ASP.NET Core - True Ultimate Guide

Section 8: Razor Views

MVC Architecture

Model-View-Controller (MVC) is a design pattern that organizes your application into three distinct components, each with a specific responsibility:

1. **Model:**
    - Represents the data and business logic of your application.
    - Encapsulates data access, validation rules, and any core business operations.
    - **Types of Models:**
        - **Business Model (Domain Model):** Represents the real-world entities and relationships of your application domain (e.g., Product, Customer, Order).
        - **View Model:** A tailored model specifically designed to provide data to a view. It might aggregate data from multiple business models or contain properties for UI-specific concerns.

2. **View:**
    - Renders the user interface (UI) based on the data provided by the controller.
    - Typically, views are HTML templates with embedded server-side code (Razor syntax in ASP.NET Core) that dynamically generate the content.

3. **Controller:**
    - Acts as the intermediary between the model and the view.
    - Receives user input (from requests), interacts with the model to perform actions or retrieve data, and then selects the appropriate view to present the results.

Execution Flow of a Request in MVC

1. **Routing:** The incoming request is processed by the routing middleware, which determines the appropriate controller and action method to handle it based on the URL pattern and HTTP method.
2. **Model Binding:** If the action method has parameters, the model binder extracts data from the request (query string, route values, form data, body) and attempts to convert it into the types expected by the parameters.

3. **Model Validation:** The model binder validates the bound data using data annotations and any custom validation logic you've implemented. If validation fails, errors are added to the ModelState object.
4. **Controller Action Execution:** If the model is valid, the controller action method executes its logic, which might involve:
    - Interacting with business models to retrieve or update data.
    - Performing calculations or other business operations.
    - Preparing a view model to pass data to the view.
5. **View Selection:** The controller selects the appropriate view to render and passes the view model to it.
6. **View Rendering:** The view engine (Razor) generates the HTML output based on the view template and the data in the view model.
7. **Response:** The rendered HTML is sent back to the client as the HTTP response.

Responsibilities of MVC Components

- **Controller:**
    - Handles incoming requests and routes them to the correct action methods.
    - Interacts with the model to fetch or modify data.
    - Selects the appropriate view and passes necessary data to it.
    - Handles errors and redirects.

- **Model (Business Model):**
    - Encapsulates the business logic and data access of the application.
    - Represents the core entities and relationships in your domain.
    - Defines validation rules for ensuring data integrity.

- **View Model:**
    - Tailored for a specific view, containing only the data required by that view.
    - Simplifies data presentation in the view and avoids exposing unnecessary details.

- **View:**
    - Renders the user interface based on the data provided by the controller.

- Uses Razor syntax to combine HTML markup with C# code for dynamic content generation.

Benefits and Goals of MVC

- **Separation of Concerns (SoC):** The clear division of responsibilities makes code more organized, maintainable, and testable.
- **Testability:** Individual components (models, views, controllers) can be tested in isolation.
- **Reusability:** Views and models can be reused in different contexts.
- **Parallel Development:** Different team members can work on different parts of the application simultaneously.
- **Maintainability:** Changes to one component are less likely to affect others.
- **Extensibility:** New features can be added more easily due to the modular structure.
- **Scalability:** MVC architecture lends itself well to scaling as the application grows.

The MVC pattern provides a structured and organized approach to building complex web applications, facilitating collaboration, code reusability, and long-term maintainability. Understanding these core concepts will make you a more effective ASP.NET Core developer.

Views

In ASP.NET Core MVC, views are responsible for rendering the user interface (UI) that your application presents to the user. They are typically HTML templates with embedded Razor syntax, which is a powerful templating engine that allows you to seamlessly blend HTML markup with C# code.

Notes

- **Dynamic Content:** Views can generate dynamic content based on data passed to them from the controller (the model). This allows you to display information fetched from databases, user inputs, and other sources.
- **Razor Syntax:** Views use Razor syntax, which starts with @ symbols, to embed C# code within the HTML. This code can perform tasks like looping through data collections, conditional rendering, and accessing model properties.

AddControllersWithViews() Method

This extension method is used to register the necessary services for MVC (models, views, controllers) with the dependency injection container. It's a shortcut for adding multiple services at once, including:

- **Controller Discovery:** Automatically discovers controller classes in your project.
- **View Engine:** Configures Razor as the view engine.
- **Model Binding:** Sets up model binding for handling form submissions.
- **Validation:** Enables model validation.

ViewResult

The ViewResult class is an action result type in ASP.NET Core MVC that represents a view to be rendered. When a controller action returns a ViewResult, the MVC framework locates the corresponding view file, passes the model data (if any) to it, and renders the view's content as HTML.

Default View Locations

ASP.NET Core MVC follows a convention for determining where to find view files:

- **By Convention:** By default, the view engine looks for views in the Views/[ControllerName]/[ActionName].cshtml path. For example, the Index action method in the HomeController would look for a view file at Views/Home/Index.cshtml.
- **Overriding with ViewName:** You can explicitly specify the view name using the ViewName property of the ViewResult or the View() helper method.
- **Shared Views:** Shared views are stored in the Views/Shared folder and can be used by multiple controllers.

Code

```
// Program.cs

builder.Services.AddControllersWithViews(); // Enables MVC features


// ...
```

```
// HomeController.cs
[Route("home")]
public IActionResult Index()
{
   return View(); // Renders Views/Home/Index.cshtml
}

// Views/Home/Index.cshtml
<!DOCTYPE html>
<html>
   <head>
      <title>Asp.Net Core App</title>
   </head>
   <body>
      Welcome
   </body>
</html>
```

1. **Enabling MVC:** The AddControllersWithViews() method configures the application for MVC, including view support.
2. **Controller Action:** The Index action method in HomeController returns a ViewResult without specifying a view name.
3. **View Location:** Since no view name is explicitly provided, the MVC framework follows the convention and looks for the view at Views/Home/Index.cshtml.
4. **View Content:** The Index.cshtml view contains simple HTML that will be rendered as the response.

Razor View Engine

Razor is the default view engine in ASP.NET Core MVC. It provides a concise and elegant way to create dynamic web pages by combining C# code with HTML markup.

Key Razor Syntax Elements

1. **Code Blocks (@{...}):**
    - Purpose: Enclose multi-line C# code statements.
    - Use Cases:
        - Declaring variables
        - Defining complex logic
        - Executing database queries or other operations
2. **Expressions (@variable or @method()):**
    - Purpose: Embed C# expressions directly into the HTML output.
    - Use Cases:
        - Displaying values from variables or model properties
        - Calling helper methods or functions
3. **Literals (@: or <text>):**
    - Purpose: Output raw text without HTML encoding.
    - Use Cases:
        - Displaying plain text, HTML snippets, or values that might contain HTML characters

Control Flow in Razor Views

1. **@if, @else, @elseif:**
    - Purpose: Conditional rendering of HTML blocks based on C# conditions.
    - Example:

```
@if (person.DateOfBirth.HasValue)
{
   <p>Age: @(Math.Round((DateTime.Now - person.DateOfBirth).Value.TotalDays / 365.25)) years old</p>
}
```

```
else
{
    <p>Date of birth is unknown</p>
}
```

2. **@switch:**
    - Purpose: Choose one of several blocks of code to execute based on the value of an expression.
    - Example:

```
@switch (person.PersonGender)
{
    case Gender.Male:
        <p>November 19 is International Men's Day</p>
        break;
    case Gender.Female:
        <p>March 8 is International Women's Day</p>
        break;
    // ... other cases ...
}
```

3. **@foreach:**
    - Purpose: Iterate over a collection (e.g., a list or array) and render HTML for each item.
    - Example:

```
@foreach (var person in people)
{
    <div>@person.Name, @person.PersonGender</div>
}
```

4. **@for:**
    - Purpose: Execute a code block a specific number of times.
    - Example:

```
@for (int i = 0; i < 5; i++)
{
   <p>Iteration: @i</p>
}
```

Notes

- **Razor Syntax:** Familiarize yourself with Razor syntax (code blocks, expressions, literals) for embedding C# code in your views.
- **Control Flow:** Understand how to use if, else, switch, foreach, and for to control the flow of logic and create dynamic views.
- **Model Binding:** Views often work with models passed from the controller. Use Razor expressions to access and display model properties.

Local Functions in Razor Views

Local functions are C# functions defined within the scope of a Razor code block (a code block enclosed in @{ ... }). They allow you to encapsulate reusable logic directly in your views, making your view code more modular, organized, and easier to read.

Syntax and Features

- **Declaration:** Local functions are declared using the standard C# method syntax, typically within an @functions { ... } block.
- **Scope:** They can only be called within the code block or section where they are defined.
- **Parameters and Return Types:** Local functions can take parameters and return values, just like regular C# methods.
- **Accessibility:** They are implicitly private to the view.
- **Access to View Data:** Local functions can access variables and model properties declared within the same code block.

Why Use Local Functions?

- **Encapsulation:** Group related logic into self-contained functions, improving code readability.
- **Readability:** Break down complex code into smaller, more manageable chunks.
- **Reusability:** Call local functions multiple times within the same view, avoiding code duplication.
- **Cleaner Views:** Reduce the amount of inline C# code scattered throughout your HTML markup.

Code

```
@using ViewsExample.Models

@{
    string appTitle = "Asp.Net Core Demo App";
    List<Person> people = new List<Person>()
    {
        // ... (person data)
    };
}

@functions {
    double? GetAge(DateTime? dateOfBirth)
    {
        if (dateOfBirth is not null)
        {
            return Math.Round((DateTime.Now - dateOfBirth.Value).TotalDays / 365.25);
        }
        else
        {
            return null;
        }
    }
```

```
    int x = 10; // Example of a local variable

    string Name { get; set; } = "Hello name"; // Example of a local property
}

<!DOCTYPE html>
<html>
    <head>
        <title>@appTitle</title>
        <meta charset="utf-8" />
    </head>
    <body>
        <h1>Welcome to @Name</h1>
        @for (int i = 0; i < 2; i++)
        {
            Person person = people[i];
            <div>
                @person.Name
                <span>, </span>
                <span>@person.PersonGender</span>
                @if (person.DateOfBirth != null)
                {
                    <span>@person.DateOfBirth.Value.ToString("MM/dd/yyyy")</span>
                    <span>@GetAge(person.DateOfBirth) years old</span>
                }
            </div>
        }
    </body>
</html>
```

In this code:

1. **Local Function:** GetAge(DateTime? dateOfBirth) calculates the age of a person based on their date of birth.
2. **Local Variable:** int x = 10; This is not used in the view but demonstrates how to declare local variables within a Razor code block.
3. **Local Property:** string Name { get; set; } = "Hello name"; This is used to set the title of the page.

The GetAge function is called within the @foreach loop to display the age of each person if their date of birth is available.

Important Considerations

- **Overuse:** Avoid overusing local functions. They're great for encapsulation, but too many can make your views harder to follow.
- **Alternatives:** Consider helper methods or view components for more complex or reusable logic.

Html.Raw()

In Razor views, the default behavior is to automatically encode HTML content to prevent potential security vulnerabilities like cross-site scripting (XSS) attacks. However, sometimes you have legitimate HTML content (e.g., from a rich text editor or a trusted source) that you want to render as-is, without encoding. This is where Html.Raw() comes in.

Purpose

- **Bypass HTML Encoding:** Html.Raw() prevents Razor from encoding the HTML content you provide, allowing it to be rendered directly in the browser.
- **Displaying Trusted HTML:** Use it when you trust the source of the HTML content and want to preserve its formatting and structure.
- **Dynamic Content Generation:** Html.Raw() can be used to dynamically insert HTML fragments into your views based on data or logic.

Syntax

@Html.Raw(htmlContent)

where htmlContent is a string variable or expression containing the HTML you want to render without encoding.

Important Considerations

- **Security Risk: Use Html.Raw() with extreme caution.** If the htmlContent originates from user input or an untrusted source, it could lead to XSS vulnerabilities. Always sanitize and validate user-generated content before rendering it with Html.Raw().
- **Content Security Policy (CSP):** Consider implementing a CSP to further mitigate XSS risks. A CSP defines a set of rules that govern how the browser handles external scripts, styles, and other resources.

Code

```
@{
   // ... (other variables and logic) ...
   string alertMessage = $"<script>alert('{people.Count} people found')</script>";
}

<!DOCTYPE html>
<html>
<head>
   </head>
<body>
   @Html.Raw(alertMessage)
   <h1>Welcome</h1>
   @for (int i = 0; i < 2; i++)
   {
      // ... (rest of the view code) ...
   }
</body>
</html>
```

In this code:

1. **Raw HTML String:** The variable alertMessage contains a string of HTML that includes a <script> tag intended to display an alert message with the number of people found.

2. **Html.Raw() Usage:** The @Html.Raw(alertMessage) line tells Razor to render the contents of alertMessage directly into the HTML output without encoding. This will result in the following HTML in the output:

<script>alert('3 people found')</script>

3. **Client-Side Execution:** When the browser parses this HTML, it will execute the JavaScript code inside the <script> tag, triggering an alert box.

4. **No sanitization:** In this example, since the number of people is not user input, there is no risk of XSS attack. However, if the content of the alert message is coming from user input, it is essential to santize it before displaying it to prevent XSS attacks.

Notes

- **Trust but Verify:** Only use Html.Raw() when you're absolutely sure the content is safe and from a trusted source.
- **Security Risks:** Be aware of the potential for XSS vulnerabilities when rendering raw HTML.
- **Alternative Approaches:** If you need to inject dynamic HTML content, consider safer alternatives like partial views or view components.
- **Content Security Policy (CSP):** Implement a CSP to add an extra layer of protection against XSS attacks.

ViewData and ViewBag

Both ViewData and ViewBag are mechanisms in ASP.NET Core MVC for passing data from your controller actions to your views. While they serve the same purpose, they differ in their implementation and syntax.

- **ViewData (Dictionary-Based):**
    - It's a dictionary-like object (ViewDataDictionary) that stores key-value pairs.
    - Keys are strings, and values can be of any type.

    - Access data using string keys: ViewData["keyName"].
    - Requires casting when retrieving non-string values.

- **ViewBag (Dynamic):**
    - It's a dynamic wrapper around the ViewData dictionary.
    - Allows you to access data using dot notation: ViewBag.keyName.
    - No explicit casting is needed for non-string values.

Availability in Controllers and Views

Both ViewData and ViewBag are accessible in:

- **Controllers:** You set values in the controller action and they are passed to the view.
- **Views:** You retrieve values from the ViewData dictionary or the ViewBag dynamic object in the view to render content.

Code of ViewData:

```
// HomeController.cs (Controller)
ViewData["appTitle"] = "Asp.Net Core Demo App";
ViewData["people"] = people;

// Index.cshtml (View)
<title>@ViewData["appTitle"]</title>
List<Person>? people = (List<Person>?)ViewData["people"];
```

Code of ViewBag:

```
// HomeController.cs (Controller)
ViewBag.appTitle = "Asp.Net Core Demo App";
ViewBag.people = people;

// Index.cshtml (View)
```

<title>@ViewBag.appTitle</title>

@foreach (Person person in ViewBag.people)

Best Practices

- **Choose One:** It's generally recommended to pick either ViewData or ViewBag and use it consistently throughout your project to maintain a clear and unified approach.
- **Strong Typing with View Models:** While ViewData and ViewBag offer flexibility, they lack compile-time type safety. For larger applications, prefer strongly typed view models to pass data to views.
- **Limited Scope:** Understand that ViewData and ViewBag data exists only for the duration of the current request and response cycle. It's not meant for persisting data between requests.
- **Meaningful Keys:** Use descriptive and meaningful keys for your data to improve code readability.

Things to Avoid

- **Overuse:** Don't overstuff ViewData or ViewBag with excessive data. Keep it concise and focused on the specific information needed by the view.
- **Magic Strings:** Avoid hardcoding strings for keys. Use constants or enums for better maintainability.
- **Sensitive Data:** Never pass sensitive data like passwords or authentication tokens through ViewData or ViewBag.

Strongly Typed Views

In ASP.NET Core MVC, a strongly typed view is a view that is associated with a specific model class. This means that the view has direct access to the properties and methods of that model, providing compile-time type checking and IntelliSense support within the view.

The @model Directive

The @model directive is used at the top of a view to specify the model type for that view. For example, @model Person indicates that the view expects to work with data of the Person class.

When to Use Strongly Typed Views

- **Complex Data:** When your view needs to display or interact with complex data structures that have multiple properties or relationships.
- **Type Safety:** When you want to catch type-related errors during development, rather than at runtime.
- **IntelliSense:** When you want the full power of Visual Studio's IntelliSense to help you code faster and with fewer errors.
- **Refactoring:** When you need to change the model, strongly typed views make it easier to update the corresponding views automatically.

Best Practices

- **Use View Models:** Instead of directly passing your business models to views, create specialized view models tailored to the data needed by each view. This helps keep your views clean and focused.
- **Naming Conventions:** Follow standard naming conventions for your view models (e.g., ProductViewModel, OrderDetailsViewModel).
- **Keep Views Simple:** Views should primarily focus on presentation logic. Avoid complex business logic within views.
- **Partial Views:** Leverage partial views for reusable components, passing view models to them as needed.
- **Leverage Tag Helpers:** Explore the use of Tag Helpers, which provide a more HTML-friendly way to interact with your models in views.

Code:

Controller (HomeController.cs)

```
// ...
[Route("home")]
[Route("/")]
public IActionResult Index()
{
   List<Person> people = new List<Person>()
   {
      new Person() { Name = "John", DateOfBirth = DateTime.Parse("2000-05-06"), PersonGender = Gender.Male},
```

```
        new Person() { Name = "Linda", DateOfBirth = DateTime.Parse("2005-01-09"), PersonGender = Gender.Female},
        new Person() { Name = "Susan", DateOfBirth = null, PersonGender = Gender.Other}
    };

    return View("Index", people); // Pass the 'people' list as the model
}

[Route("person-details/{name}")]
public IActionResult Details(string? name)
{
    // ... (Retrieve the person based on the name) ...
    Person? matchingPerson = people.Where(temp => temp.Name == name).FirstOrDefault();
    return View(matchingPerson); // Pass a single 'Person' object as the model
}
```

Both action methods pass the model(s) to the view using return View(model).

View (Index.cshtml)

```
@using ViewsExample.Models
@model IEnumerable<Person>  // Indicates that the model is a collection of Person objects

<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.appTitle</title>
    </head>
<body>
    <div class="page-content">
    <h1>Persons</h1>
```

```
    @foreach (Person person in Model)
    {
        <div class="box float-left w-50">
            <h3>@person.Name</h3>
            <table class="table w-100">
                <tbody>
                <tr>
                    <td>Gender</td>
                    <td>@person.PersonGender</td>
                </tr>
                <tr>
                    <td>Date of Birth</td>
                    <td>@person.DateOfBirth?.ToString("MM/dd/yyyy")</td>
                </tr>
                </tbody>
            </table>
            <a href="/person-details/@person.Name">Details</a>
        </div>
    }
    </div>


</body>
</html>
```

This view iterates over the Model (which is a list of Person objects) using a foreach loop. Inside the loop, it accesses the properties of each Person object directly (e.g., @person.Name, @person.PersonGender) thanks to strong typing.

View (Details.cshtml)

```
@using ViewsExample.Models
@model Person // Indicates that the model is a single Person object
```

```
<!DOCTYPE html>
<html>
<head>
   <title>Person Details</title>
   </head>
<body>
   <div class="page-content">
      <h1>Person Details</h1>
      <div class="box">
         <h3>@Model.Name</h3>
         <table class="table w-100">
            <tbody>
            <tr>
               <td>Gender</td>
               <td>@Model.PersonGender</td>
            </tr>
            <tr>
               <td>Date of Birth</td>
               <td>@Model.DateOfBirth?.ToString("MM/dd/yyyy")</td>
            </tr>
            </tbody>
         </table>
      </div>
      <a href="/home">Back to persons</a>
   </div>
</body>
</html>
```

This view displays the details of a single Person object. Notice that you can directly access properties like @Model.Name without any casting or dictionary lookups.

Strongly Typed Views with Multiple Models

When your view requires data from more than one model class, the most common and recommended approach is to create a *view model*. A view model is a custom class that encapsulates all the data necessary for a specific view, aggregating properties from multiple models or providing additional properties tailored for the view's presentation needs.

Why Use View Models?

- **Encapsulation:** Keeps your view logic organized and prevents your views from becoming tightly coupled to your underlying business models.
- **Flexibility:** Allows you to combine data from different sources (multiple models, configuration settings, etc.) into a single object for the view.
- **Type Safety:** Strongly typed views with view models offer compile-time type checking and IntelliSense, improving development efficiency.
- **Data Shaping:** You can create properties in the view model specifically designed for how you want to display data in the view, such as formatted strings or calculated values.

Creating and Using View Models

1. **Define the View Model:** Create a new class in your Models folder to represent the view model. This class will have properties to hold the data from the various models your view needs.
2. **Populate in the Controller:** In your controller action, retrieve the data from your different models and use them to create an instance of your view model.
3. **Pass to the View:** Pass the populated view model object to the view using the View method.
4. **Access in the View:** In your view, use the @model directive at the top to specify the type of your view model. You can then access the view model's properties using Model.<PropertyName>.

Code Example: PersonAndProductWrapperModel

```
// PersonAndProductWrapperModel.cs (View Model)

public class PersonAndProductWrapperModel

{

   public Person PersonData { get; set; }

   public Product ProductData { get; set; }
```

```
}

// HomeController.cs (Controller)
public IActionResult SomeAction()
{
    Person person = GetPersonData(); // Fetch person data
    Product product = GetProductData(); // Fetch product data

    var viewModel = new PersonAndProductWrapperModel
    {
        PersonData = person,
        ProductData = product
    };

    return View(viewModel);
}
```

View (PersonAndProduct.cshtml)

```
@using ViewsExample.Models
@model PersonAndProductWrapperModel

<!DOCTYPE html>
<html>
<head>
</head>
<body>
    <div class="page-content">
        <h1>Person and Product</h1>

        <div class="box w-100">
```

```
        <h3>Person</h3>
        <table class="table w-50">
        <tbody>
          <tr>
          <td>Person Name</td>
          <td>@Model.PersonData.Name</td>
          </tr>
          <tr>
          <td>Gender</td>
          <td>@Model.PersonData.PersonGender</td>
          </tr>
        </tbody>
        </table>
    </div>

    <div class="box w-100">
        <h3>Product</h3>
        <table class="table w-50">
        <tbody>
          <tr>
          <td>Product Id</td>
          <td>@Model.ProductData.ProductId</td>
          </tr>
          <tr>
          <td>Product Name</td>
          <td>@Model.ProductData.ProductName</td>
          </tr>
        </tbody>
        </table>
    </div>
</div>
```

</body>

</html>

Key Points:

- **Organization:** View models help keep your views focused on presentation and your controllers focused on data retrieval and processing.
- **Maintainability:** When your underlying models change, you only need to update your view model, not every view that uses that data.
- **Flexibility:** You can include additional properties in your view model that aren't part of your original models. This could be formatting options, computed values, or flags for controlling the view's behavior.

_ViewImports.cshtml

The _ViewImports.cshtml file is a special file in ASP.NET Core MVC that allows you to centralize common directives and settings that apply to multiple views within your application. This file typically resides in the Views folder at the root of your project, but you can also place it within subfolders to apply the settings to views within those specific folders.

What Can You Put in _ViewImports.cshtml?

- **@using Directives:** Import namespaces that you frequently use in your views. This eliminates the need to include these directives in every individual view.
- **@addTagHelper Directives:** Make Tag Helpers available to your views. Tag Helpers are server-side code snippets that generate or modify HTML elements in a more intuitive way than traditional Razor syntax.
- **@inject Directives:** Inject services into your views, making them accessible for use in your Razor code.
- **@model Directive (Optional):** Specify a default model type for all views in the directory or subdirectories (not recommended for complex projects).
- **@layout Directive (Optional):** Set a default layout for all views in the directory or subdirectories.

How It Works

When your application processes a request for a view, it looks for a _ViewImports.cshtml file in the following order:

1. **The same directory as the view:** If found, the directives and settings in this file are applied.
2. **Parent directories:** It then checks the parent directory of the view, and so on, up to the root Views folder.
3. **Root Views folder:** Finally, it checks the _ViewImports.cshtml file in the root Views folder.

Directives in a child folder's _ViewImports.cshtml file can override settings inherited from parent directories.

Code Example: _ViewImports.cshtml

```
@using System.Collections.Generic // Import the generic collections namespace
@using YourProject.Models        // Import your project's models namespace
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers // Add built-in ASP.NET Core Tag Helpers
```

Benefits of Using _ViewImports.cshtml

- **Reduced Redundancy:** Avoid repeating the same directives in multiple views.
- **Consistent Configuration:** Easily apply common settings across your views.
- **Improved Readability:** Keep your individual view files cleaner and more focused on their specific content.

Code of View:

```
@model YourProject.Models.Product
<!DOCTYPE html>
<html>
<head>
   <title>Product Details</title>
</head>
<body>
   <h1>@Model.Name</h1>
   </body>
</html>
```

In this example, the @model directive is not required in the view because it's already specified in the _ViewImports.cshtml file. Similarly, you don't need to include the @using directive for your model's namespace.

Important Considerations

- **Scoping:** Directives in _ViewImports.cshtml are scoped to the directory and its subdirectories.
- **Overriding:** Settings in a child folder's _ViewImports.cshtml file override those in parent folders.
- **Flexibility:** Use multiple _ViewImports.cshtml files in different folders to manage settings at a more granular level.

Key Points to Remember

- **Centralization:** Use _ViewImports.cshtml to centralize common view directives and settings.
- **DRY Principle:** Adhere to the Don't Repeat Yourself (DRY) principle by avoiding redundant code.
- **Scoping and Overriding:** Understand how the cascading nature of _ViewImports.cshtml files works.

Shared Views

In ASP.NET Core MVC, a shared view is a view file (.cshtml) that is not tied to a specific controller or action. These views typically reside in the Views/Shared folder and are designed to be reused across different controllers or even multiple projects.

Why Use Shared Views?

- **Reduce Duplication:** Avoid writing the same code repeatedly for common UI elements across your application.
- **Consistent UI:** Maintain a unified look and feel across different pages and sections of your site.
- **Simplified Maintenance:** Update the shared view once, and the changes automatically apply to all views that use it.

How Shared Views Work

1. **Location:** By default, ASP.NET Core MVC looks for shared views in the Views/Shared folder. You can also create subfolders within Views/Shared to further organize your shared views.

2. **Naming:** Name your shared views in a way that reflects their purpose or content. Common examples include:
    - _Layout.cshtml: The main layout for your application's pages.
    - _PartialName.cshtml: Smaller, reusable components (partial views) that can be embedded in other views.

3. **Rendering:** You can render a shared view using the following approaches:
    - **PartialView():** Renders a partial view.
    - **View():** Can be used to render a shared view directly, but it's more common to use PartialView() for partial views and reserve View() for full pages.

Code Example: Shared View All.cshtml

```
<!DOCTYPE html>
<html>
<head>
   <title>All Products</title>
   <meta charset="UTF-8" />
   <link href="~/StyleSheet.css" rel="stylesheet" />
</head>
<body>
   <div class="page-content">
      <h1>All Products</h1>
      </div>
</body>
</html>
```

This shared view (All.cshtml) can be reused by multiple controllers to display the "All Products" page with the same structure and styling.

Code Example: Controllers Using the Shared View

```
// HomeController.cs
[Route("home/all-products")]
public IActionResult All()
{
   return View(); // Will look for Views/Home/All.cshtml first, then Views/Shared/All.cshtml
}

// ProductsController.cs
[Route("products/all")]
public IActionResult All()
{
   return View(); // Will look for Views/Products/All.cshtml first, then Views/Shared/All.cshtml
}
```

In both HomeController and ProductsController, the All action method returns a ViewResult. ASP.NET Core will first look for a view named All.cshtml within the specific controller's view folder (e.g., Views/Home or Views/Products). If it doesn't find it there, it will look for it in the Views/Shared folder.

Important Considerations

- **Naming Convention:** Prefix the names of partial views with an underscore (e.g., _ProductCard.cshtml) to distinguish them from full views.
- **Data Passing:** Use view models to pass data to shared views, ensuring that the view has access to all the information it needs to render correctly.
- **Flexibility:** You can override shared views within specific controllers if you need to customize them for a particular use case.

Shared views are a cornerstone of maintainable and scalable ASP.NET Core MVC applications. By leveraging their reusability, you can significantly streamline your development process and ensure a consistent user experience across your website or application.

Key Points to Remember

1. Views

- **Purpose:** Render the user interface (UI) of your MVC application.
- **Razor View Engine:** Default engine for combining HTML markup with C# code.
- **View File Location:** Typically found in the Views folder, organized by controller.
- **View Selection:** Controllers return ViewResult or other action results to render views.
- **Model Binding:** Views often receive data from controllers (the model) for dynamic rendering.

2. Razor Syntax Essentials

- **Code Blocks (@{...}):** For multi-line C# code.
- **Expressions (@variable or @method()):** Embed C# expressions.
- **Literals (@: or <text>):** Output raw text (no HTML encoding).
- **Control Flow:** Use @if, @else, @switch, @foreach, @for for conditional and iterative logic.
- **Comments:** @*...*@ for server-side comments.

3. Local Functions

- **Purpose:** Encapsulate reusable C# functions within Razor views.
- **Syntax:** Define within @functions { ... }.
- **Benefits:** Improved organization, readability, and reusability within a view.

4. Html.Raw()

- **Purpose:** Renders raw HTML content without encoding.
- **Warning:** Use with caution due to potential XSS vulnerabilities.
- **Alternatives:** Consider partial views or view components for dynamic HTML generation.

5. ViewData and ViewBag

- **Purpose:** Pass data from controllers to views.
- **ViewData:** Dictionary-based (ViewData["key"]).
- **ViewBag:** Dynamic wrapper around ViewData (ViewBag.key).
- **Recommendation:** Prefer strongly typed views or view models for better type safety and maintainability.

6. Strongly Typed Views

- **Purpose:** Associate a view with a specific model class using the @model directive.
- **Benefits:** Type safety, IntelliSense, and improved maintainability.
- **View Models:** Combine data from multiple models into a single view model class.

7. _ViewImports.cshtml

- **Purpose:** Centralize common @using, @addTagHelper, @inject, and other directives.
- **Location:** In the root Views folder or subfolders.
- **Benefits:** Reduce redundancy, ensure consistency, and improve readability.

8. Shared Views

- **Purpose:** Reusable view components stored in the Views/Shared folder.
- **Types:**
    - **Layout Pages:** Define the overall page structure.
    - **Partial Views:** Reusable chunks of UI elements.
- **Rendering:** Use PartialView() to render partial views, View() for full views.

Interview Tips:

- **Razor Syntax:** Be ready to write examples of different Razor syntax elements and explain their use cases.
- **View Logic:** Discuss when it's appropriate to use local functions or view models in your views.
- **Security:** Emphasize the importance of security (input validation, avoiding Html.Raw() for untrusted content) when working with views.
- **Best Practices:** Demonstrate your knowledge of using view models, shared views, and _ViewImports.cshtml for cleaner and more maintainable code.