Software Development

Presented By:
Khaled Gamal

ASP.NET Core

MVC

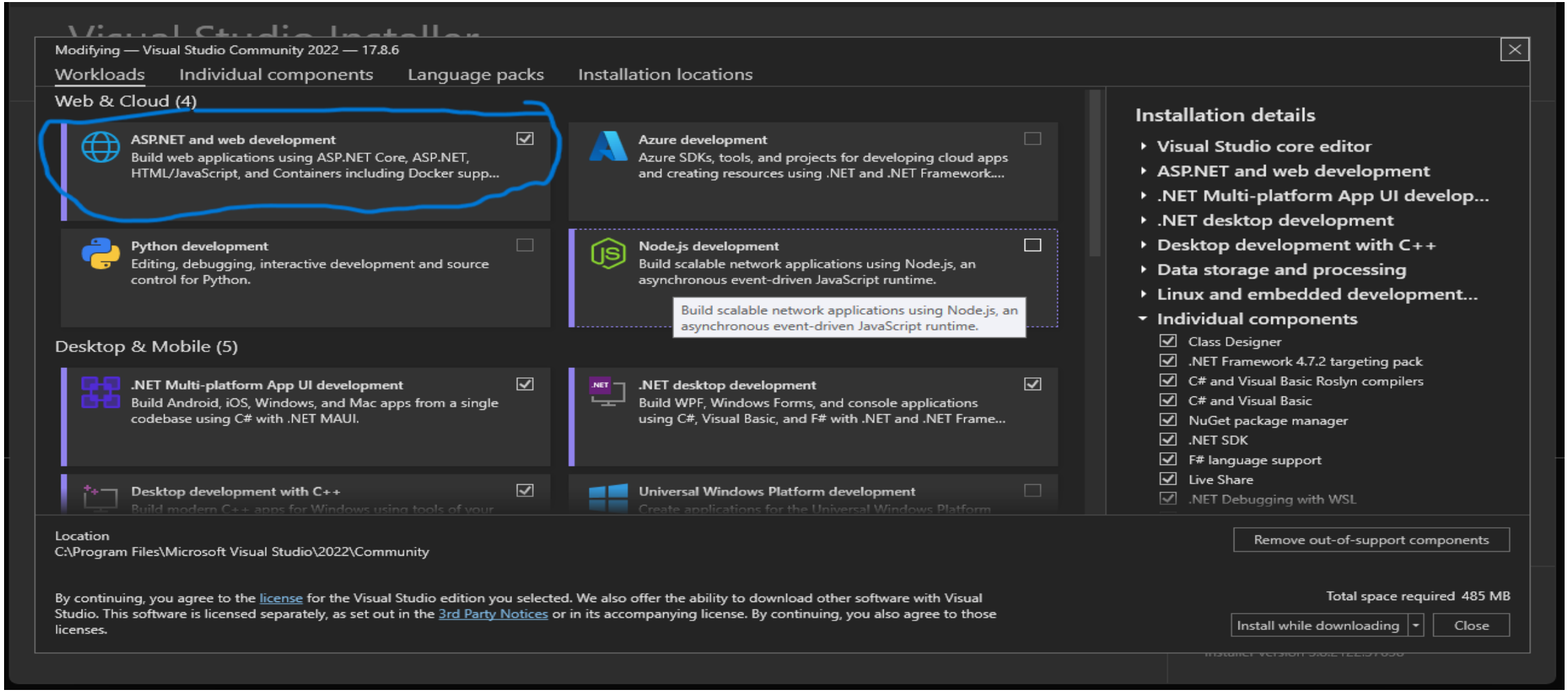
Section 2

Outlines

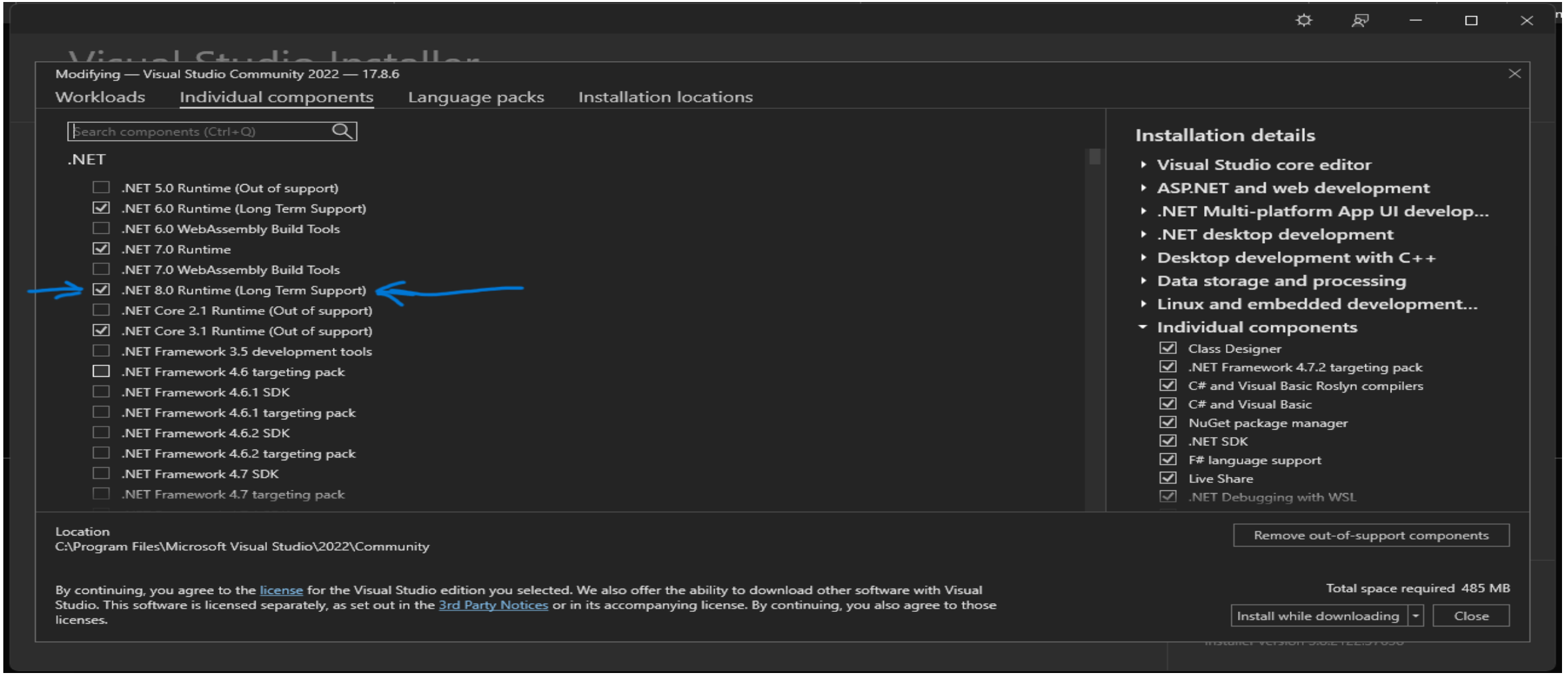
- **Installing basic Environment and running First Example**
- **The Project Layout**
- **Introduction to MVC**
- **Introduction to Controllers**

Setup Environment

Setup Environment

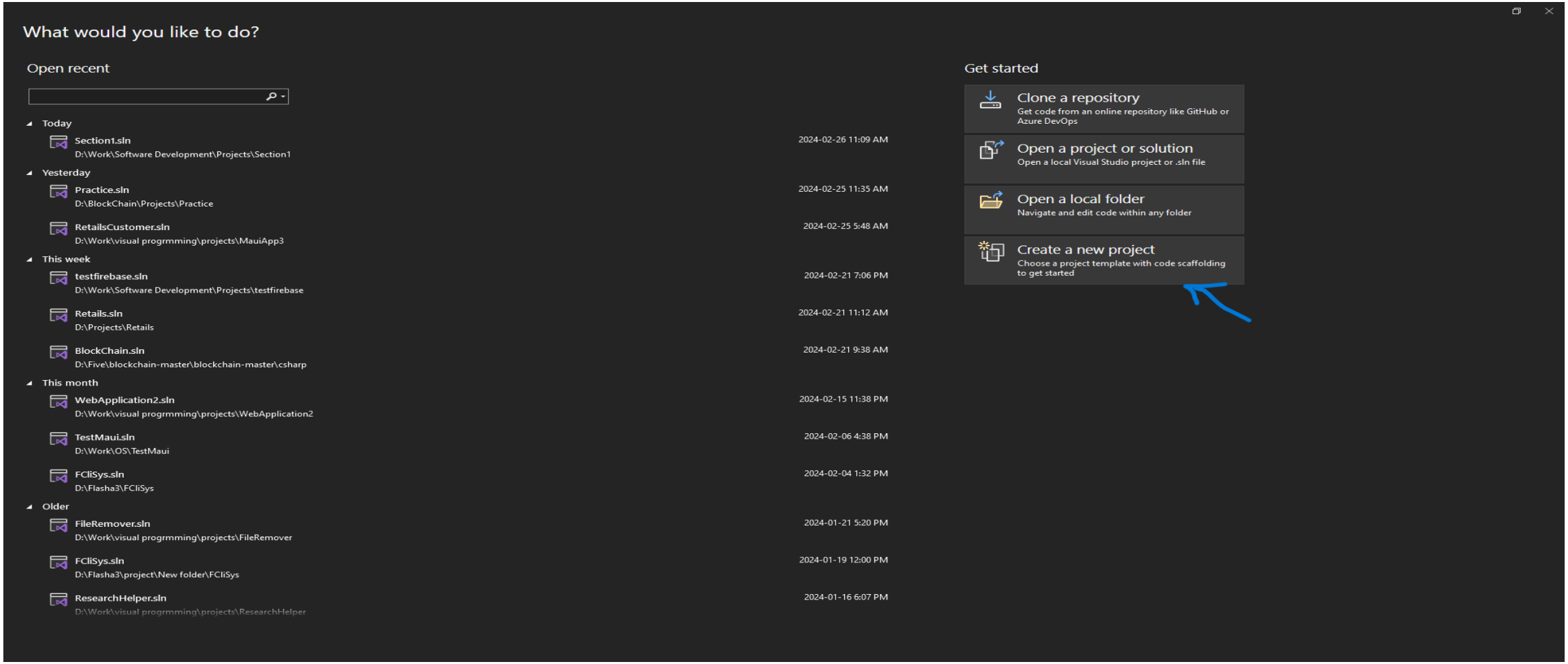


Setup Environment

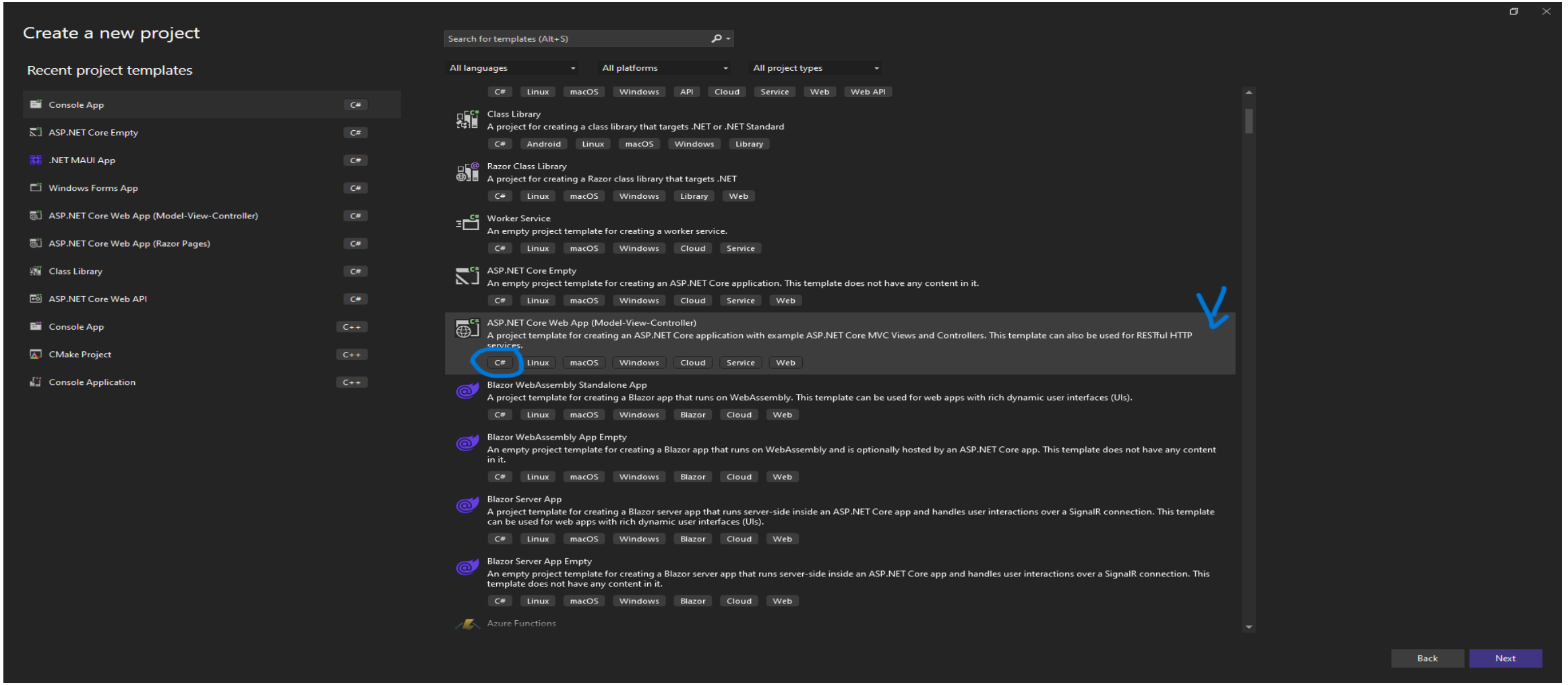


Run Your First Example

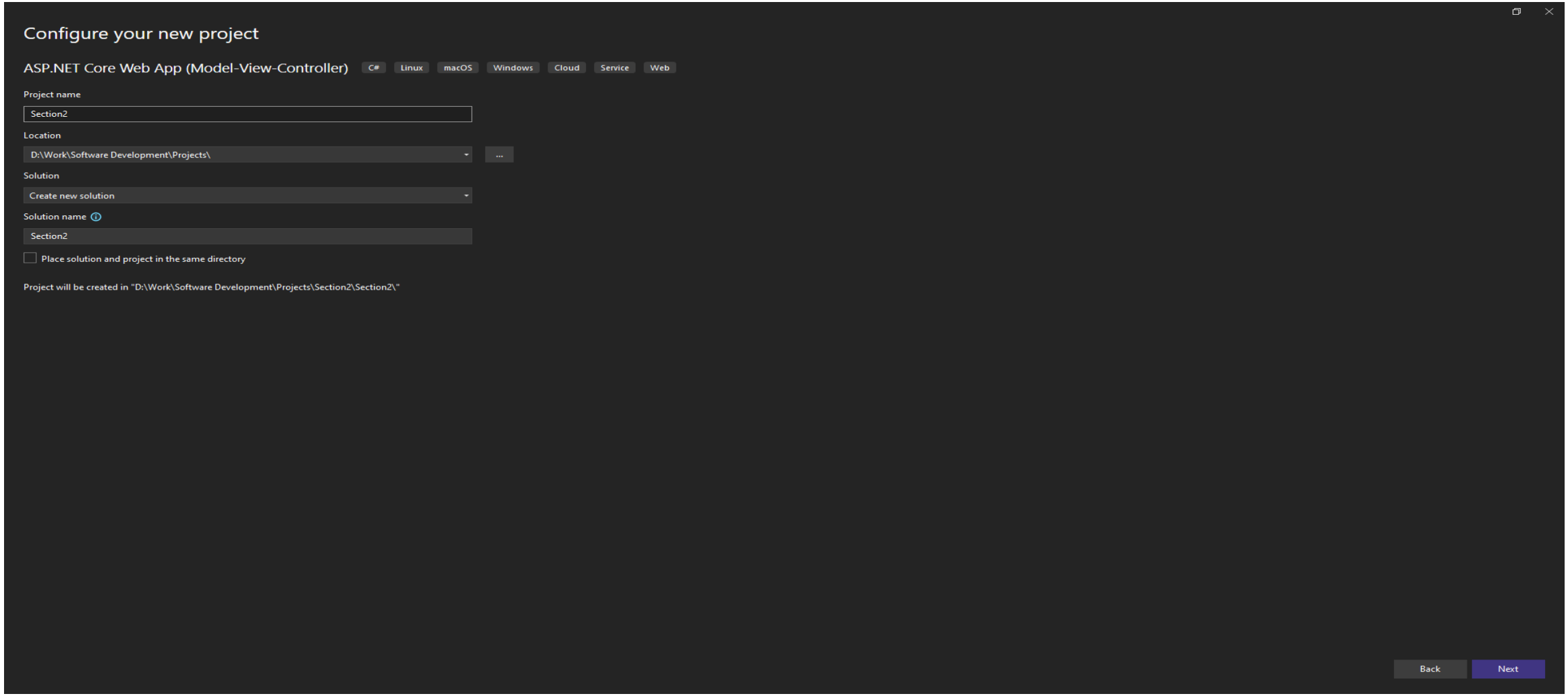
Run Your First Example



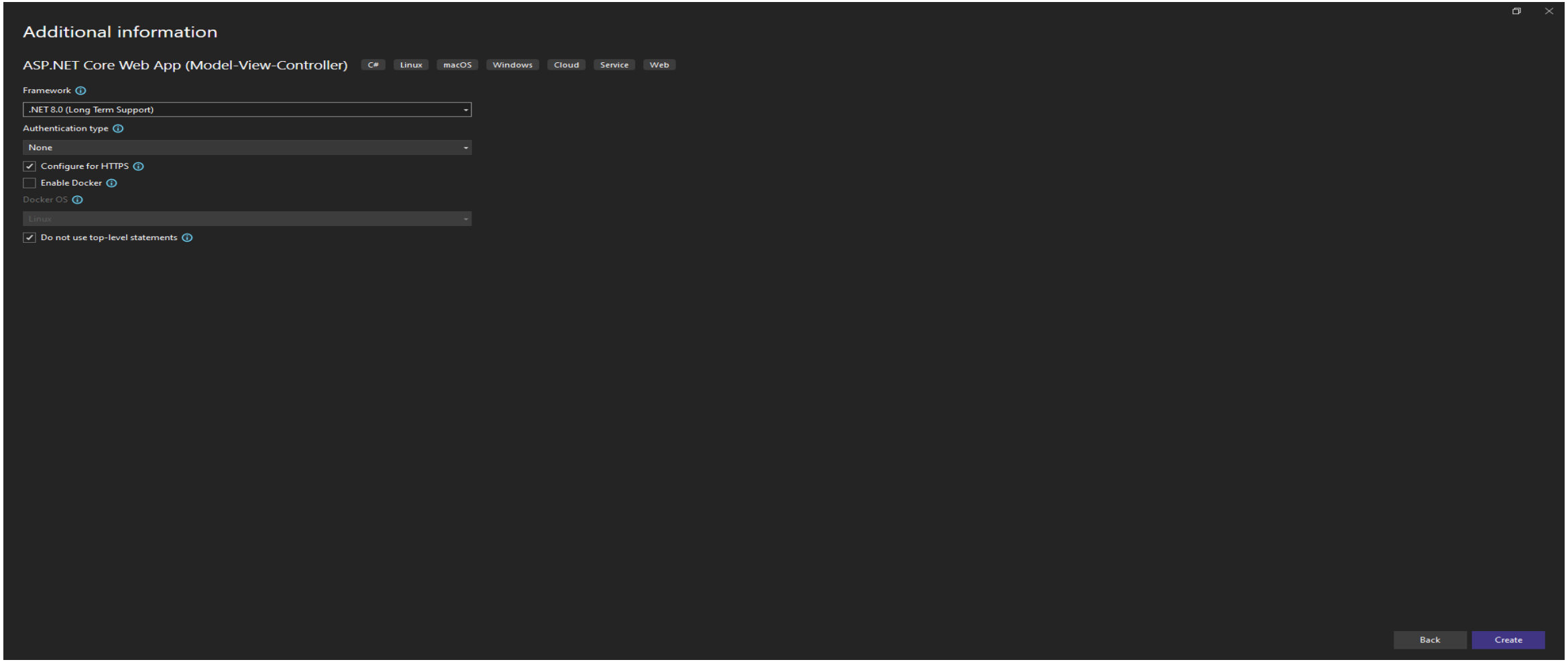
Run Your First Example



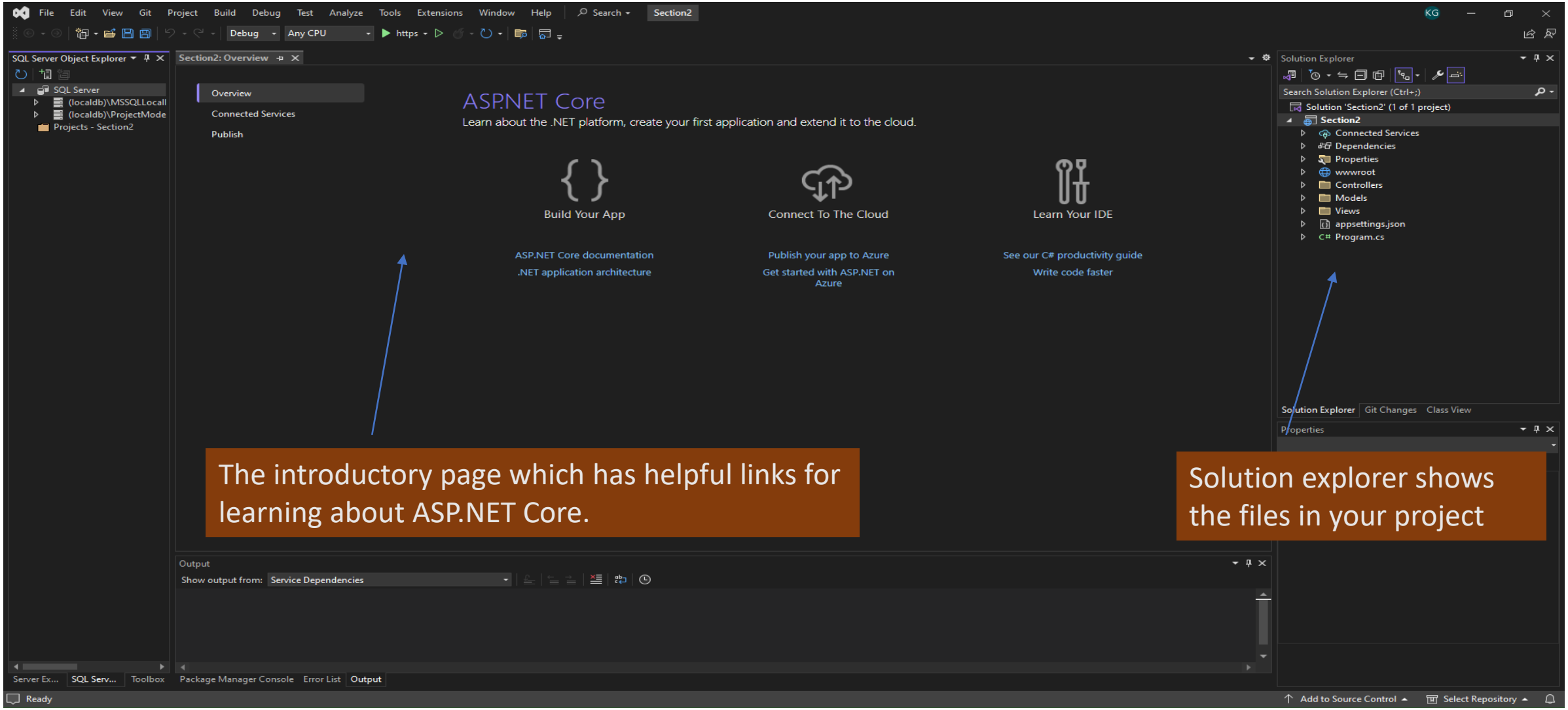
Run Your First Example



Run Your First Example

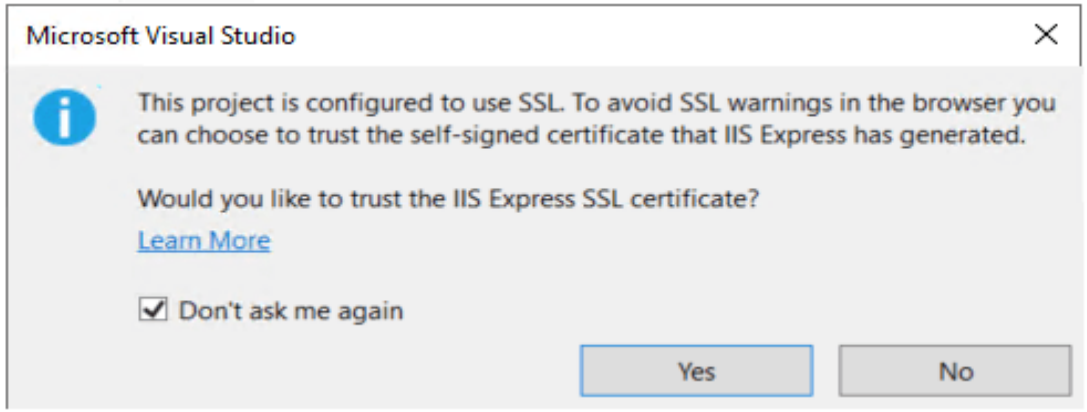


Run Your First Example



Run Your First Example

- Select *Ctrl+F5* to run the app without the debugger.
- Visual Studio displays the following dialog when a project is not yet configured to use SSL:

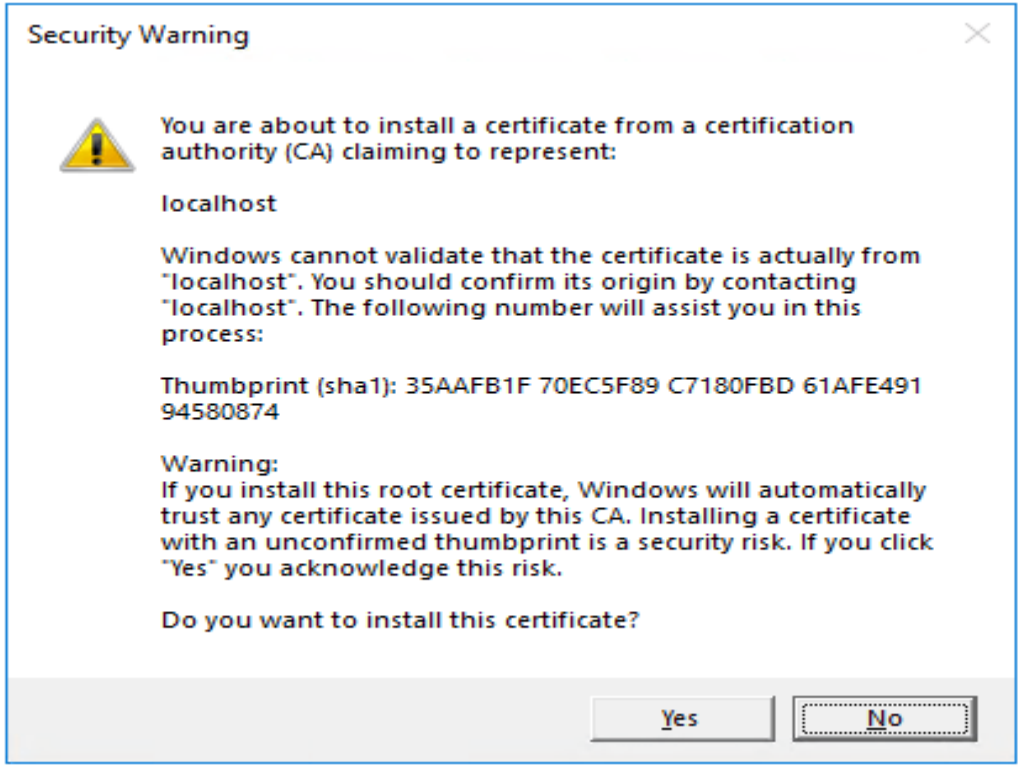


- Select *Yes* if you trust the *IIS Express SSL* certificate.

Run Your First Example

- The following dialog is displayed:

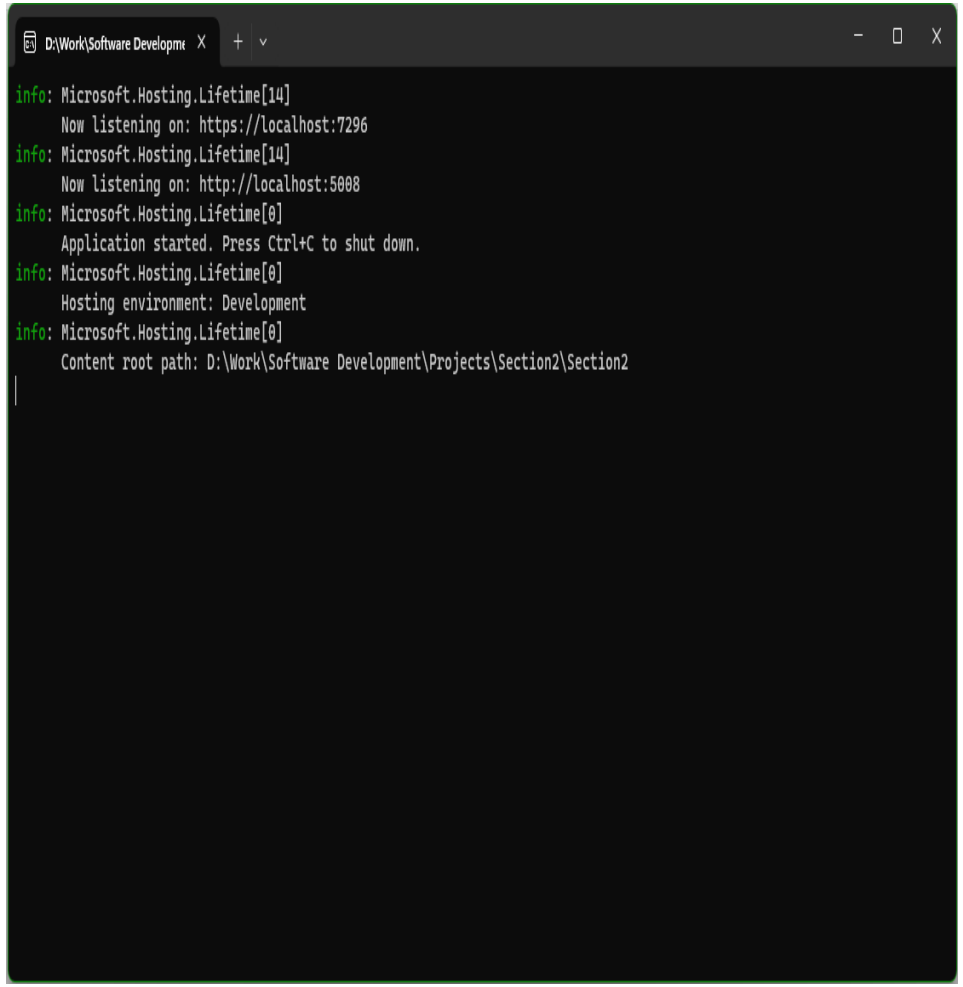
- Select *Yes* if you agree to trust the development certificate.

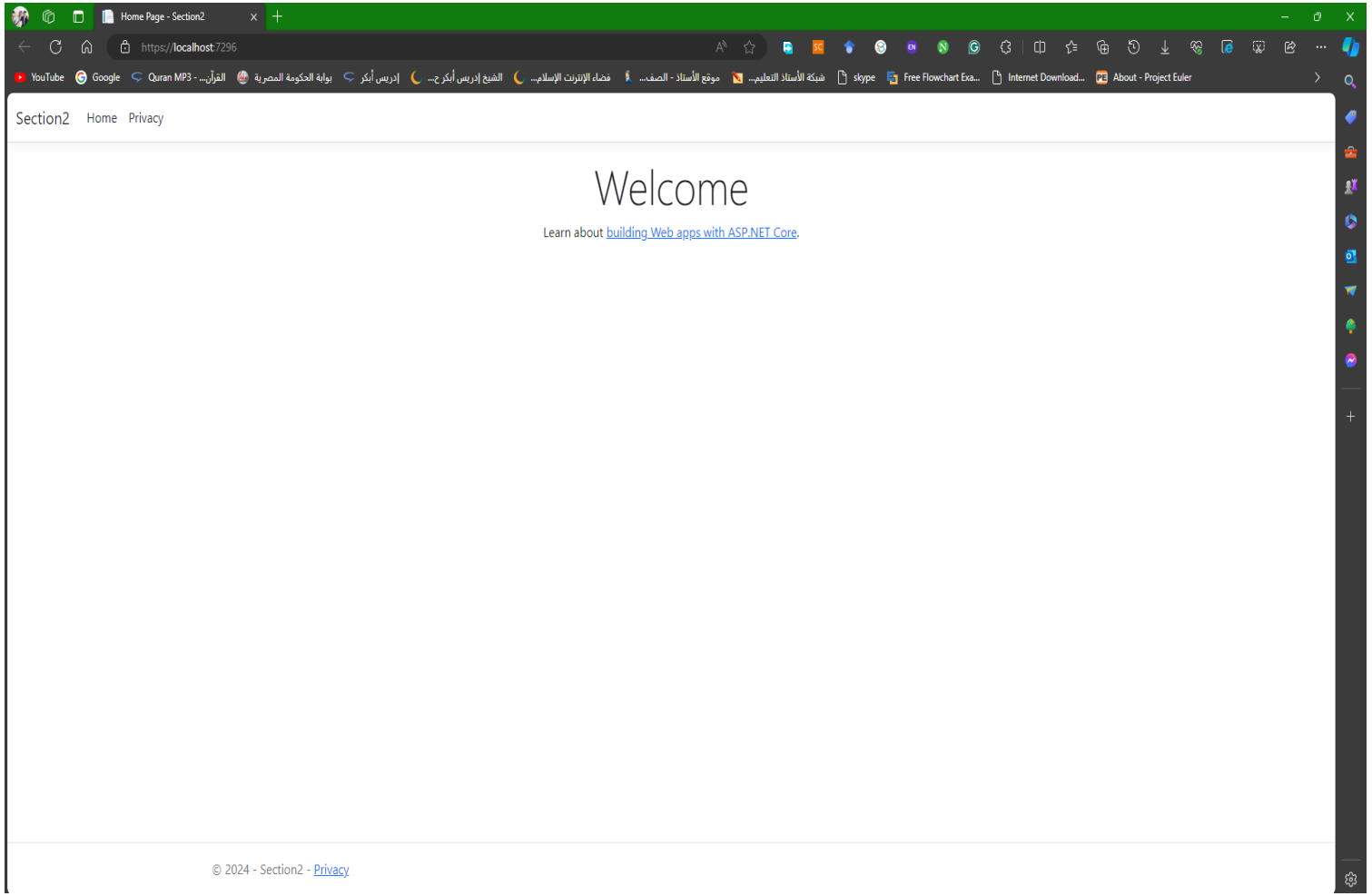


Run Your First Example

- Visual Studio starts *Kestrel* and uses a *console* window to display information about the status of the server and each HTTP request.
- To stop the server, you can close this window or type *ctrl+c*.
- Visual Studio runs the app and opens the default browser.
- The address bar shows *localhost:<port#>* and not something like *example.com* .
- The standard hostname for your local computer is *localhost* .
- When Visual Studio creates a web project, a *random* port is used for the web server.

Run Your First Example

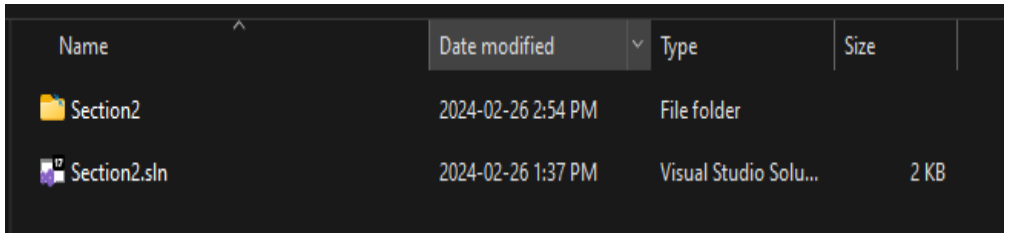


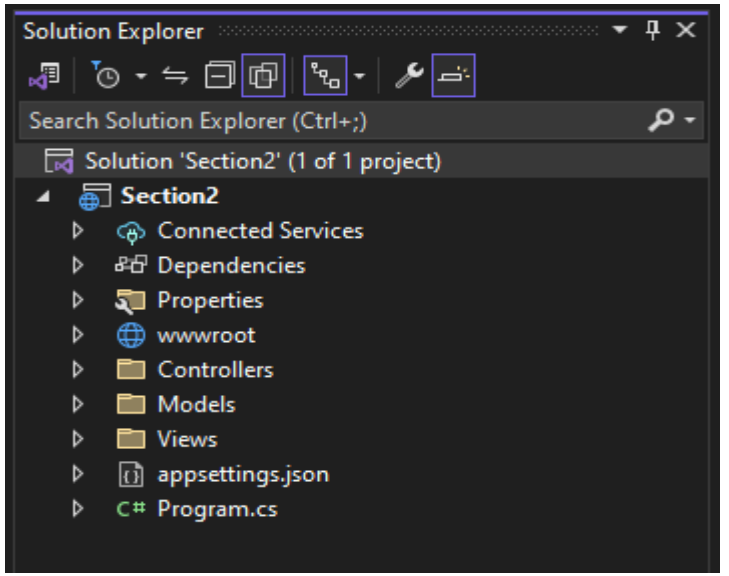


The Project Layout

The Project Layout

- The first thing to notice in the solution explorer or file explorer is that the main project, *Section2*, is nested in a top-level directory with the name of the solution, which is also *Section2* in this case.
- Within this top-level folder you'll also find the *solution (.sln) file* used by Visual Studio, though this is hidden in Visual Studio's Solution Explorer view.





The Project Layout

- Inside the solution folder you'll find your project folder, which contains the most important file in your project: Section2.csproj.
- It defines the type of project being built (web app, console app, or library), how to build your project, which platform the project targets (.NET Core 3.1, .NET 7 and so on), and which NuGet packages the project depends on.
- Visual Studio doesn't show the .csproj file explicitly, but you can edit it if you double-click the project name in Solution Explorer.

The Project Layout

- The *Sdk* attribute on the Project element includes default settings that describe how to build your project, whereas the *TargetFramework* element describes the framework your application will run on. For .NET 7.0 projects, this element will have the net7.0 value; if you're running on .NET 8, this will be net8.0.
- You can also enable and disable various features of the compiler, such as the C# 8 feature nullable reference types or the C# 10 feature implicit using statements.
- The most common changes you'll make to the project file are to add more *NuGet* packages by using the *PackageReference* element.

The Project Layout

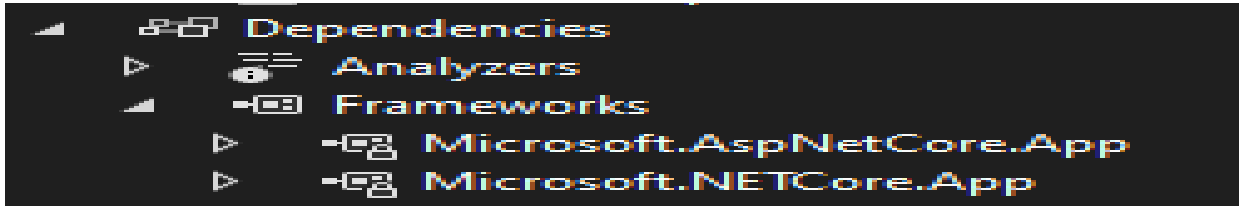
```
<Project Sdk="Microsoft.NET.Sdk.Web">
<PropertyGroup>
<TargetFramework>net8.0</TargetFramework>
<Nullable>enable</Nullable>
<ImplicitUsings>enable</ImplicitUsings>
</PropertyGroup>
<ItemGroup>
<PackageReference Include="NewtonSoft.Json" Version="13.0.1" />
</ItemGroup>
</Project>
```

The Properties

- Your project folder contains a subfolder called *Properties*, which contains a single file: *launchSettings.json*.
- This file controls how Visual Studio will run and debug the application and contains details like IIS settings, application URLs, authentication, SSL port details, etc.
- This file is only used on the local development machine and is not deployed to your production website.
- Visual Studio shows the folder as a special node in Solution Explorer, out of alphabetical order, near the top of your project.

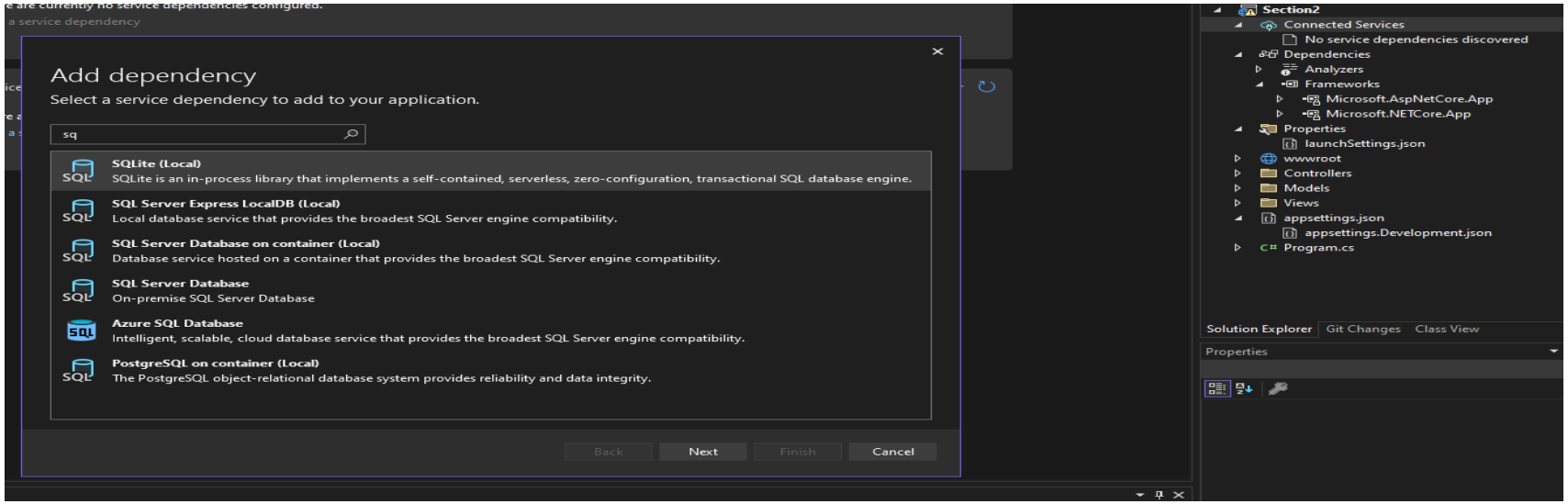
The Dependencies

- You've got two more special nodes in the project, *Dependencies* and *Connected Services*, but they don't have corresponding folders on disk.
- The *Dependencies* node contains all the references of the packages(*NuGet packages*) used in the project.
- Here the *Frameworks* node contains reference two most important *dotnet core runtime* and *asp.net core runtime* libraries.



The Connected Services

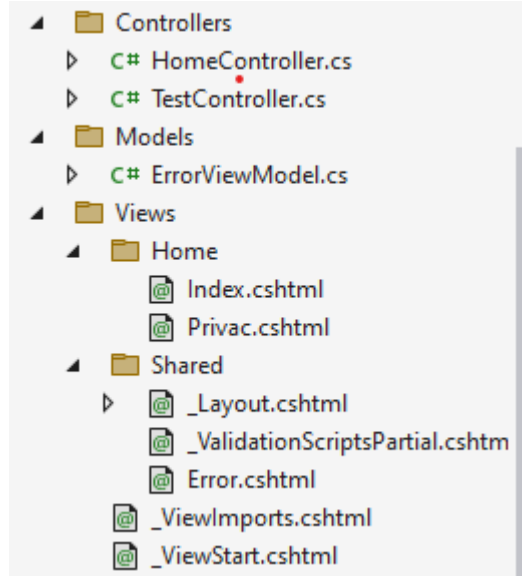
- It contains the details about all the *remote services* references added to the project.
- A new service can be added here, for example, if you want to add access to Cloud Storage of Azure Storage you can add the service here or connect to database provider.



The wwwroot

- Files that do not change at runtime are called *static* files.
- These files are not dynamically generated or modified when the user interacts with our web application, so we call them static.
- The following are some examples of static files: CSS files, (some) HTML files, JavaScript files, and some images and videos (company logo, company intro).
- We typically store static files in the project's *web root directory* (this is just a folder named *wwwroot* created directly in the root of the project).
- Visual Studio is using a special icon for this folder, which should suggest this is an important folder as it is the only folder in your application that browsers are allowed to access directly when browsing your web app, the static file middleware will serve them to browsers when requested.

The Controllers,views, and Models



- *Controllers*: This folder contains C# classes that have methods (known as actions) that fetch a model and pass it to a view, for example, HomeController.cs.
- *Models*: This folder contains C# classes that represent all of the data gathered together by a controller and passed to a view, for example, ErrorViewModel.cs.
- *Views*: This folder contains the .cshtml Razor files that combine HTML and C# code to dynamically generate HTML responses. The _ViewStart file sets the default layout and _ViewImports imports common namespaces used in all views like Tag Helpers:
  - Home: This subfolder contains Razor files for the home and privacy pages.
  - Shared: This subfolder contains Razor files for the shared layout, an error page, and two partial views for logging in and validation scripts.

The appsettings.json file

- In the root of your project folder, you'll find two JSON files: *appsettings.json* and *appsettings.Development.json*.
- These files provide configuration settings that are used at runtime to control the behavior of your app such as database connection strings.

The Project Layout

- Finally, Visual Studio shows one C# file in the project folder: *Program.cs*. which configures and runs your application.
- The *WebApplication* is the core of your ASP.NET Core application, containing the application *configuration* and the *Kestrel* server that listens for requests and sends responses.
- The framework uses a builder design pattern which allows users to configure an object, delaying its creation until all configuration has finished, and uses the Dependency Injection design patterns to register those configurations and services.

The Project Layout

- Within the context of ASP.Net Core, *service* refers to any class that provides functionality to an application.
- *Services* could be classes exposed by a library or code you've written for your application.
- A *WebApplicationBuilder*, created by the call to *CreateBuilder*, to define how the *WebApplication* is configured, before instantiating the *WebApplication* with a call to *Build*().

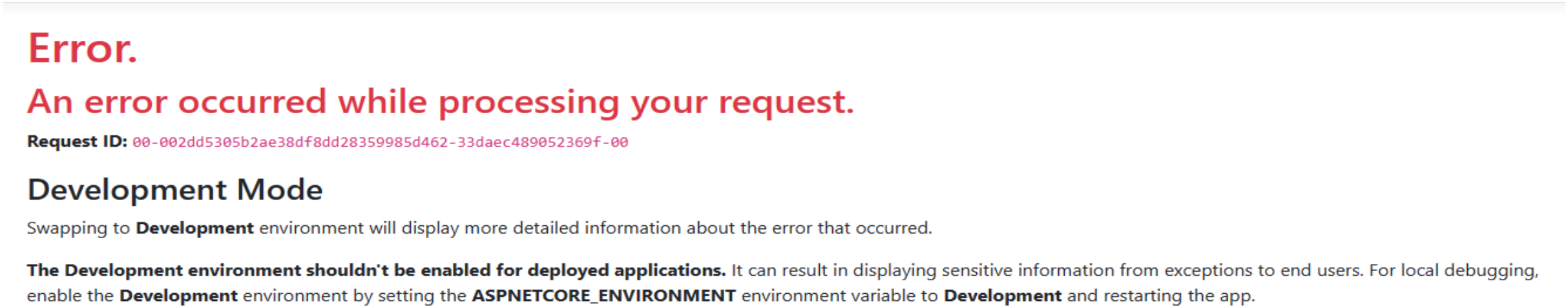
The Project Layout

- The *WebApplication* instance is where you define how your application handles and responds to requests, using two building blocks:
  - *Middleware*—These small components execute in sequence when the application receives an HTTP request. They can perform a whole host of functions, such as logging, identifying the current user for a request, serving static files, and handling errors.
  - *Endpoints*—Endpoints define how the response should be generated for a specific request to a URL in your app.
- Only after the call to Run() does the HTTP server start listening for requests from a remote browser and can respond to them.

Middlewares

```
UseExceptionHandler("/Home/Error");
```

- A middleware for handling exceptions and send the exception information to a given action here the Error action in the HomeController which will display these information to end-user and will not send the request forward to other middlewares or endpoint.



Middlewares

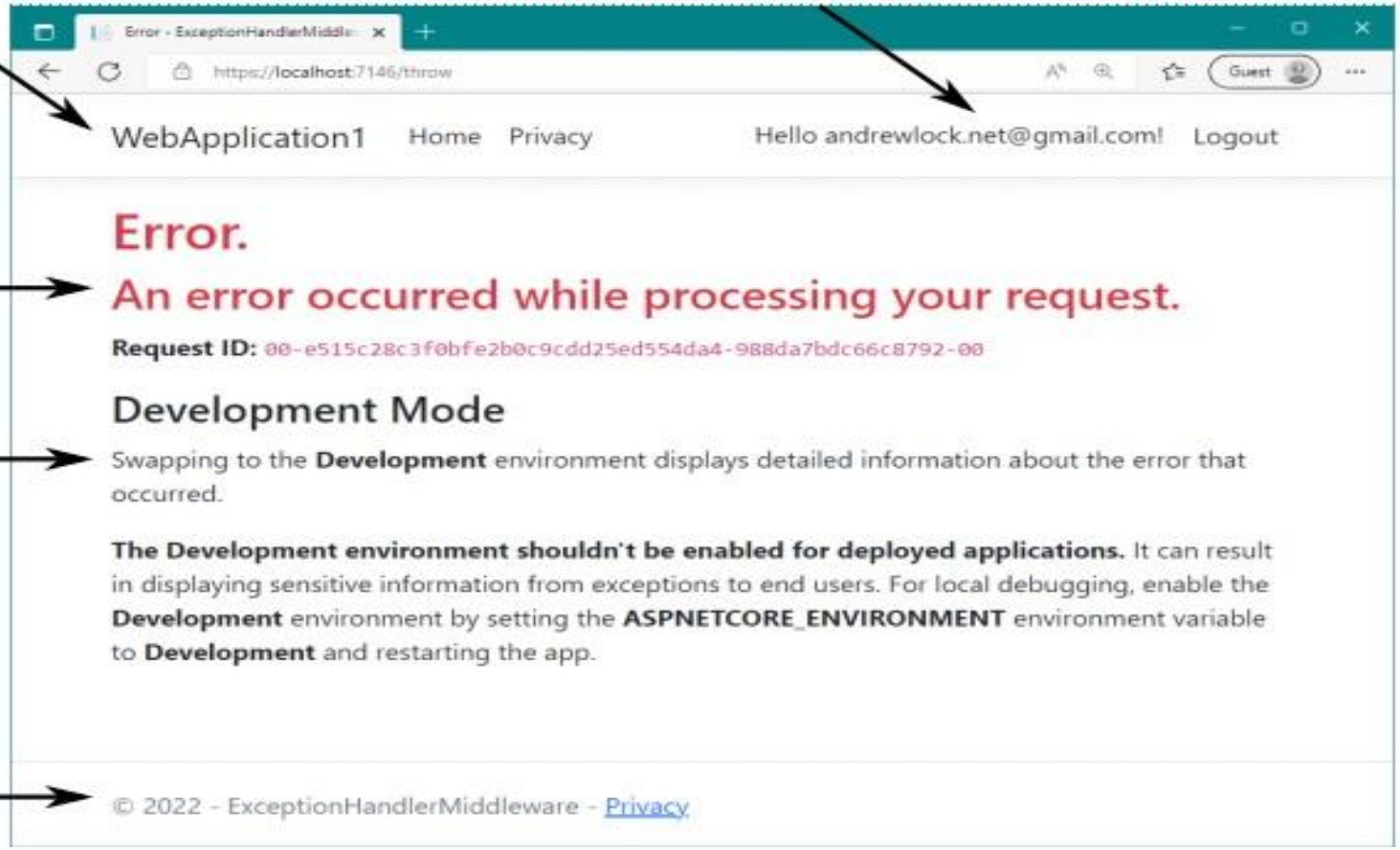
Menu bar consistent with the rest of your application.

Dynamic details such as the current user can be shown on the error page.

Error page contains appropriate details for the user.

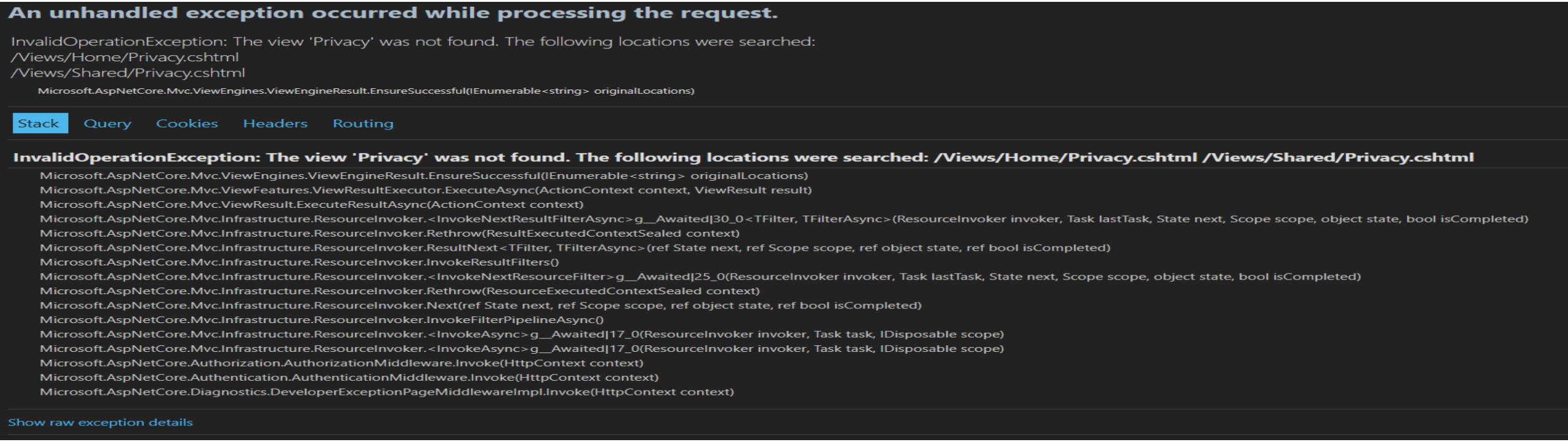
The default error page reminds you about the developer exception page. You would change this text in your application to something more generic.

Footer consistent with the rest of your application.



Middlewares

- By default the middleware for handling exceptions in your development environment is *UseDeveloperExceptionPage();* which handles the exceptions and displays information for developer in details.



Middlewares

Title indicating the problem.

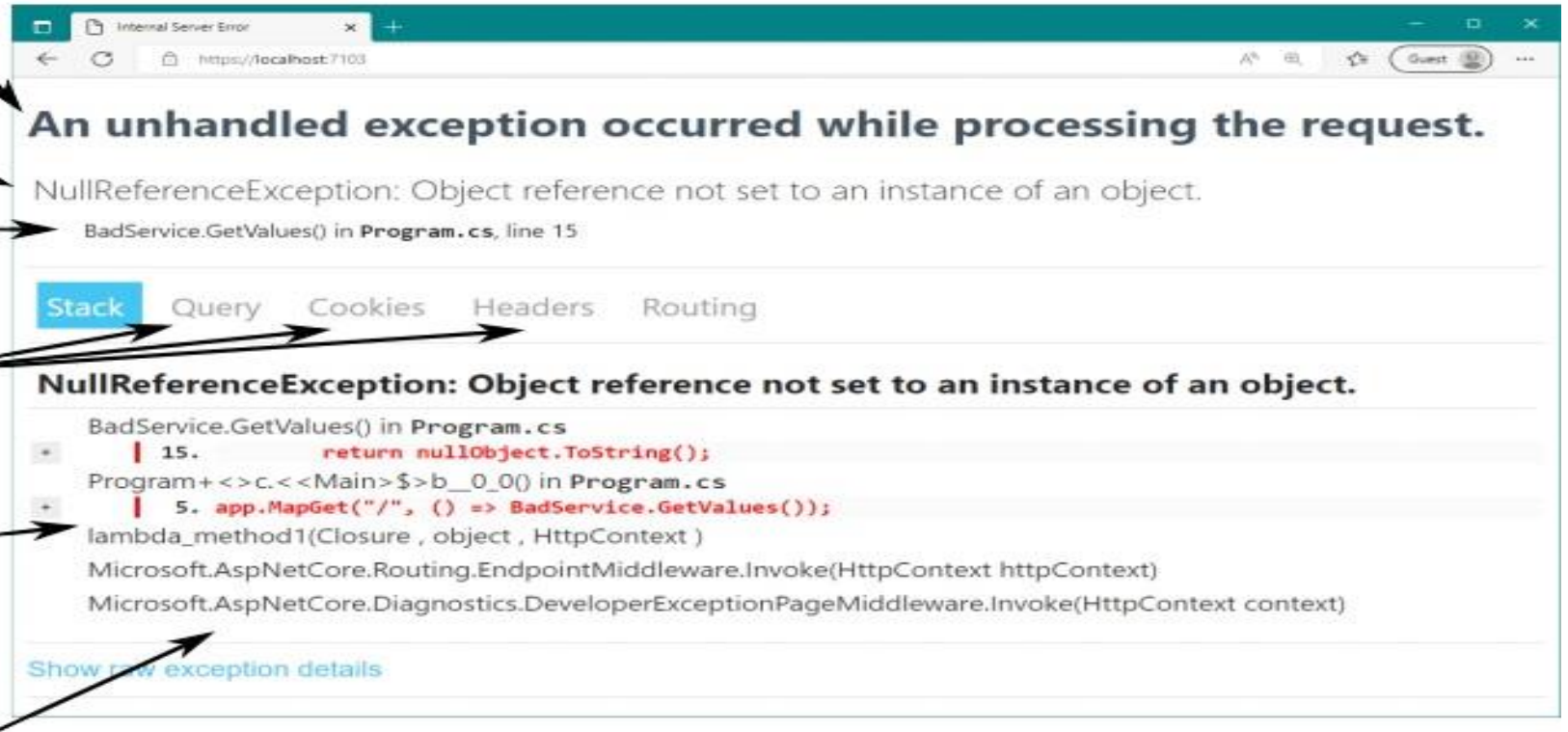
Detail of the exception that occured.

Location in the code where exception occured.

Buttons to click that reveal further details about the request that caused the exception.

Code that caused the exception. You can click the "+" symbol to expand the code around the exception.

Full stack trace for the exception.



Middlewares

UseHsts();

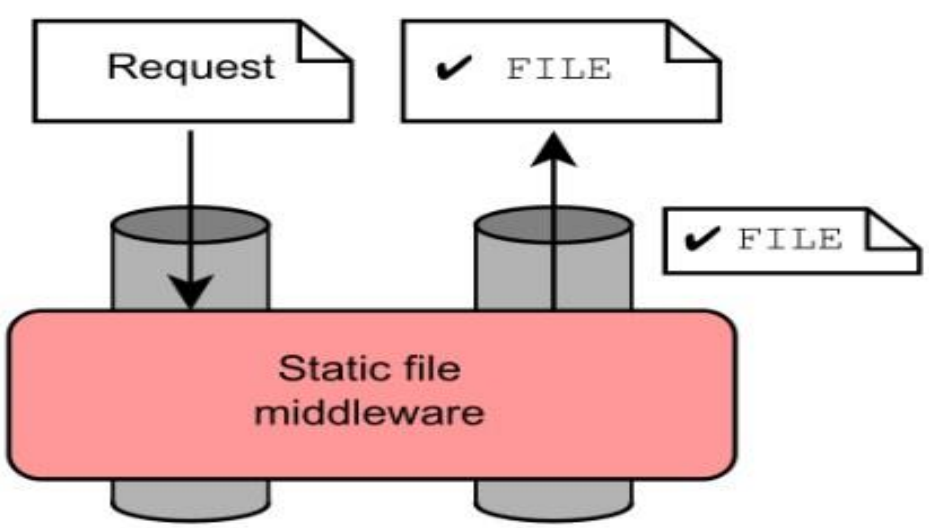
- HTTP Strict TransportSecurity(HSTS) Security enhancement middleware that adds a special response header.
- If a website specifies it and a browser supports it, then it forces all communication over HTTPS and prevents the visitor from using untrusted or invalid certificates.
- You should the following middleware after that which redirects all HTTP requests to HTTPS:

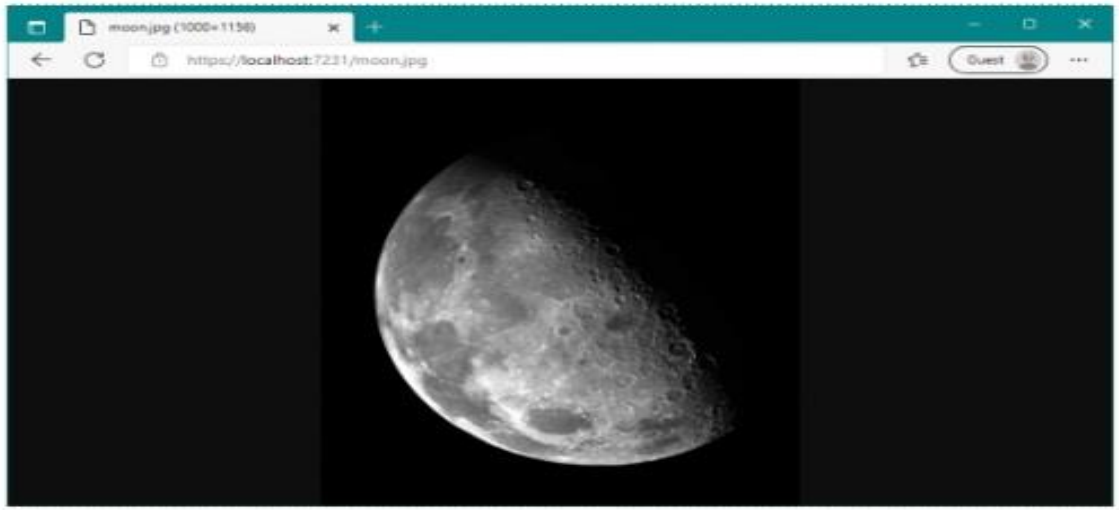
UseHttpsRedirection();

Middlewares

UseStaticFiles();

- Middleware that serves the static files in your application.





1. The static file middleware handles the request by returning the file requested.

2. The file stream is sent back through the middleware pipeline, and out to the browser.

3. The browser displays the file returned in the response.

Middlewares

UseRouting();

- Adds route matching to the middleware pipeline. This middleware looks at the set of endpoints defined in the app, and selects the best match based on the request.
- Routing is one middleware component that will send requests (route them) to actions in controllers.
- Think of routing as the middleware component that will send incoming HTTP requests (only those that follow a specific format) to the MVC part of our web application.
- Routing gives us, the developers, full control over the format of the URLs in our web application.
- It lets us describe what URL paths will be matched to what actions.

Middlewares

UseAuthorization();

- Middleware for authorize user's access to a resource it uses the credentials from the Authentication middleware, so, if our websites allow user to login with their accounts you should add the middleware of authentication using *UseAuthentication();* and should be added before the authorization middleware.
- It's important to consider the order of middleware when adding it to the pipeline, as middleware can use only objects created earlier in the pipeline.

EndPoints

```
app.MapControllerRoute(
name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

- *Endpoint*: define how the response should be generated for a specific request to a URL in your app.
- It is used to create a single route. The single route is named default route.
- Most apps with controllers and views use a route template similar to the default route.
  - The route template "{controller=Home}/{action=Index}/{id?}":
  - A name that must be present (and distinct from other route names), but it is not used in the routing itself.
  - A pattern that describes what requests it needs to match.
- In our program so far, if a request does not match the route, we'll get the HTTP 404 error.
- -Matches a URL path like /Products/Details/5
- -Extracts the route values { controller = Products, action = Details, id = 5 } by tokenizing the path.
- The extraction of route values results in a match if the app has a controller named ProductsController and a Details action.

EndPoints

```
app.MapControllerRoute(
name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

- The route pattern has parts in curly brackets {} called segments, and they are like named parameters of a method.
- The value of these segments can be any string. Segments in URLs are not case-sensitive.
- The route pattern looks at any URL path requested by the browser and matches it to extract the name of a controller, the name of an action, and an optional id value (the ? symbol makes it optional).
- If the user hasn’t entered these names, it uses defaults of Home for the controller and Index for the action (the = assignment sets a default for a named segment).

An introduction to the MVC pattern

An introduction to the MVC pattern

- **What is the MVC pattern?**
- The *MVC (Model-View-Controller)* is a software architectural design pattern commonly used to structure web apps that have significant processing requirements.
- The MVC pattern divides an app into separate components where each component has a specific area of concern.
- This is known as *separation of concerns,* and it leads to many benefits including making the app easier to code, test, debug, and maintain.

An introduction to the MVC pattern

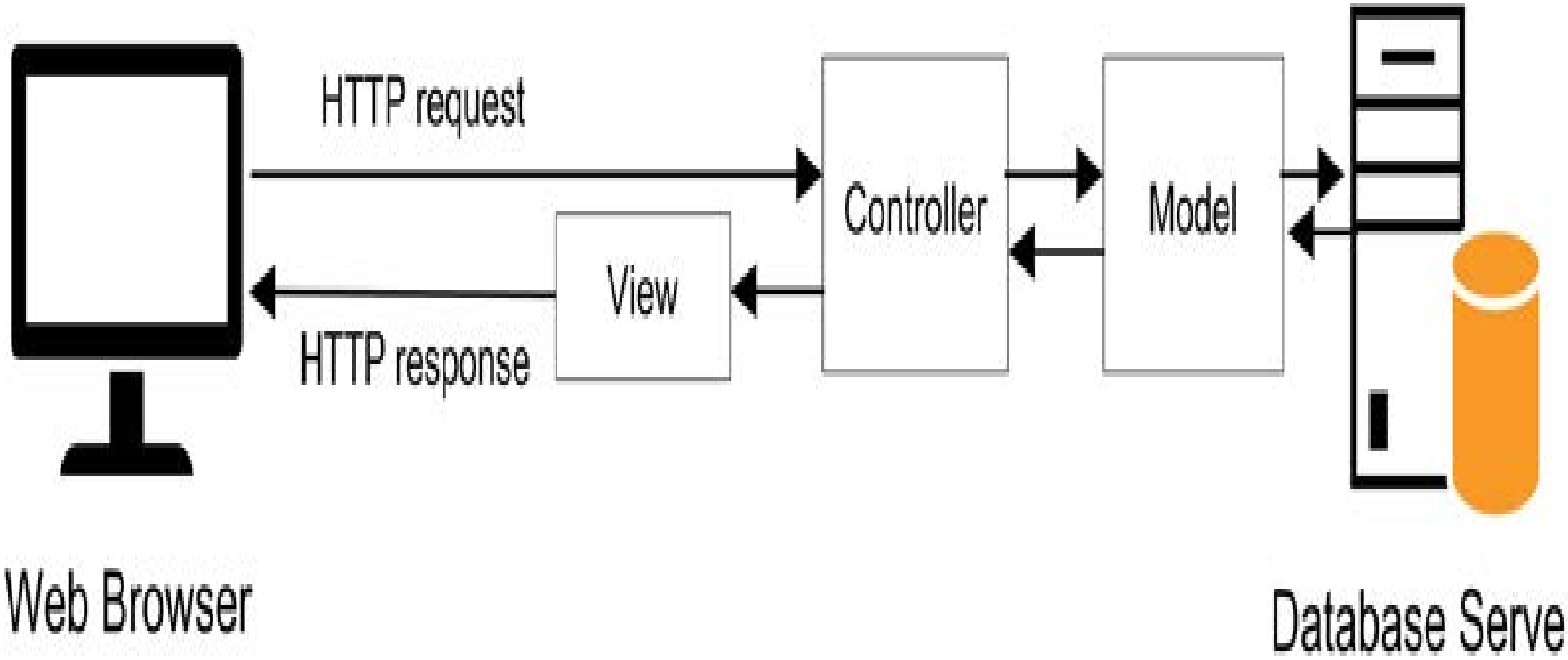
- **What is the MVC pattern?**
- Using this pattern, user requests are routed to a Controller which is responsible for working with the Model to perform user actions and/or retrieve results of queries.
- The Controller chooses the View to display to the user and provides it with any Model data it requires.

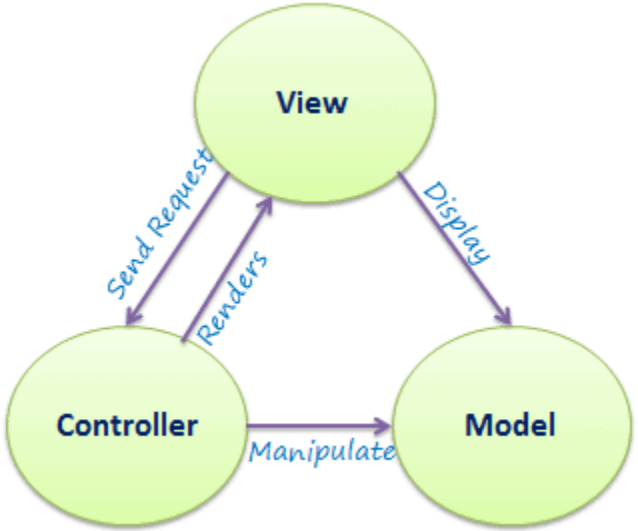
An introduction to the MVC pattern

- **What is the MVC pattern?**
- Both the view and the controller depend on the model.
- However, the model depends on neither the view nor the controller.
- This is one of the key benefits of the separation. This separation allows the model to be built and tested independent of the visual presentation.

An introduction to the MVC pattern

- **What is the MVC pattern?**





An introduction to the MVC pattern

- **What are the components of MVC Pattern?**
- The *model* is in charge of data.
  - Specifically, it gets and updates the data in a data store such as a database, it applies business rules to that data, and it validates that data.
  - The *model* consists of the code that provides the data access and business logic.

An introduction to the MVC pattern

- **What are the components of MVC Pattern?**
- The *view* is in charge of the user interface.
  - Specifically, it creates the HTML, CSS, and JavaScript that the app sends to the browser in response to the browser's HTTP request.
  - The *view* consists of the code that generates the user interface and presents it to the user, some special syntax (Razor syntax) that makes it easy to communicate with the model and the controller.
  - Razor is a compact, expressive and fluid template markup language for defining views using embedded C# code.
  - Razor is used to dynamically generate web content on the server.

An introduction to the MVC pattern

- **What are the components of MVC Pattern?**
- The *controller* is in charge of controlling the flow of data between the model and the view.
  - The *controller* consists of the code that receives requests from users, gets the appropriate data, stores it in the model, and passes the model to the appropriate view.
  - Controllers are the components that handle user interaction, work with the model, and ultimately select a view to render.
  - In the MVC pattern, the controller is the initial entry point and is responsible for selecting which model types to work with and which view to render (hence its name - it controls how the app responds to a given request).

An introduction to the MVC pattern

- It’s important to note that each component should stick to its own area of concern and be as independent as possible.
- For example, the model should retrieve and validate data, but it shouldn’t have anything to do with formatting or displaying it.
- Similarly, the controller should move data between the model and the view, but it shouldn’t apply any business rules to it.
- **What are the Drawbacks of MVC Pattern?**
  - Requires more work to set up.

An introduction to the MVC pattern

- **What are the Benefits of MVC Pattern?**
- Makes it easier to have different members of a team work on different components.
- Makes it possible to automate testing of individual components.
- Makes it possible to swap out one component for another component.
- Makes the app easier to maintain.

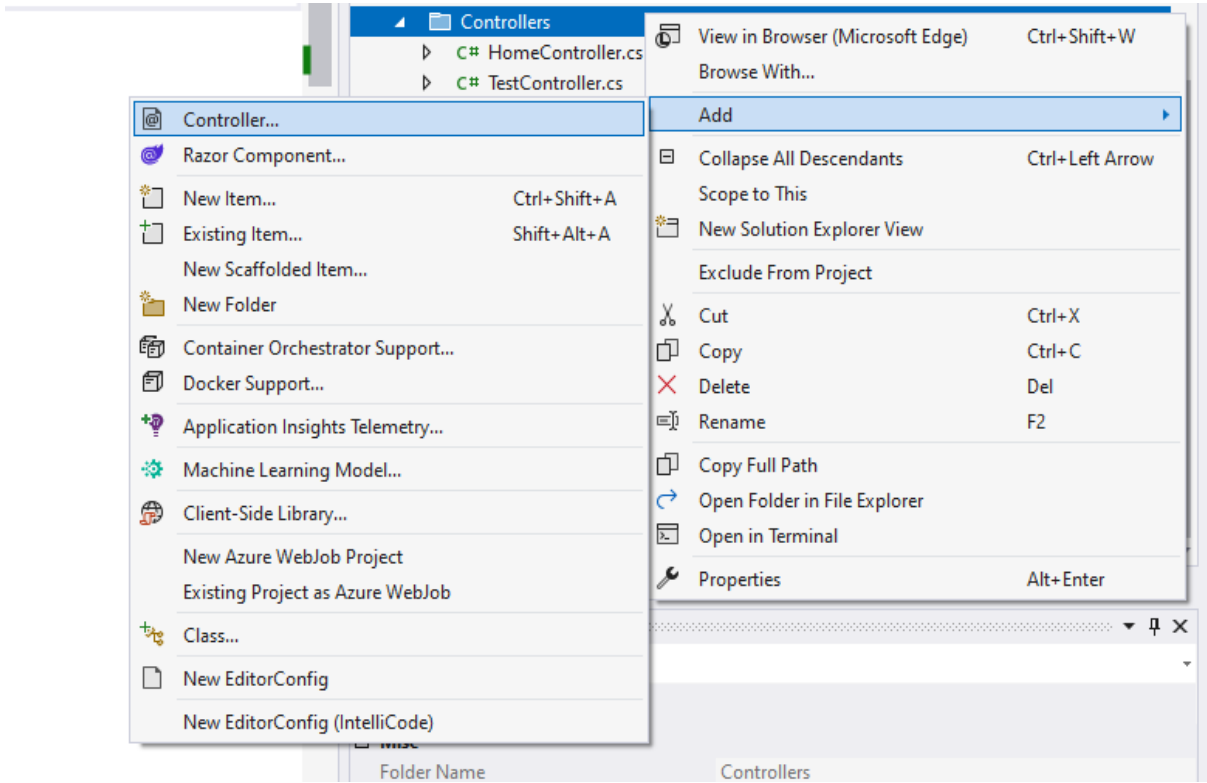
An introduction to Controller

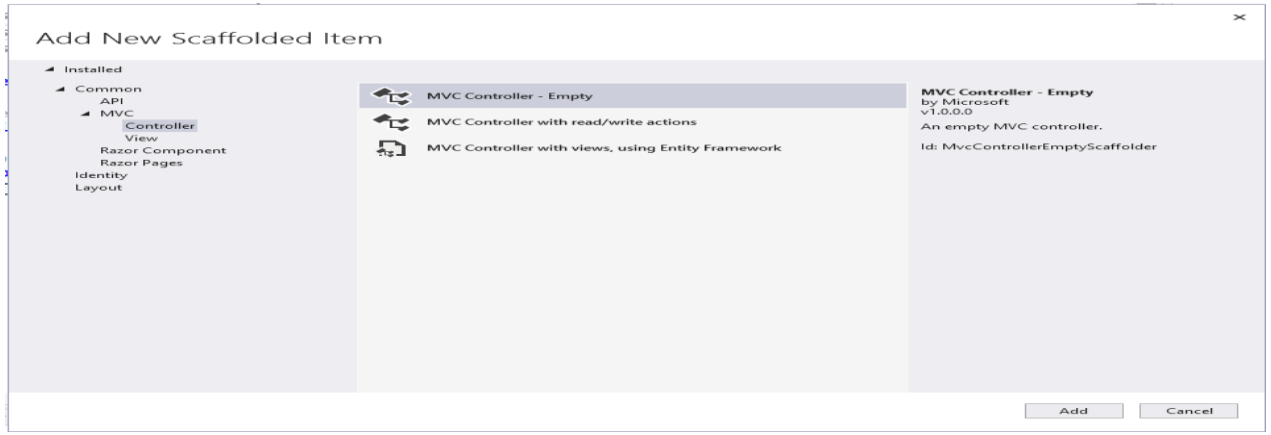
```
using Microsoft.AspNetCore.Mvc;

namespace Section2.Controllers
{
    public class TestController : Controller
    {
        /*
```

- A *controller* is a class derived from the *Microsoft.AspNetCore.Mvc.Controller* class and is used to define a set of related methods called actions.
- An *action* is any public method defined inside a controller class (as long as it does not have the[NoAction] attribute).
- All controller classes must reside in the project's root-level Controllers folder.
- In ASP.NET MVC, every controller class name must end with a word "Controller".
  - For example, the *home* page controller name must be *`HomeController`*, and for the *student* page, it must be the *`StudentController`*.
- MVC will throw "The resource cannot be found" error when you do not append "Controller" to the controller class name.

Adding a New Controller

- In the Visual Studio, right click on the Controller folder -> select **Add** -> click on **Controller..**
- This opens Add Scaffold dialog, select *MVC controller – Empty* and click *Add*.

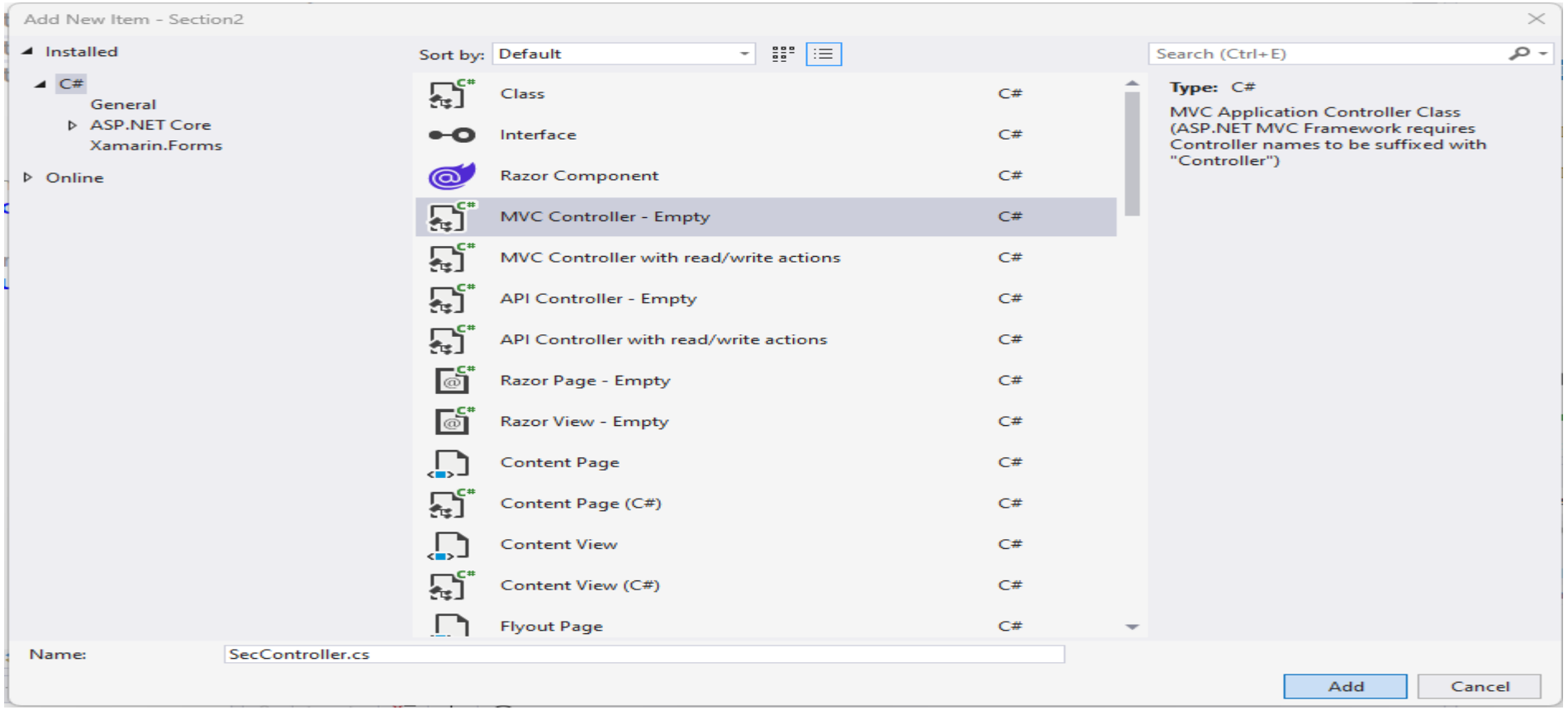




Note: Scaffolding is an automatic code generation framework for ASP.NET web applications. Scaffolding reduces the time taken to develop a controller, view, etc.

Adding a New Controller

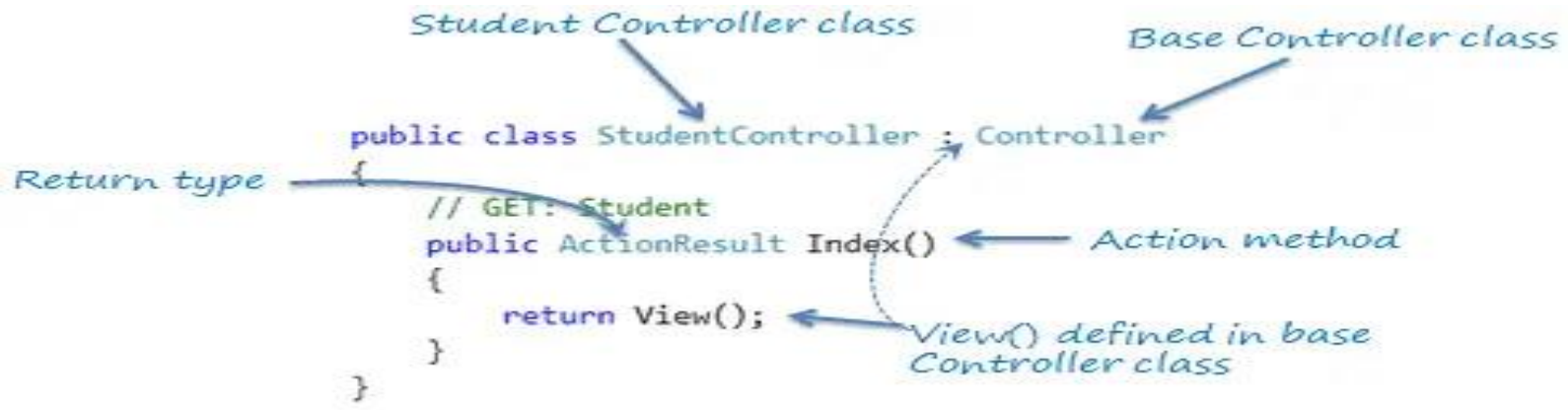
- Add New Item dialog will appear, enter the name of the controller. Remember, the controller name must end with **Controller** and then click Add.



Action Methods

- All the public methods of the Controller class are called Action methods.
- They are like any other normal methods with the following restrictions:
  - Action method must be public. It cannot be private or protected
  - Action method cannot be overloaded
  - Action method cannot be a static method.

Action Methods



As you can see in the above figure, the *Index()* method is *public*, and it returns the `ActionResult` using the `View()` method.
The **`View()`** method is defined in the `Controller` base class, which returns the appropriate `ActionResult`.

Action Methods

- **Default Action Method:**
- Every controller can have a default action method as per the configured in the routing pattern.

```
app.MapControllerRoute(
name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

- By default, the Index() method is a default action method for any controller, however, you can change the default action name as per your requirement in the routing pattern.

Action Methods

- Our actions can have various return types:
- Like primitive data types=> int, float, double, string, etc..

```
public string search() {
return "hello from search Action";
}
```

- Also, the framework includes various Result classes, which can be returned from an action method.
- The result classes represent different types of responses, such as HTML, file, string, JSON, javascript, etc.

Action Methods

- *ContentResult*: Use this to send responses containing plain text
- or XML.
  - Note: We used return *Content(…)* and this has the return type ContentResult.

```
public IActionResult testactions() {
return Content("Hello World from testactions action, TestController!");
}
```

- *ViewResult*: Use this to render a View as a response (we'll build entire HTML pages).
  - Note: We used return *View(…)* and this has the return type ViewResult.

```
public IActionResult Index() {
return View();
}
```

Action Methods

- *RedirectResult*: Use this to redirect a request to a different URL or to a different action.
  - Note: We used return *Redirect(…)* and this has the return type RedirectResult.

```
public IActionResult GoToGoole() {
return Redirect("https://www.google.com/");
}

public RedirectResult GoToHome() {
return Redirect("/home/index");
 }
```

Action Methods

- *RedirectToActionResult*: Use this to redirect a request to another action (from the same or another Controller by passing the name of the controller in the second parameter).
  - Note: We used return *RedirectToAction(…)* and this has the return type RedirectToActionResult.

```
public RedirectToActionResult GoToTestactions() {
return RedirectToAction("testactions");
 }

public RedirectToActionResult GoToprivacy() {
return RedirectToAction("Privacy","Home");
}
```

Action Methods

- *StatusCodeResult*: used when we want to send an HTTP status code as a result.

```
public StatusCodeResult result() {
return StatusCode(403);
}
```

- *EmptyResult*

```
public EmptyResult empty() {
return new EmptyResult();
}
```

- *NotFoundResult*

```
public NotFoundResult NotFoundACT() {
return new NotFoundResult();
}
```

Action Methods

- The *ActionResult* class is a base class of all the above result classes, so it can be the return type of action method that returns any result listed above.
- And it implements the *IActionResult* interface which also can be the return type of action methods.
- However, you can specify the appropriate result class as a return type of action method.
- The following table lists some of the result classes available in ASP.NET MVC.

Action Methods

Result Class	Description	Base Controller Method
ViewResult	Represents HTML and markup.	View()
EmptyResult	Represents No response.	
ContentResult	Represents string literal.	Content()
FileContentResult, FilePathResult, FileStreamResult	Represents the content of a file.	File()
JavaScriptResult	Represents a JavaScript script.	JavaScript()
JsonResult	Represents JSON that can be used in AJAX.	Json()
RedirectResult	Represents a redirection to a new URL.	Redirect()
RedirectToRouteResult	Represents redirection to another route.	RedirectToRoute()
PartialViewResult	Represents the partial view result.	PartialView()
HttpUnauthorizedResult	Represents HTTP 403 response.	

Action Methods

- **Action Method Parameters:**
- Every action methods can have input parameters as normal methods.
- Note: If you don’t pass an Id, and one is needed by the action, the model binder will use the default value (for integers that is 0).
- https://localhost:7296/Home/testparam/10

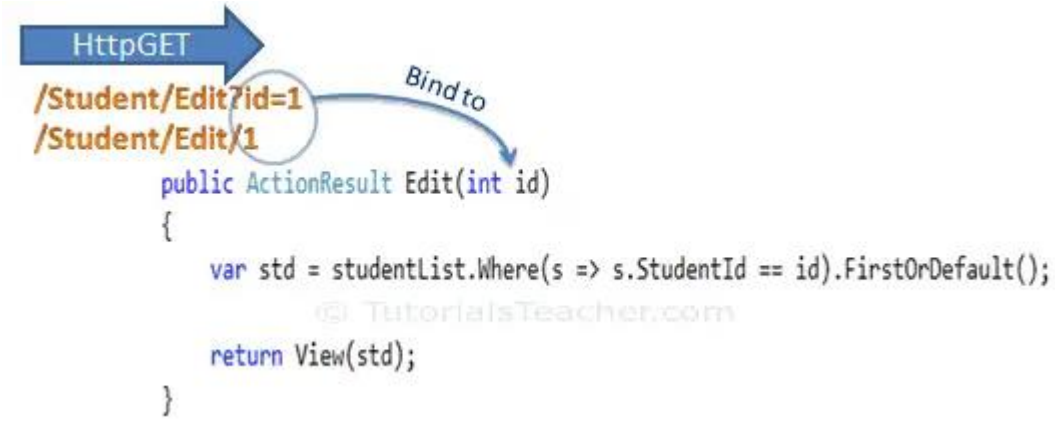
```
public string testParam(int id) {
return $"({id})^2 = {id * id}";
}
```

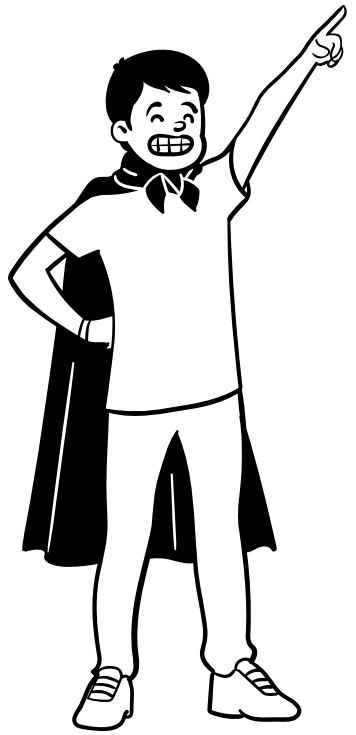
- Note: One can also use query strings to pass values. The model binder is clever enough to use them like https://localhost:7296/Home/testparam?id=10

Action Methods

- **Action Method Parameters:**
- You can pass multiple parameters using query string and use the & for parameters separating as follow:
- https://localhost:7296/Home/testparam?num1=10&num2=50

```
public string testParam(int num1, int num2) {
return $" {num1} + {num2} = {num1 * num2}";
}
```





Any Questions?


THANK
YOU!