Calculator: C# and ASP.NET

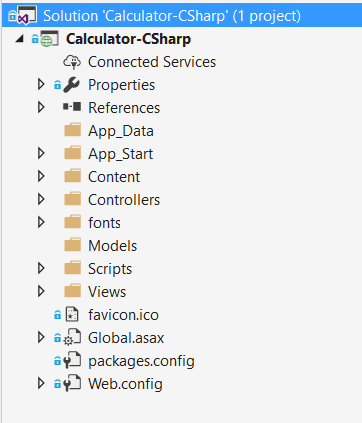
This document defines a complete walkthrough of creating a **Calculator** application with the [ASP.NET](https://www.asp.net/) Framework, from setting up the framework to implementing the fully functional application.

# 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**:

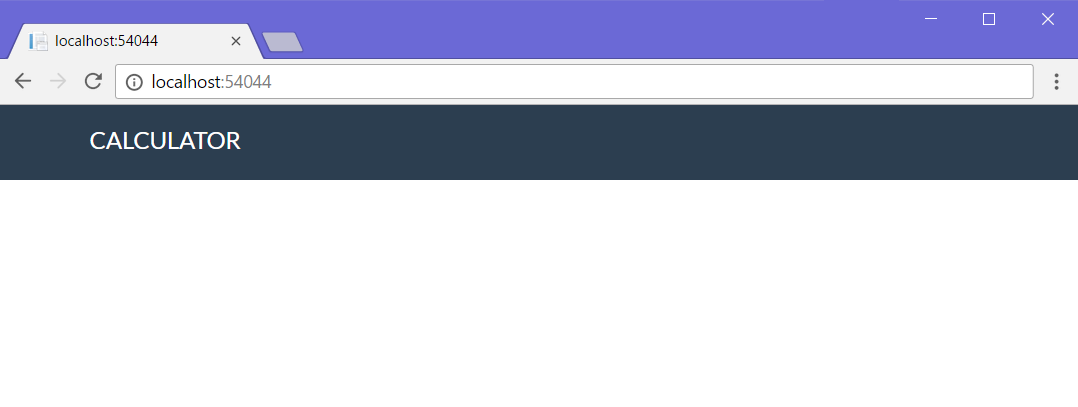


We can see several folders here. Let look at them one by one and see what are they for:

1. **App\_Data** –usually contains the project **database**. We won’t be using this folder for the calculator, as we won’t be needing a database.
2. **App\_Start** –contains various configuration files, such as **RouteConfig.cs** (routes configuration), **BundleConfig.cs** (ASP.NET supports [bundles](https://docs.microsoft.com/en-us/aspnet/mvc/overview/performance/bundling-and-minification), which essentially combine several JS/CSS files into one for better performance) and others.
3. **Content** – everything that is in our static folder (files, images, stylesheets, JavaScript scripts, etc.) will be accessible by every user.
4. **Controllers –** we’ll put all of our controllers here.
5. **fonts –** font storage.
6. **Models** – model classes (we’ll put our Calculator model here).
7. **Scripts** – JavaScript files, which ASP.NET can turn into [minified](https://docs.microsoft.com/en-us/aspnet/mvc/overview/performance/bundling-and-minification#minification) and [bundled](https://docs.microsoft.com/en-us/aspnet/mvc/overview/performance/bundling-and-minification#bundling) versions.
8. **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** mightbe **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

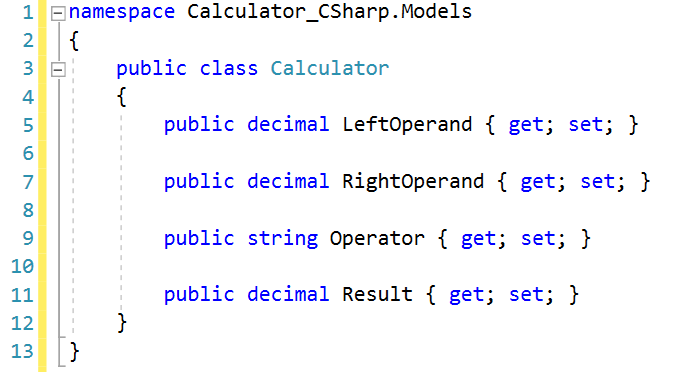
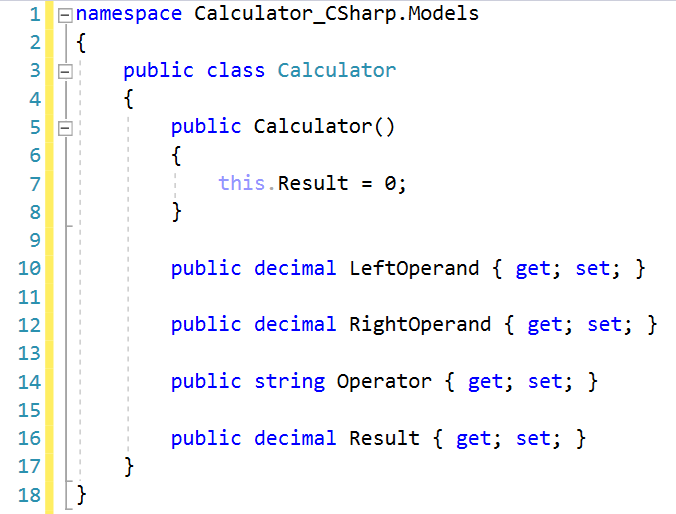
## Create Calculator Model

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]:

|  |  |  |
| --- | --- | --- |
|  | 🡪 | C:\Users\Housey\AppData\Local\Microsoft\Windows\INetCacheContent.Word\cropped.png |

1. **Define** the calculator **properties**:  
   
2. Create a **constructor** for **instantiating** the calculator:  
   

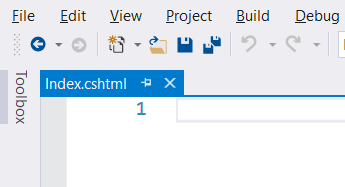
Now all that’s left is to connect it to the rest of our little web application.

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**:

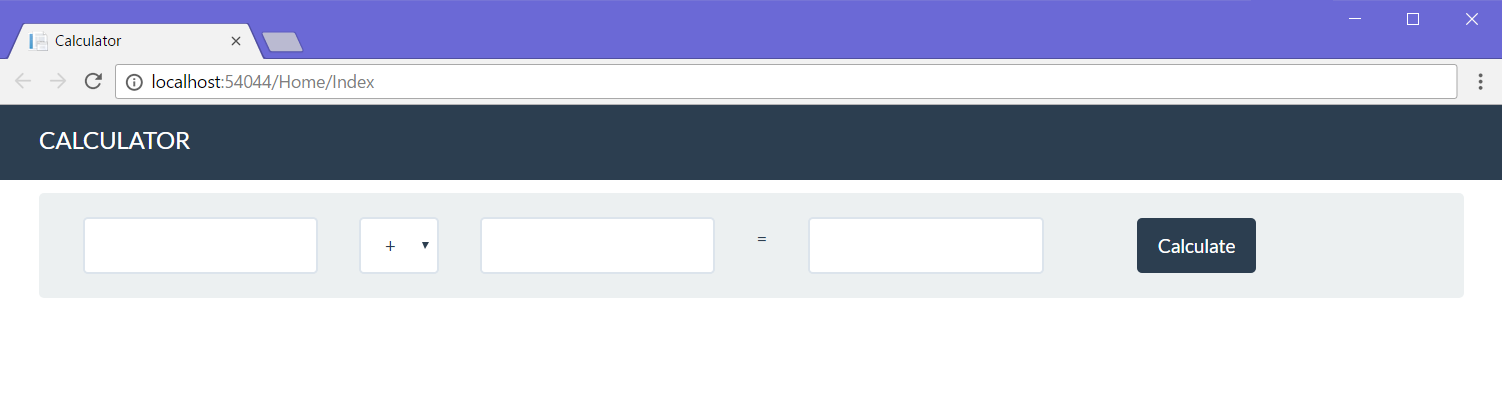
Go into the /Views/Home/ folder and open the Index.cshtml file:



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 Calculator\_CSharp.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> |

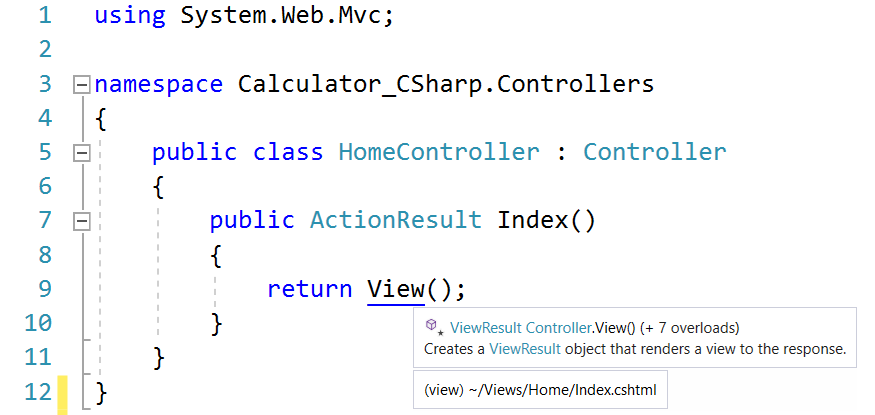
Just like with the Java blog, we will **save the state** of the operands and operator for ease of use, so the **Razor syntax** you see here does just that. The SelectListItem template is a bit more special: it selects the operator from the dropdown list, **based on** the last used operator. We’ll see 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:

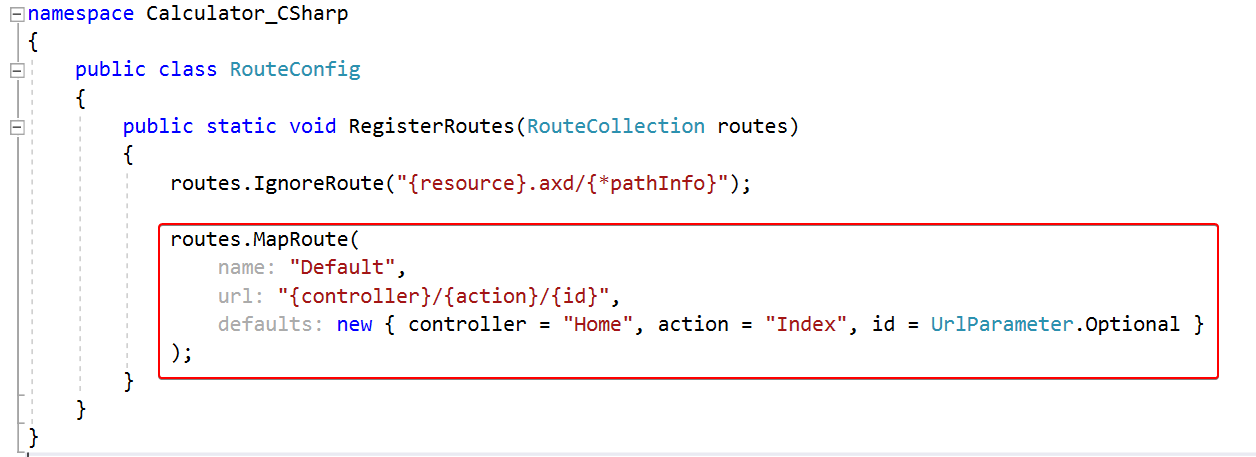


The @RenderBody() line of code expects to be fed a **view** **template** to display around the header and footer. But how does it know **which view** to render? Let’s go into the HomeController.cs file and check out what the **index** action does:



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 App\_Start/RouteConfig.cs class:*

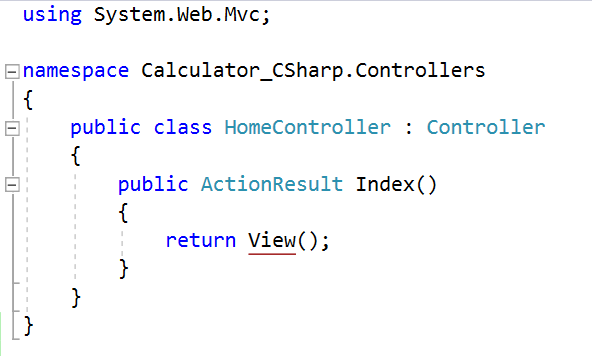


So, for example, if we had to render an **article details** view, we would create a “Details” method inside ArticleControler.cs, and ASP.NET would **automatically** map the /Article/Details/{id} route and also try to find the view, located in the “Views/Article” folder.

## 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:



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

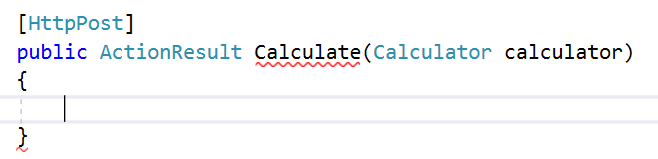
* public ActionResult Index() 🡪 This is the actual **controller action**. It’s a method, which **holds the** **logic**, which will be **executed**, when it’s **called**.
* return View() 🡪 This function **renders** a **view** in the **response** (in essence, takes whatever’s inside of “Views/Shared/\_Layout.cshtml”, 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**.

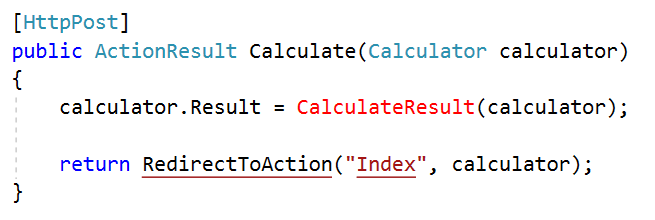
First, we’ll start off by declaring what kind of **HTTP method** this method will be handling (either GET or POST). In our case, since we’re processing **form data**, we’ll add an [HttpPost] **attribute**:



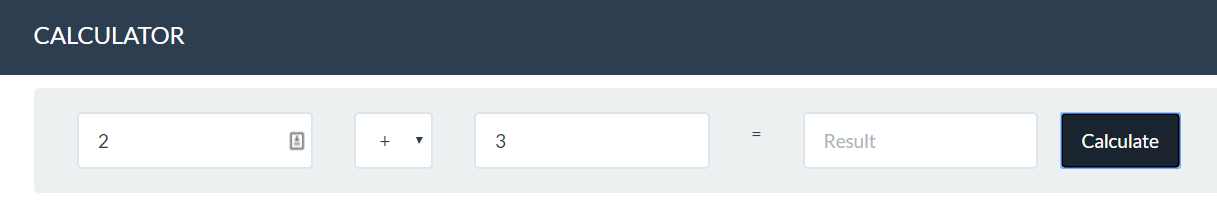
Under it, let’s **declare** our Calculate method. Since the form in the view is defined by a **special Razor form syntax**, we can just pass a **parameter** of the **Calculator** type to the method and it’ll automatically populate it with the form data:

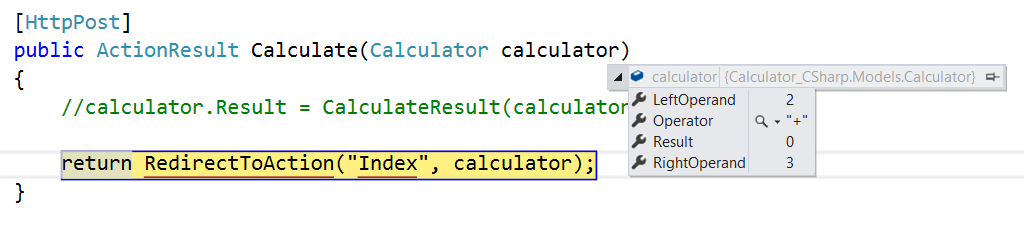


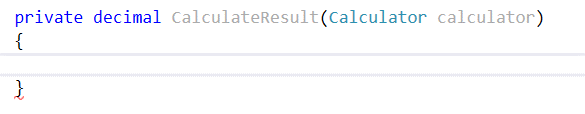
All this method should do at this point is **calculate** the result and return the Index view with all the data (which the view can get from the **calculator object** itself:



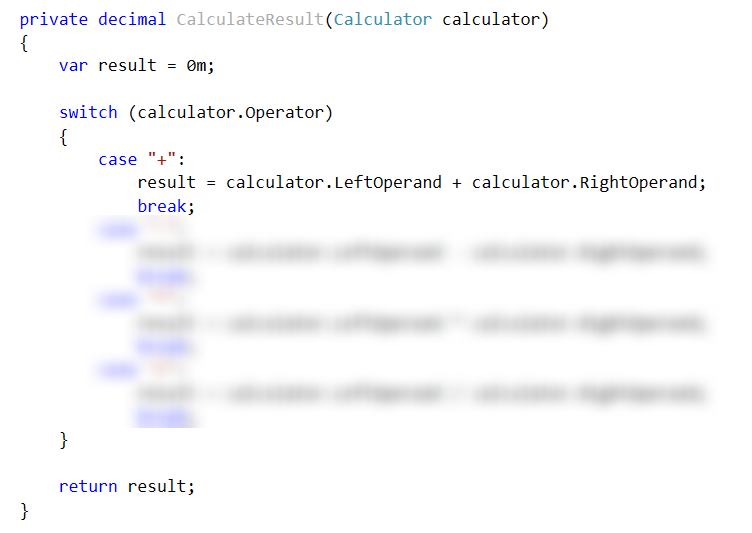
Let’s see what a **debug session** would show us if we were to **debug** this method:



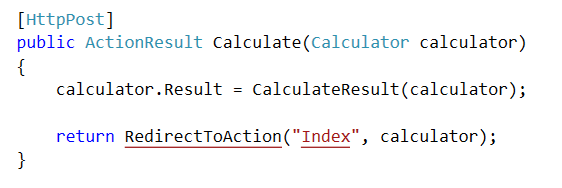
  
The LeftOperand, Operator, and RightOperand variables are automatically **parsed** as **decimal**. All that’s left is to calculate the actual result. Create a CalculateResult method inside the HomeController.cs class:



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

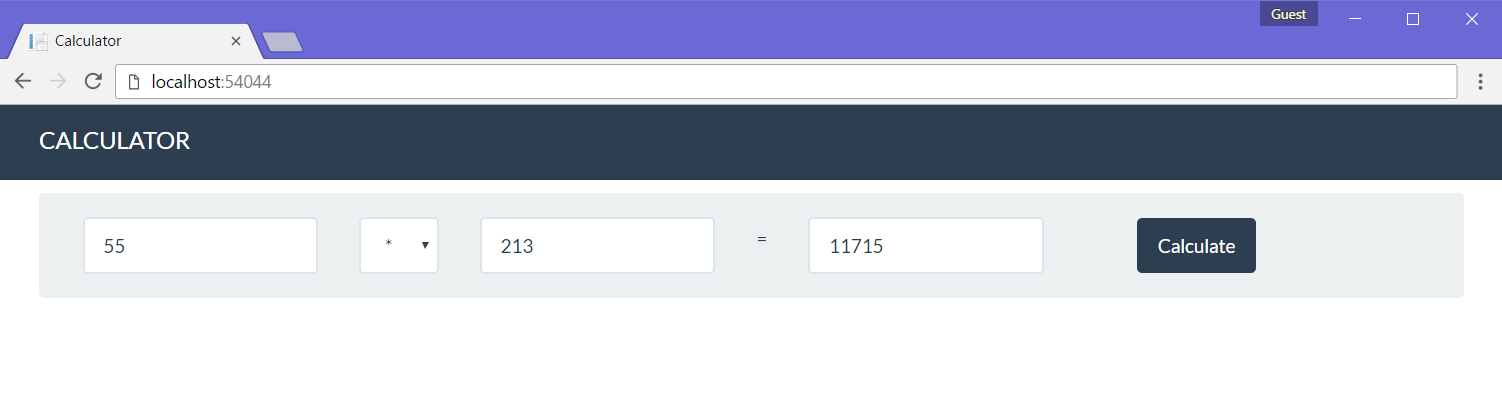


Now that we’ve implemented the controller action, it should look like this:



# 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:



# \* Implement Extra Functionality

Just like last time, you’re free to implement extra functionality like **extra operators**, **input validation**, and whatever else you can think of. Happy coding. ☺

Next time we’ll be using the same logic as in this lab to implement a **fully functioning blog system**, with a **database** behind it for storing everything and even **user authentication** and **security**. Have fun with C# and ASP.NET! ☺