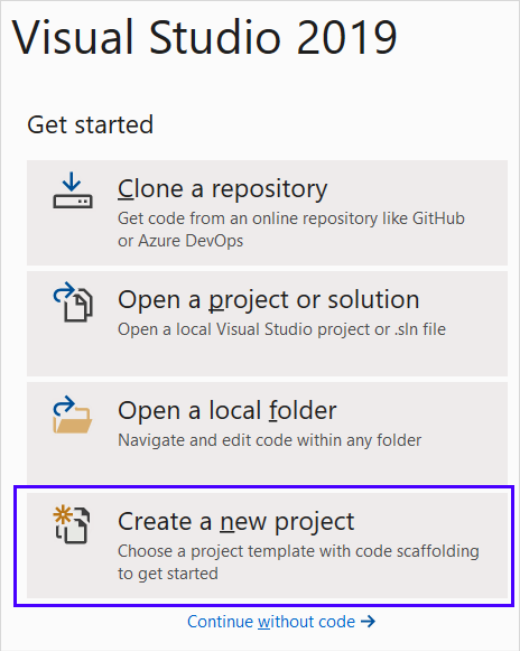
Exercises: ASP.NET Core Introduction

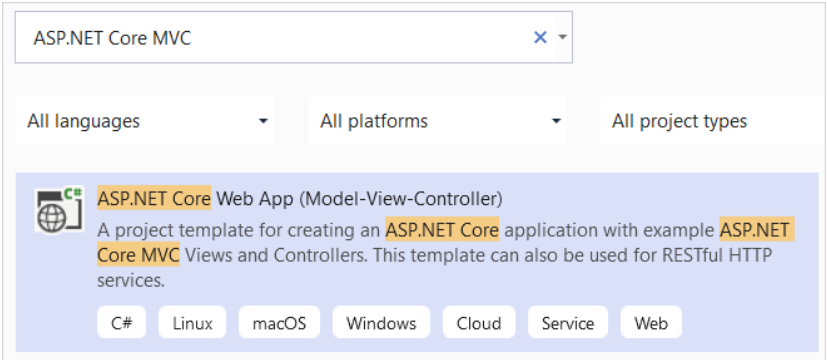
Problems for exercises and homework for the "ASP.NET MVC" course from the official "Applied Programmer" curriculum.

In this exercise we will create **simple ASP.NET Core apps** – the **one from the slides** and one for **exchanging messages.**

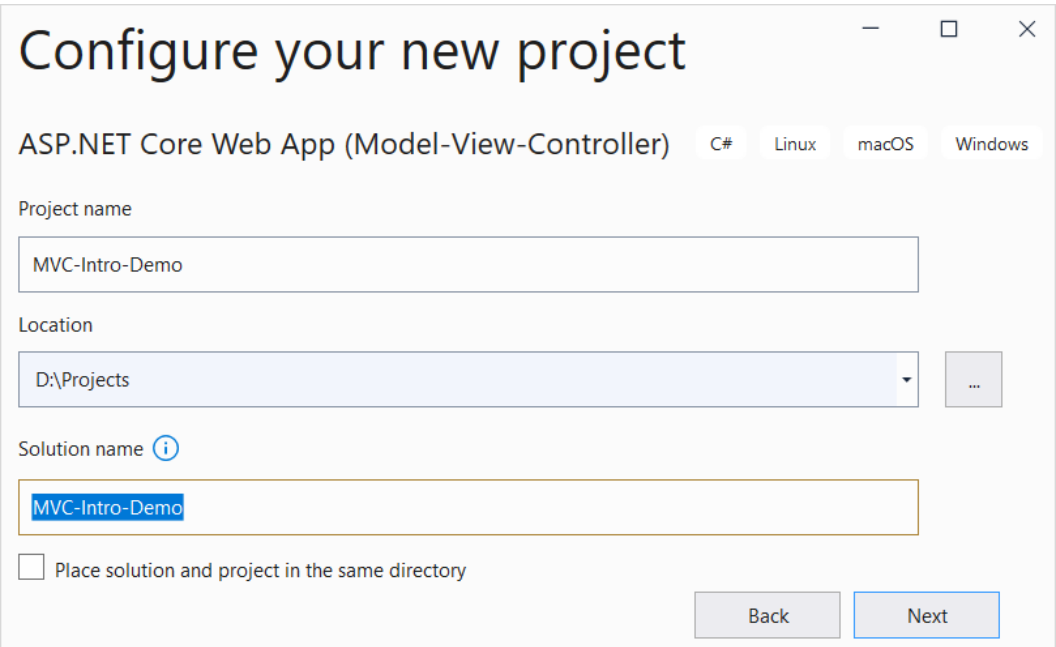
1. Create Simple Pages in an ASP.NET Core App

In this task you should **implement the pages** from the slides demo. 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:

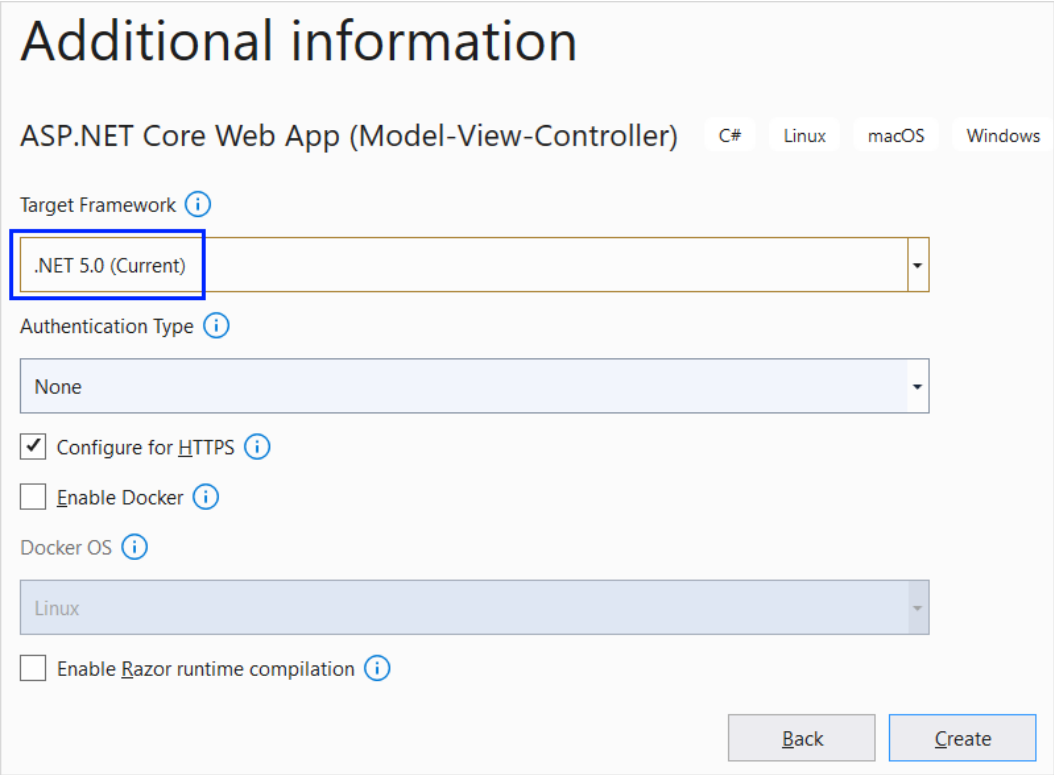




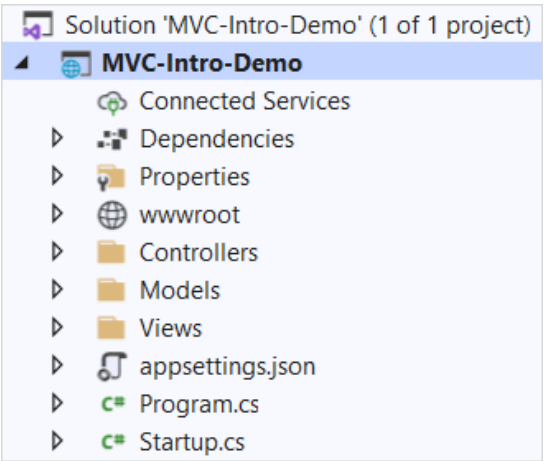
Give a **name** to your project and solution:



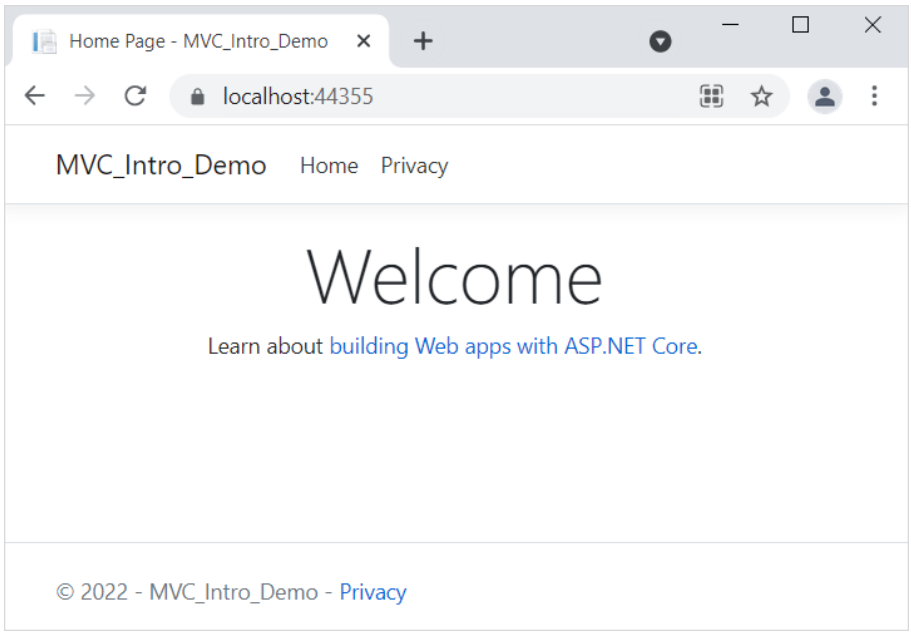
On the next step you should **choose .NET 5.0** as a **target framework** and click on the **[Create] button**:



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:



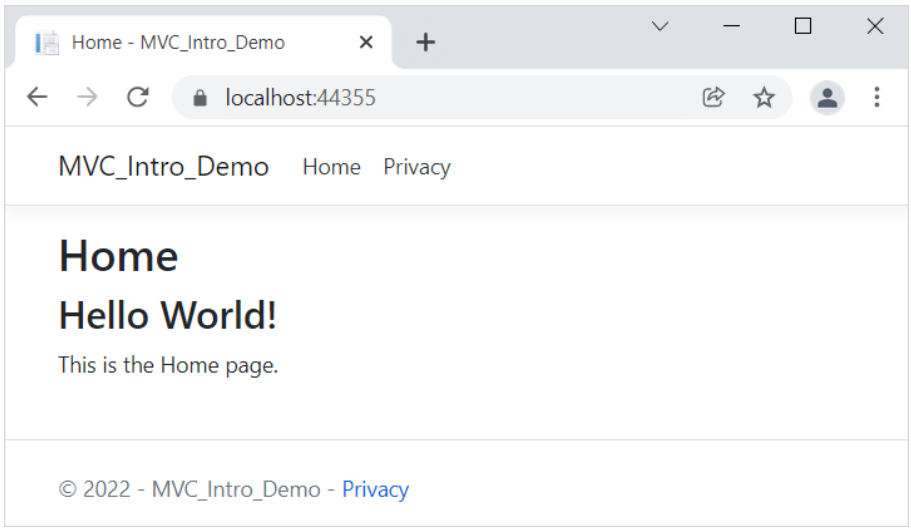
If you **run the app**, you will see the **default** "**Home**" **page**, which is served by the **HomeController** in the app:



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:

```
public class HomeController : Controller
{
    0 references
    public IActionResult Index()
    {
        ViewBag.Message = "Hello World!";
        return View();
    }
```

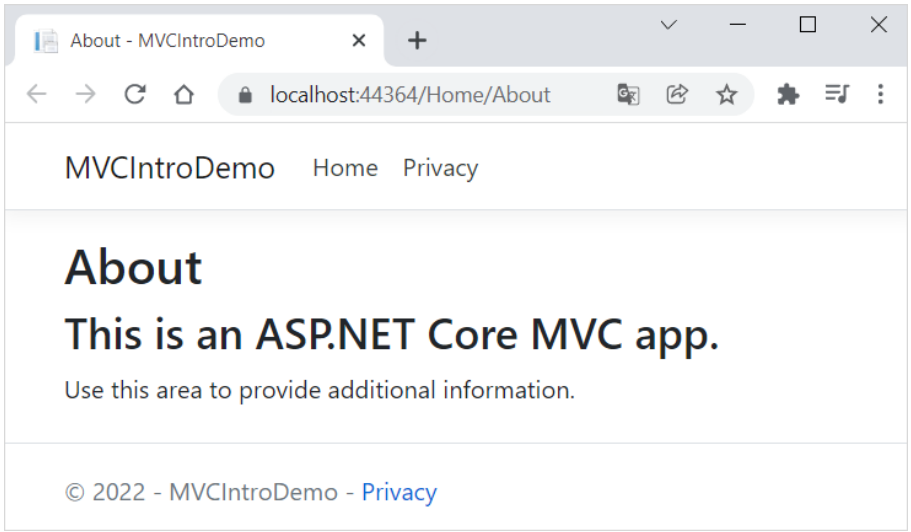
Now you should modify the **Index.cshtml view** in the "**/Views/Home**" **folder** to **display the page differently**. Use the **ViewBag class** to **get the message** from the controller. Note how the **Razor views** able us to use **C# code** inside **HTML**:

```
Index.cshtml*
@{
    ViewBag.Title = "Home";
}

<h2>@ViewBag.Title</h2>
<h3>@ViewBag.Message</h3>
<p>This is the Home page.</p>
```

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

Create the "About" Page

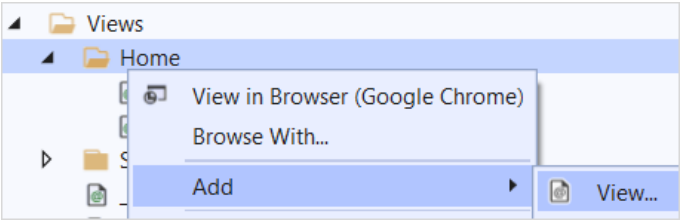
Create an "**About**" **page** in the app, which should look like this:



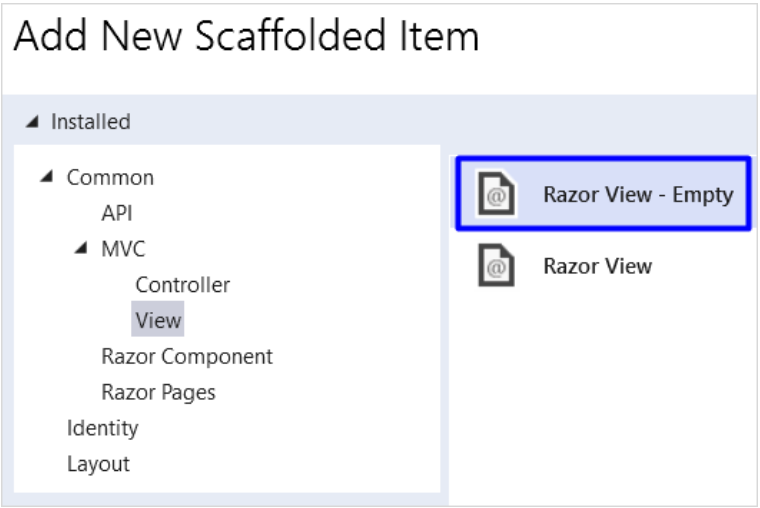
The page should be **accessed on "/Home/About"**. Create an **About() controller action** in the **HomeController class** for the "**About**" **page**. The controller method should return a **view**. It should also use the **ViewBag class** to **pass a message** to the returned view. Write the method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

```
public class HomeController : Controller
{
    ...
    public ActionResult About()
    {
        ViewBag.Message = "This is an ASP.NET Core MVC app.";
        return View();
    }
```

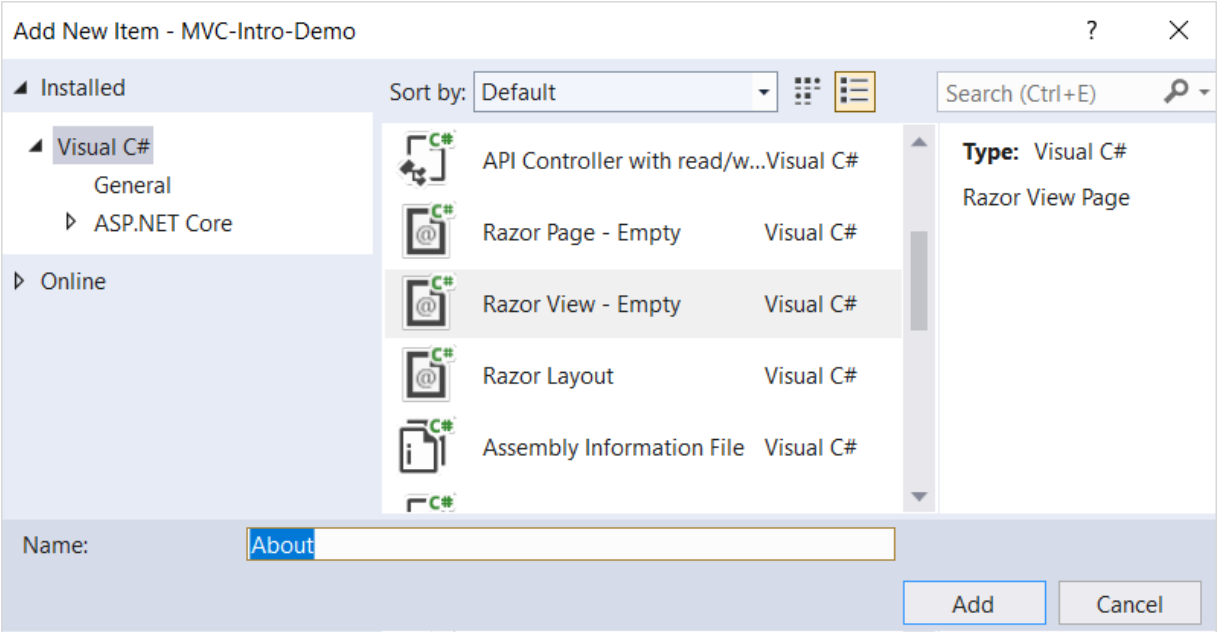
Now you should create an **About.cshtml view** in the "**/Views/Home**" **folder**. To do this, **right-click** on the folder and **choose [Add] → [View]**:



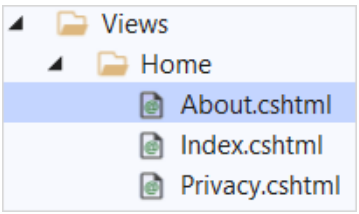
Then, choose the **[Razor View – Empty] option** to **create an empty view** and name it "**About**":



→



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



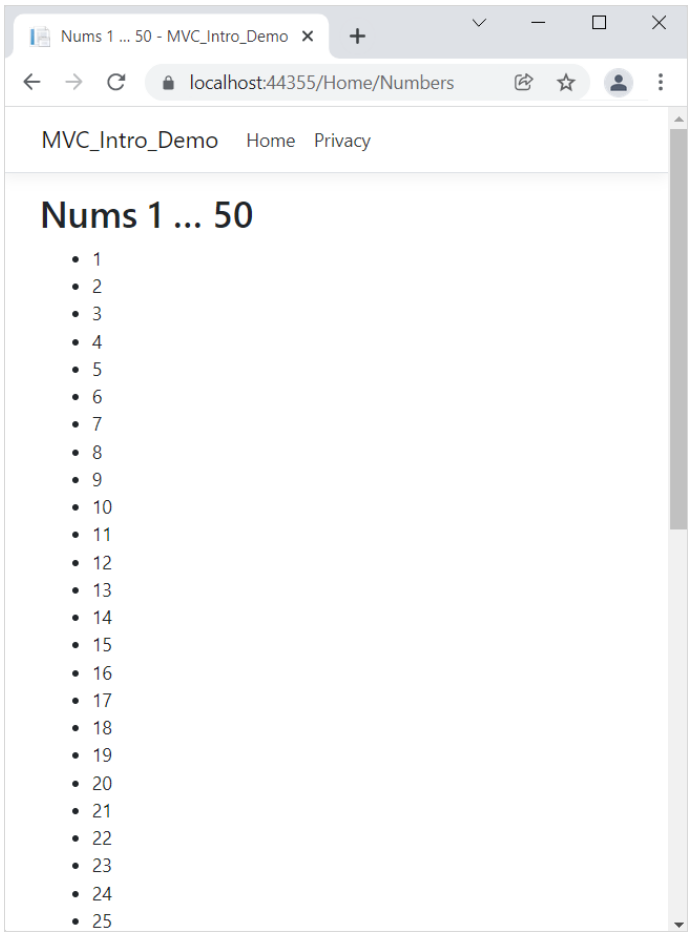
→

```
About.cshtml*
@{
    ViewBag.Title = "About";
}

<h2>@ViewBag.Title</h2>
<h3>@ViewBag.Message</h3>
<p>Use this area to provide additional information.</p>
```

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

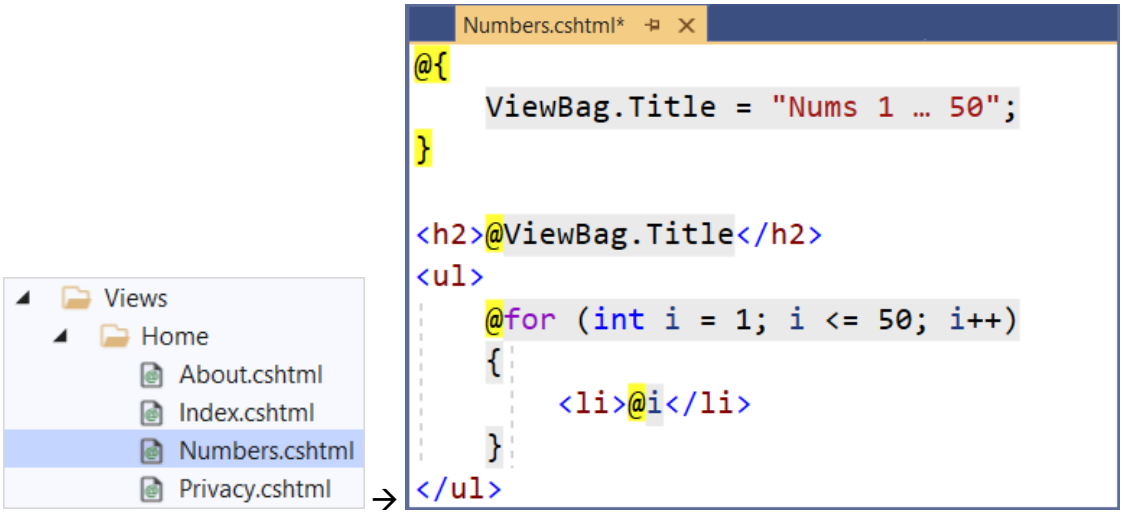
The "**Numbers 1…50**" page should **display the numbers from 1 to 50**. It should be accessed on "/**Home**/**Numbers**" and be the following:



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

```
public class HomeController : Controller
{
    ...
    public ActionResult Numbers()
    {
        return View();
    }
```

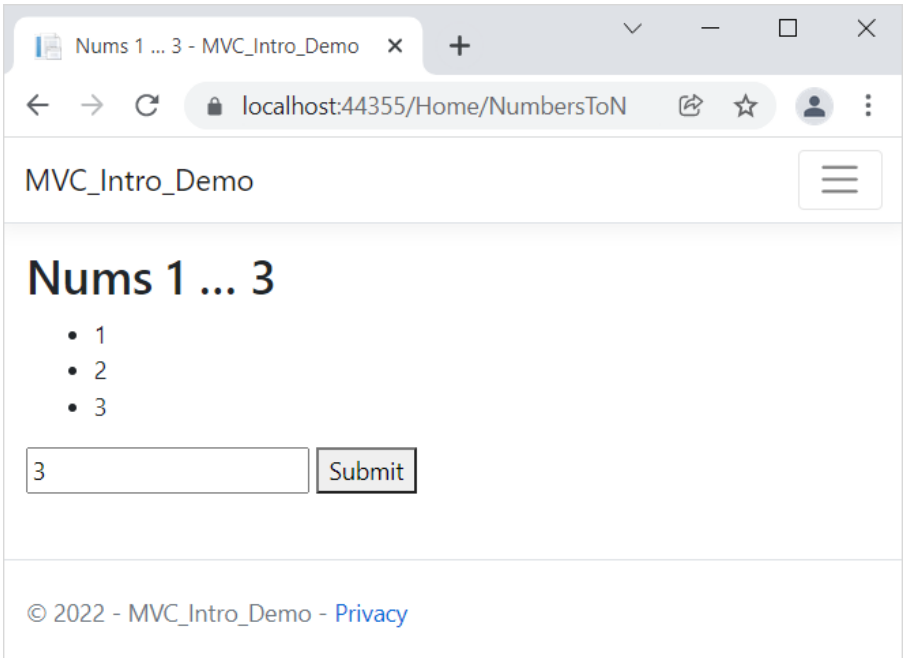
Create a **Numbers.cshtml view**, which should use a **for loop** to **display each number**. Write the **view** like this:



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:



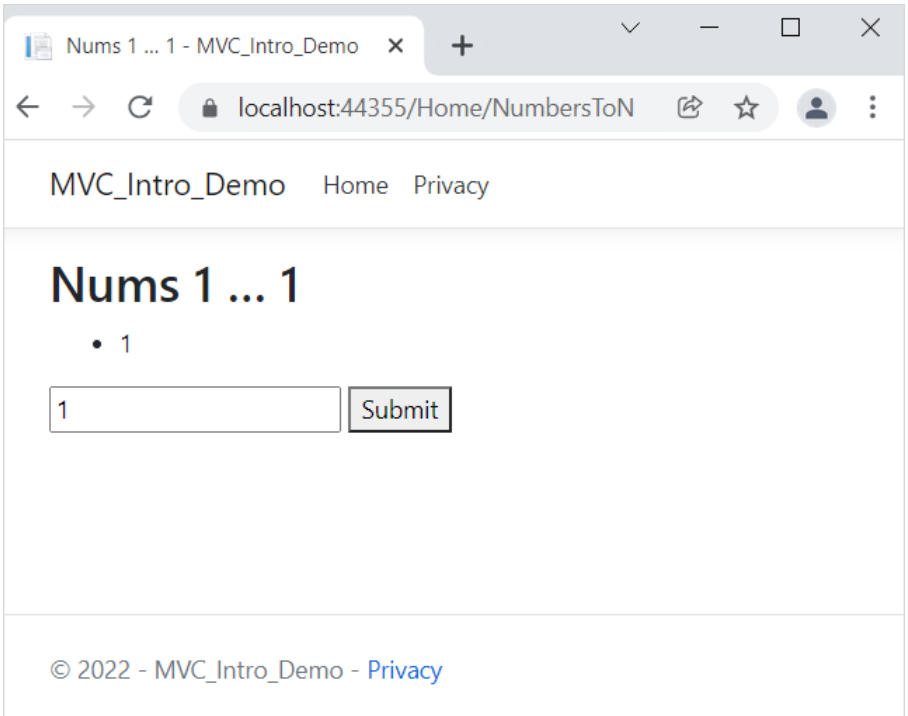
Write a **NumbersToN() method** in the **HomeController**. It should **accept a count parameter** from the **view** (with **default value** of the parameter 3). Then, it should **add the count number** to a **ViewBag** and **return a view**:

```
public class HomeController : Controller
{
    ...
    public ActionResult NumbersToN(int count = 3)
    {
        ViewBag.Count = count;
        return 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:

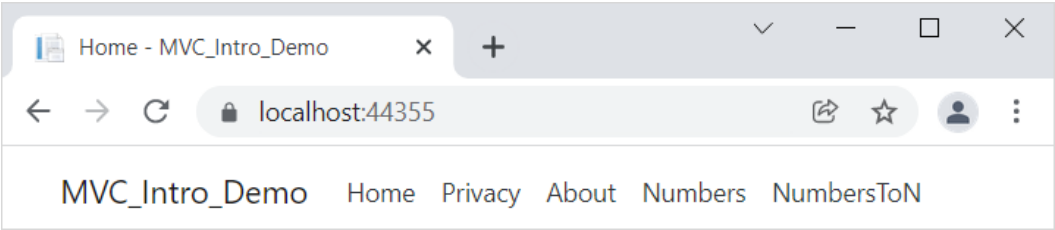
```
NumbersToN.cshtml*
@{
    ViewBag.Title = "Nums 1 … " + ViewBag.Count;
}

<h2>@ViewBag.Title</h2>
<ul>
    @for (int i = 1; i <= ViewBag.Count; i++)
    {
        <li>@i</li>
    }
</ul>

<form method="post">
    <input name="count" value="@ViewBag.Count">
    <button type="submit">Submit</button>
</form>
```

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:

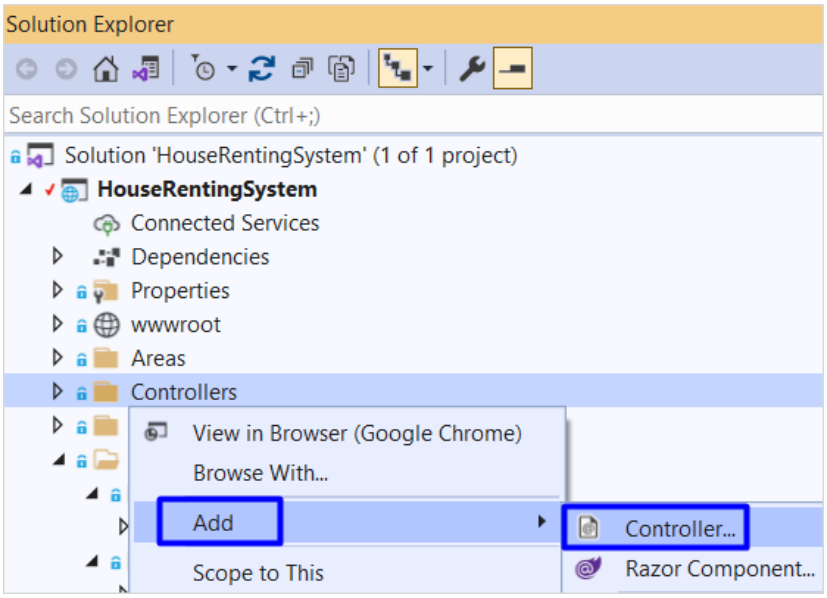
```
_Layout.cshtml
<!DOCTYPE html>
<html lang="en">
<head>…</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light b
            <div class="container">
                <a class="navbar-brand" asp-area="" asp-controller="Home" asp-a
                <button class="navbar-toggler">…</button>
                <div class="navbar-collapse collapse d-sm-inline-flex justify-c
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">…</li>
                        <li class="nav-item">…</li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-controller="Home"
                               asp-action="About">About</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-controller="Home"
                               asp-action="Numbers">Numbers</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-controller="Home"
                               asp-action="NumbersToN">NumbersToN</a>
                        </li>
                    </ul>
```

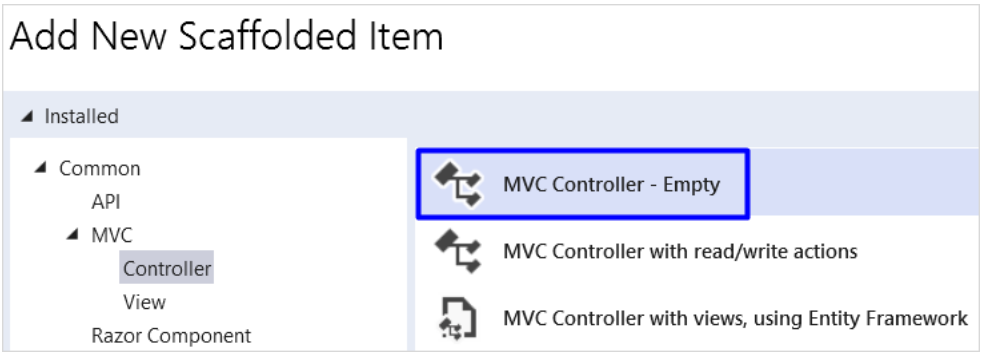
The **asp-controller** and **asp-action tag helpers** set the **controller** and **action names** of the page, which should be accessed.

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

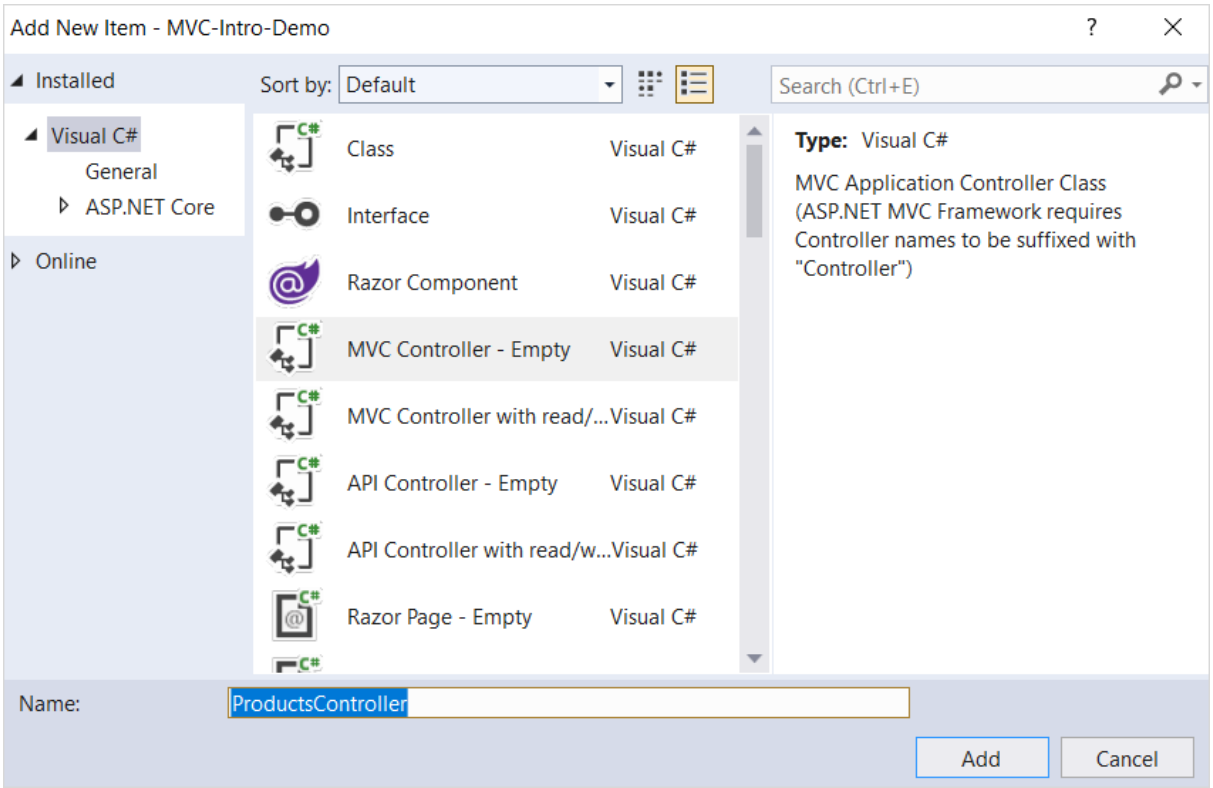
ProductsController Pages

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**:

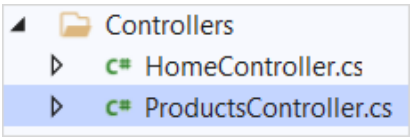




Set the **controller name** like this:



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




```
public class ProductsController
    : Controller
{
}
```

As we already know, 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**:

```
public class ProductViewModel
{
    0 references
    public int Id { get; set; }
    0 references
    public string Name { get; set; }
    0 references
    public double Price { get; set; }
}
```

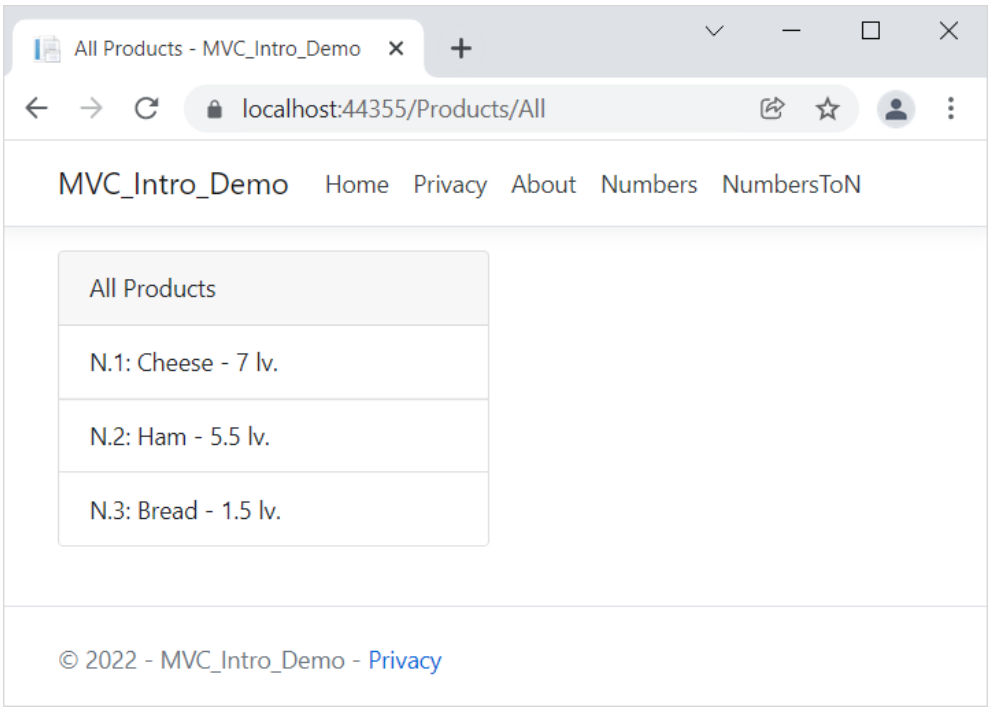
Now go back to the **ProductsController** and **add a field with products**. The field should have a **collection of ProductViewModel** with **three products** like this:

```
public class ProductsController : Controller
{
    private IEnumerable<ProductViewModel> products
        = new List<ProductViewModel>()
        {
            new ProductViewModel()
            {
                Id = 1,
                Name = "Cheese",
                Price = 7.00
            },
            new ProductViewModel()
            {
                Id = 2,
                Name = "Ham",
                Price = 5.50
            },
            new ProductViewModel()
            {
                Id = 3,
                Name = "Bread",
                Price = 1.50
            }
        };
}
```

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

Create the "All Products" Page

The "**All Products**" **page** should **display the products** from the controller field like this:



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

```
public class ProductsController : Controller
{
    ...
    public IActionResult All()
    {
        return View(this.products);
    }
```

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:

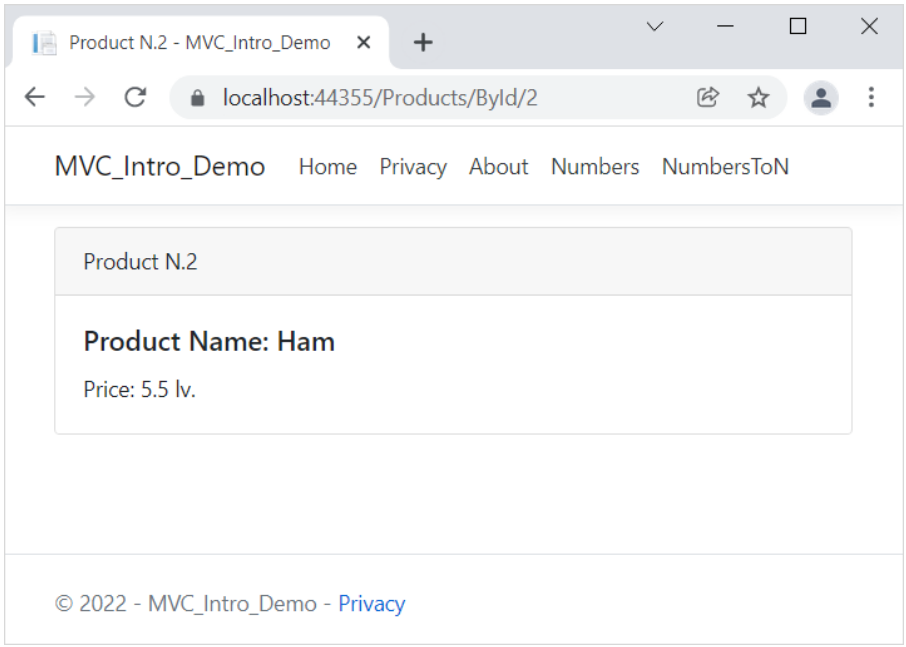
```
All.cshtml*
@model IEnumerable<ProductViewModel>

@{
    ViewBag.Title = "All Products";
}

<div class="card" style="width: 18rem;">
    <div class="card-header">@ViewBag.Title</div>
    <ul class="list-group list-group-flush">
        @foreach (var pr in Model)
        {
```

```
            <li class="list-group-item">
                N.@pr.Id: @pr.Name - @pr.Price lv.
            </li>
        }
    </ul>
</div>
```

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

Create the "Product By Id" Page

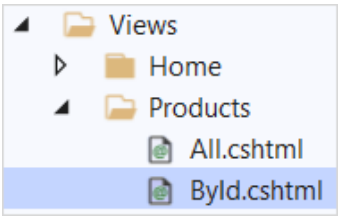
The "**Product By Id**" page should **display a product by id**:



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**:

```
public class ProductsController : Controller
{
    ...
    public IActionResult ById(int id)
    {
        var product = this.products
            .FirstOrDefault(p => p.Id == id);

        if (product == null)
            return BadRequest();

        return View(product);
    }
```

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



→

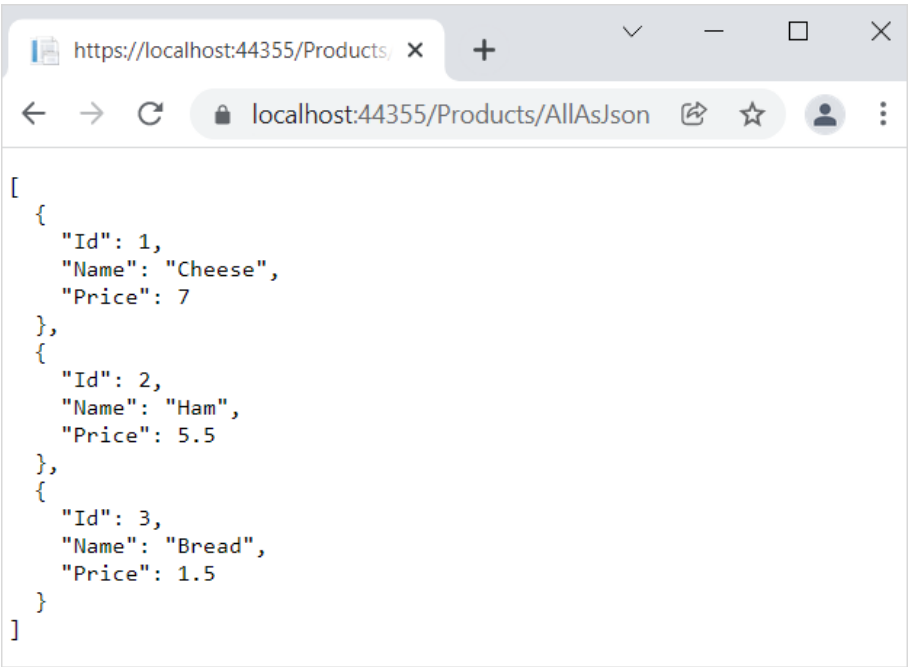
ById.cshtml*

```
@model ProductViewModel

@{ ViewBag.Title = "Product N." + Model.Id;}

<div class="card">
    <div class="card-header">@ViewBag.Title</div>
    <div class="card-body">
        <h5 class="card-title">Product Name: @Model.Name</h5>
        <p class="card-text">Price: @Model.Price lv.</p>
    </div>
</div>
```

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**":



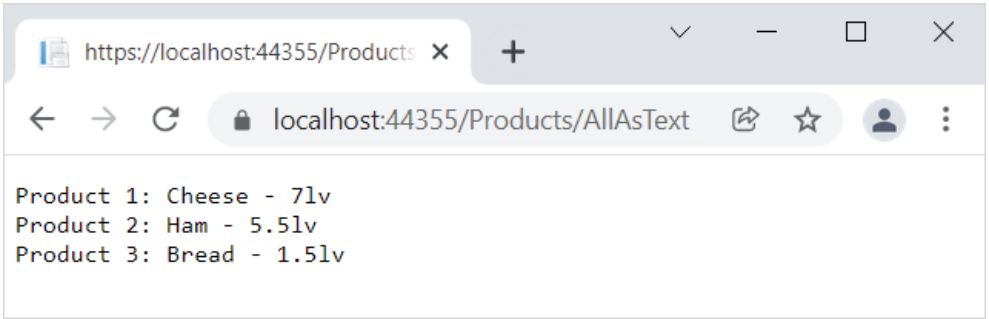
Create the **AllAsJson()** method in the **HomeController**, which should return a **JSON with the products** as shown below. It should use **JSON options** to display the JSON **indented**:

```
public class ProductsController : Controller
{
    ...
    public IActionResult AllAsJson()
    {
        var options = new JsonSerializerOptions
                { WriteIndented = true };

        return Json(products, options);
    }
```

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**":



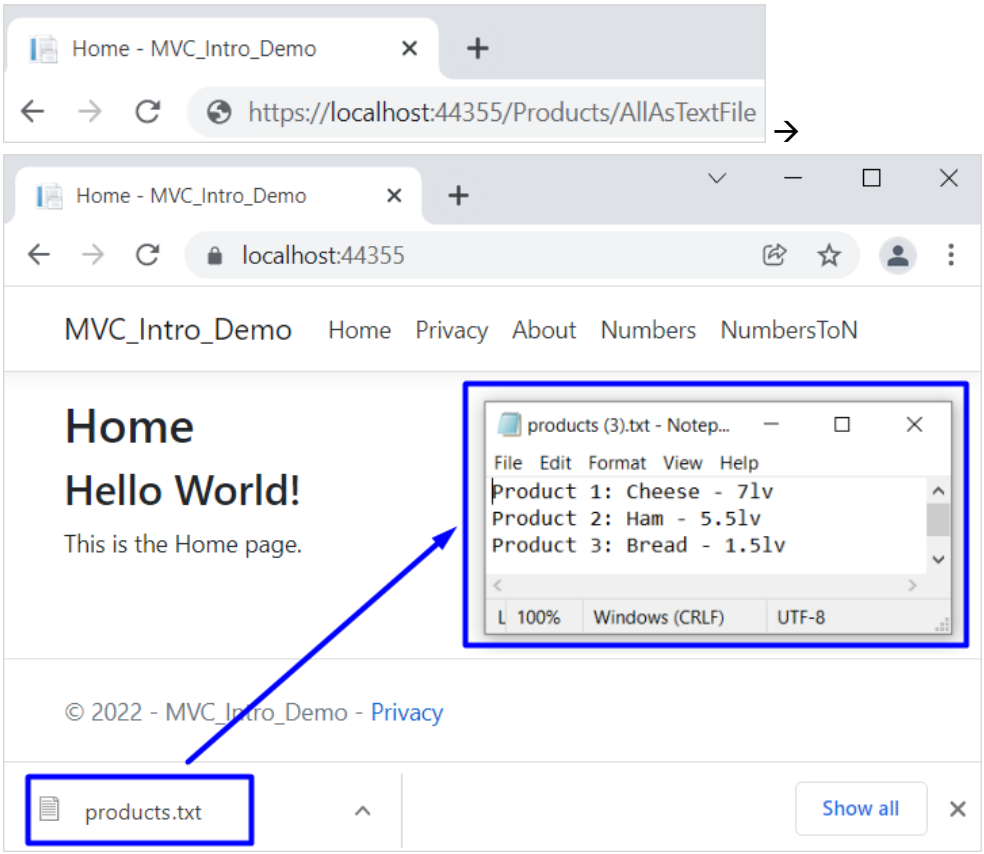
The **AllAsText() method** in the **ProductsController** should **create a string of all products** and return it as a **plain text response**:

```
public class ProductsController : Controller
{
    ...
    public IActionResult AllAsText()
    {
        var text = string.Empty;
        foreach (var pr in products)
        {
            text += $"Product {pr.Id}: {pr.Name} - {pr.Price}lv";
            text += "\r\n";
        }
        return Content(text);
    }
```

Try it in the browser.

Download Products As Text File

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

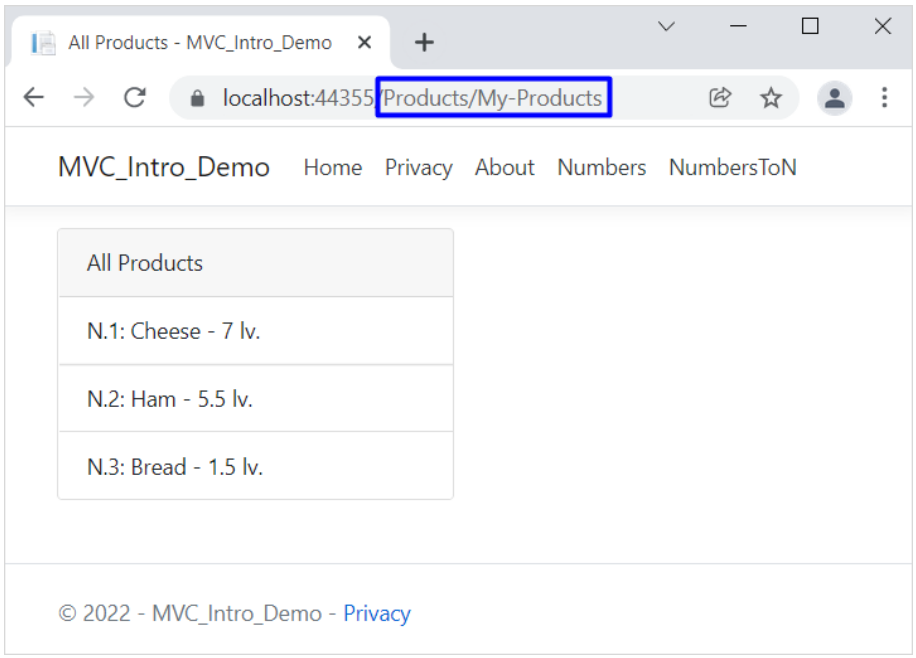


The **AllAsTextFile() method** in the **ProductsController** should form a **text with the products**. Then, it should **add the Content-Disposition header** to the **Response**, so that the **file is downloaded** as an **attachment**. At the end, it should **return a file with the text as a byte array**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

```
public class ProductsController : Controller
{
    ...
    public IActionResult AllAsTextFile()
    {
        var text = string.Empty;
        foreach (var pr in products)
        {
            text += $"Product {pr.Id}: {pr.Name} - {pr.Price}lv";
            text += "\r\n";
        }

        Response.Headers.Add(HeaderNames.ContentDisposition,
            @"attachment;filename=products.txt");

        return File(Encoding.UTF8.GetBytes(text), "text/plain");
    }
```

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:

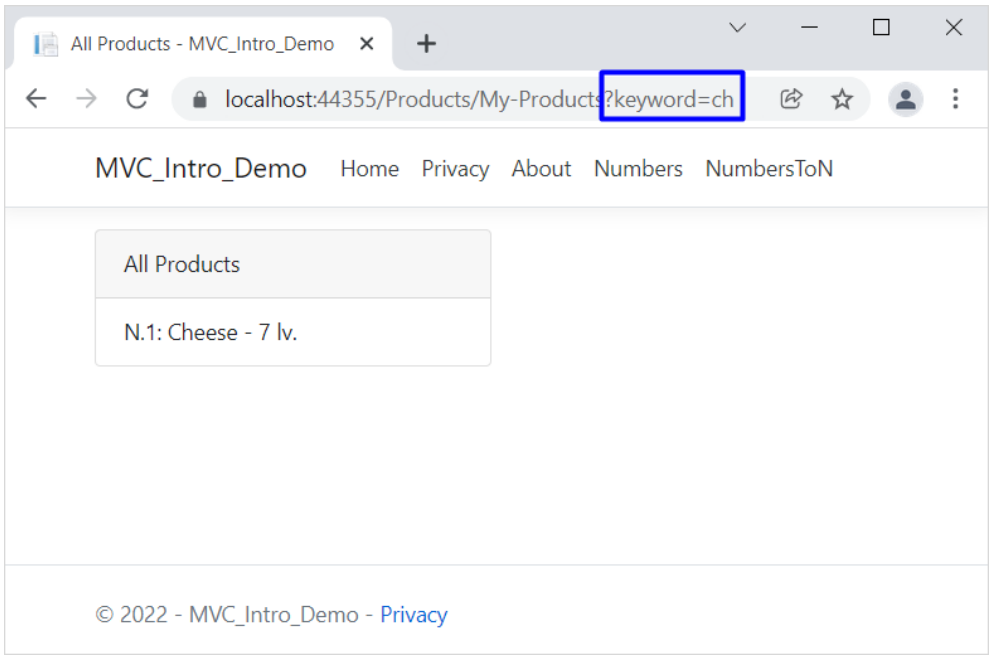
```
public class ProductsController : Controller
{
    ...
    [ActionName("My-Products")]
    0 references
    public IActionResult All()
    {
        return View(this.products);
    }
```

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**:

Now access the "**All Products**" **page** on "**/Products/My-Products**" in the browser.

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:



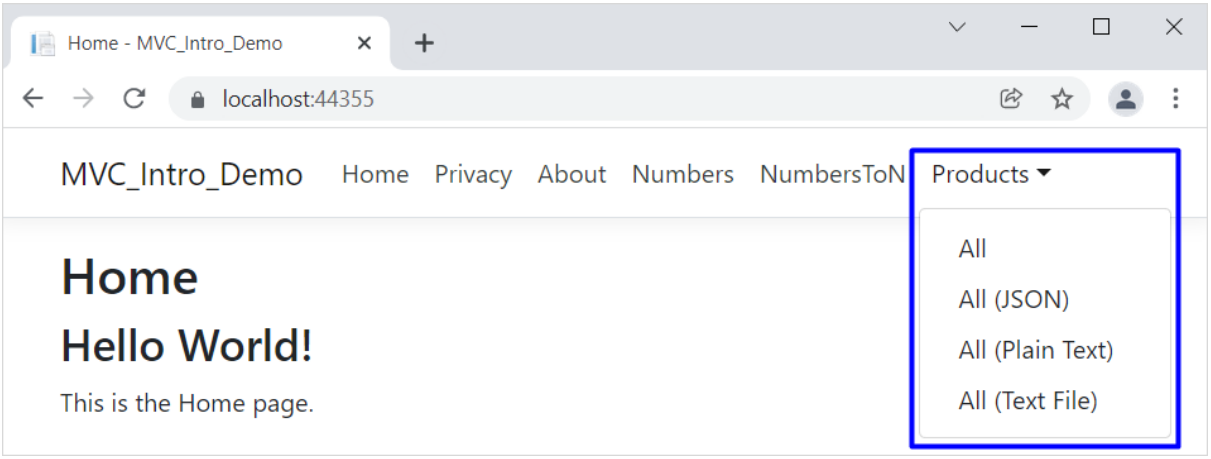
To do this, make the **All() controller action accept a keyword** and return only the **filtered products**, when there is a **keyword** in the **URL**:

```
public class ProductsController : Controller
{
    ...
    [ActionName("My-Products")]
    0 references
    public IActionResult All(string keyword)
    {
        if (keyword != null)
        {
            var foundProducts = this.products
                .Where(pr => pr.Name.ToLower()
                    .Contains(keyword.ToLower()));

            return View(foundProducts);
        }

        return View(this.products);
    }
```

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

Now it is time to **add links to all the pages** we created in the **navigation pane**. They will all be in a "**Products**" **dropdown** like this:



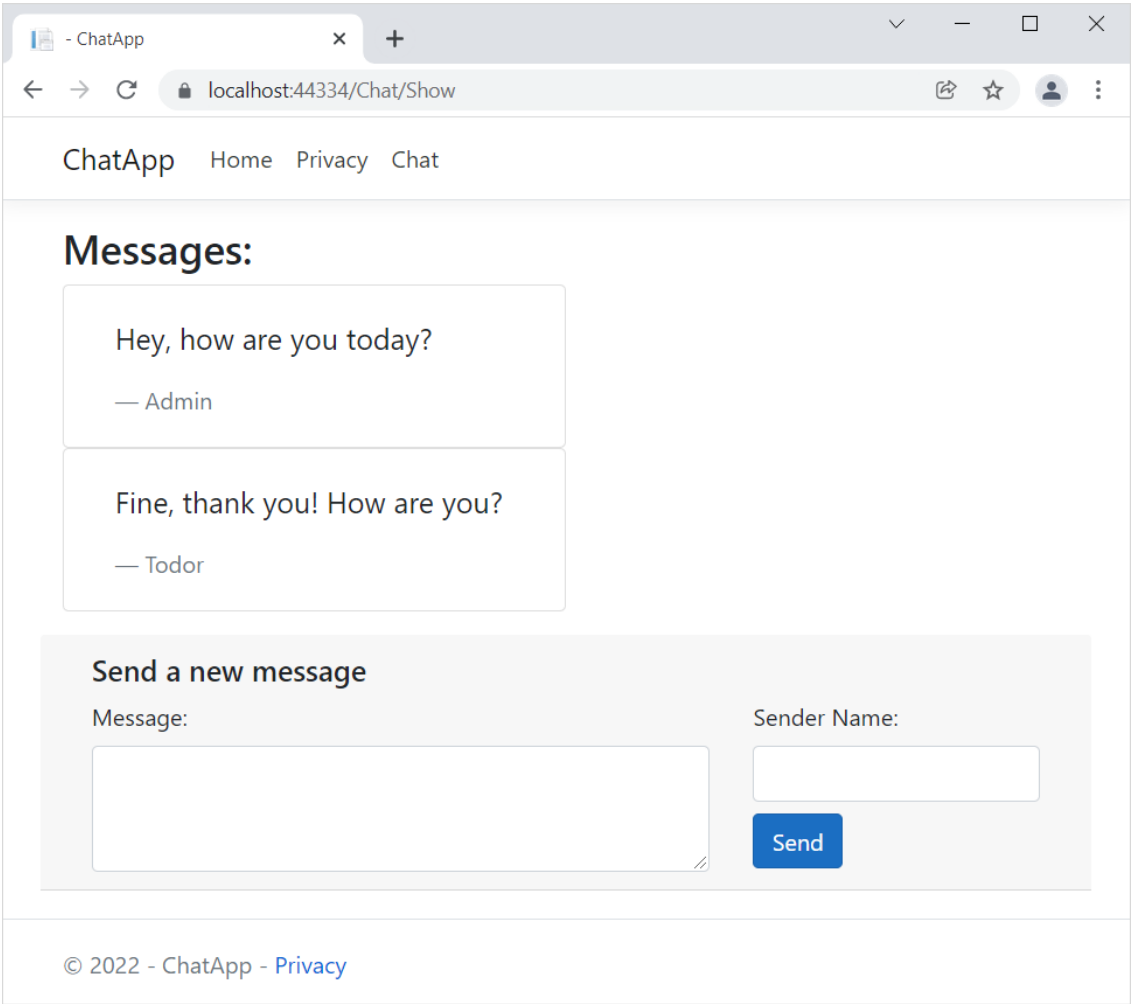
Modify the **_Layout.cshtml view** like this to have the above **links**:

```
_Layout.cshtml*
<html lang="en">
<head>…</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bo
            <div class="container">
                <a class="navbar-brand" asp-area="" asp-controller="Home" asp-action="Index">MVC
                <button class="navbar-toggler">…</button>
                <div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">…</li>
                        <li class="nav-item">…</li>
                        <li class="nav-item">…</li>
                        <li class="nav-item">…</li>
                        <li class="nav-item">…</li>
                        <li class="nav-item dropdown">
                            <a class="nav-link dropdown-toggle" data-toggle="dropdown"
                               aria-haspopup="true" aria-expanded="false">Products</a>
                            <div class="dropdown-menu" aria-labelledby="navbarDropdownMenuLink">
                                <a class="dropdown-item" asp-controller="Products"
                                   asp-action="My-Products">All</a>
                                <a class="dropdown-item" asp-controller="Products"
                                   asp-action="AllAsJson">All (JSON)</a>
                                <a class="dropdown-item" asp-controller="Products"
                                   asp-action="AllAsText">All (Plain Text)</a>
                                <a class="dropdown-item" asp-controller="Products"
                                   asp-action="AllAsTextFile">All (Text File)</a>
                            </div>
                        </li>
                    </ul>
```

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

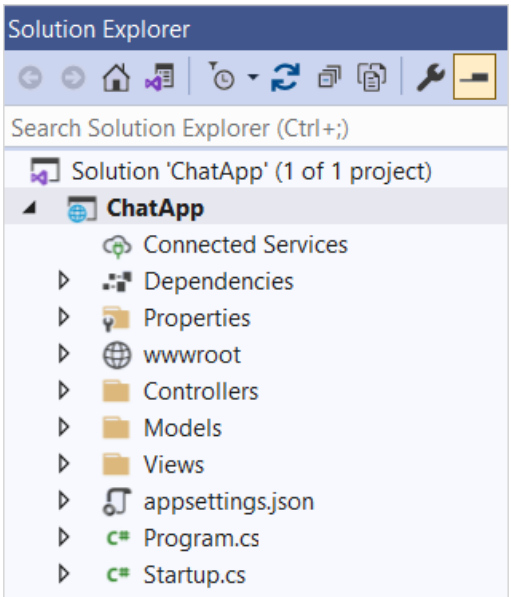
2. Simple Chat ASP.NET Core MVC App

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"**:



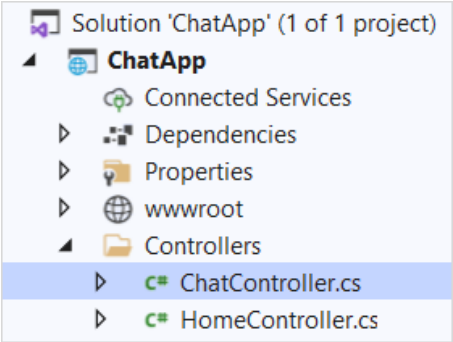
The **workflow** of the **chat functionality** in the app will be the following:

- 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**:




```
public class ChatController : Controller
{
    0 references
    public IActionResult Index()
    {
        return View();
    }
}
```

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 return 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

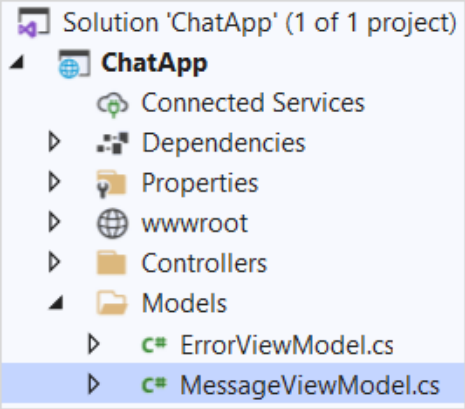
Write the above **class field** and **properties**:

```
public class ChatController : Controller
{
    private static List<KeyValuePair<string, string>> Messages =
        new List<KeyValuePair<string, string>>();

    0 references
    public IActionResult Show()
    {
        return View();
    }

    [HttpPost]
    0 references
    public IActionResult Send()
    {
        return RedirectToAction("Show");
    }
}
```

⚠	**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**):




```
public class MessageViewModel
{
    0 references
    public string Sender { get; set; }
    0 references
    public string MessageText { get; set; }
}
```

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

```
public class ChatViewModel
{
    0 references
    public MessageViewModel CurrentMessage { get; set; }

    0 references
    public List<MessageViewModel> Messages { get; set; }
}
```

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:

```
public class ChatController : Controller
{
    private static List<KeyValuePair<string, string>> Messages =
        new List<KeyValuePair<string, string>>();

    0 references
    public IActionResult Show()
    {
        if (Messages.Count() < 1)
        {
            return View(new ChatViewModel());
        }
```

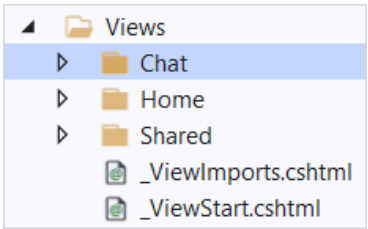
```
        var chatModel = new ChatViewModel()
        {
            Messages = Messages
                .Select(m => new MessageViewModel()
                {
                    Sender = m.Key,
                    MessageText = m.Value
                })
                .ToList()
        };

        return View(chatModel);
    }
```

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:

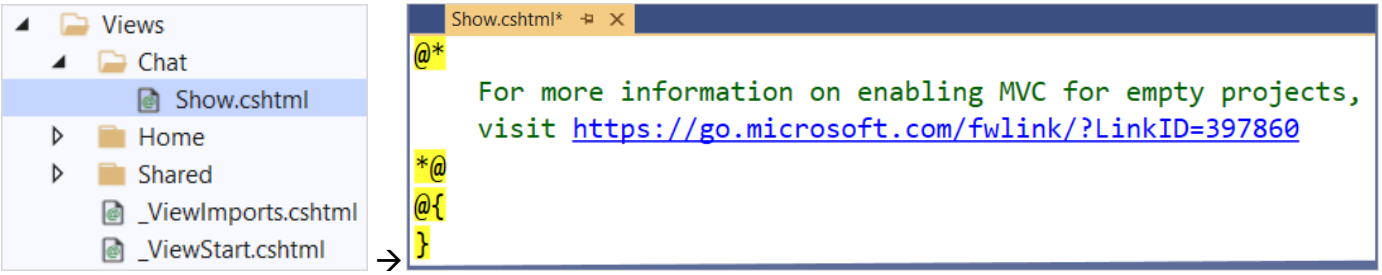
```
    [HttpPost]
    0 references
    public IActionResult Send(ChatViewModel chat)
    {
        var newMessage = chat.CurrentMessage;

        Messages.Add(new KeyValuePair<string, string>
            (newMessage.Sender, newMessage.MessageText));

        return RedirectToAction("Show");
    }
}
```

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**:

```
Show.cshtml
@model ChatViewModel
```

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

```
<h3>Messages:</h3>
```

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:

```
@if (Model.Messages != null)
{
    @foreach (var message in Model.Messages)
    {
        <div class="card .bg-light col-6">
            <div class="card-body">
                <blockquote class="blockquote mb-0">
                    <p>@message.MessageText<p>
                        <footer class="blockquote-footer">@message.Sender</footer>
                </blockquote>
            </div>
        </div>
    }
}
else
{
    <p>No messages yet!</p>
}
```

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:

```
<p></p>
<form asp-controller="Chat" asp-action="Send" method="post">
    <div class="form-group card-header row">
        <div class="col-12">
            <h5>Send a new message</h5>
        </div>
        <div class="col-8">
            <label>Message: </label>
            <textarea asp-for="CurrentMessage.MessageText"
                      class="form-control" rows="3"></textarea>
        </div>
        <div class="col-4">
            <label>Sender Name: </label>
            <input asp-for="CurrentMessage.Sender" class="form-control">
            <input class="btn btn-primary mt-2
                   float-lg-right" type="submit" value="Send" />
        </div>
    </div>
</form>
```

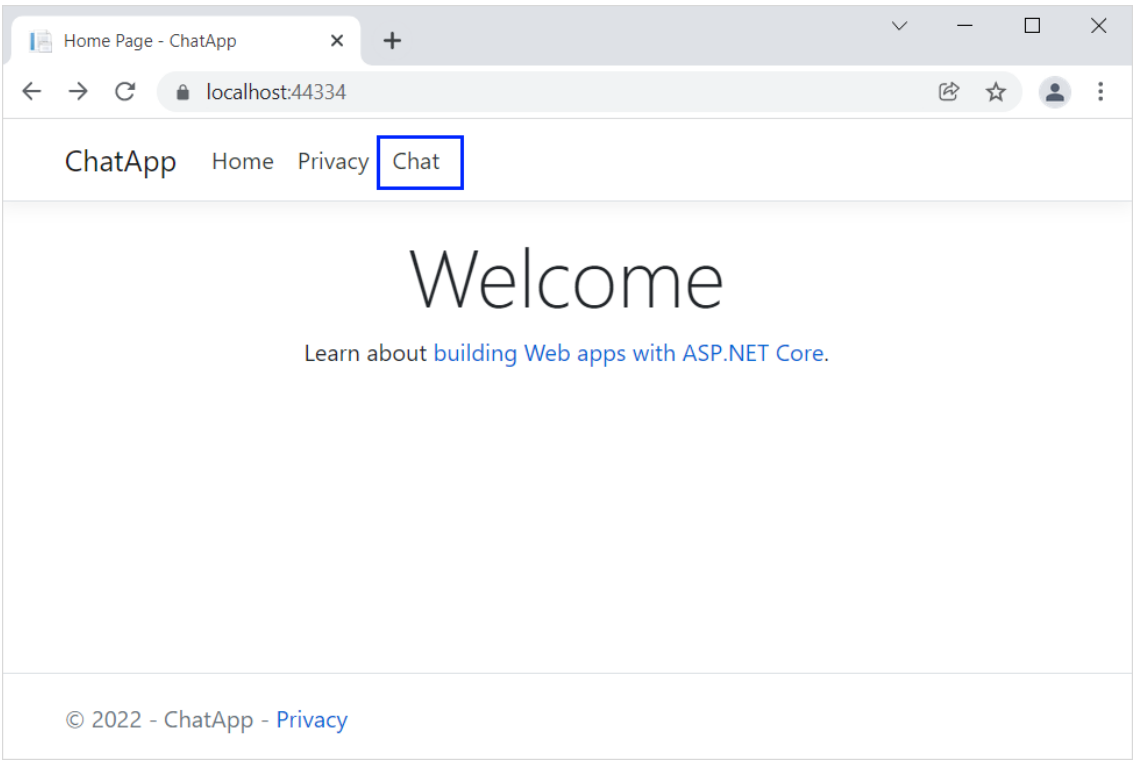
Now if we access "/**Chat/Show**" we will see the **Show.cshtml view**.

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

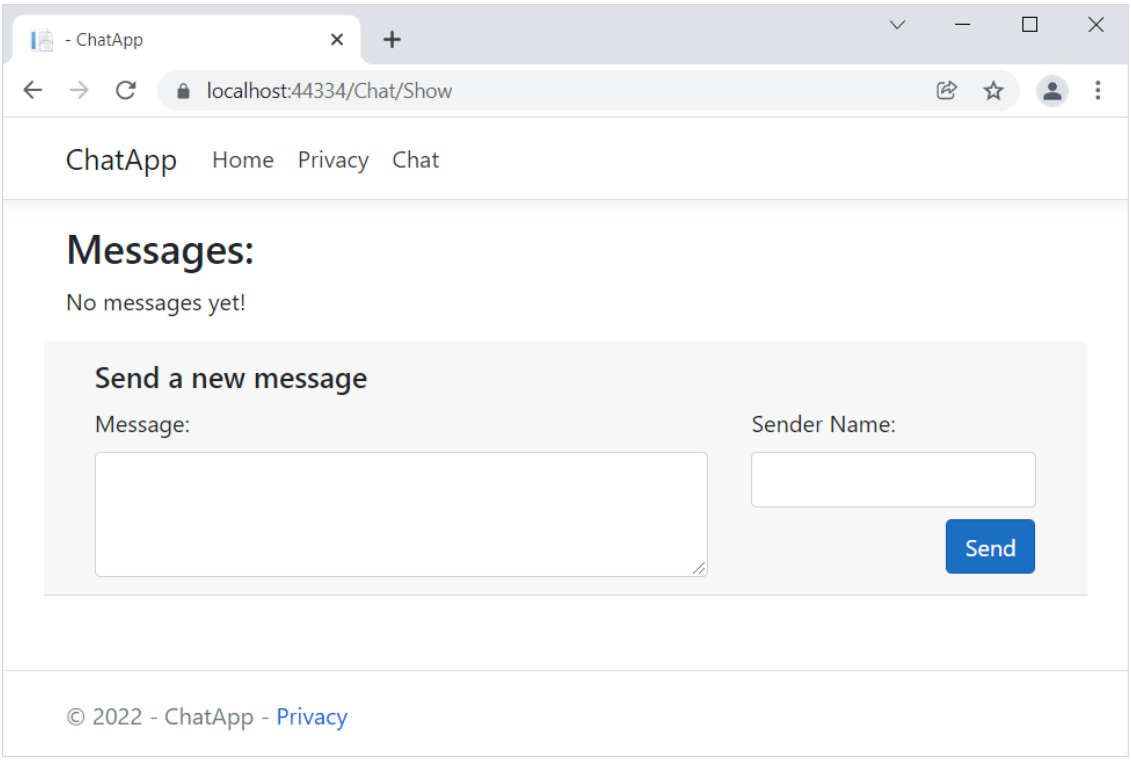
```
_Layout.cshtml
<!DOCTYPE html>
<html lang="en">
<head>…</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white
             border-bottom box-shadow mb-3">
            <div class="container">
                <a class="navbar-brand" asp-area="" asp-controller="Home"
                   asp-action="Index">ChatApp</a>
                <button class="navbar-toggler">…</button>
                <div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Home"
                               asp-action="Index">Home</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area=""
                               asp-controller="Home" asp-action="Privacy">Privacy</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area=""
                               asp-controller="Chat" asp-action="Show">Chat</a>
                        </li>
                    </ul>
```

Try the App

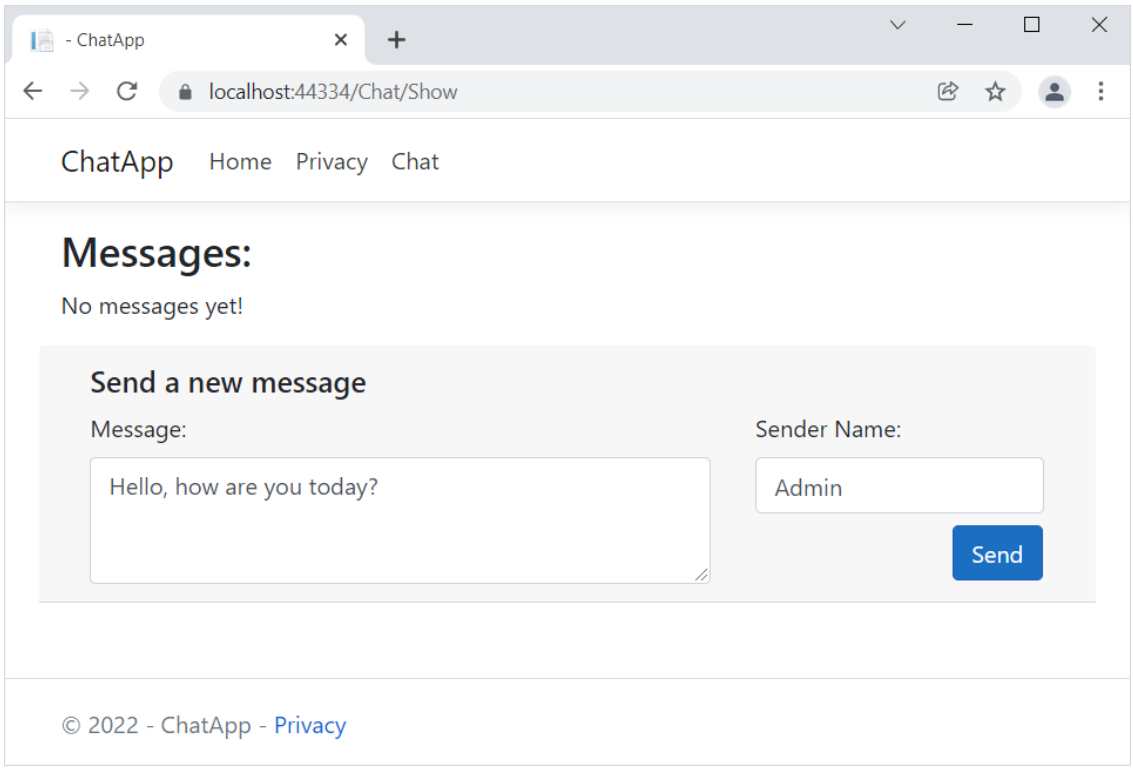
Run the app and examine it in the **browser**. It should have the "**Chat**" **navigation link**, which we have just added:



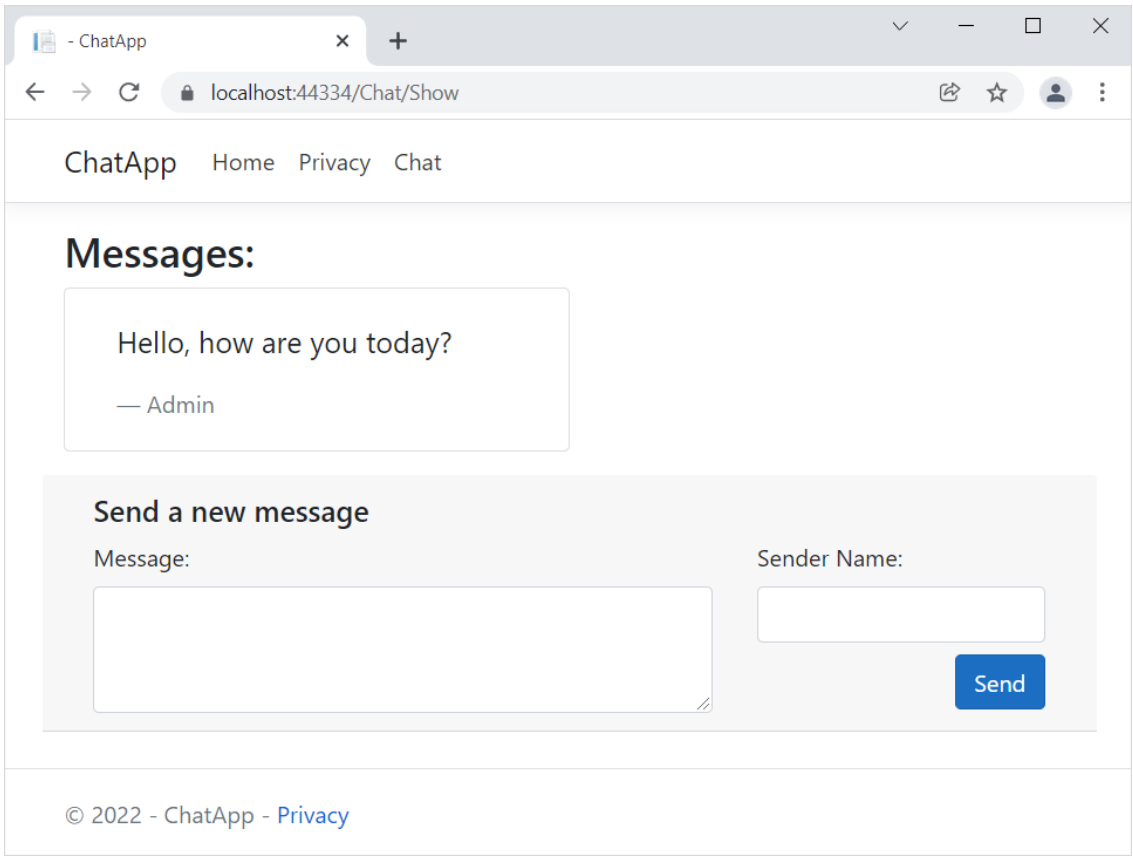
Click on the **[Chat] link**. You should be **redirected** to "**/Chat/Show**" and see the **Show.cshtml view**:



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.

Next time we will **create an ASP.NET Core MVC app**, which we will **develop till the end of this course** and it will be your **project for the final exam**.