**Page Contents**

* [Shared Folder](https://www.yogihosting.com/aspnet-core-views/#shared-folder)
* [Shared View](https://www.yogihosting.com/aspnet-core-views/#shared-view)
* [How MVC search the View file](https://www.yogihosting.com/aspnet-core-views/#view-search)
  + [Views for Areas](https://www.yogihosting.com/aspnet-core-views/#area-views)
* [Razor Layout Page ‘\_Layout.cshtml’](https://www.yogihosting.com/aspnet-core-views/#layout)
  + [Sections inside Layout](https://www.yogihosting.com/aspnet-core-views/#sections)
  + [Optional Sections](https://www.yogihosting.com/aspnet-core-views/#optional-sections)
* [Inline Code in a View](https://www.yogihosting.com/aspnet-core-views/#inline-code)
* [Partial Views](https://www.yogihosting.com/aspnet-core-views/#partial-views)
  + [Strongly Typed Partial View](https://www.yogihosting.com/aspnet-core-views/#strongly-typed-partial-views)
* [View Components](https://www.yogihosting.com/aspnet-core-views/#view-components)
  + [Return Types of View Components](https://www.yogihosting.com/aspnet-core-views/#view-components-return-types)
  + [View Component Returning ‘ContentViewComponentResult’](https://www.yogihosting.com/aspnet-core-views/#ContentViewComponentResult)
  + [View Component Returning ‘HtmlContentViewComponentResult’](https://www.yogihosting.com/aspnet-core-views/#HtmlContentViewComponentResult)
  + [View Component Returning Partial View](https://www.yogihosting.com/aspnet-core-views/#view-components-returning-partial-views)
  + [Providing values to a View Component from Parent View](https://www.yogihosting.com/aspnet-core-views/#providing-values-to-view-component)
* [Asynchronous View Component](https://www.yogihosting.com/aspnet-core-views/#asynchronous-view-component)
* [Download Source Codes](https://www.yogihosting.com/aspnet-core-views/#download)

## Shared Folder

Create a Shared folder inside the Views folder. Inside this Shared folder you can put the items that are shared in the application. Note that:

* 1. View that are shared with other portions of the application.
* 2. Layout files contains common portions of the UI.

## Creating a Shared View

Shared Views are kept inside the Views ➤ Shared folder. They can be shared among different controllers in the application. Let me show how to work with them.

Go to Home Controller and create a new Action method called CallSharedView as shown below:

*Learn*[*How to use Globalization and localization in ASP.NET Core to create a Multilingual site*](https://www.yogihosting.com/how-to-use-globalization-and-localization-in-asp-net-core/)

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Threading.Tasks;  using Microsoft.AspNetCore.Mvc;  using UnderstandingControllersViews.Models;    namespace UnderstandingControllersViews.Controllers  {      public class HomeController : Controller      {          // removed for clarity          public IActionResult CallSharedView()          {              return View();          }      }  } |

Also create the same action CallSharedView on another controller whose name is ExampleController.

|  |  |
| --- | --- |
|  | using System;  using System.Collections.Generic;  using System.Linq;  using System.Threading.Tasks;  using Microsoft.AspNetCore.Http;  using Microsoft.AspNetCore.Mvc;    namespace UnderstandingControllersViews.Controllers  {      public class ExampleController : Controller      {          // removed for clarity          public IActionResult CallSharedView()          {              return View();          }      }  } |

These are the 2 action methods. Now I will create only **1 View for both of these action methods**. To do this I will create a new View inside the Views ➤ Shared folder and simple name it CallSharedView. In this way this will be shared by these 2 actions.

So create a CallSharedView inside the Views ➤ Shared folder having the code shown below:

|  |  |
| --- | --- |
|  | @{      Layout = null;  }    <!DOCTYPE html>    <html>  <head>      <meta name="viewport" content="width=device-width" />      <title>CallSharedView</title>  </head>  <body>      <h1>Displaying from Shared View</h1>  </body>  </html> |

Run your application and go to the following URL to invoke these 2 actions:

|  |
| --- |
| 1. http://localhost:63657/Home/CallSharedView  2. http://localhost:63657/Example/CallSharedView |

In both of these URL you will find the shared view getting invoked, as shown in the image below:

## How MVC search the View file

I previously explained the [Different types of View methods](https://www.yogihosting.com/aspnet-core-actions/) that tell MVC to render a particular View. The question arises how MVC searches the View file? The answer for this is MVC looks for the following 2 locations:

|  |  |
| --- | --- |
| 1  2 | /Views/<ControllerName>/<ViewName>.cshtml  /Views/Shared/<ViewName>.cshtml |

For example consider the following Action method List of the AdminController:

|  |  |
| --- | --- |
|  | using System;  using System.Collections.Generic;  using System.Linq;  using System.Threading.Tasks;  using Microsoft.AspNetCore.Mvc;    namespace UnderstandingControllersViews.Controllers  {      public class AdminController : Controller      {          public IActionResult List()          {              return View();          }      }  } |

See the code of the line of the action method – return View(). I am telling MVC to return and render a View. So it searches for the View files and looks in 2 locations:

* 1. /Views/Admin/List.cshtml => If it finds the file here then it renders the View file. If not, then the point no 2 location is searched.
* 2. /Views/Shared/List.cshtml => If it finds the file here then it renders the View file. If not then View not found error is shown in the browser.

Run the project and go to the url – https://localhost:44339/Admin/List. You will see an exception message telling the view is not found.

See the below screenshot where I have marked the 2 locations where the view file is searched.

To remove this exceptions you can simply add the List view in either /Views/Admin or /Views/Shared folder.

Views for Areas

If the Controller is the part of an Area then the following 3 locations are searched:

|  |  |
| --- | --- |
| 3 | /Areas/<AreaName>/Views/<ControllerName>/<ViewName>.cshtml  /Areas/<AreaName>/Views/Shared/<ViewName>.cshtml  /Views/Shared/<ViewName>.cshtml |

Let me create an example for this. So first add a new folder called Areas in the root folder. Next create a new folder called Sales inside the Areas folder.

Now add a new [ASP.NET Core Controller](https://www.yogihosting.com/aspnet-core-controllers/) called AdminController inside the Areas ➤ Sales folder. Inside this controller add a new action called List whose code is shown below.

|  |  |
| --- | --- |
|  | using System;  using System.Collections.Generic;  using System.Linq;  using System.Threading.Tasks;  using Microsoft.AspNetCore.Mvc;    namespace UnderstandingControllersViews.Areas.Sales  {      [Area("Sales")]      public class AdminController : Controller      {          public IActionResult List()          {              return View("Show");          }      }  } |

Also add the area route in the Startup.cs class:

|  |
| --- |
| app.UseEndpoints(endpoints =>  {      endpoints.MapControllerRoute(          name: "MyArea",          pattern: "{area:exists}/{controller=Home}/{action=Index}/{id?}");        endpoints.MapControllerRoute(          name: "default",          pattern: "{controller=Home}/{action=Index}/{id?}");  }); |

Now initiate this action in the browser whose URL in my case is https://localhost:44339/Sales/Admin/List. When the action returns the MVC will search the View in the following 3 locations:

* 1. /Areas/Sales/Views/Admin/Show.cshtml => If it finds the file in this location then it renders the View file. If not, then the 2nd location is searched.
* 2. /Areas/Sales/Views/Shared/Show.cshtml => If it finds the view file here then it renders it. If not, then the 3rd location is searched.
* 3. /Views/Shared/Show.cshtml => If it finds the view file here then it renders the View. Else View not found error is shown in the browser.

See the below image to find the **error displayed** on the browser and the **View locations** checked by MVC.

You can also give the full path of the View file. In the following View() method I have told MVC my View file’s location as – */Views/Customer/List.cshtml*.

|  |  |
| --- | --- |
|  | public ViewResult List() {      return View("/Views/Customer/List");  } |

## Razor Layout Page ‘\_Layout.cshtml’

The **UI of an application** may contain common parts that remain the same throughout the application. Examples are – logo, header, left navigation bar, right bar or footer section. ASP.NET Core has a Layout view provided by the \_Layout.cshtml file. It contains common UI parts, so that you don’t have to write the same code in every page.

Cookie authentication uses HTTP cookies to authenticate client requests and maintain session information. Learn [How to Implement Cookie Authentication in ASP.NET Core](https://www.yogihosting.com/aspnet-core-cookie-authentication/)

The Layout View resides inside the Views ➤ Shared folder.

Right click on the Shared folder and select Add New Item. Then select the Razor Layout from the item list and click the Add button. Check the below image that tells this:

The \_Layout.cshtml will be created and will open for editing. It will have the following code by default.

|  |  |
| --- | --- |
|  | <!DOCTYPE html>    <html>  <head>      <meta name="viewport" content="width=device-width" />      <title>@ViewBag.Title</title>  </head>  <body>      <div>          @RenderBody()      </div>  </body>  </html> |

It is just an HTML page with few Razor blocks like @ViewBag.Title which puts the title from ViewBag variable, and @RenderBody() which renders all the content of child view which is not wrapped in a section. I will tell you about Sections in just a moment but before that let’s understand how layout works.

You can configure the application so that the \_Layout.cshtml file is applied by default. This is done by adding an Razor View Start file. This file is given the name \_ViewStart.cshtml and you have to add it in the Views folder. The contents of the \_ViewStart.cshtml is given below. You can see it specifies the Layout to be selected for all Views of the application by default.

|  |  |
| --- | --- |
| 1  2  3 | @{      Layout = "\_Layout";  } |

The View Start file is not needed by an application. So if this file is absent and you want your views to have a layout. In this case your views need to specify a layout by setting this property:

|  |  |
| --- | --- |
|  | @{      Layout = Layout;  } |

Similarly if your application has a View Start file but you do not want some views to have any layout file applied to them then in this case add @{ Layout = null; } on the Views.

Now add a new Action to the Home Controller and name it TestLayout. Its code is given below:

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Threading.Tasks;  using Microsoft.AspNetCore.Mvc;  using UnderstandingControllersViews.Models;    namespace UnderstandingControllersViews.Controllers  {      public class HomeController : Controller      {          // removed for clarity          public IActionResult TestLayout()          {              ViewBag.Title = "Welcome to TestLayout";              return View();          }      }  } |

In the Action I am setting the ViewBag.Title variable to a string Welcome to TestLayout. The layout will read this value and show it in the page title.

Also add the TestLayout view inside the Views ➤ Home folder with code given below:

|  |  |
| --- | --- |
| 1  2 | <h2>TestLayout starts here</h2>  <p>This text is coming from child view</p> |

Notice there is no @{ Layout = null; } on this view and also the application has a View Start file where I have specified the default layout for every view. So ASP.NET Core applies the html given on the \_Layout.cshtml file over this view.

In short the html of the TestLayout will be inserted inside of \_Layout.cshtml file where there is @RenderBody() code.

So the html of the TestLayout view when rendered in the browser will become:

<!DOCTYPE html>

<html>

<head>

<meta name="viewport" content="width=device-width" />

<title>@ViewBag.Title</title>

</head>

<body>

<div>

<h2>TestLayout starts here</h2>

<p>This text is coming from child view</p>

</div>

</body>

</html>

Run the application and open the URL of the TestLayout view which in my case is https://localhost:44339/Home/TestLayout, you will see it as shown in the below image:

## Sections inside Layout

**Sections** help to insert content of Views at specified locations inside the \_Layout.cshtml. Sections are defined using the @Section razor express followed by the name of the section.

I have added 3 sections that are Top, Script and Bottom to the TestLayout View as shown below:

|  |  |
| --- | --- |
| 14  15 | @section Top {      This is Top  }    <h2>TestLayout starts here</h2>    <p>This text is coming from child view</p>    @section Script {      <script src="https://code.jquery.com/jquery-3.2.1.min.js"></script>  }    @section Bottom {      This is Bottom  } |

Now on your **Layout file**, in order to render these sections, use the razor expression like @RenderSection. See the updated code of the \_Layout.cshtml file given below

|  |  |
| --- | --- |
|  | <!DOCTYPE html>    <html>  <head>      <meta name="viewport" content="width=device-width" />      <title>@ViewBag.Title</title>  </head>  <body>      @RenderSection("Top")        <div>          @RenderBody()      </div>        @RenderSection("Bottom")        @RenderSection("Script")  </body>  </html> |

Once you go to the https://localhost:44339/Home/TestLayout URL, you will find the display as shown by the image given below:

Are you deploying APIs on live server then you need to implement CORS else the browser will stop all client requests made to the APIs. Therefore read this tutorial – [How to Enable Cross-Origin Requests (CORS) in ASP.NET Core](https://www.yogihosting.com/aspnet-core-enable-cors/)

Clearly you will see the parts of the TestLayout view that are contained within the Top section comes first. Then comes those parts that are not contained within any section (these are inserted by the @RenderBody() helper tag).

Next you get the Bottom section’s contents and finally you will get the Script section’s content which adds the jQuery script at the very end.

You can define as many sections you want. But remember that it is necessary to defines all section in the view for which there is a @RenderBody() in the layout file, failing to do so results in an runtime error. The below topic of **Optional Sections** explains this with an example.

### Optional Sections

Let us add another section called Ads to the layout file:

|  |
| --- |
| <!DOCTYPE html>    <html>  <head>      <meta name="viewport" content="width=device-width" />      <title>@ViewBag.Title</title>  </head>  <body>      @RenderSection("Top")        <div>          @RenderBody()      </div>        @RenderSection("Bottom")        @RenderSection("Ads")        @RenderSection("Script")  </body>  </html> |

Do not define this section in the TestLayout.cshtml view and reload your page. You will see an unhandled exception saying:

InvalidOperationException: The layout page '/Views/Shared/\_Layout.cshtml' cannot find the section 'Ads' in the content page '/Views/Home/TestLayout.cshtml'

The reason is that **ASP.NET CORE** needs you to **define a section** in your View if your layout page has the @RenderSection helper for that section.

One way to remove this error is by adding the following section in the TestLayout view:

@section Ads {

}

There is also another way to resolve this error. Make that section in the layout page as **Optional Section**. With optional section ASP.NET Core MVC will **not give you an error even if that section is missing** from the view.

Simply pass **false**as the second argument to the @RenderSection helper method to make a **section as optional**.

So go to your Layout page and change @RenderSection("Ads") to @RenderSection("Ads", false) and that will make this section optional.

Rerun your application and this time everything will work perfectly.

## Inline Code in a View

You can put **inline codes in your View** to self-contained pieces of logic, such as if and foreach statements.

Example: Adding inline if-else block in the view:

|  |  |
| --- | --- |
|  | @if (DateTime.Now.DayOfWeek == DayOfWeek.Monday)  {      <p>It's a Boring Day</p>  }  else  {      <p>It's a Party Day</p>  } |

You can also have block of codes in your View such as:

|  |  |
| --- | --- |
|  | @{      string[] countries = new string[] { "England", "France", "Germany", "China" };      <ul>          @foreach (string country in countries)          {              <li>@country</li>          }      </ul>  } |

## Partial Views

**Partial Views** are separate View files whose content can be included in other Views. Example – If your view has contents that are repeated again and again then you can put that content in a partial view and load that contents from the partial view. This way you can prevent code duplication.

Only for very advanced usage – [Advanced Topics on Filters in ASP.NET Core – Filters with Dependencies, Global Filters, Order of Execution of Filters & Changing Filter Order](https://www.yogihosting.com/advanced-filters-topics-aspnet-core/)

### Creating and using Partial View

Partial Views are regular .cshtml files and differentiate with regular views only on the job which they do. Let us create a partial View inside the Views ➤ Home folder.

Right click the Views ➤ Home folder and select Add ➤ View. Name the file TestPartialView, and click the Add button to create it.

Add the following code to it:

<div style="color:red">This is partial View</div>

To apply this **Partial View** inside another View we use the Razor expression @await Html.PartialAsync().

So go to the TestLayout View which is kept inside the Views/Home folder and call the partial view by adding the express – @await Html.PartialAsync("TestPartialView").

Notice I passed the name of the Partial View to the @await Html.PartialAsync() expression

This is shown in the highlighted code below:

|  |
| --- |
| @section Top {      This is Top  }    <h2>TestLayout starts here</h2>  <p>This text is coming from child view</p>    @await Html.PartialAsync("TestPartialView")    @section Script {      <script src="https://code.jquery.com/jquery-3.2.1.min.js"></script>  }    @section Bottom {      This is Bottom  } |

Run the application and go to the URL – https://localhost:44339/Home/TestLayout. You will find the content of the partial view (in red color) added to the View. This is shown by the image below:

ASP.NET Core MVC looks for Partial View in the same way like Regular Views (i.e. in Views/<controller> and Views/Shared folders). This means you can create controller specific partial views or shared partial views.

### Strongly Typed Partial View

**Strongly Typed Partial Views** are used to carry data from Regular Views i.e. they have their own model.

Update the TestPartialView.cshml file by including a model to it this will make it **Strongly Typed Partial View**. The updated code of it is given below:

|  |
| --- |
| @model List<string>  <div style="color:red">      This is partial View:      <ul>          @foreach (string str in Model)          {              <li>@str</li>          }      </ul>  </div> |

Notice this partial view has a model of type List and I am showing the model values inside the li tag by looping through all of them using foreach loop.

Now I have to call this partial view from the TestLayout view by also providing the model value. This is done by using the 2nd parameter of @Html.PartialAsync like shown below:

@await Html.PartialAsync("TestPartialView", new List<string> { "Classic ASP", "ASP.NET Web Forms", "ASP.NET MVC", "ASP.NET Core MVC" })

So update the TestLayout View code as shown below:

|  |  |
| --- | --- |
|  | @section Top {      This is Top  }    <h2>TestLayout starts here</h2>  <p>This text is coming from child view</p>    @await Html.PartialAsync("TestPartialView", new List<string> { "Classic ASP", "ASP.NET Web Forms", "ASP.NET MVC", "ASP.NET Core MVC" })    @section Script {      <script src="https://code.jquery.com/jquery-3.2.1.min.js"></script>  }    @section Bottom {      This is Bottom  } |

Reload the page (https://localhost:44339/Home/TestLayout) to see the Partial View and it’s content as shown in the image below:

## View Components

**View Components** are C# classes that can be called from a View. View Components supports Views by providing them with data. Therefore helping in embedding complex contents in Views. Some examples where you can use View Components are:

* 1. Authentication Panels in sites from where users can log in without visiting a separate page.
* 2. Creating dynamic site navigation system which changes according to the user role.
* 3. Shopping Cart panels which shows the products currently in the cart.

Related article – [Model Binding in ASP.NET Core from Beginners to Advanced](https://www.yogihosting.com/aspnet-core-model-binding/)

### Creating a View Component

View Components are just C# classes that derive from the ViewComponent class defined in the Microsoft.AspNetCore.Mvc namespace.

The **View Component** must have an Invoke() method that returns some form of data to the View.

The View Component can be created anywhere in the application but according to conventions they should be created inside the Components folder that resides on the root folder of the application.

So right click on the project file and select Add ➤ New Folder then name that folder as Components. Create a class inside this folder and name it Cart.cs. Derive this class from ViewComponent base class and add an Invoke method to it. This is what I have done in the given below code:

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Threading.Tasks;  using Microsoft.AspNetCore.Mvc;    namespace UnderstandingControllersViews.Components  {      public class Cart: ViewComponent      {          public string Invoke()          {              return "This is from View Component";          }      }  } |

To invoke the View Component from a View use the Razor expression @await Component.InvokeAsync("NameofViewComponent") which will call the Invoke method of the View Component.

So go to the \_Layout.cshtml file and add @await Component.InvokeAsync("Cart") to it as shown in the code below:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>    <html>  <head>      <meta name="viewport" content="width=device-width" />      <title>@ViewBag.Title</title>  </head>  <body>      @RenderSection("Top")        <div style="color: blue;border: 1px solid #b6ff00">          @await Component.InvokeAsync("Cart")      </div>        <div>          @RenderBody()      </div>        @RenderSection("Bottom")        @RenderSection("Ads", false)        @RenderSection("Script")  </body>  </html> |

Test view component by running your application and go to the URL – https://localhost:44339/Home/TestLayout. You will see the returned string from the **Invoke method of View Component** being displayed. Check the below given image.

### Return Types of View Components

In the previous section I have shown View Components that returns a string. View Components can also return IViewComponentResult interface from the Invoke method.

With IViewComponentResult interface and you can use the Invoke method of **View Components** to return string, html and partial view.

To return string, html and partial view from View Components we use the classes which are defined below:

| **Name** | **Description** |
| --- | --- |
| ContentViewComponentResult | Is is used to return encoded HTML. Eg Content("<h2>some text</h2>") |
| HtmlContentViewComponentResult | It is used to return HTML without encoding. Eg HtmlContentViewComponentResult("<h2>some text</h2>") |
| ViewViewComponentResult | It is used to return a Partial View with optional view model data. Eg View("NameofView") |

### View Component Returning ContentViewComponentResult

The ContentViewComponentResult class is used for **returning encoded HTML from View Components**. Instances of the ContentViewComponentResult class are created using the Content() method.

Let me demonstrate this. Change the Cart View Component’ Invoke method as shown below.

|  |  |
| --- | --- |
|  | using System;  using System.Collections.Generic;  using System.Linq;  using System.Threading.Tasks;  using Microsoft.AspNetCore.Mvc;    namespace UnderstandingControllersViews.Components  {      public class Cart: ViewComponent      {          public IViewComponentResult Invoke()          {              return Content("This is from <h2>View Component</h2>");          }      }  } |

I changed the Return type of Invoke method to IViewComponentResult interface. Then I returned the encoded HTML by using the Content() method like:

Content("This is from <h2>View Component</h2>")

Run your application and go to URL – https://localhost:44339/Home/TestLayout. You will see the encoded html displayed as shown in the image below:

If you check the page source you will find the HTML is encoded by the Content method like:

This is from &lt;h2&gt;View Component&lt;/h2&gt;

The encoding of HTML is done by ASP.NET Core MVC for safety purpose and this prevent hackers from adding unwanted scripts to the website.

### View Component Returning HtmlContentViewComponentResult

The HtmlContentViewComponentResult class is used for returning **“non-encoded HTML”** from View Components.

The HtmlContentViewComponentResult class resides inside the Microsoft.AspNetCore.Mvc.ViewComponents namespace.

Change the Cart View Component so that it returns a new instance of HtmlContentViewComponentResult class. This is shown in the code below:

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Threading.Tasks;  using Microsoft.AspNetCore.Mvc;  using Microsoft.AspNetCore.Html;  using Microsoft.AspNetCore.Mvc.ViewComponents;    namespace UnderstandingControllersViews.Components  {      public class Cart: ViewComponent      {          public IViewComponentResult Invoke()          {              return new HtmlContentViewComponentResult(new HtmlString("This is from <h2>View Component</h2>"));          }      }  } |

Run your application and go to the URL – https://localhost:44339/Home/TestLayout. You will see this time the HTML is rendered on the browser as shown below:

Use this method when your are 100% sure that your returned output will be safe and will not be tampered.

### View Component Returning Partial View

You can also return a **Partial View from your View Component**. The ViewViewComponentResult class tells ASP.NET Core to **render the Partial View**. The ViewComponent base class provides the View() method for creating ViewViewComponentResult class objects.

There are 4 versions of the View method:

|  |  |
| --- | --- |
|  | View(); // selects default partial view  View(model); // selects default partial view and provides model to it  View("viewname"); // selects a partial view by its name  View("viewname ", model); // selects a partial view by its name and provides model to it. |

Here ASP.NET Core will look for the following locations to find the Partial View:

|  |  |
| --- | --- |
|  | /Views/{controller}/Components/{view component}/{partial view name}  /Views/Shared/Components/{view component}/{partial view name} |

* The controller is the one that handles the HTTP request.
* If you don’t specify the view name in the View() method then {partial view name} is taken as Default.cshtml.

Do you want to learn how to create URLs based on routing. This reduces code redundancy in your application. So read [ASP.NET Core Routing: Generating Outgoing URLs in Views](https://www.yogihosting.com/aspnet-core-generating-url-routing/)

Let us understand it by creating a **View Component that returns a Partial View**.

Create a model class Product.cs inside the Models folder with the content as given below:

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Threading.Tasks;    namespace UnderstandingControllersViews.Models  {      public class Product      {          public string name { get; set; }          public int price { get; set; }      }  } |

Now update your Cart View Component to return a View with an array of products as model.

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Threading.Tasks;  using Microsoft.AspNetCore.Mvc;  using Microsoft.AspNetCore.Html;  using Microsoft.AspNetCore.Mvc.ViewComponents;  using UnderstandingControllersViews.Models;    namespace UnderstandingControllersViews.Components  {      public class Cart: ViewComponent      {          public IViewComponentResult Invoke()          {              Product[] products = new Product[] {                  new Product() { name = "Women Shoes", price = 99 },                  new Product() { name = "Mens Shirts", price = 59 },                  new Product() { name = "Children Belts", price = 19 },                  new Product() { name = "Girls Socks", price = 9 }              };                return View(products);          }      }  } |

Run your application and go to URL – https://localhost:44339/Home/TestLayout. You will find an error stating that:

The view ‘Components/Cart/Default’ was not found. The following locations were searched:

/Views/Home/Components/Cart/Default.cshtml

/Views/Shared/Components/Cart/Default.cshtml

Notice the controller handling the HTTP request is HomeController and I did not specify the Partial View name in the View() method.

So MVC searches the Default Partial View (Default.cshtml) in the 2 locations:

|  |  |
| --- | --- |
| 1  2 | /Views/Home/Components/Cart/Default.cshtml  /Views/Shared/Components/Cart/Default.cshtml |

To fix this problem simply create this partial view. So create a View file called Default.cshtml in the folder /Views/Home/Components/Cart/ and add the following code to it.

|  |
| --- |
| @model Product[]  <table style="width:50%">      <tr>          <td><u>Product Name</u></td>          <td><u>Price</u></td>      </tr>      @{          foreach (Product p in Model)          {              <tr>                  <td>@p.name</td>                  <td>@p.price</td>              </tr>          }      }  </table> |

Now again re-run your application and go to URL – https://localhost:44339/Home/TestLayout. This time you will find the partial view included in the layout. Check the image given below:

You can access all the properties like Request, ViewBag, RouteData inside the View Component just like Controllers. For example use this code – string target = RouteData.Values["id"] as string inside the View Component will get your the value of Id segment of the route. In the same way you can also use the [Dependency Injection](https://www.yogihosting.com/aspnet-core-dependency-injection/) feature of ASP.NET Core in the View Component.

## Providing values to a View Component from Parent View

You can also provide any **value to the View Component from the Parent View**. This is done by passing an anonymous object to the 2nd argument of the @await Component.InvokeAsync() method.

Go to your \_Layout.cshtml file and change the InvokeAsync() by passing it a false value to a variable showCart like this:

@await Component.InvokeAsync("Cart", new { showCart = false })

Now change your Cart View Component code by adding the showCart parameter to the Invoke() method as shown in the code below:

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Threading.Tasks;  using Microsoft.AspNetCore.Mvc;  using Microsoft.AspNetCore.Html;  using Microsoft.AspNetCore.Mvc.ViewComponents;  using UnderstandingControllersViews.Models;    namespace UnderstandingControllersViews.Components  {      public class Cart : ViewComponent      {          public IViewComponentResult Invoke(bool showCart)          {              Product[] products;              if (showCart)              {                  products = new Product[] {                      new Product() { name = "Women Shoes", price = 99 },                      new Product() { name = "Mens Shirts", price = 59 },                      new Product() { name = "Children Belts", price = 19 },                      new Product() { name = "Girls Socks", price = 9 }                  };              }              else              {                  products = new Product[] { };              }                return View(products);          }      }  } |

Because of this change when the showCart variable is true then only the cart will show products.

Run your application and this time you will not get any product shown to you. See the image below:

Once again go to the \_Layout.cshtml file and change the InvokeAsync() method to pass true value to a variable showCart like this:

@await Component.InvokeAsync("Cart", new { showCart = true})

And this time you will see the products shown on the view.

## Asynchronous View Component

So for in the above examples I have used synchronous view components. There is also an **Asynchronous View Component** that can be used to do Asynchronous tasks.

The **Asynchronous View Component** has an InvokeAsync() method that returns a task. The MVC will wait for the task to complete and then insert the result in the view.

Note that you need to include the System.Net.Http package in your project from NuGet so that the application can make Asynchronous HTTP requests.

Let’s create an **Asynchronous View Component** which is going to provide the page size of any web page.

So right click on the components folder and add a new class and name it PageSize.cs. Add the following code to it:

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Net.Http;  using System.Threading.Tasks;  using Microsoft.AspNetCore.Mvc;    namespace UnderstandingControllersViews.Components  {      public class PageSize: ViewComponent      {          public async Task<IViewComponentResult> InvokeAsync()          {              HttpClient client = new HttpClient();              HttpResponseMessage response = await client.GetAsync("http://www.msn.com");              return View(response.Content.Headers.ContentLength);          }      }  } |

The **Asynchronous View Component** will get the page size of MSN page asynchronously through the HTTP GET request and then passes the size to the default view.

Now create a View file called Default.cshtml inside the View/Home/Components/PageSize folder. Add the following code to this file:

@model long

<div>Page size: @Model</div>

Now finally to call this Asynchronous View component in your \_Layout.cshtml file like this:

<div>

@await Component.InvokeAsync("PageSize")

</div>

Run your application and go to the URL – https://localhost:44339/Home/TestLayout. You will see the page size of the MSN page is shown. See the image below: