Tutorial for Class number 2

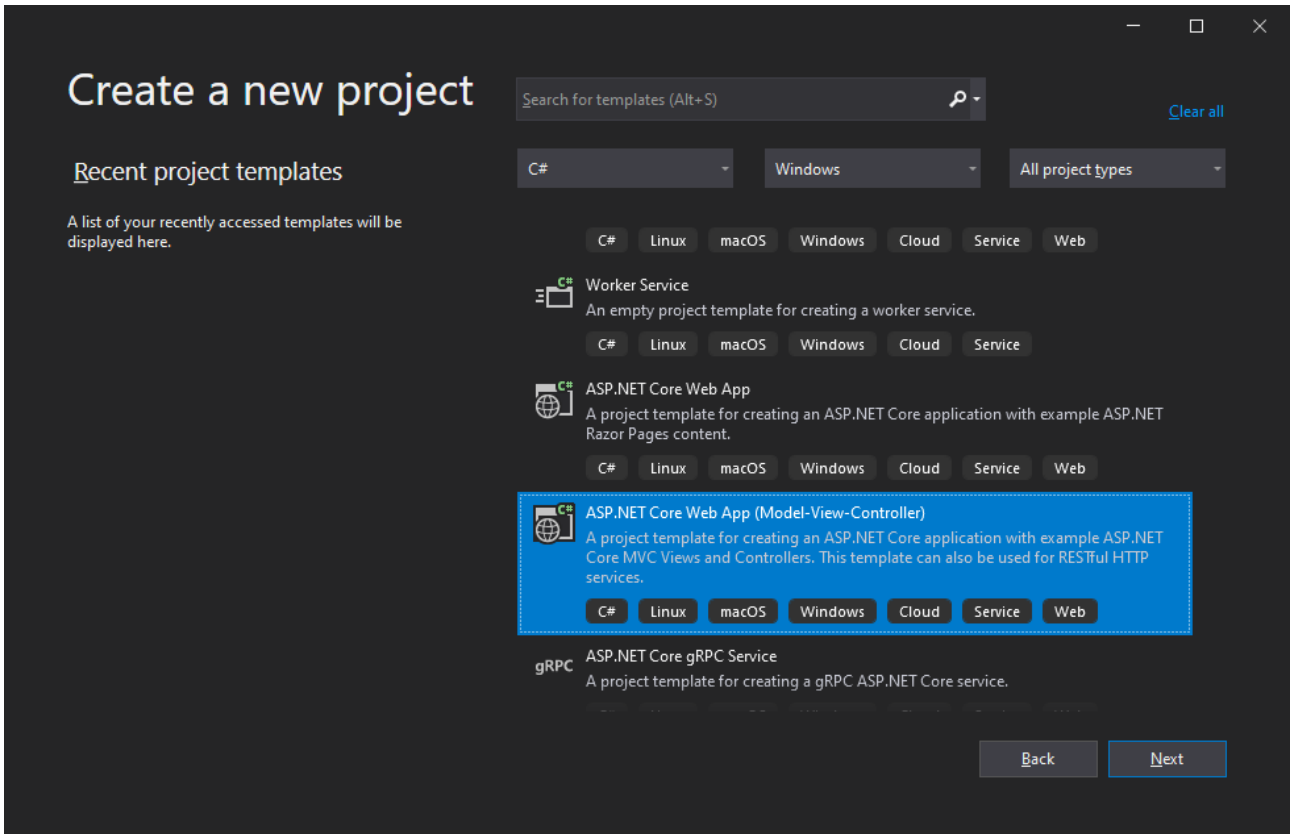
The exercise in this class explores the functioning of web forms. In the first phase, it uses a static form, in HTML, and demonstrates how it sends information to a controller.

In the 2nd phase of the exercise, a data model is defined that represents the information to be obtained on a form. In this case, during the processing of the submitted form, the fields are validated according to the restrictions defined for each one of them. Error messages are also displayed in the form in context with the errors found.

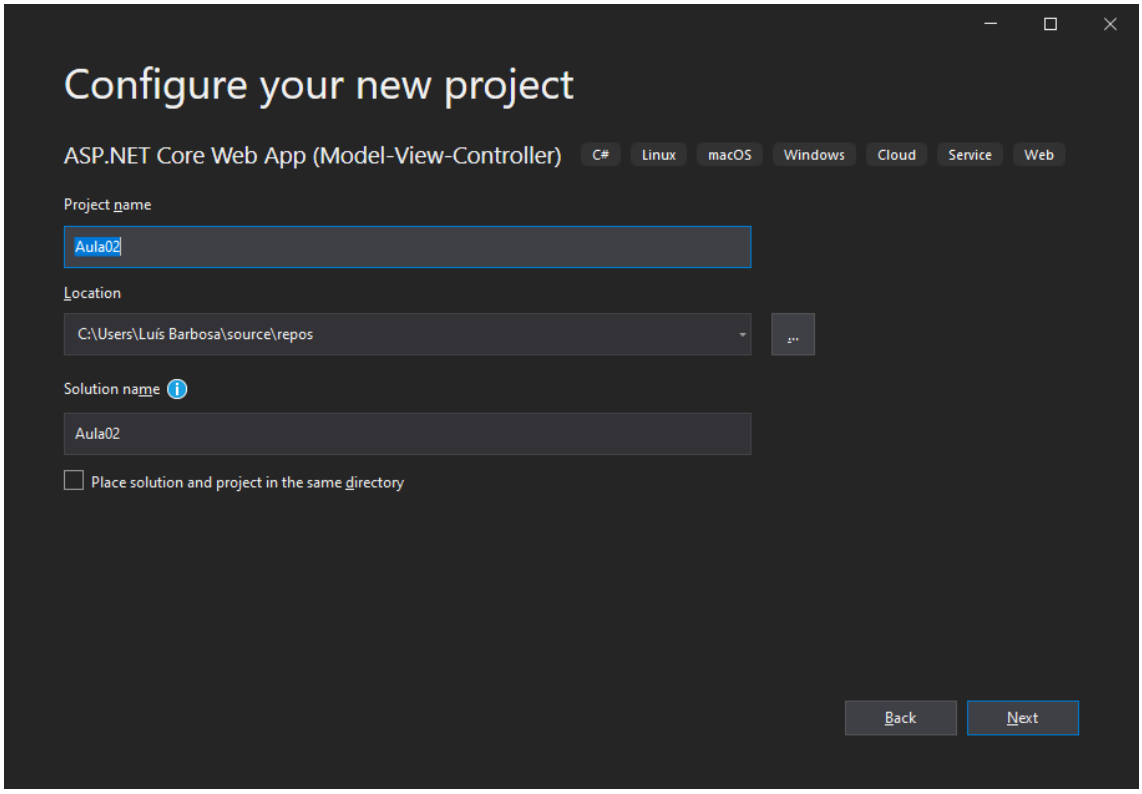
In the 3rd phase, the same exercise is built but using annotations in the data model. Annotations define, for example, restrictions on the data used in those fields (properties) of information. The validation of these fields, in this case, is done automatically when the class is instantiated.

Start to build a base web project:

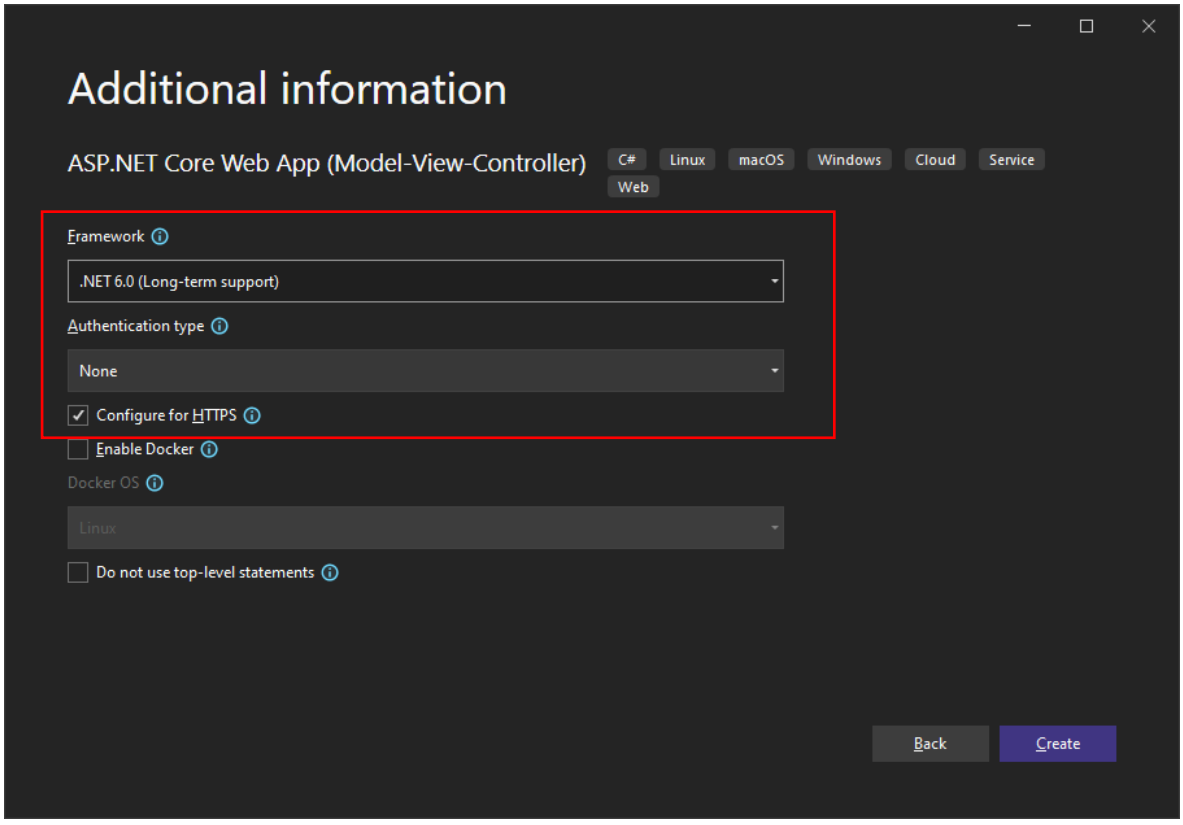
- From the Visual Studio select **Create a new project**.
- Select **ASP.NET Core Web App (Model-View-Controller)** and then select **Next**.



- Name the project **Aula02** and select **Next**. It's important to name the project **Aula02** so when you copy the code from this tutorial, the namespace will match.



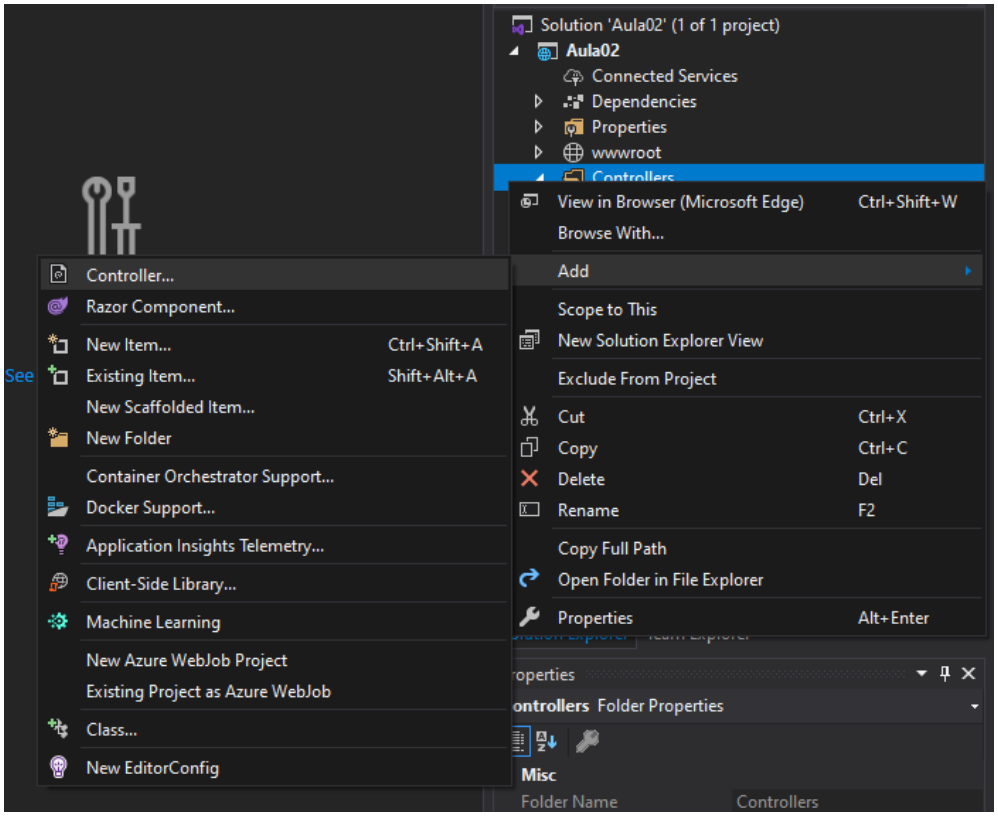
- Chose **.NET 6.0 (Current)** target framework, authentication type **None** and then select **Create**.



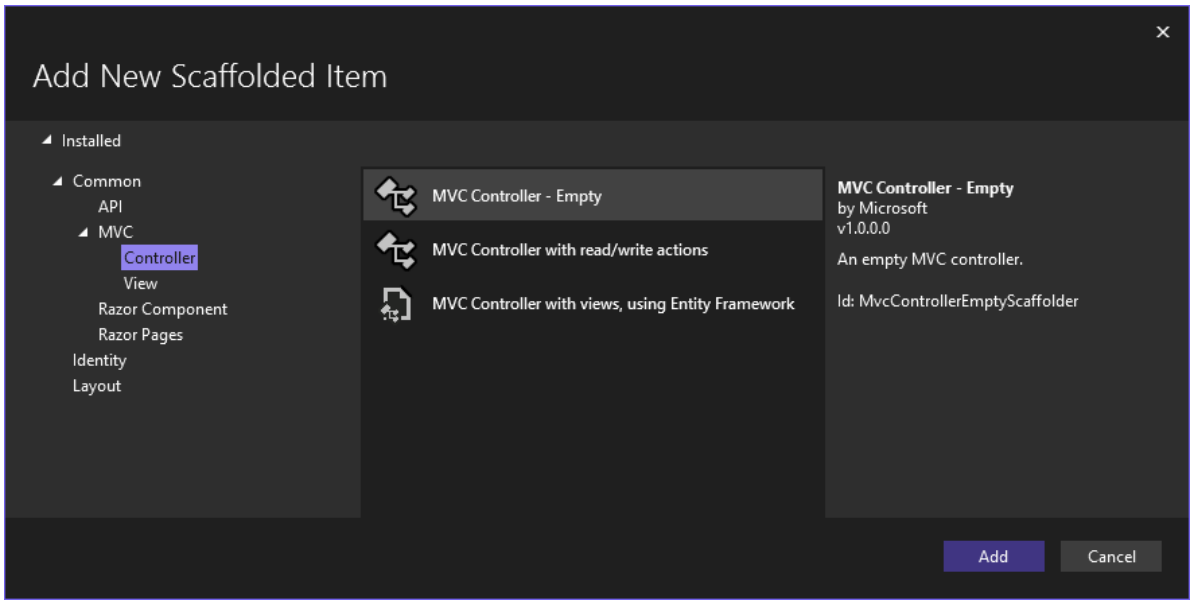
Visual Studio used the default template for the MVC project you just created. You have a working app right now by entering a project name and selecting a few options. This is a basic starter project.

1st customization step ----------

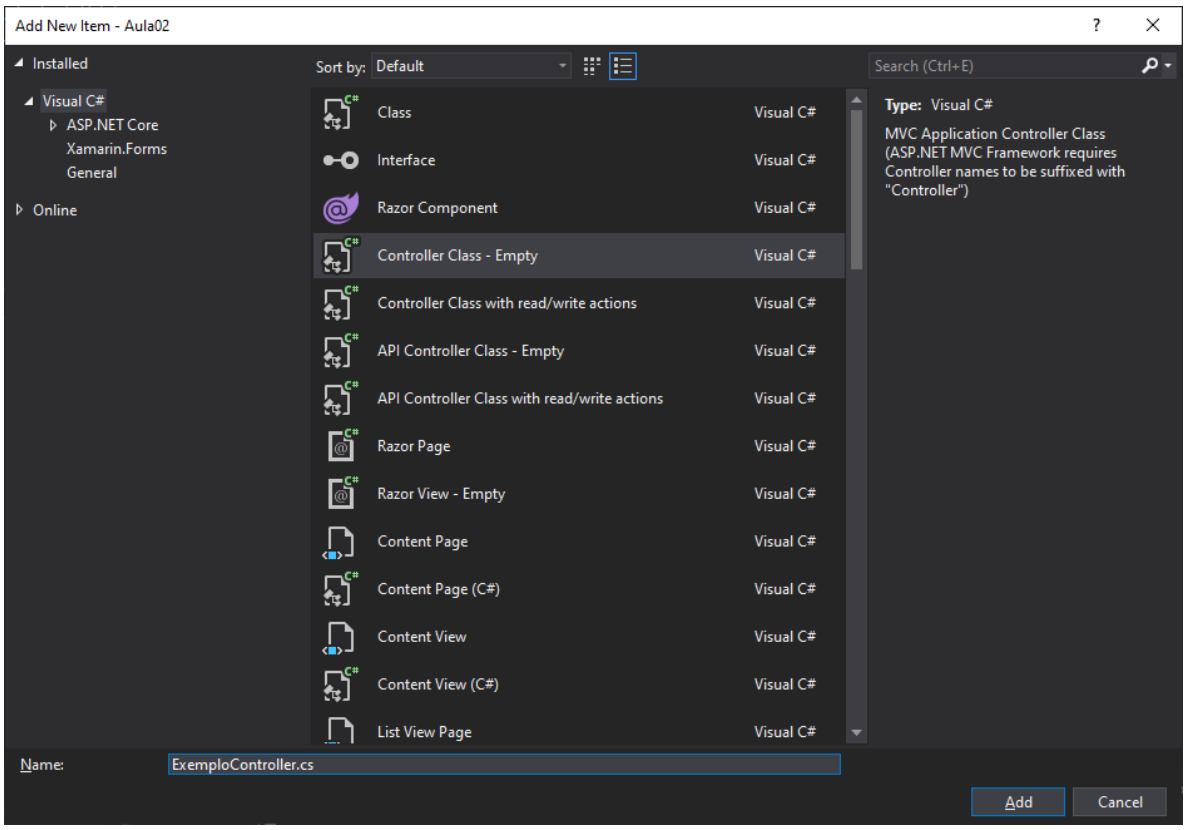
- Create an empty controller (with right mouse button over **Controller** folder in **Solution Explorer**).



- Choose **MVC Controller – Empty** to get a controller class with a simple Index action.



- Give name **ExampleController** to the controller class.



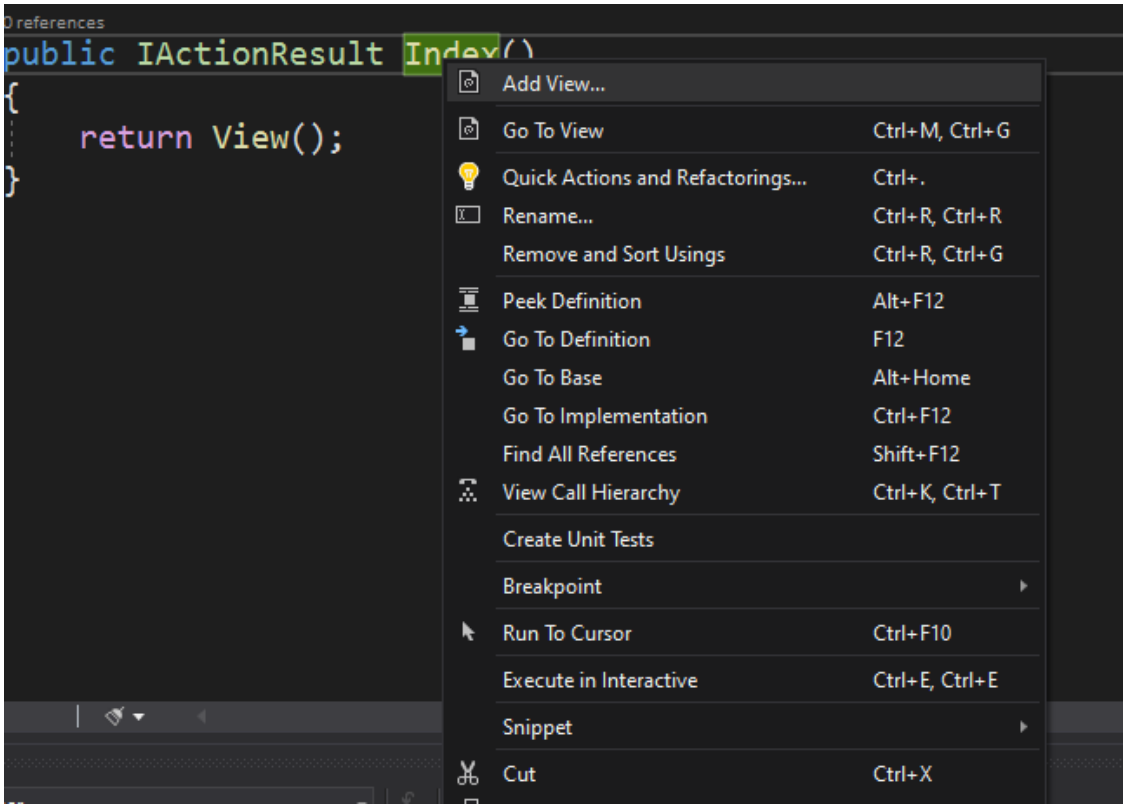
- Verify the class stored in **Controllers** folder in **Solution Explorer** tab.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;

namespace Aula02.Controllers
{
    0 references
    public class ExemploController : Controller
    {
        0 references
        public IActionResult Index()
        {
            return View();
        }
    }
}
```

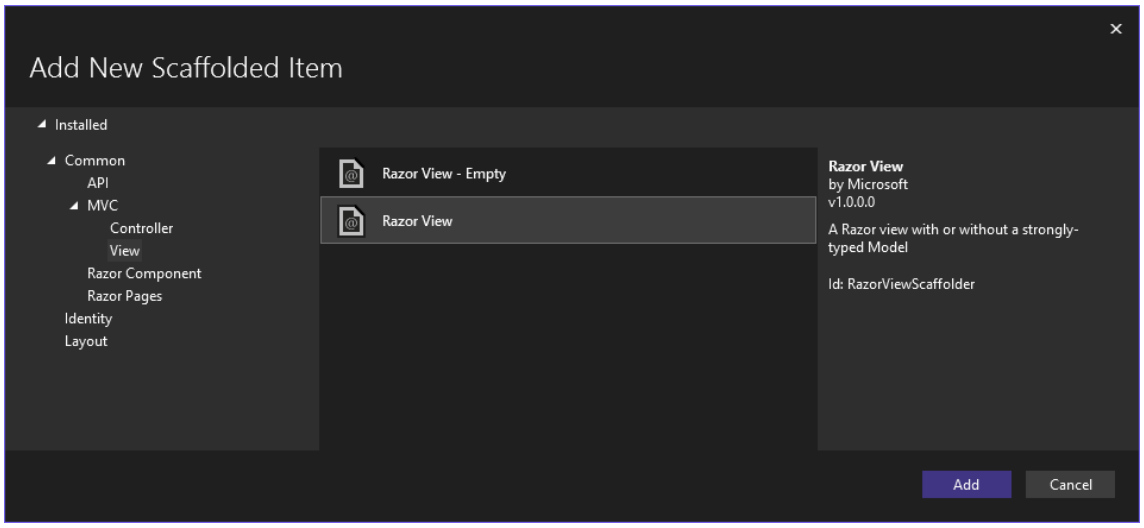
- Create an empty controller (with right mouse button over Controller folder).

Implement a page with a formulary for the **Index** action.

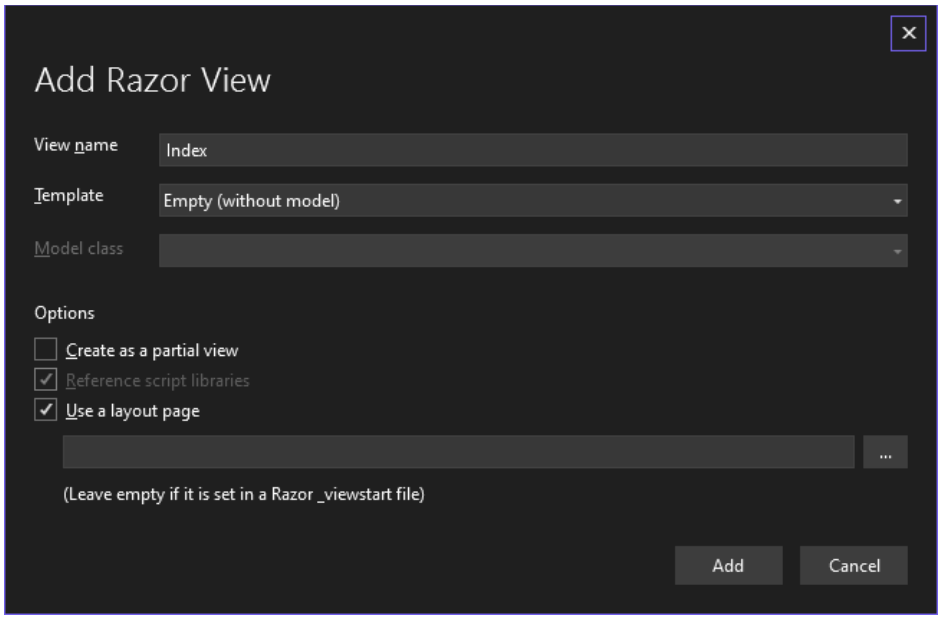
- Create a View with right mouse button over Index method.



- Choose the **Razor View** option…



- …and name the view file as suggested - **Index** (the same name as the method).



- Verify the generated file stored in **Views** folder, from **Solution Explorer**, inside a subfolder with same name as the controller (**Example**). The view file as the name of the action with **cshtml** file extension.

```
@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>
```

- Customize the view file adding the formulary HTML code.

```
@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<form method="POST" action="">
    <label>Name</label>
    <input type="text" name="name" />
    <label>Age</label>
    <input type="text" name="age" />
    <input type="submit" value="let's go submit this" />
</form>
```

- Add a new method to the controller, responsible for processing the form submission.

The new method must have same name of the form action property in the view file. As it has no text in it (**action=""**) the action is the same (Index).

For differing this action from the previous one we must add the **HttpPost** filter and **IFormCollection** parameter. The first method is used for GET requests and the second is for POST requests.

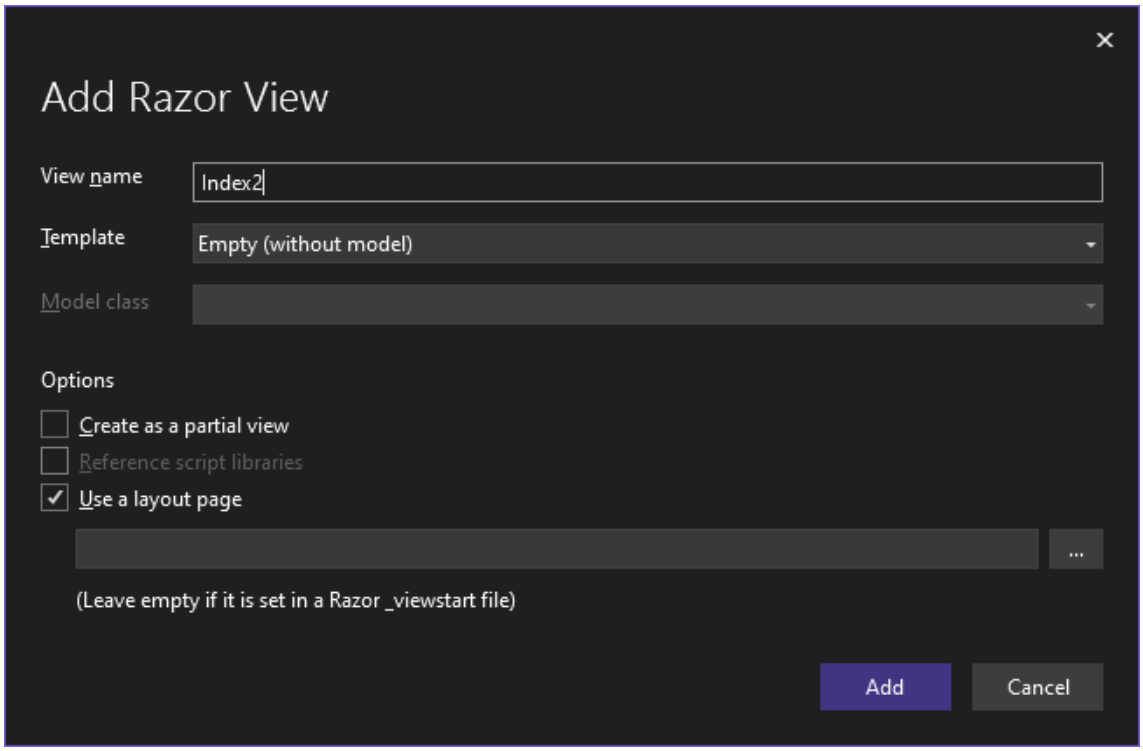
```
    public IActionResult Index()
    {
        return View();
    }

    [HttpPost]
    0 references
    public IActionResult Index(IFormCollection formData)
    {
        return View();
    }
}
```

- Add some **ViewData** fields to transfer data from submitted form to the result view.

ViewData is a Dictionary structure with scope controller and view processing. It can be used to transfer data between controller and view.

```
[HttpPost]
0 references
public IActionResult Index(IFormCollection formData)
{
    // action used to process the form submition
    ViewData["text_inserted"] = formData["name"]; // transfer datas to the View
    ViewData["other_text_inserted"] = formData["age"]; // transfer datas to the View

    return View("Index2"); // uses another view instead using the default view name
                           //(usualy as the same name as the method - Index)
}
```

- Add another view file, now with name **Index2**, to present the submission result.



- Add custom **Razor** code to present data received through **ViewData** structure.

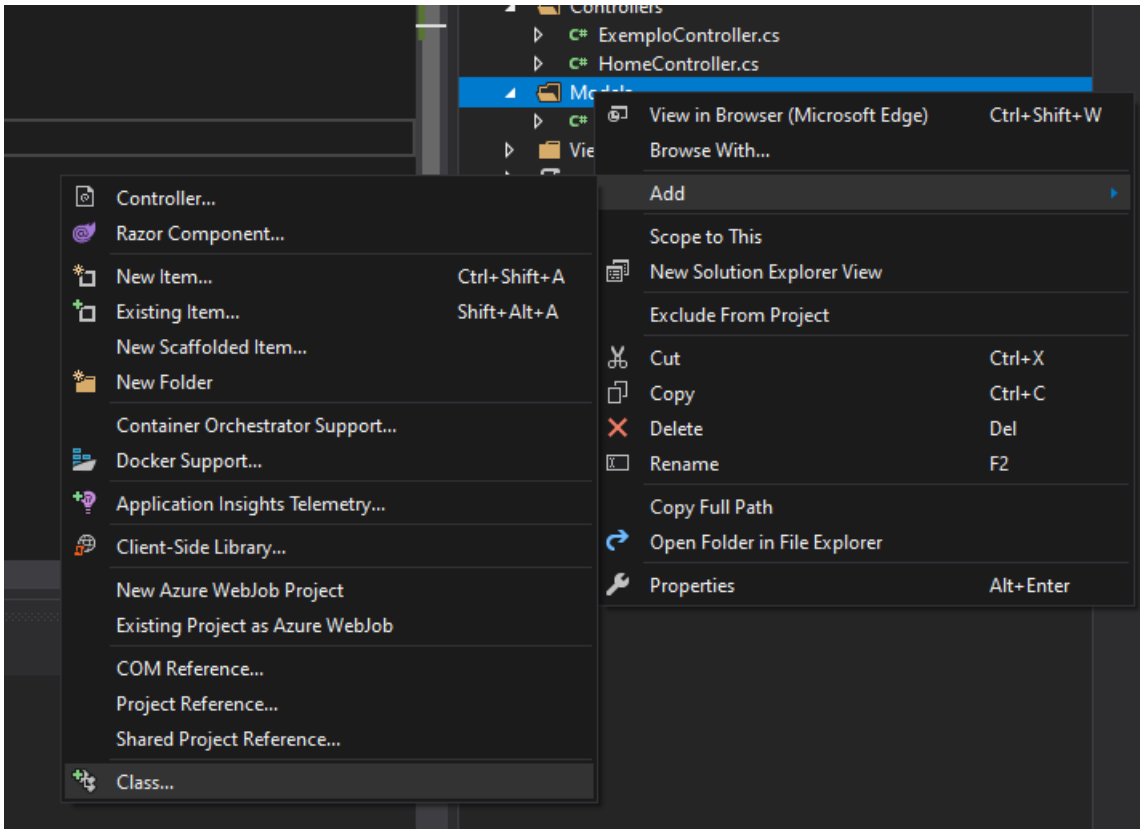
```
@{
    ViewData["Title"] = "Index2";
}

<h1>Index2</h1>

Data inserted in FORM
<br />
Name field : @ViewData["text_inserted"]<br />
Age field : @ViewData["other_text_inserted"]
```

Now you can try it accessing **/Example/Index** resource. ✓

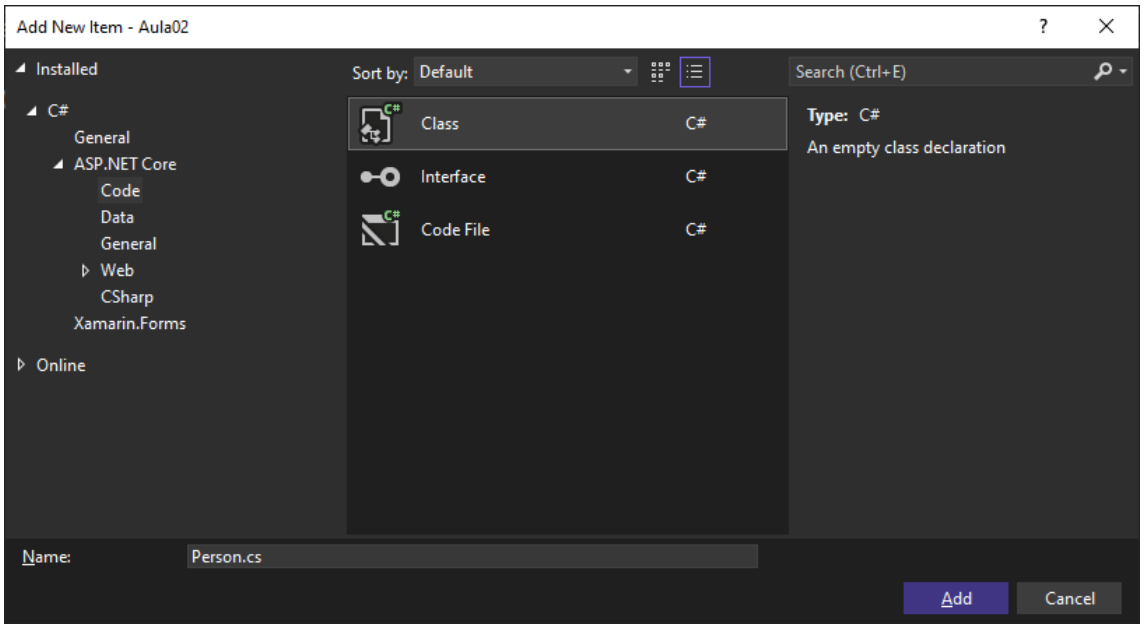
2nd step ----------

Implement a page with a formulary generated from a model class.

- Create a model class (with right mouse button over **Models** folder in **Solution Explorer**).



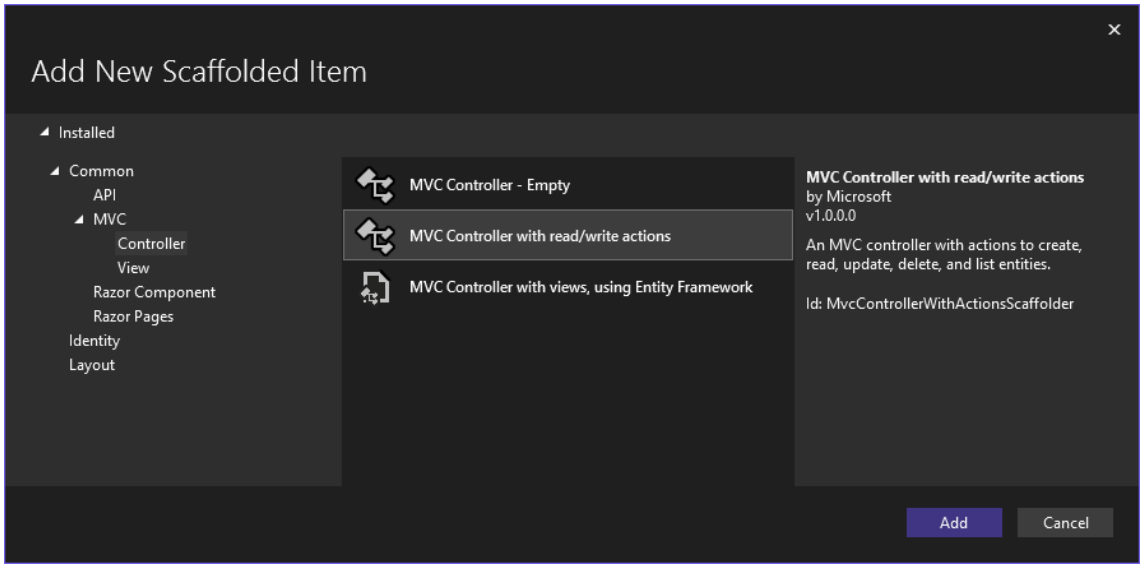
- Choose **Person** name to the file.



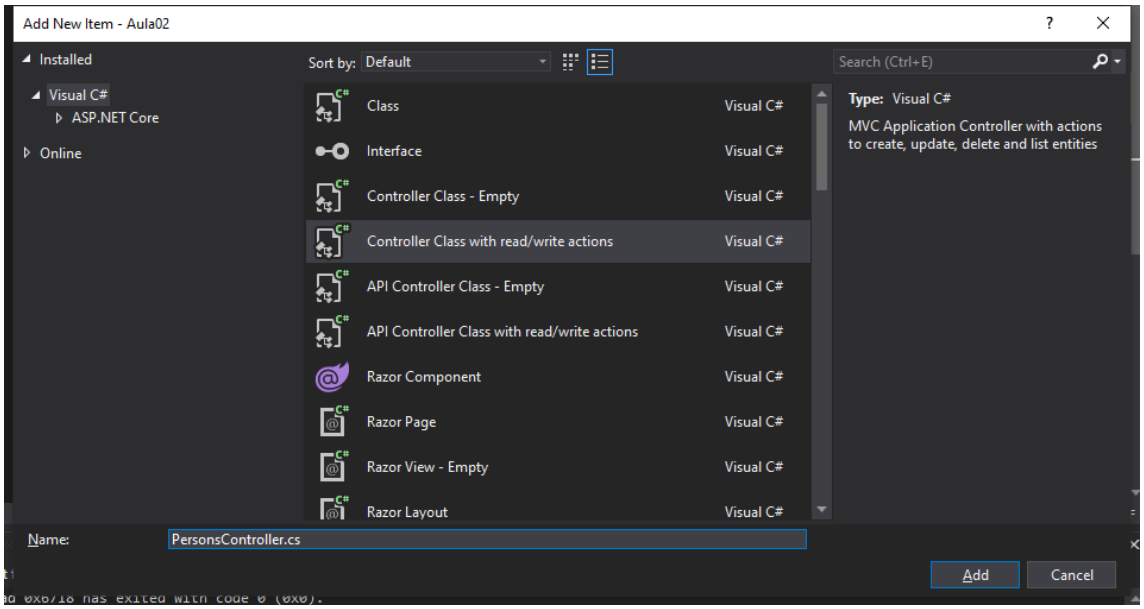
- In the generated file, add the properties corresponding to the class fields (**Name** and **Age**).

```
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Threading.Tasks;
5
6   namespace Aula02.Models
7   {
        0 references
8       public class Person
9       {
            0 references
10          public string Name { get; set; }
            0 references
11          public int Age { get; set; }
12      }
13  }
14
```

- Add a controller, based on model Person, with read/write operations.

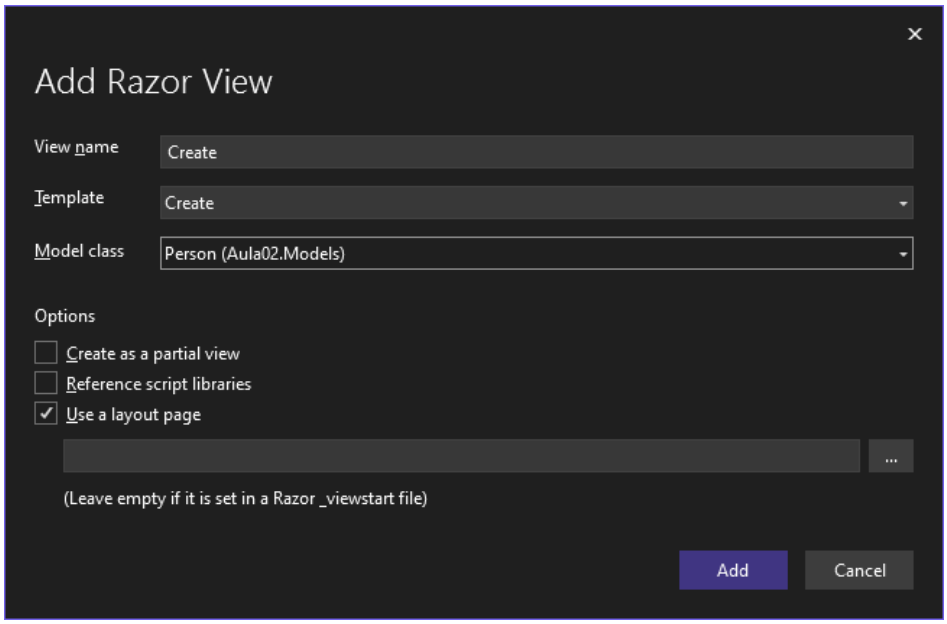


- Name it **PersonsController**.



This process generates a controller with all CRUD operations over **Person** data (**C**reate, **R**etrieve, **U**pdate and **D**elete). For our exercise we will use only the **Create** action.

- Add a view to Create action, based on create data operation - **Template** (it generates a view with a formulary based in all Person fields/properties – **Model class**).



Verify in generated view file, where we can see all the HTML code corresponding to the FORM element. This code have **Razor** instructions and a lot of **Tag Helper attributes**.

In the FORM element, the `asp-action="Create"` indicates the method that will process the form submission.

For each one of the **Person** properties, the form have three elements: a label, an input field and a text span. They use Tag Helpers attributes `asp-for="?"` and `asp-validation-for="?"` important to associate those elements to the Person properties.

```
<h1>Create</h1>

<h4>Person</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Create">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Name" class="control-label"></label>
                <input asp-for="Name" class="form-control" />
                <span asp-validation-for="Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Age" class="control-label"></label>
                <input asp-for="Age" class="form-control" />
                <span asp-validation-for="Age" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>
```

- Add c# code in **Create** action to validate the data submitted. Use **ModelState** structure no store all error messages found in data analysis. These error messages will be shown automatically in view after return to form.

```
public ActionResult Create(IFormCollection collection)
{
    try
    {
        if (string.IsNullOrEmpty(collection["name"]) == true)
        {
            ModelState.AddModelError("name", "Mandatory field");
        }
        if (string.IsNullOrEmpty(collection["age"]) == true)
        {
            ModelState.AddModelError("age", "Mandatory field");
        }
        else
        {
            int aux;
            try
            {
                aux = int.Parse(collection["age"]);
                if (aux < 18 || aux > 100)
                {
                    ModelState.AddModelError("age", "Age should be between 18 and 100");
                }
            }
            catch (FormatException)
            {
                ModelState.AddModelError("age", "Must indicate an integer value");
            }
        }
```

The text messages inserted in **AddModelError** methods will be presented to user (in view) if it fails filling the form.

```
if (ModelState.IsValid)
{
    // process the information

    // transfers information to other action
    TempData["values"] = collection["name"] + " [" + collection["age"] + "]";
    return RedirectToAction(nameof(Index));
}
else
{
    // if model has errors, it return to the form view and show the errors
    return View();
}
```

To complete the action, after processing data fields, it returns to a different action using **`RedirectToAction(nameof(Index))`** where will be a view to show the submitted data. We must use the **TempData** structure to temporary store information between sequential action requests.

- Add a last view, for the Index action in Persons controller. It's an empty view (without model) with customized code.

```
1
2   @{
3       ViewData["Title"] = "Index";
4   }
5
6   <h1>Formulary submission</h1>
7
8   Submitted with SUCCESS!
9   <br />
10  It was inserted the following data : @TempData["values"]
11
```

3rd step ----------

Implement a feature like the previous one, where field padding control is done based on **DataAnnotations** in the model class.

- Alter the Person class definition adding data annotations to define restrictions to field's values.

```
using System.Threading.Tasks;
using System.ComponentModel.DataAnnotations;

namespace Aula02.Models
{
    6 references
    public class Person
    {
        [Required(ErrorMessage = "The field {0} is mandatory")]
        6 references
        public string Name { get; set; }

        [Required(ErrorMessage = "The field {0} is mandatory")]
        [Range(18, 100, ErrorMessage = "{0} must have values between {1} and {2}")]
        6 references
        public int Age { get; set; }
    }
}
```

After use these data annotations, we can change the form submission processing simply by verifying the **ModelState** validation. This process needs one **Person** instance to be created. In this action we must use a parameter on type **Person** instead of **IFormCollection**.

```
[HttpPost]
0 references
public ActionResult Create(Person newPerson)
{
    if (ModelState.IsValid)
    {
        // process the information

        // transfers information to other action
        TempData["values"] = newPerson.Name + " [" + newPerson.Age+ "]";
        return RedirectToAction(nameof(Index));
    }
    else
    {
        return View();
    }
}
```

With the data annotations, all field validations will be automatically done.

- Add **DisplayName** data annotation to customize label field in formulary (usually is used to generate a more friendly label other than property name).
- Add a new property to **Person** class of type **Datetime** and decorate it with the data annotation **`DataType(DataType.DateTime).`** This allows to generate a form view where the input field is a calendar (HTML5).
- Add a new property to **Person** class of type **string** and decorate it with the data annotation **`DataType(DataType.EmailAddress).`** This allows to validate input string with an email structure.