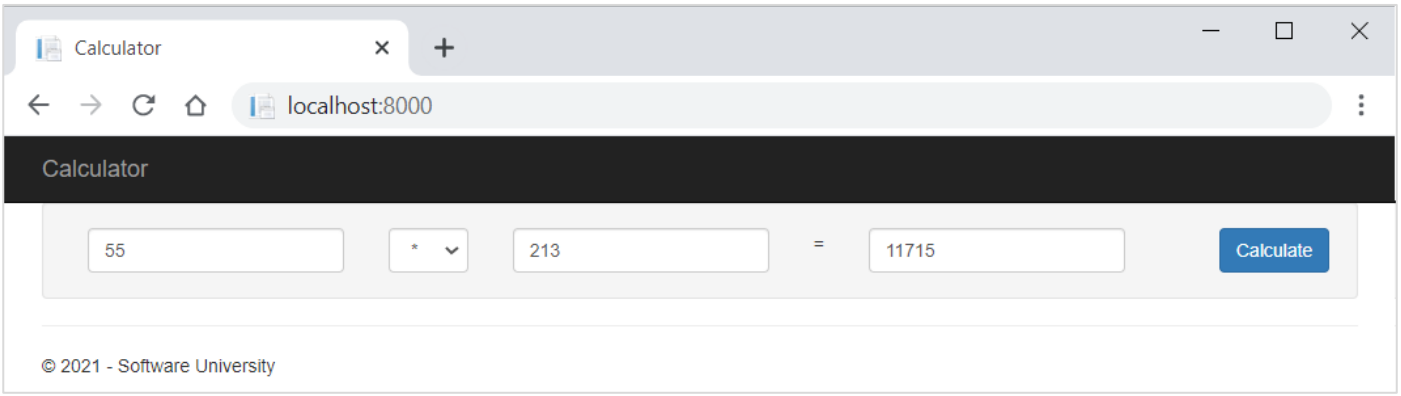
Exercises: ASP.NET Overview

Use the provided skeletons from the resources!

1. Calculator

This document defines a complete walkthrough of creating a **Calculator** application with the ASP.NET MVC Core, from setting up to implementing the fully functional application. The app will look like this:

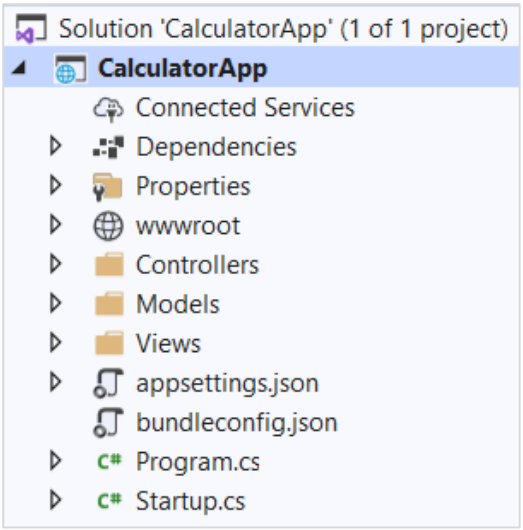


Base Project Overview

Our project will be built, using the **C#** language and the **MVC** framework **ASP.NET**. We'll use the **Razor View Engine** to define our views.

Open the Project

Let's take a look at the **project structure**:

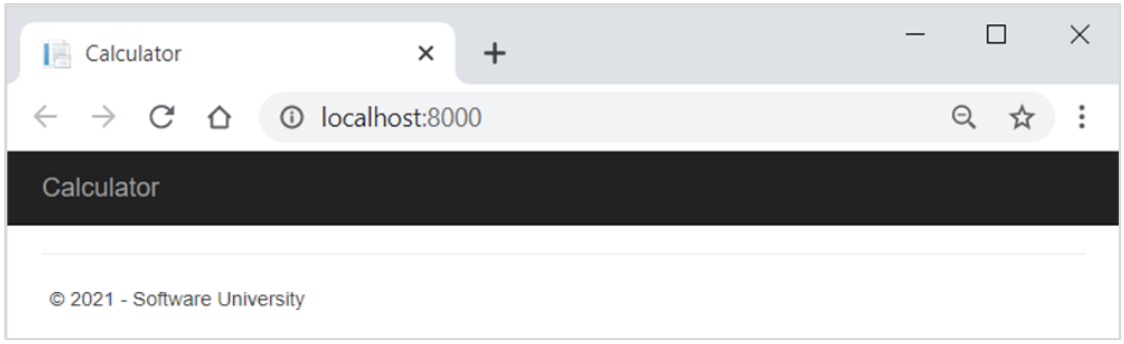


We can see several folders here. Let's look at the most important of them and see what are they for:

1. **Controllers –** we'll put all of our controllers here.
2. **Models** – model classes (we'll put our Calculator model here).
3. **Views** – we'll store our **view templates** here. We'll be using the template engine **Razor**.

Run the Project

Now that we've opened the project, let's try running it, so we can see what we're working with. Press **[Ctrl+F5]** to compile the project and run the server. The page will automatically open in your default browser (note: the **port** might be **different** than the screenshot):



It doesn't look like much, but at least we have the basic layout down! Let's get to work on implementing some functionality!

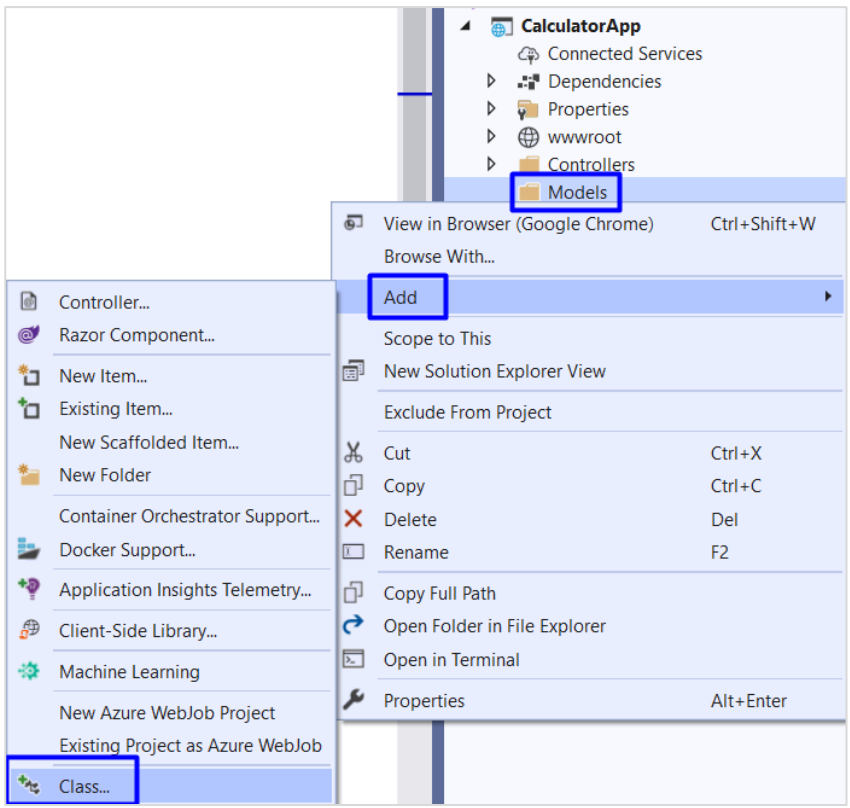
Implement Functionality

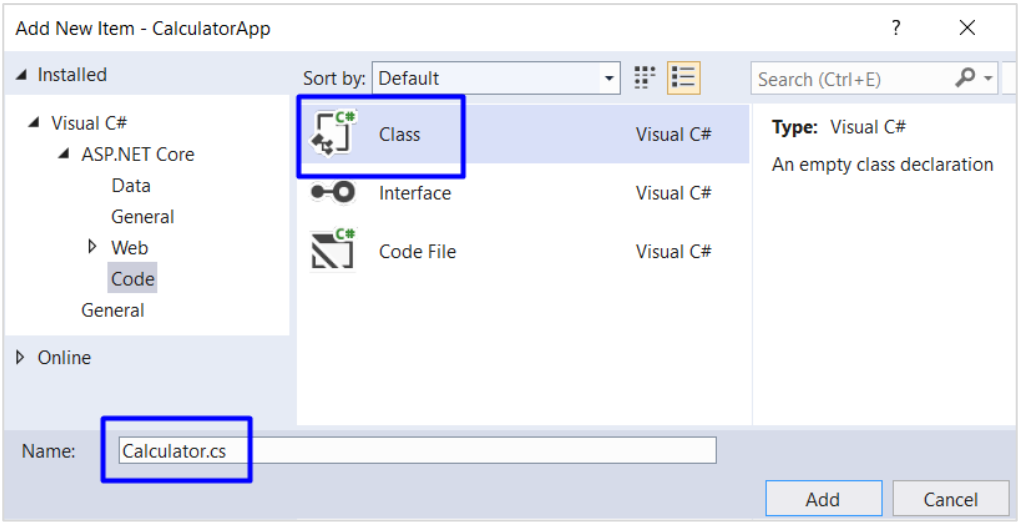
Create Calculator Model

It's time to design our main model – the **Calculator**. It will contain the following properties:

- LeftOperand
- RightOperand
- Operator
- Result

Let's create our model. Since we're **not** using a database in this exercise, we're just going to define the calculator as a **simple C# class** (the only difference between C# classes and Entity Framework models is that EF models might have attributes, which help it name database columns and set restrictions). Go into the **Models** folder and create a new C# class, called "**`Calculator.cs`**", using [Right click → Add → Class]:





1. **Define** the calculator **properties**:

```
namespace CalculatorApp.Models
{
    0 references
    public class Calculator
    {
        0 references | 0 exceptions
        public decimal LeftOperand { get; set; }
        0 references | 0 exceptions
        public decimal RightOperand { get; set; }
        0 references | 0 exceptions
        public string Operator { get; set; }
        0 references | 0 exceptions
        public decimal Result { get; set; }
    }
}
```

2. Create a **constructor** for **instantiating** the calculator:

```
namespace CalculatorApp.Models
{
    1 reference
    public class Calculator
    {
        0 references | 0 exceptions
        public Calculator()
        {
            this.Result = 0;
        }
        0 references | 0 exceptions
        public decimal LeftOperand { get; set; }
        0 references | 0 exceptions
        public decimal RightOperand { get; set; }
        0 references | 0 exceptions
        public string Operator { get; set; }
        1 reference | 0 exceptions
        public decimal Result { get; set; }
    }
}
```

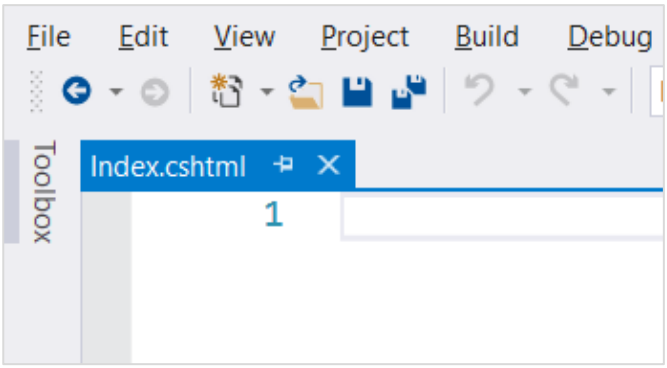
Now all that's left is to connect it to the rest of our little web application.

For our final trick, we'll create our own controller action, which will **process** what the user sent us and **return** a **view** with the **result** from the calculation.

Create Calculator View

Before we can have any functionality, it would be nice to have an idea of what we're working against, so let's go ahead and **create** a **form**, which the **user** will use for **calculations**:

Go into the **/Views/Home/** folder and open the **Index.cshtml** file:

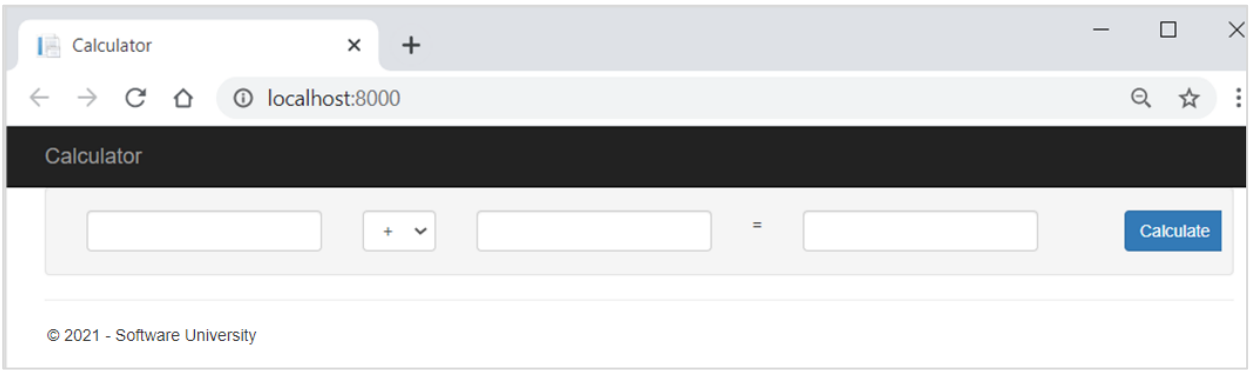


It's empty?! How does the header and footer seen above get displayed then? The answer is, we use a global **layout** file (**/Views/Shared/_Layout.cshtml**), so we don't have to copy-paste our page layout into every single view in our project (which could have tens or hundreds of views). All the **actual base design HTML** is inside **_Layout.cshtml**. We won't be touching that, so let's go to the **Index.cshtml** file and add our form:

```
@model CalculatorApp.Models.Calculator

@{
    ViewBag.Title = "Calculator";
}

<div class="well">
    @using (Html.BeginForm("Calculate", "Home", FormMethod.Post , new { @class = "form-inline"}))
    {
        <fieldset>
            <div class="form-group">
                <div class="col-sm-1">
                    @Html.TextBoxFor(model => model.LeftOperand, new { @class = "form-control" })
                </div>
            </div>
            <div class="form-group">
                <div class="col-sm-4">
                    @Html.DropDownListFor(model => model.Operator,
                    new [] {
                        new SelectListItem { Text = "+", Value = "+" },
                        new SelectListItem { Text = "-", Value = "-" },
                        new SelectListItem { Text = "*", Value = "*" },
                        new SelectListItem { Text = "/", Value = "/" },
                    }, new { @class = "form-control" })
                </div>
            </div>
            <div class="form-group">
                <div class="col-sm-2">
                    @Html.TextBoxFor(model => model.RightOperand, new { @class = "form-control" })
                </div>
            </div>

            <div class="form-group">
                <div class="col-sm-2 ">
                    <p>=</p>
                </div>
            </div>

            <div class="form-group">
                <div class="col-sm-2">
                    @Html.TextBoxFor(model => model.Result, null, new { @class = "form-control" })
                </div>
            </div>
```

```
            <div class="form-group">
                <div class="col-sm-4 col-sm-offset-4">
                    <button type="submit" class="btn btn-primary">Calculate</button>
                </div>
            </div>
        </fieldset>
    }
</div>
```

Now we will **save the state** of the operands and operator for ease of use, so the **Razor syntax** you see here does just that. The **`SelectListItem`** template is a bit more special: it selects the operator from the dropdown list, **based on** the last used operator. We'll see how that's implemented a bit later. For now, let's navigate to our web app and see how we're doing (remember to recompile the project beforehand, using **`[Ctrl+Shift+B]`**:



Let's see how this all ties together. Go into **`/Views/Shared/_Layout.cshtml`**:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Calculator</title>

    <environment>...</environment>
    <environment>...</environment>
</head>
<body>
    <nav class="navbar navbar-inverse navbar-fixed-top">...</nav>
    <div class="container body-content">
        @RenderBody()
        <hr />
        <footer>
            <p>&copy; 2018 - Software University</p>
        </footer>
    </div>
```

The **`@RenderBody()`** line of code expects to be fed a **view template** to display around the header and footer. But how does it know **which view** to render? Let's go into the **`HomeController.cs`** file and check out what the **index** action does:

```
namespace CalculatorApp.Controllers
{
    0 references
    public class HomeController : Controller
    {
        0 references | 0 requests | 0 exceptions
        public IActionResult Index()
        {
            return View();
        }

    }
}
```

ViewResult Controller.View() (+ 3 overloads)
Creates a ViewResult object that renders a view to the response.

Returns:
The created ViewResult object for the response.

As you can see, the **`Index`** action in **`HomeController.cs`** returns the **`Index.cshtml`** view inside the **`Views/Home`** folder. **ASP.NET** is smart enough to figure out **which view** to return, based on the **controller** it's inside and the **name** of the **method** (and **generate routes automatically**).

*It's actually not as magical as you think - this is all defined in the **`StartUp.cs`** class:*

```
public class Startup
{
    0 references | 0 exceptions
    public Startup(IConfiguration configuration)...

    1 reference | 0 exceptions
    public IConfiguration Configuration { get; }

    // This method gets called by the runtime. Use this method to add services to the container.
    0 references | 0 exceptions
    public void ConfigureServices(IServiceCollection services)...

    // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
    0 references | 0 exceptions
    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (env.IsDevelopment())...
        else...

        app.UseStaticFiles();

        app.UseMvc(routes =>
        {
            routes.MapRoute(
                name: "default",
                template: "{controller=Home}/{action=Index}/{id?}");
        });
    }
}
```

So, for example, if we had to render an **article details** view, we would create a "**`Details`**" method inside **`ArticleControler.cs`**, and ASP.NET would **automatically** map the **`/Article/Details/{id}`** route and also try to find the view, located in the "**`Views/Article`**" folder.

Implement the Controller Action

Now that we've created the **view**, which will **hold our data** and allow the **user** to **interact** with our web application, it's time to implement the driving force behind the whole app – **the controller action**.

As it turns out, we already have a **home controller** set up, and an action, set up on the "**/**" route, otherwise we wouldn't even be able to see our calculator. You can find the **home controller** in the **Controllers** folder. Let's see what it looks like:

```
namespace CalculatorApp.Controllers
{
    0 references
    public class HomeController : Controller
    {
        0 references | 0 requests | 0 exceptions
        public IActionResult Index()
        {
            return View();
        }

    }
}
```

Not much going on here… Let's break it down:

- **`public ActionResult Index()`** → This is the actual **controller action**. It's a method, which **holds the logic**, which will be **executed**, when it's **called**.
- **`return View()`** → This function **renders** a **view** in the **response** (in essence, takes whatever's inside of "**`Views/Shared/_Layout.cshtml`**", sends it whatever's inside "**`Views/Home/Index.cshtml`**", runs it through the **Razor** templating engine, and returns it to the user.

So, using that newfound knowledge, let's try to create our own **action**.

First, we need to modify our Index action to return an instance of our Calculator model. We'll do it this way, so we can redirect to this action to display the result whenever we calculate it. We're going to go into the **`Index`** action and modify the **method signature** and the **return value**:

```
public ActionResult Index(Calculator calculator)
{
    return View(calculator);
}
```

Now that we've modified the index action, it's time to create the action, which will **calculate the result**.

First, let's start off by declaring what kind of **HTTP method** this method will be handling (either GET or POST). In our case, since we're processing **form data**, we'll add an **`[HttpPost]` attribute**:

```
[HttpPost]
```

Under it, let's **declare** our Calculate method. Since the form in the view is defined by a **special Razor form syntax**, we can just pass a **parameter** of the **Calculator** type to the method and it'll automatically populate it with the form data:

```
[HttpPost]
public ActionResult Calculate(Calculator calculator)
{

}
```

All this method should do at this point is **calculate** the result and return the **`Index`** view with all the data (which the view can get from the **calculator object** itself:

```
[HttpPost]
public ActionResult Calculate(Calculator calculator)
{
    calculator.Result = CalculateResult(calculator);

    return RedirectToAction("Index", calculator);
}
```

Let's see what a **debug session** would show us if we were to **debug** this method:


```
[HttpPost]
public ActionResult Calculate(Calculator calculator)
{
    //calculator.Result = CalculateResult(calculator
    return RedirectToAction("Index", calculator);
}
```

calculator {Calculator_CSharp.Models.Calculator}
LeftOperand 2
Operator "+"
Result 0
RightOperand 3

The **`LeftOperand`**, **`Operator`**, and **`RightOperand`** variables are automatically **parsed** as **decimal**. All that's left is to calculate the actual result. Create a **`CalculateResult`** method inside the **`HomeController.cs`** class:

```
private decimal CalculateResult(Calculator calculator)
{

}
```

All that's left is to implement the calculation logic:

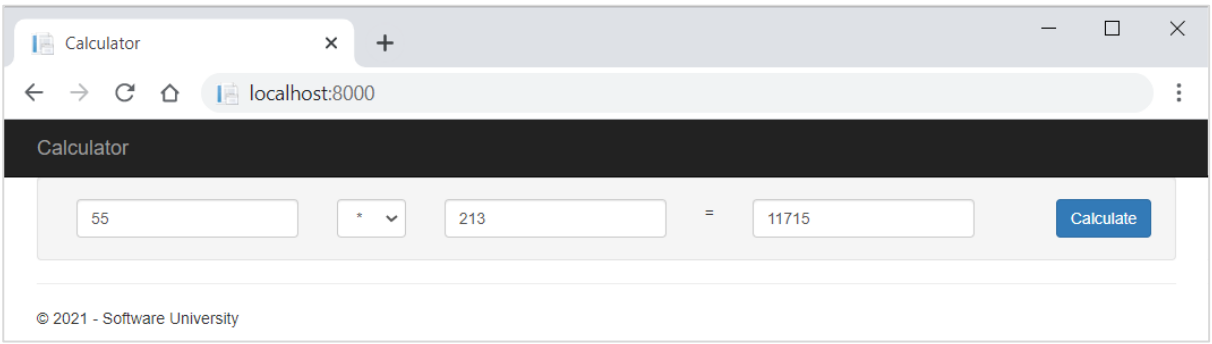
```
private decimal CalculateResult(Calculator calculator)
{
    var result = 0m;

    switch (calculator.Operator)
    {
        case "+":
            result = calculator.LeftOperand + calculator.RightOperand;
            break;
        case "-":
            result = calculator.LeftOperand - calculator.RightOperand;
            break;
        case "*":
            result = calculator.LeftOperand * calculator.RightOperand;
            break;
        case "/":
            result = calculator.LeftOperand / calculator.RightOperand;
            break;
    }

    return result;
}
```

Now that we've implemented the controller action, it should look like this:

```
[HttpPost]
public ActionResult Calculate(Calculator calculator)
{
    calculator.Result = CalculateResult(calculator);

    return RedirectToAction("Index", calculator);
}
```

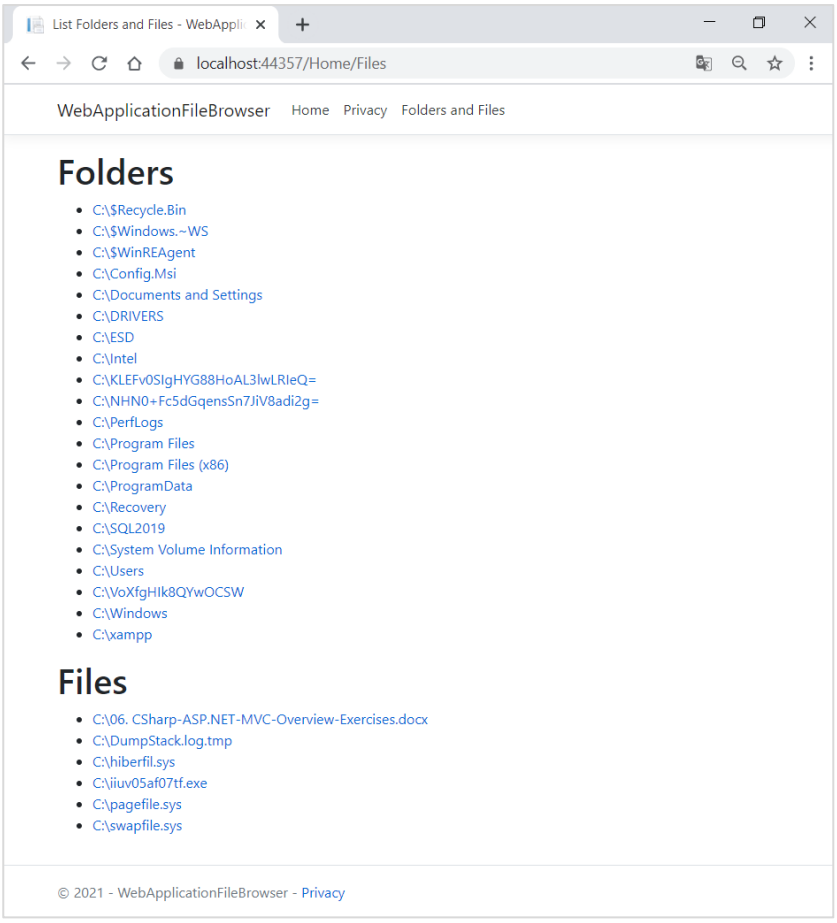
Test the Application

All our hard work should finally pay off now, right? If you've followed all the steps properly, and have **read all the explanatory text**, hopefully we should have a functioning calculator! Rebuild the application, using **`[Ctrl+Shift+B]`** and test it:



2. File Browser

Implement a **file and directory browser** in ASP.NET MVC, which shows the files and folder of drive **`C:\`** and allows browsing the directories and downloading files.

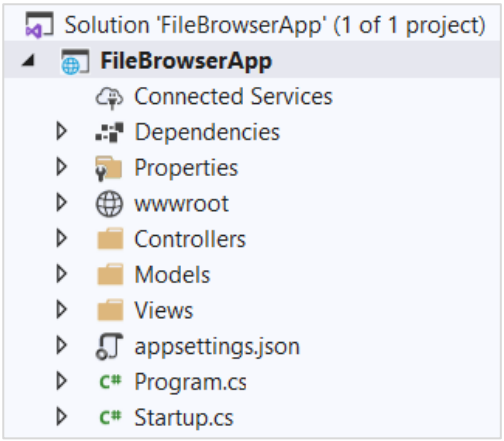


Base Project Overview

Our project will be built, using the **C#** language and the **MVC** framework **ASP.NET**. We'll use the **Razor View Engine** to define our views.

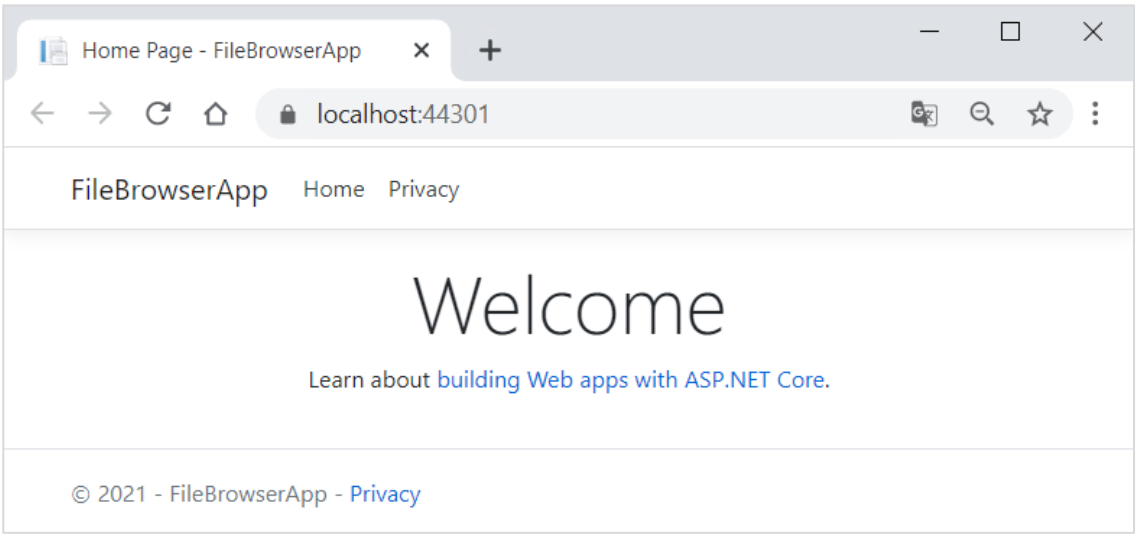
Open the Project

Let's take a look at the **project structure**:



Run the Project

Now that we've opened the project, let's try running it, so we can see what we're working with. Press **[Ctrl+F5]** to compile the project and run the server. The page will automatically open in your default browser (note: the **port** might be **different** than the screenshot):



It doesn't look like much, but at least we have the basic layout down! Let's get to work on implementing some functionality!

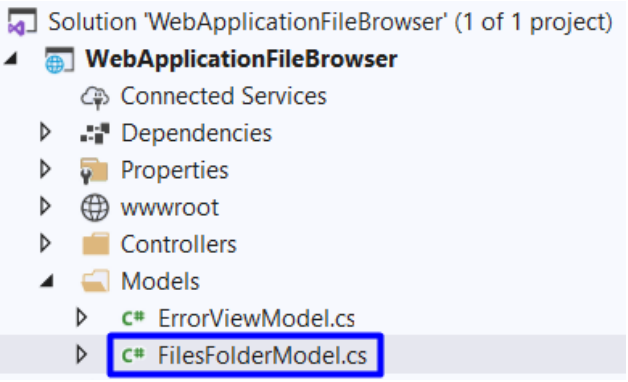
Implement Functionality

Create FilesFolder Model

It's time to design our main model – **`FilesFolderModel.cs`**. It will contain the following properties:

- ParentFolder
- Folders
- Files

Let's create our model. Go into the **Models** folder and create a new C# class, called "**`FilesFolderModel.cs`**", using **`Right click → Add → Class`**.



Define properties as shown below:

```
namespace WebApplicationFileBrowser.Models
{
    3 references
    public class FilesFolderModel
    {
        2 references
        public string ParentFolder { get; set; }
        2 references
        public string[] Folders { get; set; }
        2 references
        public string[] Files { get; set; }
    }
}
```

Now all that's left is to connect it to the rest of our little web application.

Next, we'll create our own **controller action**, which will **process** what the user sent us and **return** a **view** with the **result** from browsing.

Implement the Controller "Files" Action

First, go to **`Controllers -> HomeController.cs`** and add an **action** as shown below. We use "**`C:\`**" as the directory we will browse in.

```
public IActionResult Files(string path = @"C:\")
{

}
```

Then, we use **`Directory.GetParent(path)`** method to get the parent directory of the directory we are currently in:

```
public IActionResult Files(string path = @"C:\")
{
    var parentDir = Directory.GetParent(path);
}
```

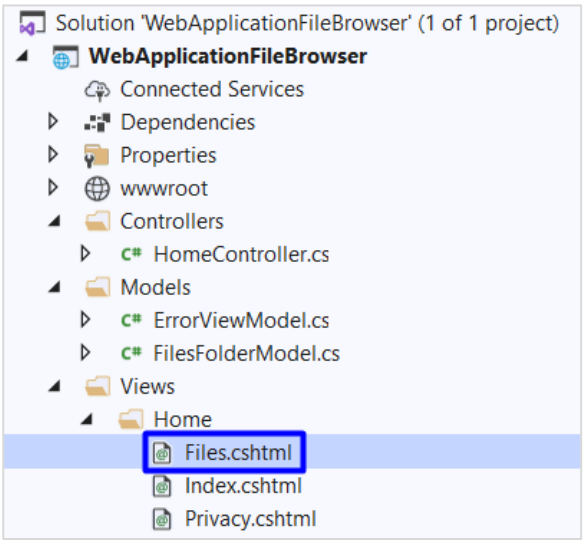
Next, we use the **`FilesFolderModel`** we already created, and get the folders and files from our directory. At last, we should return a **view** with our model:

```
public IActionResult Files(string path = @"C:\")
{
    var parentDir = Directory.GetParent(path);
    var model = new FilesFolderModel()
    {
        ParentFolder = parentDir?.FullName,
        Folders = Directory.GetDirectories(path).ToArray(),
        Files = Directory.GetFiles(path).ToArray()
    };
    return View(model);
}
```

Now we need to create the view our action is going to use.

Create View

Create **`Files.csthml`** view in **`Views -> Home`**.



Then **paste** the following code in the **`Files.csthml`** file:

```
@{
    ViewData["Title"] = "List Folders and Files";
}
@model FilesFolderModel

<h1>Folders</h1>
<ul>
    @if (Model.ParentFolder != null)
    {
        <li>
            @Html.ActionLink("(parent)", "Files", new { path = Model.ParentFolder })
        </li>
    }
    @foreach (string folder in Model.Folders)
    {
        <li>
            @Html.ActionLink(folder, "Files", new { path = folder })
        </li>
    }
</ul>

<h1>Files</h1>
<ul>
    @foreach (string file in Model.Files)
    {
        <li>
            @Html.ActionLink(file, "DownloadFile", new { path = file })
```

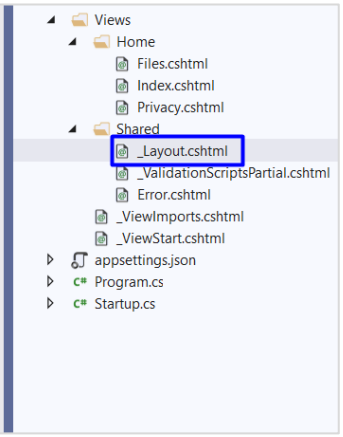
```
        </li>
    }
</ul>
```

Note that we use **`Html.ActionLink()`** method to **create links** and **browse** through the **file system**.

Modify Page Layout

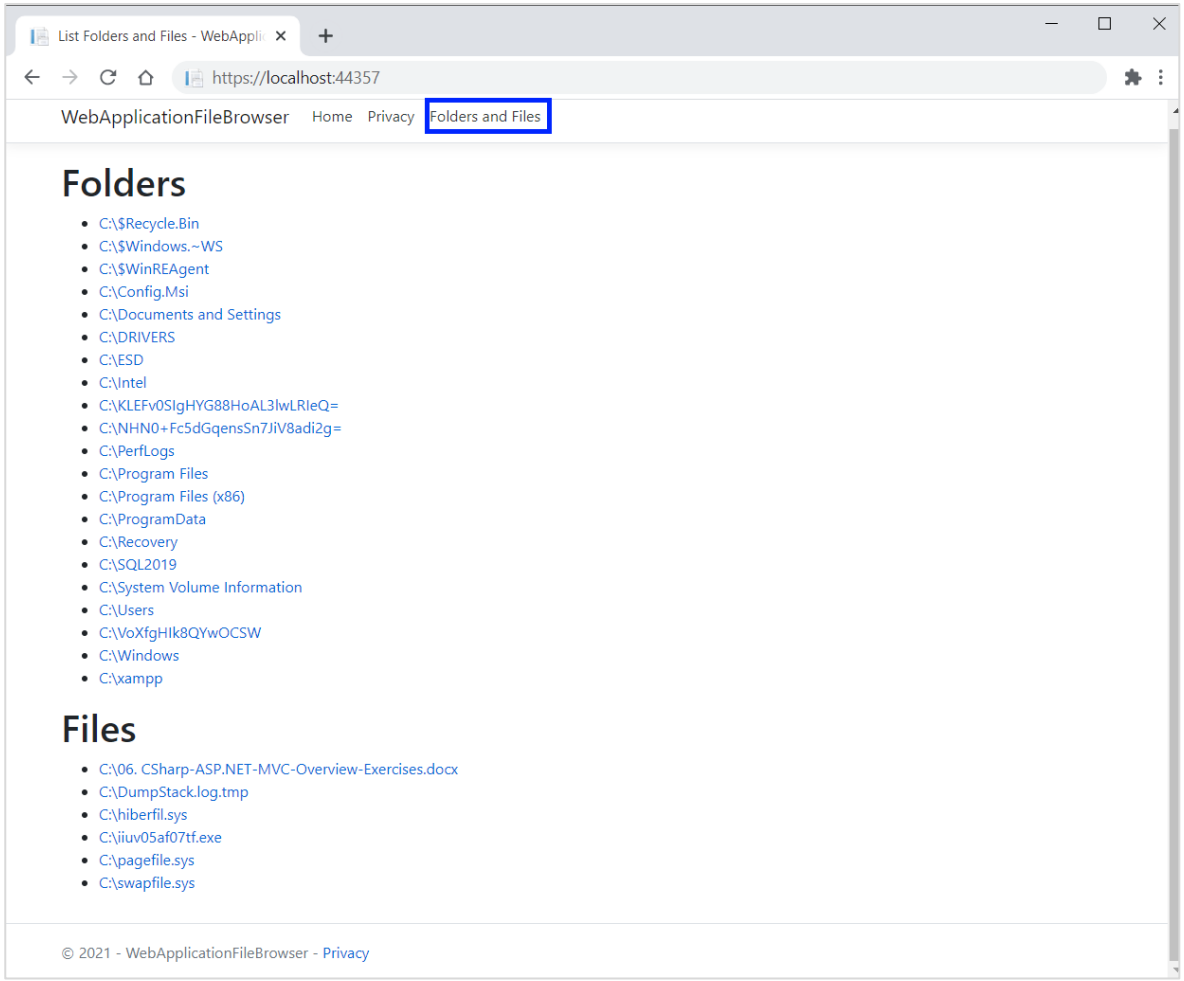
To make changes to the **page layout**, go to **`Views -> Shared -> _Layout.cshtml`**. Then, add a new **`<li>`** tag for **navigation** as shown below:

```
<ul class="navbar-nav flex-grow-1">
    <li class="nav-item">
        <a class="nav-link text-dark" asp-area="" asp-controller="Home"
           asp-action="Index">Home</a>
    </li>
    <li class="nav-item">
        <a class="nav-link text-dark" asp-area="" asp-controller="Home"
           asp-action="Privacy">Privacy</a>
    </li>
    <li class="nav-item">
        <a class="nav-link text-dark" asp-area="" asp-controller="Home"
           asp-action="Files">Folders and Files</a>
    </li>
</ul>
```



Run the App

Press **`[Ctrl+F5]`** to **run** the application. It should look like this:



Note that all **folders** and **files** from your "**C:**" drive are displayed. Now we are almost ready – our final step is to create a controller action for downloading files.

Implement the Controller "DownloadFile" Action

In order to be able to **download files**, we should create an **action**. To do so, go to **`Controllers -> HomeController.cs`** and create **`DownloadFile()`** method, which accepts the **file's path** the following way:

```
public FileResult DownloadFile(string path)
{

}
```

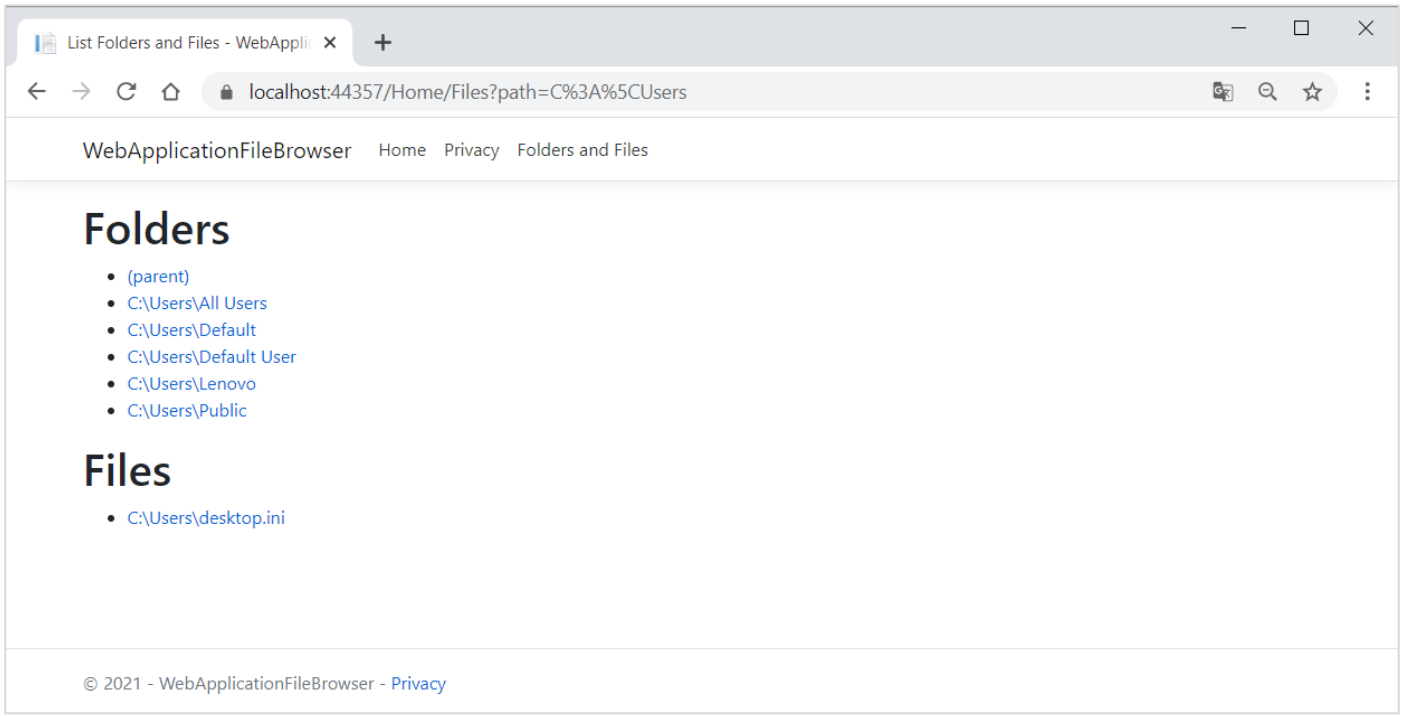
Then, get the file as an **array of bytes**, as well as the **information** of the file, using the **`FileInfo`** class. Return the **file**. Your method should look like this:

```
public FileResult DownloadFile(string path)
{
    byte[] bytes = System.IO.File.ReadAllBytes(path);
    FileInfo fInfo = new FileInfo(path);
    return File(bytes, "application/octet-stream", fInfo.Name);
}
```

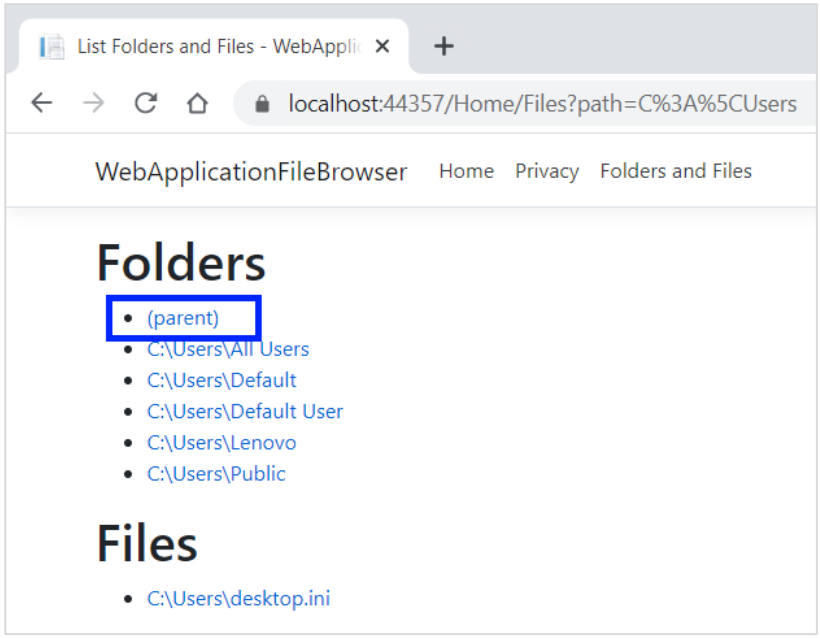
Now we can also download files from our file system through our app.

Test the Application

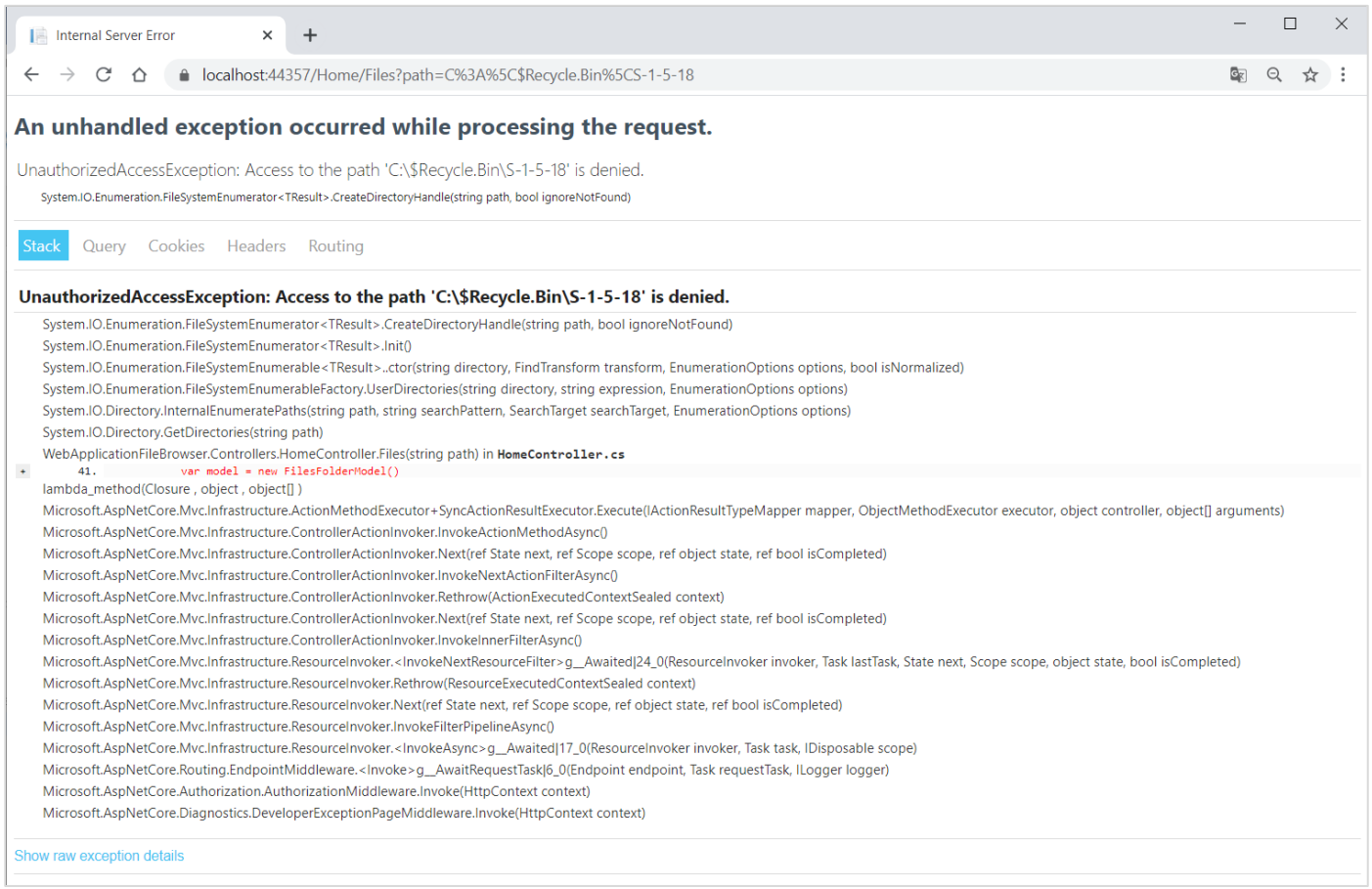
Run the application again using **`[Ctrl+F5]`** and navigate to **`[Folders and Files]`**. Now test the application by **browsing** through folders. For example, if you go to "**`C:\Users`**", you should see folders and files in it. It may look similar to this:



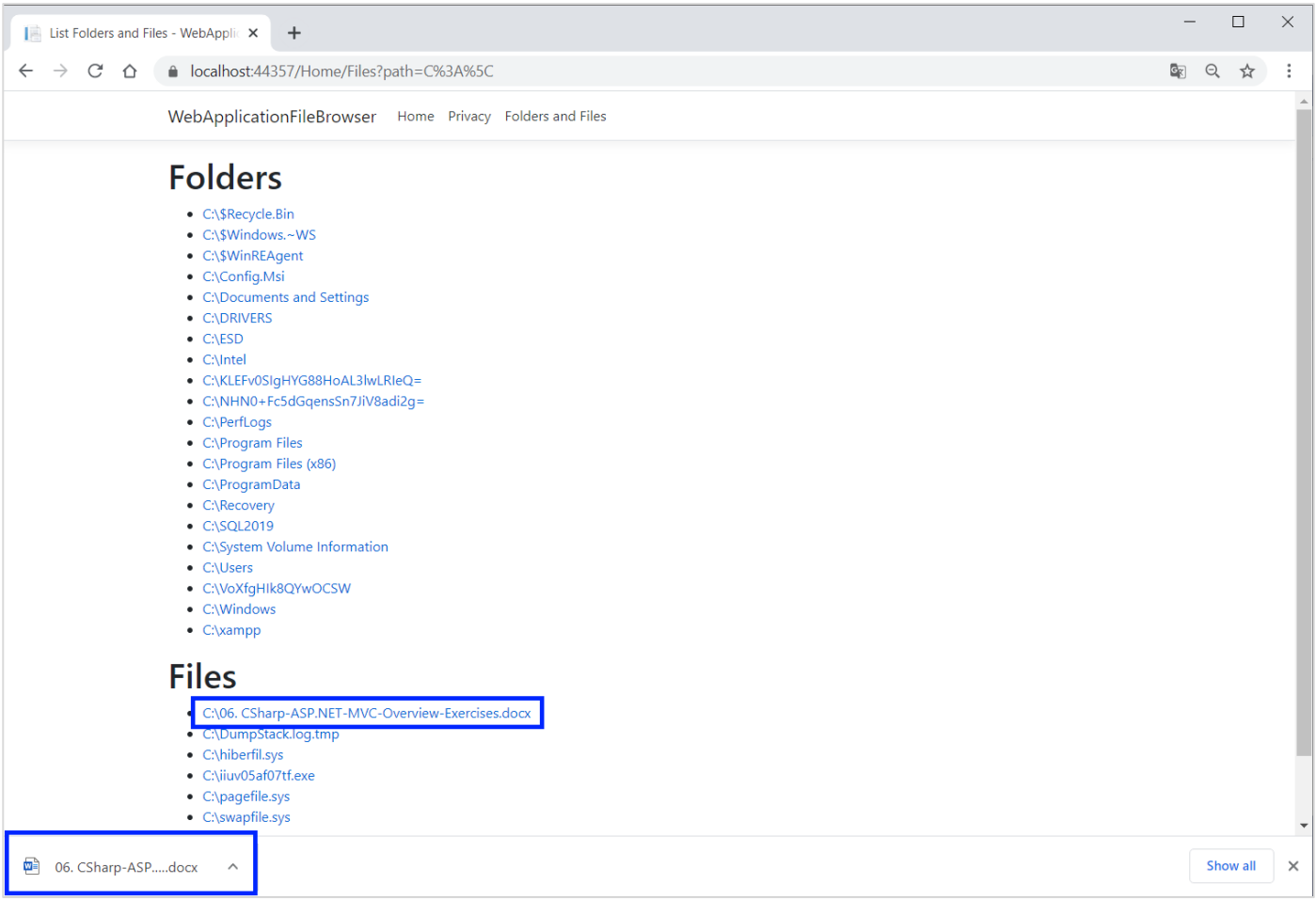
Note that we have a route to our **parent directory**. Click on **`[(parent)]`** to go back to the previous page.



Try browsing other folders as well. However, **access** to some folders is **denied**, so do not worry if the following error appears.



Finally, try **downloading** a file. For example, place this exercise's **.docx file** into the "**C:**" directory and **click** on it. It should be downloaded like this:



Our "**`File Browser`**" ASP.NET MVC app is now ready and you can use it for **browsing** through your folders and for **downloading** files.