Exercise: C# and ASP.NET

This document defines several walkthroughs for creating ASP.NET 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

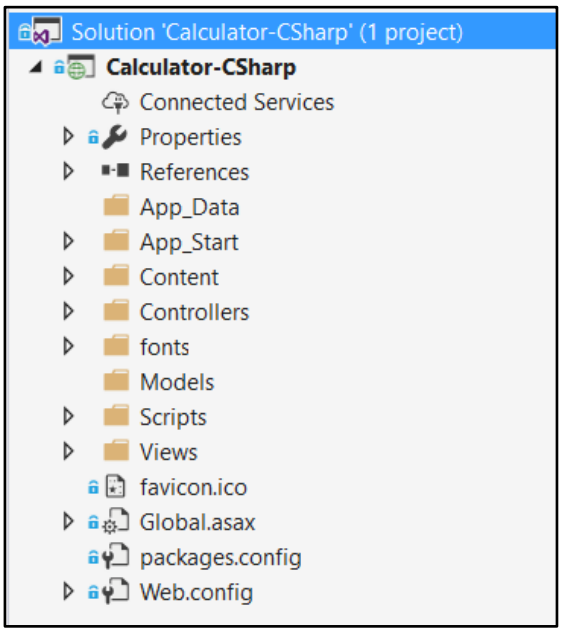
This document defines a complete walkthrough of creating a **Calculator** application with the ASP.NET Framework, from setting up the framework to implementing the fully functional application.

1. Base Project Overview

Our project will be built, using the **C#** language and the **MVC** framework **ASP.NET**. 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, 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 and bundled 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** 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!

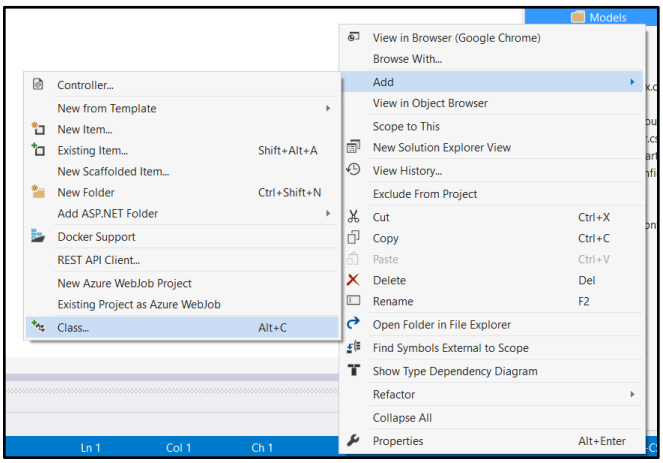
2. Implement Functionality

Create Calculator Model

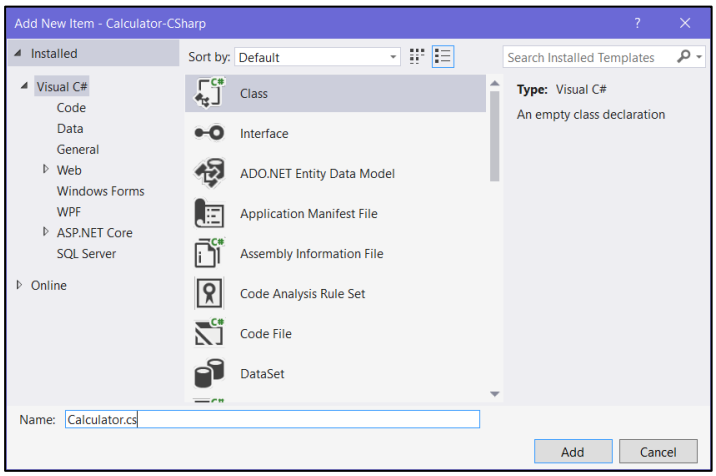
It's time to design our main model – the **Calculator**. It will contain the following properties:

- 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 → Add → Class]:



→



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

```
namespace Calculator_CSharp.Models
{
    public class Calculator
    {
        public decimal LeftOperand { get; set; }

        public decimal RightOperand { get; set; }

        public string Operator { get; set; }

        public decimal Result { get; set; }
    }
}
```

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

```
namespace Calculator_CSharp.Models
{
    public class Calculator
    {
        public Calculator()
        {
            this.Result = 0;
        }

        public decimal LeftOperand { get; set; }

        public decimal RightOperand { get; set; }

        public string Operator { get; set; }

        public decimal Result { get; set; }
    }
}
```

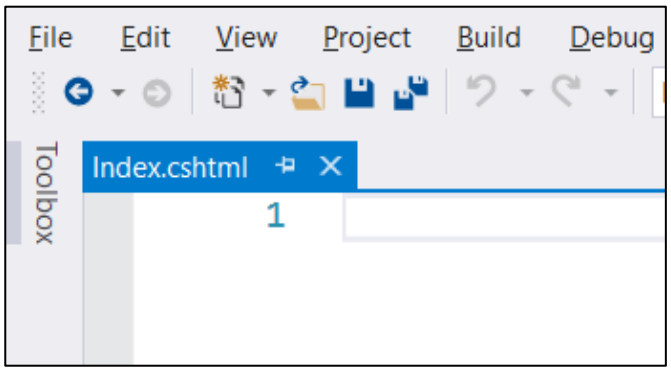
Now all that's left is to connect it to the rest of our little web application.

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:

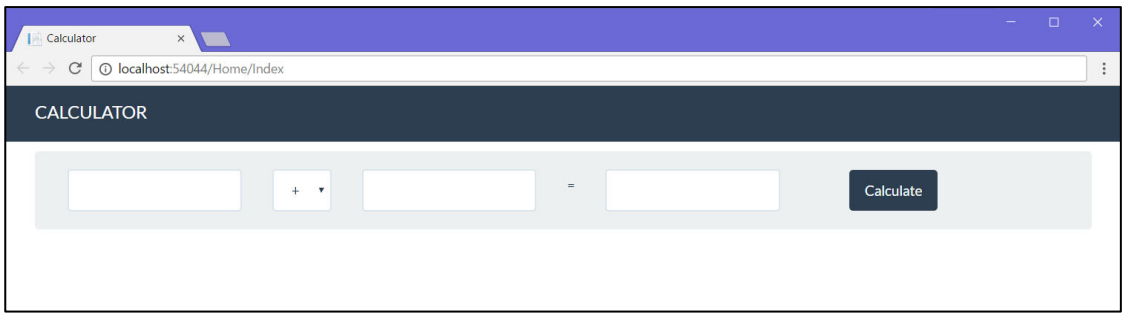


It's empty?! How does the header and footer seen above get displayed then? The answer is, we use a global **layout** file (**`/Views/Shared/_Layout.cshtml`**), so we don't have to copy-paste our page layout into every single view in

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 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`**:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>@ViewBag.Title</title>
    @Styles.Render("~/Content/css")
    @Scripts.Render("~/bundles/modernizr")
</head>
<body>
    <div class="navbar navbar-default navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                @Html.ActionLink("Calculator", "Index", "Home", new { area = "" },
                new { @class = "navbar-brand text-uppercase" })
            </div>
        </div>
    </div>
    <div class="container body-content">
        @RenderBody()
    </div>

    @Scripts.Render("~/bundles/jquery")
    @Scripts.Render("~/bundles/bootstrap")
    @RenderSection("scripts", required: false)
</body>
</html>
```

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:

```
using System.Web.Mvc;

namespace Calculator_CSharp.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }
    }
}
```

ViewResult Controller.View() (+ 7 overloads)
Creates a ViewResult object that renders a view to the response.

(view) ~/Views/Home/Index.cshtml

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:*

```
namespace Calculator_CSharp
{
    public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
            );
        }
    }
}
```

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:

```
using System.Web.Mvc;

namespace Calculator_CSharp.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult 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:

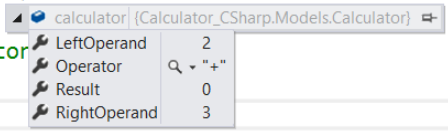
```
[HttpPost]
public ActionResult Calculate(Calculator calculator)
{

}
```

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);
}
```



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:

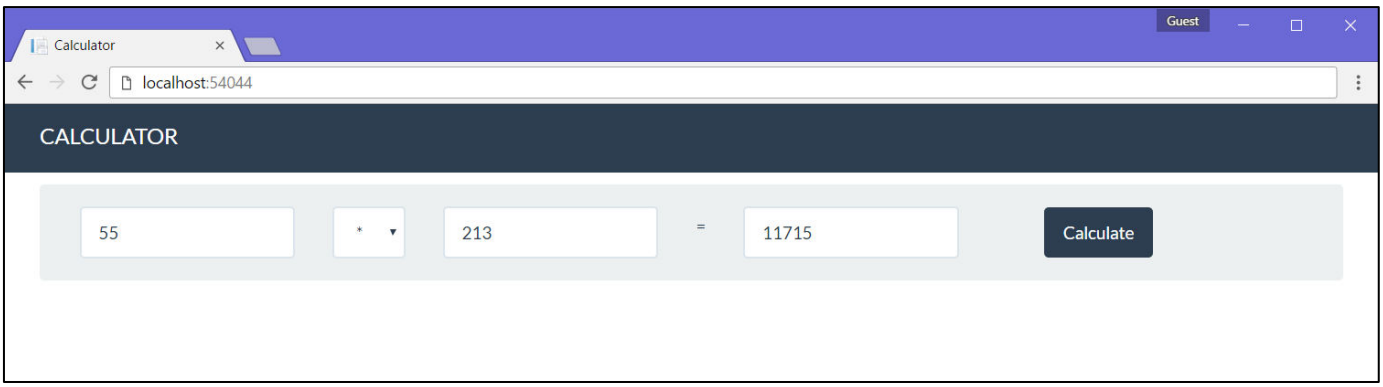
```
private decimal CalculateResult(Calculator calculator)
{
    var result = 0m;

    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);
}
```

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:



II. Data-Driven Apps

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

2. Book Library

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

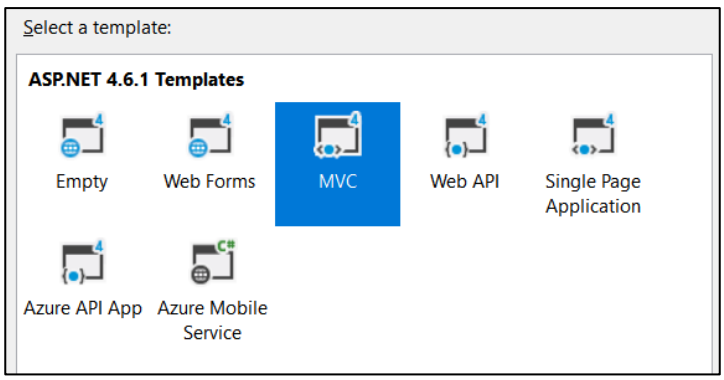
- We have users and use authorization for actions
- We can **edit books** (change title, etc...). Last time we could only create and delete them.

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**

1. 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 7.

2. Edit the Main Layout

Let's start by deleting the unnecessary lines from „`_Layout.cshtml`".

```
<div class="navbar-collapse collapse">
    <ul class="nav navbar-nav">
        <li>@Html.ActionLink("Home", "Index", "Home")</li>
        <li>@Html.ActionLink("About", "About", "Home")</li>
        <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
    </ul>
    @Html.Partial("_LoginPartial")
</div>
```

Delete this

3. Edit the Application Name

Go to your main layout and edit the following lines:

```
<body>
    <div class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                @Html.ActionLink("Application name", "Index", "Home", new { area = "" }, new { @class = "navbar-brand" })
            </div>
            <div class="navbar-collapse collapse">
                @Html.Partial("_LoginPartial")
            </div>
        </div>
    </div>
    <div class="container body-content">
        @RenderBody()
        <hr />
        <footer>
            <p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>
        </footer>
    </div>
```

We are done with the layout for now.

4. Simplify the Registration

Go to your "**App_Start/IdentityConfig.cs**" file. Find the following lines:

```
manager.PasswordValidator = new PasswordValidator
{
    RequiredLength = 6,
    RequireNonLetterOrDigit = true,
    RequireDigit = true,
    RequireLowercase = true,
    RequireUppercase = true,
};
```

Edit them to receive this result:

```
manager.PasswordValidator = new PasswordValidator
{
    RequiredLength = 3,
    RequireNonLetterOrDigit = false,
    RequireDigit = false,
    RequireLowercase = false,
    RequireUppercase = false,
};
```

This will simplify our password. Now we need to edit the models that validate our data. Go to the "**ManageViewModels.cs**". Find this line:

```
public class SetPasswordViewModel
{
    [Required]
    [StringLength(100, ErrorMessage = "The {0} must be at least {2} characters long.", MinimumLength = 6)]
    [DataType(DataType.Password)]
```

Edit the minimum length to 3 symbols. You will find a similar line in the **ChangePasswordViewModel** class in the same file. Edit the **minimum length** there too:

```
public class ChangePasswordViewModel
{
    [Required]
    [DataType(DataType.Password)]
    [Display(Name = "Current password")]
    1 reference
    public string OldPassword { get; set; }

    [Required]
    [StringLength(100, ErrorMessage = "The {0} must be at least {2} characters long.", MinimumLength = 6)]
```

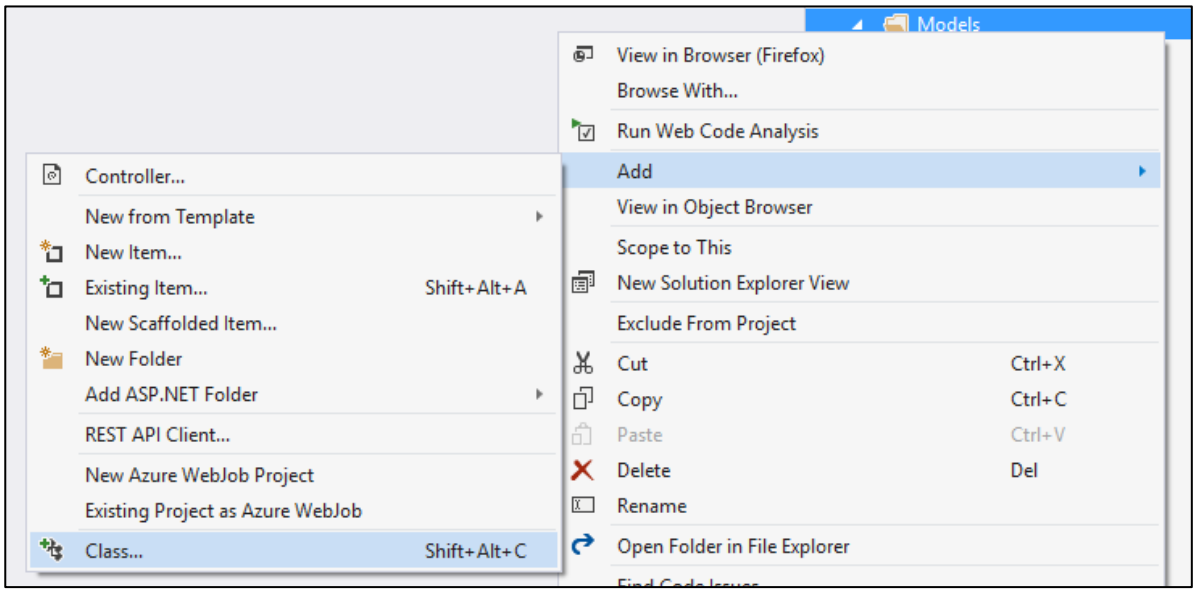
Use 3 symbols again.

Finally, we should go to the **AccountViewModels.cs** file. Find the lines that validate the password length, and change the value to 3 symbols. Here is how it should look like:

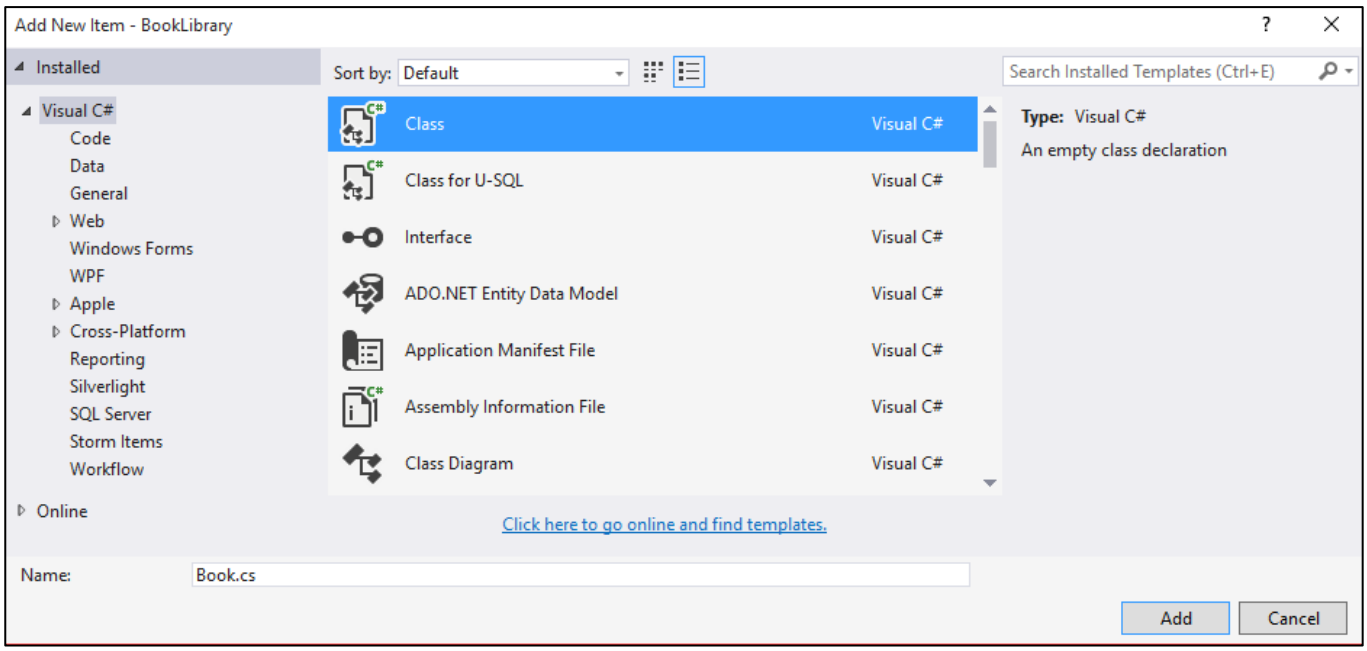
```
[StringLength(100, ErrorMessage = "The {0} must be at least {2} characters long.", MinimumLength = 3)]
```

5. Create the Book Model

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



Name the class **Book.cs**:



Add the following properties:

```
namespace BookLibrary.Models
{
    public class Book
    {
        public int Id { get; set; }

        public string Title { get; set; }

        public string Description { get; set; }

        public string AuthorId { get; set; }

        public ApplicationUser Author { get; set; }
    }
}
```

6. Extract the DbContext

Create a new model and name it "**ApplicationDbContext**".

Go to your **IdentityModels.cs** file and find the **ApplicationDbContext** class.

```
5 references
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
    1 reference
    public ApplicationDbContext()
        : base("DefaultConnection", throwIfV1Schema: false)
    {
    }

    1 reference
    public static ApplicationDbContext Create()
    {
        return new ApplicationDbContext();
    }
}
```

We need to **move** the class to a new file. We could either **cut** it and **paste** it into a file called **`ApplicationDbContext.cs`**.

After that, the **`ApplicationDbContext.cs`** file should look like this:

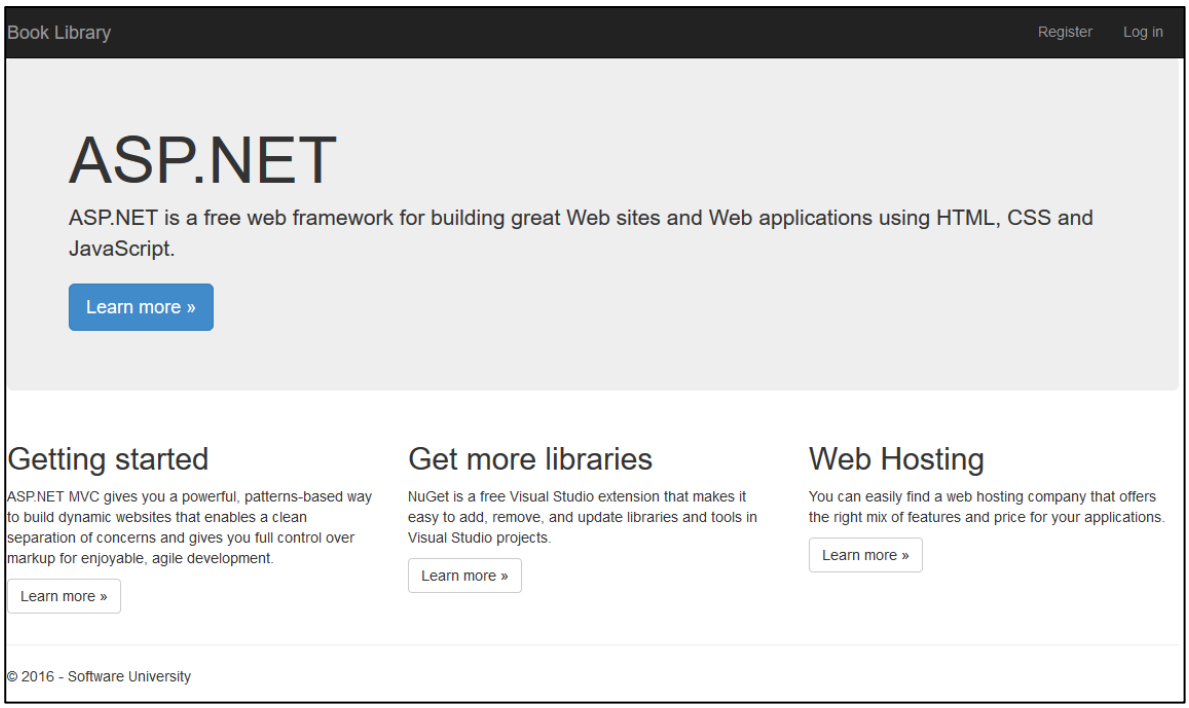
```
using Microsoft.AspNet.Identity.EntityFramework;

namespace BookLibrary.Models
{
    5 references
    public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
    {
        1 reference
        public ApplicationDbContext()
            : base("DefaultConnection", throwIfV1Schema: false)
        {
        }

        1 reference
        public static ApplicationDbContext Create()
        {
            return new ApplicationDbContext();
        }
    }
}
```

Add the following code:

```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
    public ApplicationDbContext()
        : base("DefaultConnection", throwIfV1Schema: false)
    {
    }

    public virtual DbSet<Book> Books { get; set; }

    public static ApplicationDbContext Create()
    {
        return new ApplicationDbContext();
    }
}
```

7. Run the Application for the First Time

Don't register a new user just yet! You should see this:



8. Add Full Name to Our User

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

```
public class ApplicationUser : IdentityUser
{
    public string FullName { get; set; }

    public async Task<ClaimsIdentity> GenerateUserIdentityAsync(UserManager<ApplicationUser> manager)
    {
        // Note the authenticationType must match the one defined in CookieAuthenticationOptions.AuthenticationType
        var userIdentity = await manager.CreateIdentityAsync(this, DefaultAuthenticationTypes.ApplicationCookie);
        // Add custom user claims here
        return userIdentity;
    }
}
```

Now go to the **`AccountViewModels.cs`** and find the **`RegisterViewModel`** class. Add the following lines:

```
public class RegisterViewModel
{
    [Required]
    [EmailAddress]
    [Display(Name = "Email")]
    public string Email { get; set; }

    [Required]
    [Display(Name = "Full Name")]
    public string FullName { get; set; }

    [Required]
    [StringLength(100, ErrorMessage = "The {0} must be at least {2} characters long.", MinimumLength = 3)]
    [DataType(DataType.Password)]
    [Display(Name = "Password")]
    public string Password { get; set; }

    [DataType(DataType.Password)]
    [Display(Name = "Confirm password")]
    [Compare("Password", ErrorMessage = "The password and confirmation password do not match.")]
    public string ConfirmPassword { get; set; }
}
```

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

```
// POST: /Account/Register
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser { UserName = model.Email, Email = model.Email };
        var result = await UserManager.CreateAsync(user, model.Password);
        if (result.Succeeded)
        {
```

Add this, to the existing line:

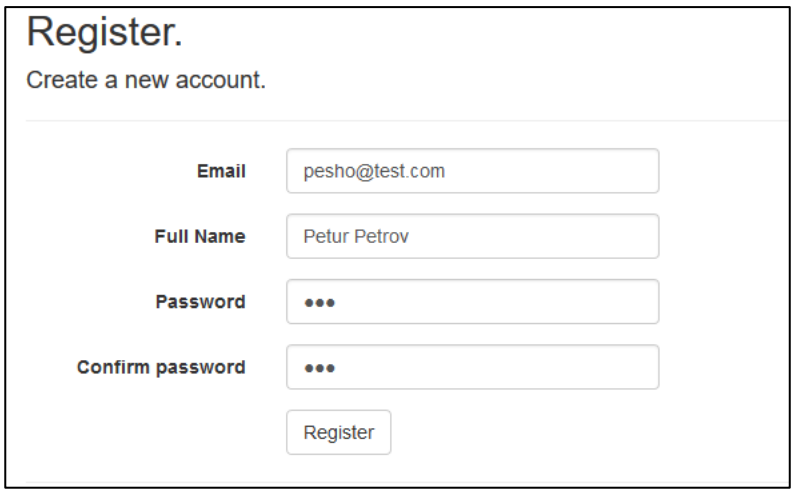
```
public async Task<ActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser { UserName = model.Email, Email = model.Email, FullName = model.FullName };
        var result = await UserManager.CreateAsync(user, model.Password);
```

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

```
<div class="form-group">
    @Html.LabelFor(m => m.Email, new { @class = "col-md-2 control-label" })
    <div class="col-md-10">
        @Html.TextBoxFor(m => m.Email, new { @class = "form-control" })
    </div>
</div>
<div class="form-group">
    @Html.LabelFor(m => m.FullName, new { @class = "col-md-2 control-label" })
    <div class="col-md-10">
        @Html.TextBoxFor(m => m.FullName, new { @class = "form-control" })
    </div>
</div>
<div class="form-group">
    @Html.LabelFor(m => m.Password, new { @class = "col-md-2 control-label" })
    <div class="col-md-10">
        @Html.PasswordFor(m => m.Password, new { @class = "form-control" })
    </div>
</div>
<div class="form-group">
    @Html.LabelFor(m => m.ConfirmPassword, new { @class = "col-md-2 control-label" })
    <div class="col-md-10">
        @Html.PasswordFor(m => m.ConfirmPassword, new { @class = "form-control" })
    </div>
</div>
<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        <input type="submit" class="btn btn-default" value="Register" />
    </div>
</div>
```

9. Register New User

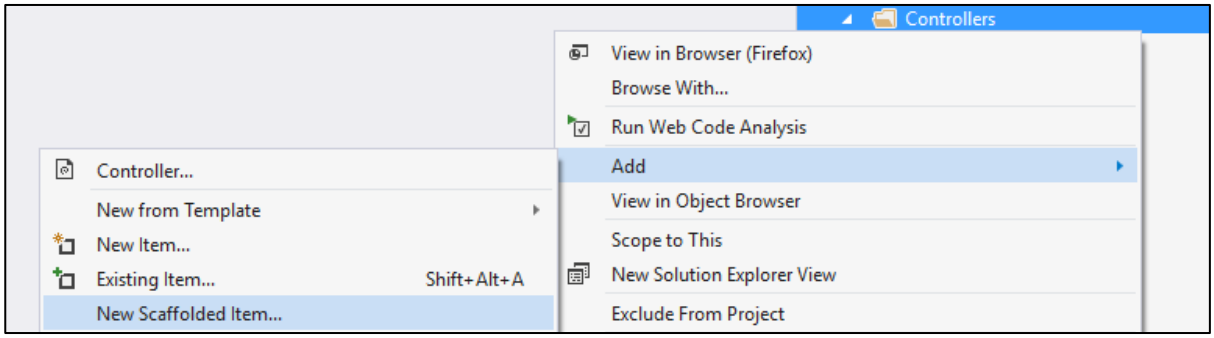
Run the application and try to register a new user:

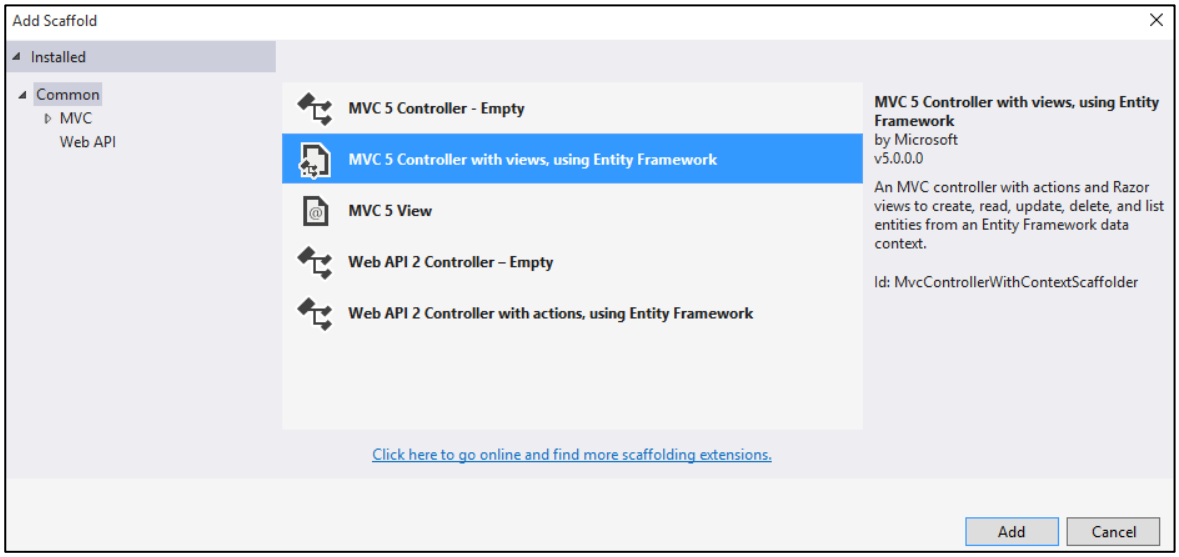


If it works you will be logged in, and you will see welcome message like this:

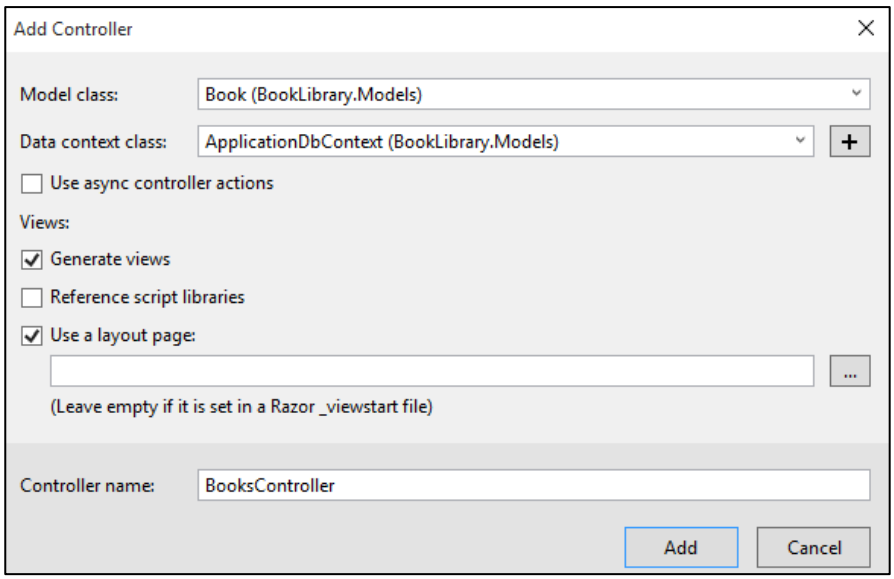
10. Create the Book Controller

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



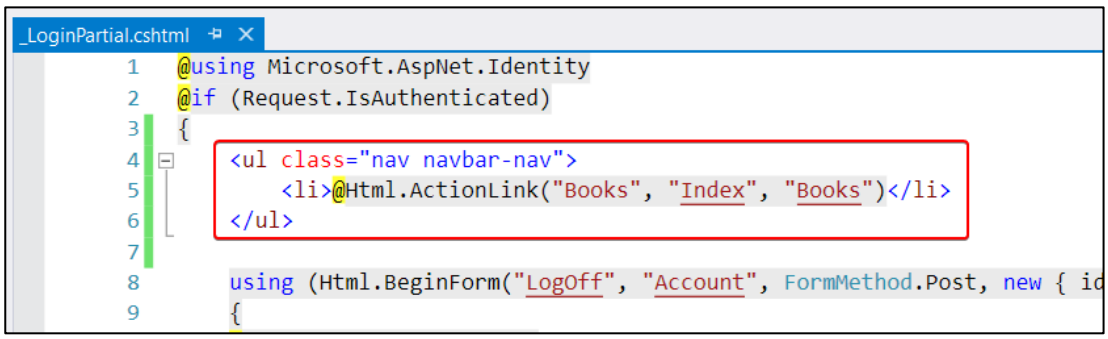


Use your **Book Model** and your **`ApplicationDbContext`** as shown in the image below:

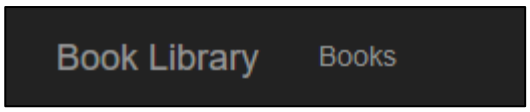


11. Edit the User Layout

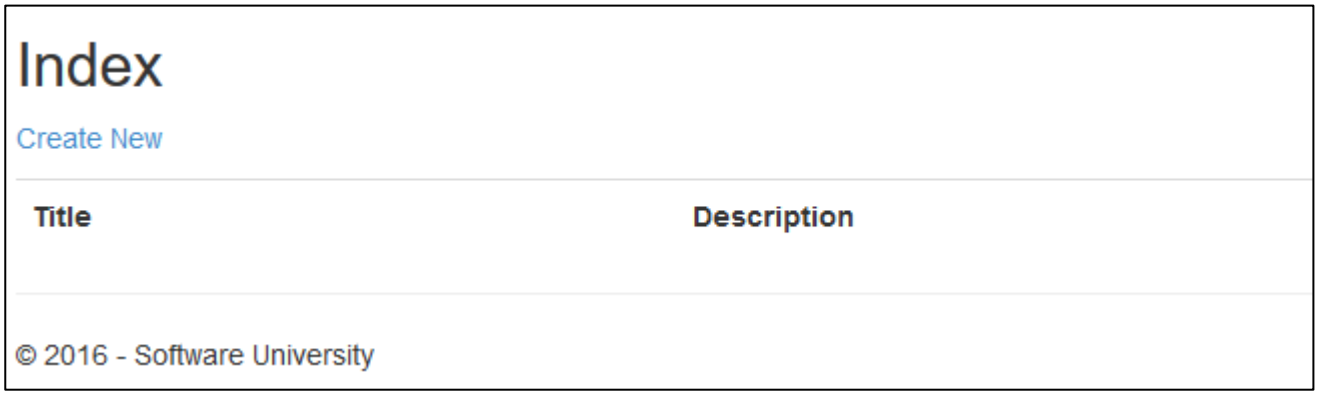
Go to the **_LoginPartial.cshtml** layout and add the following code:



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!

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

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create([Bind(Include = "Id,Title,Description")] Book book)
{
    if (ModelState.IsValid)
    {
        db.Books.Add(book);
        db.SaveChanges();
        return RedirectToAction("Index");
    }

    return View(book);
}
```

Add the following lines of code to the **`if`** statement:

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create([Bind(Include = "Id,Title,Description")] Book book)
{
    if (ModelState.IsValid)
    {
        book.AuthorId = User.Identity.GetUserId();

        db.Books.Add(book);
        db.SaveChanges();
        return RedirectToAction("Index");
    }

    return View(book);
}
```

That will add the author to the book object.

13. Show the Author

Our final step is to show the author of the book. We will do that by finding the **`Details`** method in our **`BookController`**. You should have something like this:

```
public ActionResult Details(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }

    Book book = db.Books.Find(id);

    if (book == null)
    {
        return HttpNotFound();
    }

    return View(book);
}
```

We will edit one of those lines, with LINQ query:

```
public ActionResult Details(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }

    Book book = db.Books.Include(b => b.Author).Single(b => b.Id == id);

    if (book == null)
    {
        return HttpNotFound();
    }

    return View(book);
}
```

This will **explicitly request** the author's information (full name, email, etc.) from the database alongside the book. We need to do this because by default Entity Framework doesn't load properties from other entities. This is called **Lazy Loading**.

We are almost done. Let's go to our "**`Views/Books/Details.cshtml`**" file. It should contain a **`div`** tag, that looks like this:

```
<div>
    <h4>Book</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Title)
        </dt>

        <dd>
            @Html.DisplayFor(model => model.Title)
        </dd>

        <dt>
            @Html.DisplayNameFor(model => model.Description)
        </dt>

        <dd>
            @Html.DisplayFor(model => model.Description)
        </dd>

    </dl>
</div>
```

Add the following element:

```
    <dt>
        @Html.DisplayNameFor(model => model.Description)
    </dt>

    <dd>
        @Html.DisplayFor(model => model.Description)
    </dd>

    <dt>
        @Html.DisplayNameFor(model => model.Author)
    </dt>

    <dd>
        @Html.DisplayFor(model => model.Author.FullName)
    </dd>
```

You are ready to play with your Book Library now!