# ASP.NET Core MVC

# MVC

- An ASP.NET MVC (Model-View-Controller) application follows a specific architecture pattern that separates the concerns of an application into three main components: the Model, the View, and the Controller. Here's an overview of the architecture of an ASP.NET MVC application:

# MVC

- Model:
  - The Model represents the data and the business logic of the application. It encapsulates the application's data structure, defines how to retrieve and manipulate the data, and performs validation and business rules. The model can include classes, entities, data access logic, and other components related to data management.
- View:
  - The View is responsible for presenting the user interface (UI) to the user. It defines how the data from the Model is displayed and how users can interact with it. Views are typically represented using HTML, CSS, and client-side scripting languages such as JavaScript. In ASP.NET MVC, views are usually created using the Razor view engine, which allows embedding server-side code within the view.
- Controller:
  - The Controller handles user requests, processes them, and coordinates the interaction between the Model and the View. It receives input from the user via the View, interacts with the Model to perform the necessary operations, and determines which View to display as a response. Controllers are responsible for routing incoming requests to appropriate actions, retrieving data from the Model, and passing the data to the View.

# Folder structure

- Controllers: This folder contains the controller classes that handle user requests.
- Models: This folder contains classes that define the data structure and business logic of the application.
- Views: This folder contains the view files that define the UI layout and presentation.
- wwwroot: This folder contains static files such as HTML, CSS, JavaScript, and images that are served directly to the client.
- appsettings.json: This file stores configuration settings for the application, such as database connection strings, logging settings, and other application-specific settings.
- Startup.cs: This file contains the startup code for the application. It configures services, middleware, and other components required by the application.
- Program.cs: This file contains the entry point of the application.

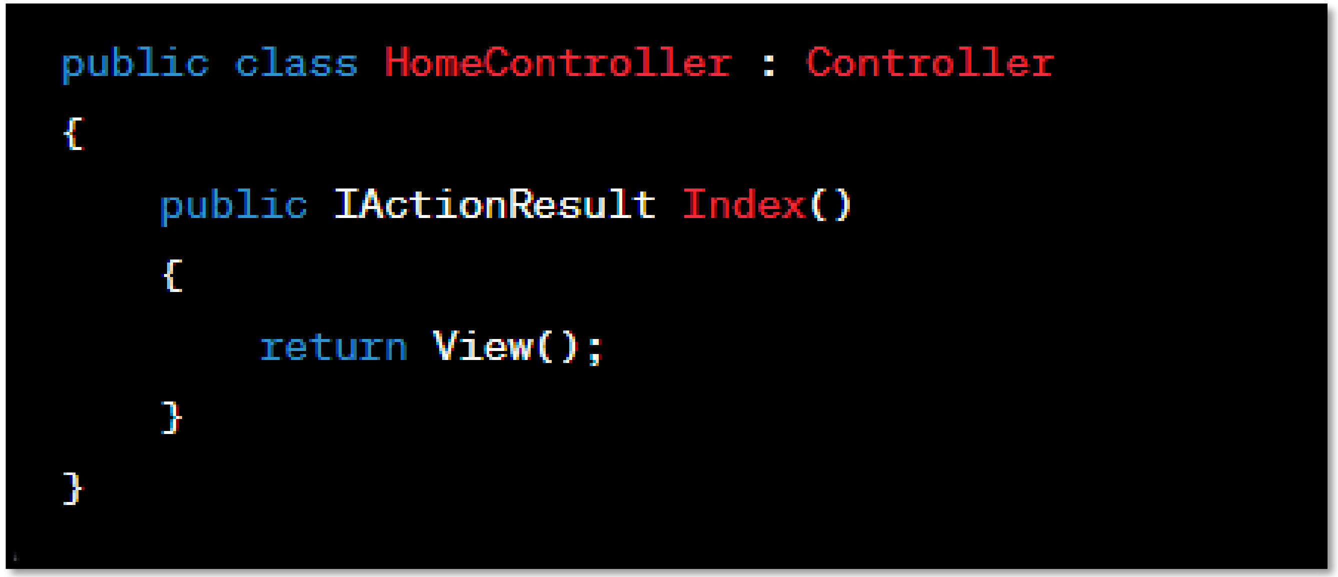
## Understanding Controllers and Actions:

Controllers in ASP.NET Core MVC are responsible for handling user requests, performing the necessary logic, and returning responses. Controllers contain action methods that are invoked based on the user's request

**Creating a Controller:**

To create a controller in ASP.NET Core MVC, you typically create a new class that inherits from the Controller base class or its derived classes. The controller class name should end with the word "Controller" (e.g., HomeController). You can create controllers using the following steps:

- Add a new class file to your project.
- Inherit from the Controller base class.
- Define action methods within the controller class.



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

**Invoking Actions:**
Actions within a controller are invoked when a user makes a request to a specific URL associated with that action. The URL mapping is defined in the routing configuration of the application. The routing system maps the URL to the appropriate controller and action. For example, a request to /Home/Index would invoke the Index action method of the HomeController.

**Action Attributes:**
Action methods can be decorated with attributes to specify additional behavior. Here are a few commonly used attributes:

[HttpGet] and [HttpPost]: These attributes specify the HTTP verb (GET or POST) that the action method will respond to. By default, action methods respond to both GET and POST requests. Using these attributes allows you to define separate action methods for each HTTP verb.

**Returning Action Results:**
Action methods return IActionResult or its derived types to provide a response to the user. Common action results include:
ViewResult: Returns a view that renders HTML to the user.

- RedirectToAction: Redirects the user to another action within the same or a different controller.
- JsonResult: Returns JSON data to the user.
- PartialViewResult: Returns a partial view that can be used to render a portion of a page.
- ContentResult: Returns a plain text or HTML content as the response.

## Understanding Views and Models:

Views in ASP.NET Core MVC are responsible for rendering the user interface and displaying data to the user. Models represent the data and business logic of the application. Views and Models work together to present and manipulate data. Here's an overview of how Views and Models are used in ASP.NET Core MVC:

Models:
Models in ASP.NET Core MVC represent the data structure and business logic of the application. They encapsulate the data and provide methods for accessing and manipulating it. Models can include classes, entities, data access logic, validation rules, and business rules.

View Models:
View Models are specific classes that are designed to serve data to the Views. They are used to pass data from the Controller to the View. View Models are created when the data needed by the View is not directly available in the Model or requires additional processing. View Models can combine data from multiple Models and include only the necessary information for a specific View.

## Razor Views:

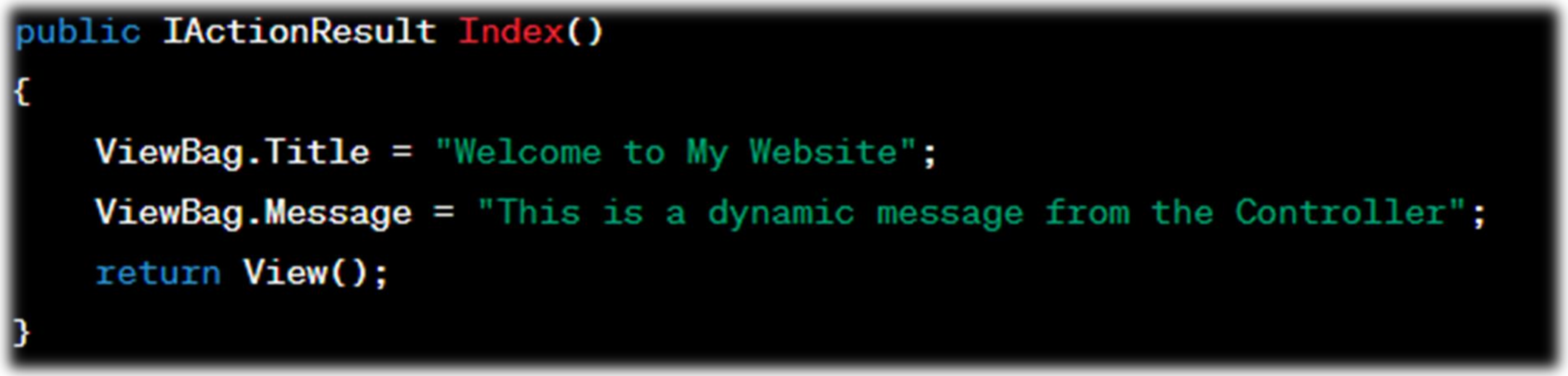
Razor Views are the templates used to define the UI layout and presentation in ASP.NET Core MVC. Razor is a markup syntax that combines HTML markup and server-side code. Razor Views can access data from the Model or View Model and display it dynamically. Views can use loops, conditions, and other logic to render different parts of the UI based on the provided data.

Razor Views are created as .cshtml files in the Views folder of the ASP.NET Core MVC project. Here are the steps to create a Razor View:

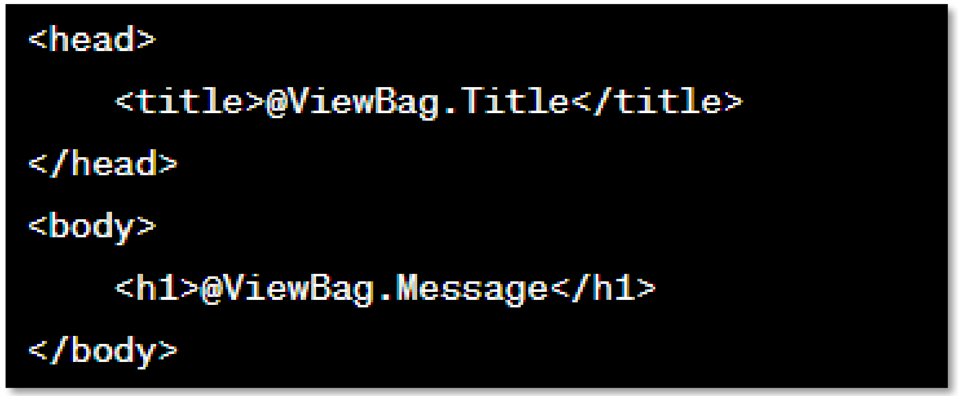
- Add a new .cshtml file to the Views folder or an appropriate subfolder.
- Define the HTML markup and UI elements within the .cshtml file.
- Use Razor syntax (@) to embed server-side code and access data from the Model or View Model.
- Save the file.

## Understanding ViewBag:

ViewBag is a dynamic property in ASP.NET Core MVC that allows you to pass data from the Controller to the View. It is a lightweight and dynamic way to share data between the Controller and the View without creating a specific View Model. ViewBag is part of the ViewData dictionary, which stores data for a specific request.



```
public IActionResult Index()
{
    ViewBag.Title = "Welcome to My Website";
    ViewBag.Message = "This is a dynamic message from the Controller";
    return View();
}
```

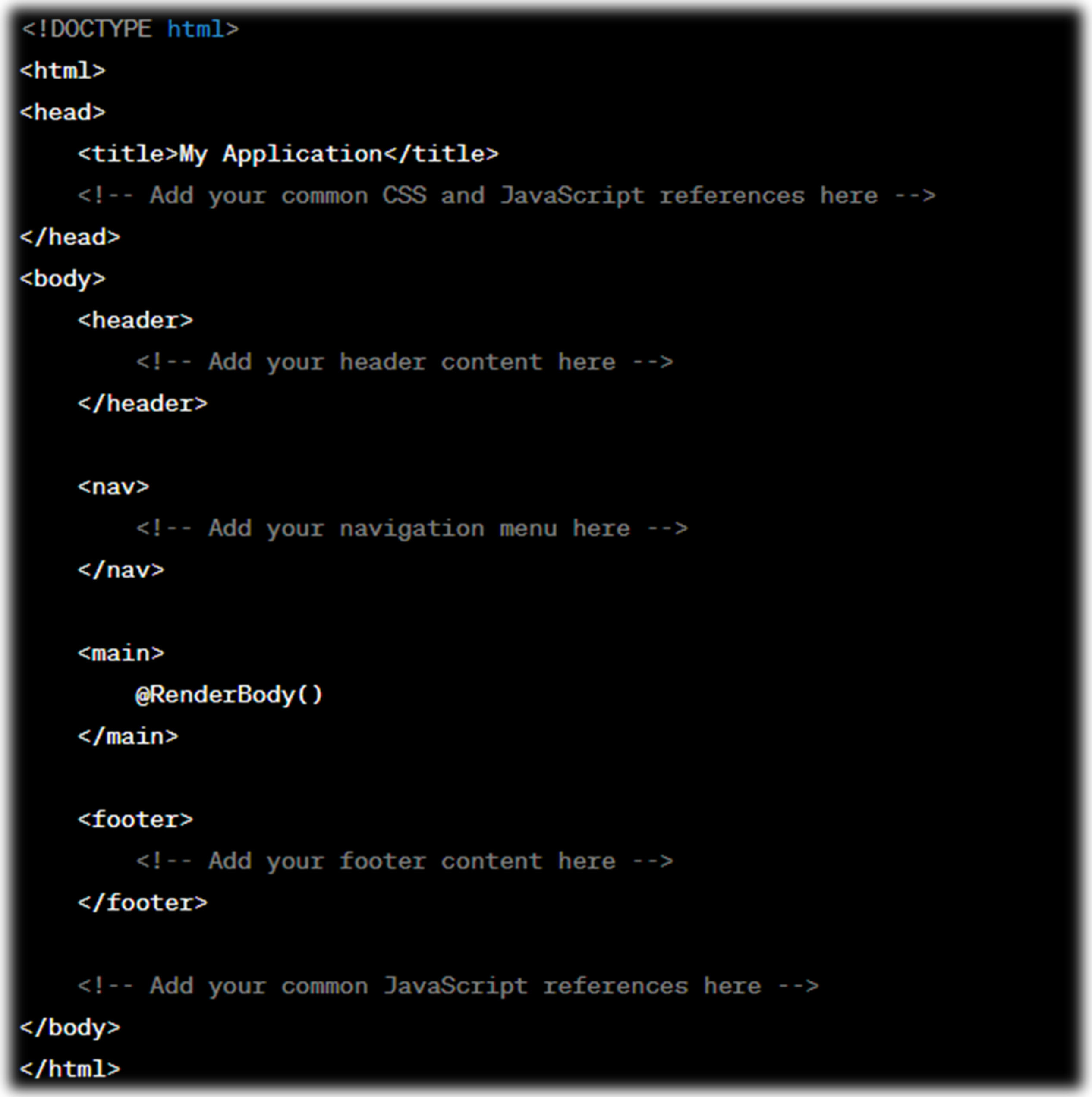


```
<head>
    <title>@ViewBag.Title</title>
</head>
<body>
    <h1>@ViewBag.Message</h1>
</body>
```

## Creating Layout and using with associated views

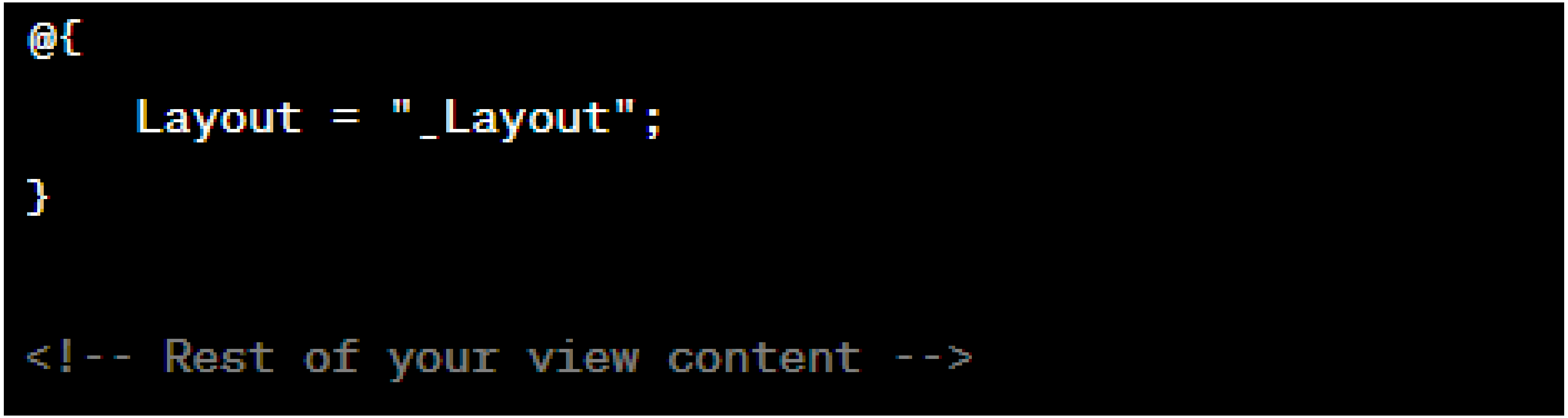
- A layout in ASP.NET Core MVC defines the common structure and UI elements shared across multiple views. It allows you to define a consistent layout for your application, including headers, footers, navigation menus, and other common elements

  - In the Views folder of your ASP.NET Core MVC project, create a new folder called "Shared" (if it doesn't already exist).
  - Inside the "Shared" folder, create a new file called "Layout.cshtml". The leading underscore "" indicates that it is a partial view.
  - Define the HTML structure of your layout, including common elements like headers, footers, and navigation menus. The main content area will be represented by @RenderBody().



```html
<!DOCTYPE html>
<html>
<head>
    <title>My Application</title>
    <!-- Add your common CSS and JavaScript references here -->
</head>
<body>
    <header>
        <!-- Add your header content here -->
    </header>

    <nav>
        <!-- Add your navigation menu here -->
    </nav>

    <main>
        @RenderBody()
    </main>

    <footer>
        <!-- Add your footer content here -->
    </footer>

    <!-- Add your common JavaScript references here -->
</body>
</html>
```

## Associate Views with the Layout:



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

<!-- Rest of your view content -->
```