Contents

[Introduction 2](#_Toc132456576)

[Chapter 02 2](#_Toc132456577)

[The location of the views 2](#_Toc132456578)

[What is Razor? 3](#_Toc132456579)

[What is an action result? 3](#_Toc132456580)

[Chapter 03 4](#_Toc132456581)

[Why is important to have nullable properties in the models? 4](#_Toc132456582)

[The concept of model binding 5](#_Toc132456583)

[Doing Validation’s 6](#_Toc132456584)

[Relation between action’s and view’s 7](#_Toc132456585)

[Chapter 04 8](#_Toc132456586)

[Configuring HTTP Pipeline to serve static content in wwwroot folder 8](#_Toc132456587)

[Installing Tool Packages for Entity Framework 9](#_Toc132456588)

[Installing Tool Packages for LibraryManager 9](#_Toc132456589)

[Installing the Bootstrap CSS Framework 9](#_Toc132456590)

[Chapter 06 – Essential Features 9](#_Toc132456591)

[Global using statements a C# 10 feature 9](#_Toc132456592)

[Null State Analysis 9](#_Toc132456593)

[Extension methods 10](#_Toc132456594)

[Single Statement Pattern to Display Data in the View 10](#_Toc132456595)

[Creating anonymous type that can represent a view model object without to have to define a class or a struct, using object initializers 10](#_Toc132456596)

[Using Default Implementations in Interfaces 10](#_Toc132456597)

[Using Async/Await keywords 11](#_Toc132456598)

[Asynchronous Enumerable, IAsyncEnumerable<long?> 12](#_Toc132456599)

[Await foreach 14](#_Toc132456600)

[Chapter 07 and 08 14](#_Toc132456601)

[Resume of View Component’s, Razor Partial View’s, Razor Page’s 15](#_Toc132456602)

[Related to MVC Framework 21](#_Toc132456603)

[What type of folders we have? 21](#_Toc132456604)

[Configuring Application Services and the Request Pipeline 21](#_Toc132456605)

[Routing 22](#_Toc132456606)

[Explain the concept of ViewModel 25](#_Toc132456607)

[Related to Razor 26](#_Toc132456608)

[How to add a link in a view that references an action in Controller 26](#_Toc132456609)

[How to specify the fields of the Model 26](#_Toc132456610)

[The @model and Model 27](#_Toc132456611)

[The @foreach element 27](#_Toc132456612)

[How to disable null state analysis warnings? 28](#_Toc132456613)

[How to add a namespace? 28](#_Toc132456614)

[What are view model objects? 28](#_Toc132456615)

[There is a quirk in Razer Views that model data is always nullable. What this means? 28](#_Toc132456616)

[What is a tag helper and how to create one? 28](#_Toc132456617)

[Tag attributes automatically set Tag Helper properties and prevent hardcoding CSS style’s on Tag Helper’s 30](#_Toc132456618)

[Creating a partial view to reduce duplicated code in View’s 31](#_Toc132456619)

[Creating a view component 31](#_Toc132456620)

[Using ViewBag to pass additional data to a View 32](#_Toc132456621)

[What is a Razor expression? 33](#_Toc132456622)

[Related to Razor Pages 34](#_Toc132456623)

[About terms/language 35](#_Toc132456624)

[Related to ViewResult class 35](#_Toc132456625)

[How to get the model from the ViewResult? 35](#_Toc132456626)

[Some Doubts 36](#_Toc132456627)

[In Chapter 2, we created a global.json file in a C# project. What is the role of this file? 36](#_Toc132456628)

[In Chapter 07, try to understand better the interface IQueryable, page 151 38](#_Toc132456629)

[In Chapter 08, page 184, I have some doubts how urlHelper.Action method work to generate urls for “a” elements 38](#_Toc132456630)

[Fixing some Issues 38](#_Toc132456631)

[ArgumentException related to startup hook Microsoft.AspNetCore.Watch.BrowserRefresh.dll 38](#_Toc132456632)

[What I have Learned 40](#_Toc132456633)

[The move from MVC model to SPA (Single Page Application’s) 40](#_Toc132456634)

# Introduction

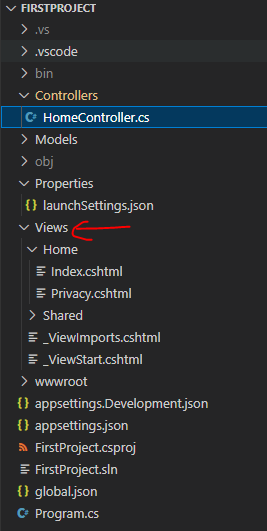
We followed Pro ASP.NET Core 9th edition book by Adam Freeman.

## Chapter 02

### The location of the views

Views\<Controller name>

Views\Shared

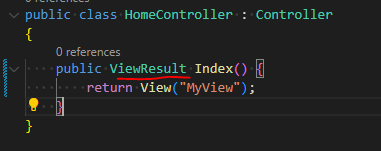


### What is Razor?

Razor is a view engine. It’s a component that process the contents of views and generates from the views, HTML content that is sent to the browser.

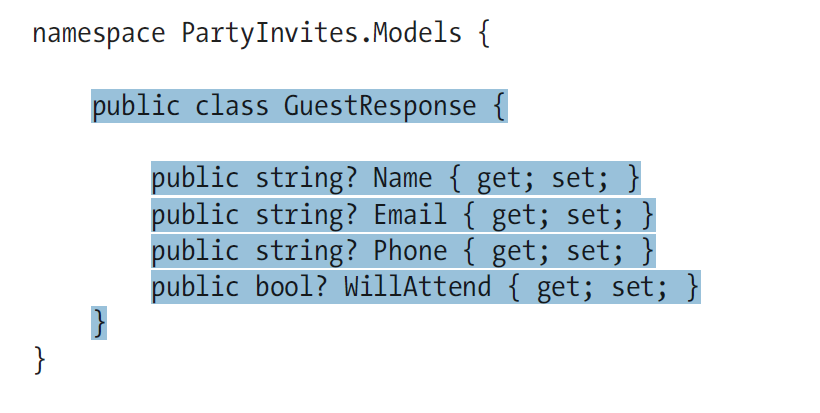
### What is an action result?

In a Controller you have action’s. The action’s can return a type like a string or it can return a result of type ViewResult. Other type of results are RedirectResult, HttpUnauthorizedResult.

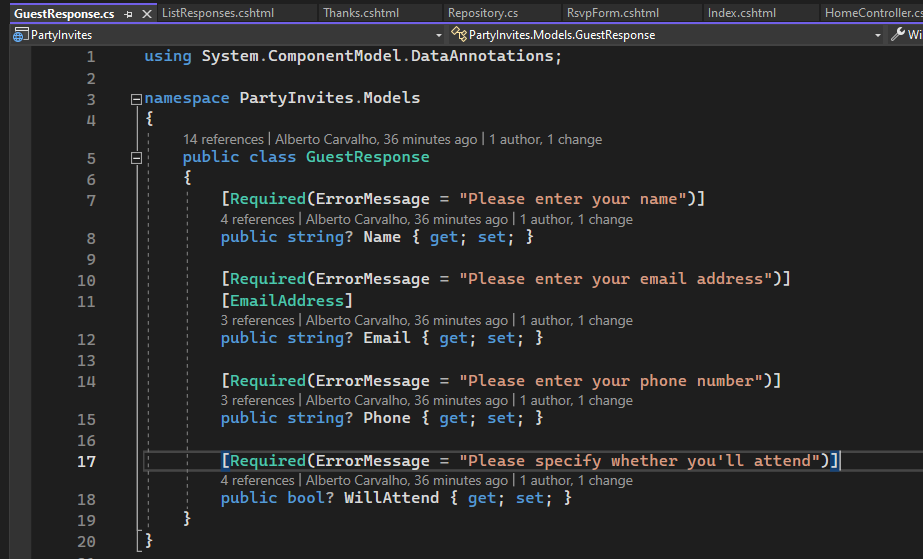


## Chapter 03

### Why is important to have nullable properties in the models?



For validation!

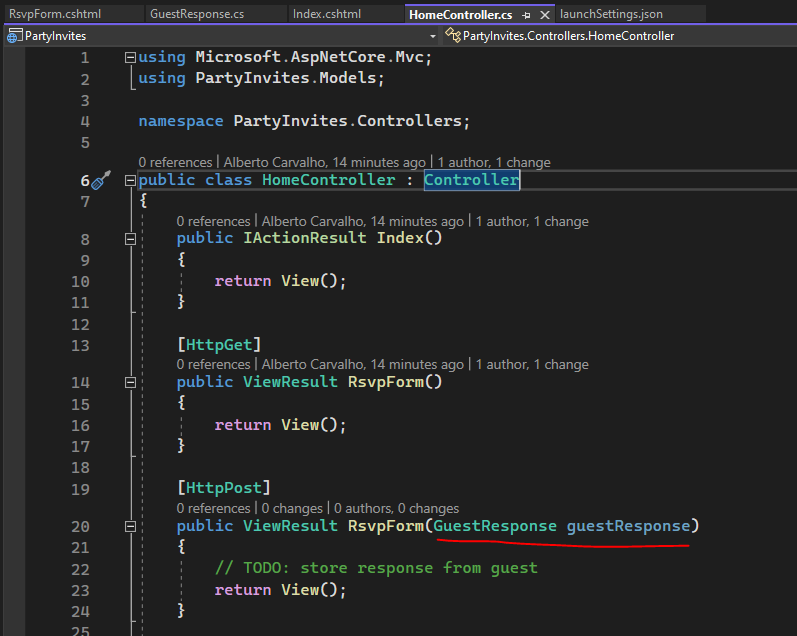


Because WillAttend is nullable type, it allow Required attribute to work! When the property is not set, Required attribute will work and a validation message will be displayed to the user.

If WillAttend is not nullable, this means when you receive a HTTP request and is passed to model binding, the WillAttend can only be true or false! But what if the user did not selected one option and the property is not set? The model binding will not work in this case.

### The concept of model binding

Model binding is a feature that parses a HTTP request (http fields into model properties) and creates the model, in this case GuestResponse:

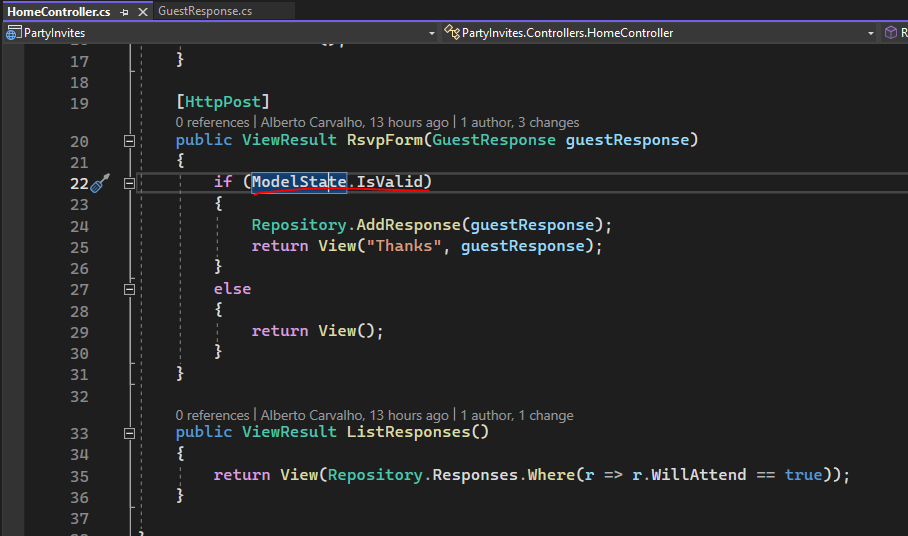


### Doing Validation’s

DataAnnotations contain several attributes like Required we can use to do validation’s.

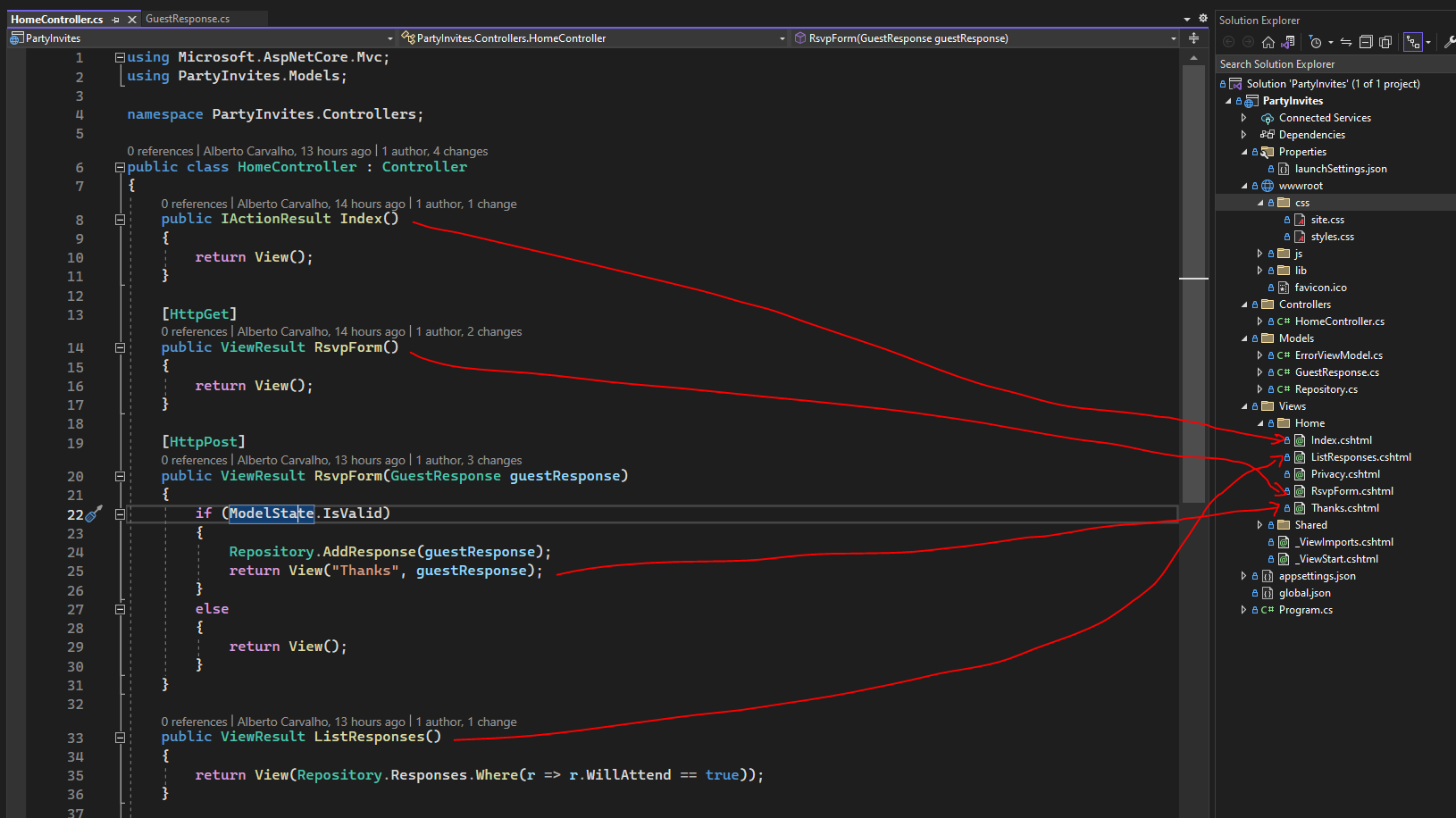


In the Controller, the Controller base class contain **ModelState.IsValid** property we can use to check if the validation succeeded.



### Relation between action’s and view’s

The following image shows the convention used. The convention is to have a folder for each controller. In this case the folder view’s for HomeController is Views\Home. Inside this folder we have view’s for each of the action’s (Index, RsvpForm..).

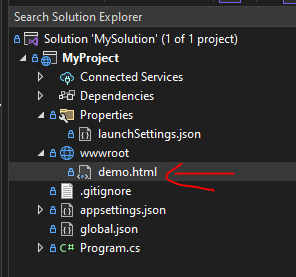


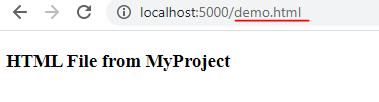
## Chapter 04

### Configuring HTTP Pipeline to serve static content in wwwroot folder

app.UseStaticFiles();

Without this, the file demo.html will not be served.





### Installing Tool Packages for Entity Framework

dotnet tool uninstall --global dotnet-ef

dotnet tool install --global dotnet-ef --version 6.0.0

### Installing Tool Packages for LibraryManager

dotnet tool uninstall --global Microsoft.Web.LibraryManager.Cli

dotnet tool install --global Microsoft.Web.LibraryManager.Cli --version 2.1.113

This allow to manage client side packages (eg bootstrap). The LibraryManager (LibMan) can download packages from different repositories. To set it to <https://cdnjs.com> (a content delivery site):

libman init -p cdnjs

### Installing the Bootstrap CSS Framework

The convention in ASP.NET Core projects is to install client-side packages in wwwroot/lib folder.

libman install bootstrap@5.2.3 -d wwwroot/lib/bootstrap

## Chapter 06 – Essential Features

### Global using statements a C# 10 feature

**Problem definition**

In most projects, some namespaces are required throughout the application. This can result in a long list of using statements, duplicated in every code file.

**Solution**

Global using statements address this problem by allowing using statements for commonly required namespaces to be defined in a single location.

You create a file GlobalUsing.cs and declare the namespaces using global keyword:

global using LanguageFeatures.Models;

global using Microsoft.AspNetCore.Mvc;

### Null State Analysis

**Problem definition**

Detecting references that can be unintentionally null.

**Solution**

null state analysis is a feature in which the compiler identifies attempts to access references that may be unintentionally null, preventing null reference exceptions at runtime.

### Extension methods

Extension methods can be applied to interface’s. This means these methods will be applied to the classes that implement the interface.

### Single Statement Pattern to Display Data in the View

public ViewResult Index() {

return View(***Product.GetProducts().Select(p => p?.Name)***);

}

***Product.GetProducts().Select(p => p?.Name) 🡪 the single statement***

### Creating anonymous type that can represent a view model object without to have to define a class or a struct, using object initializers

public class HomeController : Controller

{

public ViewResult Index()

{

var products = new[]

{

new { MyName = "Kayak", MyPrice = 275M },

new { MyName = "Lifejacket", MyPrice = 48.95M },

new { MyName = "Soccer ball", MyPrice = 19.50M },

new { MyName = "Corner flag", MyPrice = 34.95M },

};

return View(products.Select(p => p.GetType().ToString()));

}

}

### Using Default Implementations in Interfaces

Interfaces can describes methods and properties. Its possible now in C# to give to these methods and properties a default implementation.

**Problem definition**

How to extend existing classes without modifying them?

**Solution**

Using default implementation’s

### Using Async/Await keywords

**Problem**

Without async/await we need to do something like that:  
  
*public class MyAsincMethods  
    {  
        public static Task<long?> GetPageLength()  
        {  
            HttpClient client= new HttpClient();  
            var httpTask = client.GetAsync("http://apress.com");  
  
            return httpTask.ContinueWith( (Task<HttpResponseMessage> antecedent) => { return antecedent.Result.Content.Headers.ContentLength; } );  
        }  
    }*  
  
1.1 This method uses a System.Net.Http.HttpClient object to request the contents of the Apress home page and returns its length. .NET represents work that will be done asynchronously as a Task.   
  
Task objects are strongly typed based on the result that the background work produces.   
  
So, when I call the HttpClient.GetAsync method, what I get back is a Task<HttpResponseMessage>. This tells me that the request will be performed in the background and that the result of the request will be an HttpResponseMessage object.  
  
1.2 The part that most programmers get bogged down with is the continuation, which is the mechanism by which you specify what you want to happen when the task is complete. In the example, I have used the ContinueWith method to process the HttpResponseMessage object I get from the HttpClient.GetAsync method, which I do with a lambda expression that returns the value of a property that contains the length of the content I get from the Apress web server.  
  
1.3 Notice that I use the return keyword twice. This is the part that causes confusion. The first use of the return keyword specifies that I am returning a Task<HttpResponseMessage> object, which, when the task is complete, will return the length of the ContentLength header.   
  
The ContentLength header returns a long? result (a nullable long value), and this means the result of my GetPageLength method is Task<long?>.

**Solution**

2. Using async/await method, the programming model is more intuitive to the developer:  
  
*public class MyAsincMethods  
    {  
        public static****async****Task<long?> GetPageLength()  
        {  
            HttpClient client= new HttpClient();  
            var httpMessage =****await****client.GetAsync("http://apress.com");  
  
            return httpMessage.Content.Headers.ContentLength;  
        }  
    }*  
2.1 I used the await keyword when calling the asynchronous method. This tells the C# compiler that I want to wait for the result of the Task that the GetAsync method returns and then carry on executing other statements in the same method.  
  
2.2 Applying the await keyword means I can treat the result from the GetAsync method as though it were a regular method and just assign the HttpResponseMessage object that it returns to a variable.   
  
Even better, I can then use the return keyword in the normal way to produce a result from another method—in this case, the value of the ContentLength property. This is a much more natural technique, and it means I do not have to worry about the ContinueWith method and multiple uses of the return keyword.

3. We also need to change the Controller where the Index method will return a Task<ViewResult>. This means that there are a background operation that should produce a ViewResult.

public class HomeController : Controller

{

public async Task<ViewResult> Index()

{

long? length = await MyAsincMethods.GetPageLength();

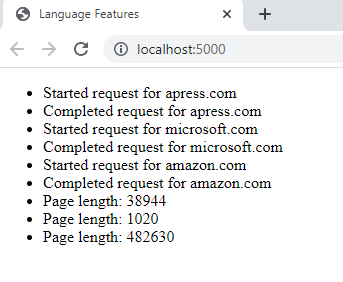
return View(new string[] { $"Length: {length}" });

}

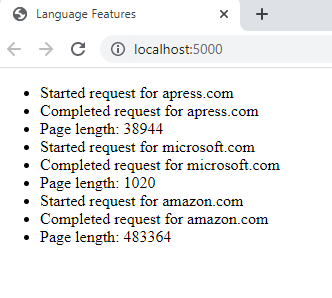
}

### Asynchronous Enumerable, IAsyncEnumerable<long?>

**Problem**



**Solution**



public static async IAsyncEnumerable<long?> GetPageLengths(List<string> output, params string[] urls)  
        {  
            //List<long?> results = new List<long?>();  
            HttpClient client = new HttpClient();  
  
            foreach(string url in urls)  
            {  
                output.Add($"Started request for {url}");  
                var httpMessage = await client.GetAsync($"http://{url}");  
                //results.Add( httpMessage.Content.Headers.ContentLength );  
                output.Add($"Completed request for {url}");  
  
                yield return httpMessage.Content.Headers.ContentLength;  
            }  
  
            //return results;  
        }  
  
public async Task<ViewResult> Index()  
        {  
            List<string> output = new List<string>();  
  
            await foreach(long? len in MyAsincMethods.GetPageLengths(output, "apress.com", "microsoft.com", "amazon.com"))  
            {  
                output.Add($"Page length: { len }");  
            }  
  
            return View(output);  
        }

### Await foreach

***await foreach*** *(long? len in MyAsincMethods.GetPageLengths(output, "apress.com", "microsoft.com", "amazon.com"))*

*{*

*output.Add($"Page length: { len }");*

*}*

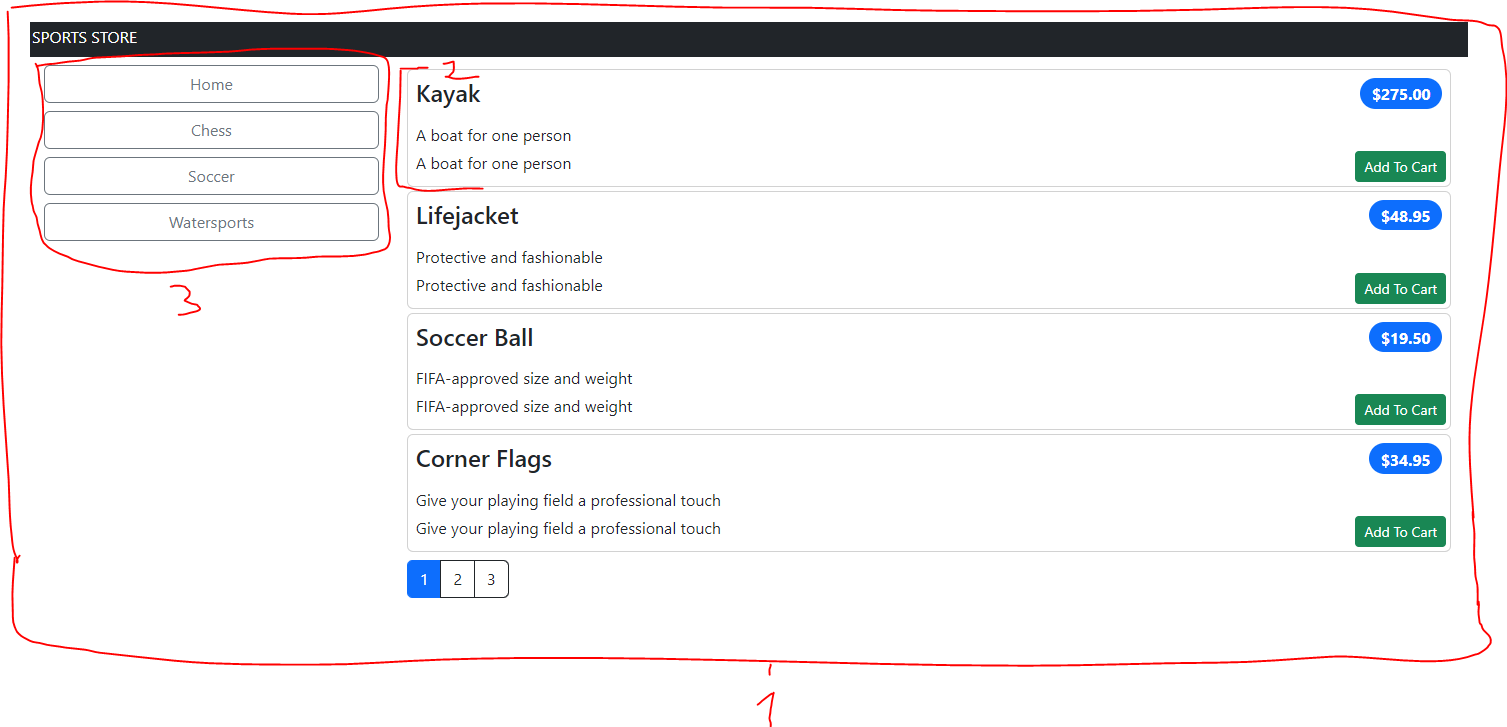
The await foreach statement is a new feature introduced in C# 8.0 and is used to iterate over a collection of items asynchronously.

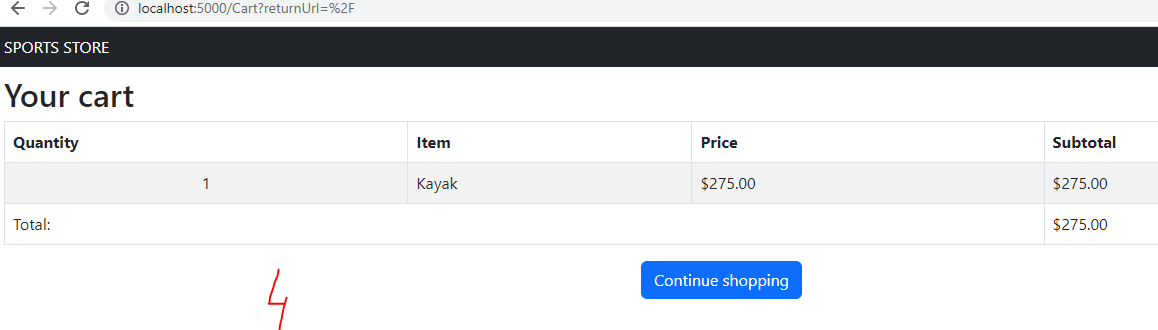
In your example, the await foreach statement is used to asynchronously iterate over the results of the GetPageLengths method, which returns an asynchronous sequence of page lengths for the specified URLs.  
  
The await foreach statement waits for the next item in the sequence to be available asynchronously, and then executes the body of the loop with the next item. This process continues until there are no more items in the sequence.  
  
Overall, the await foreach statement allows you to consume an asynchronous sequence of items in a more natural and intuitive way, without blocking the thread while waiting for the next item to become available.

## Chapter 07 and 08

### Resume of View Component’s, Razor Partial View’s, Razor Page’s

Looking to the web page, we can split it in 3 different parts:





1 – \_Layout –

* This cshtml file exist in Views/Shared folder. It defines the layout of the application.

2- ProductSummary –

* This is a razor partial view (see page 175). This cshtml file exist in Views/Shared folder.
* This is a reusable html content. In the view we use the <partial name=”PartialViewName”> tag to refer to it.
* By convention, the partial view is searched in Views/Shared folder.
* The Index.cshtml file has a reference to this partial view.

3- Default –

* This is a view component cshtml file.
* By convention the file exist in Views/Shared/Components folder.
* Its responsible to list product categories and to facilitate navigation through the product list.
* The \_Layout.cshtml file has an implicit reference to it, it references a view component through the <vc:<view component name>> tag (eg <vc:navigation-menu>).

4 – Cart

* This is a Razor page and the view is in the Cart.cshtml file.
* By convention, the file exist in Pages folder.

What is the difference between MVC and Razor Pages?

* For me, Razor Pages is complementary to MVC. In fact, looking to the namespace we see Microsoft.AspNetCore.Mvc.RazorPages, so from a logical perspective it belongs to Mvc.
* Razor pages seems to be a simplified model in the sense that it focus the application on the concept of “pages”. In contrast to Mvc that splits the code logic between the view, controller and the model (and we can have in between the concept of view model, in the sense that is the object that is transferred from the controller to the view and its not a model per se but an adaptation of the model in order to work well with the view), a Razor page consist only in a view (cshtml) file and the code behind it cshtml.cs that represents the model.
* Even the folder structure is a little different between Mvc and Razor Pages.

The ViewComponent belongs to Mvc or to Razor Pages?

* From a namespace perspective it belongs to Mvc, Microsoft.AspNetCore.Mvc.
* Its similar to the Controller class, it inherits from ViewComponent.
* By convention, the framework will look into Views/Shared/Components/<NameOfTheViewComponent> to locate the cshtml view file.

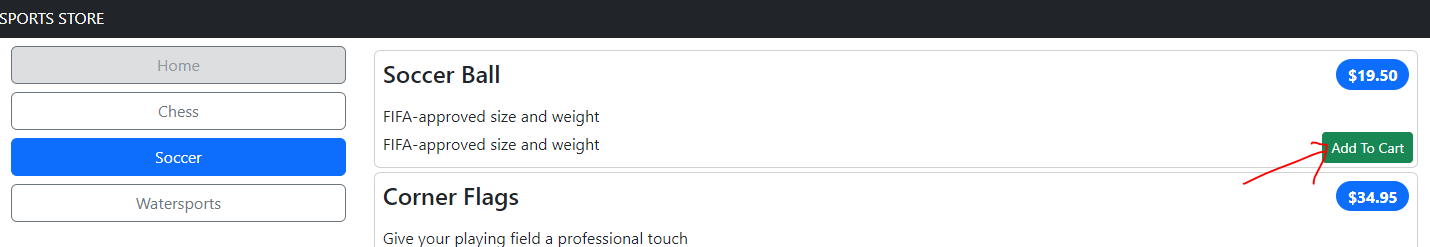
**Folder Structure similarities or differences**

|  |  |
| --- | --- |
| **MVC**    In Mvc the folder structure follows the layer approach of splitting the components between Controllers, Views and Models folder.  The ViewComponents are stored in Components folder. | **Razor Pages**    The Pages folder is similar to Views folder in the sense we can see similar views like \_ViewImports and \_ViewStart.  But the folder structure is more simpler in Razor pages than in Mvc. |

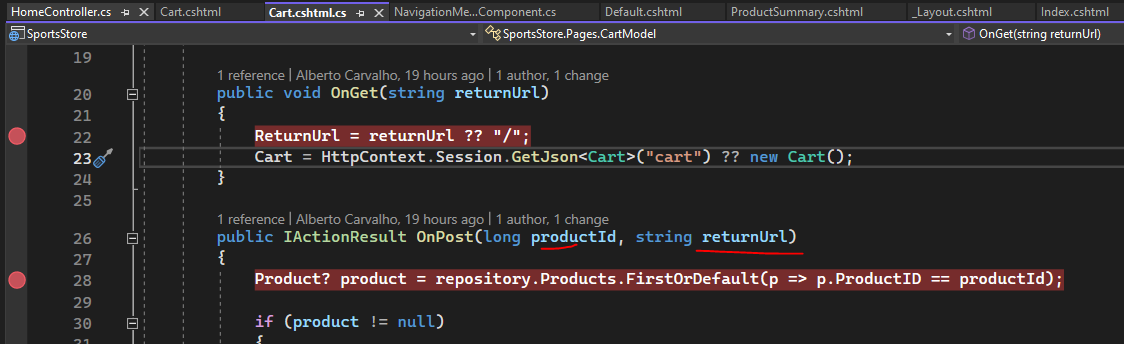
What happens when we request the page <http://localhost:5000/> ?

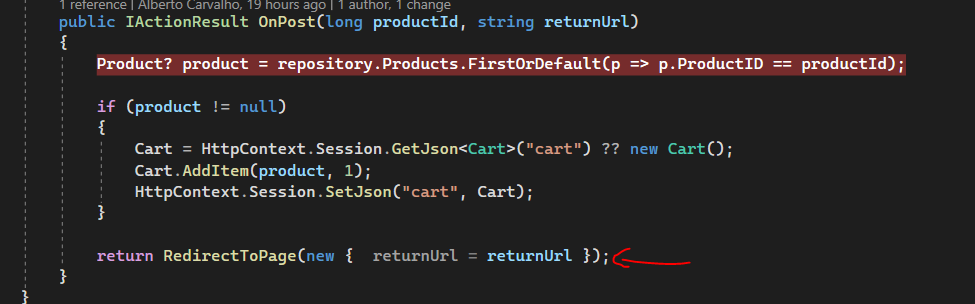
* The incoming request is forwarded to HomeController Index action method.
* The Index method returns a View. The index view is located in Views/Home/Index.cshtml folder.
* When the view is rendered, the NavigationMenuViewComponent Invoke method is called.

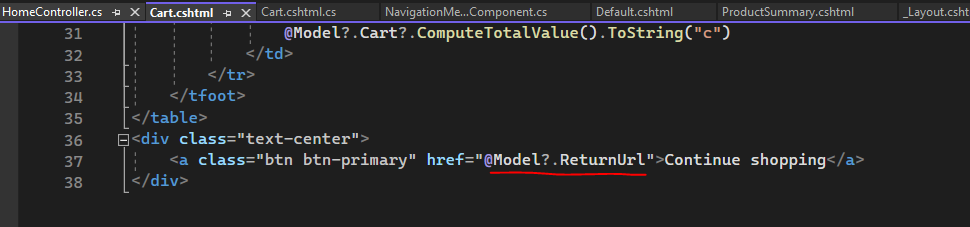
What happens when we click in the button Add to cart ?



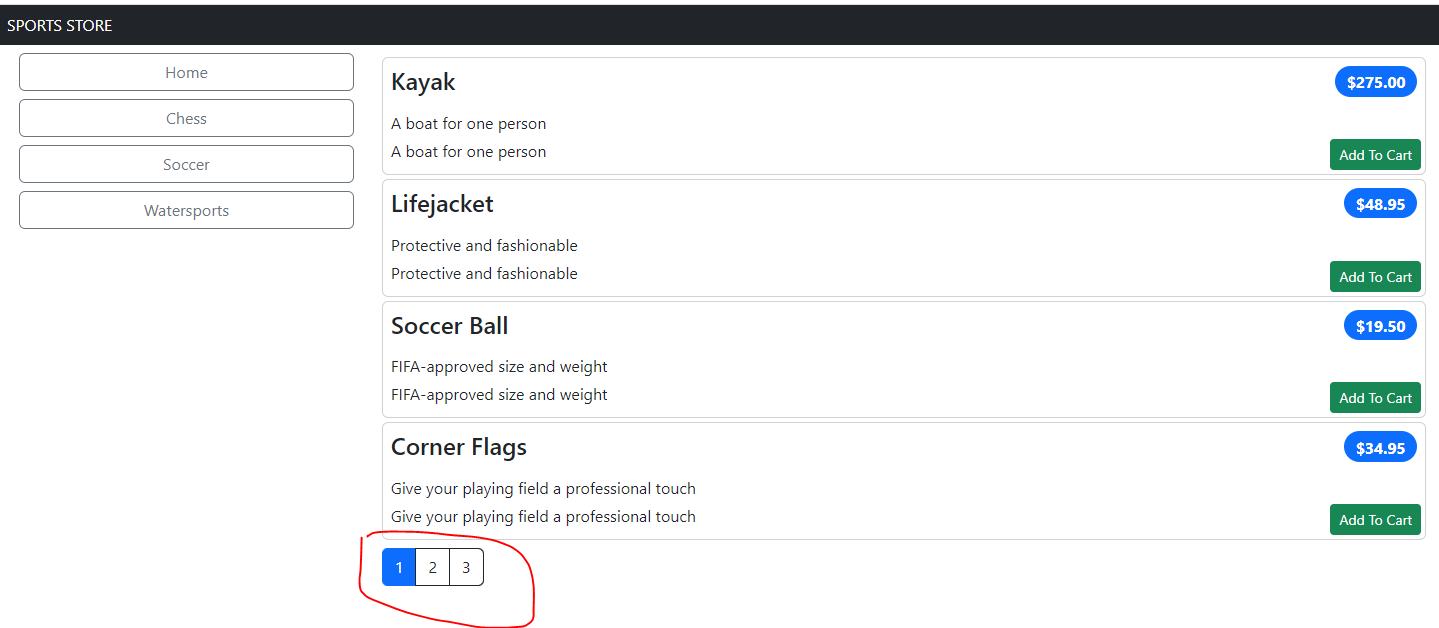






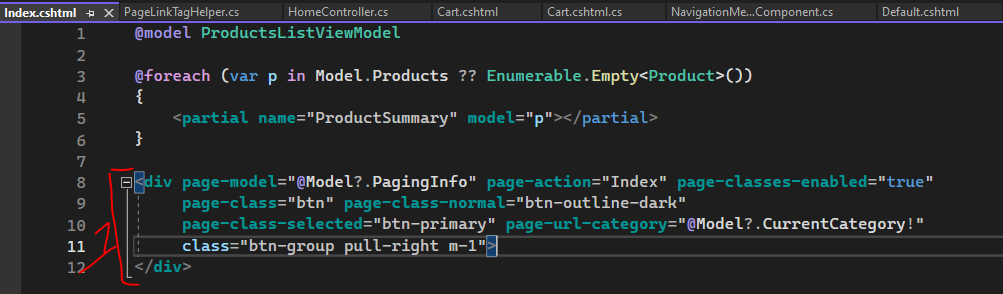
* If we look to the page source, we note 2 input hidden fields for each product summary.
* When we click in the Add to Cart button, these input hidden fields values will be passed to CartModel OnPost method through a feature called *model binding*.
* We can see that the input hidden fields ProductId and returnUrl maps to the OnPost parameter’s productid and returnUrl.
* The last line of CartModel OnPost method is to do a RedirectToPage. This makes a call to **the same page /Cart** and the CartModel OnGet method is called. This method sets the ReturnUrl CartModel property. We can see in Telerik Fiddler, that RedirectToPage is a request to the same page. This is why HomeController Index action method is not invoked. 
* The CartModel ReturnUrl property is used in the CartModel view. So when the user finish adding the product to the cart, it can click in “Continue shopping” button to return to the previous page. 

**We use a tag helper PageLinkTagHelper to generate pagination links at the bottom of the page. How the framework knows that it needs to use this tag helper? How the framework discover it?**

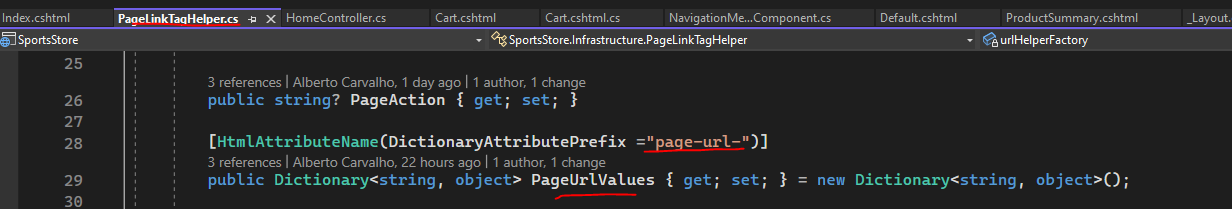


To remember a little bit, the image below shows the div element that will contain the pagination links. The tag helper purpose is to create those links. We can see that there is div attributes like **page-url-category** that will be passed to the tag helper. We also see an important attribute, the **page-model**.

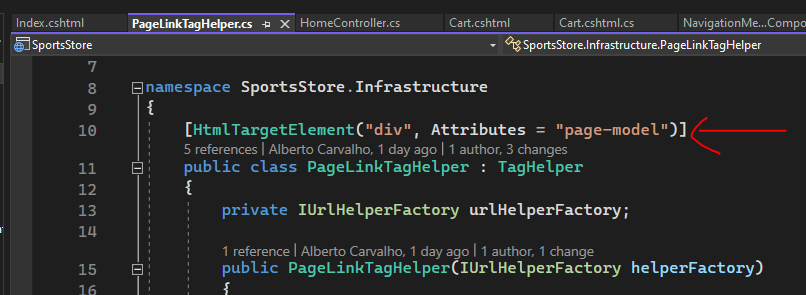
The mechanism through which those div attribute elements values are passed to the tag helper is through the use of prefix’s.



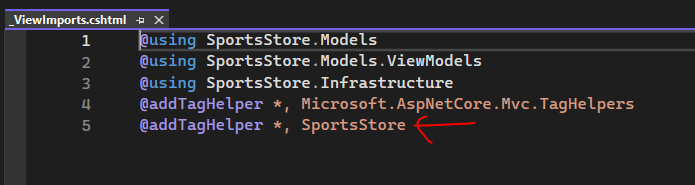
The Tag Helper PageUrlValues property will be automatically set.



For registration, we see that the tag helper is decorated with HtmlTargetElement. So when razor see div elements with an attribute page-model, it will pass the content of the div to this tag helper.



We also need to edit \_ViewImports.cshtml, to configure the location where the framework will search for tag helper’s. Here we say the framework to look in SportsStore project.



## Resume of Chapters 07 to 11

### Chapter 07 – SportsStore a Real Application

**List of Concepts**

Application services

Controller

Endpoints

HTTP request

Dependency injection

Model

Middleware

Migration (Microsoft Entity Framework)

MVC application folders

Object Relation Model (Microsoft Entity Framework)

Partial view

Razor files

Razor view engine

Request pipeline

Routing

Static content

Tag helpers

View

View Model

|  |  |
| --- | --- |
| Page |  |
| 144 | Application folders (Models, Controllers, Views, Views/Home, Views/Shared) |
| 144 | Application services and request pipeline |
| 145 | **builder.Service** property represents a collection of services; **dependency injection** |
| 145 | **app** variable is WebApplication; **middleware** |
| 145 | Serving static content through **wwwroot** folder |
| 145 | An important middleware component is the **routing** feature that matches HTTP requests to application features known as **endpoints** |
| 145 | **app.MapDefaultControllerRoute** registers MVC framework as a source of endpoints using a default convention for mapping requests to classes and methods. |
| 151 | **IConfiguration** interface provides access to ASP.NET Core configuration system which includes appsettings.json file; access to the configuration is done through **builder.Configuration** property; |
| 151 | The role of IQueryable<T> interface in Microsoft.Entity.Framework. It allow to create queries using LINQ statements. |
| 153 | **builder.Services.AddScoped**: the concept of scope; within that scope, all the objects get the same instance of a service. An example of a scope can be the duration of a request. |
| 153 | The concept of **migration**. It’s a Microsoft Entity Framework feature that allow us to generate database schema from a data model class. |
| 153 | Seeding data is the process of database initialization. |
| 155 | **WebApplication** class implements **IApplicationBuilder**.  **From chatGPT:**   * We use an **WebApplicationBuilder** to create an instance of **WebApplication** and this is done through the **Build** method. * Before using the **Build** method we need configure it. **WebApplicationBuilder** offers a fluent API that allow to configure various aspects of the application like the environment, logging and other type of configuration. * Then you use **Build** method and you get an instance of **WebApplication.** * Then you use the instance of **WebApplication** to configure the application’s request pipeline. * The request pipeline is a series of middleware components that handle incoming HTTP requests and produces HTTP responses. * To configure the request pipeline we can use the **Use** method.   **Personal note:**  From **Program.cs** file we can see the **Use** method technique to configure middleware components:    … |
| 158 | **ViewResult** is a class that implements **IActionResult**. An **IActionResult** defines a contract that represents the result of an action method.  **ViewResult** has some important properties:   * **ViewData** – this represents view data that the view can use to render HTML content * **ViewName** – the view name * **Model** – the view data model |
| 175 | Partial view represents HTML fragments. |

### Chapter 08 – Navigation and Cart

**List of Concepts**

Action methods

Idempotency

Razor pages

Tag helper

Session state

View

View component

|  |  |
| --- | --- |
| 183 | **IUrlHelper** interface provides URL functionality related to routing. |
| 186 | View component is a C# class that allow to reuse application logic. When you call a view component from **Invoke** method it will return a **IViewComponentResult**.  The **IViewComponentResult** is conceptually similar to **IActionResult**.  An example of a view component is the navigation component (is a menu composed of categories). |
| 191 | **RouteData** property; allow to access request data; |
| 192 | **ViewBag** is a dynamic object. Dynamic in the sense you can create properties.  An example of creating **SelectedCategory** property:  ViewBag.SelectedCategory = xxxxx; |
| 198 | Configuring Razor pages; this is different than configuring Razor view engine. This allow to configure services that support Razor pages. Razor pages are in a folder called **Pages**.  When we create a Razor page, the page is associated with a **PageModel**. This is different than the Controller model.  Cart example:   * Cart.cshtml – the Razor page (they are cshtml files). * Cart.cshtml.cs – page model; it represents the model that will be used in the view * Its not a requirement for a Razor page to have a page model (see page 240) |
| 202 | HTTP specification requires GET requests be idempotent, meaning that they must not cause changes. |
| 203 | Session state |
| 207 | HTTP POST requests causes changes. In this example, we update the user session with product information. |
| 210 | Razor pages are complementary to MVC framework (controllers and view’s). You can use razor pages alongside with controllers and view’s.  Razor pages are well-suited to self-contained features that do not require the complexity of MVC framework. |

### Chapter 09 – Completing the Cart

**List of Concepts**

Model binding

|  |  |
| --- | --- |
| 218 | **AddScoped** and **AddSingleton** method. |
| 223 | An example of a tag helper is asp-page-handler  ...  <form **asp-page-handler**="Remove" method="post">  ... |
| 227 | **BindNever** attribute is related to model binding system. It’s the process of mapping HTTP request values to properties. |
| 233 | Understanding the Order repository. The **Include** and **ThenInclude** methods.  When we read an Order from database, we also read the associated Lines and Produc objects.  ...  public IQueryable<Order> Orders => context.Orders  .Include(o => o.Lines)  .ThenInclude(l => l.Product);  ...  This ensure we receive all the data that we need without having to perform separate queries and then assemble the data.  The **AttachRange** method. This prevents that Entity Framework tries to write a Product that already exist in the database (if it does so, you will get an error).  ...  context.AttachRange(order.Lines.Select(l => l.Product));  ... |
| 235 | Validation attributes. We can say for each model property if the property is required or not.  A controller inherits from **Controller**. From this class we have access to **ModelState**. This allow us to check the validation of a model, **ModelState.IsValid**.  We can use ModelState.AddModelError to add/register to it error message’s. |
| 240 | **BindProperty** – interacting with the model binding system; the value for the property should be get from the request. |

### Chapter 10 - Administration

**List of Concepts**

Blazor server

Razor component

Razor pages

|  |  |
| --- | --- |
| Page |  |
| 243 | **Blazor** combines client-side javascript with server-side code executed in ASP.NET Core, connected by a persistent HTTP connection (in the previous example there is multiple requests and for each one a new connection).  Blazor server uses Razor pages.  Blazor is not suited for all projects. |
| 245 | Razor component’s are the Blazor building blocks.  Razor component’s have **razor** extension. |
| 247 | Blazor uses Razor syntax to generate HTML but introduces its own directives and features.  @page – is an example of a directive. |
| 251 | Blazor adds expressions to Razor syntax. In this example it adds @onclick attribute:  ...  <button class="btn btn-sm btn-danger"  @onclick="@(e => OrderSelected.InvokeAsync(o.OrderID))">  @ButtonLabel  </button>  ... |
| 252 | Blazor components are not like other application framework building blocks used. Instead of dealing with individual requests, components can be long-lived and deal with multiple user interactions over a longer period of time. This requires a different style of development, especially when it comes to dealing with data using EF (the **@inherits** expression ensures the component gets its own repository object, which ensures its operations are separate from those performed by other components.  @page "/admin/orders"  @inherits OwningComponentBase<IOrderRepository> |

### Chapter 11 – Security and Deployment

**List of Concepts**

Authorization policy

ASP.NET Core Identity

Error handling

Docker

Security

|  |  |
| --- | --- |
| Page |  |
| 275 | **Authorize** attribute. Authorization policy. |
| 281 | Error handling |
| 282 | **app.UseRequestLocalization**  **…**  **app.UseExceptionHandler** |

# Chapter 12 – Understanding the ASP.NET Core Platform

This chapter talks about using middleware/pipeline’s.

|  |  |
| --- | --- |
| Page |  |
| 294 | Key features are the request pipeline, middleware and services.  When ASP.NET Core receives a request it delegates processing to middleware components.  Middleware components are arranged in a chain know as request pipeline. |
| 294 | There is the concept of Service. An example is logging service. We can look to a Service as something that provides features. Services are shared by middleware components.  We can access service’s through a feature called *dependency injection*. |
| 297 | When we see this line, this means we are setting up a middleware component through app, a WebApplication object. This middleware component is responsible to handle requests with the route “/”. The delegate function “Hello World” is responsible to generate a response:  …  app.MapGet("/", () => "Hello World!");  … |
| 299 | How to create a custom middleware component?  We use the Use method from WebApplication.  …  app.Use(async (context, next) => {  …  The Use method receives a lambda function.    How to discover the arguments to this lambda function?    From the documentation we can see that, the lambda function receives a HttpContext object and a function (Func<Task>) that the lambda function must calls/invoke in order to instruct ASP.NET Core to pass the request to the next middleware component in the pipeline.  HttpContext provides access to a Request and a Response. |
| 301 | Using higher-level ASP.NET Core feature such as MVC Framework or Razor Pages we do not need to use directly HttpRequest, HttpResponse objects.  Its important to set the ContentType because this will prevents the subsequent middleware component from trying to set it. Be aware that the headers are sent to the client before the response body. In this example we writes a string to the response body. If a subsequent middleware component sets a response header an exception will be thrown. |
| 302 | Why we use async in the lambda function?  Because the next function is async. We can also use:  *next.Invoke()* |
| 304 | Using a class as a middleware  We need to define Invoke method in the class:  …  public class QueryStringMiddleware  {  private RequestDelegate next;  public QueryStringMiddleware(RequestDelegate nextDelegate)  {  next = nextDelegate;  }  public async Task Invoke(HttpContext context)  {  …  Program.cs:  …  app.UseMiddleware<Platform.QueryStringMiddleware>();  …  In Program.cs we need to define the order of the middleware’s in the request pipeline:    The first middleware component that will be invoked is 1 and then 2. |
| 305 | Middleware can operate before the request is passed on, after the request has been processed by other components, or both. The result is that several middleware components collectively contribute to the response that is produced, each providing some aspect of the response or providing some feature or data that is used later in the pipeline.      The order of middleware component’s:      **Important note:**  Middleware components must not change the response status code or headers **once ASP.NET Core has started to send the response to the client**. Check the HasStarted property. |
|  |  |
|  |  |
|  |  |

Classes:

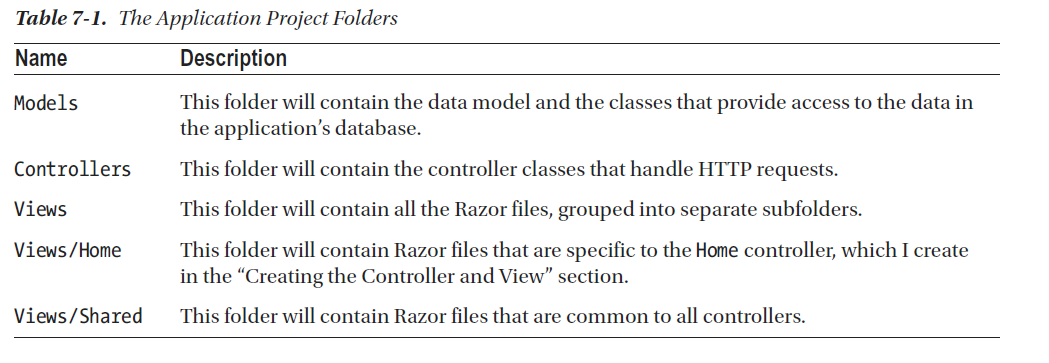
* WebApplicationBuilder
* WebApplication
* IEndPointRouteBuilder
* HttpContext
  + HttpRequest
  + HttpResponse

# Chapter xx – xx

|  |  |
| --- | --- |
| Page |  |
|  |  |
|  |  |

## Related to MVC Framework

### What type of folders we have?



### Configuring Application Services and the Request Pipeline

**Program.cs**

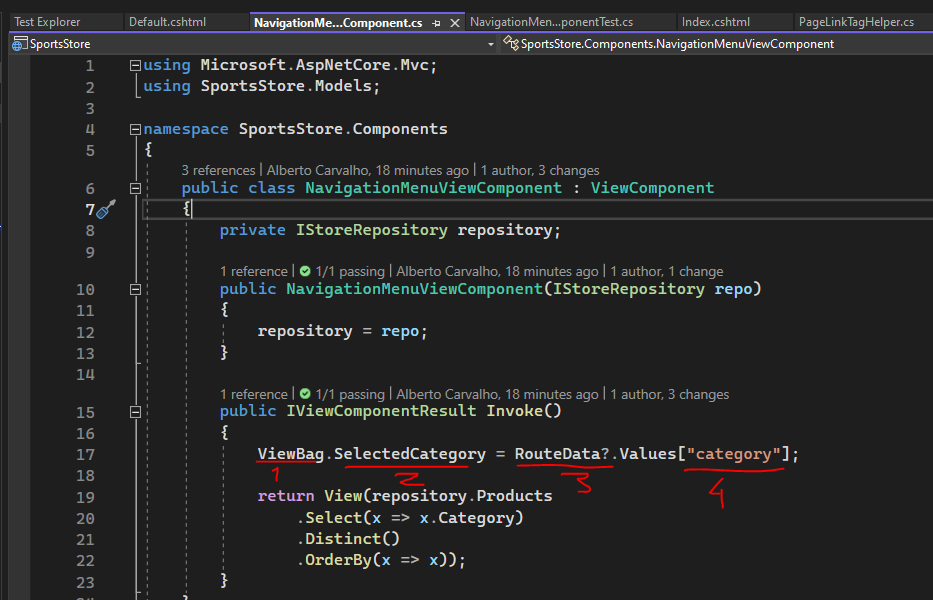
|  |  |
| --- | --- |
| builder.Services | The **builder.Service** property is used to set up objects, known as *services*, that can be used throughout the application and that are accessible through a feature called *dependency injection*; |
| builder.Services.AddControllersWithViews(); | This method sets up shared objects required by applications that uses MVC Framework and the Razor view engine. |
| app | The request pipeline is the heart of ASP.NET Core.  ASP.NET Core receives HTTP requests and passes them along a *request pipeline*. The request pipeline components are registered through the app property. Each component can inspect the request and modify it, can generate a response or modify responses from other components. |
| app.UseStaticFiles(); | This method enables support for serving static content from the **wwwroot** folder. |
| app.MapDefaultControllerRoute(); | One especially important middleware component provides the endpoint routing feature, which  matches HTTP requests to the application features—known as endpoints—able to produce responses for  them.  The endpoint routing feature is added to the request  pipeline automatically, and the MapDefaultControllerRoute registers the MVC Framework as a source of  endpoints using a default convention for mapping requests to classes and methods. |
|  |  |

### Routing

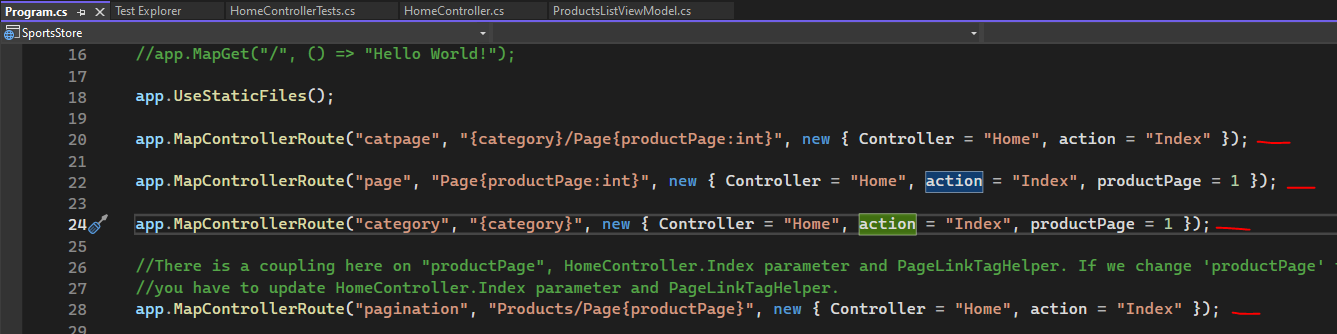
|  |  |
| --- | --- |
| http://localhost:5000/?productPage=2 | This is a valid URL when you have a HomeController with an Index action method.  Another valid url is:  http://localhost:5000/home/index?productPage=2 |
|  |  |

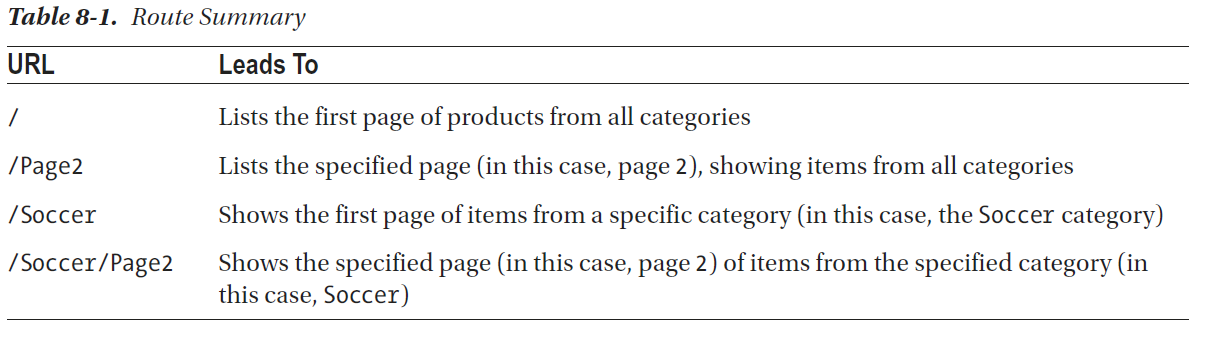
#### How you can access information about the current request from a ViewComponent (the same technique applies to a Controller or a View)

In chapter 08, page 191 you can see that we are using RouteData from ViewComponent to access route data:



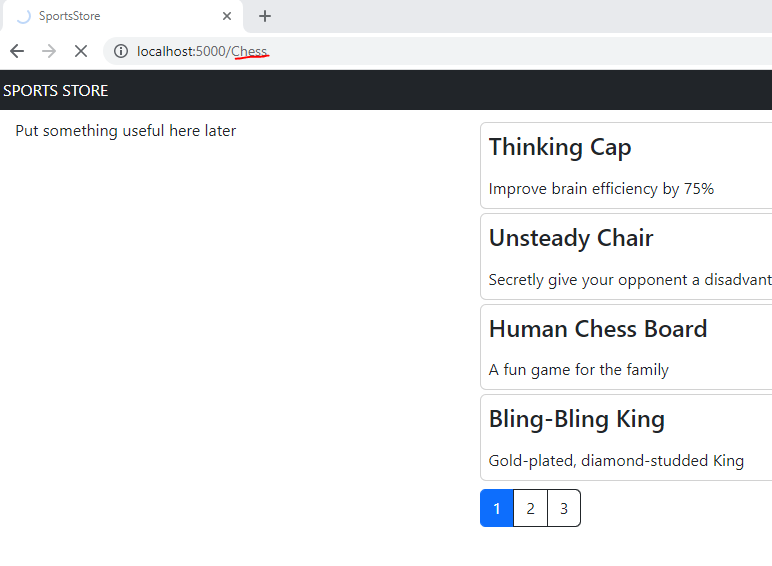
**Chapter 08, page 183, we updated the url scheme:**



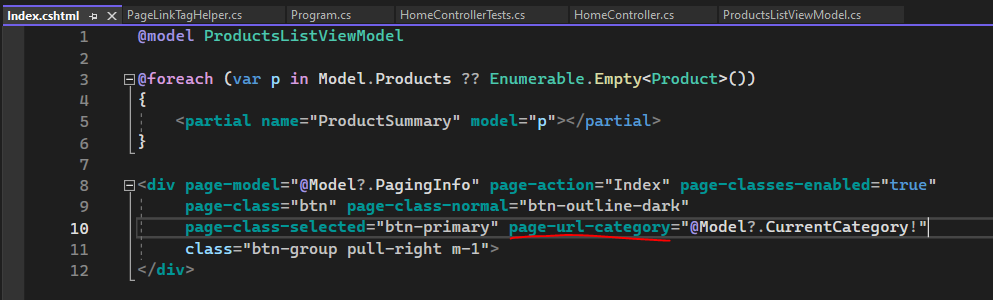


#### Filtering by category

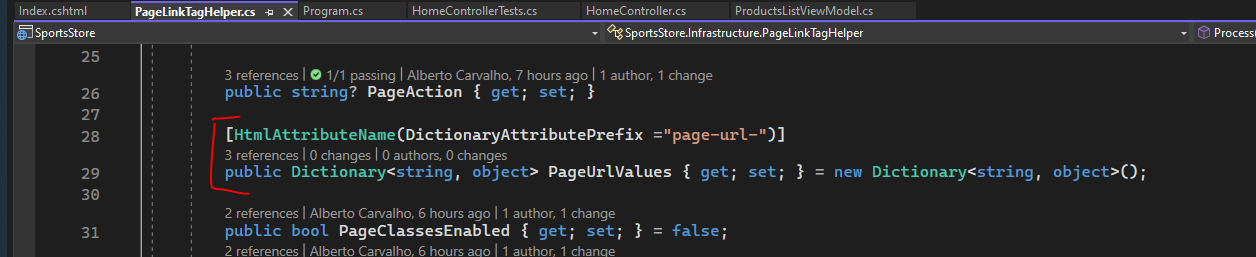
Chess is a category



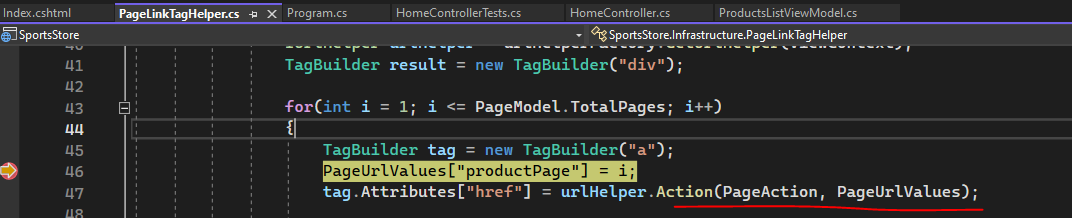
In the view, we added an html attribute to the div, page-url-category. The Tag helper operates on this div and it will receive the values of the attributes that begins by *page-url- in a dictionary*.



We can see that Tag Helper has the PageUrlValues property and it will be automatically filled (eg category = chess)



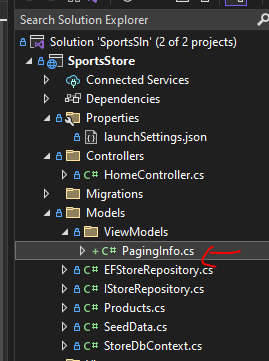
PageUrlValues property will be used by urlHelper.Action method to generate “a” elements.

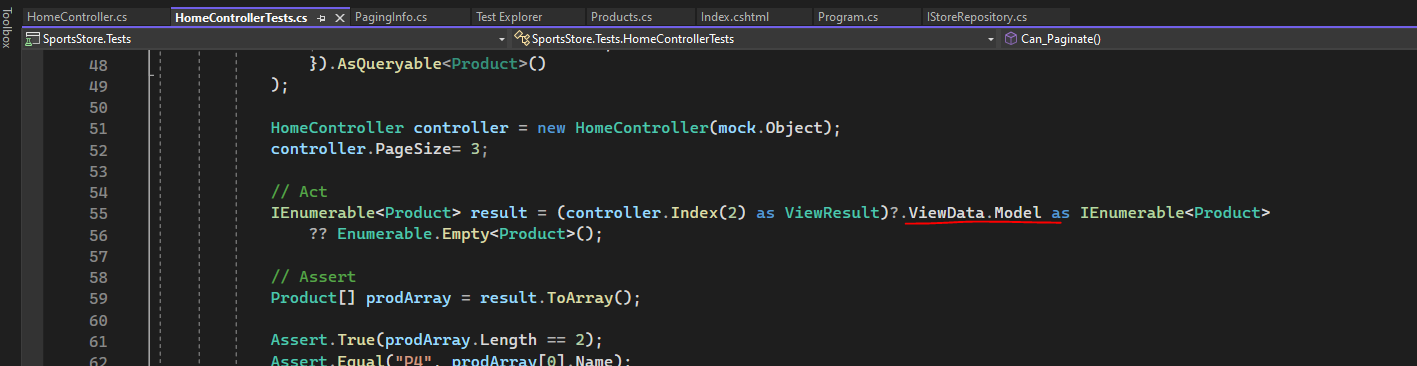


### Explain the concept of ViewModel

1. A **view model** is an object that is used *to transfer data between a controller and a view*.

This is an example of a class PagingInfo created in Models/ViewModels folder that is used to transfer data between the controller and the view.

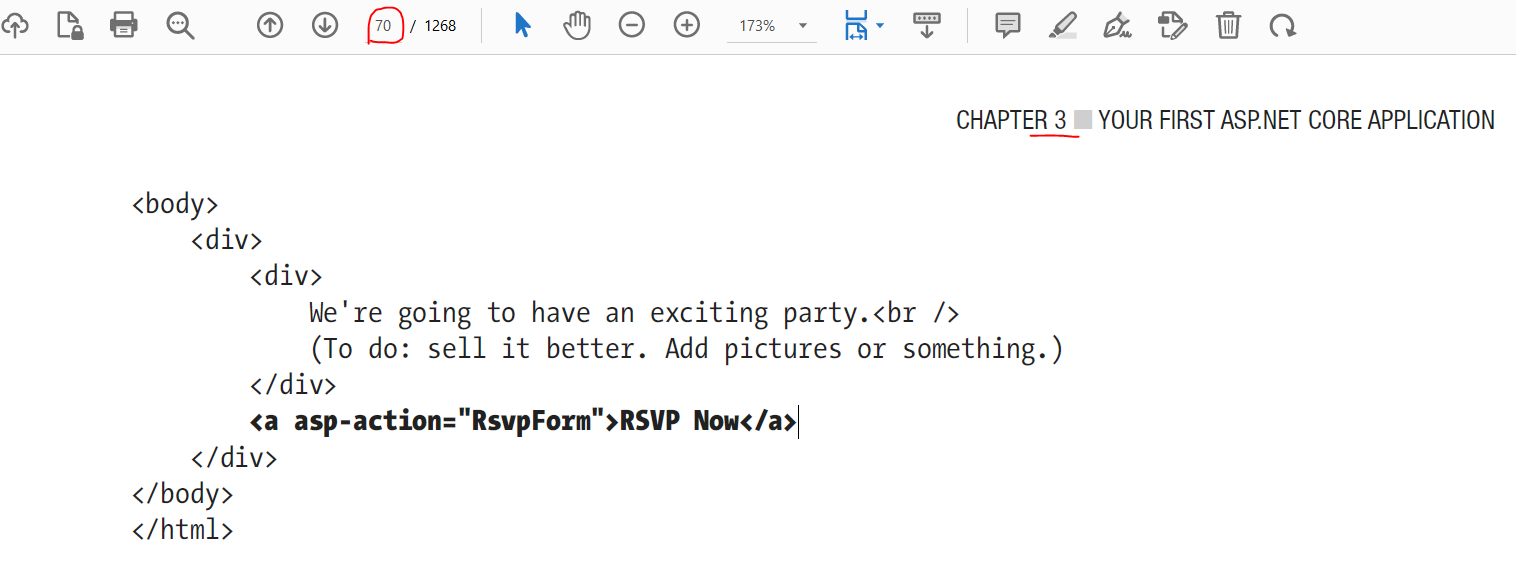


1. This another way to pass **view model** data. Here the **View** method will create a ViewResult and this ViewResult object will contain view model objects (aka model data) that the **View** will use to generate HTML content.
2. In this unit test we can see that we access the **view model** objects through ViewData.Model property:

## Related to Razor

### How to add a link in a view that references an action in Controller

<a **asp-action**="RsvpForm">RSVP Now</a>



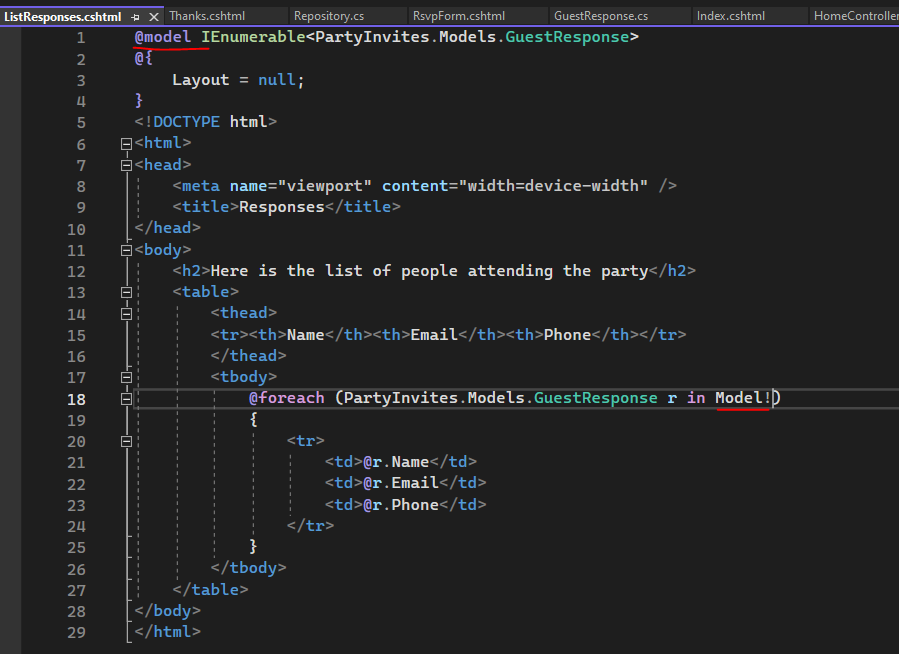
### How to specify the fields of the Model

<label **asp-for**="Name">Your name:</label>

|  |  |
| --- | --- |
| The model | The view |

### The @model and Model

The @model with “m” lower letter defines the type of the model. Then Model with “M” as capital letter is the model itself:



### The @foreach element

<tbody>

@foreach (PartyInvites.Models.GuestResponse r in Model!)

{

<tr>

<td>@r.Name</td>

<td>@r.Email</td>

<td>@r.Phone</td>

</tr>

}

</tbody>

Other technique using null coalescing operator when working with collection’s:

@foreach (var p in Model ?? Enumerable.Empty<Product>())

{

…

}

### How to disable null state analysis warnings?

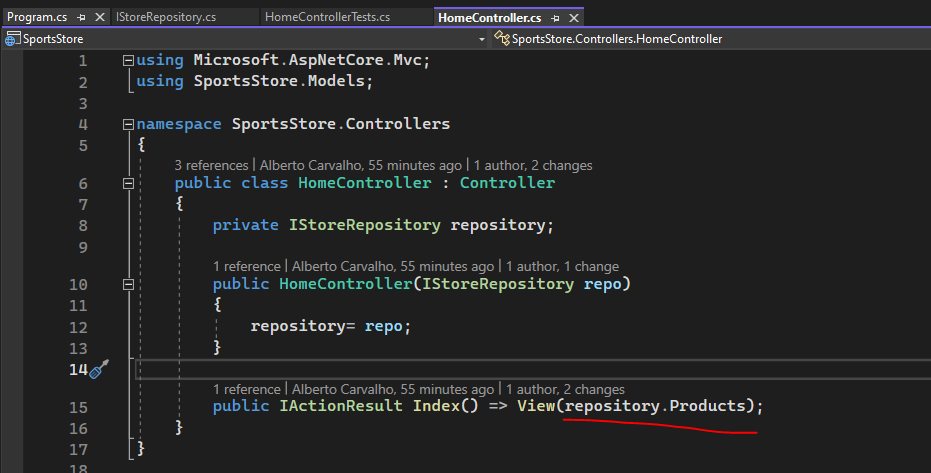
@{ #pragma warning disable CS8602 }

### How to add a namespace?

@using SimpleApp.Models

### What are view model objects?

The action method’s in the Controller, generates a view model that will be used by Razer engine to generate HTML content. In other words, the **view** generates HTML content using **view model** objects.



### There is a quirk in Razer Views that model data is always nullable. What this means?

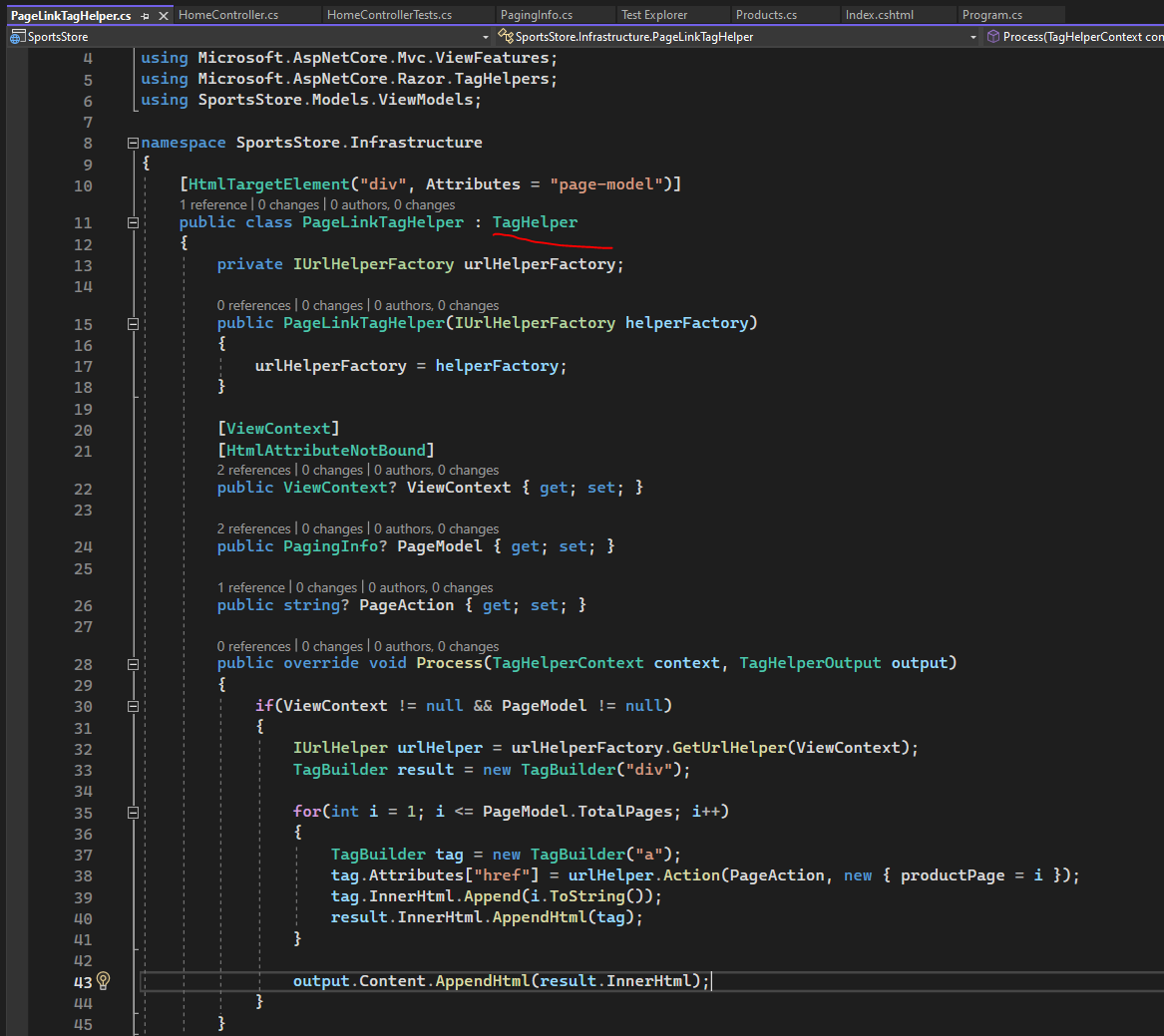
There is a quirk in the way that Razor Views work that means the model data is always nullable, even

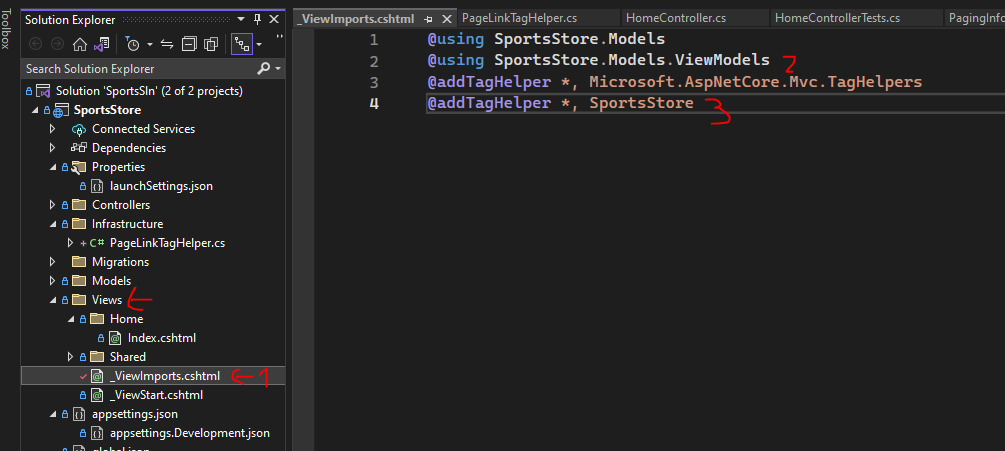
when the type specified by the @model expression is not. For this reason***, I use the null-coalescing operator in the @foreach expression with an empty enumeration***.

@foreach (var p in Model ?? Enumerable.Empty<Product>())

### What is a tag helper and how to create one?

1. Tag helper helps generating HTLML markup. The term tag relates to HTML tags. An example of a HTML tag is div.
2. A tag helper is a way to introduce C# logic into a view and its preferable than using blocks of C# code because a tag helper can be unit tested.
3. This is an example of a TagHelper that when used in a View it will help generate HTML content:



1. Tag Helper’s need to be registered (this is contrary to some ASP.NET Core components such as Controllers and Views that are automatically discovered). We have a file (1) called \_ViewImports.cshtml where we register our tag helper. The registration (3) tells ASP.NET Core to look for tag helpers in SportsStore project. We also added an using statement (2) so that we can refer to view model classes in View’s without having to qualify the classes with the namespace.

### Tag attributes automatically set Tag Helper properties and prevent hardcoding CSS style’s on Tag Helper’s

Here we can see the mapping between div attributes and tag helper properties. They are automatically set based on tag (this case div) attribute values:

|  |  |
| --- | --- |
| **Index.cshtml** | **PageLinkTagHelper** |

This show a technique that prevent you to hardcode CSS style styles (eg btn) in the Tag helper (eg PageLinkTagHelper). On the left image, you see we added custom attributes to div element (eg page-class) and these custom attributes correspond to Tag Helper properties like PageClass and are automatically set; then you use the values (eg btn for page-class/PageClass) to style the “a” elements”.

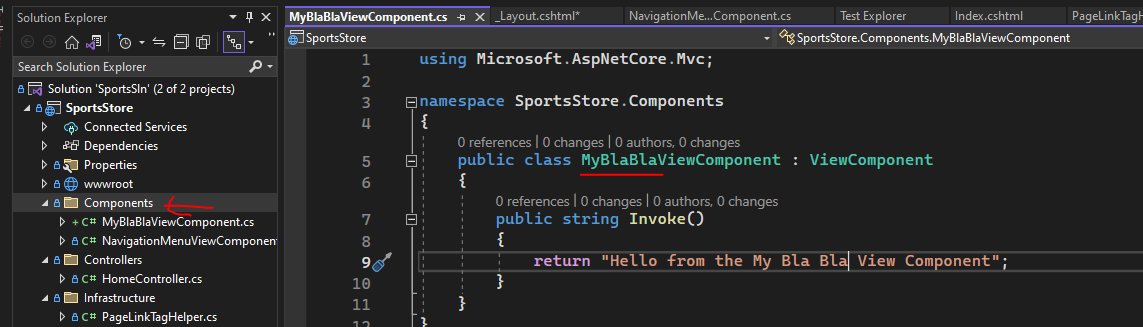
### Creating a partial view to reduce duplicated code in View’s

|  |  |
| --- | --- |
|  |  |

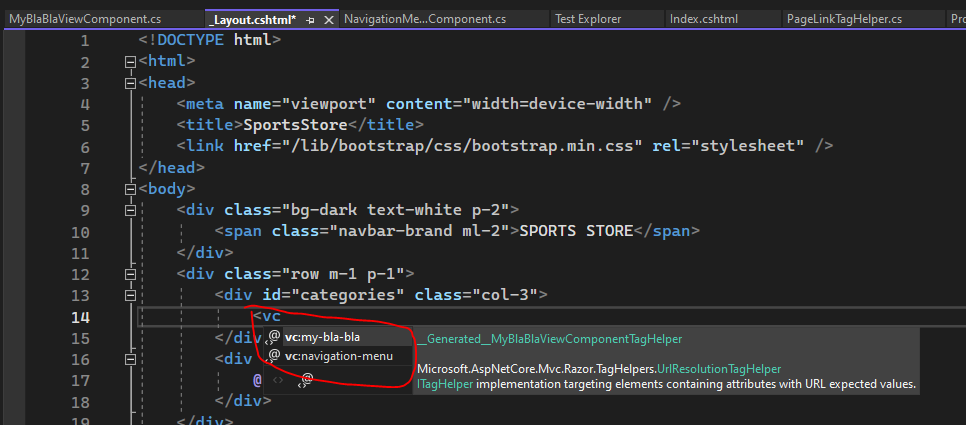
### Creating a view component

A view component is a C# class that provides a small amount of reusable application logic with the ability to select and display Razor partial views. Page 186.

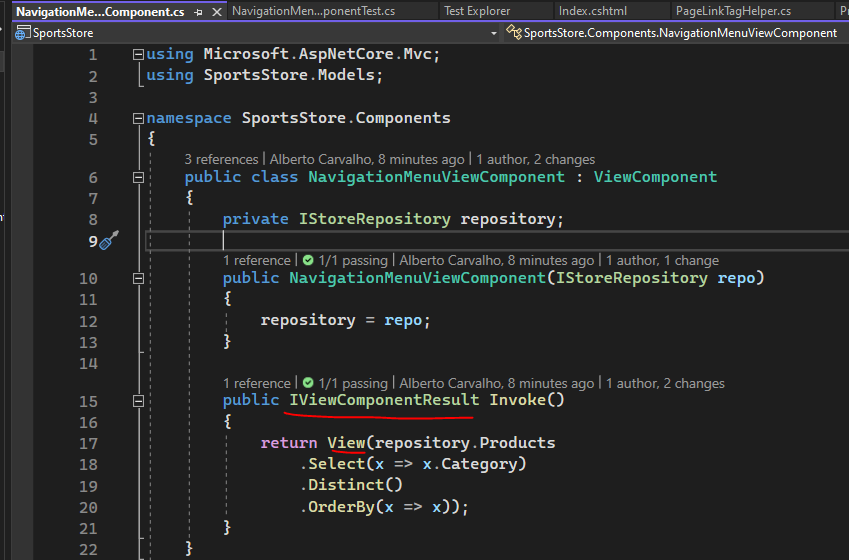
ViewComponent’s are automatically discovered:



In the view, you can begin typing by “vc” where vc stands for view component:



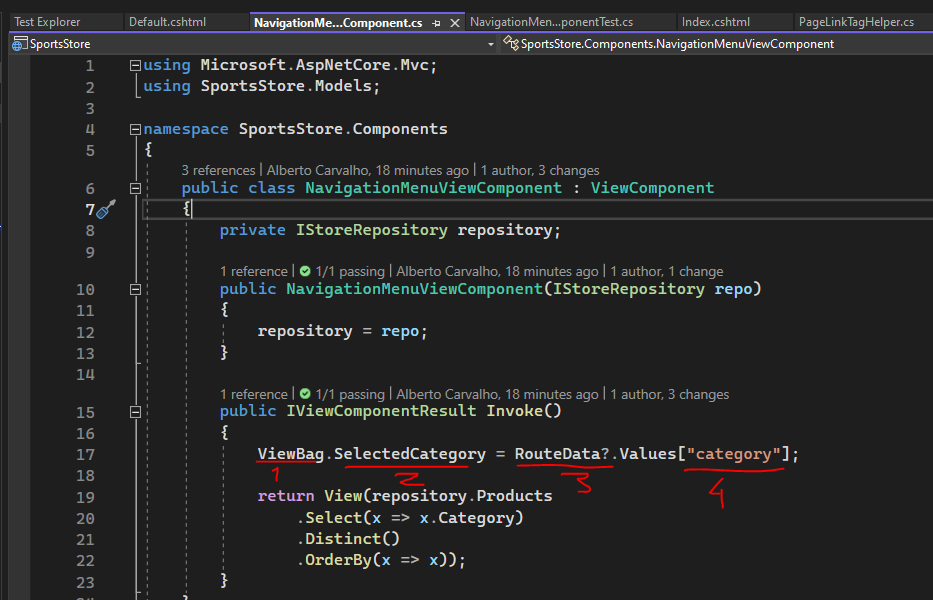
We use similar concepts as in the Controller like ViewResult (or IActionResult) or View, to generate the view model data.



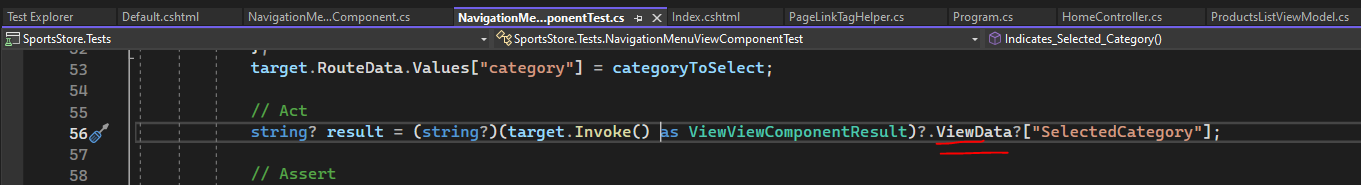
How Razor locates the View selected by the ViewComponent?

### Using ViewBag to pass additional data to a View

Here in chapter 08 page 191 we use a ViewBag. A ViewBag stores unstructured data and is dynamic. By dynamic I mean we can add new properties to the object as you can see in the image (2) (eg SelectedCategory).

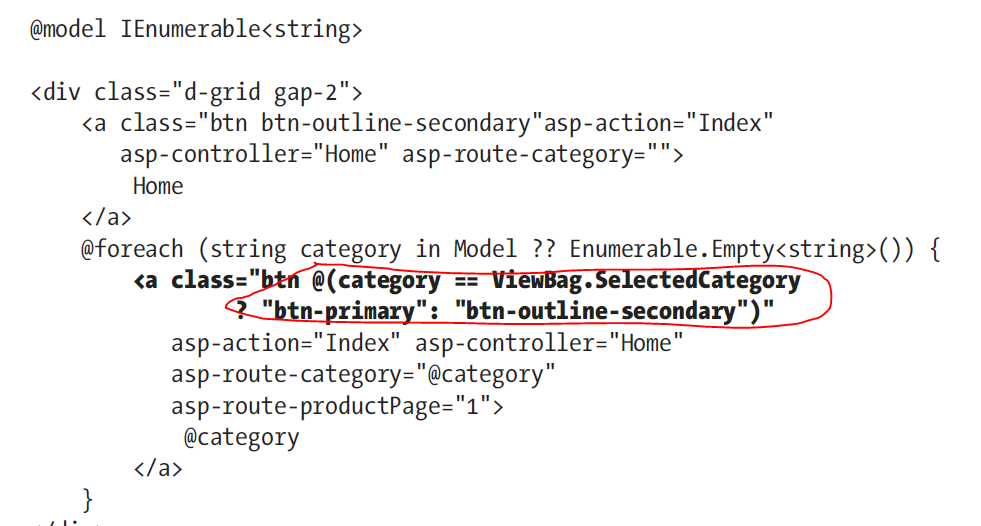


From the View, the ViewBag data is accessible through ViewData from ViewViewComponentResult:



### What is a Razor expression?

An example from chapter 08 page 193:



## Related to Razor Pages

Razor Pages is another framework supported by ASP.NET Core (other frameworks is MVC).

What you need to enable Razor pages:



### About terms/language

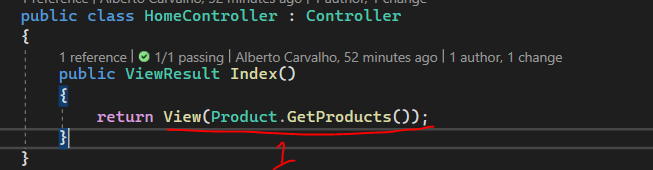
By terms/language I mean the way we express concepts.

* *Page model class for the cart razor page* – each razor page has associated a page model class. The class inherits from PageModel.

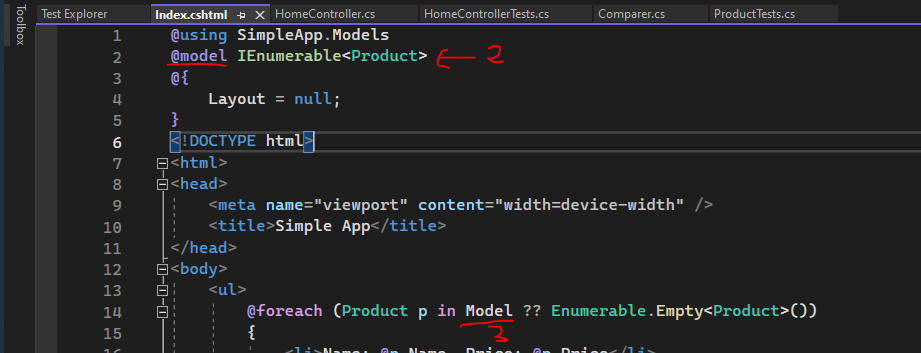
## Related to ViewResult class

### How to get the model from the ViewResult?

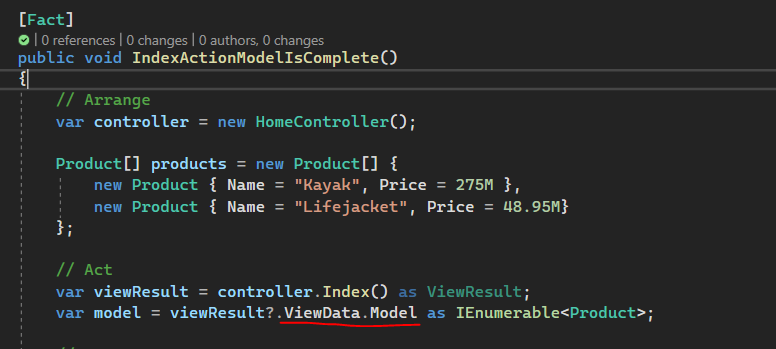
1. The View method in the Controller will produce a ViewResult that contain model data.



1. The model data is accessed in the View (cshtml file) through @model tag helper



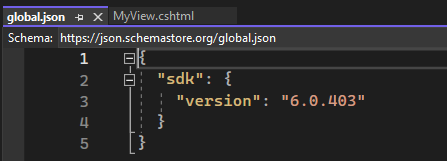
1. From the ViewResult class there is the ViewData property and this property contain the Model property where you can access the model data.

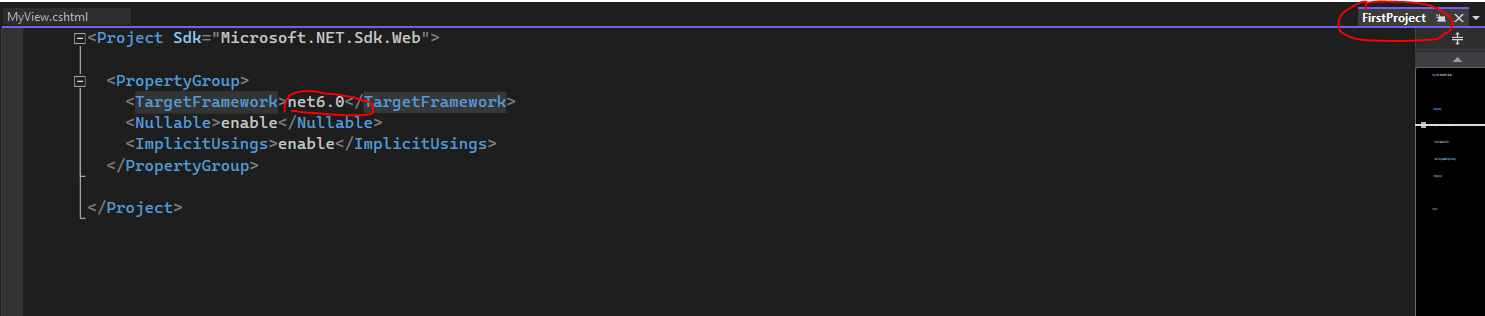


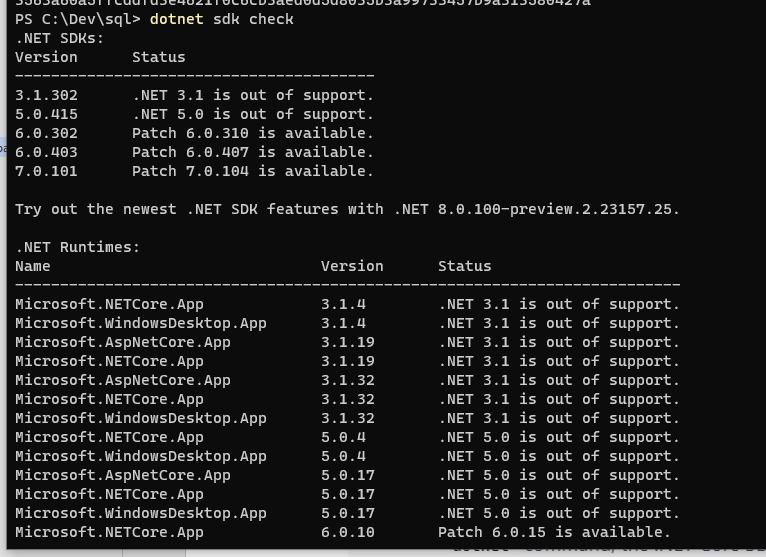
## Some Doubts

### In Chapter 2, we created a global.json file in a C# project. What is the role of this file?

<https://davidpine.net/blog/the-global-json/>





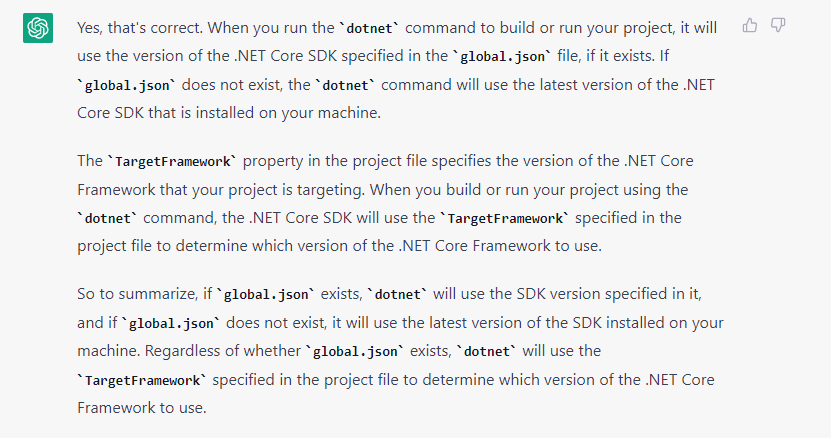


To understand this you need to know that one thing is .NET Core SDK and other totally different thing is .NET Core Framework! You also need to understand that the machine can contain multiple .NET Core SDK version’s and multiple .NET Core Framework’s!

The .NET CLI commands (eg dotnet) need to know which SDK version to use. If there is no global.json file, the latest version is used. If the file exist, dotnet command will use the sdk version specified. Then it will pass to SDK the .NET Core Framework version you have specified in the project file via TargetFramework property.

A given SDK supports a fixed set of .NET Core Frameworks:

*A given SDK supports a fixed set of frameworks, capped to the target framework of the runtime it ships with. For example, the .NET 5 SDK includes the .NET 5 runtime, which is an implementation of the net5.0 target framework. The .NET 5 SDK supports netcoreapp2.0, netcoreapp2.1, netcoreapp3.0, and so on, but not net6.0 (or higher). You install the .NET 6 SDK to build for net6.0.*

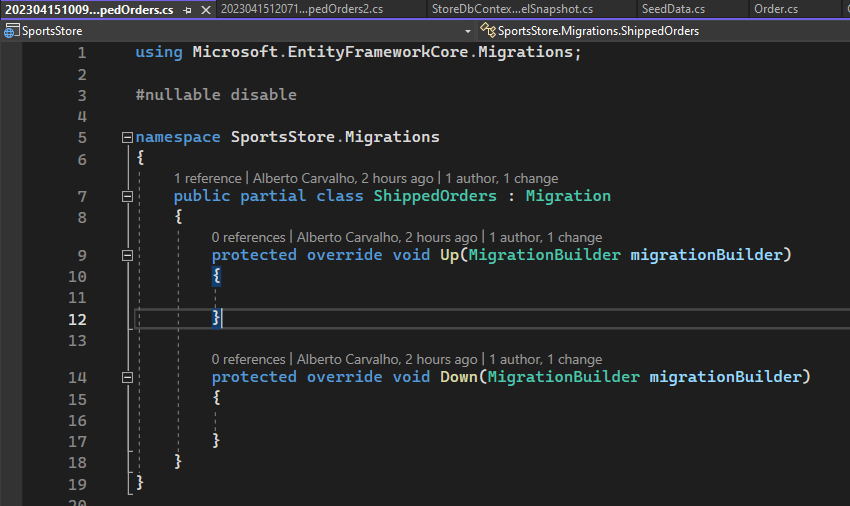


### In Chapter 07, try to understand better the interface IQueryable, page 151

### In Chapter 08, page 184, I have some doubts how urlHelper.Action method work to generate urls for “a” elements

### In Chapter 10, page 250, I had an issue creating the ShippedOrders migration. How to fix these issues in Entity Framework?

For some unknow reason, the migration was empty:



Then I created an additional migration to try removing the last one that was empty:

*dotnet ef migrations add remove*

But I did a mistake in the previously command, I used add remove instead of

*dotnet ef migrations remove*

Ok, now I understand my mistake! So we need to understand that the commands with this pattern ***dotnet ef migrations xxx*** then only apply to the project, not to the database.

This means that if we already applied a migration to a database and then we run this command:

*dotnet ef migrations remove*

You can get this error message:

*The migration '20230415120712\_ShippedOrders2' has already been applied to the database. Revert it and try again*.

Let assume we applied these migrations to the database:

* Migration1
* Migration2
* MigrationN

Now if you want to revert the database to Migration2 we need to run this command:

*dotnet ef database update Migration2*

After the previously command, now we can run this command to remove the last’s migration’s:

dotnet ef migration remove

We have commands for the project that follow this pattern:

*dotnet ef migrations*

We have commands that apply to the database:

*dotnet ef database*

# Fixing some Issues

## ArgumentException related to startup hook Microsoft.AspNetCore.Watch.BrowserRefresh.dll

*System.ArgumentException*

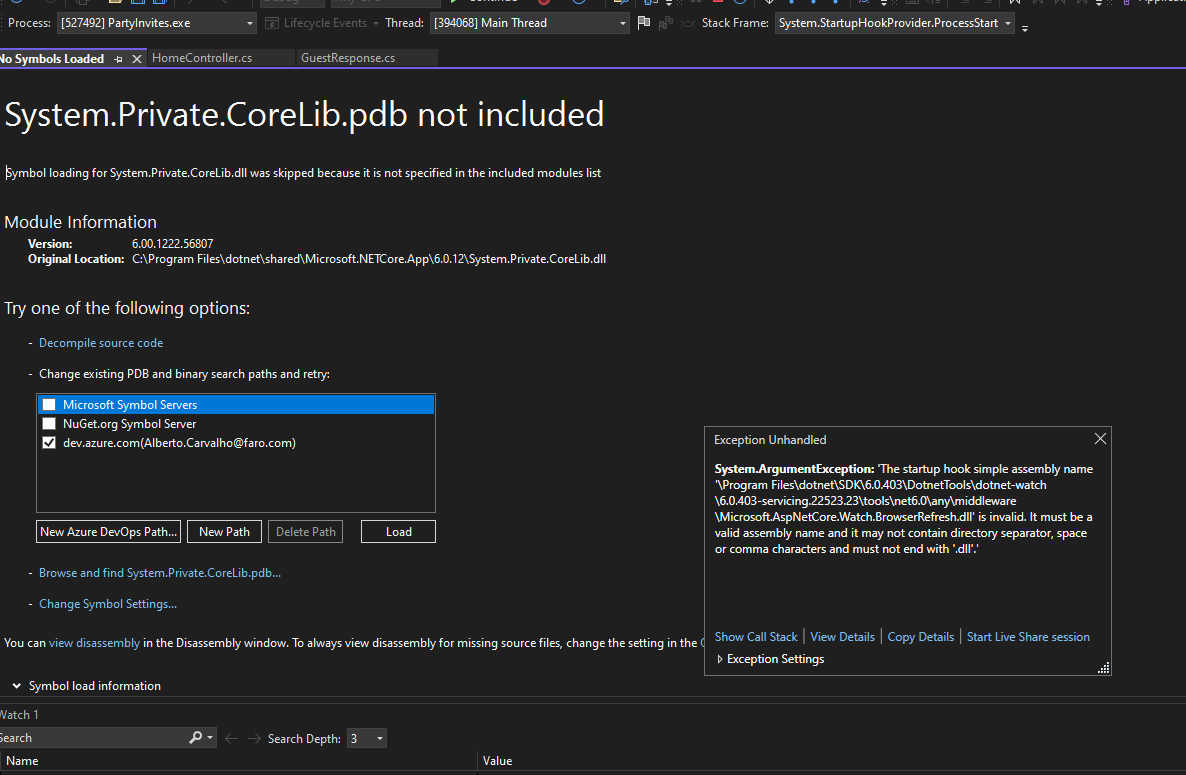
*HResult=0x80070057*

*Message=The startup hook simple assembly name '\Program Files\dotnet\SDK\6.0.403\DotnetTools\dotnet-watch\6.0.403-servicing.22523.23\tools\net6.0\any\middleware\Microsoft.AspNetCore.Watch.BrowserRefresh.dll' is invalid. It must be a valid assembly name and it may not contain directory separator, space or comma characters and must not end with '.dll'.*

*Source=System.Private.CoreLib*

*StackTrace:*

*at System.StartupHookProvider.ProcessStartupHooks()*

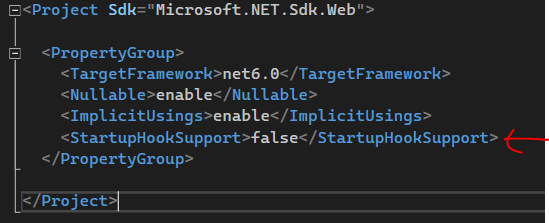


Steps to Fix

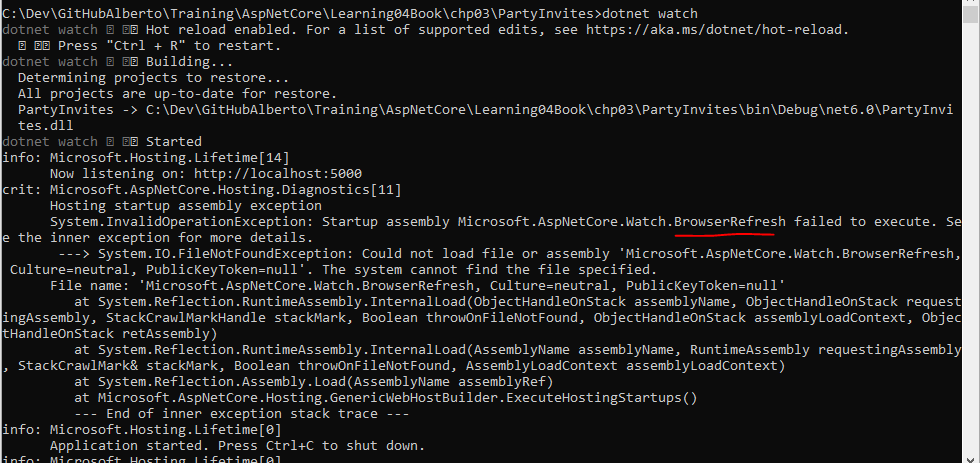
Edit the project

Add the following line in PropertyGroup:

<StartupHookSupport>false</StartupHookSupport>



But be aware that the dotnet watch command will throw an exception for this change:



# What I have Learned

## The move from MVC model to SPA (Single Page Application’s)

Page 4,

The MVC Framework remains an important part of ASP.NET Core, but the way it is commonly used has changed with the rise of single-page applications (SPAs). In an SPA, the browser makes a single HTTP request and receives an HTML document that delivers a rich client, typically written in a JavaScript client such as Angular or React. The shift to SPAs means that the clean separation that the MVC Framework was originally intended for is not as important, and the emphasis placed on following the MVC pattern is no longer essential, even though the MVC Framework remains useful (and is used to support SPAs through web services, as described in Chapter 19).