**Exercise: C# and ASP.NET Core**

This document defines several walkthroughs for creating ASP.NET Core MVC-based apps, from setting up the framework to implementing the fully functional applications.

# I. Non-Data-Driven Apps

These are apps, which do not need a database to work.

## 1. Calculator

This document defines a complete walkthrough of creating a **Calculator** application with the ASP.NET Core Framework, from setting up the framework to implementing the fully functional application.

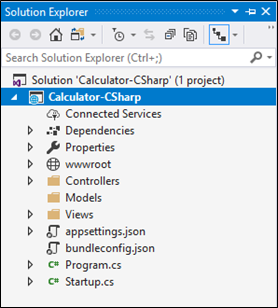
### 1. Base Project Overview

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

#### Open the Project

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

Този документ дефинира няколко стъпки за създаване на ASP.NET Core MVC-базирани приложения, от създаването на рамката за внедряване на напълно функционални приложения.  
I. Приложения без данни  
Това са приложения, които не се нуждаят от работа с база данни.  
1. калкулатор  
Този документ дефинира пълната стъпка в създаването на приложение за калкулатори с ASP.NET Core Framework, от създаването на рамката до внедряването на напълно функционалното приложение.  
1. Основен преглед на проекта  
Проектът ни ще бъде изграден, използвайки езика C # и MVC framework ASP.NET Core. Ще използваме Razor View Engine, за да дефинираме нашите изгледи.  
Отворете проекта  
Нека да разгледаме структурата на проекта:



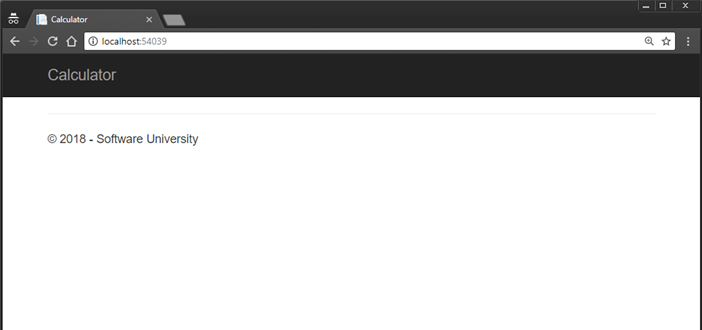
We can see several folders here. Let look at them one by one and see what are they for:

1. **wwwroot** –contains JavaScript files, css files and images.
2. **Controllers –** we’ll put all of our controllers here.
3. **Models** – model classes (we’ll put our Calculator model here).
4. **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):

Тук можем да видим няколко папки. Нека да ги разгледаме един по един и да видим за какво са:  
1. wwwroot - съдържа JavaScript файлове, CSS файлове и изображения.  
2. Контрольори - ще сложим всички контролери тук.  
3. Модели - класове модели (тук ще поставим нашия модел на калкулатор).  
4. Изгледи - тук ще запазим шаблоните за преглед. Ще използваме шаблонния двигател Razor.  
Изпълнете проекта  
След като отворихме проекта, нека се опитаме да го изпълним, за да можем да видим с какво работим. Натиснете [Ctrl + F5], за да компилирате проекта и да стартирате сървъра. Страницата автоматично ще се отвори в браузъра ви по подразбиране (забележка: портът може да е различен от екранната снимка):



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

### 2. Implement Functionality

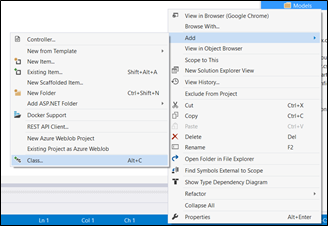
#### Create Calculator Model

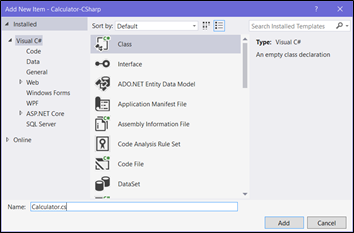
It’s time to design our main model – the **Calculator**. It will contain the following properties:

* LeftOperand
* RightOperand
* Operator
* Result

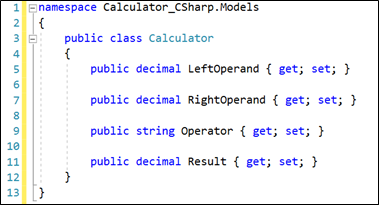
Let’s create our model. Since we’re **not** using a database in this exercise, we’re just going to define the calculator as a **simple C# class** (the only difference between C# classes and Entity Framework models is that EF models might have attributes, which help it name database columns and set restrictions).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

Тя не изглежда много, но поне имаме основното оформление надолу! Нека да се заемем с изпълнението на някои функции!  
2. Внедрете функционалност  
Създайте модел на калкулатор  
Време е да проектираме нашия основен модел - Калкулаторът. Той ще съдържа следните свойства:  
• LeftOperand  
• RightOperand  
• Оператор  
• Резултат  
Нека да създадем нашия модел. Тъй като не използваме база данни в това упражнение, ние просто ще определим калкулатора като прост клас C # (единствената разлика между C # класовете и Entity Framework модели е, че EF моделите могат да имат атрибути, които да му помагат да назове база данни колони и определени ограничения). Отидете в папката Модели и създайте нов клас C #, наречен "Calculator.cs", като използвате [Кликнете с десен бутон  Добави → Клас]:

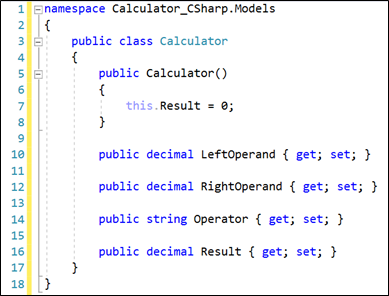
 ->



**1.Define** the calculator **properties**: Дефинирайте свойствата на калкулатора:



2. Create a **constructor** for **instantiating** the calculator: Създайте конструктор за инстанцииране на калкулатора:



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

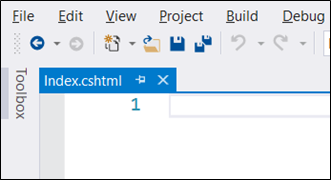
For our final trick, we’ll create our own controller action, which will **process** what the user sent us and **return** a **view** with the **result** from the calculation.

#### 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:

Сега остава само да го свържете с останалата част от нашето малко уеб приложение.  
За последния ни трик ще създадем свой собствен контролер, който ще обработва това, което потребителят ни е изпратил и ще върне изгледа с резултата от изчислението.  
Създаване на изглед Калкулатор  
Преди да можем да имаме някаква функционалност, би било хубаво да имаме представа за това, срещу което работим, така че нека продължим и да създадем формуляр, който потребителят ще използва за изчисления:  
Влезте в папката / Views / Home / и отворете файла Index.cshtml:



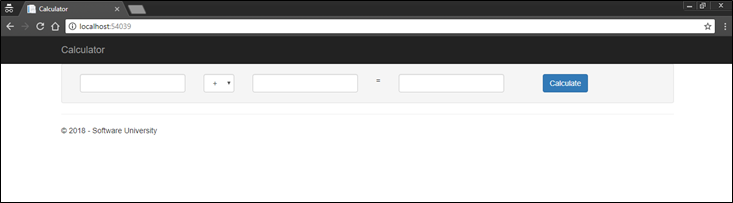
It’s empty?! How does the header and footer seen above get displayed then? The answer is, we use a global **layout** file (/Views/Shared/\_Layout.cshtml), so we don’t have to copy-paste our page layout into every single view in our project (which could have tens or hundreds of views). All the **actual 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:

Празно е?! Как се виждат горните и долните колони, показани по-горе? Отговорът е, че използваме глобален файл с оформление (/Views/Shared/\_Layout.cshtml), така че не е нужно да копирате и поставяме оформлението на страницата ни във всеки един изглед в нашия проект (който може да има десетки или стотици показвания ). Целият HTML дизайн на основната база е в \_Layout.cshtml. Няма да се докосваме до това, така че нека да отидем до файла Index.cshtml и да добавим нашия формуляр:

|  |
| --- |
| @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> |

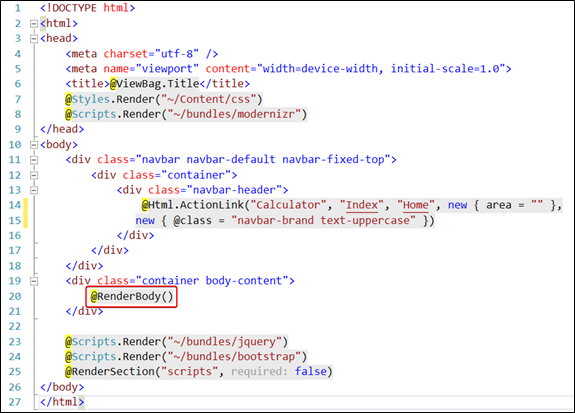
Just like with the Java blog, we will **save the state** of the operands and operator for ease of use, so the **Razor syntax** you see here does just that. The SelectListItem template is a bit more special: it selects the operator from the dropdown list, **based on** the last used operator. We’ll see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

Точно както при блога Java, ще запазим състоянието на операндите и оператора за лесна употреба, така че синтаксисът Razor, който виждате тук, прави точно това. Шаблонът за SelectListItem е малко по-специален: избира оператора от падащия списък въз основа на последния използван оператор. Ще видим как това се изпълнява малко по-късно. Засега нека да разгледаме нашето уеб приложение и да видим как се справяме (не забравяйте да прекомпилирате проекта предварително, като използвате [Ctrl + Shift + B]:



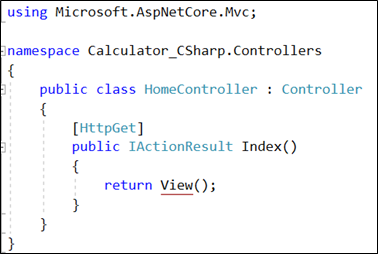
Let’s see how this all ties together. Go into /Views/Shared/\_Layout.cshtml:

Нека видим как всичко това се свързва заедно. Отидете в /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:

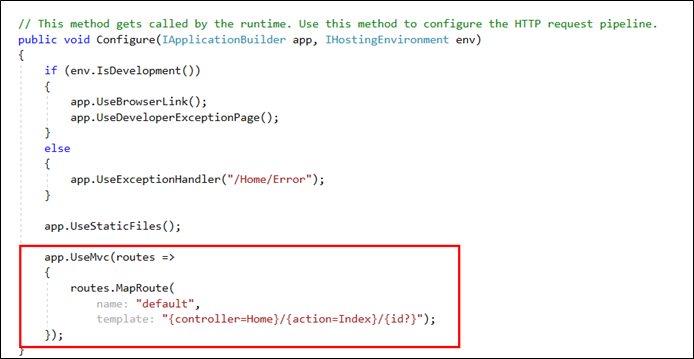
Кодът на кода на @RenderBody () очаква да бъде подаден шаблон за преглед, който да се покаже около заглавката и долния колонтитул. Но откъде знае кой изглед да се направи? Нека да отидем във файла HomeController.cs и да проверим какво прави индексното действие:



As you can see, the Index action in HomeController.cs returns the Index.cshtml view inside the Views/Home folder. **ASP.NET Core** 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:*

Както виждате, действието на индекса в HomeController.cs връща изгледа Index.cshtml в папката Views / Home. ASP.NET Core е достатъчно интелигентен, за да разбере кой изглед да се върне, въз основа на контролера, който е вътре и името на метода (и генериране на маршрути автоматично).  
Всъщност не е толкова магическо, колкото си мислите - всичко това е определено в класа Startup.cs:

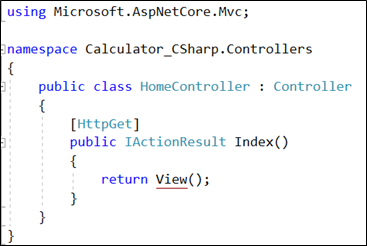
So, for example, if we had to render an **article details** view, we would create a “Details” method inside ArticleControler.cs, and ASP.NET Core would **automatically** map the /Article/Details/{id} route and also try to find the view, located in the “Views/Article” folder.

#### 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:

Така например, ако трябва да прегледаме подробности за статия, ще създадем метод "Детайли" в ArticleControler.cs и ASP.NET Core автоматично ще премести маршрута / Article / Details / {id} и ще опита да намерете изгледа, който се намира в папката "Изгледи / член".  
Изпълнете действието на контролера  
Сега, след като създадохме изгледа, който ще държи данните ни и ще позволи на потребителя да взаимодейства с нашето уеб приложение, е време да приложим движещата сила зад цялото приложение - действието на контролера.  
Както се оказа, вече имаме инсталиран домашен контролер и действие, настроено на маршрута "/", в противен случай дори няма да можем да видим нашия калкулатор. Главният контролер можете да намерите в папката "Контролери". Да видим как изглежда:



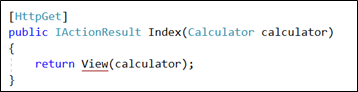
Not much going on here… Let’s break it down:

* public ActionResult Index() à This is the actual **controller action**. It’s a method, which **holds the** **logic**, which will be **executed**, when it’s **called**.
* return View() à 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**:

Не се случва много тук ... Нека да го счупим:  
• public ActionResult Index ()  Това е действителното действие на контролера. Това е метод, който държи логиката, която ще бъде изпълнена, когато се нарича.  
• връщане на изгледа ()  Тази функция прави изглед в отговора (по същество отнема всичко, което е вътре в "Views / Shared / \_Layout.cshtml", изпраща всичко, което е вътре в "Views / Home / Index.cshtml" Razor шаблонира двигателя и го връща на потребителя.  
Така че, използвайки нововъведеното знание, нека се опитаме да създадем свои собствени действия.  
Първо, трябва да променим действието на индекса, за да върнем пример на нашия модел Калкулатор. Ще го направим по този начин, така че можем да пренасочим към това действие, за да покажем резултата, когато го изчисляваме. Ще отидем в действието на индекса и ще променим подписа на метода и връщащата стойност:



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

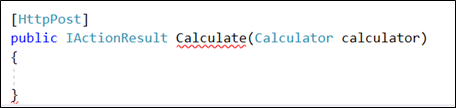
First, let’s start off by declaring what kind of **HTTP method** this method will be handling (either GET or POST). In our case, since we’re processing **form data**, we’ll add an [HttpPost] **attribute**:

След като променим действието на индекса, е време да създадем действието, което ще изчисли резултата.  
Първо, нека започнем, като обявим какви HTTP методи ще обработва този метод (GET или POST). В нашия случай, тъй като обработваме данните от формуляра, ще добавим атрибут [HttpPost]:



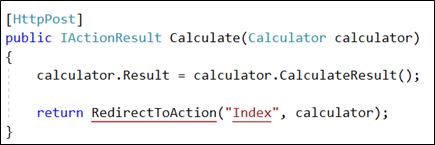
Under it, let’s **declare** our Calculate method. Since the form in the view is defined by a **special Razor form syntax**, we can just pass a **parameter** of the **Calculator** type to the method and it’ll automatically populate it with the form data:

Под нея, нека декларираме нашия метод Изчисляване. Тъй като формулярът в изгледа се определя от специален синтаксис на формата на Razor, можем просто да преминем към метода параметър от типа "Калкулатор" и автоматично да го запълни с данните за формуляра:



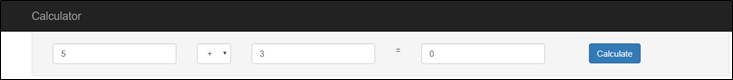
All this method should do at this point is **calculate** the result and return the Index view with all the data (which the view can get from the **calculator object** itself:

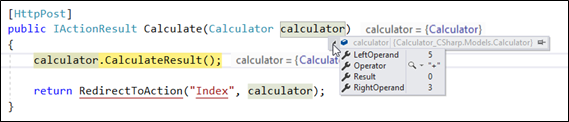
Целият този метод трябва да се направи в този момент, се изчислява резултата и се връща индексният изглед с всички данни (които изгледът може да получи от самия калкулатор обект:



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

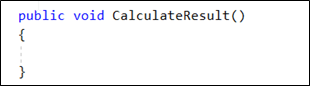
Нека видим какво ще ни покаже сесия за отстраняване на грешки, ако трябва да отстраняваме този метод:





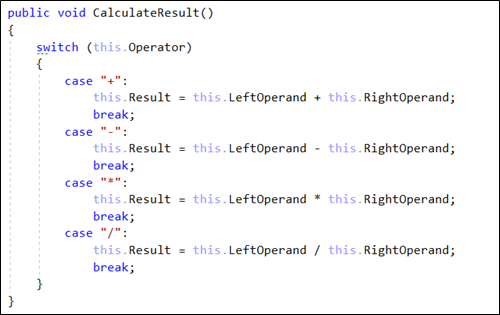
The LeftOperand, Operator, and RightOperand variables are automatically **parsed** as **decimal**. All that’s left is to calculate the actual result. Create a CalculateResult method inside the HomeController.cs class:

Променливите LeftOperand, Operator и RightOperand се анализират автоматично като десетични. Всичко останало е да се изчисли действителният резултат. Създайте метод CalculateResult в клас model Controller.cs:



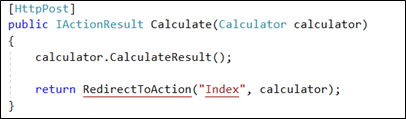
All that’s left is to implement the calculation logic:

Всичко останало е да приложим логиката на изчисление:



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

След като въведем контролера, трябва да изглежда така:



### 3. 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:

3. Проверете приложението  
Цялата ни упорита работа трябва най-накрая да се изплати, нали? Ако сте следвали всички стъпки правилно и сте прочели целия обяснителен текст, надяваме се да имаме функциониращ калкулатор! Възстановете приложението, като използвате [Ctrl + Shift + B] и го тествайте:

