

# Hello World MVC

## ASP.Net Core

### CISS 233 C#

## Table of contents:

1. Page 3 **Windows: Hello World MVC ASP.Net Core**
2. Page 6 **Mac: Hello World MVC ASP.Net Core**
3. Page 10 **What is MVC? [Model – View - Controller]**
4. Page 11 **Coding the Home View**
5. Page 18 **Coding the Home Controller**
6. Page 19 **What about the Model?**

# Windows: Hello World MVC ASP.Net Core

## Windows Version

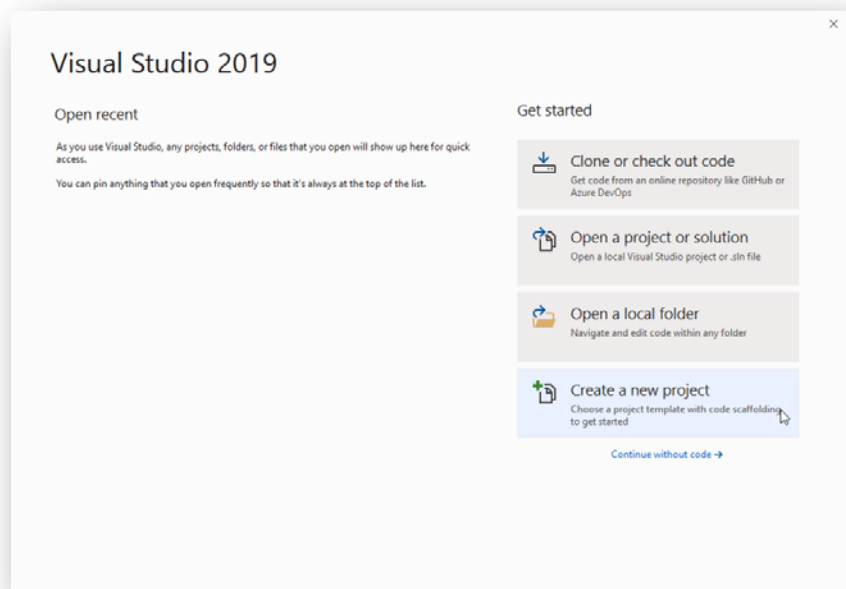
In this tutorial we're going to create an MVC ASP .NET Core Hello World Application.



Open Visual Studio 2019.

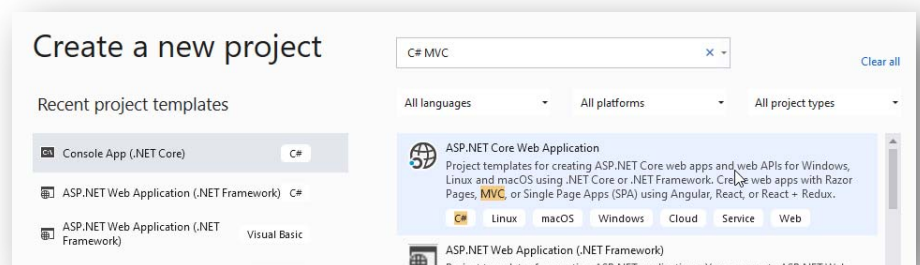
I'm going to create a new project from the splash screen menu.

I'll choose "Create a new project".

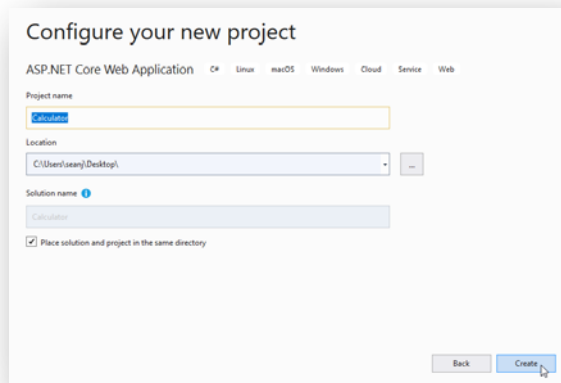


The first time you want to create an MVC .NET Core application you'll need to search to find that project type.

Search on C# MVC. Highlight ASP .NET Core Application and then click next.

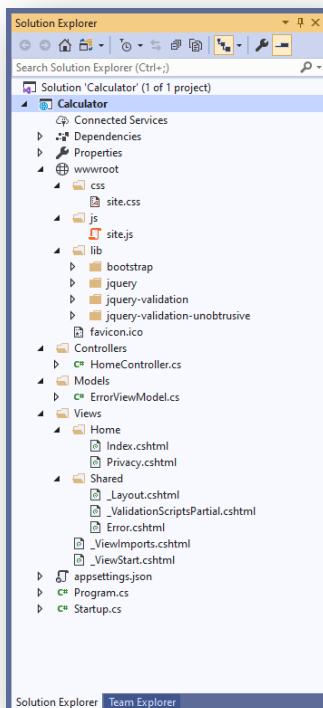
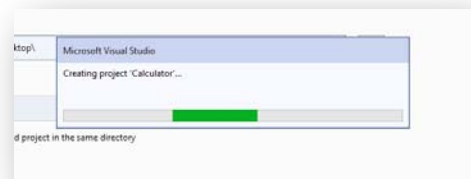
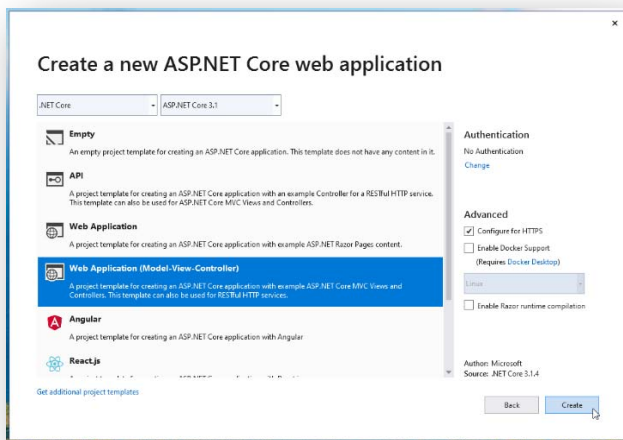


The name of the application I will create will be Calculator.



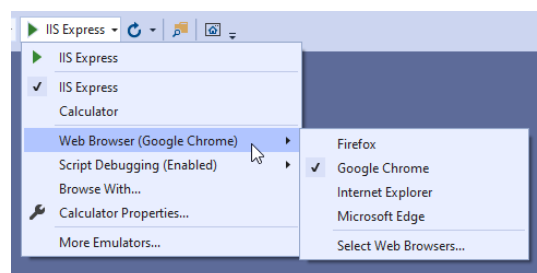
I'm going to create a web application (Model-View-Controller).

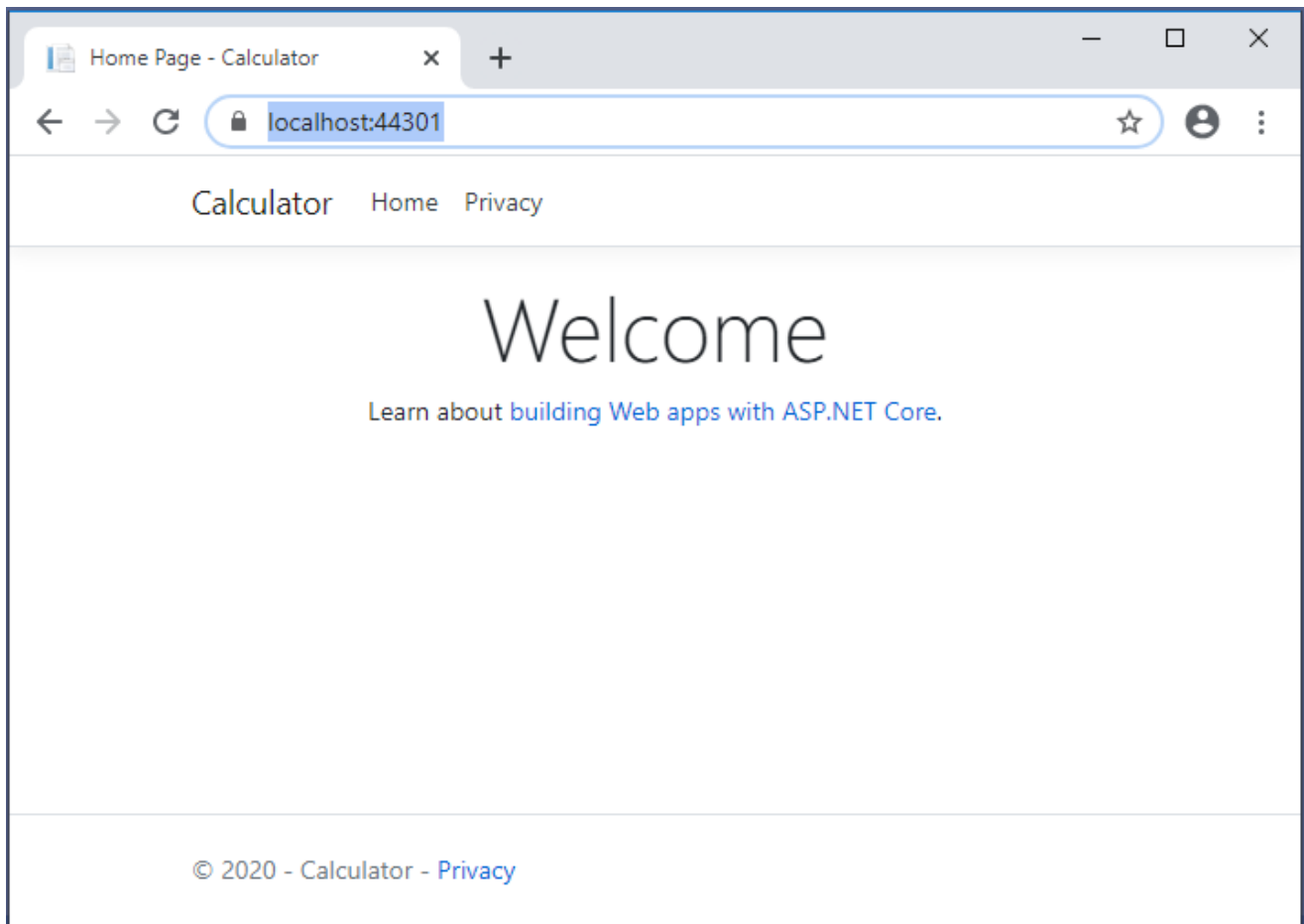
Here my project is building.



To the left are the files that are created by default when you create this type of application. We will discuss these in more detail in a bit. Let's run the code first though to see what it does.

We do this by clicking the IIS Express button. You can choose which browser you'd like to debug in by clicking the down arrow to the right of iis express and then selecting a browser under web Browser.





Here is the code running in the browser. When yours builds your port number will most likely be different. Don't worry about that. I believe that's chosen the first time you build.

Let's talk about Model View Controller. Turn to the **"What is MVC?"** section.

# Mac: Hello World MVC ASP.Net Core

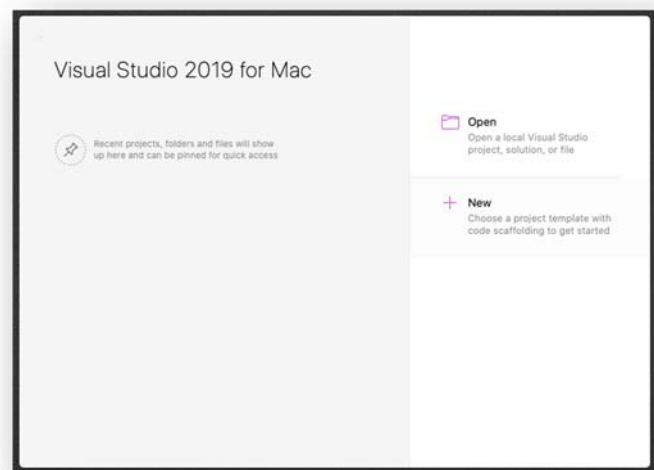
## Mac Version

In this tutorial we're going to create an MVC ASP .NET Core Hello World Application.



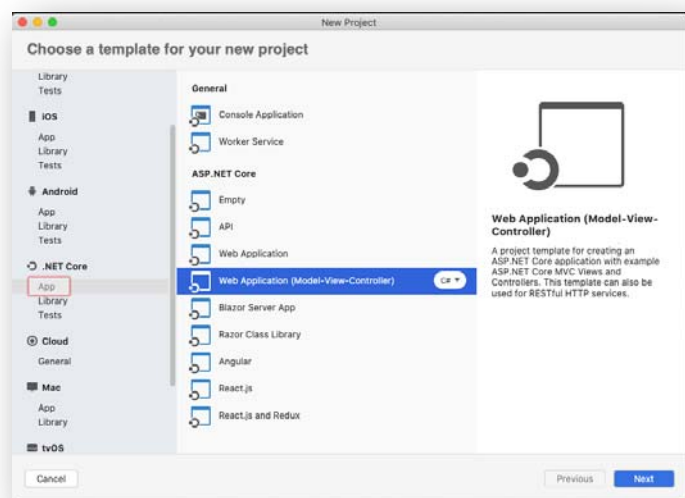
Open Visual Studio for Mac 2019.

I'm going to create a new project from the splash screen menu.



I'll choose .NET Core -> App under types on the left and then Web Application C# (Model – View – Controller).

Then I'll click next.

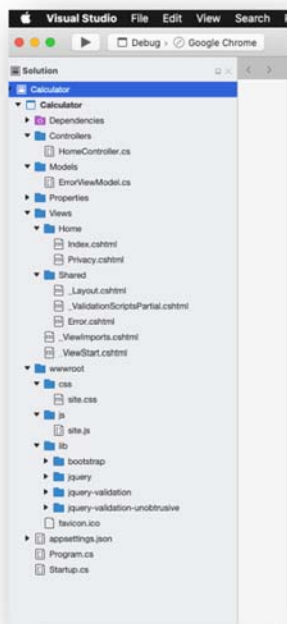
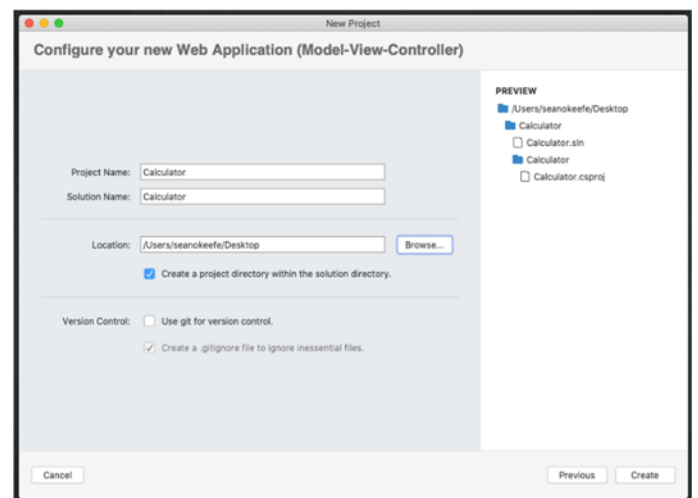




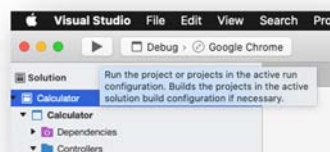
The default settings are fine.

.NET Core 3.1 or whatever the highest version is on your system, and no Authentication.

The name of the application I will create will be Calculator.



To the left are the files that are created by default when you create this type of application. We will discuss these in more detail in a bit. Let's run the code first though to see what it does.



To run the code, click on the play button to the left of the debug.

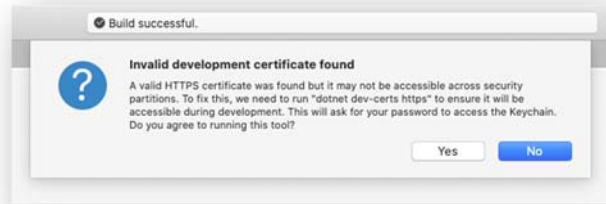
You can choose which browser to run it in. I suggest Chrome.



On my Mac I ran into a certificate issue with Visual Studio saying I needed a signed Microsoft certificate in the keychain so if you have the same issues then just follow the prompts. I will show you what I saw next.

I got this message when trying to run the project.

I clicked Yes and followed the directions.



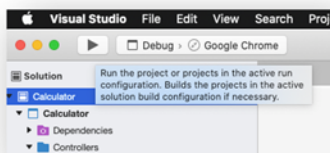
It might be hard to read because of my terminal colors. This allows you to build the certificate.

I next entered my Mac password.

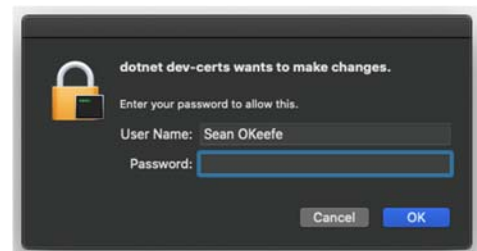
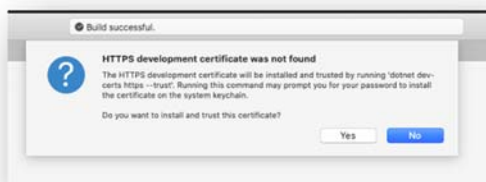


It will ask you to create a password for the certificate. I suggest making it your Mac password.

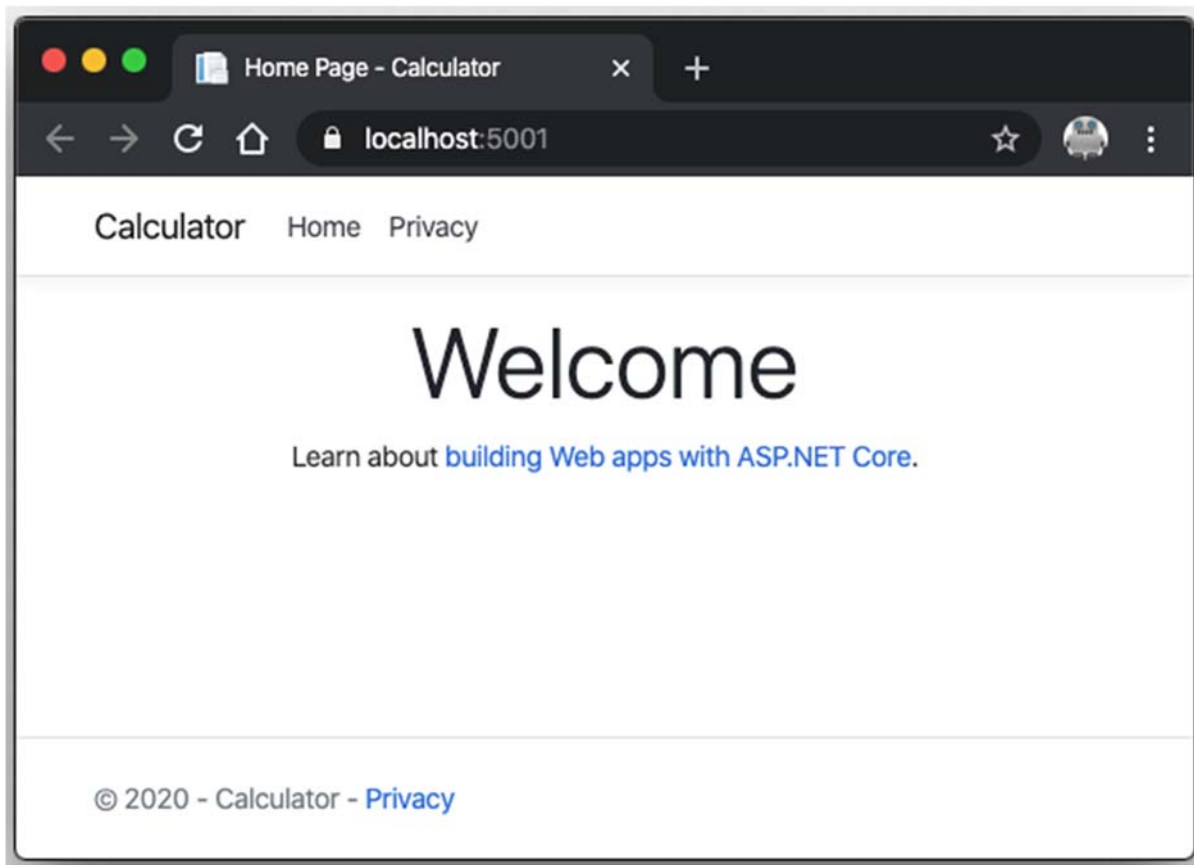
Eventually after it runs you can click the run button again.



It will probably again tell you the certificate was not found. Click yes and it will prompt you for your password. Not sure if your mac or certificate password so that's why I'm glad I made the cert password is the same.







Here is the code running in the browser. When yours builds your port number will most likely be different. Don't worry about that. I believe that's chosen the first time you build.

Let's talk about Model View Controller. Turn to the **"What is MVC?"** section.

# What is MVC? [Model – View - Controller]

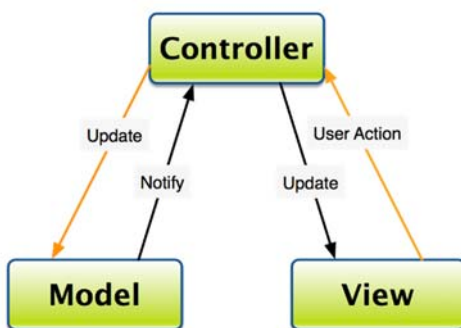
What is MVC? MVC is Model-View-Controller. MVC is a way to design a web project into three discrete distinct components to distribute functionality between tiers.

Starting with the user, the user sees the **View**. The **View** is all of the HTML that is displayed in the browser. All of your HTML/CSS/JavaScript is in the View.

The **Model** is what represents the data or connects to the database. Often in the **Model** there are entities that represent tables in the database.

The **Controller** is the conduit that connects the **View** and the **Model**. It handles the requests and responses that flow between the **View** and the **Model**.

Here is a simplified diagram:



This diagram from

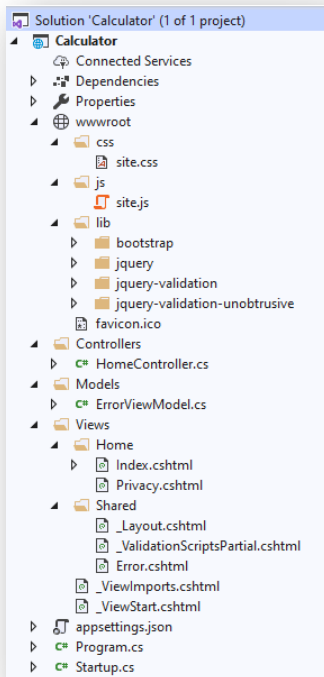
<https://stackoverflow.com/questions/29594105/mvc-is-it-model-to-view-or-controller-to-view/29597838>

shows a simplified flow between the layers or tiers.

An application can be written as MVC and never implement a **Model**. This is because if the application has no database then the **Model** is not necessary. In the Calculator example that I'm writing the program is just going to add, subtract, multiply, and divide two numbers. There will be no storage of data. I won't even touch the **Model** for this example. All of my coding will be in the **View** and the **Controller**. In the next section I will be coding those to build my calculator. I will do all of my coding in the Home **View** and **Controller**. When a basic MVC application is created in Visual Studio usually you'll be given the Home **Controller** and **View**, and for a **Model** just a generic ErrorViewModel.

One more thing before we start coding the Home **View**, under Shared you'll find shared files for things like common page navigation and errors. Remember that they are there if you're trying to figure out where common page components live.

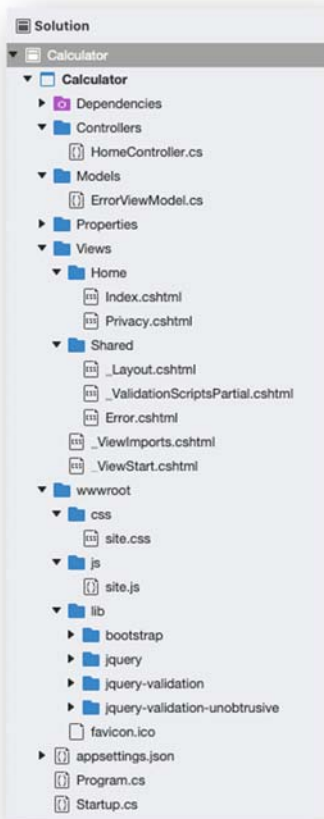
# Coding the Home View



Here is the solution in Visual Studio for Windows.

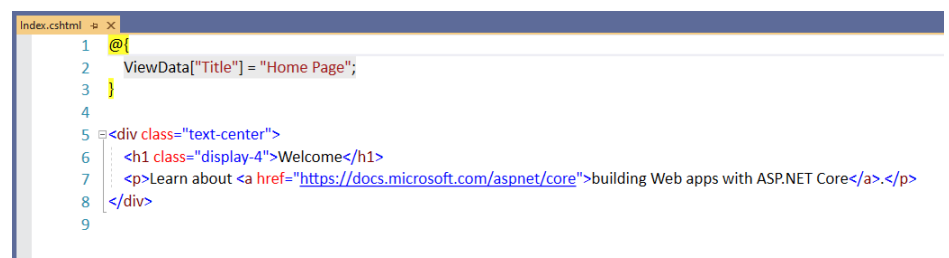
Notice the project is broken down into a `wwwroot` folder with your JavaScript and CSS folders as well as any third-party libraries that you're using in your project. By default, when Visual Studio creates an MVC project Bootstrap and jQuery are added. jQuery is a JavaScript library that makes JavaScript much easier to develop with. Bootstrap is the styling of the pages that allows your application to be responsive or reactive to the size of a user's screen. It allows it to scale and function based on the device you're viewing the application on.

The other portions of the project are **Controllers**, **Models**, and **Views** which are Home and Shared.



Here is the Mac solution and notice that it has exactly the same components.

To start coding we can open `index.cshtml` in Home under Views.

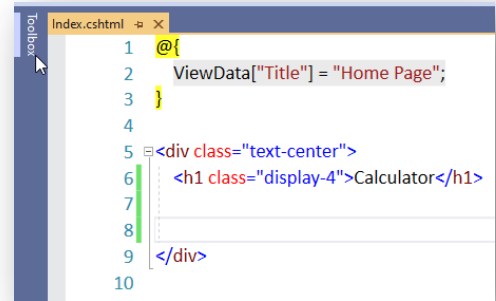


Here is the code in the Home **View** `index.cshtml`.

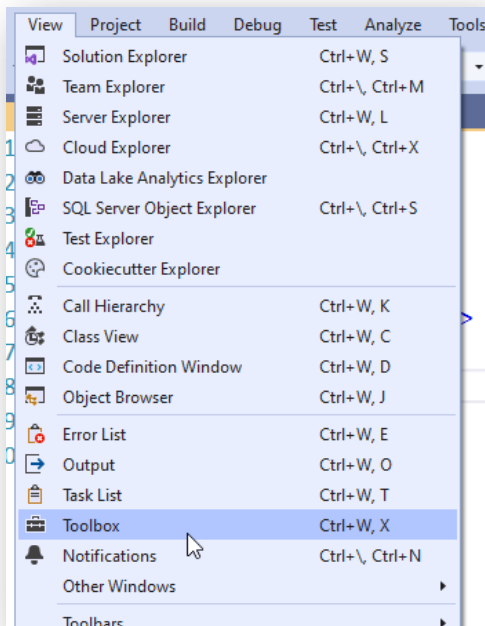
The first thing I'm going to do is change what's displayed so instead of Welcome being displayed it'll be Calculator. I'm also going to remove the link on line 7.

In Visual Studio for Windows you can access the Toolbox for HTML elements that you can drag and drop components onto the page.

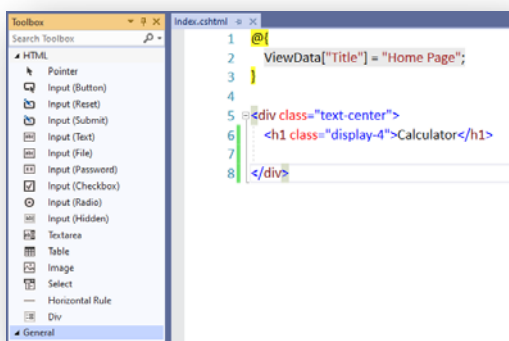
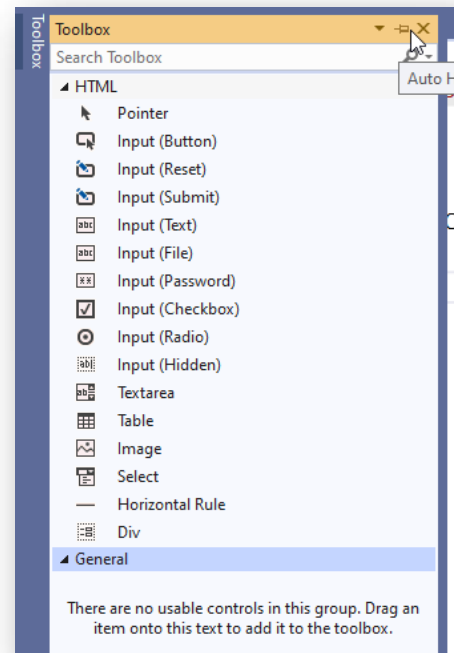
With the code window open you should see the Toolbox tab to the left of the code window.



If for some reason your Toolbox tab isn't visible you can click on **View -> Toolbox** and the tab should be there.



To pin the Toolbox open you can open the Toolbox and then click on the pushpin.



With the Toolbox pinned and the HTML label expanded you'll see all of the HTML elements available when a .cshtml code page is viewed in the editor. It must be some type of HTML page which cshtml is. You can then drag and drop elements onto the page.

For Visual Studio on Mac I haven't been able to find a Toolbox with HTML to drag onto the code window. Unfortunately, from what I've seen, Visual Studio for Mac doesn't provide you with a Toolbox for HTML so Mac users will have to code their HTML from scratch. In the last section of this guide I'll have an HTML guide.

For our calculator I'm going to add a form. I'll need a form to interact with the controller, do the calculations and return the results. I will explain more of how the controller works in the next section.

Other than the form I'm going to add two text boxes for the first and the second number. I'll also add a submit button, or else the form won't submit, and a reset button to clear the form. I'll also add a div for the results. Here is my code for the form:

```
5 <div class="text-center">
6   <h1 class="display-4">Calculator</h1>
7   <form asp-controller="Home" method="post">
8     <p>
9       Number 1:
10      <input id="tbNumber1" name="number1" type="number" step=".01" required />
11    </p>
12    <p>
13      Number 2:
14      <input id="tbNumber2" name="number2" type="number" step=".01" required />
15    </p>
16    <p>
17      <input id="btnSubmit" type="submit" name="submit" value="submit" />
18      &nbsp;
19      <input id="btnReset" type="reset" value="reset" />
20    </p>
21  </form>
22  <div id="divResult">
23  </div>
24 </div>
```

There are a few things I'd like to point out. The form is the vehicle to send data back to the server.

```
<form asp-controller="Home" method="post">
```

```
</form>
```

This code does a couple of things. It tells which controller to post to (`asp-controller="Home"`). In our case the **Home Controller** is the one that we link to the **Home View**. It also signifies that the way we will send data back to the server is through Post. There are two ways forms can be submitted, Get and Post. Get sends data through the URL and should never be used in a secure application. Post sends data through the message header.

To submit a form you'll need a submit button. A button of type Submit sends the data. So here is a submit button, and a button of type Reset to clear the form.

```
<p>
```

```
<input id="btnSubmit" type="submit" value="submit" />
```

```
&nbsp;
```

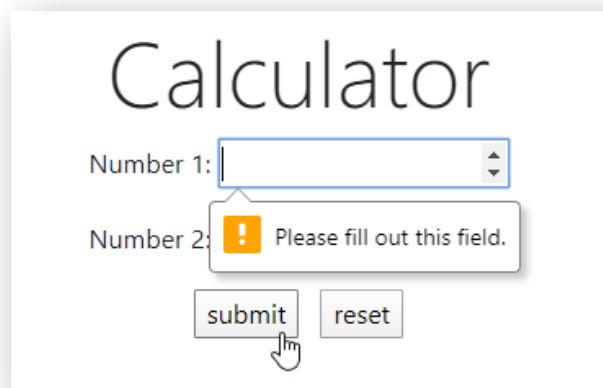
```
<input id="btnReset" type="reset" value="reset" />
```

```
</p>
```

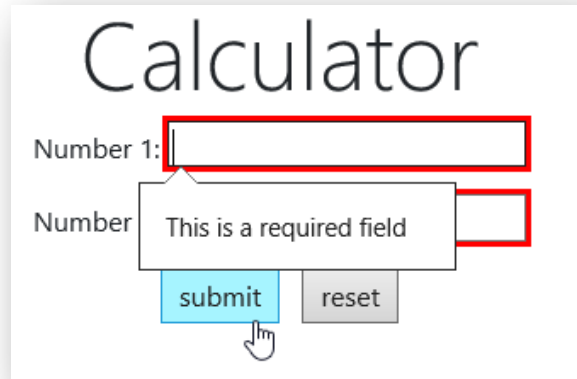
The next thing to point out is the use of the word `required` on all form controls like text boxes, radio buttons, selects (drop downs). When you add the attribute `required` it causes HTML5 validation to happen when the submit button is pressed before the form can be submitted.

```
<input id="tbNumber1" name="number1" type="number" step=".01" required />
```

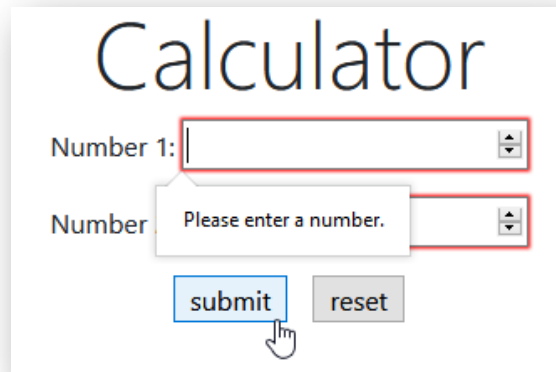
HTML5 validation looks like this in Chrome:



It looks like this in IE:



And like this in Firefox:



<p>

Number 1:

<input id="tbNumber1" name="number1" type="number" step=".01" required />

</p>

<p>

Number 2:

<input id="tbNumber2" name="number2" type="number" step=".01" required />

</p>

These are the text fields I'm using for the calculator. There are different types of text fields including text, password, and number. Number text fields won't let you put characters in them. You can also set their step value which is what I've

done. It will only allow numbers with two decimal places that can step by .01. In other words, I couldn't put 123.456 but I could 123.45.

There's one very important point I need to make here. The **name** attribute is what allows the data to be submitted when the data is valid (passes HTML5 validation) after the submit button is pressed. If you notice in the previous example the **names** for the text fields are "number1" and "number2". These are what I will reference in the **Controller**.

```
<input id="tbNumber1" name="number1" type="number" step=".01" required />
```

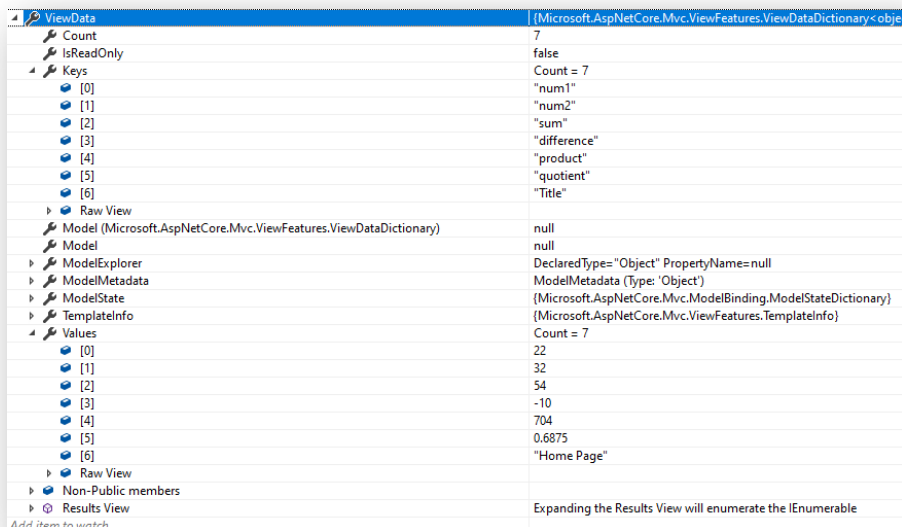
Again, just to reiterate, the **required** attribute is what triggers HTML5 validation on the submission of the form. If validation fails then the form isn't submitted.

One more thing before I move on to the **Controller**. When the data is sent to the View from the **Controller** it contains a ViewData object which contains all information for the **View**.

We can add anything we like to the ViewData object. The ViewData object acts as an associative array where the key is what is in the quotes of the ViewData and the value is what is assigned. On the controller side I can add anything to the ViewData like this.

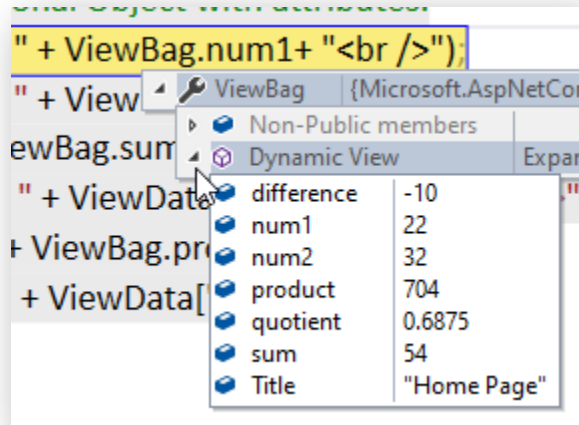
```
ViewData["Hello"] = "Hello World!!!";
```

And on the View side I could access that by either referencing the ViewData like this: @ViewData["HelloWorld"]





Or I can use the ViewBag object to access it like an attribute.



To finish off the View I'll write code to handle the data coming back from the Controller. Here is the data to display the information inside of the div for results. Notice that I use ViewBag and ViewData interchangeably in the **View**. In the **Controller** it's more standard to use just ViewData. The @ symbol references server-side code and Html.Raw displays it to the screen.

```
<div id="divResult">
@{
    if (ViewData["num1"] != null)
    {
        //ViewData[] and ViewBag are the same thing.
        //ViewData is referenced as a key->value set while
        //ViewBag acts more like a traditional Object with attributes.
        @Html.Raw("<b>Number 1:</b> " + ViewBag.num1+ "<br />");
        @Html.Raw("<b>Number 2:</b> " + ViewData["num2"] + "<br />");
        @Html.Raw("<b>Sum:</b> " + ViewBag.sum + "<br />");
        @Html.Raw("<b>Difference:</b> " + ViewData["difference"] + "<br />");
        @Html.Raw("<b>Product:</b> " + ViewBag.product + "<br />");
        @Html.Raw("<b>Quotient:</b> " + ViewData["quotient"] + "<br />");
    }
}
</div>
```

# Coding the Home Controller

The Home **Controller** serves the **View** data. It acts as a conduit between the **Model** and the **View**. By default, you're given a method for each **View** in the **Controller**. In the case of the HomeController.cs which feeds Home/Index.cshtml there is an Index method which initially feeds the **View** first time through. The return type of the Index method is an IActionResult which is the response sent to the **View**. As it says in the Microsoft documentation: [Defines a contract that represents the result of an action method.](#)

Here is the code for that method:

```
21 0 references
22 public IActionResult Index()
23 {
24     return View();
25 }
```

When we connect a form and want to interact with the **Controller** we can write another method also called Index. This method we write to respond to the HTTP Post is set to type **[HttpPost]**. Here is the code for handling the request and returning the response for our calculator:

```
26 [HttpPost]
27 0 references
28 public IActionResult Index(String number1, String number2)
29 {
30     Decimal num1 = Decimal.Parse(number1);
31     Decimal num2 = Decimal.Parse(number2);
32     ViewData["num1"] = num1;
33     ViewData["num2"] = num2;
34     ViewData["sum"] = num1 + num2;
35     ViewData["difference"] = num1 - num2;
36     ViewData["product"] = num1 * num2;
37     ViewData["quotient"] = "Quotient: Cannot divide by zero!!!";
38     if (num2 != 0m)
39     {
40         ViewData["quotient"] = num1 / num2;
41     }
42     return View();
}
```

Notice that the values being passed in to the method are the form fields as noted by the names. Remember that my two number fields were number1 and number2. This is how they are passed into the method. These names **MUST** match the names in the form.

# What about the Model?

**For right now since we aren't working with Databases, we will forego writing the Model. I will post more tutorials on Models in future modules.**

**This should be enough for you to get started in MVC, especially for the first assignment which is write an MVC Magic 8 Ball.**