ASP.NET Core - True Ultimate Guide

Section 10: Partial Views

Partial Views

In ASP.NET Core MVC, a partial view is a reusable chunk of Razor markup (.cshtml) that can be embedded within other views. They are designed to encapsulate specific UI elements, such as lists, forms, or widgets, allowing you to avoid repeating code and maintain a more organized structure.

Why Use Partial Views?

- **Reusability:** A single partial view can be used in multiple views, saving development time and reducing the potential for inconsistencies.
- **Modularity:** Partial views help break down complex views into smaller, more manageable components, making your code easier to read and maintain.
- **Dynamic Content:** You can pass data to partial views, making them dynamic and adaptable to different contexts.

Notes

- **Naming Convention:** Partial views are typically named with a leading underscore (e.g., _ListPartialView.cshtml). This convention helps distinguish them from full views.
- **Location:** By default, partial views are located in the Views/Shared folder or in the same folder as the view that uses them. You can override the default location by specifying the full path.
- **Rendering:** You can render partial views in your main views using:
  - **Tag Helpers:** <partial name="_ListPartialView" /> or <partial name="_ListPartialView" model="yourModel" />
  - **HTML Helper:** @await Html.PartialAsync("_ListPartialView") or @await Html.PartialAsync("_ListPartialView", yourModel)

Code Example

_ListPartialView.cshtml (Partial View)

```
<div class="list-container">
  @{
    //ViewBag.ListTitle = "Updated"; // Not recommended
  }


  <h3>@ViewBag.ListTitle</h3>
  <ul class="list">
  @foreach (string item in ViewBag.ListItems)
  {
    <li>@item</li>
  }
  </ul>
</div>
```

ViewBag: This example uses ViewBag to pass the ListTitle and ListItems data to the partial view. However, it is generally recommended to use a strongly typed model to pass data to partial views for better type safety and maintainability.

Views/Home/Index.cshtml (Main View)

```
<h1>Home</h1>


<partial name="_ListPartialView" />


@{
    var myViewData = new ViewDataDictionary(ViewData); // Create a new ViewDataDictionary
    myViewData["ListTitle"] = "Countries";
    myViewData["ListItems"] = new List<string>() { "USA", "Canada", "Japan", "Germany", "India" };
}
<div class="box">
    <partial name="_ListPartialView" view-data="myViewData" />
</div>


<h3>ListTitle in View: @ViewData["ListTitle"]</h3>
```

This main view renders the _ListPartialView twice:

1. **First Rendering:**
   - The @ViewBag.ListTitle value in the partial view will be null (as it wasn't explicitly set before the partial view is rendered).
   - The @ViewBag.ListItems will also be null and the loop will not execute.
   - A new ViewDataDictionary named myViewData is created.

2. **Second Rendering:**
   - The myViewData dictionary is used to pass the ListTitle ("Countries") and ListItems (a list of countries) to the partial view.
   - The partial view will display the list of countries because ViewBag.ListItems is not null.
   - After the partial view has rendered, the ViewData["ListTitle"] still refers to the original value ("Asp.Net Core Demo App") set in the controller. This is because the myViewData dictionary is a separate instance of ViewDataDictionary.

Views/Home/About.cshtml (Main View)

```
<h1>About</h1>


<h2>About company here</h2>
<p>Lorem Ipsum is simply dummy text...</p>
@{
  await Html.RenderPartialAsync("_ListPartialView");
}
```

This view also renders _ListPartialView using the Html.RenderPartialAsync method. The result is the same as the first render in Index.cshtml.

Notes

- **Reusability:** Partial views are essential for DRY (Don't Repeat Yourself) development in your views.
- **Flexibility:** You can pass data to partial views to make them dynamic.
- **Rendering Options:** Use tag helpers or HTML helpers to render partial views.
- **Data Sharing:** Ensure that your partial views have access to the necessary data using view models.

- **Organization:** Partial views contribute to a well-structured and maintainable codebase.
- **View Reusability:** The same partial view can be rendered multiple times on the same page, each time with a different set of data.

Strongly Typed Partial Views

A strongly typed partial view, just like a strongly typed view, is associated with a specific model class using the @model directive. This means that the partial view has direct access to the properties and methods of that model, providing compile-time type checking and IntelliSense support within the partial view.

Benefits of Strongly Typed Partial Views

- **Type Safety:** Catches errors during development due to mismatched data types or property names.
- **IntelliSense:** Provides autocompletion and suggestions for model properties, leading to faster and more accurate coding.
- **Refactoring:** Makes it easier to update partial views when your model changes.

How to Use Strongly Typed Partial Views

1. **Create a Model Class:** Define a class that represents the data structure you want to pass to your partial view.
2. **Use @model in the Partial View:** Add the @model directive at the top of your partial view, specifying the model class: @model YourNamespace.YourModel.
3. **Pass the Model:** When rendering the partial view from your main view, pass an instance of your model class using the model attribute of the <partial> tag helper or the model parameter of the Html.PartialAsync method.

Code Example

ListModel.cs (Model)

```
namespace PartialViewsExample.Models
{
    public class ListModel
    {
        public string ListTitle { get; set; } = "";

        public List<string> ListItems { get; set; } = new List<string>();

    }

}
```

This model will be used to represent the data for the list that's rendered in the partial view.

_ListPartialView.cshtml (Strongly Typed Partial View)

```
@model ListModel // Specify the model type


<div class="list-container">

    <h3>@Model.ListTitle</h3>

    <ul class="list">

    @foreach (string item in Model.ListItems)

    {

        <li>@item</li>

    }

    </ul>

</div>
```

- **@model ListModel:** This directive indicates that this partial view expects a model of type ListModel.
- **Access Model Properties:** You can directly access properties of the model object using @Model.PropertyName.

Index.cshtml (Main View)

```
@using PartialViewsExample.Models


<h1>Home</h1>


@{

    ListModel listModel = new ListModel();
    listModel.ListTitle = "Countries";

    listModel.ListItems = new List<string>() { "USA", "Canada", "Japan", "Germany", "India" };

}

<partial name="_ListPartialView" model="listModel" />
```

- **Creating the Model:** An instance of ListModel is created and populated with data.
- **Passing the Model:** The model attribute in the <partial> tag helper is used to pass the listModel to the _ListPartialView partial view.

About.cshtml (Main View)

```
@{
    ListModel listModel = new ListModel();

    listModel.ListTitle = "Programming Languages";

    listModel.ListItems = new List<string>()

    {

    "Java",

    "C#",

    "Python"

    };


    await Html.RenderPartialAsync("_ListPartialView", listModel);

}
```

Same as the Index.cshtml, but using the Html.RenderPartialAsync HTML helper.

Notes

- **Strongly Typed:** Use @model to specify the type of data expected by the partial view.
- **Pass the Model:** Pass an instance of your model class when rendering the partial view.
- **Type Safety and IntelliSense:** Benefit from compile-time checking and code completion when working with model properties.
- **Best Practice:** Strongly typed partial views promote clean, maintainable, and less error-prone code.

PartialViewResult

In ASP.NET Core MVC, a PartialViewResult is a specific type of ActionResult designed for returning partial views from your controller actions. Partial views, as you know, are reusable chunks of Razor markup (.cshtml) that can be embedded within other views. However, instead of being rendered as a full page, they're intended to be a fragment of HTML that can be inserted into a larger view.

Why Use PartialViewResult?

- **Dynamic Content Delivery:** You can use PartialViewResult to load content dynamically into your main view using AJAX or other client-side techniques.
- **Separation of Concerns:** This result type helps keep your controller actions focused on returning data and leaves the rendering of that data to the view.
- **Simplified Testing:** Since PartialViewResult doesn't involve rendering a complete page, it's often easier to unit test your action methods that return partial views.

Creating a PartialViewResult

In your controller actions, you can return a PartialViewResult in a couple of ways:

1. **Direct Instantiation:**

```
return new PartialViewResult

{

  ViewName = "_ListPartialView", // Name of the partial view

  ViewData = new ViewDataDictionary(ViewData, model) // Pass the model data

};
```

2. **Using the PartialView() Helper Method:**

```
return PartialView("_ListPartialView", model); // Much simpler way
```

The PartialView() method is a shorthand provided by the Controller base class for conveniently creating a PartialViewResult.

Code Example

Views/Home/Index.cshtml (Main View)

```
<button class="button button-blue-back" type="button" id="button-load">Load Programming
Languages</button>

<div class="programming-languages-content">


</div>


<script>
  document.querySelector("#button-load").addEventListener("click", async function() {
    var response = await fetch("programming-languages");
    var languages = await response.text();
    document.querySelector(".programming-languages-content").innerHTML = languages;
  });
</script>
```

This view contains a button, a div, and some JavaScript to load the partial view's content when the user clicks a button using fetch function.

HomeController.cs (Controller)

```
[Route("programming-languages")]
public IActionResult ProgrammingLanguages()
{
  ListModel listModel = new ListModel() {
    ListTitle = "Programming Languages List",
    ListItems = new List<string>() { "Python", "C#", "Go" }
  };

  return PartialView("_ListPartialView", listModel);
}
```

In this action method:

1. **Data Preparation:** A ListModel object is created and populated with data for the list (title and items).
2. **Partial View Returned:** The PartialView() method is used to return a PartialViewResult. The method takes two arguments:
    - "_ListPartialView": The name of the partial view file (located in Views/Shared by default).
    - listModel: The model data to pass to the partial view.
    - When this action method is called via the URL /programming-languages by the javascript code on the page, it returns the partial view along with the data that is then injected into the div with class programming-languages-content by the javascript.

Important Points:

- **AJAX and Dynamic Loading:** PartialViewResult is frequently used in conjunction with AJAX to load content dynamically without full page refreshes.
- **View Model (Best Practice):** Always try to use a view model to pass data to your partial views for better type safety and maintainability.
- **Caching:** Consider using output caching on your partial views to improve performance if they render data that doesn't change frequently.

Key Points to Remember

Partial Views: Reusable View Components

- **Purpose:** Encapsulate reusable UI elements or chunks of Razor markup (.cshtml) to avoid repetition.
- **Benefits:**
    - Increased code reusability
    - Improved code organization and maintainability
    - Dynamic content generation
- **Naming Convention:** Prefixed with an underscore (e.g., _PartialName.cshtml).
- **Location:** Typically in Views/Shared or alongside the main view.
- **Rendering Options:**

    - **Tag Helpers:** <partial name="_PartialName" /> or <partial name="_PartialName" model="yourModel" />
    - **HTML Helper:** @await Html.PartialAsync("_PartialName", model)

ViewData in Partial Views

- **Purpose:** Pass data to partial views from the parent view or controller.
- **Usage:**
    - In the parent view or controller: ViewData["key"] = value
    - In the partial view: @ViewData["key"]
- **Caveat:** ViewData is not strongly typed; be careful with type casting and null checks.

Strongly Typed Partial Views

- **Purpose:** Associate a partial view with a specific model class using the @model directive.
- **Benefits:**
    - Type safety: Catches errors during development.
    - IntelliSense: Code completion for model properties.
    - Maintainability: Easier updates when models change.
- **How to Use:**

1. Create a model class.
2. Use @model YourModel in the partial view.
3. Pass a model instance when rendering: <partial model="yourModelInstance" /> or @await Html.PartialAsync("_PartialName", yourModelInstance)

PartialViewResult

- **Purpose:** Return a partial view from a controller action (often for AJAX requests).
- **Creation:**
    - return PartialView("_PartialName", model); (preferred)
    - return new PartialViewResult { ViewName = "_PartialName", ViewData = ... };