Software Development

Presented By:
Khaled Gamal

ASP.NET Core

MVC

Section 3

Outlines

- **Action selectors:**
  - **ActionName**
  - **NonAction**
- **ActionVerbs**
- **Introduction to Views**
- **Introduction to Razor**
- **Introduction to Layouts**
- **Passing Data between controllers and Views**

Teams Code is ***bgdylhk***

Action Selectors

Action Selectors

- *Action selector* is the *attribute* that can be applied to the action methods.
- In c#, *attributes* are classes that inherit from the *System.Attribute* base class directly or not directly, and they can be used as sort of “tag”, “metadata” or “declarative Information”, on other pieces of code (assemblies, types, methods, properties, and so forth).
- Attributes can be applied by giving their name, along with any arguments, inside square brackets[] just before the associated declaration.
- If an attribute's name ends in *Attribute*, that part of the name can be omitted when the attribute is referenced.
- Attributes can accept arguments in the same way as methods and properties.

Action Selectors

- *Actions Selectors* are attributes that can be applied to action methods and are used to influence which action gets invoked in response to a request.
- They help the routing engine to select the correct action method to handle a particular request.
- We have the following:
  - ActionName
  - NonAction
  - ActionVerbs

ActionName

- *ActionName*: a class represents an attribute that is used for the name of an action it also allows to use a different action name than the method name.

```
[ActionName("CurrentTime")]
public string testactions3()
{
    return DateTime.Now.ToString();
}
```

NonAction

- *NonAction:* indicates that a public method of a controller is not an action method.

```
[ActionName("CurrentTime")]
public string testactions3()
{
    return GetTime();
}

[NonAction]
public string GetTime()
{
    return DateTime.Now.ToString();
}
```

ActionVerbs

- The *ActionVerbs* selector is to handle different type of Http requests.
- *ActionVerbs:* restricts the indication of a specific action to specific *httpverbs* so you can define multiple different action methods with the same name but each responds different httpverb (HttpMethods).
- The MVC framework includes HttpGet, HttpPost, HttpPut, HttpDelete, HttpOptions, and HttpPatch action verbs.
- You can apply one or more action verbs to an action method to handle different HTTP requests.
- If you don't apply any action verbs to an action method, then it will handle HttpGet request by default.

ActionVerbs

- The following table lists the usage of HTTP methods:

Http method	Usage
GET	To retrieve the information from the server. Parameters will be appended in the query string.
POST	To create a new resource.
PUT	To update an existing resource.
HEAD	Identical to GET except that server do not return the message body. Returns only the head of response.
OPTIONS	It represents a request for information about the communication options supported by the web server.
DELETE	To delete an existing resource.
PATCH	To full or partial update the resource.

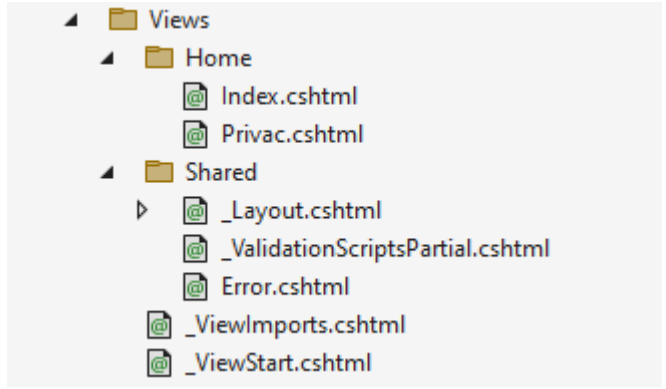
ActionVerbs

```
[HttpPost]
public string search(string query)
{
    return $"returns the result of the query: {query}";
}
[HttpGet]
public string search()
{
    return "hello from GET search";
}
```

Introduction to Views

- The responsibility of a view is to transform a model into HTML or other formats.
- In the Model-View-Controller (MVC) pattern, the view handles the app's data presentation and user interaction.
- In ASP.NET Core MVC, views are .cshtml (HTML templates) files that use the C# programming language in Razor markup.
- Razor markup is code that interacts with HTML markup to produce a webpage that's sent to the client.

Introduction to Views



- A controller can have one or more action methods, and each action method can return a different view.
- In short, a controller can render one or more views.
- So, for easy maintenance, the MVC framework requires a separate sub-folder for each controller with the same name as a controller, under the Views folder.
- The Home controller is represented by a Home folder inside the Views folder. The Home folder contains the views for the Privacy and Index (homepage) webpages.
- When a user requests one of these two webpages, controller actions in the Home controller determine which of the two views is used to build and return a webpage to the user.

Returning a view from action

- Views are typically returned from actions as a *ViewResult*, which is a type of *ActionResult*.
- Since most controllers inherit from Controller, you simply use the View helper method to return the *ViewResult*:

- The View helper method has several overloads.
- You can optionally specify:
  - An explicit view to return:
  - A model to pass to the view:
  - Both a view and a model:
  - Both a view and a Layout:

```
return View("Orders");
```

```
return View(Orders);
```

```
return View("Orders", Orders);
```

```
return View("Orders", “_MainLayout");
```

View discovery

- When an action returns a view, a process called view discovery takes place.
- This process determines which view file is used based on the view name.
- The default behavior of the View method *(return View();)* is to return a view with the same name as the action method from which it's called.
- For example, the About ActionResult method name of the controller is used to search for a view file named About.cshtml.
- First, the runtime looks in the Views/[ControllerName] folder for the view.
- If it doesn't find a matching view there, it searches the Shared folder for the view.

View discovery

- It doesn't matter if you implicitly return the *ViewResult* with *return View();* or explicitly pass the view name to the View method with *return View("<ViewName>");*.
- In both cases, view discovery searches for a matching view file in this order:
  1. Views/\[ControllerName]/\[ViewName].cshtml
  2. Views/Shared/\[ViewName].cshtml
- A view file path can be provided instead of a view name.
- If using an absolute path starting at the app root (optionally starting with "/" or "~/"), the .cshtml extension must be specified:

```
return View("Views/Home/About.cshtml");
```

View discovery

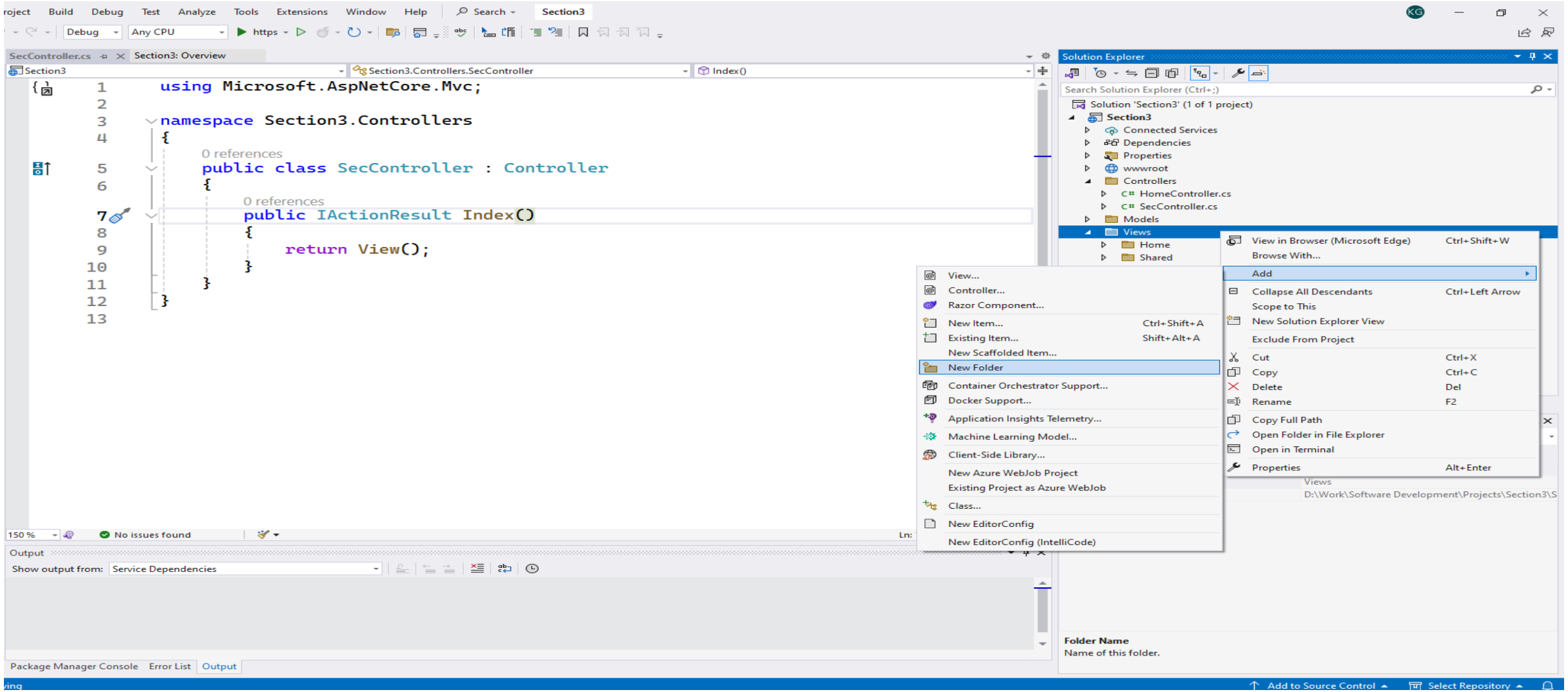
- You can also use a relative path to specify views in different directories without the .cshtml extension.
- Inside the HomeController, you can return the Index view of your Manage views with a relative path: `return View(“../Manage/Index");`
- Similarly, you can indicate the current controller-specific directory with the "./" prefix: `return View(“./Index");`

- **Note:** If the underlying file system is case sensitive, view names are probably case sensitive.

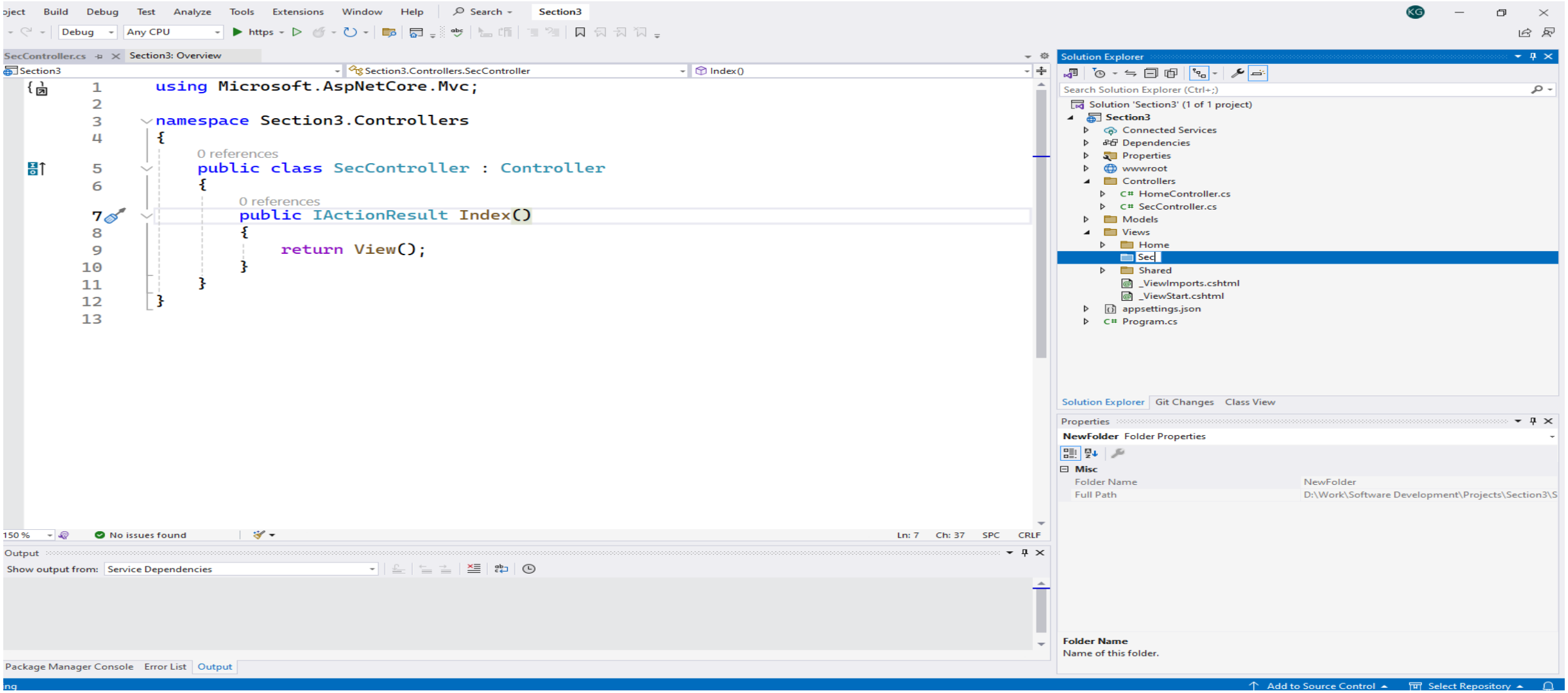
Creating View

- Views that are specific to a controller are created in the Views/[ControllerName] folder.
- Views that are shared among controllers are placed in the Views/Shared folder.
- The **Shared** folder contains views, layout views, and partial views, which will be shared among multiple controllers.
- To create a view, add a new file and give it the same name as its associated controller action with the .cshtml file extension.

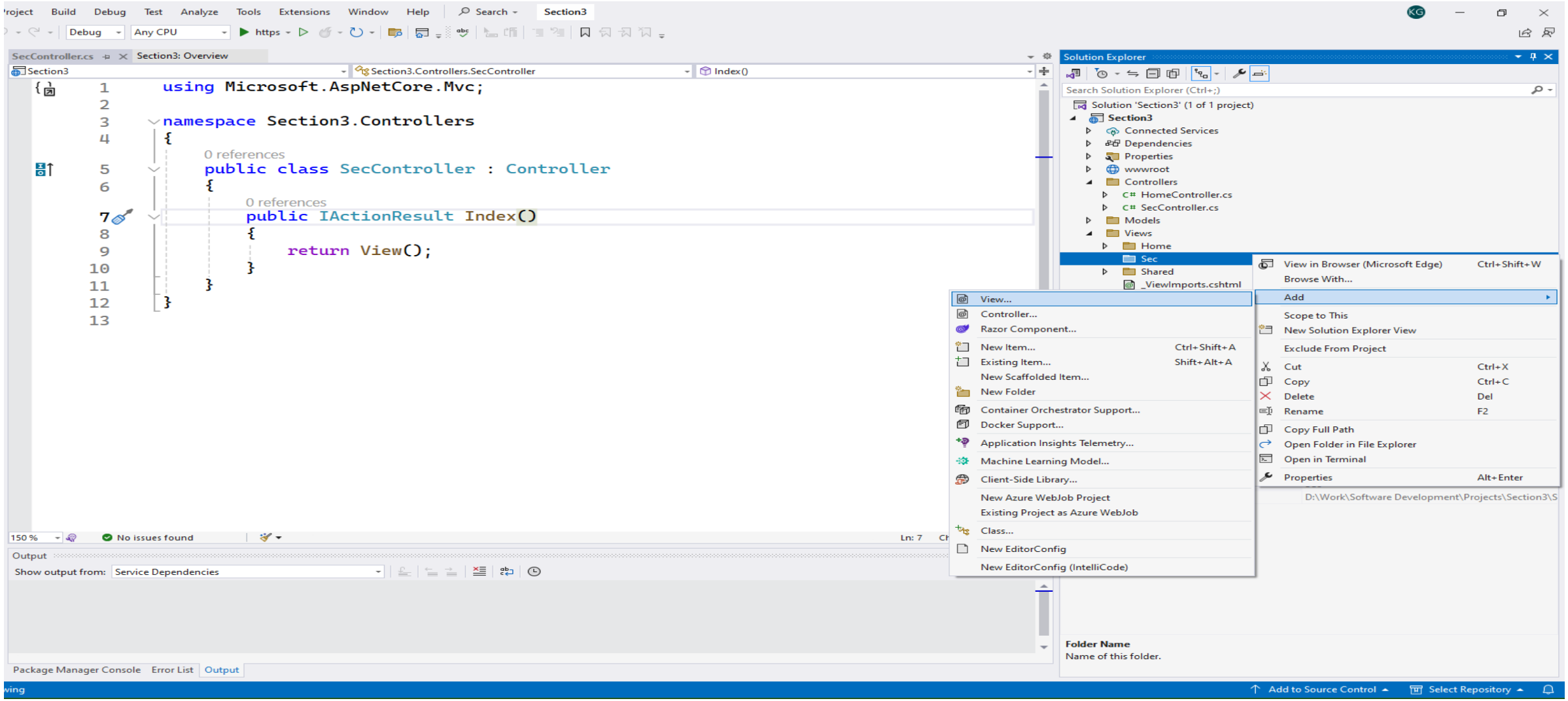
Creating View



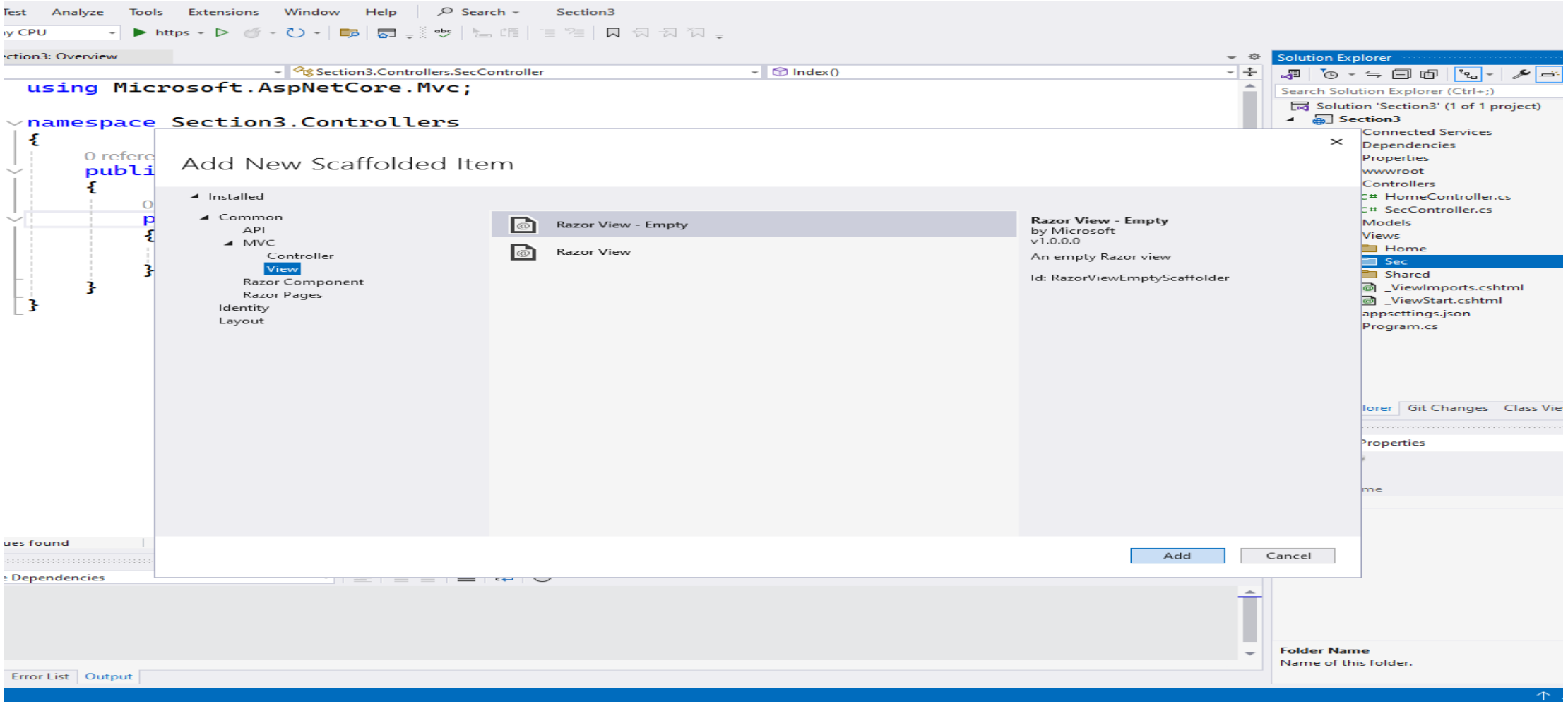
Creating View



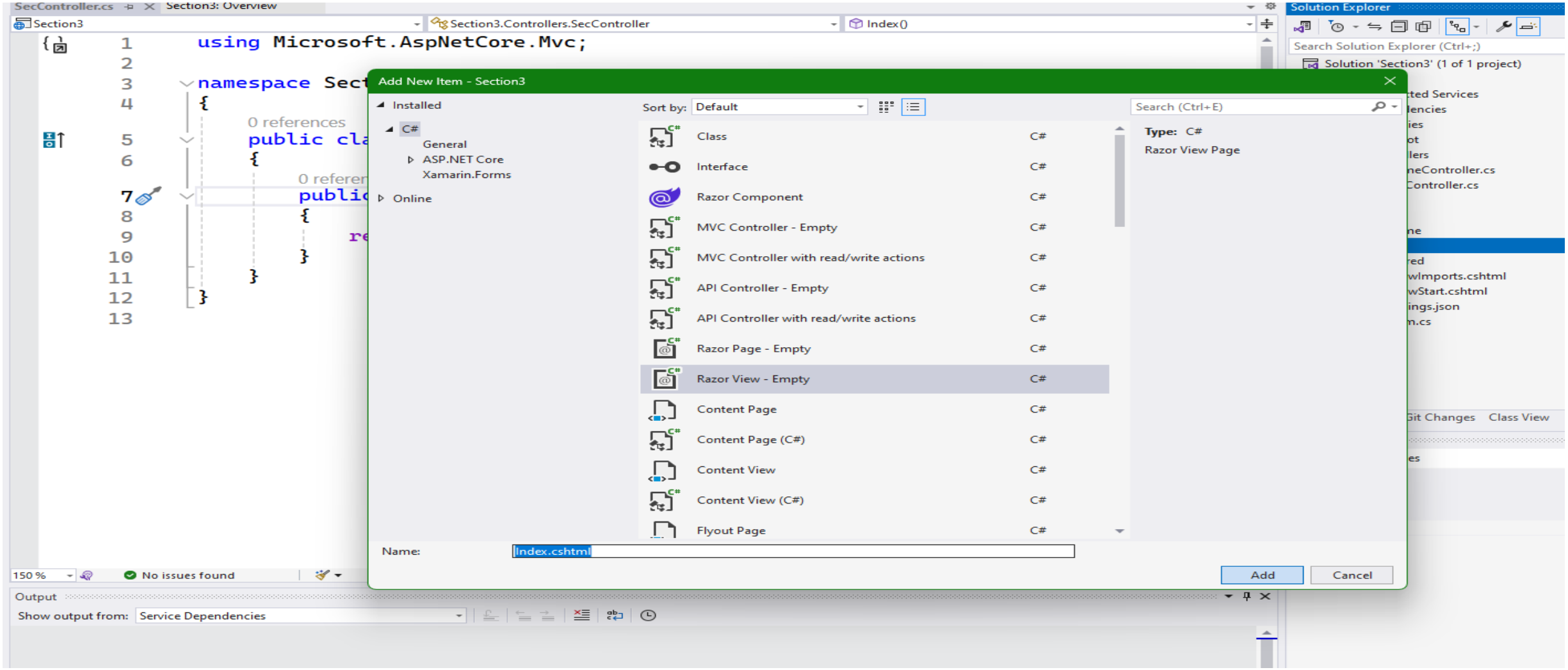
Creating View



Creating View

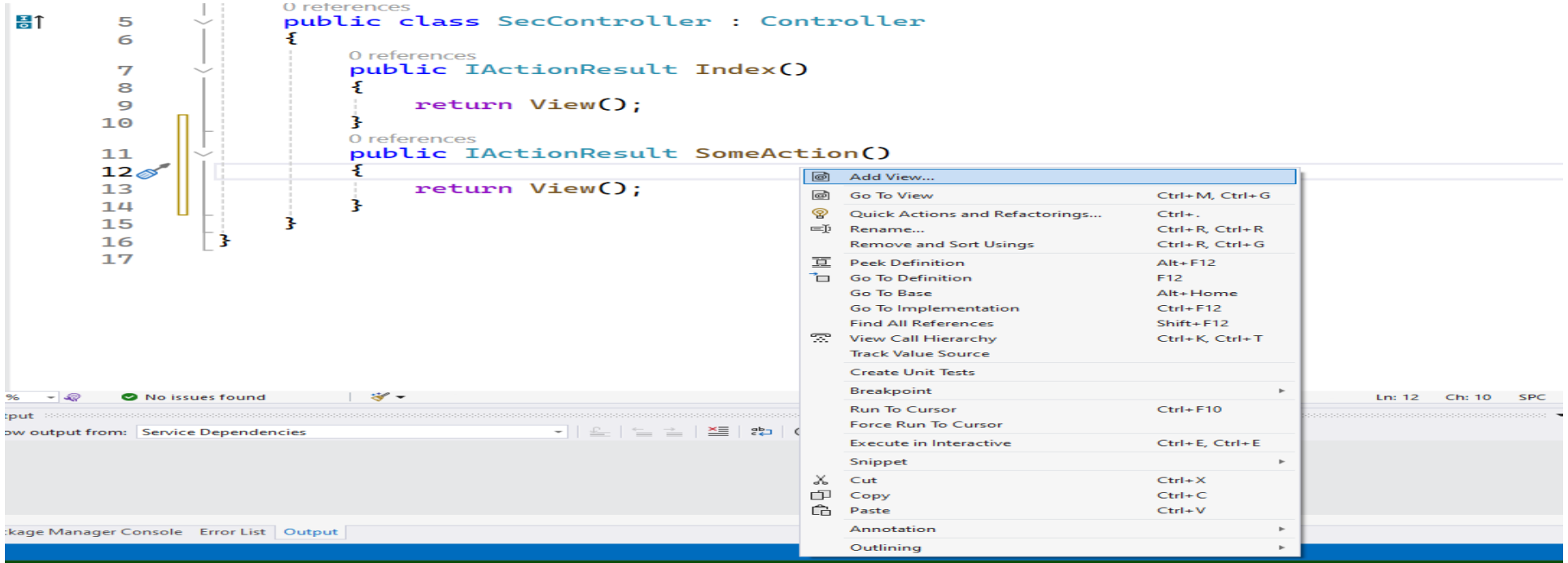


Creating View

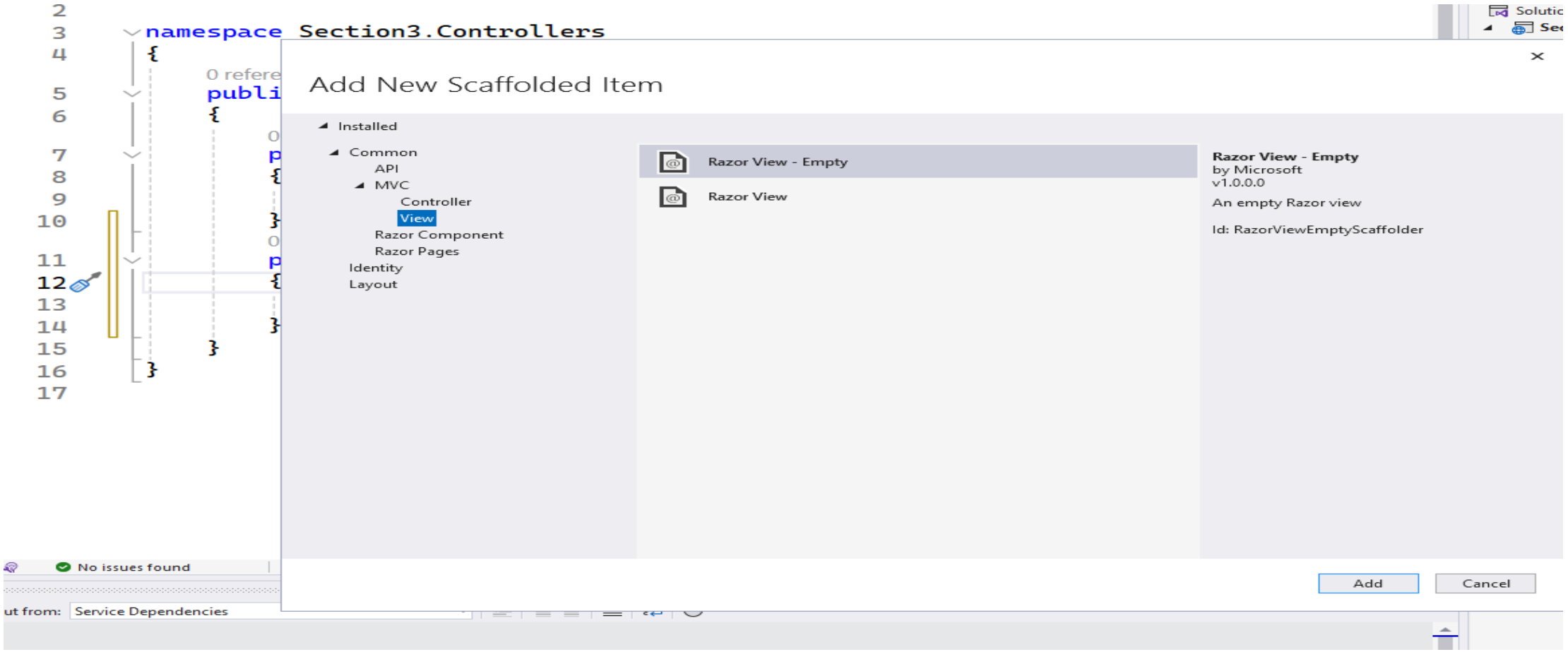


Creating View

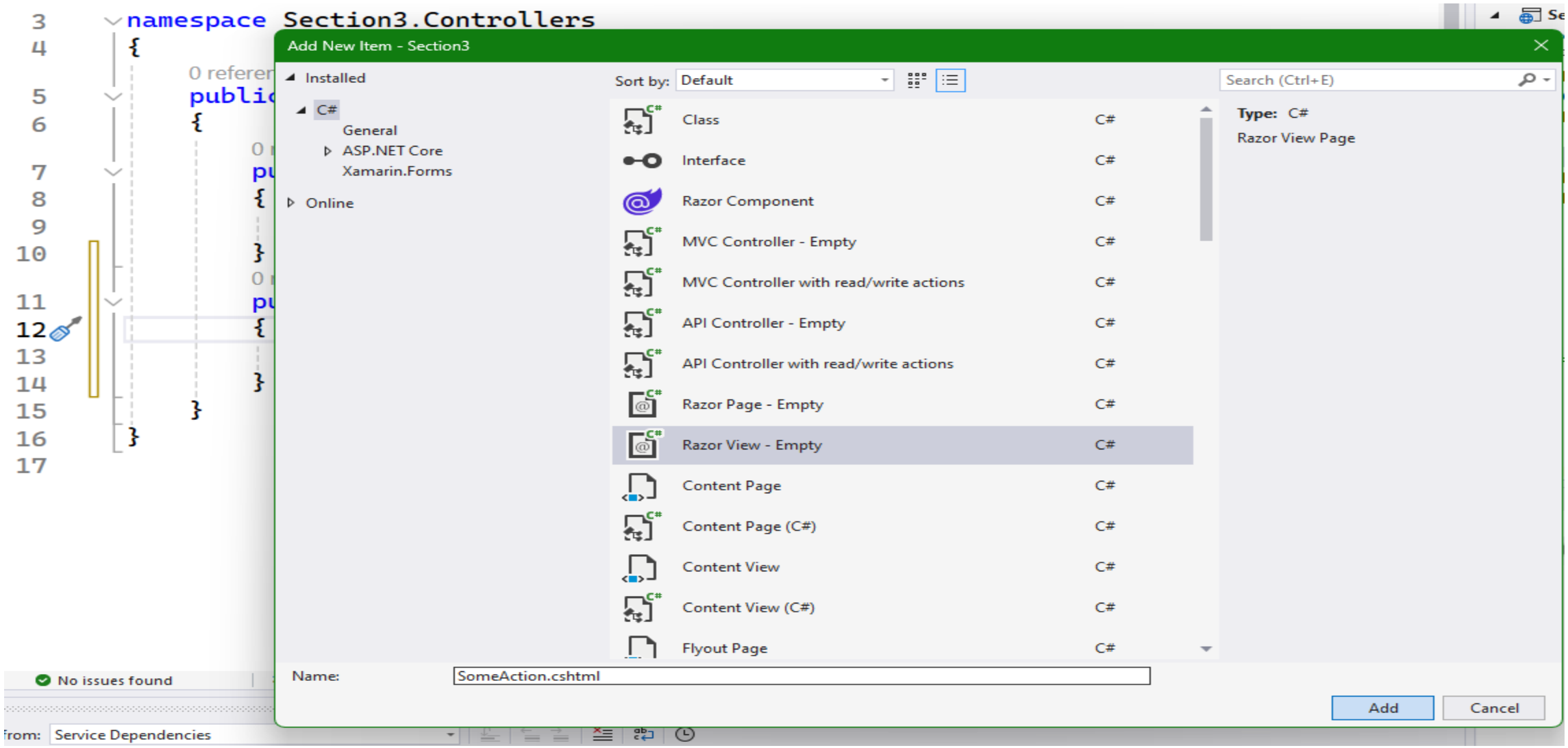
- Or inside the body of the action right-click and select add view.
- Automatically you will find a (controllerName) folder created in Views folder and a view called (actionName).cshtml created inside it.



Creating View



Creating View



Introduction to Razor

- Razor is a markup syntax for embedding .NET based code into webpages.
- The Razor syntax consists of Razor markup, C#, and HTML.
- Files containing Razor generally have a .cshtml file extension.
- Razor syntax is similar to the templating engines of various JavaScript single-page application (SPA) frameworks, such as Angular, React, VueJs, and Svelte.
- Razor markup starts with the @ symbol.

Rendering HTML

- The default Razor language is HTML.
- Rendering HTML from Razor markup is no different than rendering HTML from an HTML file.
- HTML markup in .cshtml Razor files is rendered by the server unchanged.

Razor syntax

- Razor supports C# and uses the @ symbol to transition from HTML to C#.
- Razor (Razor view engine) evaluates C# expressions and renders them in the HTML output.
- To escape an @ symbol in Razor markup, use a second @ symbol:

```
<p>@@Username</p>
```

- The code is rendered in HTML with a single @ symbol:

```
<p>@Username</p>
```

Razor syntax

- HTML attributes and content containing email addresses don't treat the @ symbol as a transition character.
- The email addresses in the following example are untouched by Razor parsing:

```
<a href="mailto:Support@contoso.com">Support@contoso.com</a>
```

Implicit Razor expressions

- Implicit Razor expressions start with @ followed by C# code:

```
<p>@DateTime.Now</p>
```

- With the exception of the C# await keyword, **implicit expressions must not contain spaces**. `<p>@awiat DoSomethingAsync()</p>`
- Implicit expressions **cannot** contain C# generics, as the characters inside the brackets (<>) are interpreted as an HTML tag.
- The following code is not valid: `<p>@GenericMethod<int>()</p>`

Explicit Razor expressions

- Explicit Razor expressions consist of an @ symbol with balanced parenthesis ().
- Any content within the @() parenthesis is evaluated and rendered to the output.

```
<p>Last week this time: @(DateTime.Now - TimeSpan.FromDays(7))</p>
```

- Explicit expressions can be used to concatenate text with an expression result:

```
@{
    var joe = new Person("Joe", 33);
 }
< p>Age@(joe.Age)</p>
```

- Without the explicit expression, * Age@joe.Age * is treated as an email address, and * Age@joe.Age * is rendered.
- When written as an explicit expression, * Age33 * is rendered.

Explicit Razor expressions

- Explicit expressions can be used to render output from generic methods in .cshtml files.
- The following markup shows how to correct the error shown earlier caused by the brackets of a C# generic.

```
<p>@(GenericMethod<int>())</p>
```

Razor code blocks

- Razor code blocks start with @ and are enclosed by {} .
- Unlike expressions, C# code inside code blocks isn't rendered.
- Code blocks and expressions in a view share the same scope and are defined in order:

CSHTML

```
@{
    var quote = "The future depends on what you do today. - Mahatma Gandhi";
}
<p>@quote</p>
@{
    quote = "Hate cannot drive out hate, only love can do that. - Martin Luther King, Jr.";
}
<p>@quote</p>
```

HTML

```
<p>The future depends on what you do today. - Mahatma Gandhi</p>

<p>Hate cannot drive out hate, only love can do that. - Martin Luther King, Jr.</p>
```

Razor code blocks

- In code blocks, declare local functions with markup to serve as templating methods:

CSHTML

```
@{
    void RenderName(string name)
{
  <p>Name: <strong>@name</strong></p>
}
RenderName("Mahatma Gandhi");
RenderName("Martin Luther King, Jr.");
}
```

HTML

```
<p>Name: <strong>Mahatma Gandhi</strong></p>
<p>Name: <strong>Martin Luther King, Jr.</strong></p>
```

Transitions

- **Implicit transitions**
- The default language in a code block is C#, but the Razor Page can transition back to HTML but to transition back to c# use @:

CSHTML

```
@{
  var inCSharp = true;
  <p>Now in HTML, was in C# @inCSharp</p>
}
```

- **Explicit delimited transition**
- To define a subsection of a code block that should render HTML (Display text from code block), surround the characters for rendering with the Razor <text> tag:

CSHTML

```
@{
var MyName = “Khaled”;
<text>Name: @MyName</text>
}
```

Only the content between the <text> tag is rendered.
No whitespace before or after the <text> tag appears in the HTML output.

Transitions

- **Explicit line transition**
- To render the rest of an entire line as HTML (Display text from code block) inside a code block, use @: syntax:

CSHTML

```
@{
    var MyName = “Khaled”;
    @:Name: @MyName
}
```

- Without the @: in the code, a Razor runtime error is generated.

Control structures

- Control structures are an extension of code blocks.
- All aspects of code blocks (transitioning to markup, inline C#) also apply to the following structures:
- **Conditionals @if, else if, else, and @switch**
- *@if* controls when code runs, *else* and *else if* don't require the *@* symbol:

CSHTML

```
@{
var value = 50;
}
@if(value==0)
{
    <p>value is zero</p>
}
else if(value %2 == 0)
{
    <p>@value is EVEN</p>
}
else
{
    <p>@value is ODD</p>
}
```

Control structures

- The following markup shows how to use a *switch* statement:

CSHTML

```
@{
var value = 49;
}
@switch(value)
{
    case 0:
        <p>value is Zero</p>
        break;
    case 50:
        <p>@value is 50!</p>
        break;
    default:
        <p>@value is @value</p>
        break;
}
```

Control structures

- **Looping *@for*, *@foreach*, *@while*, and *@do while***
- Templated HTML can be rendered with looping control statements.
- The following looping statements are supported:
- *@for*

CSHTML

```
@for(int i=0;i<10;i++)
{
    <p>hello @i</p>
}
```

Control structures

- *@foreach*

CSHTML

```
@{
    List<string> strings = new List<string> { "khaled", "Gamal" };
}
@foreach(var s in strings)
{
    <h1>@s</h1>
}
```

- *@while*

CSHTML

```
@{
    List<string> strings = new List<string> { "khaled", "Gamal" };
    var c = 0;
}
@while(c < strings.Count)
{
    <h2>@strings[c]</h2>
    c++;
}
```

- *@do while*

CSHTML

```
@{
    List<string> strings = new List<string> { "khaled", "Gamal" };
    var c = 0;
}
@do
{
    <h5>@strings[c]</h5>
    c++;
}while(c<strings.Count);
```

Control structures

- *@try, catch, finally*

CSHTML

```
@try
{
    throw new InvalidOperationException("You did something invalid.");
}
catch (Exception ex)
{
    <p>The exception message: @ex.Message</p>
}
finally
{
    <p>The finally statement.</p>
}
```

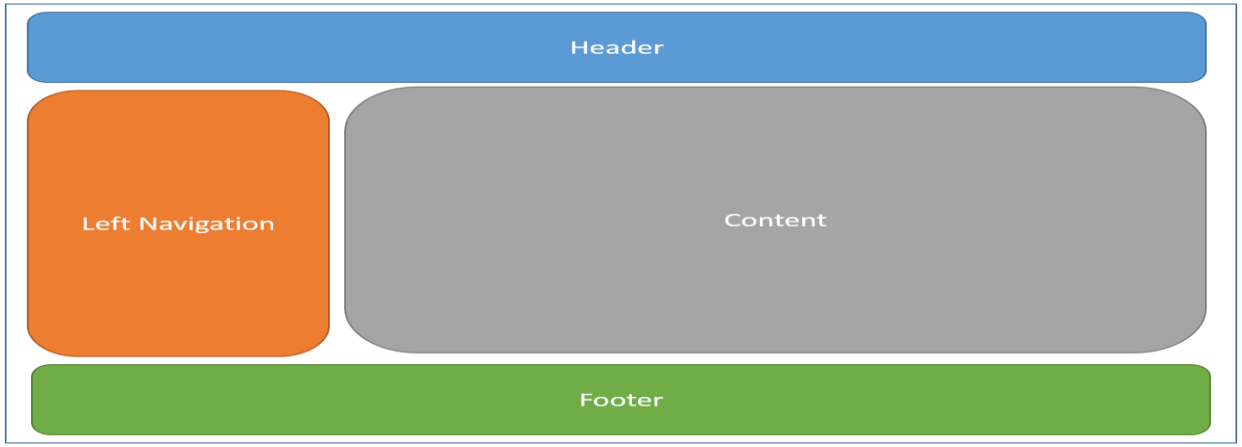
- *Comments:* Razor supports C# and HTML comments, Razor uses *@* *@* to delimit comments.

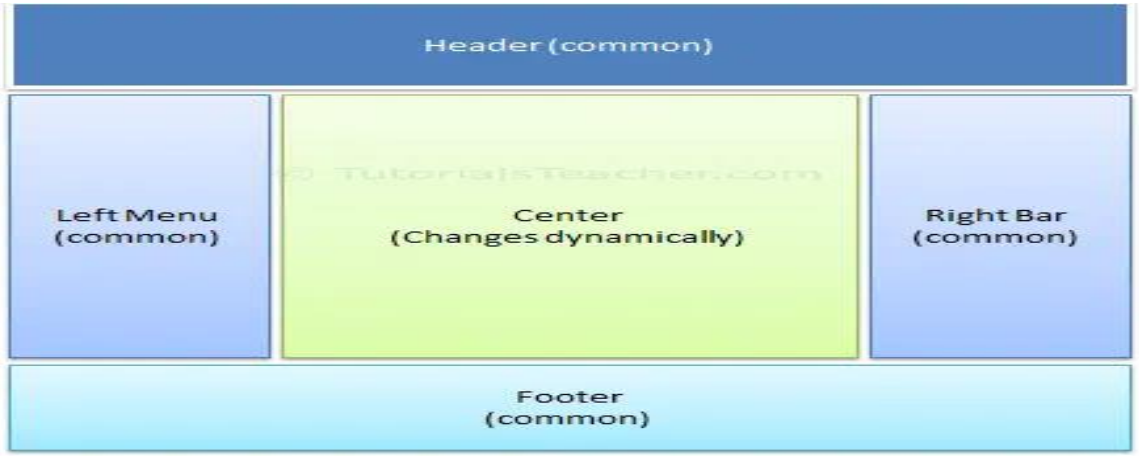
CSHTML

```
@*
@{
     /* C# comment */
     // Another C# comment
}
<!-- HTML comment -->
*@
```

Introduction to Layouts

- `If you select every page like Section3, Home, and Privacy.`
- Each page shows the same menu layout.
- Most web apps have a common layout that provides the user with a consistent experience as they navigate from page to page.
- The layout typically includes common user interface elements such as the app header, navigation or menu elements, and footer.





Introduction to Layouts

- Layout templates allow:
  - Specifying the HTML container layout of a site in one place.
  - Applying the HTML container layout across multiple pages in the site.
  - Make it easy to make changes that apply across all of the pages in an app.

Introduction to Layouts

- Layout has a placeholder for the center section that changes dynamically, as shown below.

- Common HTML structures such as scripts and stylesheets are also frequently used by many pages within an app.
- All of these shared elements may be defined in a layout file, which can then be referenced by any view used within the app.

Introduction to Layouts

- The layout view has the same extension as other views, .cshtml.
- By convention, the default layout for an ASP.NET Core app is named *_Layout.cshtml* which is implemented in the *Views/Shared/_Layout.cshtml* file.
- Apps can define more than one layout, with different views specifying different layouts.

Specifying a Layout

- Razor views have a Layout property. Individual views specify a layout by setting this property: `@{Layout = "_Layout";}`
- The Layout property can be used to set a different layout view or set it to null so no layout file will be used.
- The layout specified can use a full path (for example, */Views/Shared/_Layout.cshtml*) or a partial name (example: *_Layout*).
- When a partial name is provided, the Razor view engine searches for the layout file using its standard discovery process. The folder where the handler method (or controller) exists is searched first, followed by the Shared folder.

RenderBody Method

- By default, every layout must call *RenderBody*. Wherever the call to *RenderBody* is placed, the contents of the view will be rendered.
- Find the *@RenderBody()* line. RenderBody is a placeholder where all the view-specific pages you create show up, wrapped in the layout page.
- For example, if you select the Privacy link, the Views/Home/Privacy.cshtml *view* is rendered inside the *RenderBody* method.
- The *RenderBody* method injects the contents of a specific Razor view for a page like the *Index.cshtml* file at that point in the shared layout.
- The content in the *Index.cshtml* view template is merged with the *Views/Shared/_Layout.cshtml* view template. A single HTML response is sent to the browser.

_ViewStart.cshtml

```cshtml
@{
    Layout = "_Layout";
}
```

- The *Views/_ViewStart.cshtml* file brings in the *Views/Shared/_Layout.cshtml* file to each view.
- The file above specifies that all views will use the _Layout.cshtml layout.
- Code that needs to run before each view should be placed in the *_ViewStart.cshtml* file.
- The statements listed in *_ViewStart.cshtml* are run before every full view (not layouts, and not partial views).

Importing Shared Directives

- Views and pages can use Razor directives to import namespaces and use dependency injection.
- Directives shared by many views may be specified in a common *_ViewImports.cshtml* file placed in the Views folder.
- The _ViewImports file supports the following directives and more and doesn't support other Razor features, such as functions and section definitions:
  - @addTagHelper
  - @removeTagHelper
  - @tagHelperPrefix
  - @using
  - @model
  - @inherits
  - @inject
  - @namespace

Tag Helpers

CSHTML

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

- The following **Views/_ViewImports.cshtml** file
- The *@addTagHelper* directive makes Tag Helpers available to the all views.
- The first parameter after *@addTagHelper* specifies the Tag Helpers to load (*we are using "*" for all Tag Helpers*), and the second parameter *"Microsoft.AspNetCore.Mvc.TagHelpers"* specifies the assembly containing the Tag Helpers.
- Microsoft.AspNetCore.Mvc.TagHelpers is the assembly for the built-in ASP.NET Core Tag Helpers.
- The code above uses the wildcard syntax ("*") to specify that all Tag Helpers in the specified assembly (Microsoft.AspNetCore.Mvc.TagHelpers) will be available to every view file in the Views directory or subdirectory.

Passing Data from the Controller to the View

- Controllers are responsible for providing the data required in order for a view template to render a response.
- View templates should **not**:
  - Do business logic
  - Interact with a database directly
- A view template should work only with the data that's provided to it by the controller.
- Maintaining this "separation of concerns" helps keep the code:
  - Clean.
  - Testable.
  - Maintainable.

Passing Data from the Controller to the View

- You can pass data in a strongly typed approach (next sections):
  - With models
  - Dynamic Views
- Also, You can pass data in weakly typed data approach using:
  - ViewData
  - [ViewData] attribute
  - ViewBag
  - TempData
  - [TempData] attribute
- Weak types (or loose types) means that you don't explicitly declare the type of data you're using.
- You can use the collection of weakly typed data for passing small amounts of data in and out of controllers and views.

Temp Data

- The temp data feature allows a controller to preserve data from one request to another, which is useful when performing redirections.
- *TempData* is used to transfer data:
  - From controller to view.

CS (in controller)

```
public IActionResult SomeAction()
{
    List<string> list = new List<string>();
    list.AddRange(new[] { "Egypt", "Turkey" });
    TempData["Countries"] = list;
    return View();
}
```

CSHTML (in View)

```
@foreach (var cn in TempData["Countries"]as List<string>)
{
    <h1>@cn</h1>
}
```

Temp Data

- TempData is used to transfer data:
  - From view to controller.

CSHTML (in SomeAction View)

```
@{
    TempData["render"] = false;
}
```

CS (in controller)

```
public IActionResult AnAction()
{
    if ((bool)TempData["render"])
    {
        return Content("render was true");
    }
    else
    {
        return Content("render was false");
    }
}
```

Temp Data

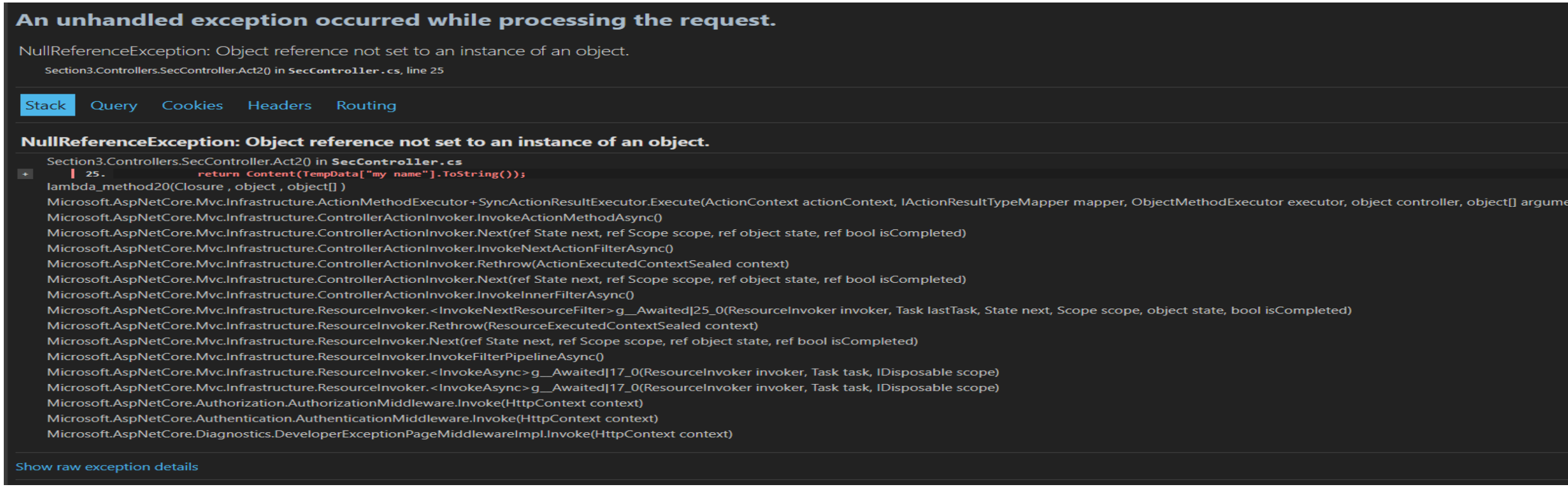
- TempData is used to transfer data:
  - from one action method to another action method of the same or a different controller.

CS (in controller)

```
public IActionResult Act1()
{
    TempData["my name"] = "khaled";
    return RedirectToAction("Act2");
}
public IActionResult Act2()
{
    return Content(TempData["my name"].ToString());
}
```

Temp Data

- TempData stores the data temporarily and automatically removes it after retrieving a value.



Temp Data

- Although, TempData removes a key-value once accessed, you can still keep it for the subsequent request by call in *TempData.Keep()* method to retain TempData value for the subsequent requests even after accessing it:
  - *TempData.Keep("name");* // Marks the specified key in the TempData for retention.
  - *TempData.Keep();* // Marks all keys in the TempData for retention

CS (in controller)

```
public IActionResult Act1()
{
    TempData["my name"] = "khaled";
    return RedirectToAction("Act2");
}
public IActionResult Act2()
{
    string res = TempData["my name"].ToString();
    TempData.Keep();
    return Content(res);
}
```

THE TEMP DATA ATTRIBUTE

- Controllers can define properties that are decorated with the TempData attribute, which is an alternative to using the TempData property.
- The values assigned to these properties are automatically added to the temp data store, and there is no difference in the way they are accessed in the view.

CSHTML (in View)

```
@TempData["TempProp"]
```

CS (in controller)

```
public class SecController : Controller
{
    [TempData]
    public string TempProp { get; set; }
    public IActionResult Index()
    {
        TempProp = "Hello from attributes";
        return View();
    }
}
```

ViewData

- **ViewData** is a ViewDataDictionary object accessed through string keys.
- The ViewData dictionary is a dynamic object, which means any type can be used.
- For this dictionary type, the key is a string and the value is an object.
- The ViewData object has no defined properties until something is added.
- String data can be stored and used directly without the need for a cast, but you must cast other ViewData object values to specific types when you extract them.
- You can use ViewData to pass data from controllers to views and within views, including partial views and layouts.
- The ViewData property is shared with the associated view, including the view's layout. As a result, any data stored in the ViewData dictionary is available to the view and its layout.

ViewData

- ViewData transfers data from the Controller to View, not vice-versa.
- ViewData's life only lasts during the current HTTP request. ViewData values will be cleared if redirection occurs.
- Let’s try Example:
- https://localhost:{PORT}/sec/Welcome?Message=Hello&numtimes=4
- Data is taken from the URL and passed to the controller using the MVC model binder.
- The controller packages the data into a ViewData dictionary and passes that object to the view. The view then renders the data as HTML to the browser.

CSHTML (in View)

```
@for (int i = 0; i <
(int)ViewData["Number"]; i++)
{
    <p>@ViewData["Message"]</p>
}
```

CS (in controller)

```
public IActionResult welcome(string Message, int numtimes)
{
    ViewData["Message"] = Message;
    ViewData["Number"] = numtimes;
    return View();
}
```

ViewData

- You can also add KeyValuePair objects into the ViewData, as shown below.

Example: Add KeyValuePair in ViewData

```
public ActionResult Index()
{
    ViewData.Add("Id", 1);
    ViewData.Add(new KeyValuePair<string, object>("Name", "Bill"));
    ViewData.Add(new KeyValuePair<string, object>("Age", 20));
    return View();
}
```

[ViewData] attribute

- Another approach that uses the ViewDataDictionary is ViewDataAttribute.
- Properties on controllers marked with the [ViewData] attribute have their values stored and loaded from the dictionary.

CSHTML (in View)

```
@ViewData["TempProp"]
```

CS (in controller)

```
public class SecController : Controller
{
    [ViewData]
    public string TempProp { get; set; }
    public IActionResult Index()
    {
        TempProp = "Hello from ViewData attribute";
        return View();
    }
}
```

ViewBag

- *ViewBag* is a Microsoft.AspNetCore.Mvc.ViewFeatures.Internal.DynamicViewData object that provides dynamic access to the objects stored in *ViewData*.
- *ViewBag* can be more convenient to work with, *since it doesn't require casting*.
- Since the ViewBag is dynamically typed, you don’t need to explicitly cast property values to work with the data that they contain.
- The ViewBag property is inherited from the Controller base class and returns a dynamic object.
- This allows action methods to create new properties just by assigning values to them.
- .NET determines the type of those properties at runtime.

ViewBag

- ViewBag doesn't require typecasting while retrieving values from it. This can throw a run-time exception if the wrong method is used on the value.
- ViewBag is a dynamic type and skips compile-time checking. So, ViewBag property names must match in controller and view while writing it manually.
- The values assigned to the ViewBag property by the action method are available to the view through a property also called ViewBag.
- Like the ViewData property, the ViewBag property is shared with the associated view, including its layout. As a result, any data stored in the ViewBag is available to the view and its layout.

ViewBag

- ViewBag only transfers data from controller to view, not visa-versa. ViewBag values will be null if redirection occurs.

CS (in controller)

```
public IActionResult CoursesNames()
{
    List<string> CN = new List<string>() { "Software development", "OOP", "OS" };
    ViewBag.CN = CN;
    return View();
}
```

CSHTML (in View)

```
@foreach(var n in ViewBag.CN)
{
    <h1>@n</h1>
}
```

Using ViewData and ViewBag simultaneously

- Under the hood, the ViewBag property stores its data in the ViewData dictionary. Thus, you can use the ViewData property to get values that were added to the ViewBag property.
- Also,the ViewBag property is a wrapper around ViewData that provides dynamic properties for the underlying ViewData collection.
- Since ViewData and ViewBag refer to the same underlying ViewData collection, you can use both ViewData and ViewBag and mix and match between them when reading and writing values.
- **Note:** Key lookups are case-insensitive for both ViewData and ViewBag.

Using ViewData and ViewBag simultaneously

CS (in controller)

```
public IActionResult CoursesNames()
{
    List<string> CN = new List<string>() { "Software development", "OOP", "OS" };
    ViewData["CN"] = CN;
    return View();
}
```

CSHTML (in View)

```
@foreach(var n in ViewBag.CN)
{
    <h1>@n</h1>
}
```

CS (in controller)

```
public IActionResult CoursesNames()
{
    List<string> CN = new List<string>() { "Software development", "OOP", "OS" };
    ViewBag.CN = CN;
    return View();
}
```

CSHTML (in View)

```
@foreach(var n in ViewData["CN"] as List<string>)
{
    <h1>@n</h1>
}
```

The differences between ViewData and ViewBag

- ViewData
  - Derives from ViewDataDictionary, so it has dictionary properties that can be useful, such as *ContainsKey, Add, Remove, and Clear*.
  - Keys in the dictionary are strings, so whitespace is allowed. Example: *ViewData["Some Key With Whitespace"]*
  - Any type other than a string must be cast in the view to use ViewData.

The differences between ViewData and ViewBag

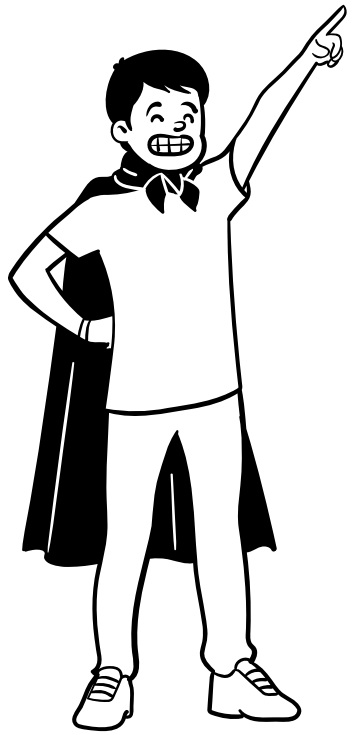
- ViewBag
  - Derives from Microsoft.AspNetCore.Mvc.ViewFeatures.Internal.DynamicViewData, so it allows the creation of dynamic properties using dot notation (*@ViewBag.SomeKey = <value or object>*), and no casting is required.
  - The syntax of ViewBag makes it quicker to add to controllers and views.
  - Simpler to check for null values. Example: *@ViewBag.Person?.Name*

When to use ViewData or ViewBag

- Both ViewData and ViewBag are equally valid approaches for passing small amounts of data among controllers and views.
- The choice of which one to use is based on preference.
- You can mix and match ViewData and ViewBag objects, however, the code is easier to read and maintain with one approach used consistently.
- In general, the ViewBag is easier to work with than the ViewData dictionary. As a result, we recommend using it in most scenarios. However, there are a few scenarios where it makes sense to use ViewData. For example, you can use ViewData if you need to use names for your keys that aren’t valid in C# or if you need to be able to access the properties of the ViewDataDictionary class.

When to use ViewData or ViewBag

- One disadvantage of both the ViewBag and ViewData properties is that Visual Studio doesn’t provide compile-time checking or IntelliSense for these properties. This is true for both the controller and the view. Because of this, some programmers avoid using either of these properties and use view models Instead.
- Both approaches are dynamically resolved at runtime and thus prone to causing runtime errors. Since they don't offer compile-time type checking, both are generally more error-prone than using a viewmodel. For that reason, Some development teams avoid them.



Any Questions?


THANK
YOU!