Tutorial for Class number 3

This exercise explores the submission of files (documents, images, etc.) through web forms. Emphasizes the essential characteristics of the form and the proper processing of the submitted data.

In the third part is demonstrated the necessary steps for the delivery of the stored documents (files download)

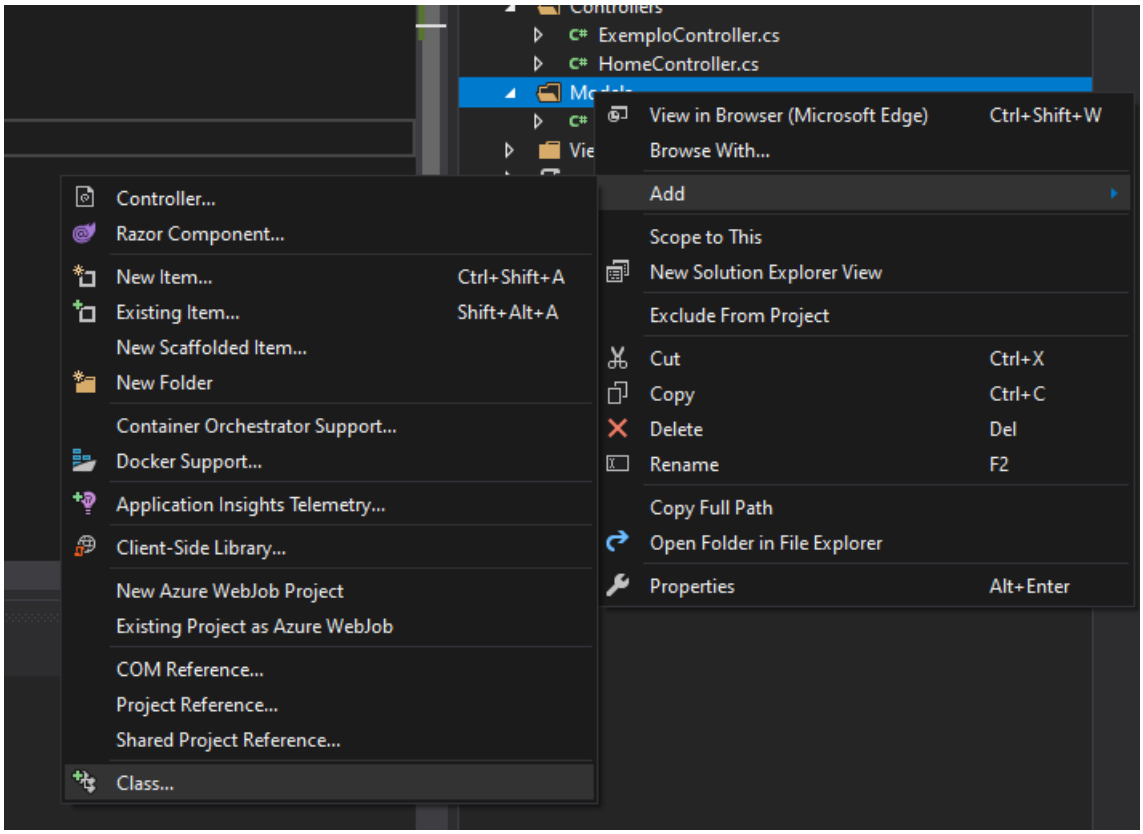
Upload and download files.

Start to build a base web project:

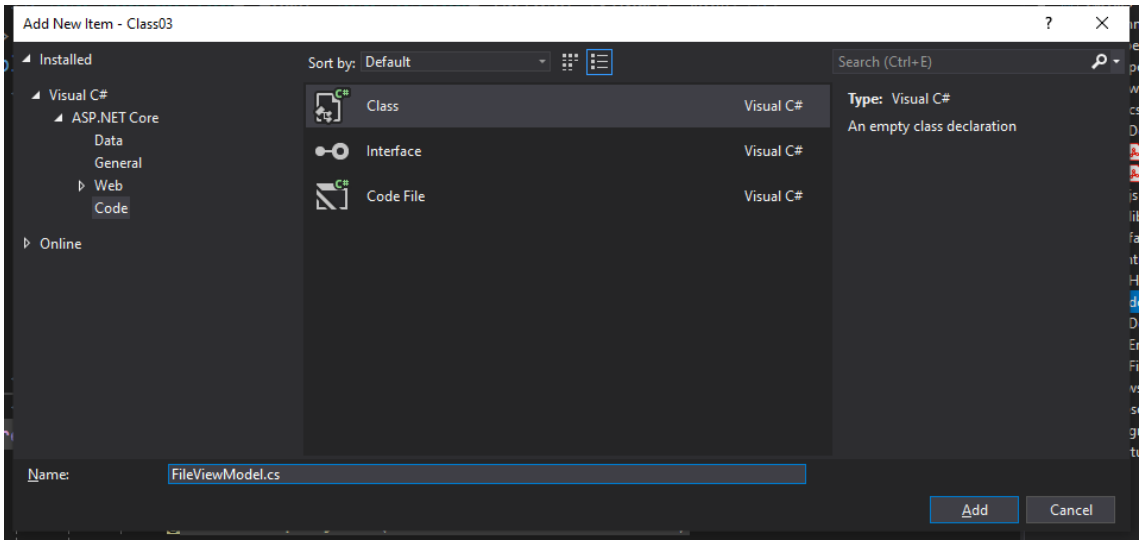
- From the Visual Studio, create a new project **ASP.NET Core Web App (Model-View-Controller)**. Name de project **Aula03.** (See Tutorial 02 to remember).

1st step:

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



- Use **FileViewModel.cs** name to the file.



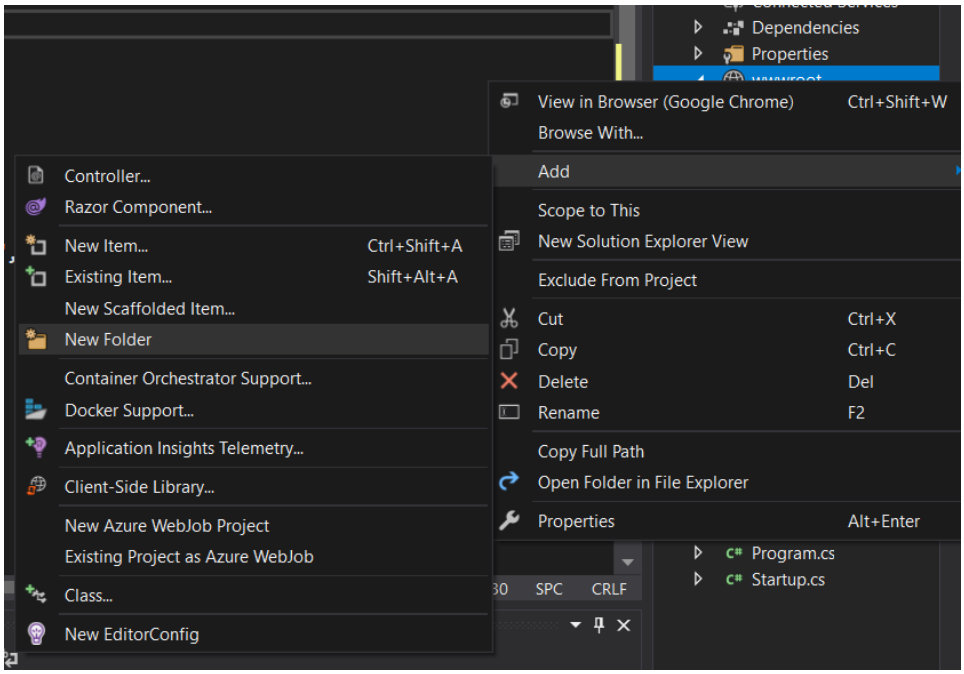
- In the generated file, add the properties corresponding to the class fields (**Name** and **Size**).

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

namespace Aula03.Models
{
    0 references
    public class FileViewModel
    {
        [Required]
        [RegularExpression(@"^.+\.([pP][dD][fF])$", ErrorMessage = "Only Pdf Files")]
        0 references
        public string Name { get; set; }

        [Display(Name="Size in Bytes")]
        0 references
        public long Size { get; set; }
    }
}
```

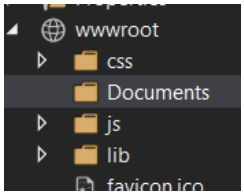
The data annotation **RegularExpression** validates the string **Name** with de regular expression presented. In this case, we use it verify if the filename ends with "**.pdf**" (as file extension).

More info at: https://en.wikipedia.org/wiki/Regular_expression

- Create a subfolder inside **wwwroot** folder (with right mouse button over **wwwroot** folder in **Solution Explorer**).



- Name it **Documents**



This is the folder where submitted files will be stored.

- Create a new class in **Model** folder. Name it **DocFiles**

```
public class DocFiles
{
    // this method gets all filenames from folder...
    //...instead of getting them from a database or other storage kind
    1 reference
    public List<FileViewModel> GetFiles(IHostEnvironment e)
    {
        List<FileViewModel> list = new List<FileViewModel>();

        // get all information from "Documentos" folder
        DirectoryInfo dirInfo = new DirectoryInfo(
                Path.Combine(e.ContentRootPath, "wwwroot/Documents")
            );

        // use the information from folder to get the filenames
        foreach (var item in dirInfo.GetFiles())
        {
            list.Add(new FileViewModel
            {
                Name = item.Name,
                Size=item.Length
            });
        }
        return list;
    }
}
```

This class have a method **GetFiles** to get the list of files in the folder *wwwroot/Documents*.

To access this folder in the disc of the server machine, is needed to use the **IHostEnvironment** interface. The **ContentRootPath** property gets the absolute path to the directory that contains the application content files.

- **2nd step**

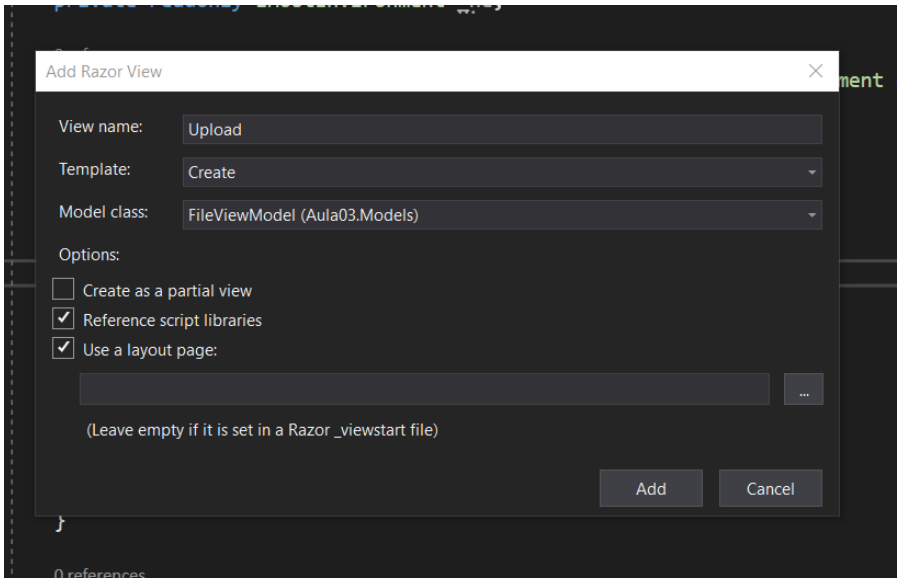
- Alter the **HomeController** to have a new property of the type **IHostEnvironment** and inject it in the constructor:

```
public class HomeController : Controller
{
    private readonly ILogger<HomeController> _logger;
    private readonly IHostEnvironment _he; // necessary to get application folder

    0 references
    public HomeController(ILogger<HomeController> logger, IHostEnvironment e)
    {
        _logger = logger;
        _he = e; // injects in the constructor information about hostEnvironment
    }

    0 references
    public IActionResult Index()
```

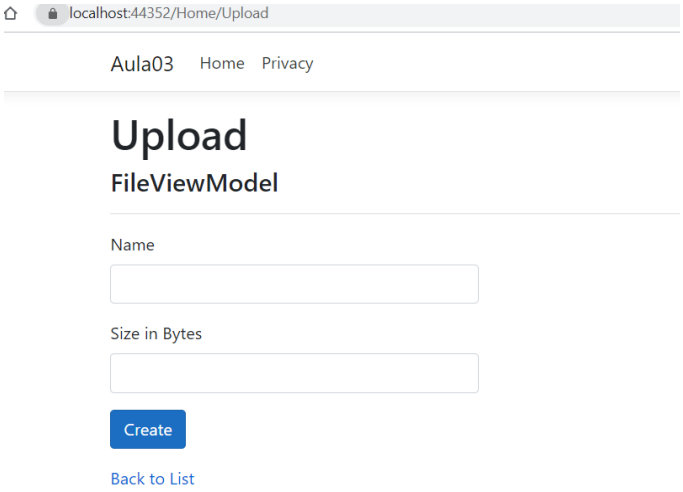
- Create a new action (method) **Upload**

```
0 references
public IActionResult Upload()
{
    return View();
}
```

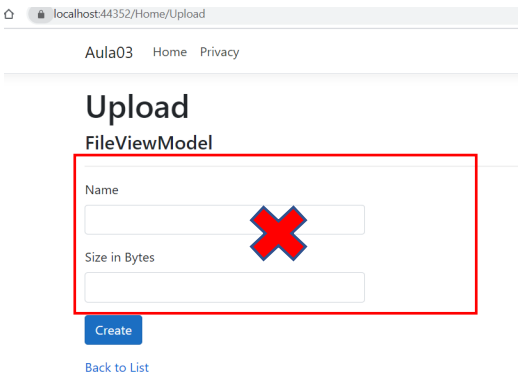
- Create a Razor View to this action, with template **Create** and based on **FileViewModel** model class:



- Test the View.



- **Alter** the view, adding `enctype="multipart/form-data"` property to `form` element to allow the formulary to send files to the web server.
- **Alter** the view, cutting the size field input (we don't need this field) and changing the input type of Name field to `type="file"`. This change defines a button to choose a file from the computer file system:



```
<h1>Upload</h1>

<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Upload" enctype="multipart/form-data">
            <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" type ="file" class="form-control" />
                <span asp-validation-for="Name" class="text-danger"></span>
            </div>

            <div class="form-group">
                <input type="submit" value="Submit" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>
```

- Test again:

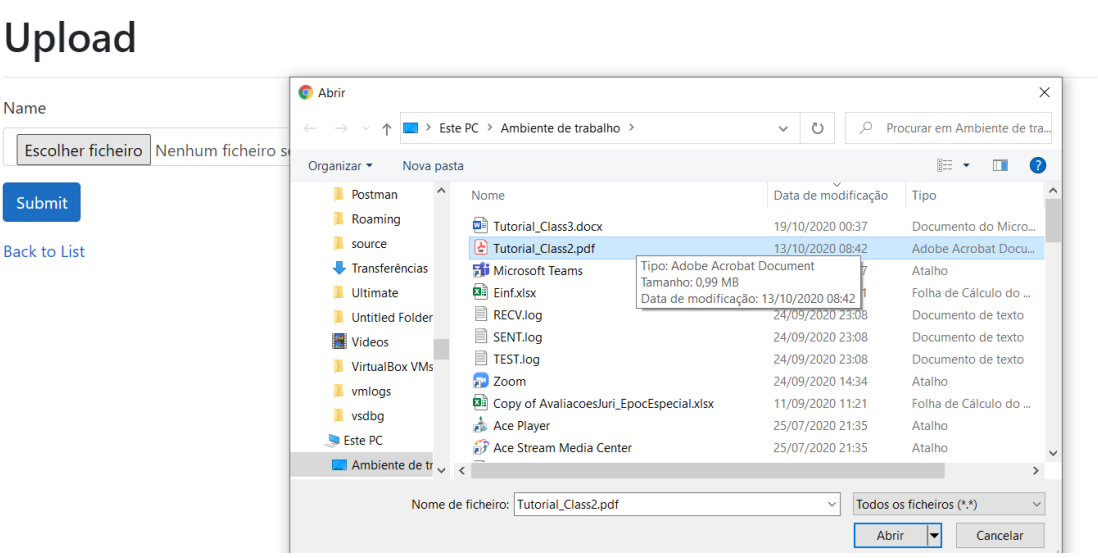
- Add a new method to the controller, responsible for processing the form submission.

This action needs an **`IFormFile`** instance parameter for each uploaded file.

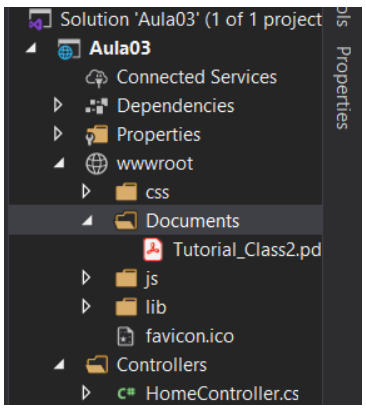
Another solution is to use an instance of the **`IFormCollection`** class, which has a property named **`Files`**, consisting in a collection of **`IFormFile`**, where all submitted files are stored.

```
[HttpPost]
0 references
public IActionResult Upload(IFormFile Name)
{
    // other file properties could be checked here but we assume everything is OK
    if (ModelState.IsValid) {
        string destination = Path.Combine(
                _he.ContentRootPath, "wwwroot/Documents/", Path.GetFileName(Name.FileName)
            );

        //creates a filestream to store the file bytes
        FileStream fs = new FileStream(destination, FileMode.Create);

        Name.CopyTo(fs);
        fs.Close();

        // after saving file, redirects to file listing
        return RedirectToAction("Index");
    }
    else{
        return View();
    }
}
```

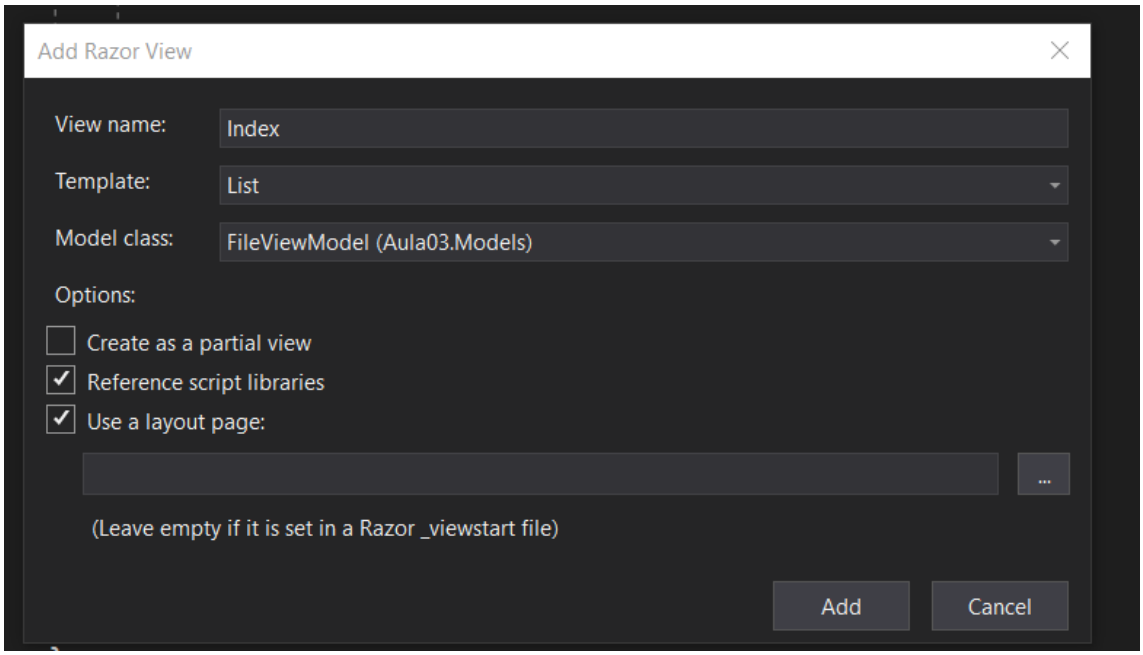
- Teste Upload again:



- Confirm if the file is in the Documents folder:



- Create a view to the **Index** action, with template **List** and based on **FileViewModel** model class:



Opps…Index file view already exists? Yes…replace it.

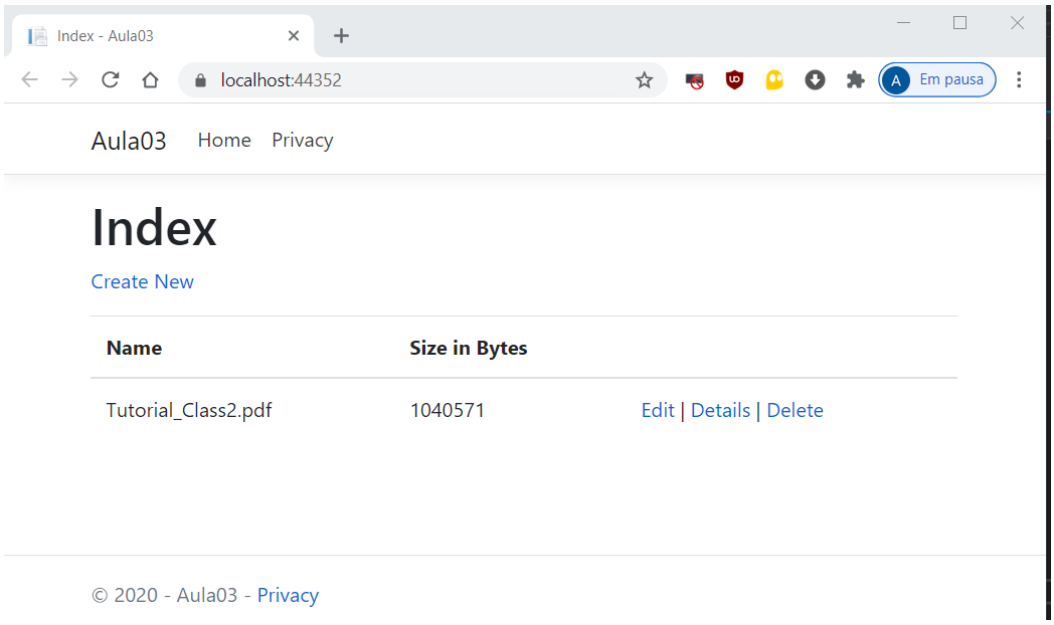
The view with **List** template needs a list of elements, in this case **FileViewModel**, to work (see next picture).

```
@model IEnumerable<Aula03.Models.FileViewModel>

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

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Name)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Size)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model) {
```

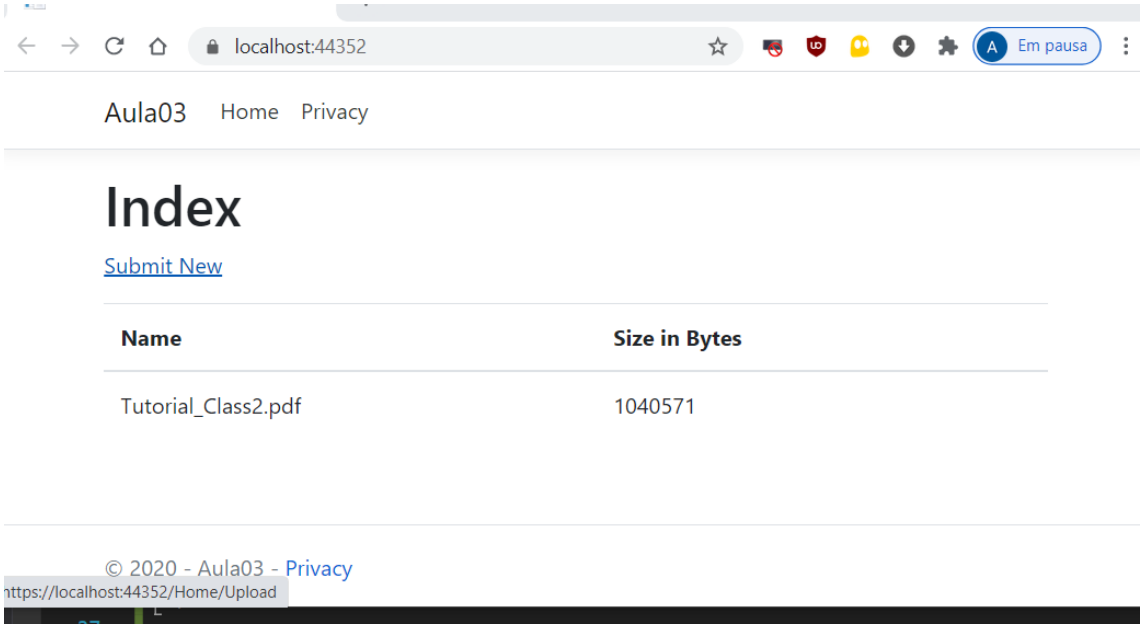
For this reason, we need to inject the correct collection by using the list as argument of `return View()` instruction:

```
public IActionResult Index()
{
    //get the information of the files in the Documents folder
    //using the classe DocFiles
    DocFiles files = new DocFiles();


    return View(files.GetFiles(_he));
}
```

- Run the application:



- Alter the link “Create New” to presents the text “Submit New” and to navigate to the action **Upload**, and remove the links “Edit”, “Details” and “Delete” in each listed file.



To change the link, we only need to change `asp-action` value because the new method is on same controller class. If it is on another controller, we also need to add the link to the `asp-controller` property with the appropriate value (you can try it in this exercise by using it with `"Home"` value).

- **3rd step**

Download Files

- Create a new Action **Download**.

```
public IActionResult Download(string id)
{
    // 'id' is the filename
    string pathFile = Path.Combine(_he.ContentRootPath, "wwwroot/Documents/", id);

    byte[] fileBytes = System.IO.File.ReadAllBytes(pathFile);

    string mimeType;
    // this code assumes that content type is always obtained.
    // Otherwise, the result should be verified (boolean value)
    new FileExtensionContentTypeProvider().TryGetContentType(id, out mimeType);

    return File(fileBytes, mimeType);
}
```

The parameter **id** is the name of the file.

The **mimeType** variable represent the mime type value of the file contents, who is needed to pass to the browser to process it.

More info: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types/Common_types

- Alter de View **Index** to create a link to the **Download** Action.

```
@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Name)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Size)
        </td>
        <td>
            <a asp-controller="Home"  asp-action="Download" asp-route-id=@item.Name target="_blank">Download</a>
        </td>
    </tr>
}
```

The **asp-route-id=@item.Name** is the parameter to pass to the Download action (Download(string id)).

target=”_blank” open a file in a new page in the browser.

Homework:

- Alter de View **Index** to show the number of Documents and the total Bytes.

- Add a link and the correspondent Action to Delete a Document.