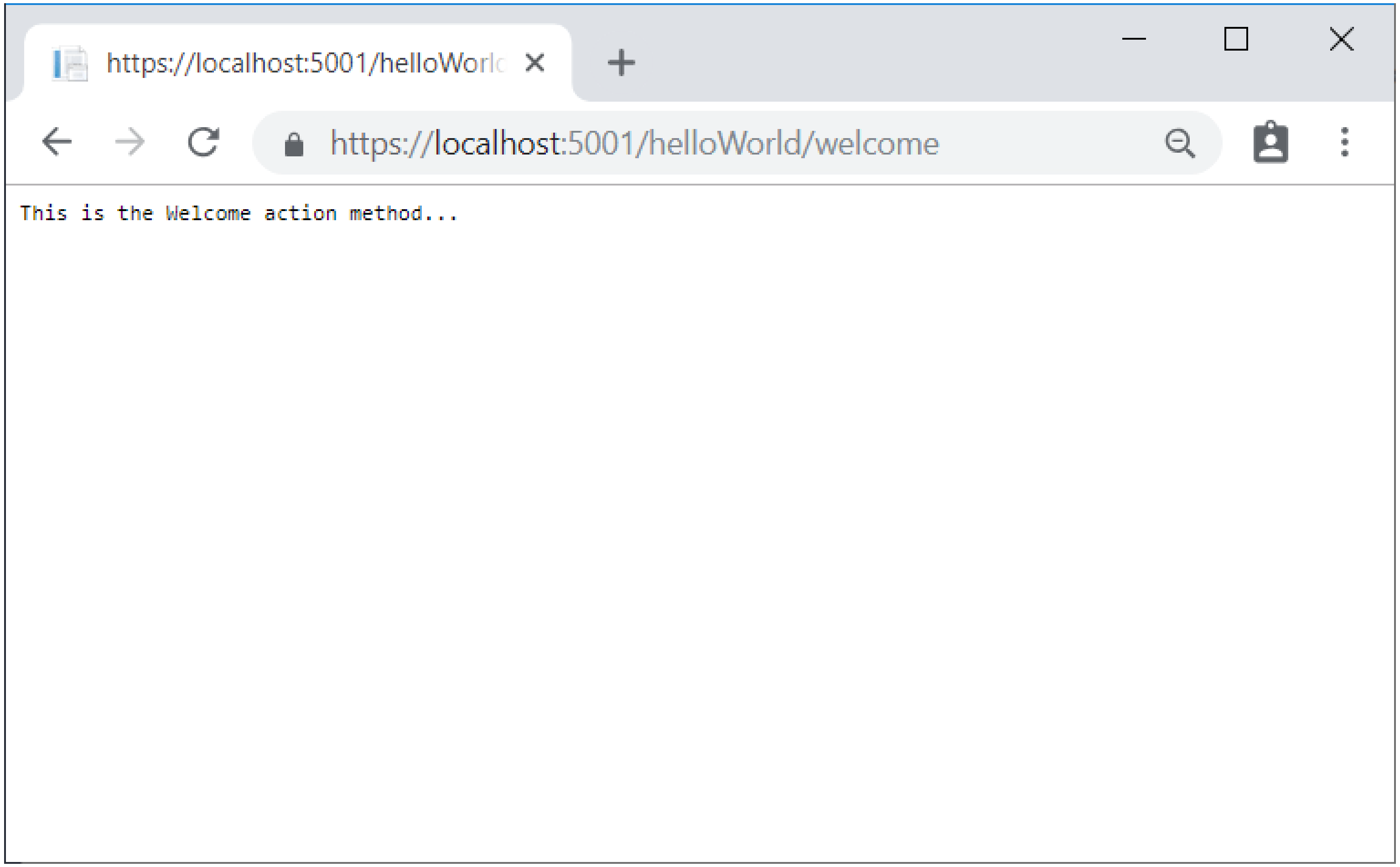
```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

- When you browse to the app and don't supply any URL segments, it defaults to the "Home" controller and the "Index" method specified in the template line highlighted above. In the preceding URL segments:
  - The first URL segment determines the controller class to run. So localhost:5001/HelloWorld maps to the **HelloWorld** Controller class.
  - The second part of the URL segment determines the action method on the class. So localhost:5001/HelloWorld/Index causes the Index method of the HelloWorldController class to run. Notice that you only had to browse to localhost:5001/HelloWorld and the Index method was called by default. Index is the default method that will be called on a controller if a method name isn't explicitly specified.



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

- The third part of the URL segment ( id) is for route data.


https://localhost:5001/helloWorl
https://localhost:5001/helloWorld/welcome

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

## add a view to an ASP.NET Core MVC app

- View templates are created using Razor. Razor-based view templates:
  - Have a .cshtml file extension.
  - Provide an elegant way to create HTML output with C#.

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

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

The preceding code:

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

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

## Add a view

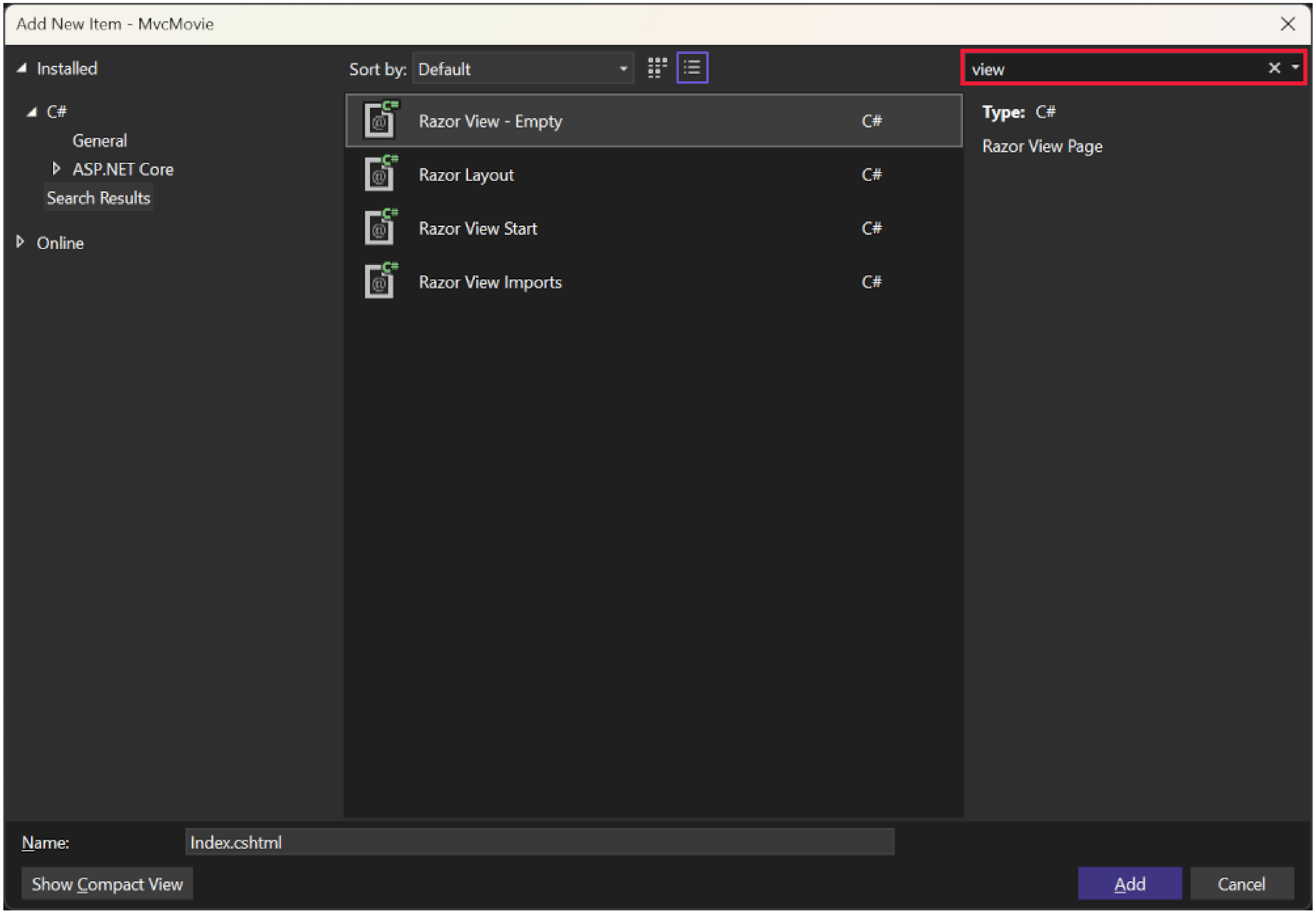
- Right-click on the Views folder, and then Add > New Folder and name the folder HelloWorld.

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

- In the Add New Item dialog select Show All Templates.

- In the Add New Item - MvcMovie dialog:

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


Add New Item - MvcMovie
Installed
C#
General
ASP.NET Core
Search Results
Online
Sort by: Default
view
Razor View - Empty C#
Razor Layout C#
Razor View Start C#
Razor View Imports C#
Type: C#
Razor View Page
Name: Index.cshtml
Show Compact View
Add
Cancel

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

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

<h2>Index</h2>

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

## Change views and layout pages

- Open the Views/Shared/_Layout.cshtml file.

- Layout templates allow:

- Specifying the HTML container layout of a site in one place.
- Applying the HTML container layout across multiple pages in the site.
- Find the @RenderBody() line. RenderBody is a placeholder where all the view-specific pages you create show up, wrapped in the layout page. For example, if you select the Privacy link, the Views/Home/Privacy.cshtml view is rendered inside the RenderBody method.

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

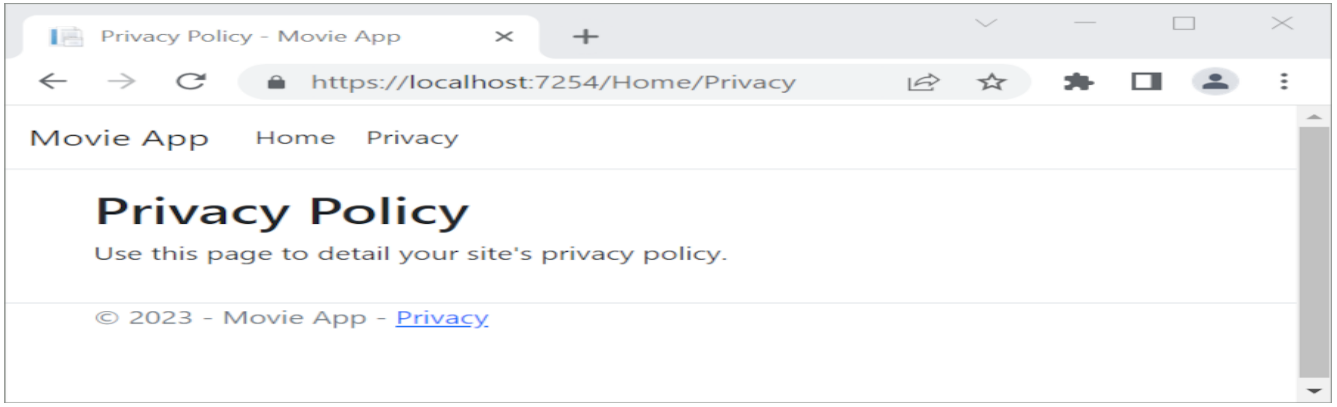
- Replace the content of the Views/Shared/_Layout.cshtml file with the following markup. The changes are highlighted:

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

```
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-
white border-bottom box-shadow mb-3">
            <div class="container-fluid">
                <a class="navbar-brand" asp-area="" asp-controller="Movies" asp-
action="Index">Movie App</a>
                <button class="navbar-toggler" type="button" data-bs-toggle="collapse"
data-bs-target=".navbar-collapse" aria-controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex justify-content-
between">
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Home"
asp-action="Index">Home</a>
                        </li>
```

```
 <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2024 - movieApp - <a asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
        </div>
    </footer>
```

- Note: The Movies controller hasn't been implemented. At this point, the Movie App link isn't functional.
- Save the changes and select the Privacy link. Notice how the title on the browser tab displays Privacy Policy - Movie App instead of Privacy Policy



Privacy Policy - Movie App

https://localhost:7254/Home/Privacy

Movie App Home Privacy

Privacy Policy

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

© 2023 - Movie App - Privacy

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

- Open the Views/HelloWorld/Index.cshtml view file. @{

```
  ViewData["Title"] = "Movie List";
}
<h2>My Movie List</h2>
<p>Hello from our View Template!</p>
```

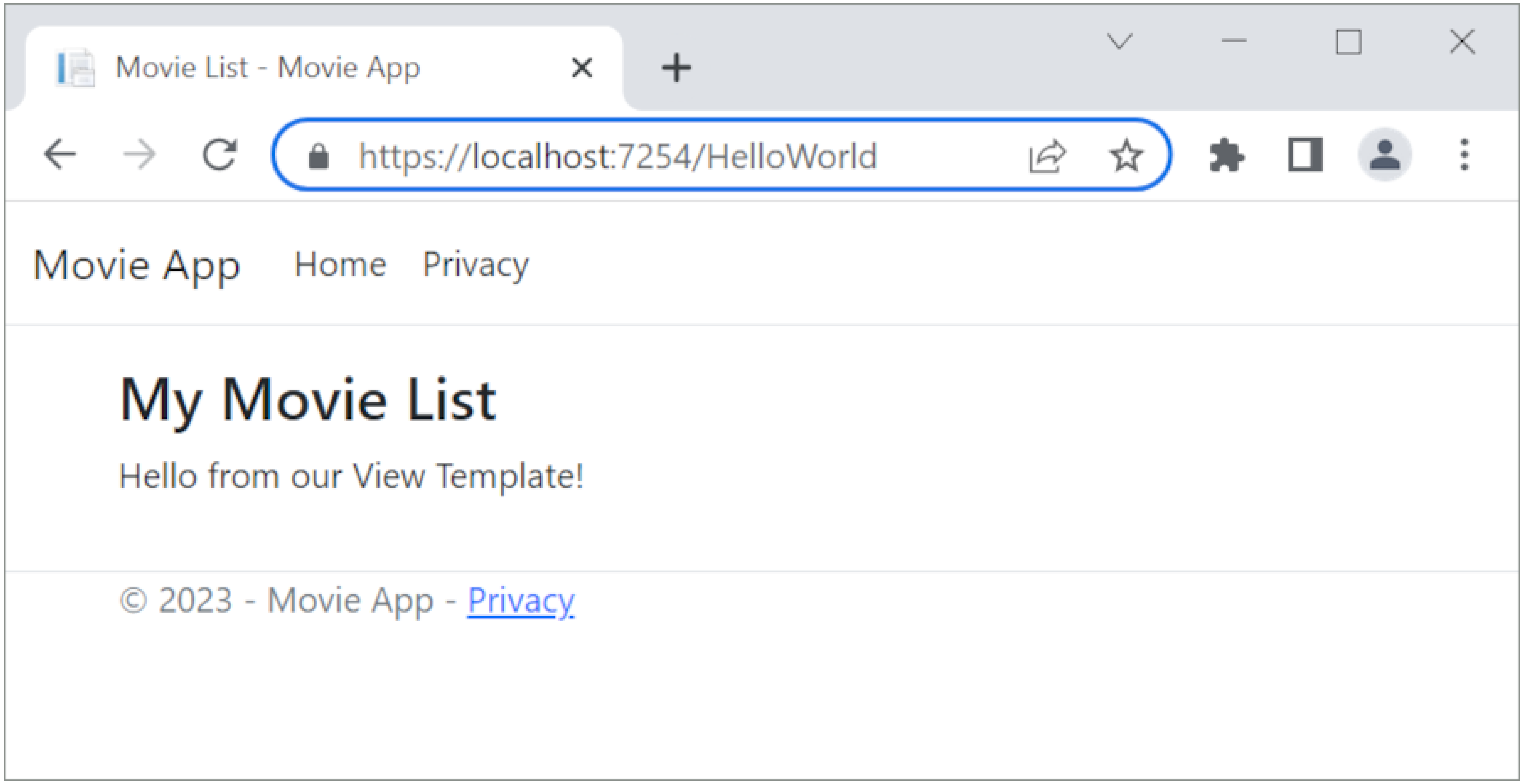
- The title and <h2> element are slightly different so it's clear which part of the code changes the display.

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

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

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

- Notice that the following have changed:
- Browser title.
- Primary heading.
- Secondary headings.



My Movie List

Hello from our View Template!

© 2023 - Movie App - Privacy

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