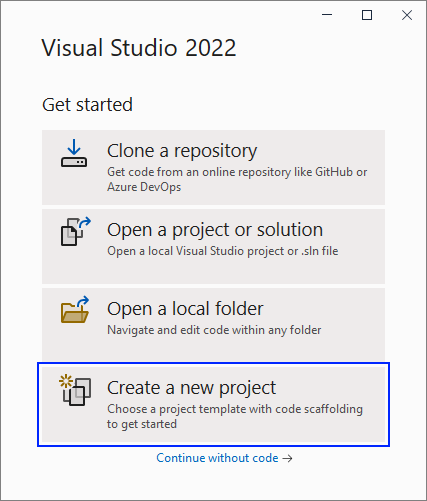
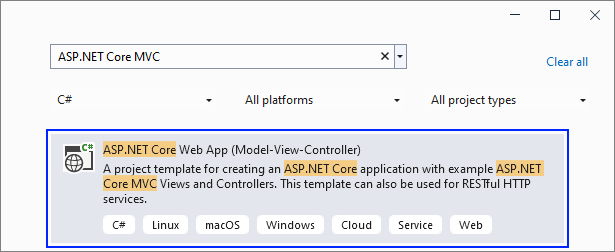
# Demo: ASP.NET Core MVC

Demo problems for the ["Back-End Technologies Basics"](https://softuni.bg/trainings/4398/back-end-technologies-basics-january-2024) Course @ SoftUni.

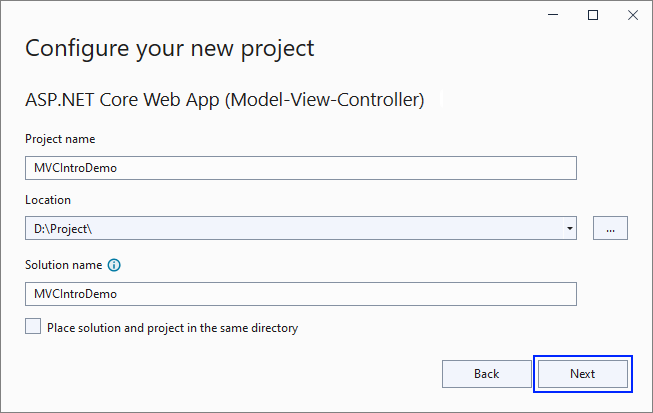
## Create Simple Pages in an ASP.NET Core App

In this task you should implement the pages from the demo from the slides for this topic. To do so, create a **new ASP.NET Core MVC app**. Open **Visual Studio** and **choose** [Create a new project]. On the next step, **choose** [ASP.NET Core Web App (Model-View-Controller)] as a **project template**. The steps are shown below:

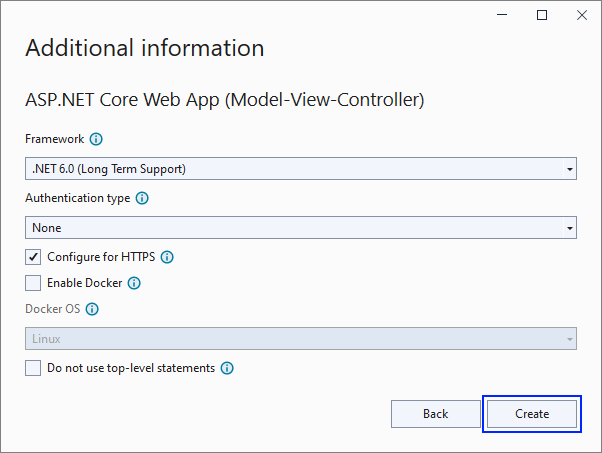


****

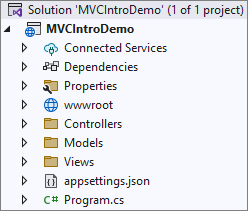
Give a **name** to your project and solution:



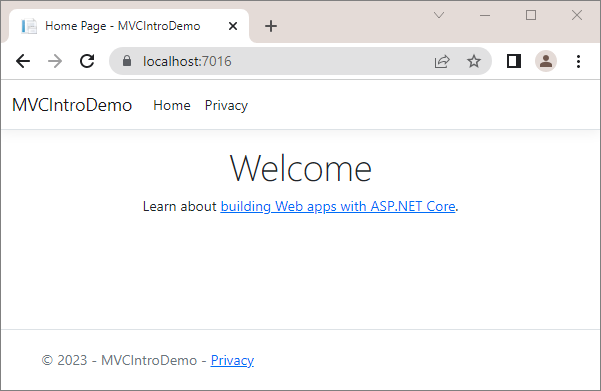
On the next step you should **choose** your target frameworok and click on the [Create] **button**:

****

Now your **app is created** and looks as shown below. Note that it has **folders** for **controllers**, **models** and **views** because of the template we chose:



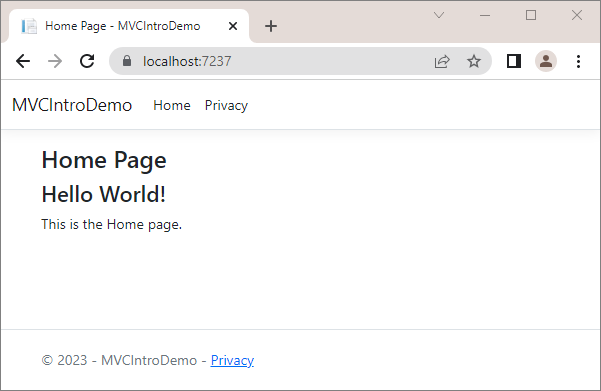
If you **run the app**, you will see the **default** "Home" **page**, which is served by the HomeController in the app:



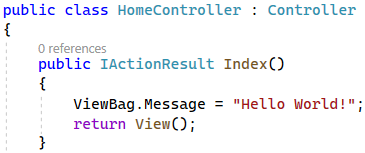
### HomeController Pages

#### Modify the "Home" Page

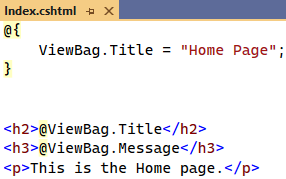
Now we want to **modify** the "Home" **page** to look like this:



Change the Index() **method** of the HomeController to **change the page**. The **controller action** should return a **view**, as it does already, but also use the ViewBag **class** to **create a message**, which will be **used in the view**. Modify the method like this:



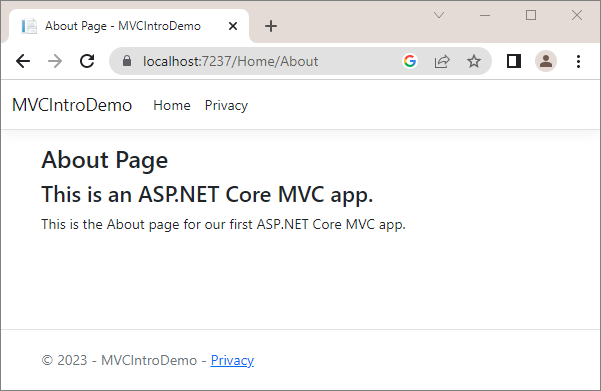
Now you should modify the Index.cshtml **view** in the "/Views/Home" **folder** to **display the page differently**. Use the ViewBag **class** to **get the message** from the controller. Note how the Razor **views** allow us to use **C# code** inside **HTML**:



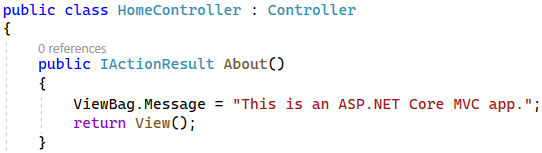
**Run the app** with [Ctrl] + [F5] and make sure the "Home" **page** looks as shown on the screenshot above.

#### Create the "About" Page

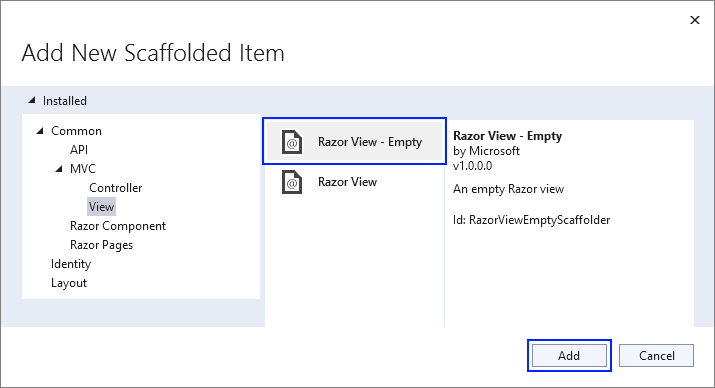
Create an "About" **page** in the app, which should look like this:

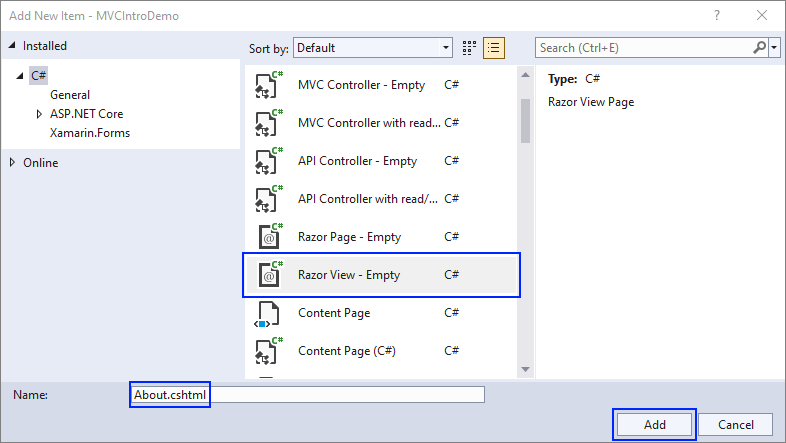


The page should be **accessed on "**/Home/About". Create an About() **controller action** in the HomeController **class** for the "About" **page**. The controller method should return a **view**. It should also use the ViewBag **class** to **pass a message** to the returned view. Write the method like this:

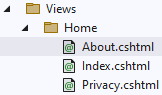
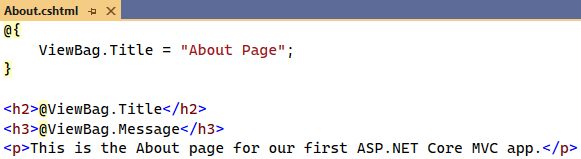


Now you should create an About.cshtml **view** in the "/Views/Home" **folder**. To do this, **right-click** on the "/Views/Home" folder and **choose** [Add] 🡪 [View]:





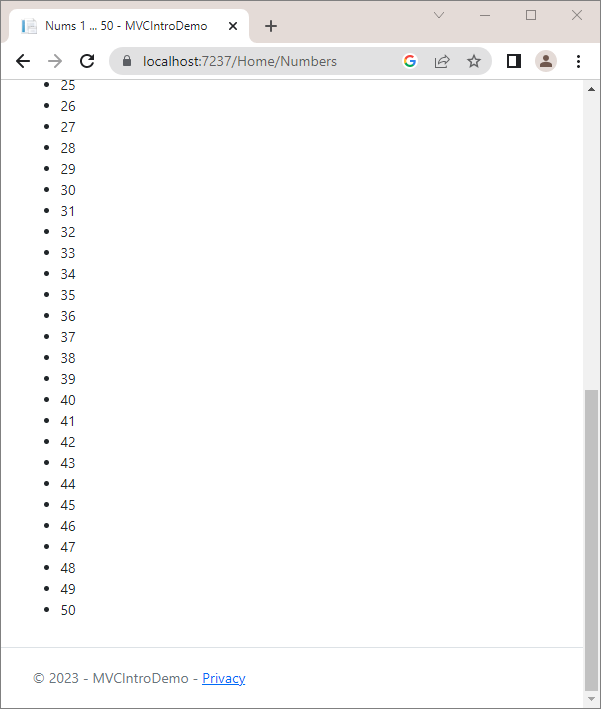
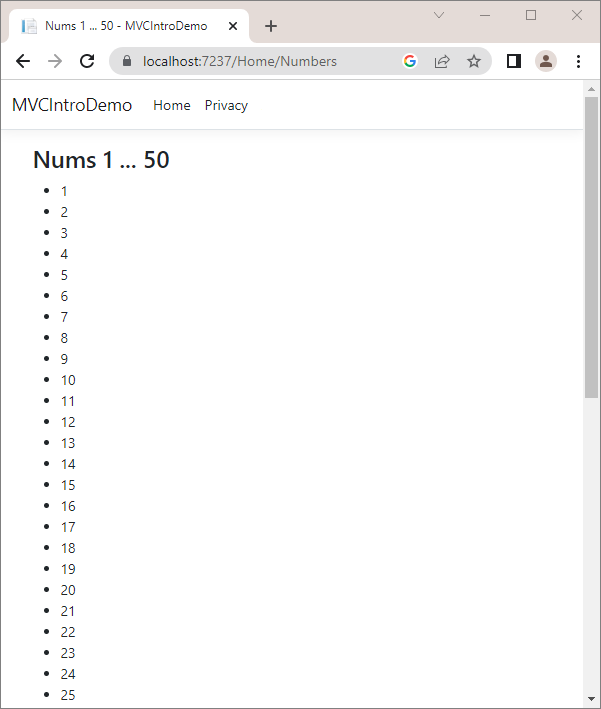
Now the About.cshtml **view** should be created. Write it like this:

 🡪 

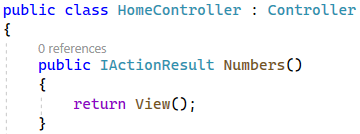
Look at the "About" **page** in the **browser**. You can access it on "/Home/About". It should look as shown above.

#### Create the "Numbers 1…50" Page

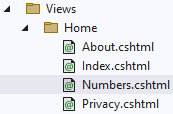
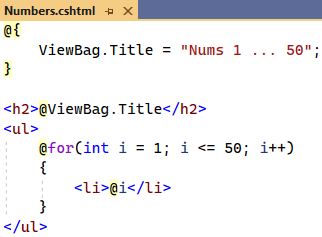
The "Numbers 1…50" page should **display the numbers from 1 to 50**. It should be accessed on "/Home/Numbers" and shoud look like this:



Create a Numbers() **controller method** in the HomeController, which should only **return a view**:



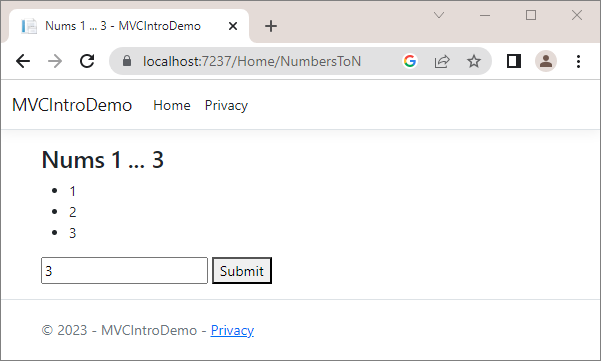
Create a Numbers.cshtml **view**, which should use a for **loop** to **display each number**. Write the **view** like this:

 🡪 

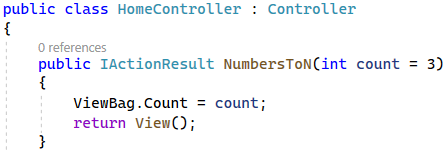
Make sure the **numbers from 1 to 50 are displayed** on the "Numbers 1…50" **page** on "/Home/Numbers".

#### Create the "Numbers 1…N" Page

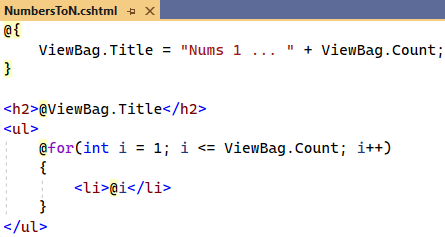
This page is similar to the previous one but the **user should enter a number** N. Then, only **numbers from 1 to** N should be displayed:



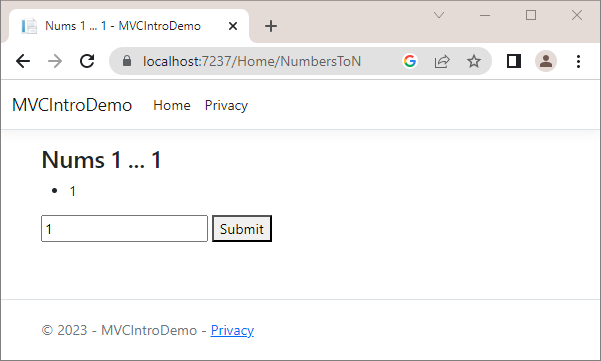
Write a NumbersToN() **method** in the HomeController. It should **accept a count parameter** from the **view** (with **default value** of the parameter **3**). Then, it should **add the count number** to a ViewBag and **return a view**:



Then, the NumbersToN.cshtml view should **display the numbers in a** for **loop** and should have a **form for** **submitting a count number**. The **number input field** should have a "name" **attribute**, so that its **value** is passed to the **controller action**. Do it like this:

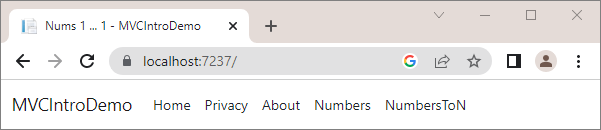


Try out the page in the browser on "/Home/NumbersToN". It should **display different numbers**, depending on the **count** you enter in the form:

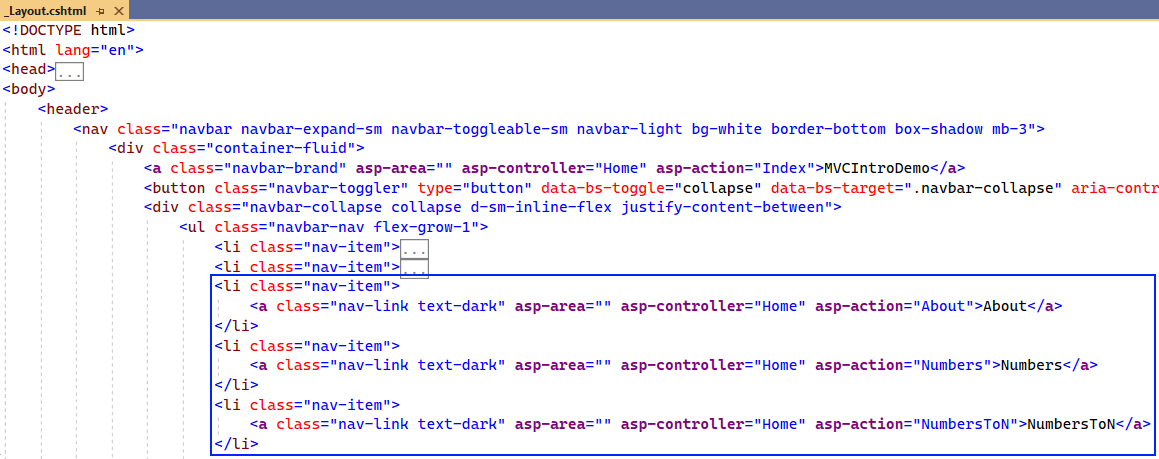


#### Add Navigation Links

As we have **created the pages** we need, let's **add links to the navigation pane** to access them easier. The **navigation** **pane** should look like this:



To **add links**, go the \_Layout.cshtml **partial view** in the "/Views/Shared" **folder**, as this view is responsible for the **common design** of all pages. Add the following lines:

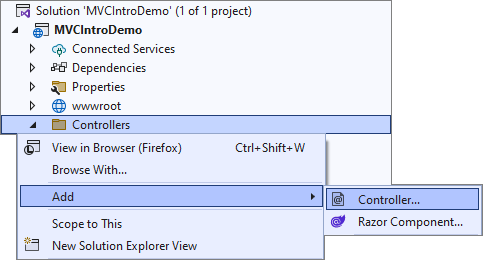


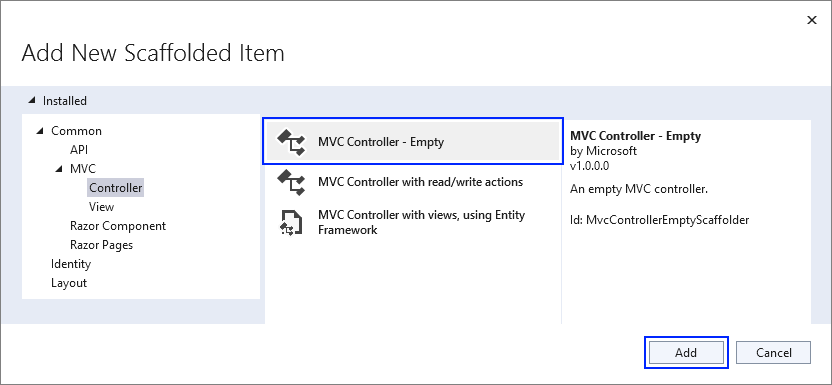
The asp-controller and asp-action **tag helpers** set the **controller** and **action names** of the page, which should be accessed.

Try out if the **links work correctly** and open the correct pages in the browser.

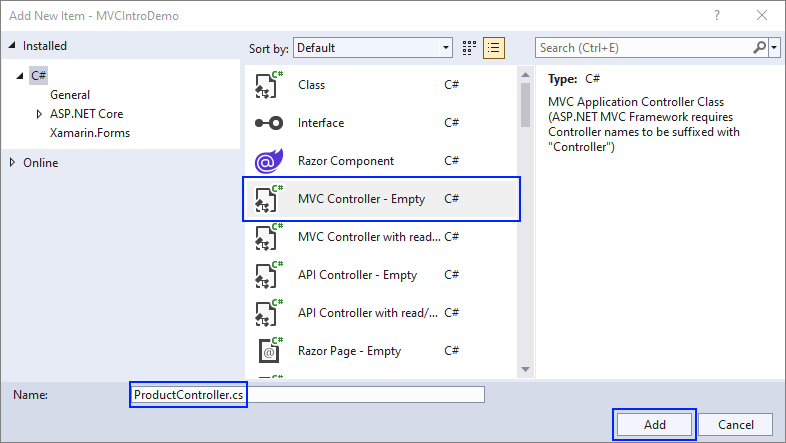
### ProductController Pages

The ProductController will have **controller actions for displaying hardcoded products on pages**. Create the ProductController in the "Controllers" folder. To **create a controller**, **right-click** on the "Controllers" folder, click on [Add] 🡪 [Controller] and **choose [MVC Controller – Empty]** to create an **empty controller class**:

 🡪



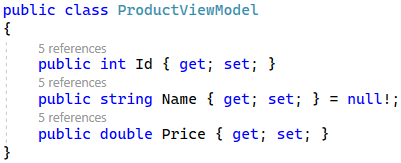
Set the **controller name** like this:



Now your **controller class** is created in the "Controllers" **folder** and **inherits the** Controller **base class**:

 🡪

We will **display hardcoded products**. First, you should **create a model for these products**, which should have an **id**, **name** and **price**. Create a new folder "**Product"** in the "**Models**" folder and add a new ProductViewModel class in the "Product" **folder** with the following **properties**:

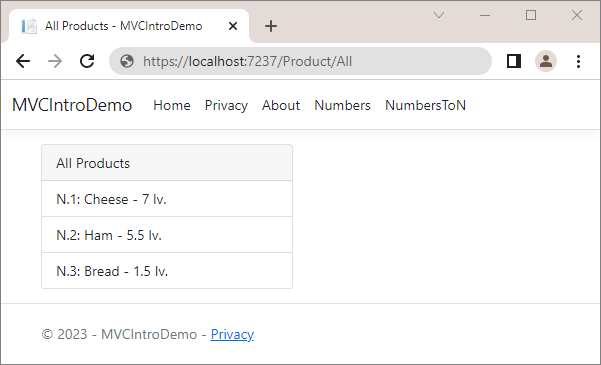


Now go back to the ProductController and **add a field with products**. The field should have a **collection of** ProductViewModel with **three products** like this:

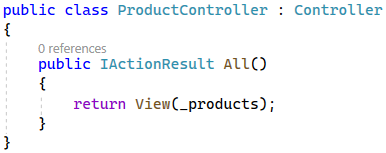


Now use these **products in controller methods**.

#### Create the "All Products" Page



Create an All() **controller method** in the ProductController, which should only **return a view** with the **products collection**:

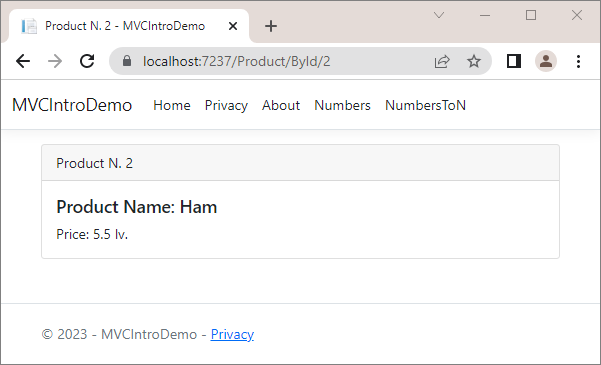


Now you should **create a** "Product" **folder** in the "Views" **folder**, which will have **all views for the** ProductController **methods**. In it, add an All.cshtml **view**, which should **accept a collection of** ProductViewModel. Then, **foreach the products** and **use the model properties** to display the product data:

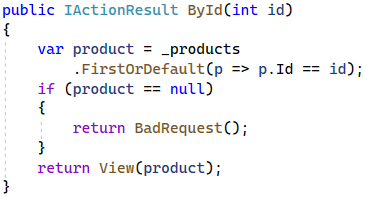


Try the "All Products" **page** on "/Product/All" in the browser.

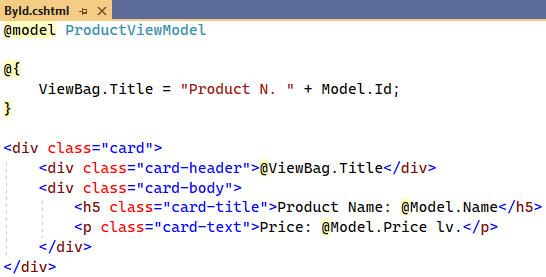
#### Create the "Product By Id" Page



Write the ById(int id) **method** in the ProductController. It should **pass a product by a given id** to the **view**, if it **exists**. If it does not, it should return a BadRequest:



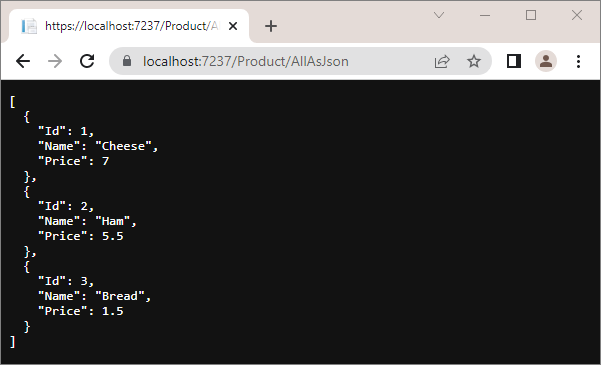
The ById.cshtml **view** is the following:

 🡪

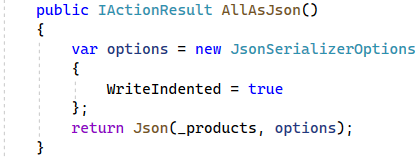
Go to the "Page By Id" **page** on "/Product/ById/{id}" with a **valid** and an **invalid** **product** **id**.

#### Return Products as JSON

Our task now is to **return the products in a JSON format** when the user accesses "/Product/AllAsJson":



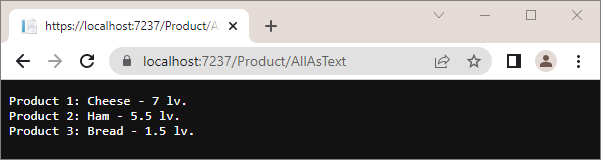
Create the AllAsJson() method in the HomeController, which should return a **JSON with the products** as shown below. It should use **JSON options** to display the JSON **indented**:



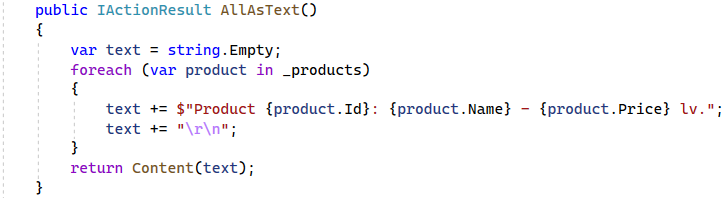
Try the page in the browser and make sure that **products are displayed correctly** as **JSON**.

#### Return Products as Plain Text

Now we should **return the products as a plain text** in a **custom format** when the user accesses "/Product/AllAsText":



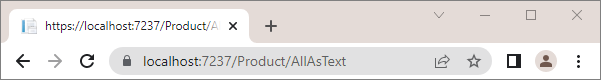
The AllAsText() **method** in the ProductController should **create a** string **of all products** and return it as a **plain text response**:

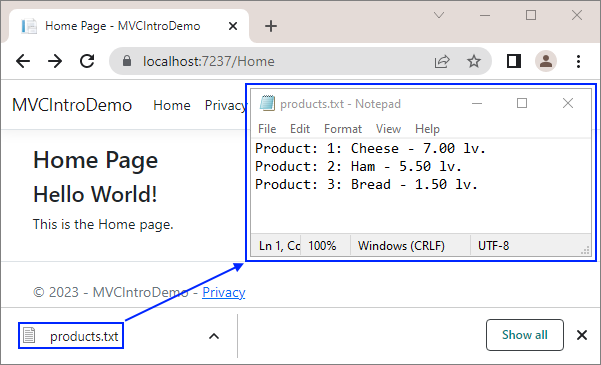


Try it in the browser.

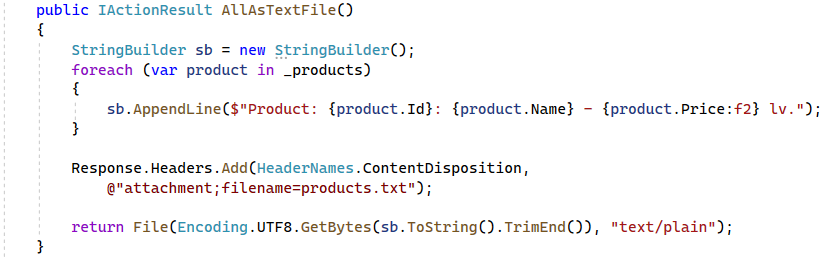
#### Download Products As Text File

Now we want to **download a text file with the products** by accessing "/Product/AllAsTextFile":

 🡪

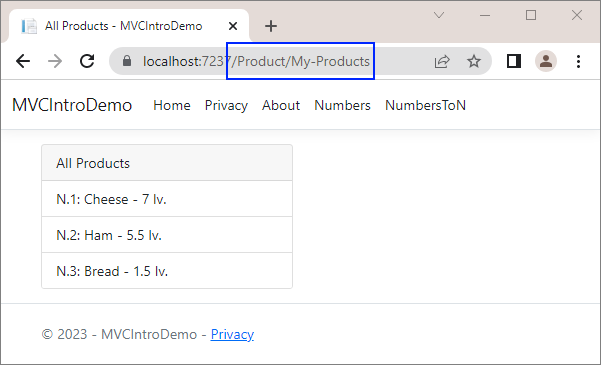


The AllAsTextFile() **method** in the ProductController should form a **text with the products**. Then, it should **add the** Content-Disposition **header** to the Response, so that the **file is downloaded** as an **attachment**. At the end, it should **return a file with the text as a byte array** and the **plain text type**. Write it like this:

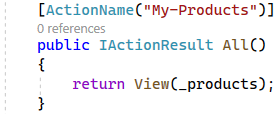


#### Access the "All Products" Page on Another URL

Now our task is to make the "All Products" page **accessible** **on** "/Product/My-Products":



To do this, add the [Action Name] **attribute** over the All() **method** of the ProductController. In this way, you will **set an action name**, different from the real one:

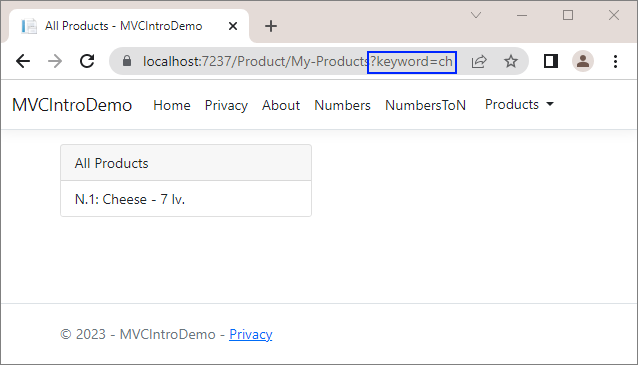


You should also **change** the All.cshtml **view** **name** to My-Products.cshtml, as the **view** and the **controller** **action** should have the **same** **names**:

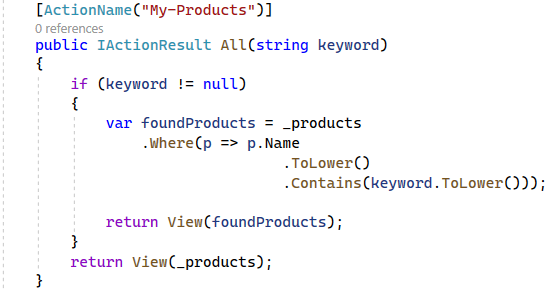


#### Add Search to the "All Products" Page

Finally, we want to **modify** the "All Products" page to use a **keyword** in the **URL** to **filter the displayed products** like this:

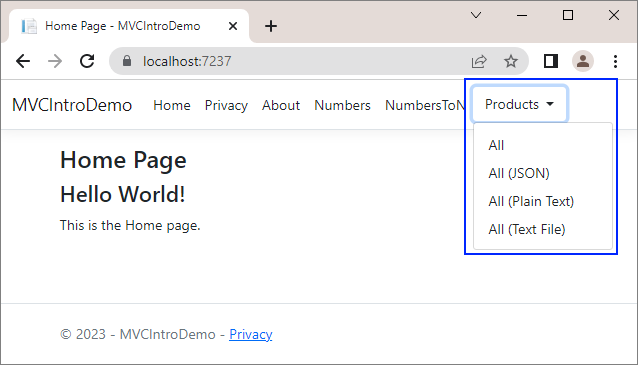


To do this, make the All() **controller action** **accept a keyword** and return only the **filtered products**, when there is a **keyword** in the **URL**:



Enter **different keywords** on "/Product/My-Product?keyword={keyword}" in the **browser** and make sure that only **products with the keyword in their name** are shown.

#### Add Navigation Links



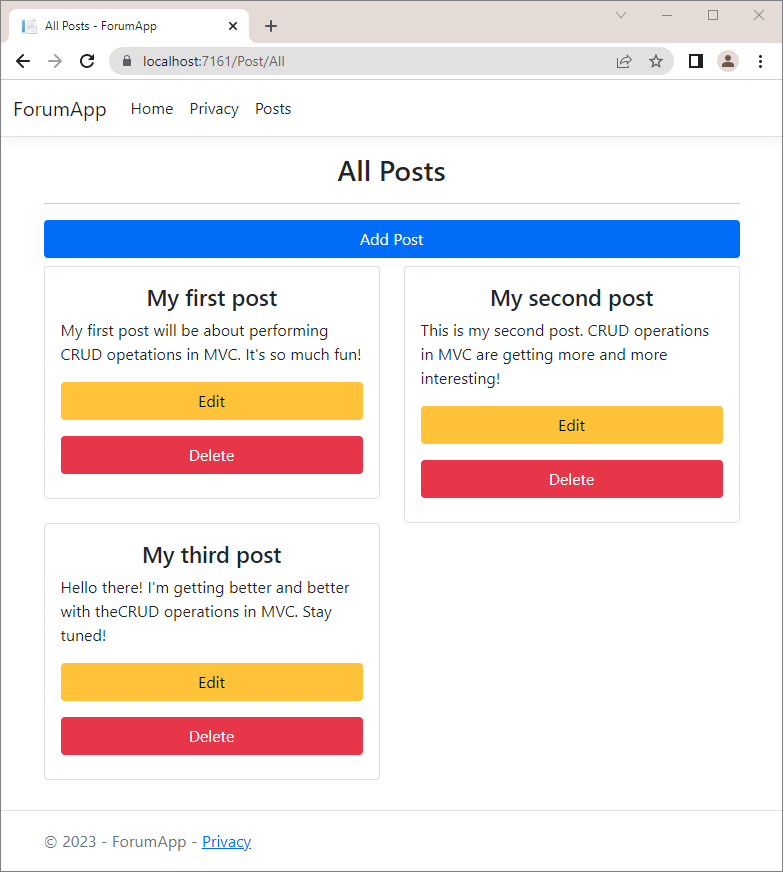
Modify the \_Layout.cshtml **view** like this to have the above **links**:

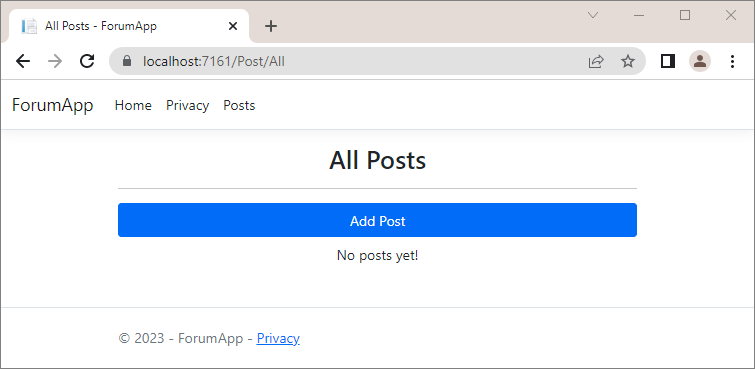


Try out all **new links** in the browser. They should **access the correct pages**.

## Forum

In this workshop we will create a **simple ForumApp, similar to the ShoppingListApp** from the slides from the previous session. Our app will have a page for **displaying all added forum posts** with their content and with [**Edit**] and [**Delete**] buttons and an [**Add Post**] button. It also should display a form for adding a post and editing a post and it should ask for a confirmation when deleting a post. It will look like this:

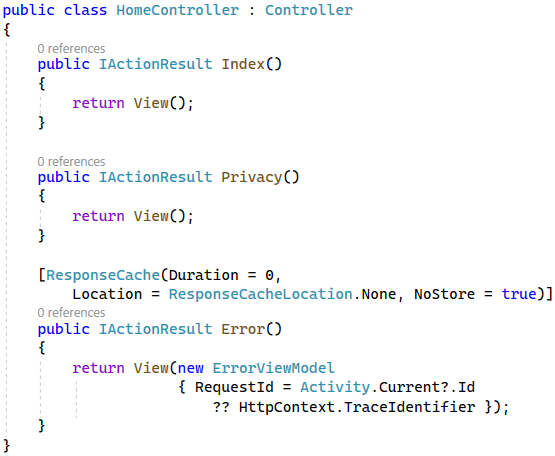




## Create a New App

Create a new **ASP.NET Core Web App (Model-View-Controller)** and name it **ForumApp**.

Remove the unnecessary code from the **HomeController** class. It should look like this:



## Define Entities

We should start with creating the data model, which we will need for our database. We will have only one data model class – **Post**.

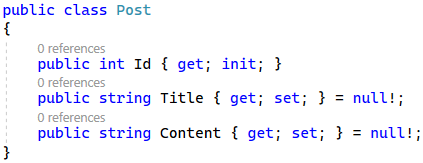
Create a "**Data**" folder in the project. It will hold the context of our app and the data models. The data models will be placed in a separate folder in the "**Data**" folder, which we will name "**Models**". It's a good practice for the data model classes to be in a separate folder from the **ForumAppDbContext** class. Create a new class and name it **Post**:



The **Post** class should have the following properties:

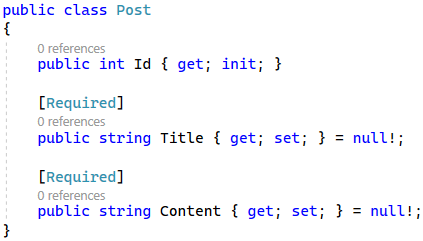
* **Id** – an unique **integer**, primary key
* **Title** – a **string** with min length **10** and max length **50** (required)
* **Content** – a **string** with min length **30** and max length **1500** (required)

First, create the **Post** class in the "**Models**" folder with its **properties without restrictions:**

****

Note that we use **init** setters only for **properties**, which **won't be changed after the initialization**. For example, the **Id** property will always be the same.

Now, add the **[Required]** **attribute** to those **properties**, which **must have a value**.

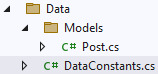


Note that you don't need to set the **Id** property as a **primary** **key** in any way – this is done **by** **default** in **EF Core** because of the "**Id**" property name.

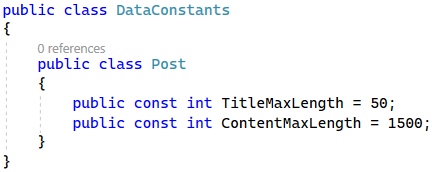
Now we should **restrict the properties**. The only restriction we will set is for the **max** **length**, as the database doesn't care about other validations. **Min length**, **range** and other **restrictions** are added to **model classes**.

The **properties**, which should have a **max length,** are Title and Content.

It is a good idea to create a **separate class with constants** for the max length values. In the "Data" **folder**, create the DataConstants **class**:



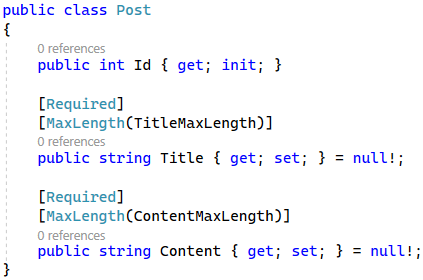
In this class, create a **Post** class for the constants, connected to a post entity, and the constants themselves:



Add the needed namespace to use the constant class in the **Post** class:



Use the **constants** in the [MaxLength] **attribute** to **set the max length** like this:

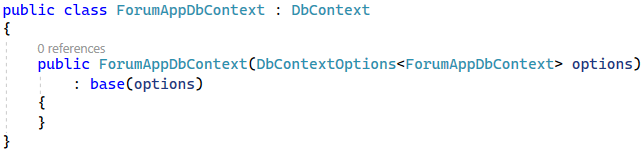


Now the model class is ready.

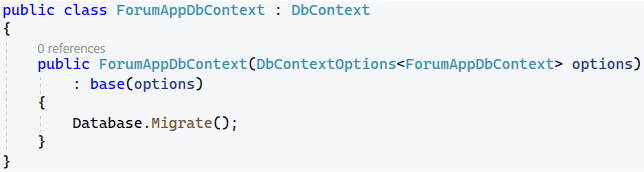
## Define the DbContext

Now let's add our **ForumAppDbContext** class in the "**Data**" folder. In order for the ForumAppDbContext to inherit the DbContext class, we have to install the **Microsoft.EntityFrameworkCore** package. Don't forget to install the following packages, too, as we will need them for this workshop:

* **Microsoft.EntityFramewrokCore.Design**
* **Microsoft.EntityFramewrokCore.SqlServer**
* **Microsoft.EntityFramewrokCore.Tools**



We should use the **Migrate()** method in the constructor in order for the changes to be applied to tbe database directly.

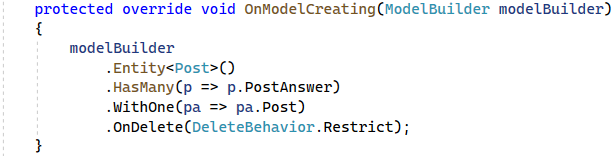


Create the **DbSet** properties for the table in the database:

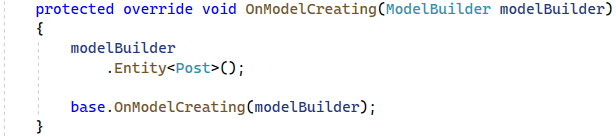


|  |  |
| --- | --- |
| Icon  Description automatically generated | EF Core uses a "cascade" **delete** by default when **removing an entity**. This means that if a record in the **parent table is deleted**, then the **corresponding records** in the child table **will automatically be deleted**. To prevent this from happening, it's a good practice to set the **delete behavior** to "restrict", so that an entity, which has **connections** to other entities in the database, **won't be deleted.** |

In our case, there's no entity that has connections to other entities. If there was such entity, the **OnModelCreating(ModelBuilder builer)** method in the **ForumAppDbContext** class should be overridden like in the example below:



At the end, invoke the **base** OnModelCreating() **method**:

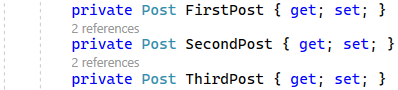


Now our **database structure** is ready. If you migrate it now, however, it will be **created with empty tables**. For this reason, let's see how to **seed some data** **to fill in the database tables**.

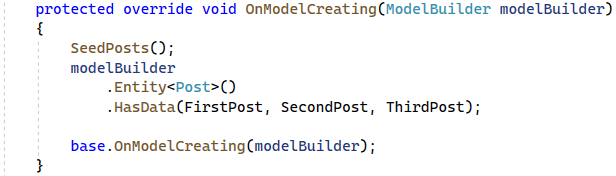
## Seed the Database

Now, we want to **populate the database** with an **initial set of data**. This will include **three posts.**

First, create properties for the above objects in the **ForumAppDbContext** class:

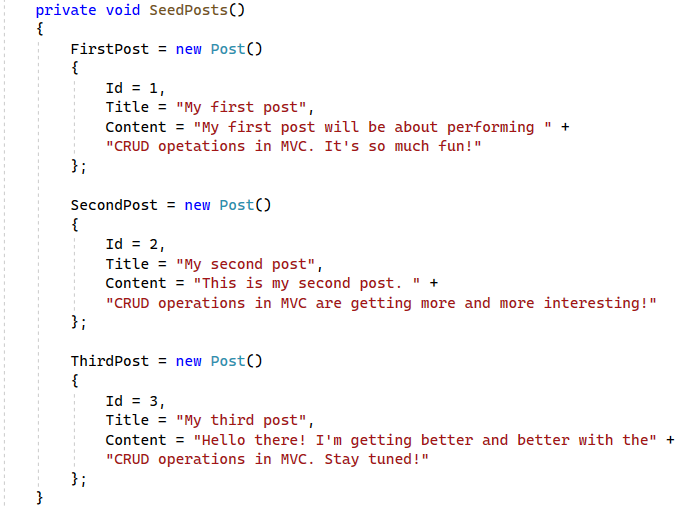


Then, we will use a **separate** **method** to **add data to these objects**, which will be **added** to the **corresponding** **database** **table** in the **OnModelCreating(ModelBuilder builder)** **method**. Add the following lines of code to the method, before invoking the base one:



|  |  |
| --- | --- |
| Icon  Description automatically generated | When **invoking more that one seeding method**, it is important that the **seeding methods are invoked in the correct order**, as they **depend** on each other. |

Let's implement the **SeedPosts()** class, which will create the posts for the database.

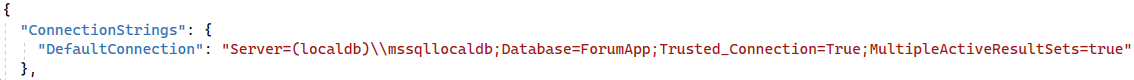


Now we have a db context with **seeded data** and our **database is ready to be migrated**.

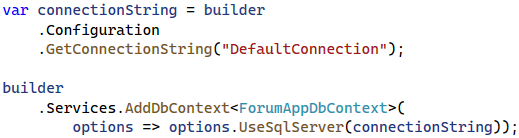
## Create a Migration

Now we will now **create a migration** to the database.

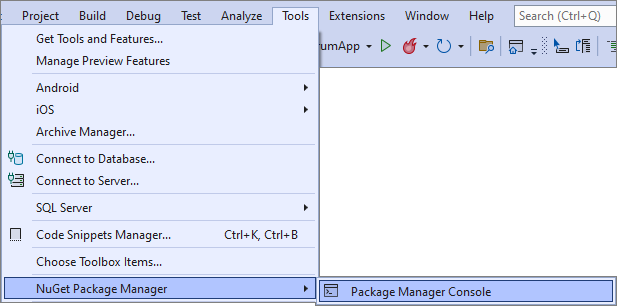
To do that, first we need to add the connection string in the "**appsettings.json"** file:



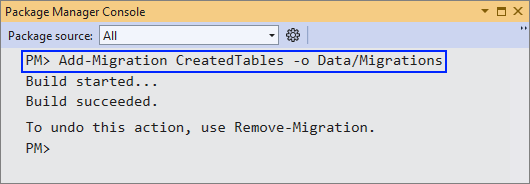
and set the **DbContext** to use SQL with the connection string in the **Program** class.



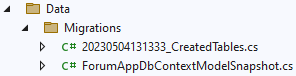
Next, open the Package Manager Console from [Tools] 🡪 [NuGet Package Manager] 🡪 [Package Manager Console] to write **commands** for **managing migrations**:

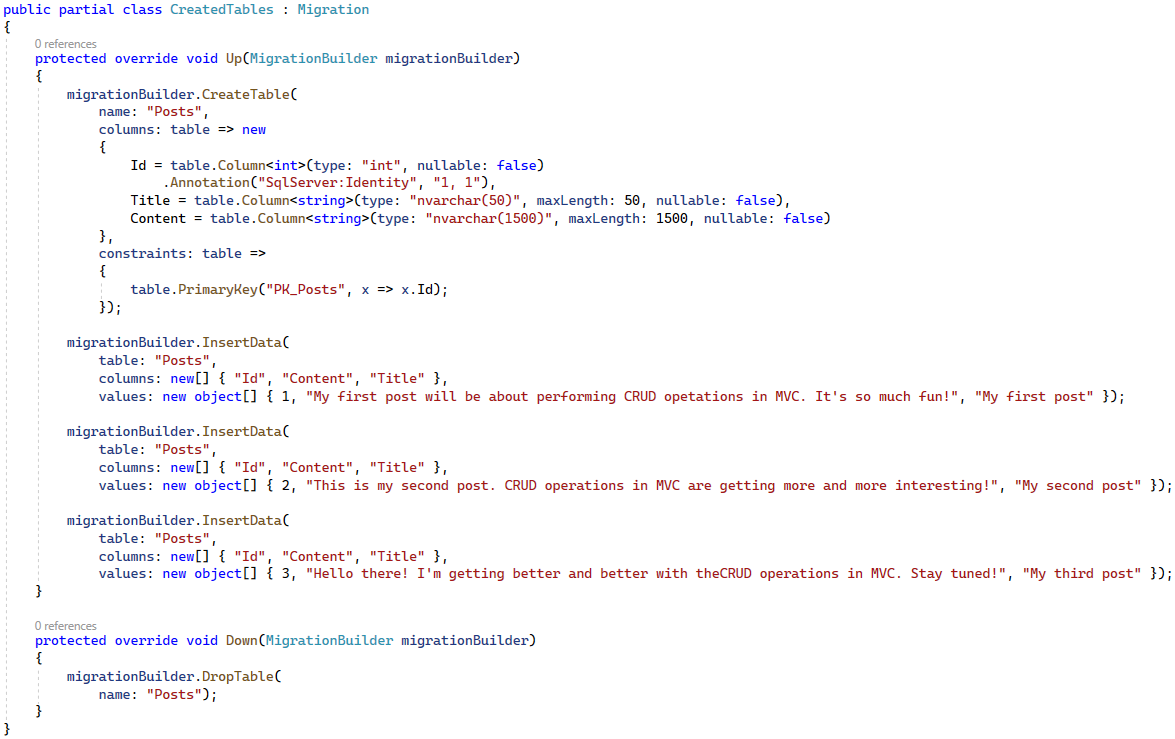


In the **console**, write a command for **adding a migration** to the "Data/Migrations" **folder** with a given **name** and **press** [Enter] to **execute it**. The **result** should be the following:



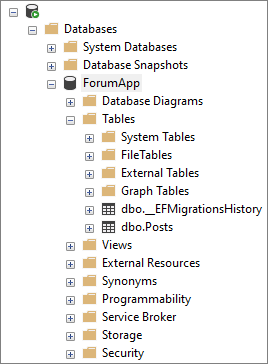
Now you should have a new migration in the "**Migrations**" folder:

 🡪

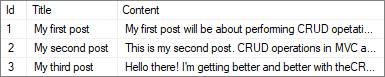


**Examine the tables** and its **restrictions** in the **new migration** – if something is wrong, **delete the migration** with the "Remove-Migration" **command** or **delete the migration file**. Don't forget that you should also **delete the database** from SQL Server Management Studio, or errors will appear.

Now **run the app** in the browser – there should not be **any errors**. Then, look at the **newly-created database** in SQL Server Management Studio and **examine the table** – it should be **present** and have the **right restrictions** and **relationships**:



Look at the "**Posts**" table – it should contain the three posts that we seeded:



## CRUD operations

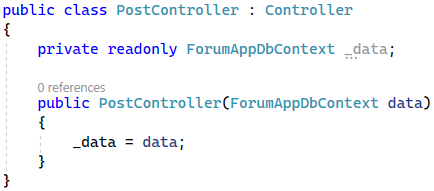
After we have populated our database, we can start executing CRUD operations.

### Reading Data

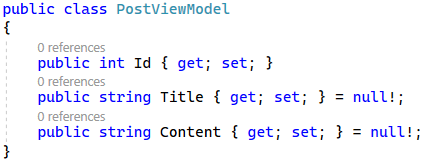
First, we need to create a new **PostController** in the "**Controllers**" folder".



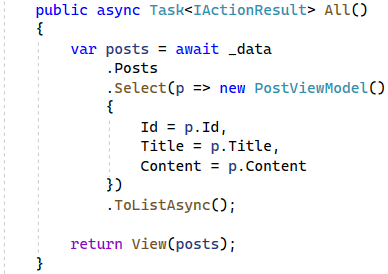
Let's inject the **ForumAppDbContext** through the constructor and assign it to a variable to use it:



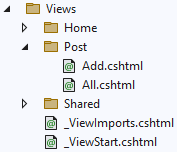
Create a "**Post**" folder in the "**Models**" folder of the project. Our next step should be creating the **PostViewModel** in the **"Models/Post"** folder:

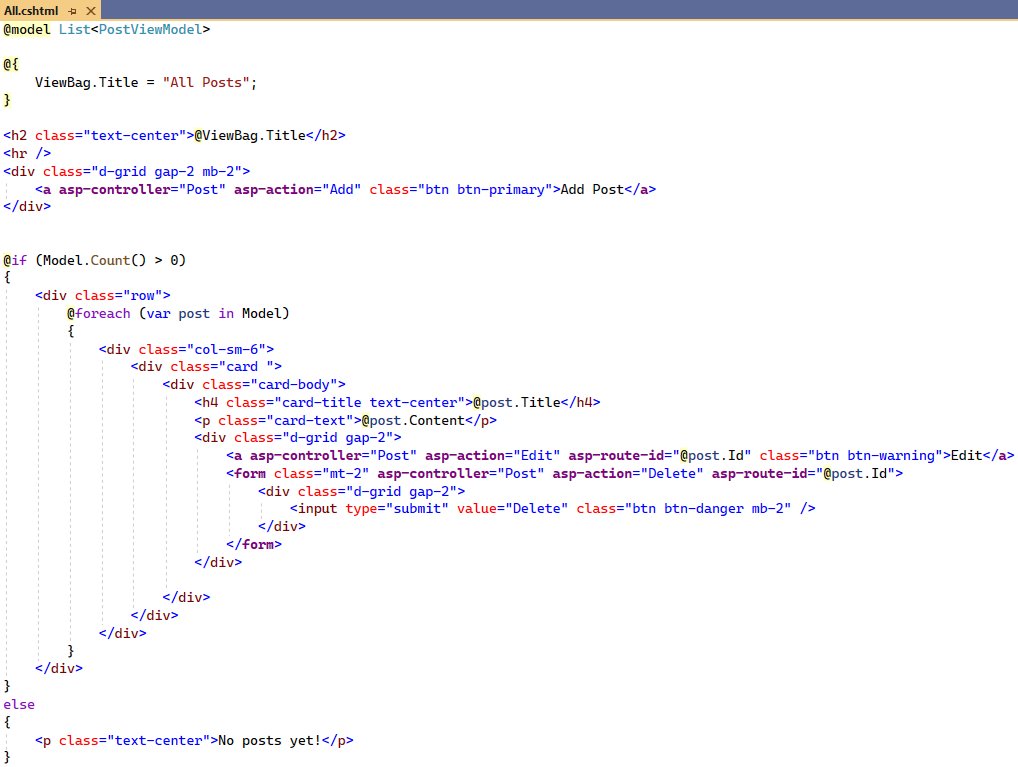


After the **PostViewModel** is created, we'll go back to the **ProductController** and add a new method **All()**. The task of this method is to extract the products from the database to a model collection, which will be passed to a view.



Now we should create the "**All.csthml"** view file. We will create all of the views in a new folder, called "**Post**", which will be located in the "**Views"** folder.

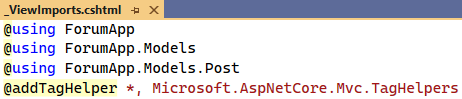
 →



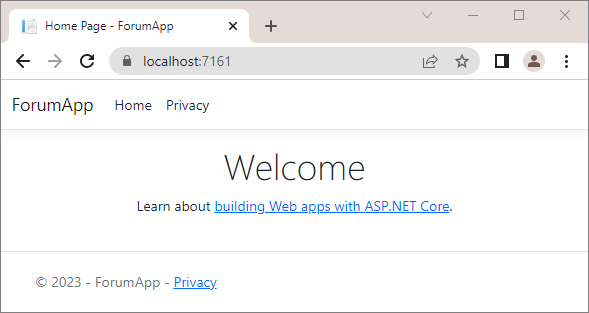
As this file contains more code, you can copy it from here:

|  |
| --- |
| @model List<PostViewModel>  @{  ViewBag.Title = "All Posts";  }  <h2 class="text-center">@ViewBag.Title</h2>  <hr />  <div class="d-grid gap-2 mb-2">  <**a** **asp-controller**="Posts" **asp-action**="Add" class="btn btn-primary">Add Post</**a**>  </div>  @if (Model.Count() > 0)  {  <div class="row">  @foreach (var post in Model)  {  <div class="col-sm-6">  <div class="card ">  <div class="card-body">  <h4 class="card-title text-center">@post.Title</h4>  <p class="card-text">@post.Content</p>  <div class="d-grid gap-2">  <**a** **asp-controller**="Post" **asp-action**="Edit" **asp-route-id**="@post.Id" class="btn btn-warning">Edit</**a**>  <**form** class="mt-2" **asp-controller**="Post" **asp-action**="Delete" **asp-route-id**="@post.Id">  <div class="d-grid gap-2">  <input type="submit" value="Delete" class="btn btn-danger mb-2" />  </div>  </**form**>  </div>    </div>  </div>  </div>  }  </div>  }  else  {  <p class="text-center">No posts yet!</p>  } |

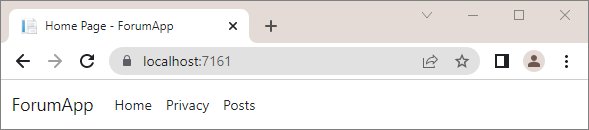
Don't forget that you have to add the "**Views/Post**" folder in the **\_ViewImports.cshtml** file in order for the application to be able to use the models in the "**/View/Post**" folder:



Now **run the app** in the browser – there should not be **any errors** and the **Home page** should be displayed:



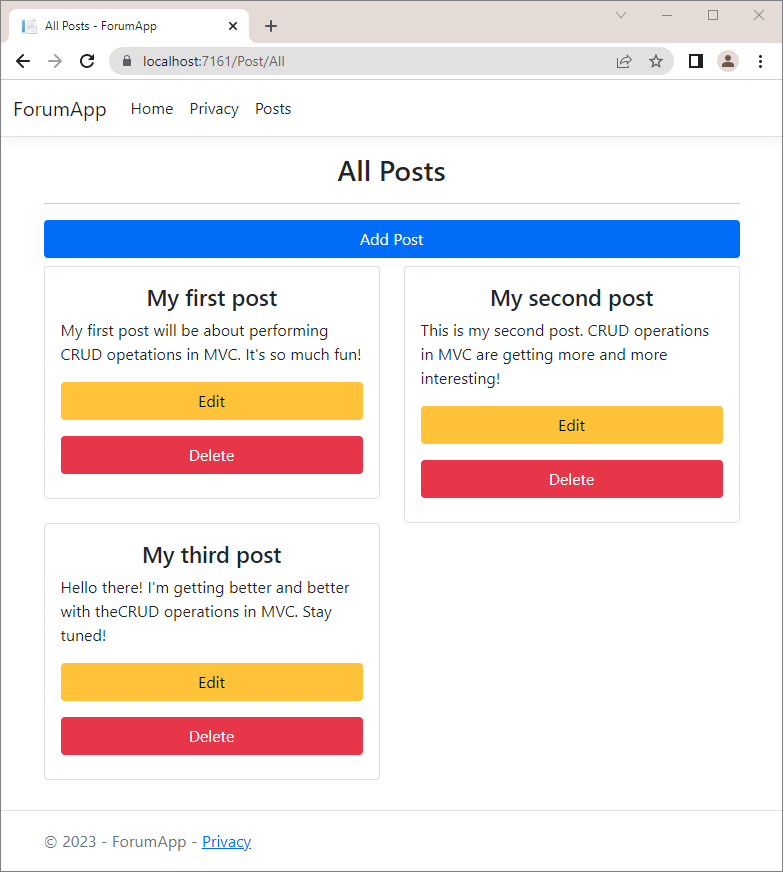
You can access the page that contains the collection with the posts on "/Posts/All". For our convenience, we will add a link to the navigation pane to access the Posts page. The navigation pane should look like this:

****

To **add links**, go the \_Layout.cshtml **partial view** in the "/Views/Shared" **folder**, as this view is responsible for the **common design** of all pages. Add the following lines:

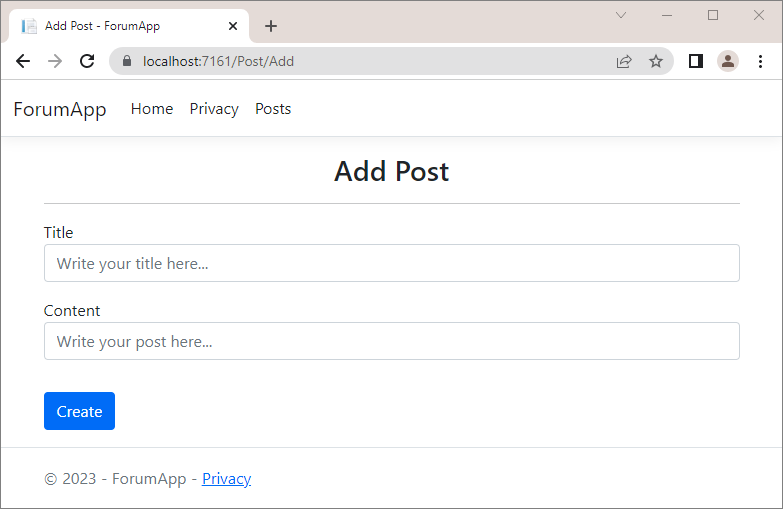
****

If we follow the newly added link, we should be able to see the Posts page and it should look like this:



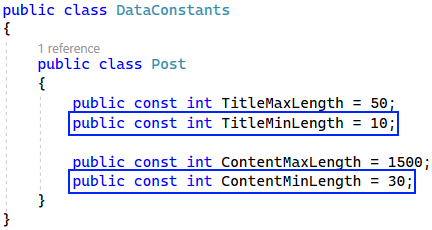
### Creating New Data

The next step in creating the **ForumApp** is adding the "**Add Post**" page. It will display a **form** for **adding a post** and it will look like this:

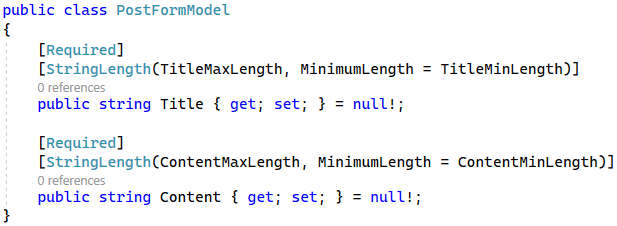


First, we need to create the **PostViewModel** in the **"Models"** folder.

Before that, let's go back to the **DataContants** class in the "**Data**" folder and add the values for the min length constants:



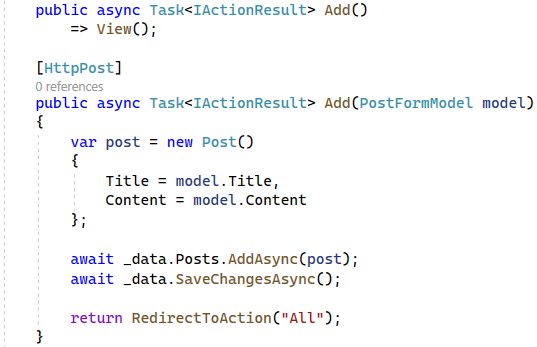
We will add validation attributes to the model properties of the **PostViewModel**. The **[Required]** attribute will check if the model property holds any value and the **[StringLength]** will check the length of the string that is held as a value.



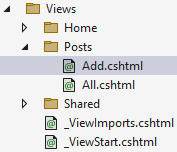
Don't forget to add the needed namespace to use the constant class in the **PostFormModel** class:



After we have created the **PostFormModel** we should go to the **PostController** and implement the **Add()** method. This method will create a new **Post** object and then add it to the **DbSet**.



Finally, we have to add a new "**Add.cshtml"** in the **"/Views/Posts"** folder for the **"Add Product"** Page view:



The file should look like this:



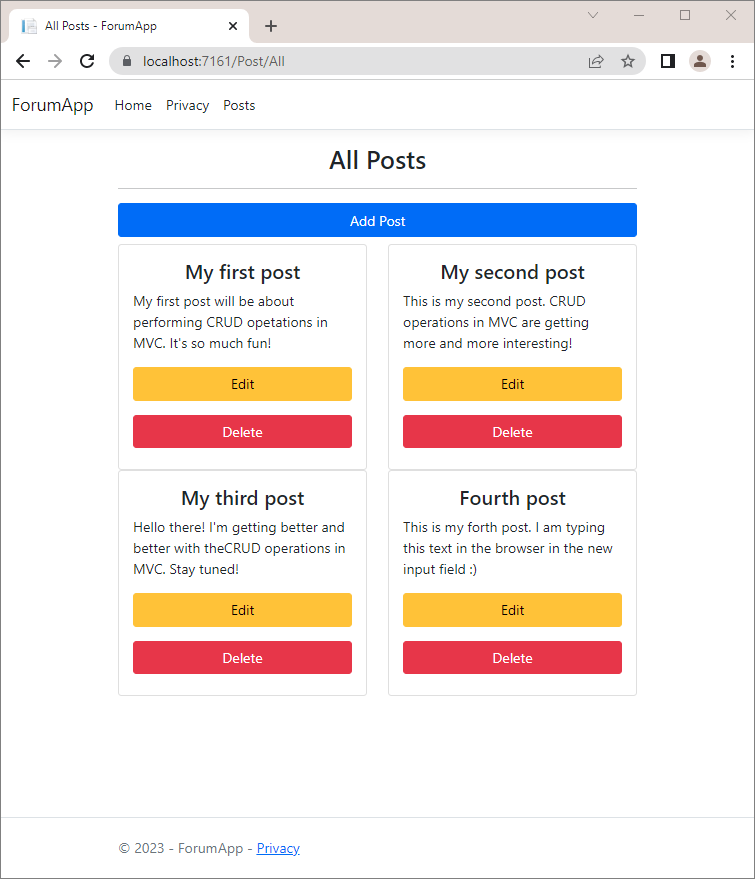
You can copy the code from here:

|  |
| --- |
| @model PostFormModel  @{  ViewBag.Title = "Add Post";  }  <h2 class="text-center">@ViewBag.Title</h2>  <hr />  <div class="row">  <**form** method="post">  <div class="form-group">  <div class="mb-3">  <**label** **asp-for**="Title"></**label**>  <**input** **asp-for**="Title" class="form-control" placeholder="Write your title here...">  <**span** **asp-validation-for**="@Model.Title" class="small text-danger"></**span**>  </div>  <div class="mb-3">  <**label** **asp-for**="Content"></**label**>  <**input** **asp-for**="Content" class="form-control" placeholder="Write your post here...">  <**span** **asp-validation-for**="@Model.Content" class="small text-danger"></**span**>  </div>  </div>  <input class="btn btn-primary mt-3" type="submit" value="Create" />  </**form**>  </div> |

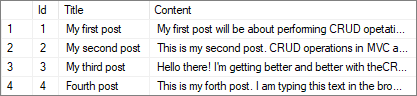
Let's not forget to add the following code in the **Add.cshtml** file in order for the app to be able to validate the input data.



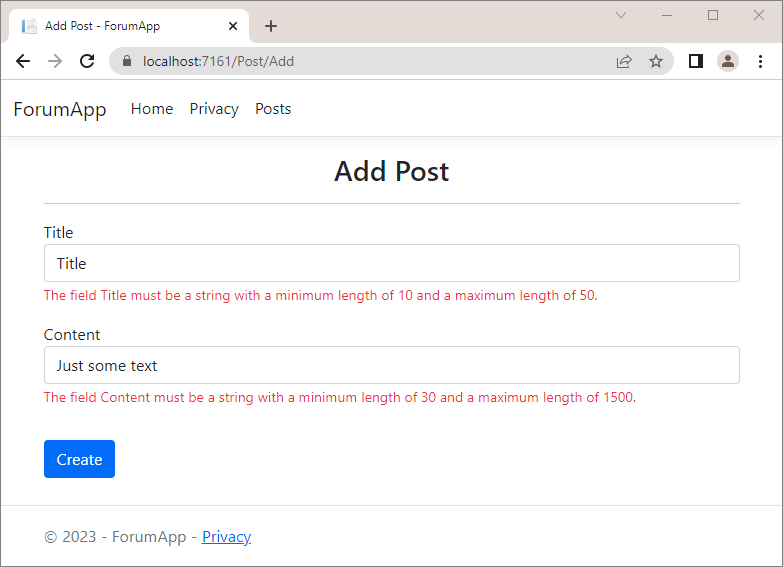
Now we can run the app in the browser and add a new post. After adding the new post, the **All Posts** page should look like this:



Now let's check the database in SQL Server Management Studio and **examine the "Posts" table** – it should **contain the new post that we just created:**

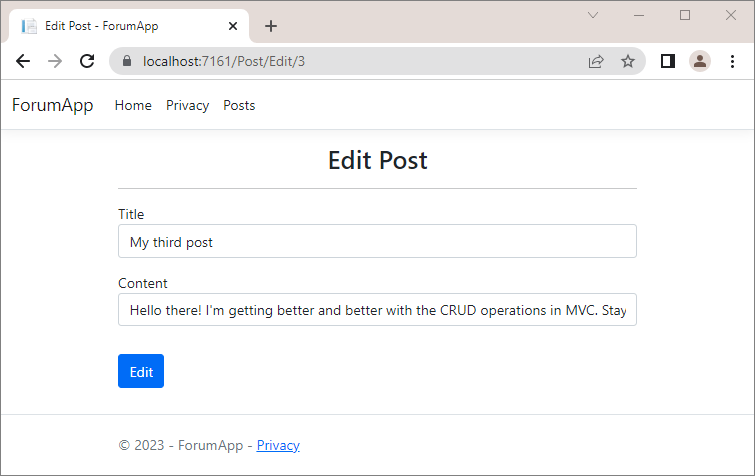


We can check if the app validates the title and content, which we want to add for our post. If their length doesn't meet the requirements, the app should look like this:

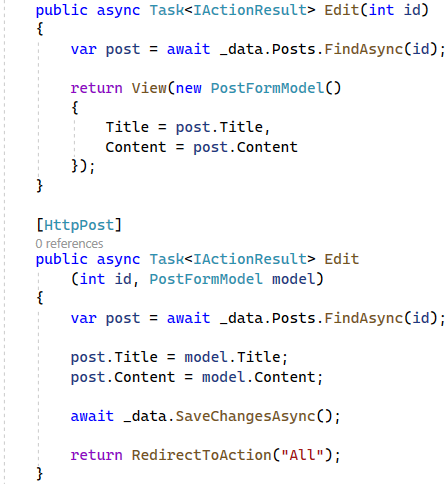


### Updating Existing Data

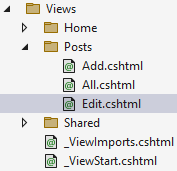
The next step in creating the **ForumApp** is adding the "**Edit Post**" page. It will display a **form** for **editing a post** and it will look like this:



Now let's edit one of the posts that we already have in the collection. First, we need to modify the **ProductsController** and add an **Edit()** method, which will pass a **Post** model to the view and find and update the post in the database.



Our next step is to add a new "**Edit.cshtml"** file to display the form for editing posts.



Let's not forget that we have to add validation for the length of the post's title and content.



You can copy the code from here:

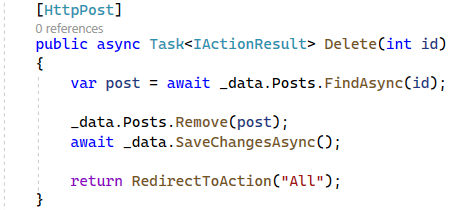
|  |
| --- |
| @model PostFormModel  @{  ViewBag.Title = "Edit Post";  }  <h2 class="text-center">@ViewBag.Title</h2>  <hr />  <div class="row">  <**form** method="post">  <div class="form-group">  <div class="mb-3">  <**label** **asp-for**="Title"></**label**>  <**input** **asp-for**="Title" class="form-control" placeholder="Write your title here...">  <**span** **asp-validation-for**="@Model.Title" class="small text-danger"></**span**>  </div>  <div class="mb-3">  <**label** **asp-for**="Content"></**label**>  <**input** **asp-for**="Content" class="form-control" placeholder="Write your post here...">  <**span** **asp-validation-for**="@Model.Content" class="small text-danger"></**span**>  </div>  </div>  <input class="btn btn-primary mt-3" type="submit" value="Edit" />  </**form**>  </div>  @section Scripts {  <**partial** **name**="\_ValidationScriptsPartial" />  } |

Now run the app in the browser and test it. You can also examine the "**Posts**" table in the SQL Server Management Studio.

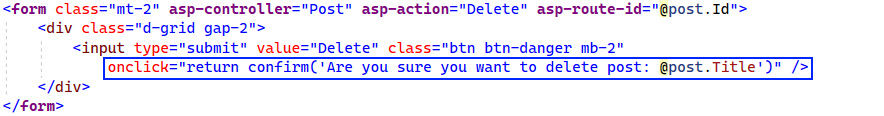
### Deleting Existing Data

The final step in creating our **ForumApp** is adding the functionality to perform a **Delete** operation. This means that when we click on the [**Delete]** button of a post, it has to send a **POST** reques to the controller. It's a good practice to add a client-side delete confirmation, so we will do that, too.

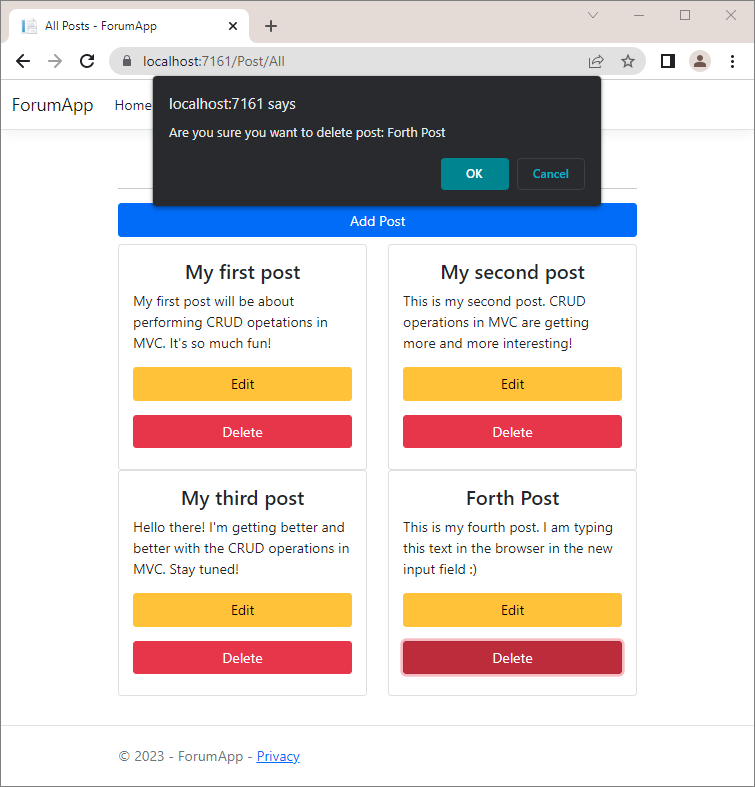
In order to do that, we need to add a new method **Delete()** in the **PostController**:



Next, we need to go to the "**All.cshtml**" file and add the following code:



The **onclick attribute** ofthe **[Delete]** button enables the JavaScript **confirm()** function, which displays the confirmation dialog to the user. If the [**Cancel**] button is clicked, the **confirm()** function returns **false** and nothing happens. If the [**OK**] button is clicked, the form is submitted and the post is deleted from the database.



Now run the app in the browser and test it. You can also examine the "**Posts**" table in the SQL Server Management Studio.