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.

# Non-Data-Driven Apps

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

## Calculator

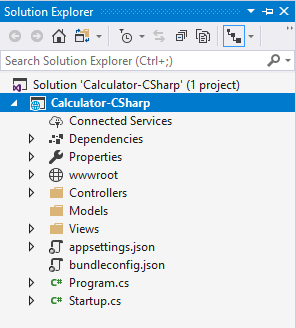
This document defines a complete walkthrough of creating a **Calculator** application with the ASP.NET Core Framework, from setting up the framework to implementing the fully functional application.

### 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**:

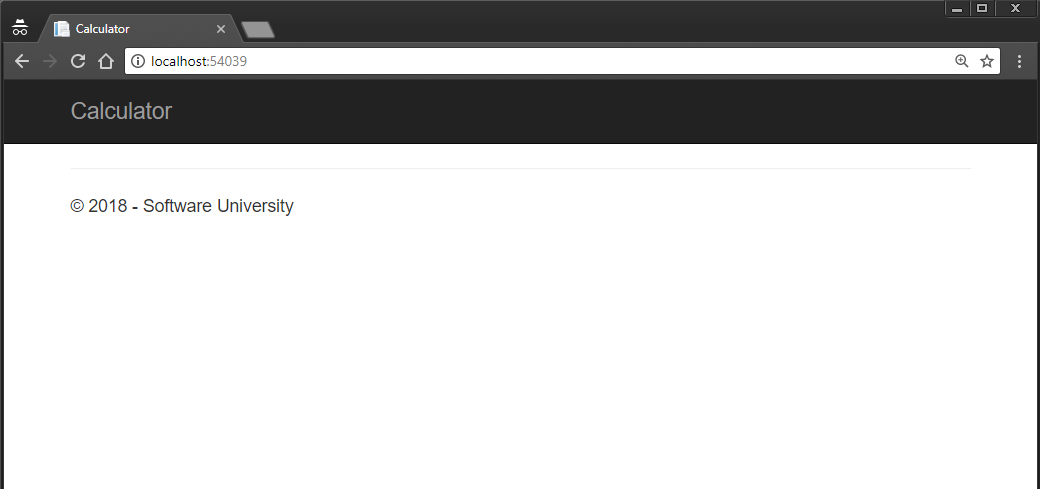


We can see several folders here. Let look at them one by one and see what are they for:

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):

  
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

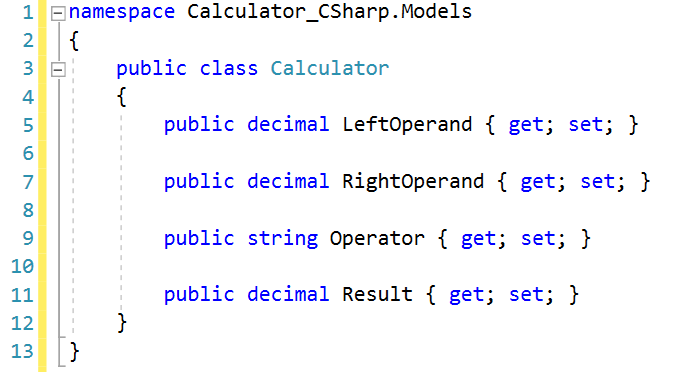
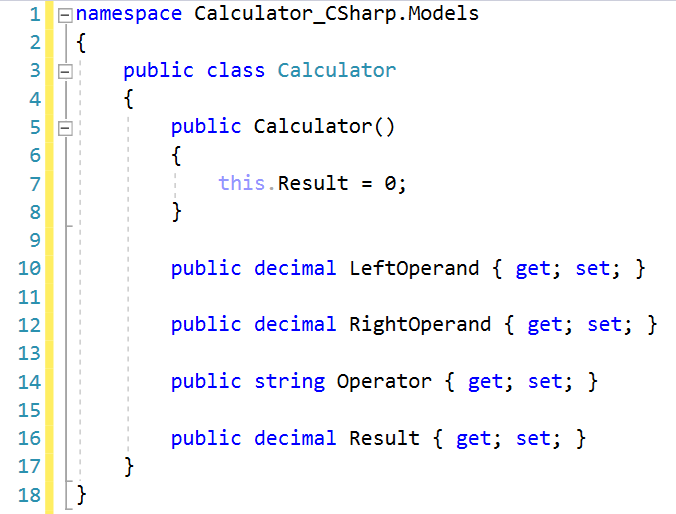
#### Create Calculator Model

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:  
   

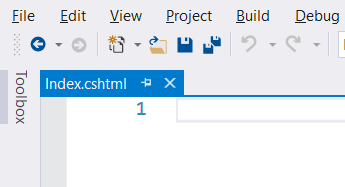
Now all that’s left is to connect it to the rest of our little web application.

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**:

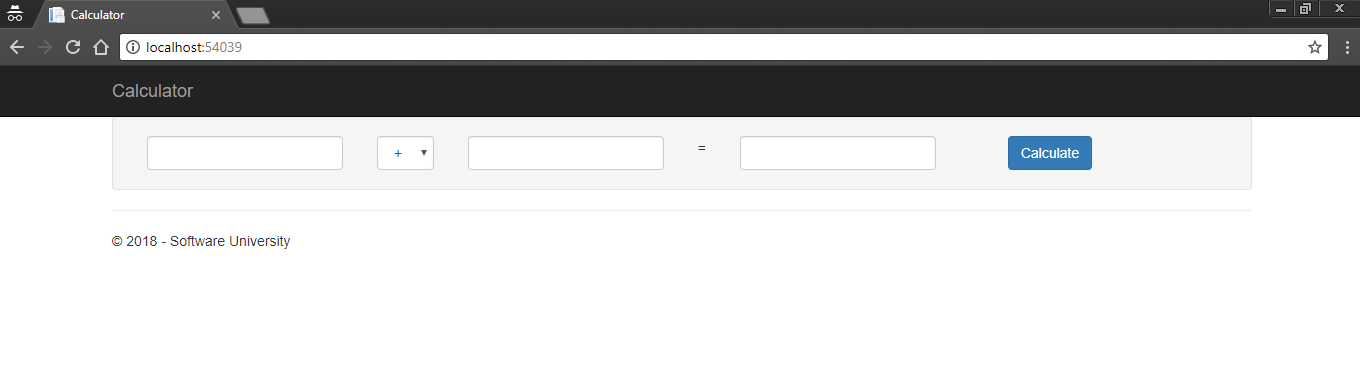
Go into the /Views/Home/ folder and open the Index.cshtml file:



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> |

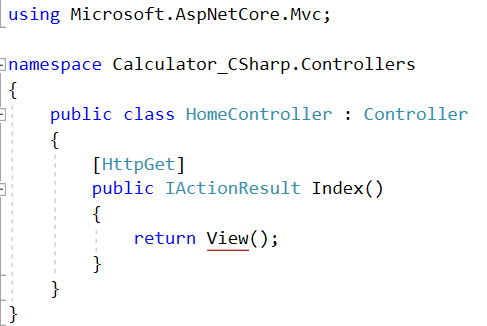
Just like with the Java blog, we will **save the state** of the operands and operator for ease of use, so the **Razor syntax** you see here does just that. The SelectListItem template is a bit more special: it selects the operator from the dropdown list, **based on** the last used operator. We’ll see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

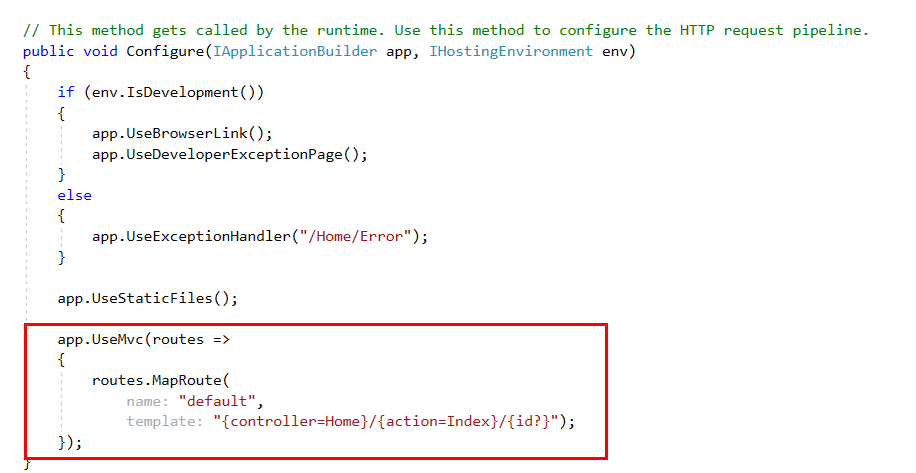


The @RenderBody() line of code expects to be fed a **view** **template** to display around the header and footer. But how does it know **which view** to render? Let’s go into the HomeController.cs file and check out what the **index** action does:



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

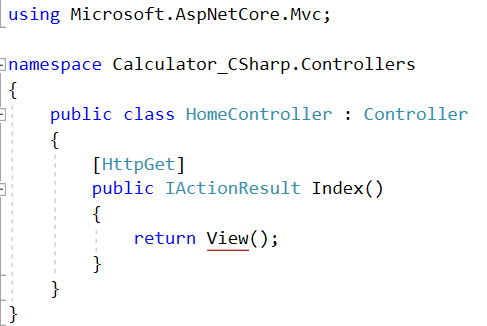


So, for example, if we had to render an **article details** view, we would create a “Details” method inside ArticleControler.cs, and ASP.NET Core would **automatically** map the /Article/Details/{id} route and also try to find the view, located in the “Views/Article” folder.

#### 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:

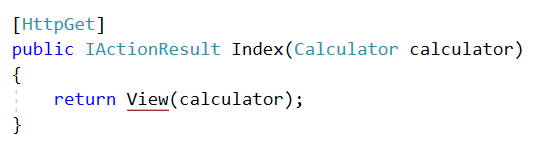


Not much going on here… Let’s break it down:

* 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**:

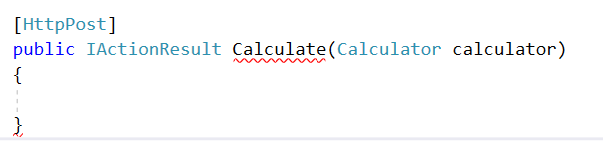


Now that we’ve modified the index action, it’s time to create the action, which will **calculate the result**.

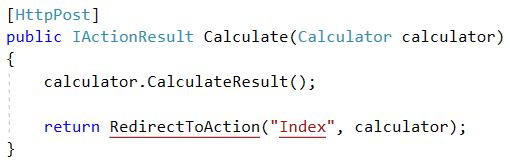
First, let’s start off by declaring what kind of **HTTP method** this method will be handling (either GET or POST). In our case, since we’re processing **form data**, we’ll add an [HttpPost] **attribute**:



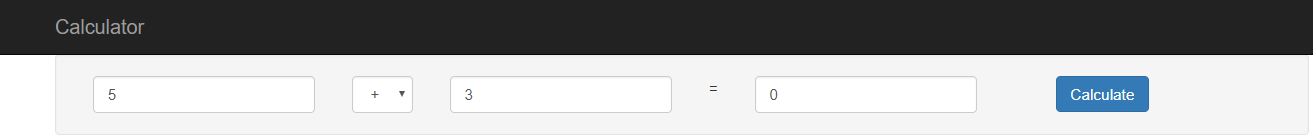
Under it, let’s **declare** our Calculate method. Since the form in the view is defined by a **special Razor form syntax**, we can just pass a **parameter** of the **Calculator** type to the method and it’ll automatically populate it with the form data:

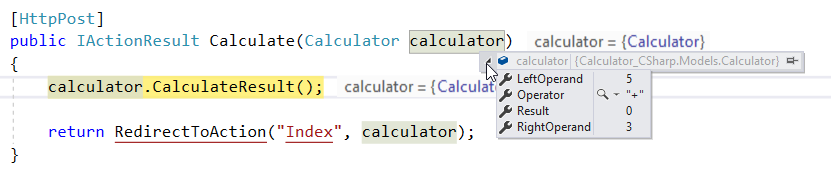


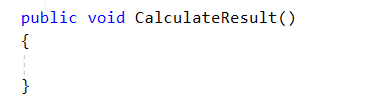
All this method should do at this point is **calculate** the result and return the Index view with all the data (which the view can get from the **calculator object** itself:



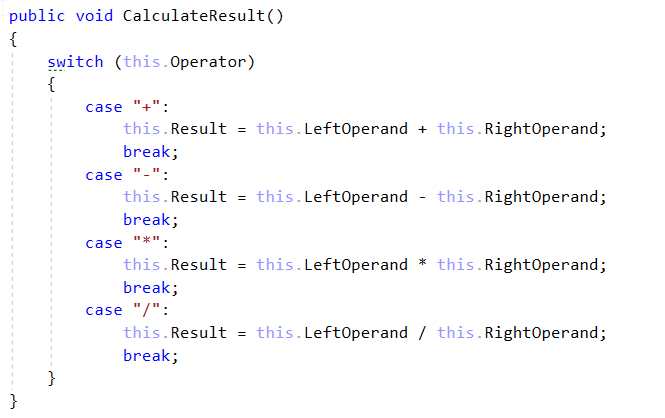
Let’s see what a **debug session** would show us if we were to **debug** this method:



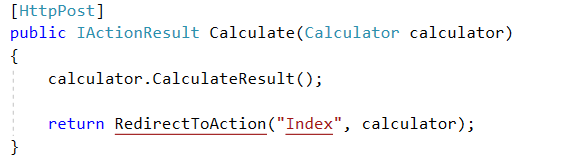
  
The LeftOperand, Operator, and RightOperand variables are automatically **parsed** as **decimal**. All that’s left is to calculate the actual result. Create a CalculateResult method inside the Calculator.cs class:



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

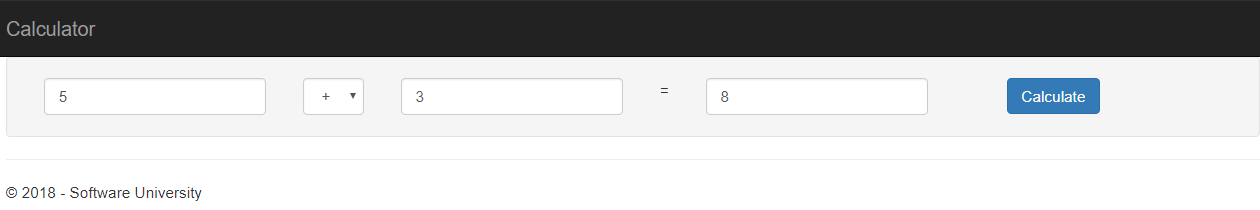


Now that we’ve implemented the controller action, it should look like this:



### 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:



# Data-Driven Apps \*

These are apps, which need to store data in a database to work properly.

## Book Library \*

The Book Library we’re going to create is a lot like the TODO List, with two main differences:

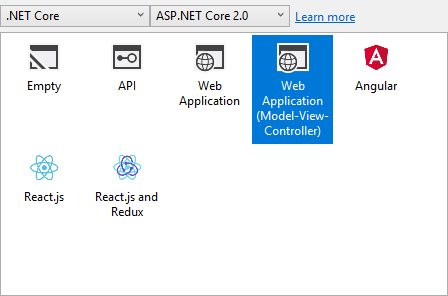
* We have users and use authorization for actions
* We can **edit books** (change title, etc...). Last time we could only create and delete them.

If you successfully complete all steps you will have a “**Book Library**” application with the following functionality:

* **Register User**
* **Login User**
* **Create New Book**
* **Edit Existing Book**
* **Delete Existing Book**
* **List All Books**

### Create a New Project

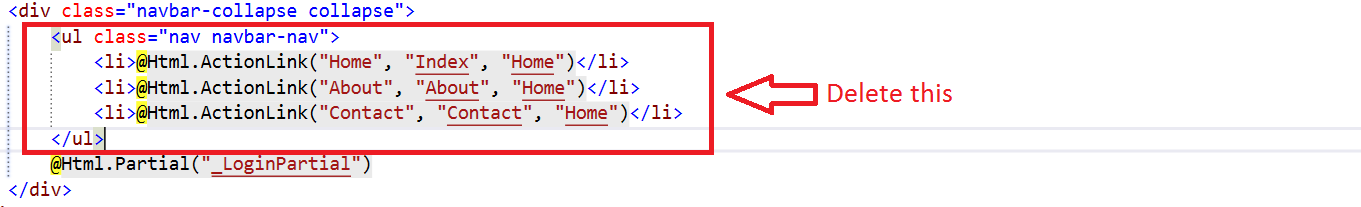
Creating the project is similar to the “**TODO List**” problem. This time leave the authentication to “**Individual User Authentication**”:



**Important: Don’t run the project before step 9.**

### Edit the Main Layout

Let’s start by deleting the unnecessary lines from „\_Layout.cshtml“.



### Edit the Index view

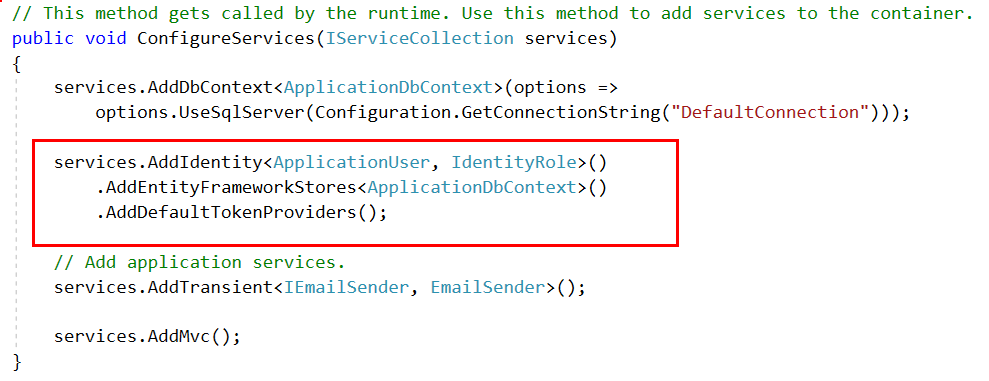
Go to **Views/Home/Index.cshtml** and replace the code with:

|  |
| --- |
| @{  ViewData["Title"] = "Home Page";  } |

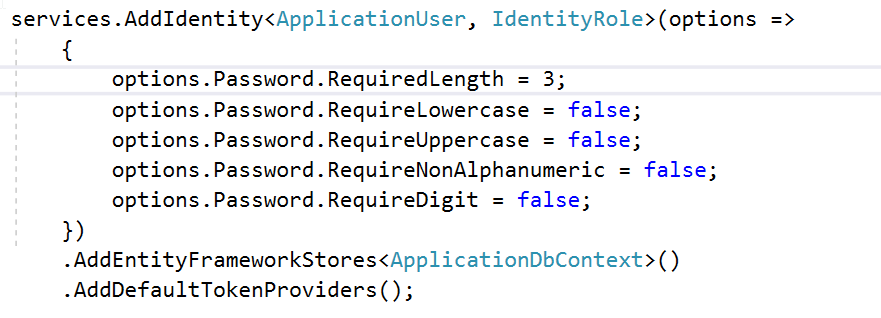
We are done with the layout for now.

### Simplify the Registration

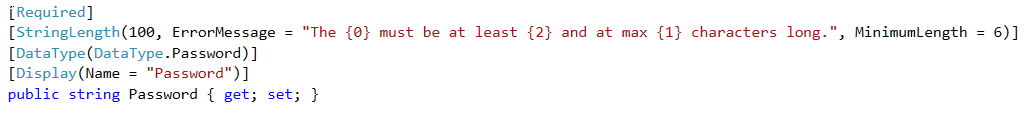
Go to your “Startup.cs” file. Find the following lines:

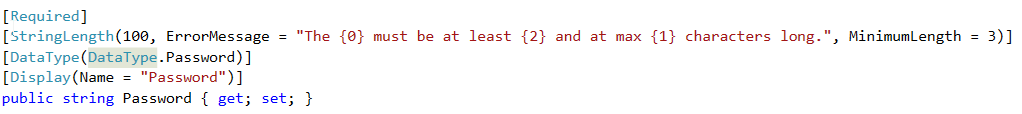


Edit them to receive this result:



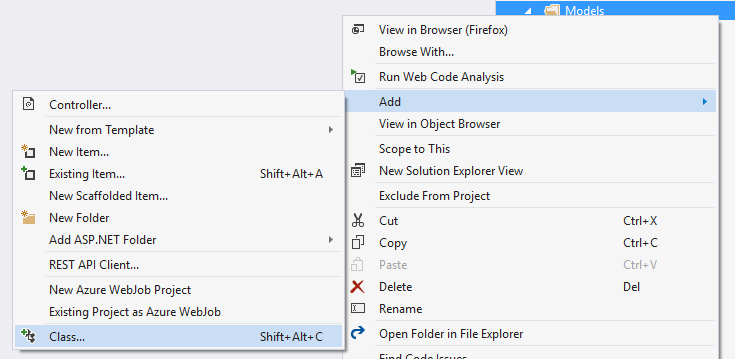
This will simplify our password. Now we need to edit the models that validate our data. Go to the “Models/AccountViewModel/RegisterViewModel.cs”. Find this line:

Edit the minimum length to 3 symbols.

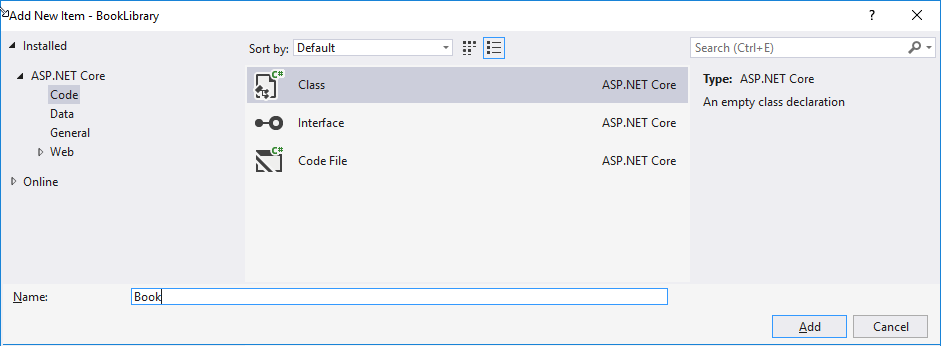


### Create the Book Model

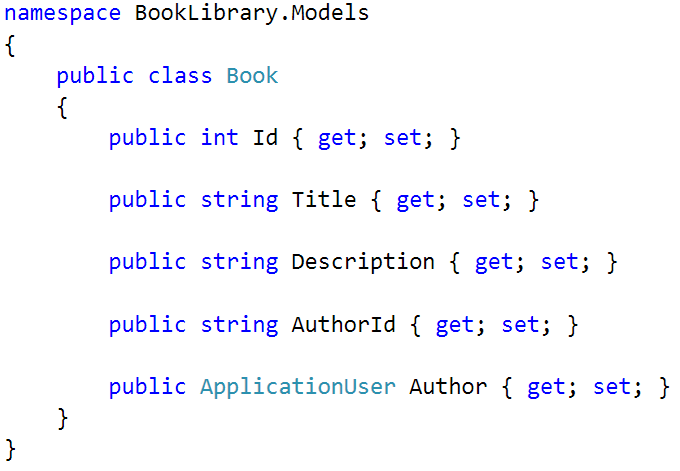
Create a new class in your **Models** folder:



Name the class Book.cs:

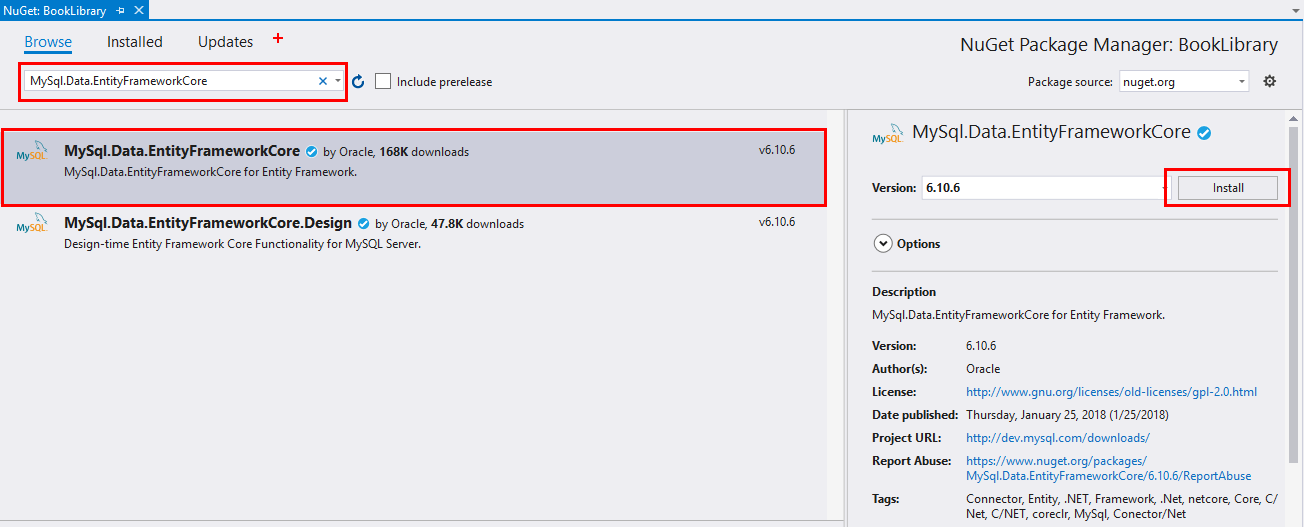


Add the following properties:



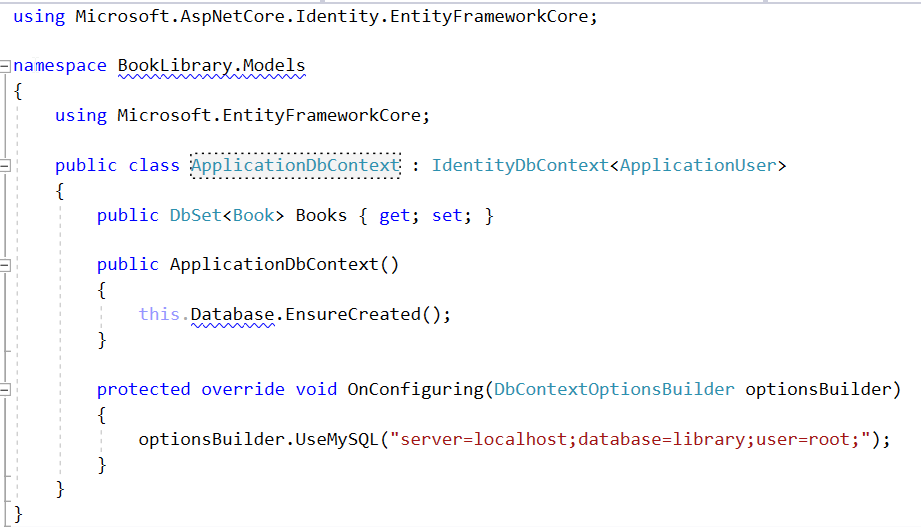
### Add MySQL Data Entity Framework Core

We need to **add MySql Data Entity Framework Core** package. Right click on the **Dependencies** and click on "**Manage NuGet Packages**". Click "Browse", find **MySql.Data.EntityFrameworkCore** and install the package.



### Edit the Database Context

Go to **Data** folder and find the **ApplicationDbContext.cs**. We need to edit our database context to work properly with MySQL. Your ApplicationDbContext.cs should look like this:



A **DbSet** works a lot like a C# List<T>. List<T> accesses items in RAM, whereas DBSet<T> accesses items in a database. It’s a little complicated than that, but that’s essentially what it does on the surface. It also supports several list-like methods, such as Add(), Remove() and so on...

**EnsusreCreated** method ensures that the database for the context exists. If it exists, no action is taken. If it does not exist then the database and all its schema are created. If the database exists, then no effort is made to ensure it is compatible with the model for this context.

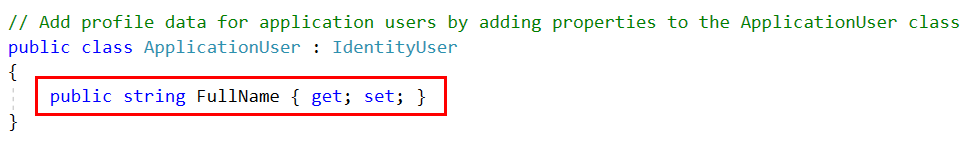
We override **OnConfiguring** method to connect our application with **MySQL**.

### Add DbContext to the services

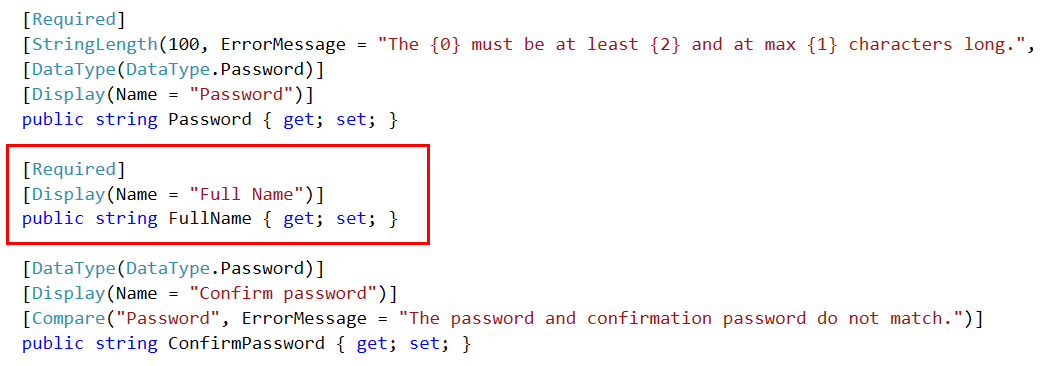
We need to add our DbContext in our services. Go to Startup.cs and in method ConfigureServices() add the DbContext:

### Add Full Name to Our User

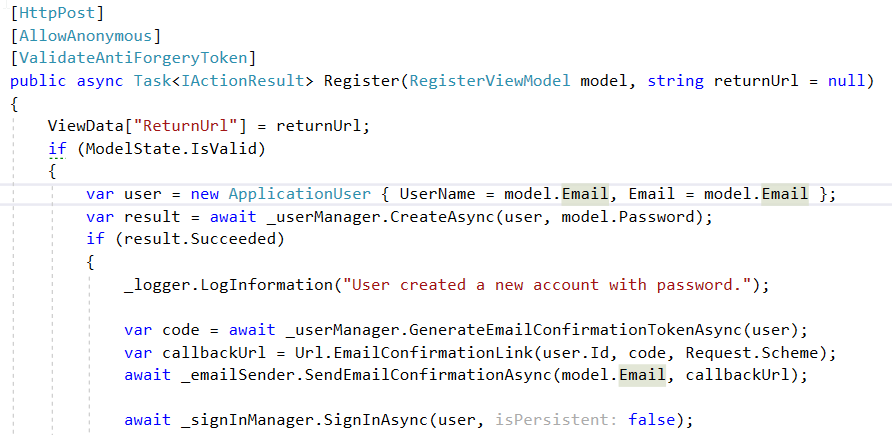
Go to your **ApplicationUser.cs** file and write the following line:



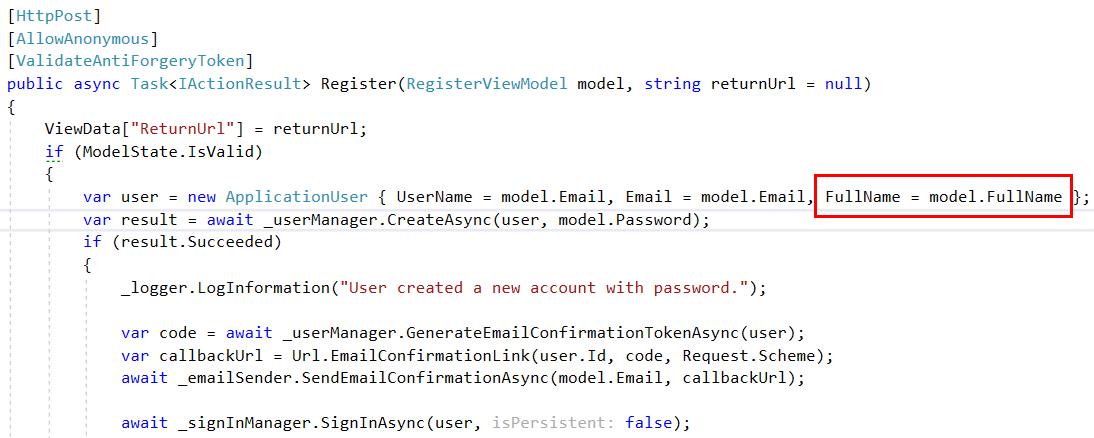
Now go to **Models/AccountViewModel** folder and find **RegisterViewModel.cs**. Add the following lines:



We are almost done. Go to your **AccountController.cs**. Find the Register post method:



Add this, to the existing line:

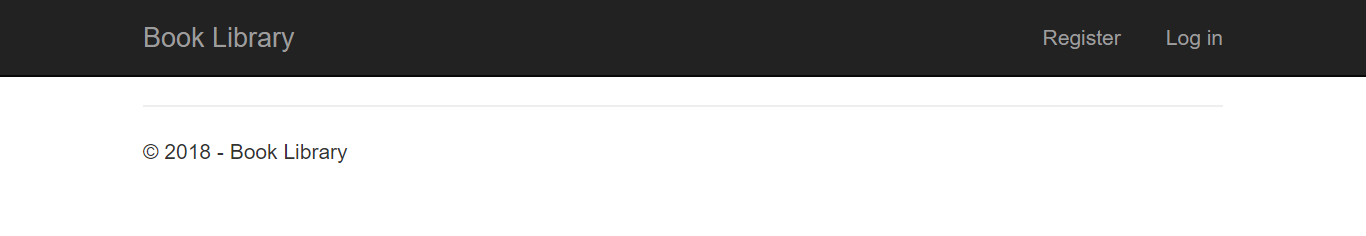


Finally go to the register view located in "**Views/Account/Register.cshtml**". Write the following code:



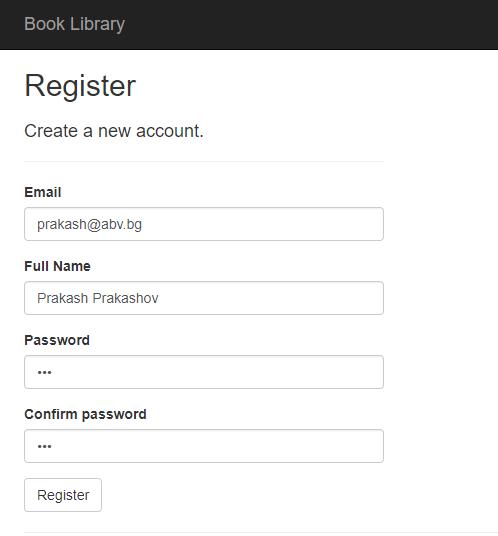
### Run the Application for the First Time

You should see this:



### Register New User

Run the application and try to register a new user:

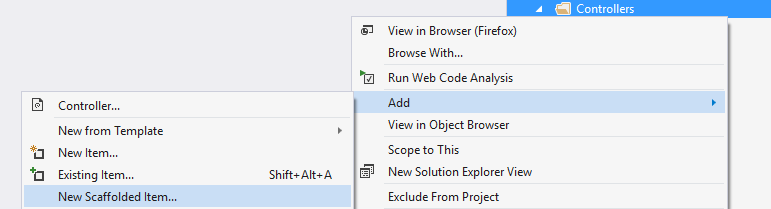


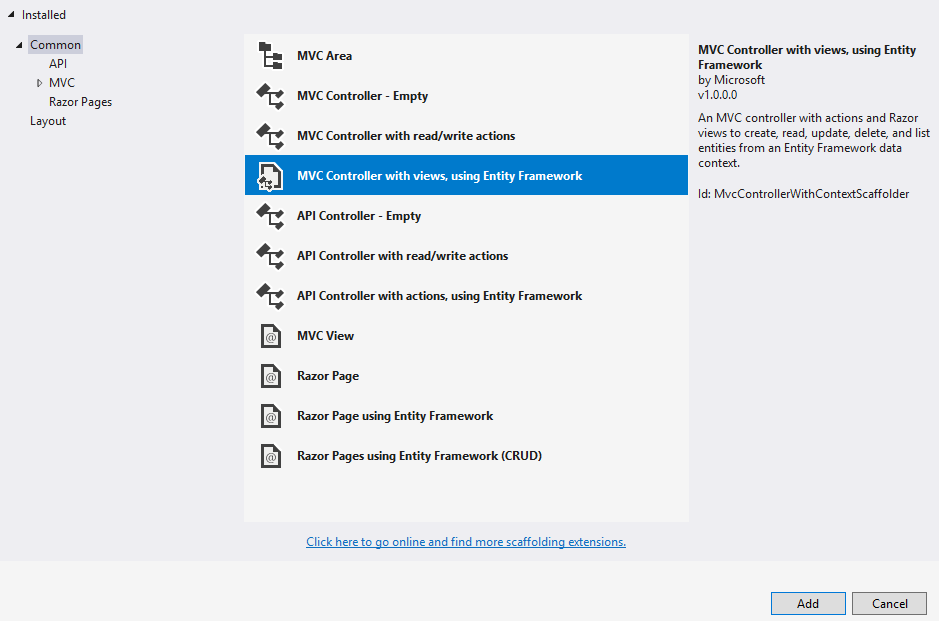
If it works you will be logged in, and you will see welcome message like this:



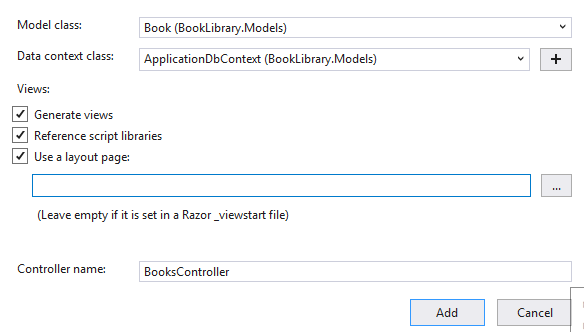
### Create the Book Controller

Add a **new scaffolded item** to your **Controllers** folder:





Use your Book **Model** and your ApplicationDbContext as shown in the image below:



### Edit the User Layout

Go to the \_LoginPartial.cshtml layout and add the following code:



Now if you log in, you will see this button:



When you click on it, you will see this page:



**Don’t create new books yet!**

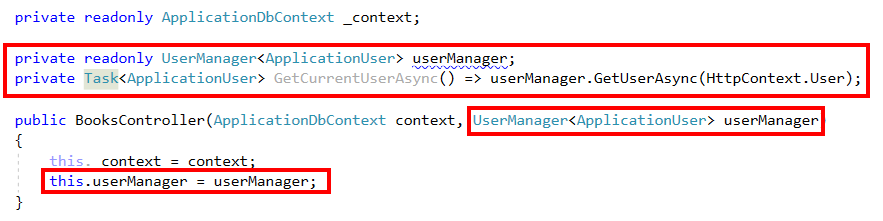
### Edit the Create View

Go to **Views/Books** folder and open **Create.cshtml**

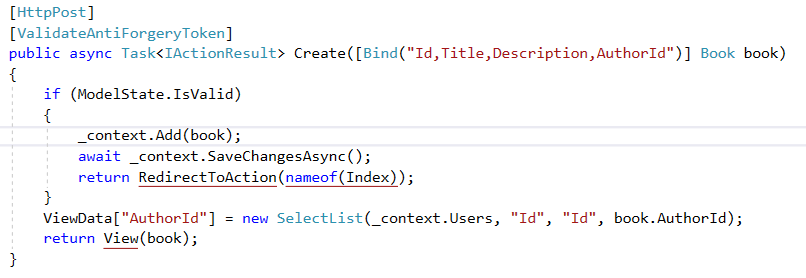
****

### Add Author Info

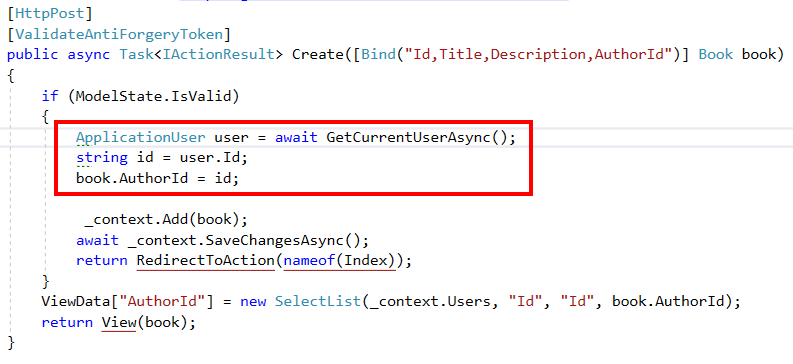
Right now, our **CRUD** operations **don’t** add the current user as an author of the books they created to our DB. We need to fix that. First, we need to add a **User Manager**:



Find the Create method in your BooksController. It should look like this:



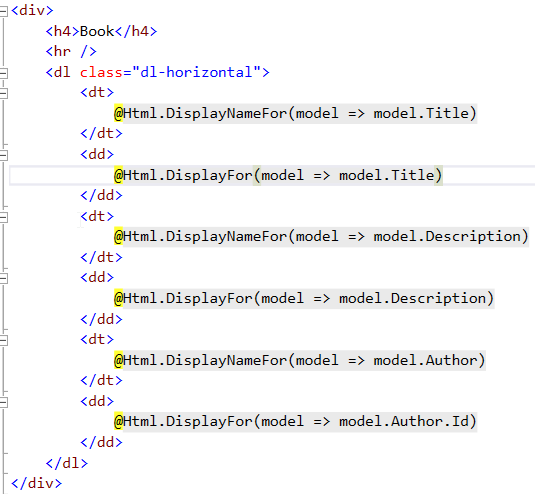
Add the following lines of code to the if statement:



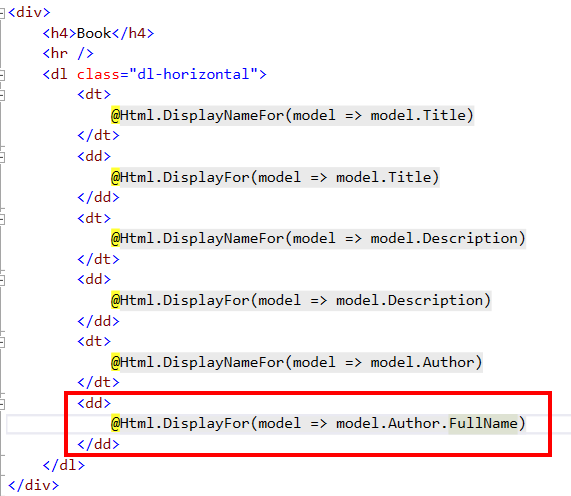
That will add the author to the book object.

### Show the Author

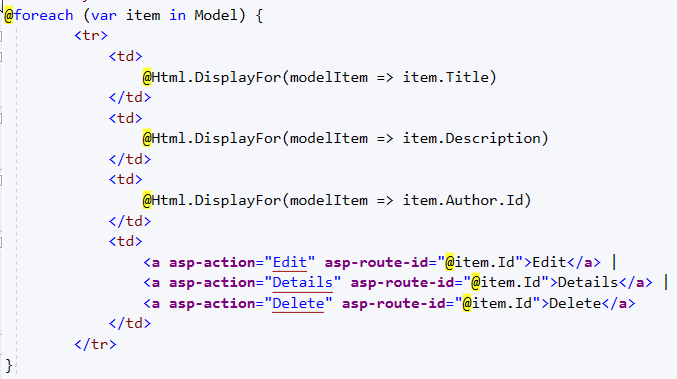
Our final step is to show the author of the book. Let’s go to our "Views/Books/Details.cshtml" file. It should contain a div tag, that looks like this:



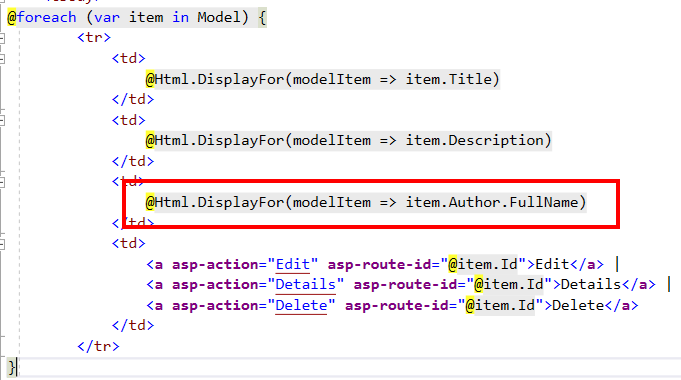
Change it to:



Now go to **Views/Books/Index.cshtml**.It should contain a foreach, that looks like this:

****

We want to display the **author’s name**, so we need to change it to:



You are ready to play with your Book Library now!