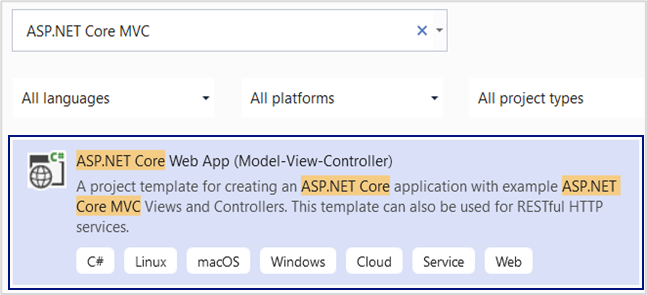
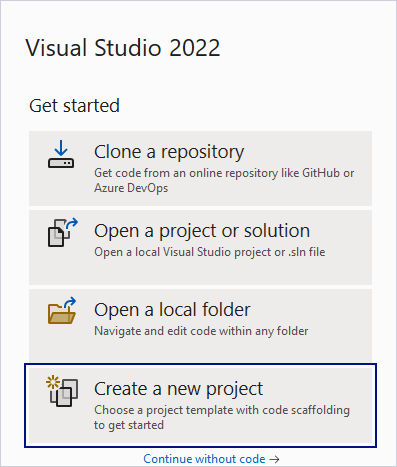
# Exercises: ASP.NET Core Introduction

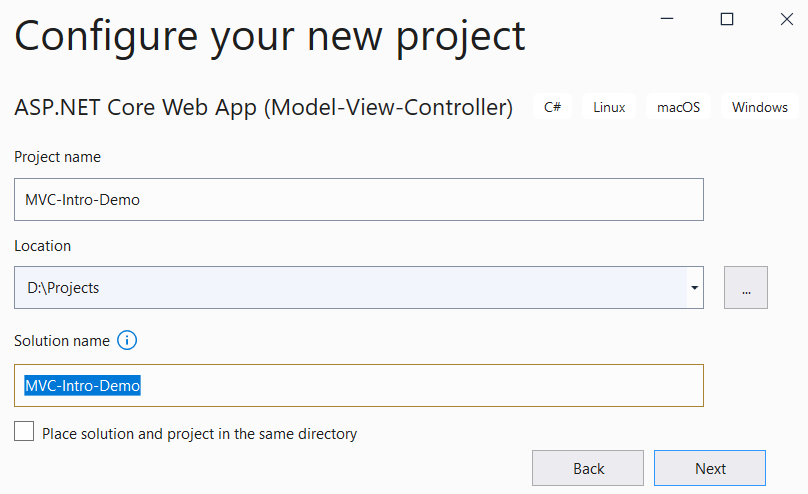
Problems for exercises for the ["ASP.NET Core Fundamentals" course @ SoftUni](https://softuni.bg/trainings/3853/asp-net-fundamentals-september-2022)

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

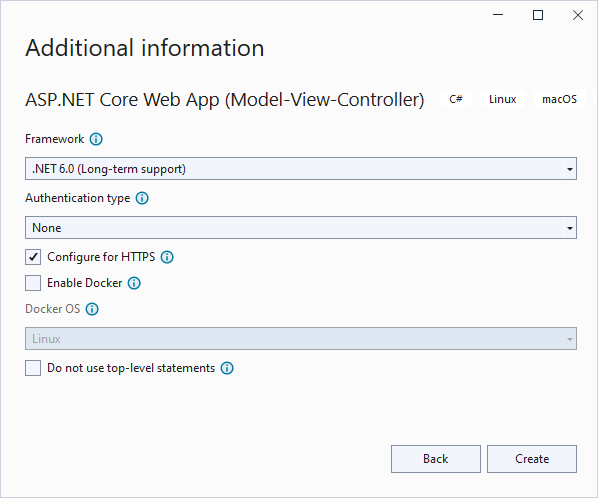
In this task you should implement the pages from the demo from the slides for this topic. To do so, create a **new ASP.NET Core MVC app**. Open **Visual Studio** and **choose** [Create a new project]. On the next step, **choose** [ASP.NET Core Web App (Model-View-Controller)] as a **project template**.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

****

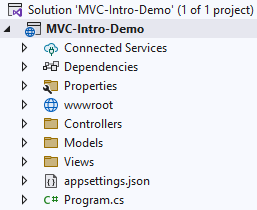
Give a **name** to your project and solution:



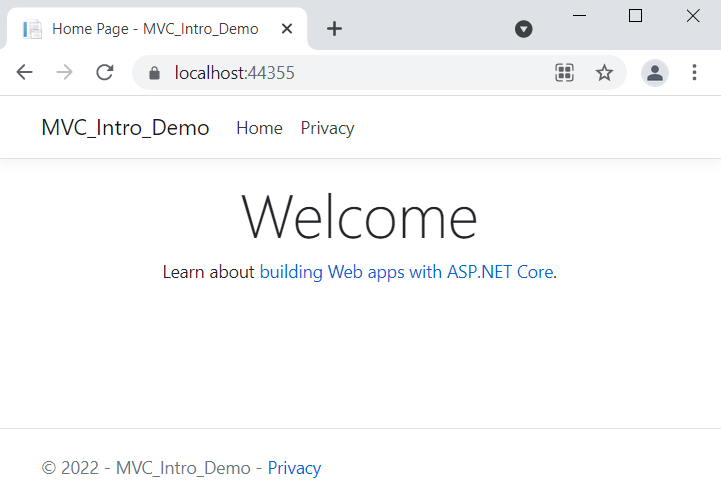
On the next step you should **choose** your target frameworok and click on the [Create] **button**:



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:



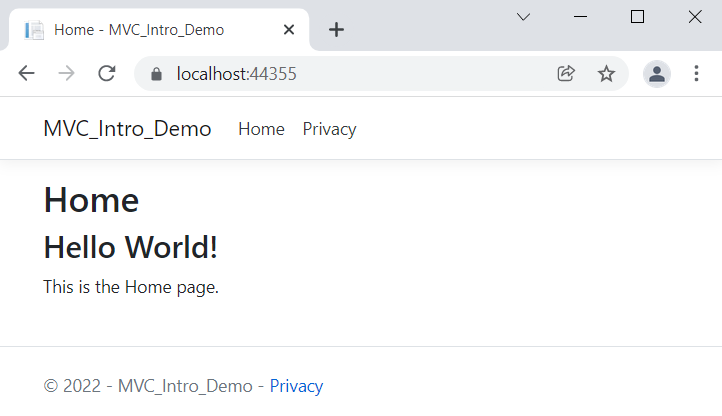
If you **run the app**, you will see the **default** "Home" **page**, which is served by the HomeController in the app:



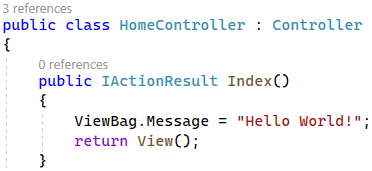
### HomeController Pages

#### Modify the "Home" Page

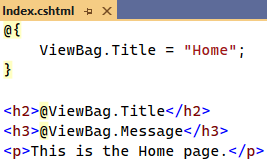
Now we want to **modify** the "Home" **page** to look like this:



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:



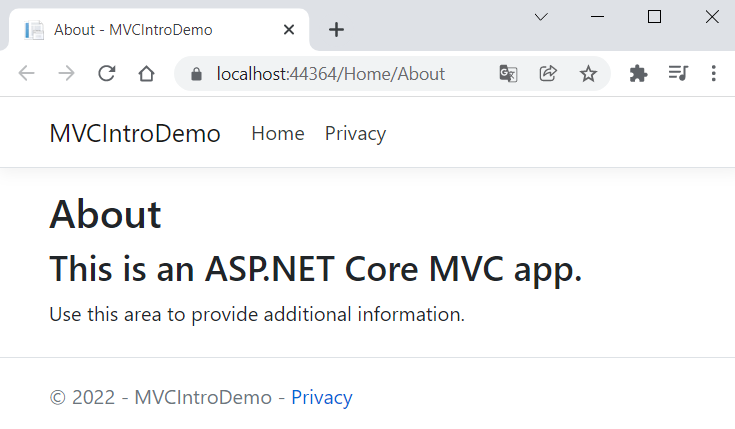
Now you should modify the Index.cshtml **view** in the "/Views/Home" **folder** to **display the page differently**. Use the ViewBag **class** to **get the message** from the controller. Note how the Razor **views** able us to use **C# code** inside **HTML**:



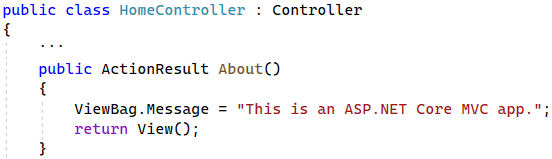
**Run the app** with [Ctrl] + [F5] and make sure the "Home" **page** looks as shown on the screenshot above.

#### Create the "About" Page

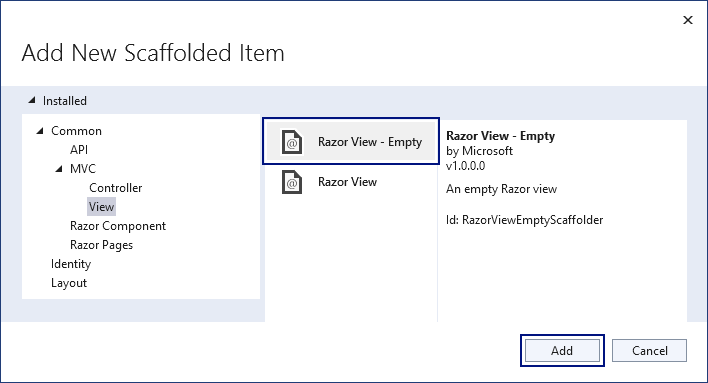
Create an "About" **page** in the app, which should look like this:

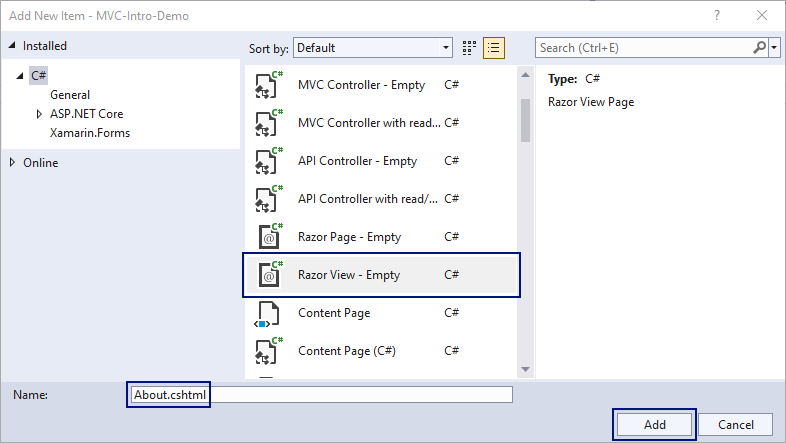


The page should be **accessed on "**/Home/About". Create an About() **controller action** in the HomeController **class** for the "About" **page**. The controller method should return a **view**. It should also use the ViewBag **class** to **pass a message** to the returned view. Write the method like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

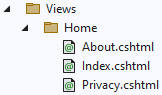
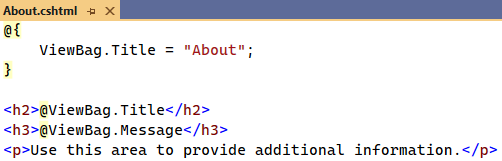


Now you should create an About.cshtml **view** in the "/Views/Home" **folder**. To do this, **right-click** on the folder and **choose** [Add] 🡪 [View]:





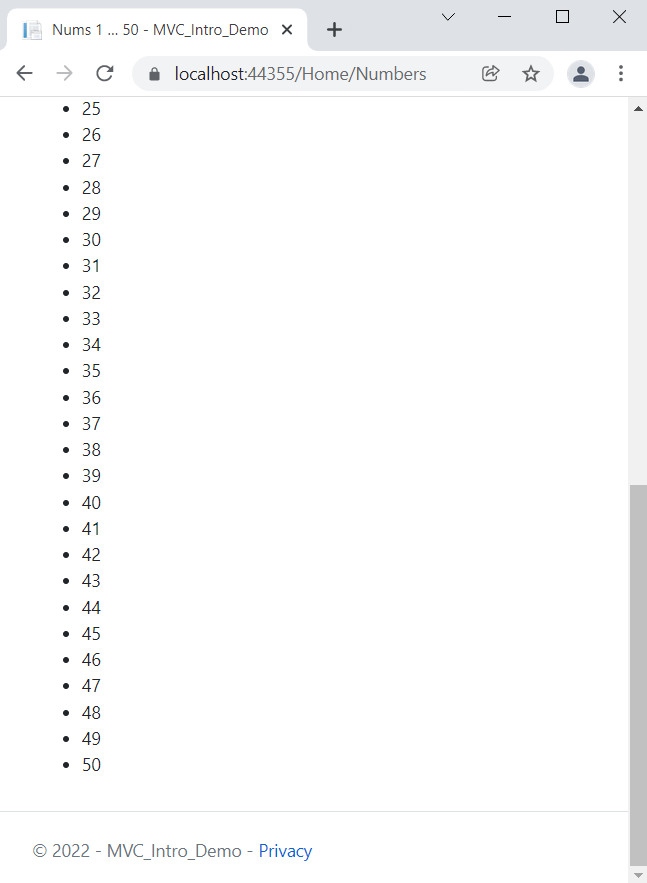
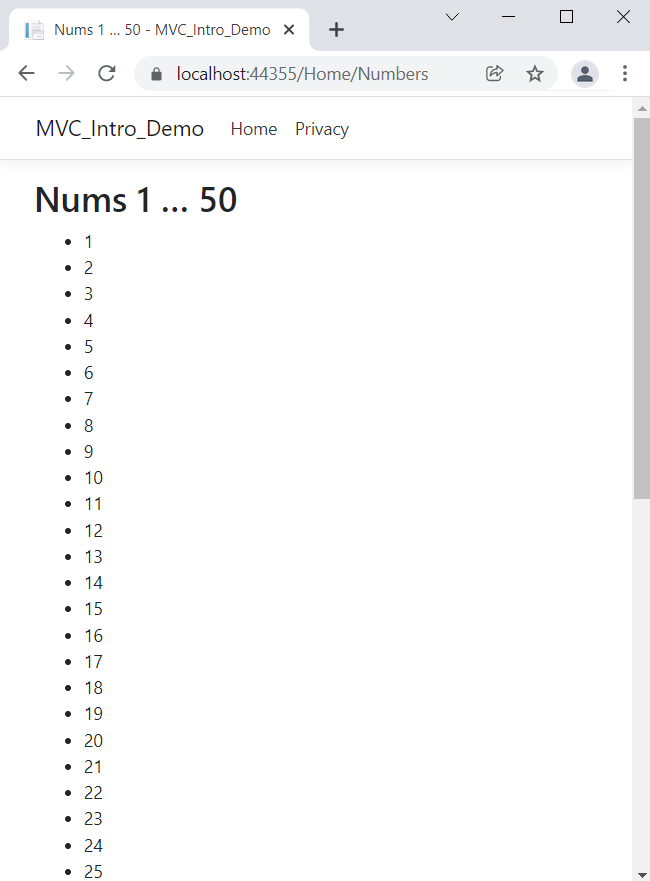
Now the About.cshtml **view** should be created. Write it like this:

 🡪 

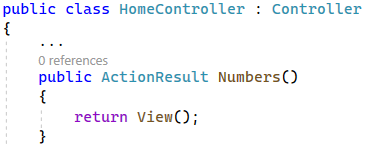
Look at the "About" **page** in the **browser**. You can access it on "/Home/About". It should look as shown above.

#### Create the "Numbers 1…50" Page

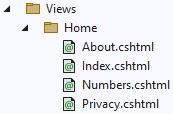
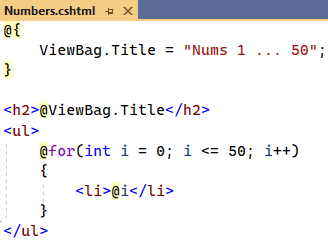
The "Numbers 1…50" page should **display the numbers from 1 to 50**. It should be accessed on "/Home/Numbers" and shoud look like this:



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



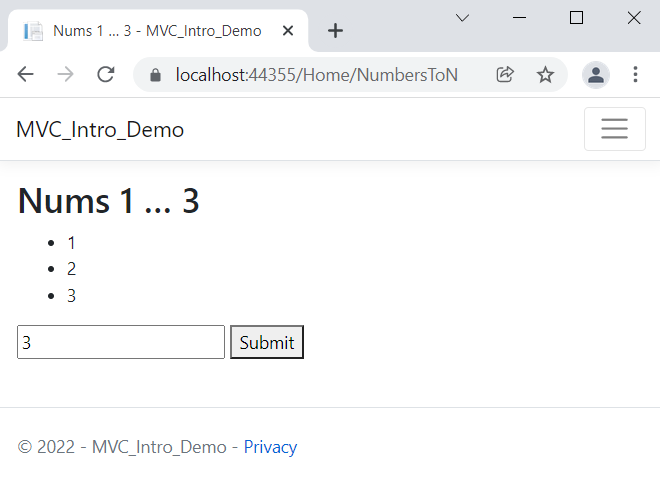
Create a Numbers.cshtml **view**, which should use a for **loop** to **display each number**. Write the **view** like this:

 🡪 

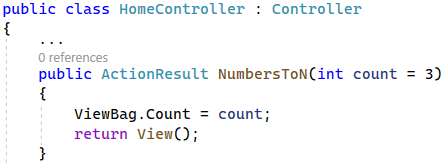
Make sure the **numbers from 1 to 50 are displayed** on the "Numbers 1…50" **page** on "/Home/Numbers".

#### Create the "Numbers 1…N" Page

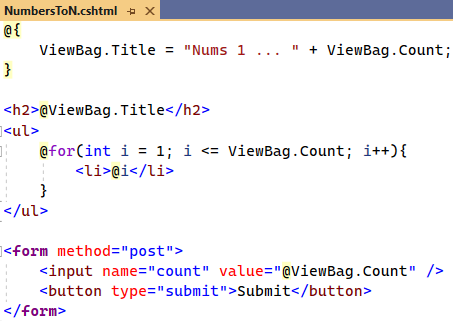
This page is similar to the previous one but the **user should enter a number** N. Then, only **numbers from 1 to** N should be displayed:



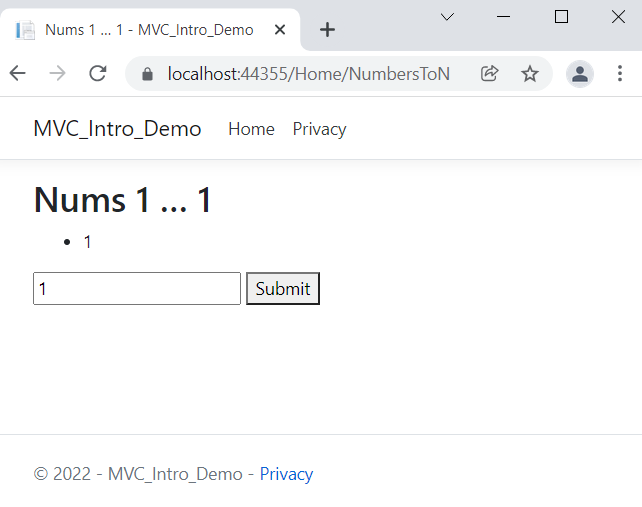
Write a NumbersToN() **method** in the HomeController. It should **accept a count parameter** from the **view** (with **default value** of the parameter 3). Then, it should **add the count number** to a ViewBag and **return a view**:



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:

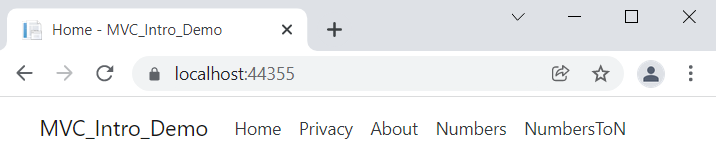


Try out the page in the browser on "/Home/NumbersToN". It should **display different numbers**, depending on the **count** you enter in the form:

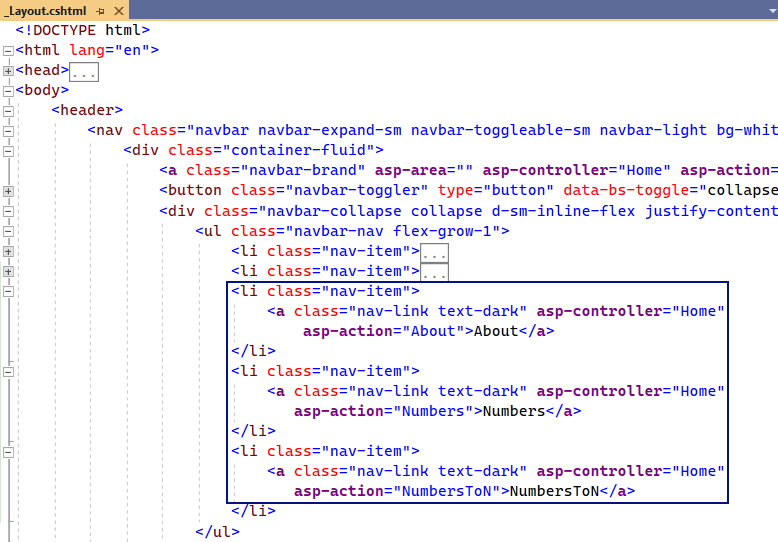


#### Add Navigation Links

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:

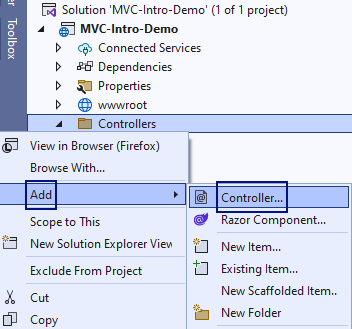


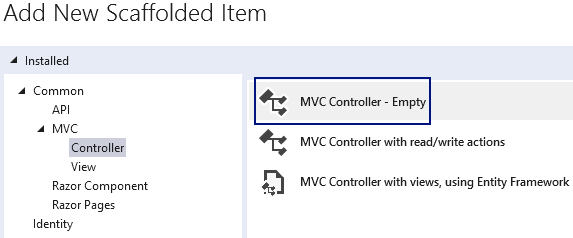
The asp-controller and asp-action **tag helpers** set the **controller** and **action names** of the page, which should be accessed.

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

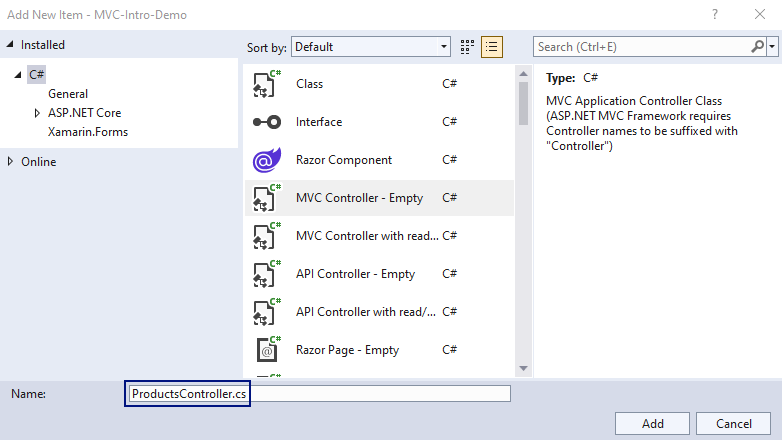
### ProductsController Pages

The ProductsController will have **controller actions for displaying hardcoded products on pages**. Create the ProductsController 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**:

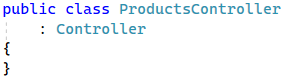
 🡪



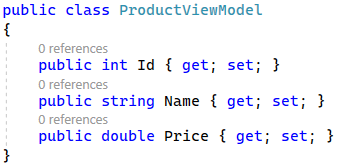
Set the **controller name** like this:



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 ProductViewModel class in the "Models" **folder** with the following **properties**:

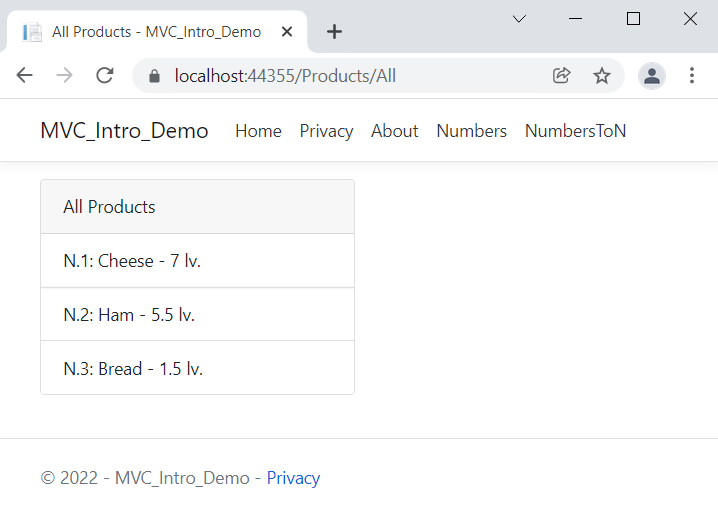


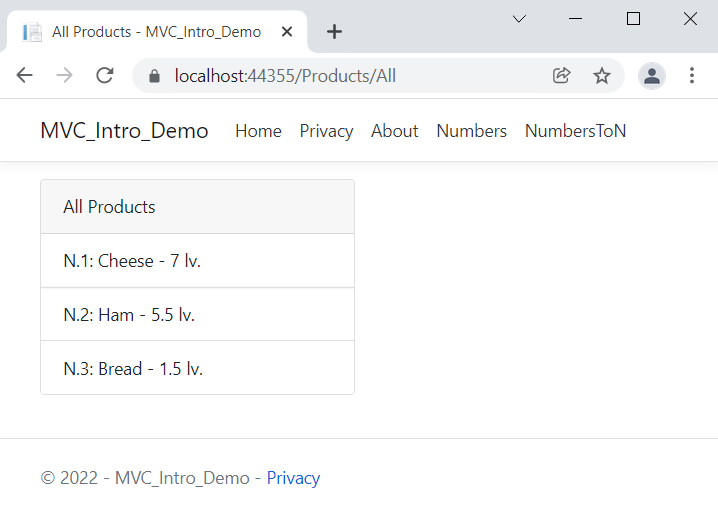
Now go back to the ProductsController and **add a field with products**. The field should have a **collection of** ProductViewModel with **three products** like this:



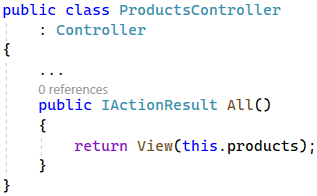
Now use these **products in controller methods**.

#### Create the "All Products" Page

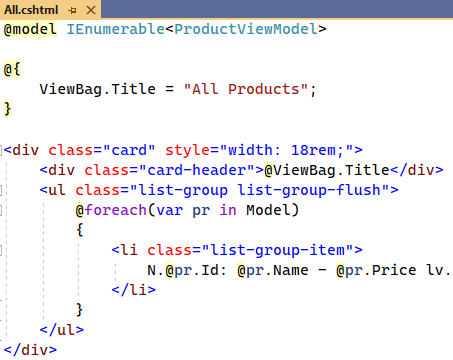




Create an All() **controller method** in the ProductsController, which should only **return a view** with the **products collection**:

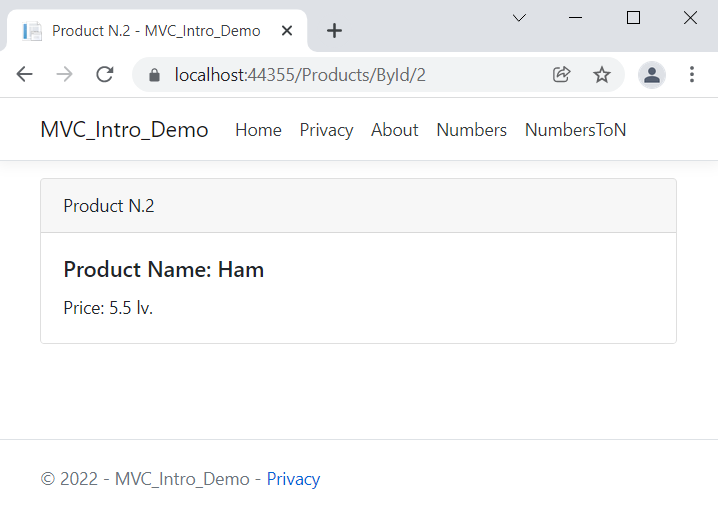


Now you should **create a** "Products" **folder** in the "Views" **folder**, which will have **all views for the** ProductsController **methods**. In it, add an All.cshtml **view**, which should **accept a collection of** ProductViewModel. Then, **foreach the products** and **use the model properties**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

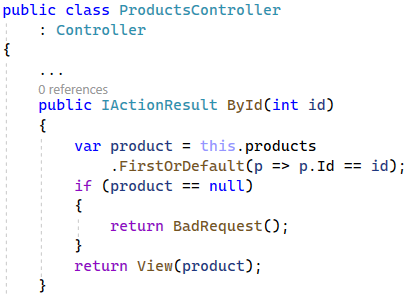


Try the "All Products" **page** on "/Products/All" in the browser.

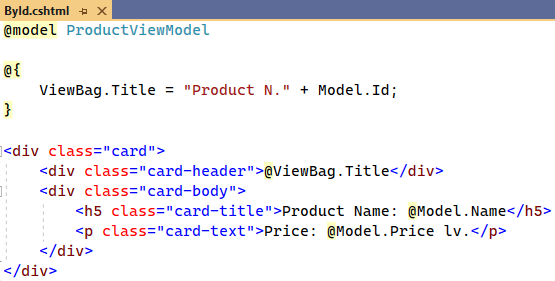
#### Create the "Product By Id" Page



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



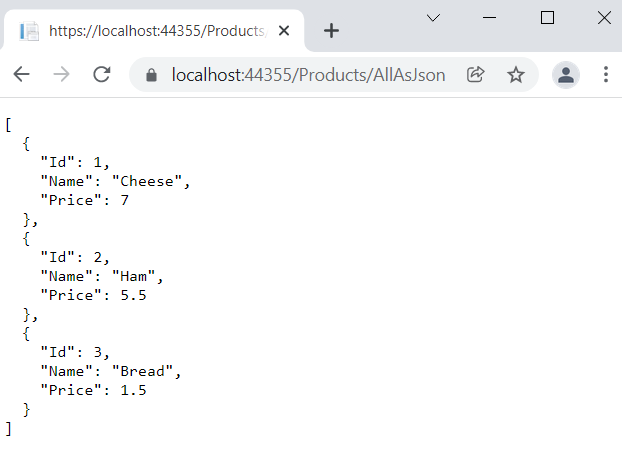
The ById.cshtml **view** is the following:

 🡪

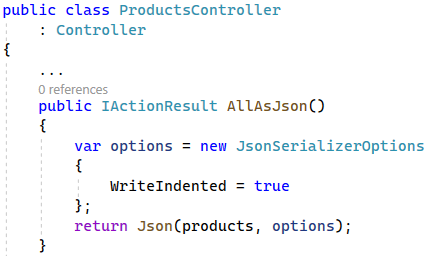
Go to the "Page By Id" **page** on "/Products/ById/{id}" with a **valid** and an **invalid** **product** **id**.

#### Return Products as JSON

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



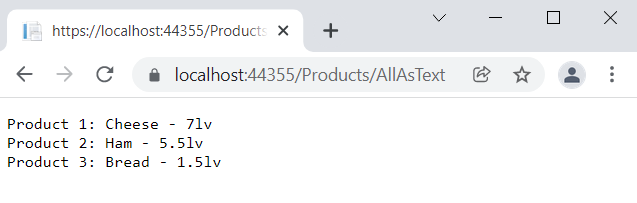
Create the AllAsJson() method in the HomeController, which should return a **JSON with the products** as shown below. It should use **JSON options** to display the JSON **indented**:



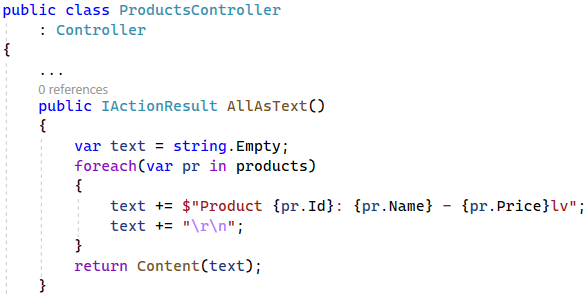
Try the page in the browser and make sure that **products are displayed correctly** as **JSON**.

#### Return Products as Plain Text

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



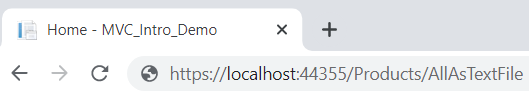
The AllAsText() **method** in the ProductsController should **create a** string **of all products** and return it as a **plain text response**:

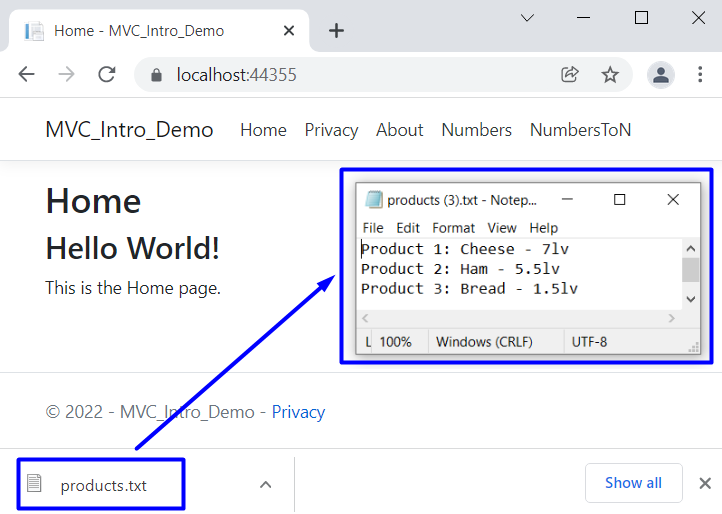


Try it in the browser.

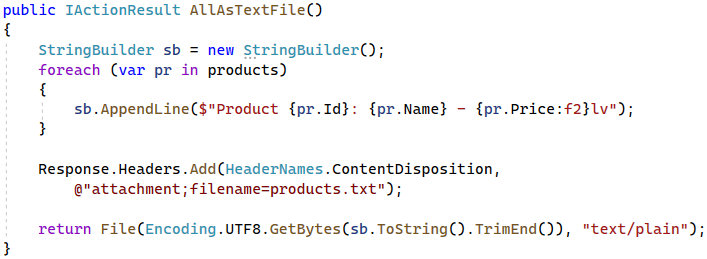
#### Download Products As Text File

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

 🡪

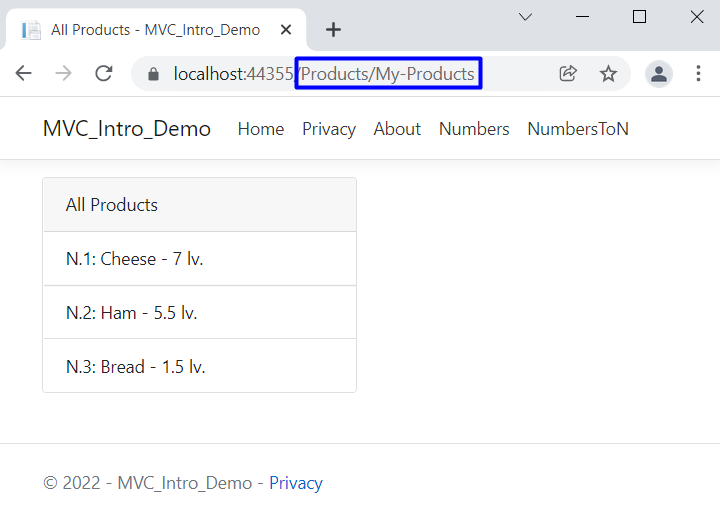


The AllAsTextFile() **method** in the ProductsController should form a **text with the products**. Then, it should **add the** Content-Disposition **header** to the Response, so that the **file is downloaded** as an **attachment**. At the end, it should **return a file with the text as a byte array**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

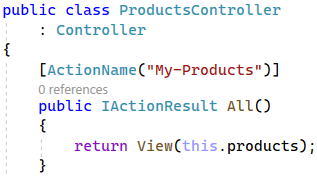


#### Access the "All Products" Page on Another URL

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



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

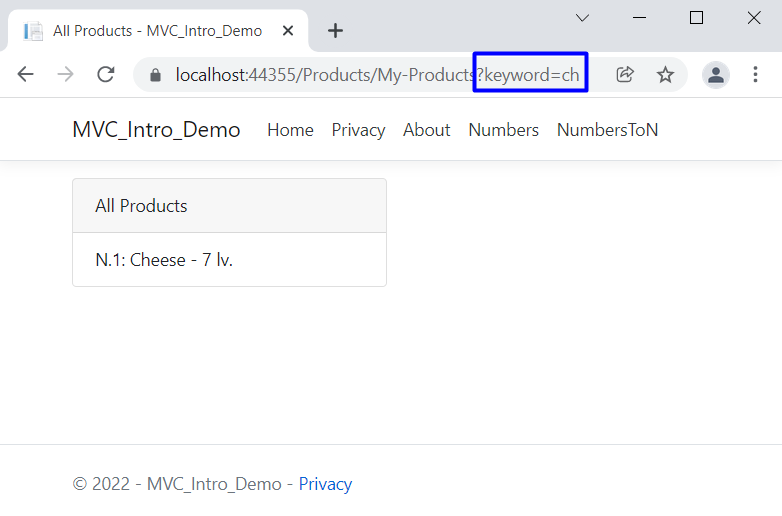


You should also **change** the All.cshtml **view** **name** to "My-Products.cshtml", as the **view** and the **controller** **action** should have the **same** **names**:

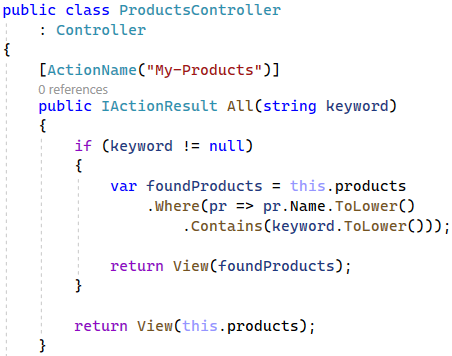


#### Add Search to the "All Products" Page

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

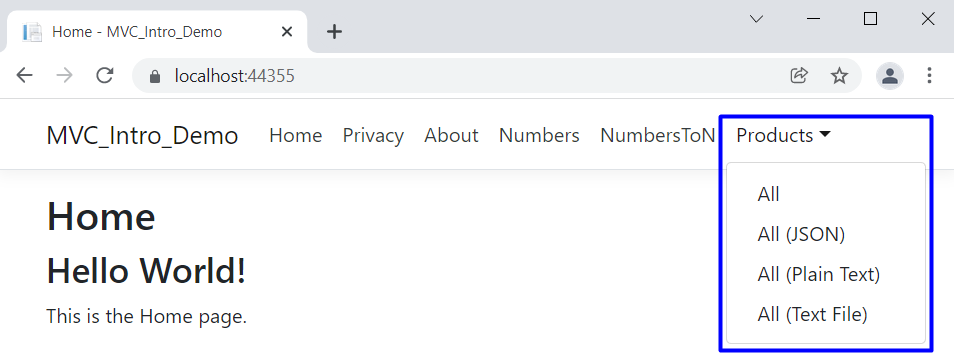


To do this, make the All() **controller action** **accept a keyword** and return only the **filtered products**, when there is a **keyword** in the **URL**:

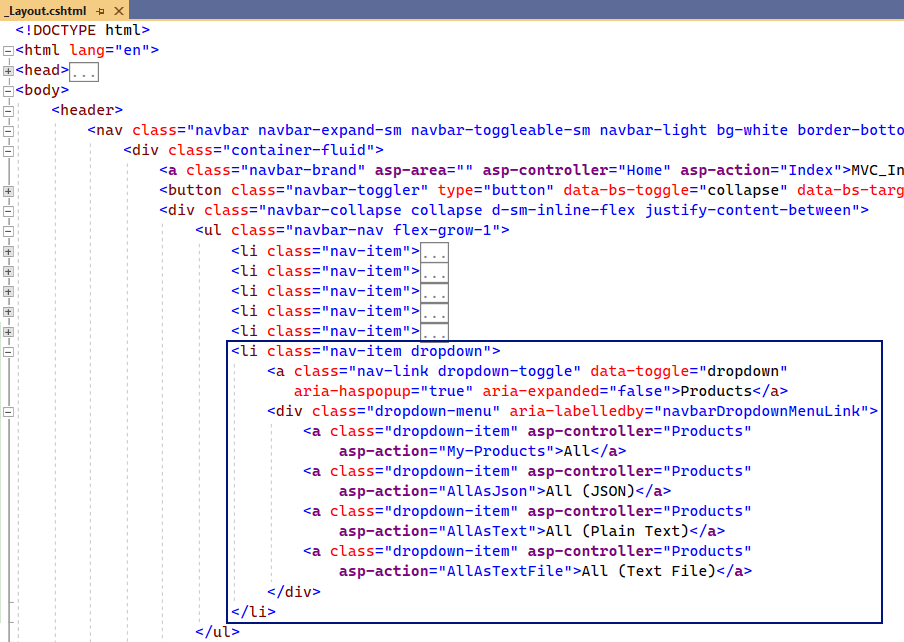


Enter **different keywords** on "/Products/My-Products?keyword={keyword}" in the **browser** and make sure that only **products with the keyword in their name** are shown.

#### Add Navigation Links



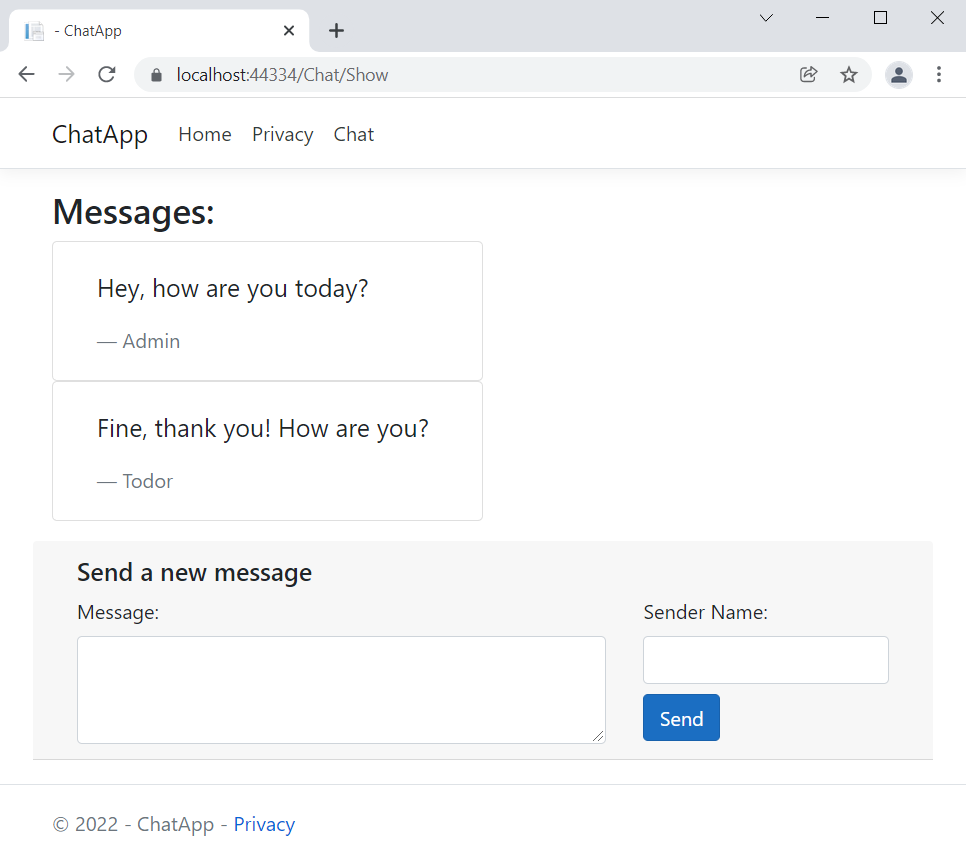
Modify the \_Layout.cshtml **view** like this to have the above **links**:



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

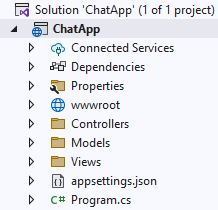
## Simple Chat ASP.NET Core MVC App

We will begin this exercise by creating a **simple ASP.NET Core MVC app** called "ChatApp". Our app will have a page for **displaying and adding chat messages**. It will look like this:



### Create the Project

First, **create the app and name it** "ChatApp":

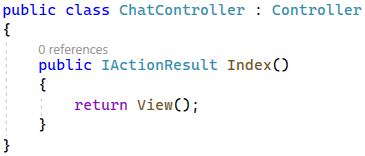


The **workflow** of the **chat** **functionality** in the app will be the following:

* A **controller action** passes the **current messages** (if any) to a **view** as a **model**
* The **view** **displays the messages** (if they exist). Also, the view displays a **form for creating a new message** and **passes a model to the controller** when the form is submitted
* Another **controller action** **accepts the model** and **adds a new message with the model data** to the other messages
* The **second method invokes the first one** by **redirection**, which again passes **all messages to the view** (including the new one)

### Create Controller and Models

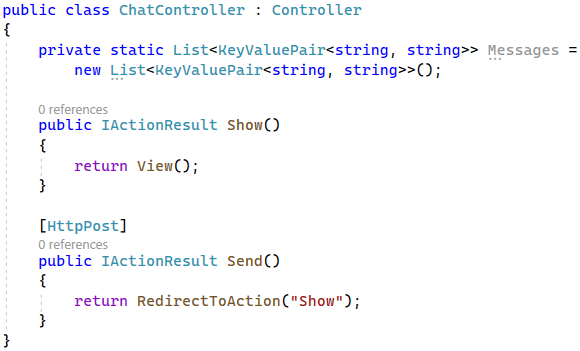
Create a ChatController **controller class** in the "Controllers" **folder**:

**Delete** the Index() **method**, as we will **create our own actions**. The ChatController should have:

* A **collection of messages**, which has the **message sender as key** and the **message text as value**
* A "GET" method Show(), which returns a **view with model** (the model may hold the **messages**)
* A "POST" method Send(), which **accepts a model** from the **view** and **adds a message to the collection**. Then, it **redirects** to the Show() action.

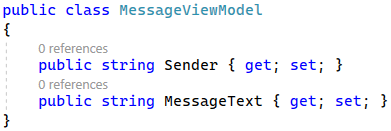
Write the above **class field** and **properties**:



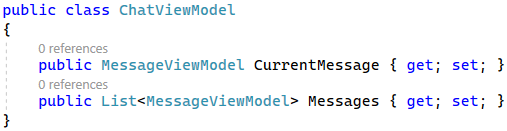
|  |  |
| --- | --- |
| Icon  Description automatically generated | **Warning**: the above code holds the shared app data in a **static field** in the controller class. This is just for the example, and it is generally a **bad practice**! Use a **database** or other **persistent storage** to hold data, which should survive between the app requests and should be shared between all app users. |

Note that the **message collection** is of type List<KeyValuePair<string, string>>, not Dictionary<string, string>, as it does **not allow duplicate keys**, but we may want to have **several messages** **by the same sender**.

Before we implement the Show() **method** of the ChatController, create the **needed models**, which will be passed to the **view**. In the "/Models" **folder**, create a MessageViewModel **class** (this is an ordinary class), which will hold **properties for each message** (**message sender** and **text**):

 → 

Then, create the ChatViewModel, which will be **passed to the view** and then **returned to the controller**. Write the ChatViewModel **class** like this:



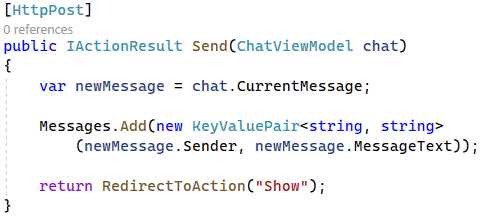
The Messages **property** has a **collection of messages** (the already created messages), which will be passed to and displayed by the **view**. Then, the user will **submit a form for creating a new message**, which will be saved to the CurrentMessage **property** and **passed to the controller**.

Now go to the ChatController and **implement the above logic**. Write the Show() **method** first. If the **messages** **collection** of the class is **empty**, the controller action should return a **view with an empty** ChatViewModel. If there are messages, a view with a ChatViewModel should be returned. This time, however, the Messages **collection** of the ChatViewModel should have the **messages as a collection of type** MessageViewModel.

Implement the **action** like this:

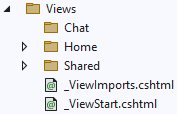


Now write the Send() **method**, as well. It should have the [HttpPost] **attribute**, which means that the action will be invoked on a "POST" **request** to "/Chat/Send". The method should also **accept a** ChatViewModel (from the **view**) and use its CurrentMessage **property** **values** to **add a new message** to the message collection. Finally, it should **redirect** to the Show() **action**. Do it like this:

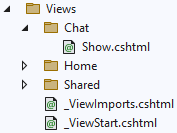
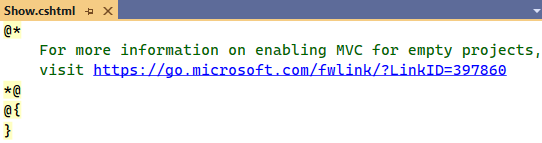


### Create a View

Finally, we should create a Show.cshtml **view**. First, create a **new folder** "Chat" (the name of the **controller**) in the "/Views" **folder**:



In it, create a Show.cshtml **view**:

 🡪 

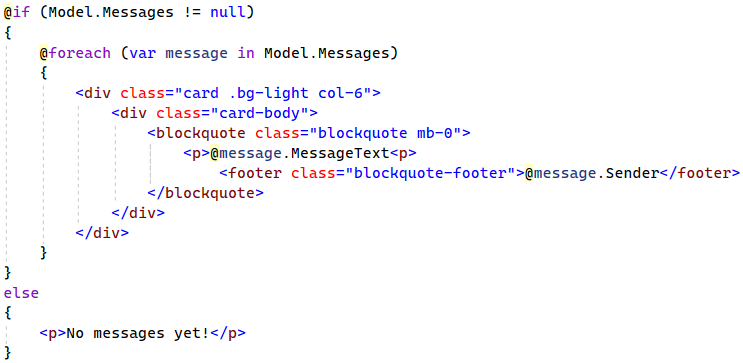
**Clear the view file** and let's write our own code. First, use the @model **directive** to make the **view accept** a ChatViewModel:



Add a **heading** to the view with a pure **HTML** like this:



Next, we want to **show each message with its sender and text** if the ChatView model has any. Otherwise, we should just display the "**No** **messages** **yet!**" **message**. To do this, use an if **statement** and a foreach **loop** in the **Razor view**. Also, use the @ **symbol** to switch to C# **code** and **use the model properties**. Do it like this:



Then, create a **form**, which should send a "POST" **request** to "/Chat/Send" and **fill in the** CurrentMessage **property** of the ChatViewModel. Use **different tag helpers** (will be examined during the next topics) to **set the controller** and **action** and to **extract the name of a specified model property into the rendered HTML**. Write the rest of the view code like this:



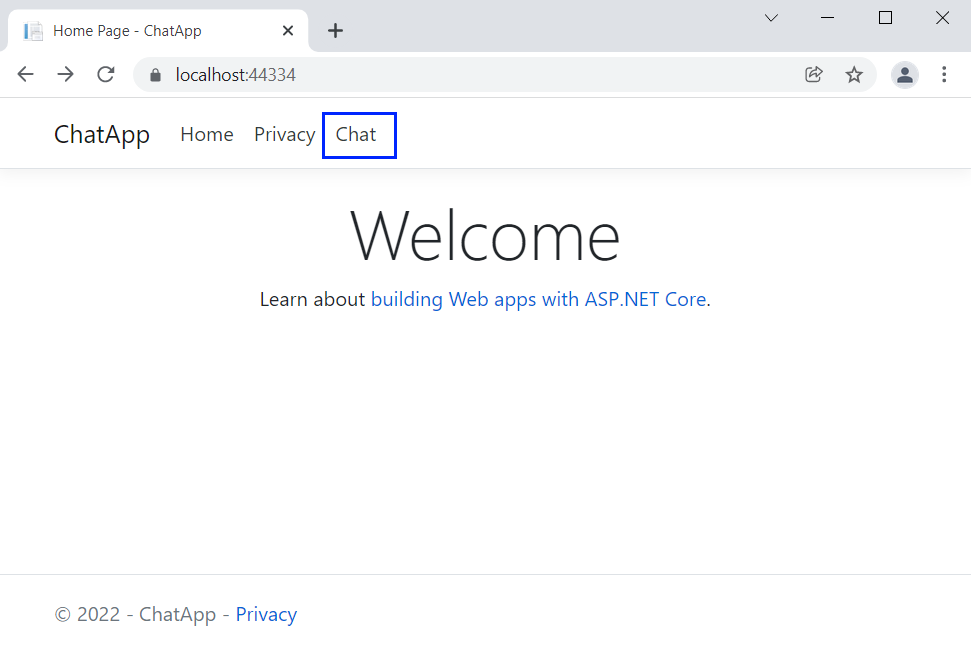
Now if we access "/Chat/Show" we will see the Show.cshtml **view**.

To **add a link** to the page, go to the \_Layout.cshtml.cshtml **view** in "/Views/Shared" and add the following code:

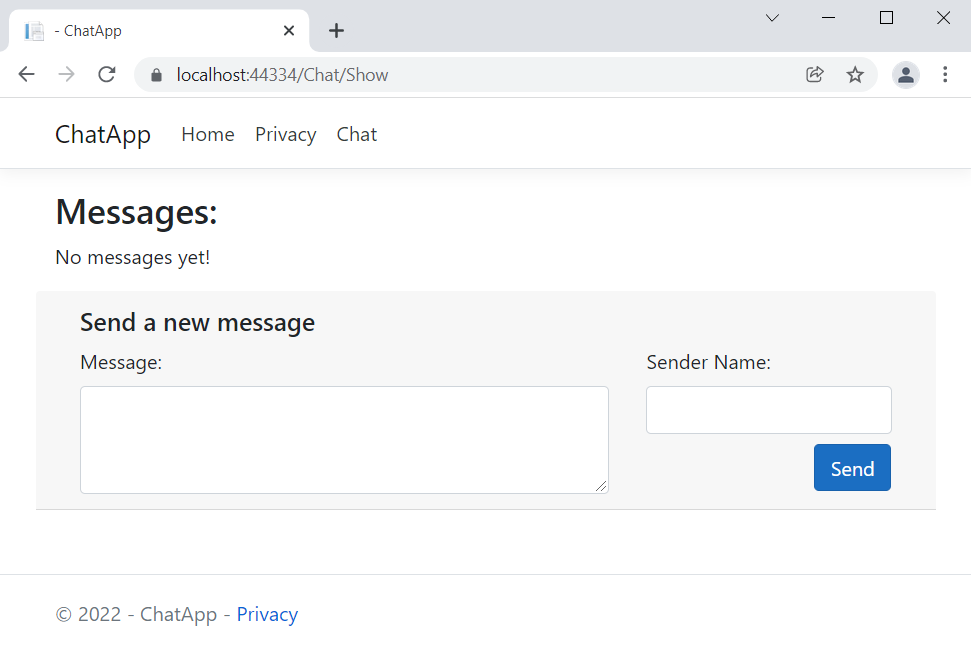


### Try the App

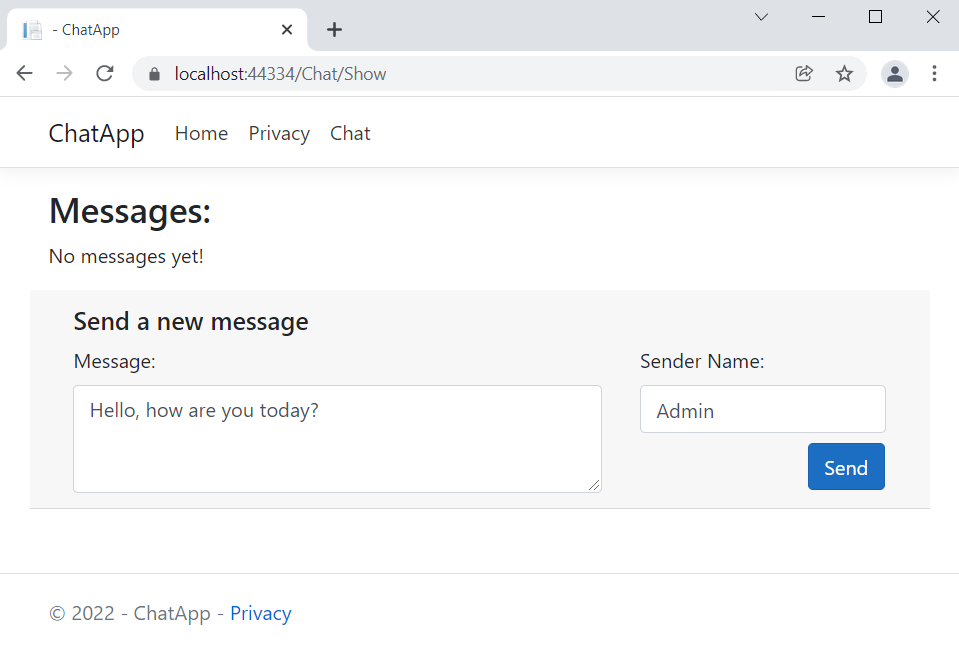
**Run the app** and examine it in the **browser**. It should have the "Chat" **navigation link**, which we have just added:

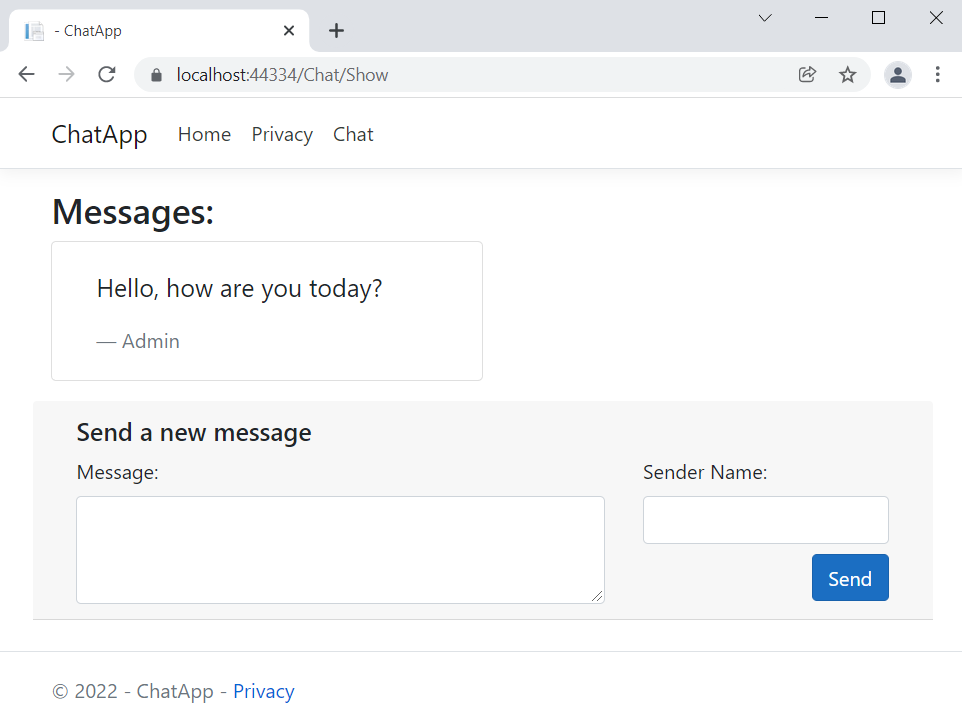


Click on the [Chat] **link**. You should be **redirected** to "/Chat/Show" and see the Show.cshtml **view**:



We have **no messages yet**, so let's **add** one. **Fill in the form** and **click** on the [Send] **button**. The **new message** should be **displayed on the page**:

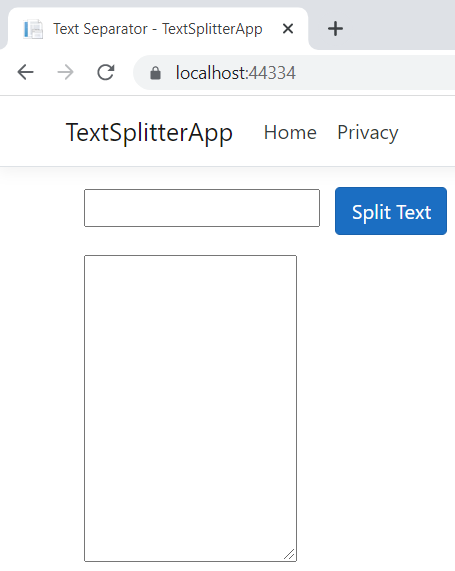
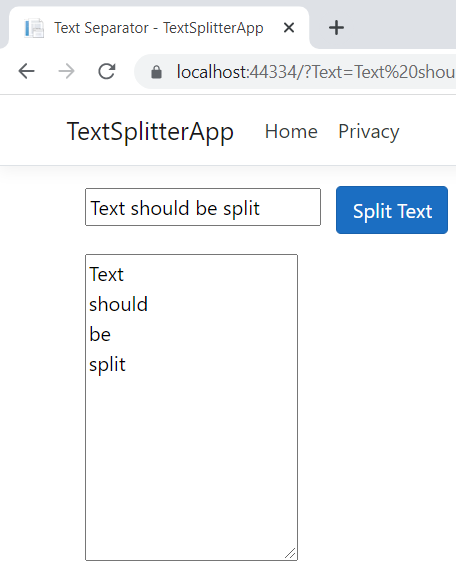




Make sure that your **app works correctly**. **Debug the code**, so that you fully understand the **MVC pattern**. Don't forget that **messages are deleted every time you close the app** because they are **stored in a variable** – that's why we often create **databases** for our apps.

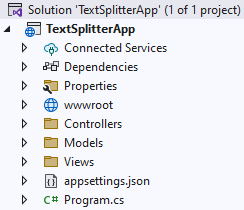
## Text Splitter App

We will begin this exercise by creating a **simple ASP.NET Core MVC app** called "Text Splitter". Our app will split text, entered by the user and then display the splitted words. It will look like this:

 → 

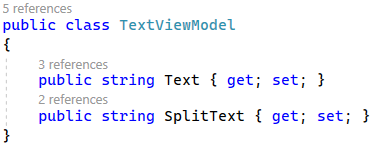
### Create the Project

First, **create the app and name it** "TextSplitterApp":

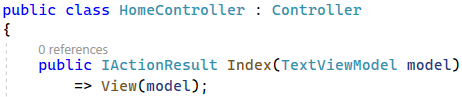


### Create Controller and Models

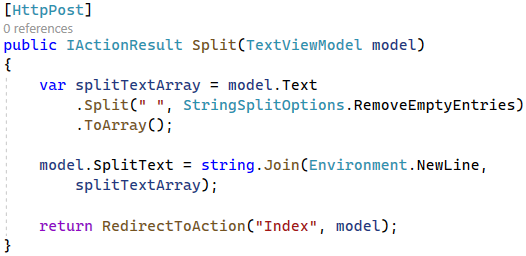
Before implementing the methods in the **HomeController**, create the needed models, which will be passed to the view. In the "**/Models**" folder, create a **TextViewModel** class (this is an ordinary class), which will hold the properties.



After we have created the TextViewModel class, it's time to modify the **Index()** method from the **HomeController** controller class to return a view and a model.



Now write the Split() **method**, as well. It should have the [HttpPost] **attribute**, which means that the action will be invoked on a "POST" **request** to "/Split". The method should also **accept a** TextViewModel (from the **view**), then update it and pass it to the **Index()** method.



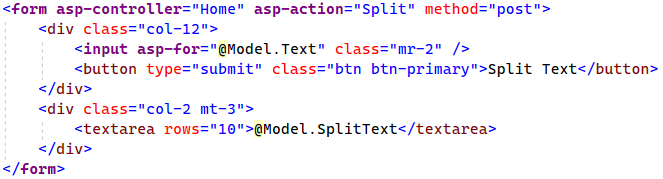
### Create a View

Now we should modify the **Index.cshtml** file.

We should accept a model in the view and change the **ViewBag.Title** to "**Text Splitter**"

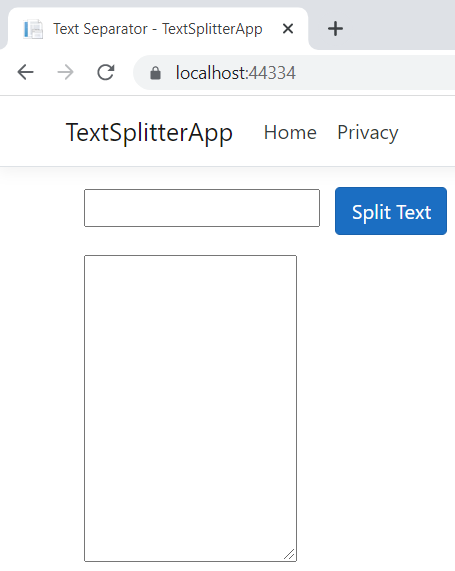
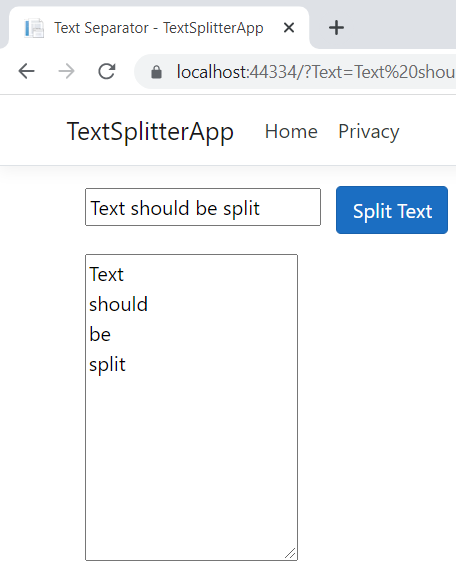


Now, we need to create a form, which should send a "**POST**" request and submit the information from the form to the **Split(TextViewModel model)** action of the **HomeController**.We will use the **@** symbol to switch to C# code in order to assign input data to model properties.



### Try the App

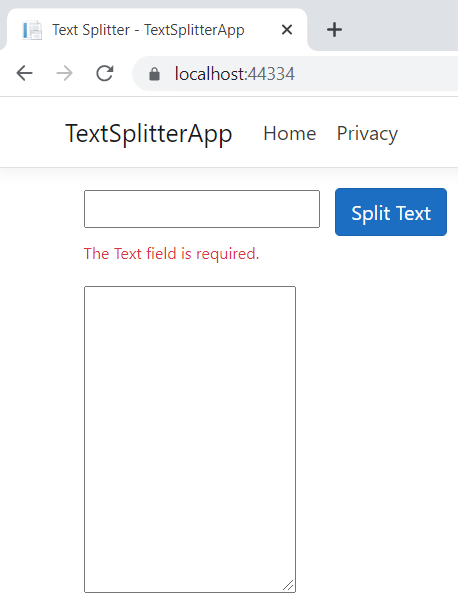
**Run the app** and examine it in the **browser**. Try splitting the sentence "**Text should be split**" and the result should look like this:

 → 

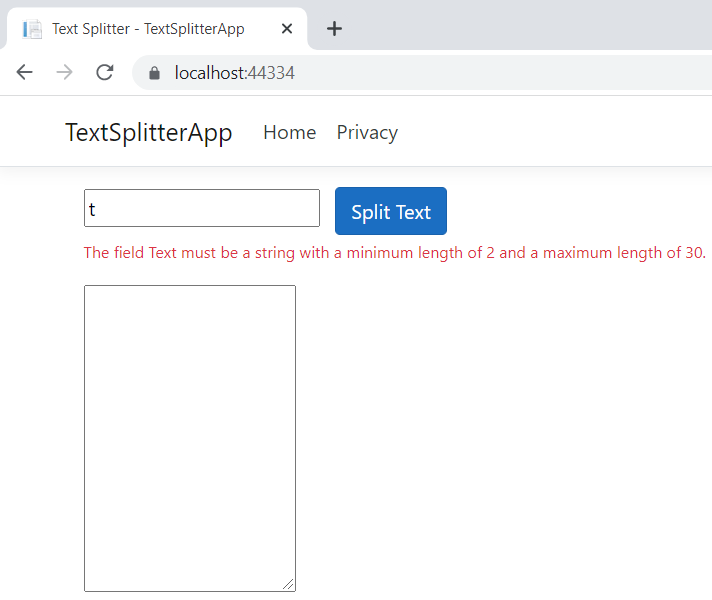
### Adding App Validations

Now, let's add some requirements:

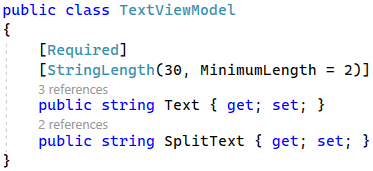
* The "Text" field should
  + Be **required** (not left empty)
  + Have a **minimum** length of **2** characters
  + Have a **maximum** length of **30** characters
* In case any **validation** **fails**, an **error** should be displayed.
  + If the text field is left empty, a "**The Text field is required**" message should be displayed like shown below:



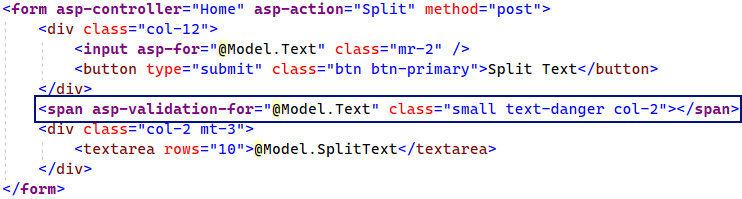
* + In case the length of the input is shorter than the minimum length or longer than the maximum length, a "**The field Text must be a string with a minimum length of 2 and maximum length of 30.**" should be displayed like shown below:



First, we will add validation attributes to the model property. 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.



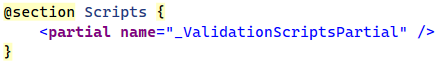
We will use the following tag helper in order to generate the validation message.



Finally, we will render a partial view, which executes the validation scripts. In order to do that, first we need to create a new **\_ValidationScriptsPartial.cshtml** file in the "**/Views/Shared**" folder and write the code as shown below:

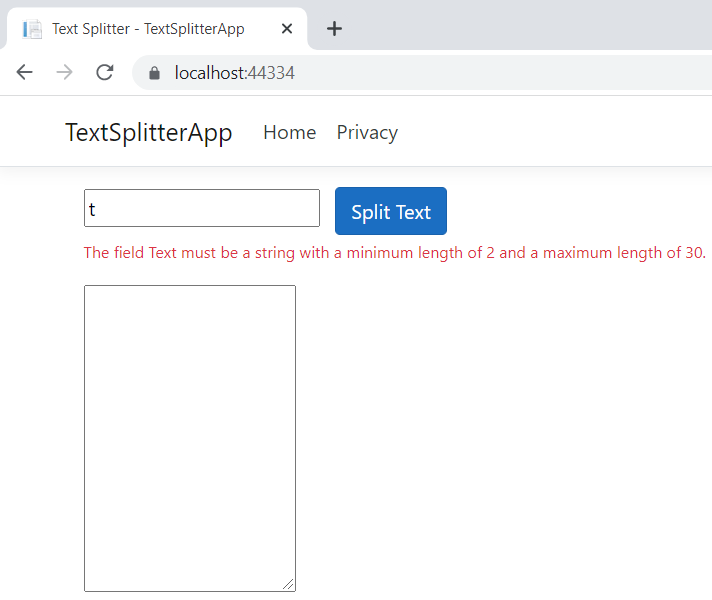


Add the following code to the end of the **Index.cshtml** file:



### Try the App

**Run the app** again and examine it in the **browser**. Try splitting "**t**" and the result should look like this:



When you press the **[Split Text]** button when the text field is empty, the app should look like this:

