ASP.NET Core - True Ultimate Guide

Section 11: View Components

View Components

View Components are self-contained, reusable UI building blocks in ASP.NET Core MVC. They are designed to encapsulate rendering logic that's more complex than what you'd typically put in a partial view but doesn't warrant the complexity of a full controller and view.

Purpose

- **Encapsulate Complexity:** Group related UI rendering logic into a cohesive unit.
- **Reusability:** Use View Components across multiple views to avoid code duplication.
- **Testability:** Easier to unit test View Components due to their self-contained nature.
- **Rendering Logic:** Ideal for dynamic widgets, navigation menus, login forms, shopping cart summaries, or any UI element that involves fetching data or performing logic before rendering.

Best Practices

- **Naming:** View Component classes should end with the suffix ViewComponent (e.g., ProductListViewComponent).
- **Structure:** Place View Components in the ViewComponents folder. The associated view file should be placed in Views/Shared/Components/{ViewComponent Name}/{View Name}.cshtml.
- **Data:** Pass data to the View Component using a strongly typed model or view model.
- **Asynchronous:** Use asynchronous methods (InvokeAsync) to avoid blocking threads when performing data access or other I/O operations.
- **Simplicity:** Keep the View Component's logic focused on rendering the UI element. Avoid complex business logic within the component.

Things to Avoid

- **Overuse:** Don't use View Components for simple UI elements that can be handled by partial views.
- **Tight Coupling:** Avoid tightly coupling View Components to specific controllers or actions. Make them reusable across different parts of your application.
- **Complex Logic:** View Components should not be responsible for handling business logic. Instead, delegate that to your models or services.

- **Direct Database Access:** Avoid directly accessing your database from within a View Component. Use services for data access.

When to Use View Components

- **Complex UI Elements:** When a UI element requires more complex rendering logic than a simple partial view.
- **Data-Driven Elements:** When you need to fetch data or perform some computation before rendering the UI element.
- **Reusable Widgets:** When you want to create a reusable widget that can be used in different parts of your application.

How to Implement View Components

1. **Create a View Component Class:** Derive from ViewComponent and implement an Invoke or InvokeAsync method.
2. **Create a View:** Create a Razor view file (.cshtml) within the Views/Shared/Components/{ViewComponent Name} folder.
3. **Invoke in Your View:** Use the @await Component.InvokeAsync("ViewComponent Name", arguments) helper in your main view to render the View Component.

Code

```
// GridViewComponent.cs (View Component)
public class GridViewComponent : ViewComponent
{
   public async Task<IViewComponentResult> InvokeAsync()
   {
      // Data for the view component (could come from a database, service, etc.)
      PersonGridModel model = new PersonGridModel()
      {
         GridTitle = "Persons List",
         Persons = new List<Person>()
         {
            new Person() { PersonName = "John", JobTitle = "Manager" },
            // more people
```

```
        }
    };

    // ViewBag is not ideal; prefer using a strongly typed model
    ViewData["Grid"] = model;

    return View("Sample"); // This will look for the view in
Views/Shared/Components/Grid/Sample.cshtml
  }
}

// Views/Home/Index.cshtml (Main View)
@await Component.InvokeAsync("Grid") // Invokes the "Grid" view component

// Views/Home/About.cshtml
@await Component.InvokeAsync("Grid")

<vc:grid></vc:grid> // alternative syntax (not preferred, as it is not strongly typed)
```

Explanation:

- The GridViewComponent class is a view component that fetches a list of people and passes it to the "Sample" partial view along with a grid title using ViewBag.
- Views/Home/Index.cshtml and Views/Home/About.cshtml invoke the Grid view component using @await Component.InvokeAsync("Grid") (or, less ideally, the vc:grid tag helper), rendering the person list in a grid format.

Notes

- **Purpose:** Encapsulate complex, reusable UI rendering logic.
- **Naming:** ViewComponent classes end with ViewComponent.
- **Location:** ViewComponents folder for classes, Views/Shared/Components/{ViewComponent Name} for views.
- **Data Passing:** Use strongly typed models or view models (preferable over ViewBag).
- **Rendering:** @await Component.InvokeAsync("ViewComponent Name", arguments)

Strongly Typed View Components

Just like strongly typed views, strongly typed view components are associated with a specific model class using the @model directive. This model class, often referred to as a "view model," provides a structured way to pass data from the view component to its corresponding view. The primary benefit is enhanced type safety and improved code maintainability.

Benefits of Strongly Typed View Components

- **Type Safety:** Catches errors during development due to mismatched data types or property names, as the Razor engine enforces type checking based on your model class.
- **IntelliSense:** Enhances productivity by providing code completion and suggestions for model properties directly within the Razor view. This leads to fewer typos and faster development.
- **Refactoring:** Simplifies the process of updating views when your model changes. Since the view is strongly typed, renaming or modifying properties in the model will automatically update the references in the view.
- **Clarity and Readability:** Makes your code more self-documenting by clearly defining the data structure expected by the view component.

How to Implement Strongly Typed View Components

1. **Create a View Model:** Define a class that represents the data you want to pass to your view component. This class should contain all the necessary properties that the view component will use to render its output.
2. **Use @model in the View:** In your view component's Razor view file (e.g., Default.cshtml), use the @model directive to specify the view model class.
3. **Return the Model from InvokeAsync:** In your view component's InvokeAsync method, create an instance of your view model, populate it with the required data, and return it using View(model).

Code

PersonGridModel.cs (View Model)

```
public class PersonGridModel
{
   public string GridTitle { get; set; }
```

```
    public List<Person> Persons { get; set; }
}
```

This model will be used to represent the data passed from the view component to the view.

GridViewComponent.cs

```
// ViewComponent
public class GridViewComponent : ViewComponent
{
    public async Task<IViewComponentResult> InvokeAsync()
    {
        PersonGridModel personGridModel = new PersonGridModel()
        {
            GridTitle = "Persons List",
            Persons = new List<Person>() {
                new Person() { PersonName = "John", JobTitle = "Manager" },
                new Person() { PersonName = "Jones", JobTitle = "Asst. Manager" },
                new Person() { PersonName = "William", JobTitle = "Clerk" },
            }
        };
        return View("Sample", personGridModel);
    }
}
```

The method now creates a PersonGridModel object, populates it, and passes it as the second argument of the View() method.

Views/Shared/Components/Grid/Sample.cshtml

```
@model PersonGridModel

<div class="box">
    <h3>@Model.GridTitle</h3>
    <table class="table w-100">
```

```
    <thead>
      <tr>
        <th>Sl. No</th>
        <th>Name</th>
      </tr>
    </thead>
    <tbody>
      @foreach (Person person in Model.Persons)
      {
        <tr>
          <td>@person.PersonName</td>
          <td>@person.JobTitle</td>
        </tr>
      }
    </tbody>
  </table>
</div>
```

- **@model PersonGridModel:** This specifies that the view is strongly typed and expects an object of type PersonGridModel.
- **Accessing Properties:** The view directly accesses properties of the Model object like Model.GridTitle and Model.Persons.

Notes

- **Strongly Typed:** Use @model in the view component's view file to specify the view model type.
- **Pass the Model:** Pass an instance of your view model class when returning the view from the InvokeAsync method: return View("ViewName", model);
- **Naming:** View components should be named with the suffix "ViewComponent" (e.g., ProductListViewComponent).
- **Location:** View component classes go in a ViewComponents folder, and their associated views go in Views/Shared/Components/{ViewComponent Name}/.

View Components with Parameters

While view components can function without parameters, passing parameters to them significantly enhances their flexibility and reusability. Parameters allow you to customize the behavior and output of a view component based on data provided by the calling view or controller. This makes view components more dynamic and adaptable to different scenarios within your application.

How to Pass Parameters to View Components

1. **Define Parameters in InvokeAsync:** In your view component class, define the parameters that you want to receive within the InvokeAsync (or Invoke) method. These parameters will become part of the method's signature.
2. **Pass Arguments When Invoking:** When invoking the view component using @await Component.InvokeAsync("ViewComponent Name", arguments) or the <vc:grid> tag helper, pass an anonymous object containing the parameter values. The property names in the anonymous object must match the parameter names in the InvokeAsync method.

Code

GridViewComponent.cs

```
public class GridViewComponent : ViewComponent
{
   public async Task<IViewComponentResult> InvokeAsync(PersonGridModel grid)
   {
      return View("Sample", grid);
   }
}
```

Parameter: The InvokeAsync method now takes a parameter of type PersonGridModel. This means the view component expects to receive a model containing grid data when it's invoked.

Views/Home/Index.cshtml

```
@{
   PersonGridModel personGridModel = new PersonGridModel() { /* ... (data initialization) ... */ };
}
```

```
@await Component.InvokeAsync("Grid", new { grid = personGridModel })


@{
   PersonGridModel personGridModel2 = new PersonGridModel() { /* ... (different data initialization) ... */ };
}
@await Component.InvokeAsync("Grid", new { grid = personGridModel2 })
```

Two instances of PersonGridModel are created and passed as an argument to the view component when invoking it.

Views/Home/About.cshtml

```
@{
   PersonGridModel personGridModel = new PersonGridModel() { /* ... (data initialization) ... */ };
}


<vc:grid grid="personGridModel"></vc:grid>
```

- **Tag Helper Usage:** The <vc:grid> tag helper is used here to invoke the view component. The grid attribute is set to the personGridModel, passing the model data to the view component.

Benefits of Strongly Typed View Components with Parameters

- **Type Safety:** Ensures that you pass the correct data types to the view component, preventing runtime errors due to type mismatches.
- **IntelliSense:** You get code completion suggestions for parameter names and model properties, making development faster and more efficient.
- **Flexibility:** Customize the view component's behavior based on the provided parameters, making it more adaptable to different scenarios.
- **Maintainability:** Easier to refactor and modify the view component and its usage in your views.

Notes

- **Define Parameters:** Clearly define the parameters your view component expects in its InvokeAsync method.
- **Pass Parameters:** When invoking the view component, pass an anonymous object with properties matching the parameter names and types.
- **Strongly Typed:** Use a view model (like PersonGridModel) to pass structured data to your view component and benefit from type safety and IntelliSense.

ViewComponentResult

While view components are primarily invoked from within views, the ViewComponentResult class allows you to directly return a rendered view component from a controller action. This provides a powerful way to integrate view components with your MVC controller logic and leverage their reusable rendering capabilities.

Purpose

- **Dynamic View Component Loading:** Use ViewComponentResult to load view components on demand based on the controller's logic or data. This is particularly useful for rendering complex UI elements that depend on specific data or conditions.
- **Separation of Concerns:** Keep your controller actions focused on handling requests and data retrieval, while delegating the rendering of specific UI components to view components.
- **API-like Endpoints:** Create API-like endpoints that return HTML fragments (view components) instead of JSON or XML data. This can be helpful for building hybrid applications where you need to combine server-side rendering with client-side JavaScript.

Creating a ViewComponentResult

In your controller actions, you can return a ViewComponentResult using the ViewComponent() helper method:

return ViewComponent("ViewComponent Name", arguments);

- **ViewComponent Name:** The name of the view component you want to invoke. This should match the class name of your view component (without the "ViewComponent" suffix).
- **arguments (Optional):** An anonymous object containing the parameters you want to pass to the view component's InvokeAsync method.

Code

```
// HomeController.cs (Controller)
[Route("friends-list")]
public IActionResult LoadFriendsList()
{
   PersonGridModel personGridModel = new PersonGridModel()
   {
      GridTitle = "Friends",
      Persons = new List<Person>()
      {
         // ... (list of friends) ...
      }
   };

   return ViewComponent("Grid", new { grid = personGridModel });
}
```

In this code:

1. **Data Preparation:** The personGridModel object is created and populated with data for the list of friends.
2. **ViewComponentResult:** The ViewComponent() method is used to return a ViewComponentResult. It takes two arguments:
    - "Grid": This indicates that we want to invoke the GridViewComponent.
    - new { grid = personGridModel }: This anonymous object passes the personGridModel as the grid parameter to the InvokeAsync method of the GridViewComponent.

How It Works

1. **Request:** A request to /friends-list triggers the LoadFriendsList action.
2. **View Component Invocation:** The ViewComponentResult returned from the action causes ASP.NET Core MVC to locate and invoke the GridViewComponent.

3. **View Component Execution:** The InvokeAsync method within the GridViewComponent receives the personGridModel and uses it to render the "Sample" partial view.
4. **Response:** The rendered output of the GridViewComponent (the HTML for the grid of friends) is returned as the HTTP response.

Notes

- **Purpose:** Render view components directly from controller actions.
- **Flexibility:** Load view components dynamically based on controller logic.
- **API-Style Endpoints:** Useful for creating endpoints that return HTML fragments.
- **Syntax:** Use ViewComponent("ViewComponent Name", arguments) to create a ViewComponentResult.
- **Parameters:** Pass parameters to the view component using an anonymous object.

Key Points to Remember

1. View Components: Modular UI Components

- **Purpose:** Encapsulate reusable UI rendering logic more complex than partial views.
- **Benefits:**
    - Modularization and reusability
    - Separation of concerns (UI logic from controllers)
    - Improved testability
- **Structure:**
    - Class: Inherits from ViewComponent (e.g., ProductListViewComponent).
    - View: Razor file in Views/Shared/Components/{ViewComponent Name}/{View Name}.cshtml.

2. Methods

- **Invoke or InvokeAsync:** The main method for your component's logic.
    - Returns IViewComponentResult.

- Can take parameters for customization.
- Use async for asynchronous operations.

3. Invoking View Components

- **In Views:**
  - @await Component.InvokeAsync("ViewComponent Name", arguments)
  - <vc:view-component-name></vc:view-component-name> (Tag Helper syntax, less preferred)
- **In Controllers:**
  - return ViewComponent("ViewComponent Name", arguments);

4. Strongly Typed View Components

- **Purpose:** Pass data to the view using a strongly typed model.
- **Benefits:** Type safety, IntelliSense, better maintainability.
- **How to Use:**
  1. Create a view model class.
  2. Use @model YourViewModel in the view component's view file.
  3. Return the view model from InvokeAsync: return View(model);

5. Passing Parameters

- **InvokeAsync Parameters:** Define parameters in the InvokeAsync method signature.
- **Invocation Arguments:** Pass arguments as an anonymous object when invoking the component.

Example:

```
// ViewComponent
public async Task<IViewComponentResult> InvokeAsync(int categoryId)
{
   var products = _productService.GetProductsByCategory(categoryId);
   return View(products);
```

```
}

// View
@await Component.InvokeAsync("ProductList", new { categoryId = 5 })
```

7. Best Practices

- **Naming:** Use the suffix "ViewComponent" for class names.
- **Folder Structure:**
    - View components in the ViewComponents folder.
    - Views in Views/Shared/Components/{ViewComponent Name}/.
- **Strongly Typed:** Use strongly typed view models.
- **Async Operations:** Use InvokeAsync for asynchronous tasks.
- **Limit Logic:** Keep logic focused on UI rendering, not business operations.
- **Avoid Direct Database Access:** Use services for data access.

8. Things to Avoid

- **Overuse:** Don't use for simple UI elements.
- **Tight Coupling:** Keep them independent of specific controllers.
- **Complex Logic:** Avoid extensive business logic within components.

9. When to Use

- **Complex UI Elements:** When a partial view is not enough.
- **Data-Driven Elements:** Dynamic content based on data/logic.
- **Reusable Widgets:** To create reusable UI components.