Tutorial for Class number 6

This exercise intends develop a web application that applies Endpoint definitions to mask resources and simultaneously validate their fields.

The application uses data already stored in a database system. The database system operations will be supported by **Entity Framework Core** and generated by the process **Database First**.

You can jump immediately to 2nd step by cloning this repository
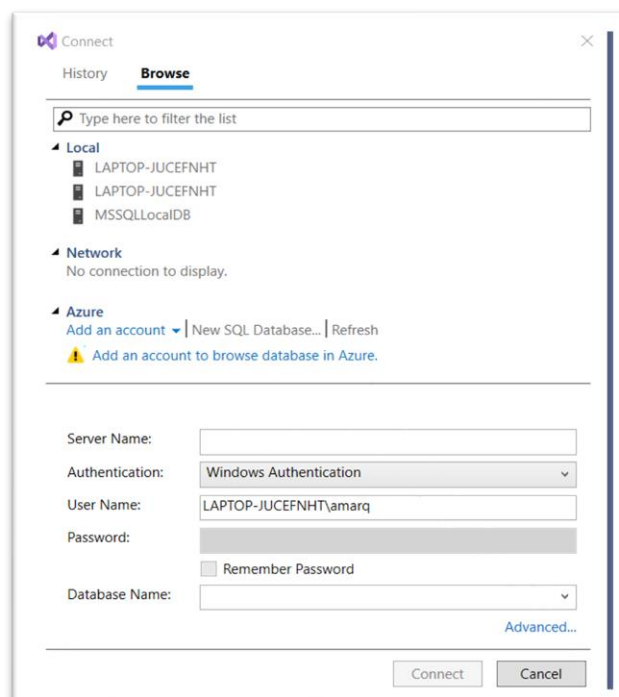**https://github.com/Engenharia-Web24-25/Tutorial_Class06_Preparation/**

First step - Preparing que database

- Create a new "ASP.NET Core Web App (Model-View-Controller)"

SQL Server Explorer to connect to database server (LocalDB or other)
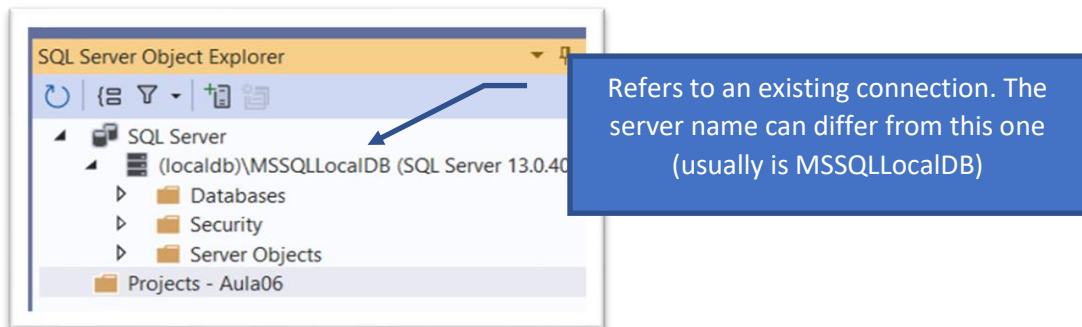
If you don't have any connection to a SQL Server or if you want to use another one, choose Add Sql Server to database option.
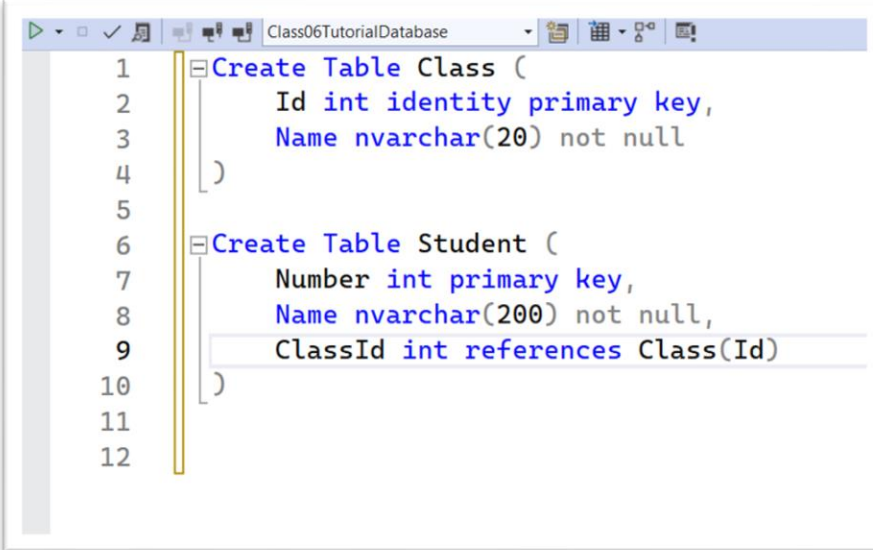
- Browse the desired connection and, on Databases folder, use the mouse right button do "Add New Database" with name **Class06TutorialDatabase**.
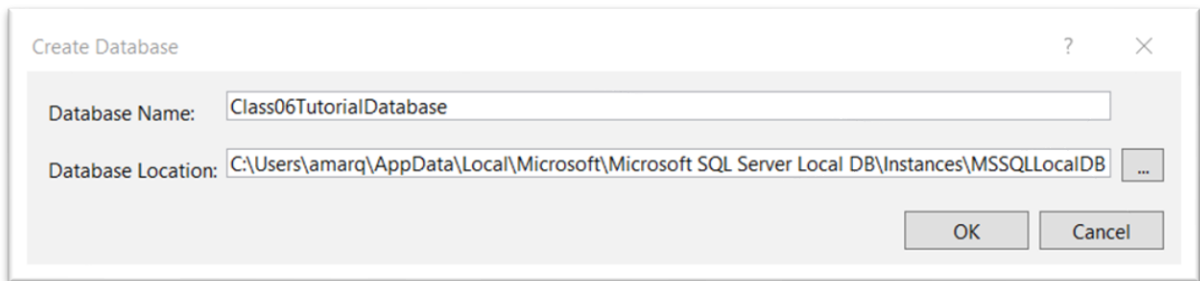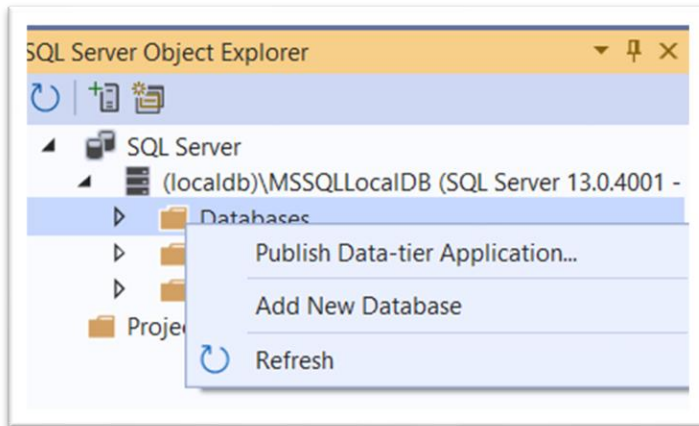
- After the database is created, using the mouse right button over it, choose "New Query" option.
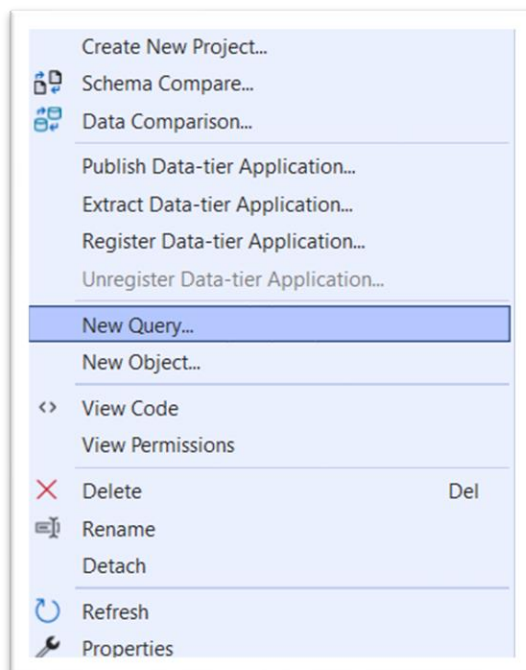
- Execute the following SQL script to create the database tables reflecting two entities that relate.

- Executing the script and confirm the created tables browsing in "SQL Server Explorer" as shown in next picture.
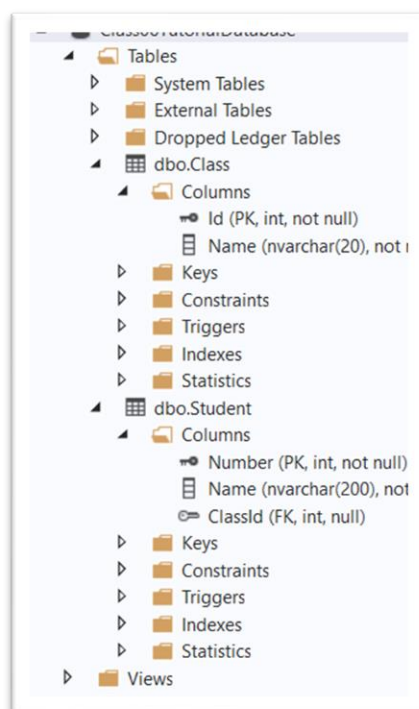
- In "Solution Explorer" edit the **appsettings.json** file to define the connection string necessary to connect de database (attention to server information).

Use same SQL Server reference as in previous step.

```
"AllowedHosts": "*",
"ConnectionStrings": {
  "Class06Context": "Server=(localdb)\\mssqllocaldb;Database=Class06TutorialDatabase;Trusted_Connection=True;MultipleActiveResultSets=true"
}
```
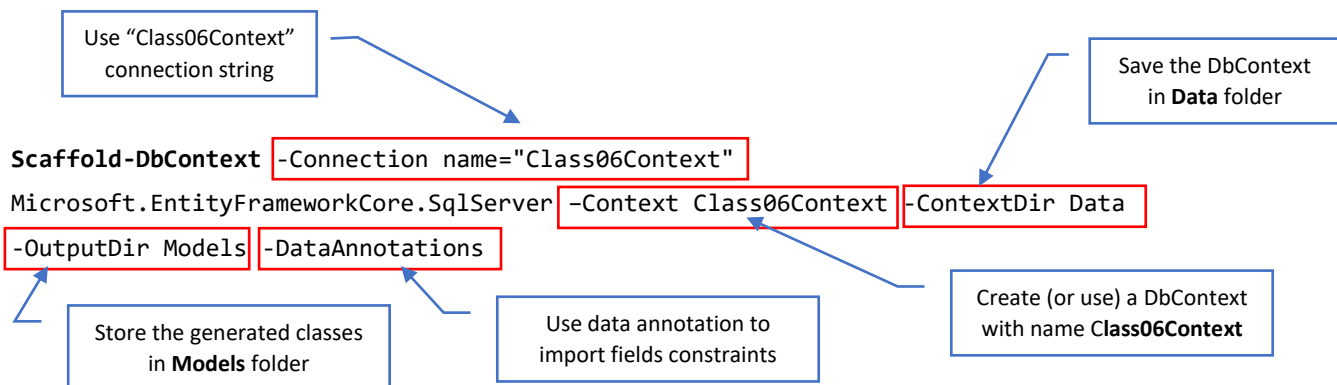
Now it's time to generate the data model from the existing database. For this we will use the "Package Manager Console".

Before importing the classes, we must ensure that we have the packages **EntityFrameworkCore.Tools** and **EntityFrameworkCore.SqlServer** installed in the project. If you don't have then, execute the following commands:

```
Install-Package Microsoft.EntityFrameworkCore.Tools

Install-Package Microsoft.EntityFrameworkCore.SqlServer
```

- Now we are able to scaffold the database structure, by executing the next command (the presented 3 text lines are a single command):

Use "Class06Context" connection string

Save the DbContext in **Data** folder

Scaffold-DbContext -Connection name="Class06Context"

Microsoft.EntityFrameworkCore.SqlServer –Context Class06Context -ContextDir Data

-OutputDir Models -DataAnnotations

Store the generated classes in **Models** folder

Use data annotation to import fields constraints

Create (or use) a DbContext with name **Class06Context**

Copy full command here: Scaffold-DbContext -Connection name="Class06Context" Microsoft.EntityFrameworkCore.SqlServer –Context Class06Context -ContextDir Data -OutputDir Models -DataAnnotations

The DbContext generated from this process have some code we will not use and, because of that, we will remove it.

- Edit the **Class06Context** class and comment the `OnConfiguring`, `OnModelCreating` and `OnModelCreatingPartial` methods. After this, the class should only have the following code.

```
public partial class Class06Context : DbContext
{
    public Class06Context()
    {
    }

    public Class06Context(DbContextOptions<Class06Context> options)
        : base(options)
    {
    }

    public virtual DbSet<Class> Classes { get; set; }

    public virtual DbSet<Student> Students { get; set; }

    // protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder) ...
}
```

We need to registry this class to allow the dependency injection in the controllers.

- In file **Program.cs** write the following code into the **CreateBuilder** process code:

```
// Add services to the container.
builder.Services.AddControllersWithViews();

builder.Services.AddDbContext<Class06Context>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("Class06Context") ));
```

If we intend to keep control of data model and subsequently have the chance to change the data model itself from the project, we must make a migration.

Still in "Package Manager Console" execute the command:

`Add-migration FirstStep`

Because this first migration does not take into account the existing database, so that it can be used later, it is necessary to **eliminate the code inside the Up and Down** methods so that the application or removal of the migration does not destroy the integrity of the existing database. Thus, new migrations can be added without generating application problems (updating the database).

```
public partial class FirstStep : Migration
{
    /// <inheritdoc />
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        //migrationBuilder.CreateTable(
        //    name: "Class",
        //    columns: table => new
        //    {
        //        Id = table.Column<int>(type: "int", nullable: false)
        //            .Annotation("SqlServer:Identity", "1, 1"),
        //        Name = table.Column<string>(type: "nvarchar(20)", maxLength: 20, nullable: false)
        //    },
        //    constraints: table =>
        //    {
        //        table.PrimaryKey("PK_Class", x => x.Id);
        //    });
```

```
protected override void Down(MigrationBuilder migrationBuilder)
{
    //migrationBuilder.DropTable(
    //     name: "Student");

    //migrationBuilder.DropTable(
    //     name: "Class");
}
```

Now we need to initialize the database data. Initialize the database with the file "StudentsList_Class06.txt" and containing the structure **<number>;<name>;<class>** in each line of information.

- Add the file to que **Data** folder (use Add Existing Item).

```
◢  📁 Data
   ▷  C# Class06Context.cs
      📄 StudentsList_Class06.txt
▷  📁 Migrations
```
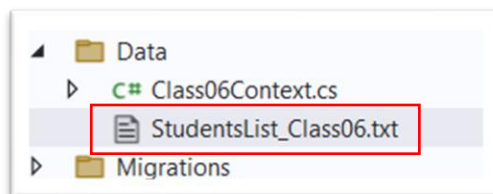
- In same folder, add a class with name **DbInitialiser** and with following code:

```
public class DbInitializer
{
    private readonly Class06Context _context;

    public DbInitializer(Class06Context context) { _context = context; }

    public void Run()
    {
        _context.Database.EnsureCreated();

        // Look for any student.
        if (_context.Students.Any())
        {
            return;    // DB has been seeded
        }


        var classes = new List<Class>();
        var students = new List<Student>();

        using (StreamReader sr = File.OpenText("Data\\StudentsList_Class06.txt"))
        {
            string line;
            while ((line = sr.ReadLine()) != null)
            {

                string[] parts = line.Split(";");

                if (!classes.Any(x => x.Name.Equals(parts[2])))
                {
                    classes.Add(new Class { Name = parts[2] });
                }
                students.Add(new Student
                {
                    Number = Int32.Parse(parts[0]),
                    Name = parts[1],
                    Class = classes.SingleOrDefault(x => x.Name == parts[2])
                });
            }
            _context.Classes.AddRange(classes);
            _context.Students.AddRange(students);
            _context.SaveChanges();
        }

    }
}
```

- Finally, in **Program.cs** file, change the code to register the **DbInitializer** class and to launch the application.

```
builder.Services.AddDbContext<Class06Context>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("Class06Context") ));

builder.Services.AddTransient<DbInitializer>();

var app = builder.Build();

using var scope = app.Services.CreateScope();
var services = scope.ServiceProvider;

var initializer = services.GetRequiredService<DbInitializer>();
initializer.Run();
```

Now we are ready to start writing code for the web application.

If you jumped 1st step, you should start from here, executing the command **update-database** in Package Manager Console *(there should be no errors resulting from this operation)*. If not, ignore this alert.

Second step – Building the actions

- Add to project an "MVC Controller – Empty" with name **StudentsController.cs**.
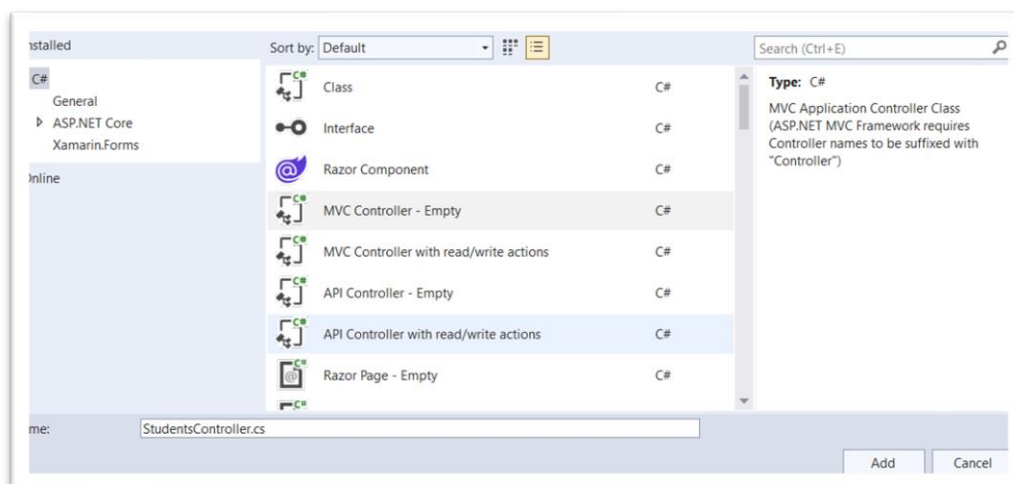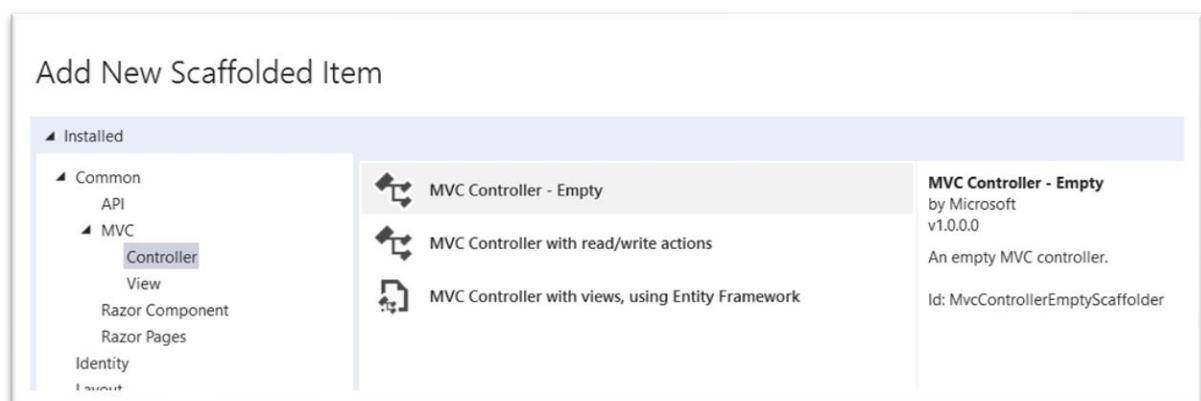
- Adapt the generated code, including the dependency injection of the DbContext in the constructor of the controller and adapting the **Index** action for listing of the elements of the **Students** entity.

```
using Aula06.Data;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;

namespace Aula06.Controllers
{
    public class StudentsController : Controller
    {
        private readonly Class06Context _context;
        public StudentsController(Class06Context context)
        {
            _context = context;
        }
        public IActionResult Index()
        {
            var allStudents = _context.Students.Include(c=>c.Class);
            return View(allStudents);
        }
    }
}
```

- Add a View (Razor View) for the Index action without any associated Template or Model.

Add Razor View

View name	Index
Template	Empty (without model)
Model class	
DbContext class	
Database provider	

Options

- [ ] Create as a partial view
- [✓] Reference script libraries
- [✓] Use a layout page

(Leave empty if it is set in a Razor _viewstart file)

Add   Cancel

- Adapt the generated code to reference a model (**Student** collection) and to make use of a partial view called **Listing** that uses the same data model.

```
@model IEnumerable<Aula06.Models.Student>

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


<h1>Students list</h1>

<partial name="Listing" model="Model" />
```

- Create the **Listing** view as partial and based on the **List** template for model **Student**.

Add Razor View

View name	Listing
Template	List
Model class	Student (Aula06.Models)
DbContext class	Class06Context (Aula06.Data)
Database provider	Configured from the selected DbContext

Options

☑ Create as a partial view
☐ Reference script libraries
☑ Use a layout page

(Leave empty if it is set in a Razor _viewstart file)
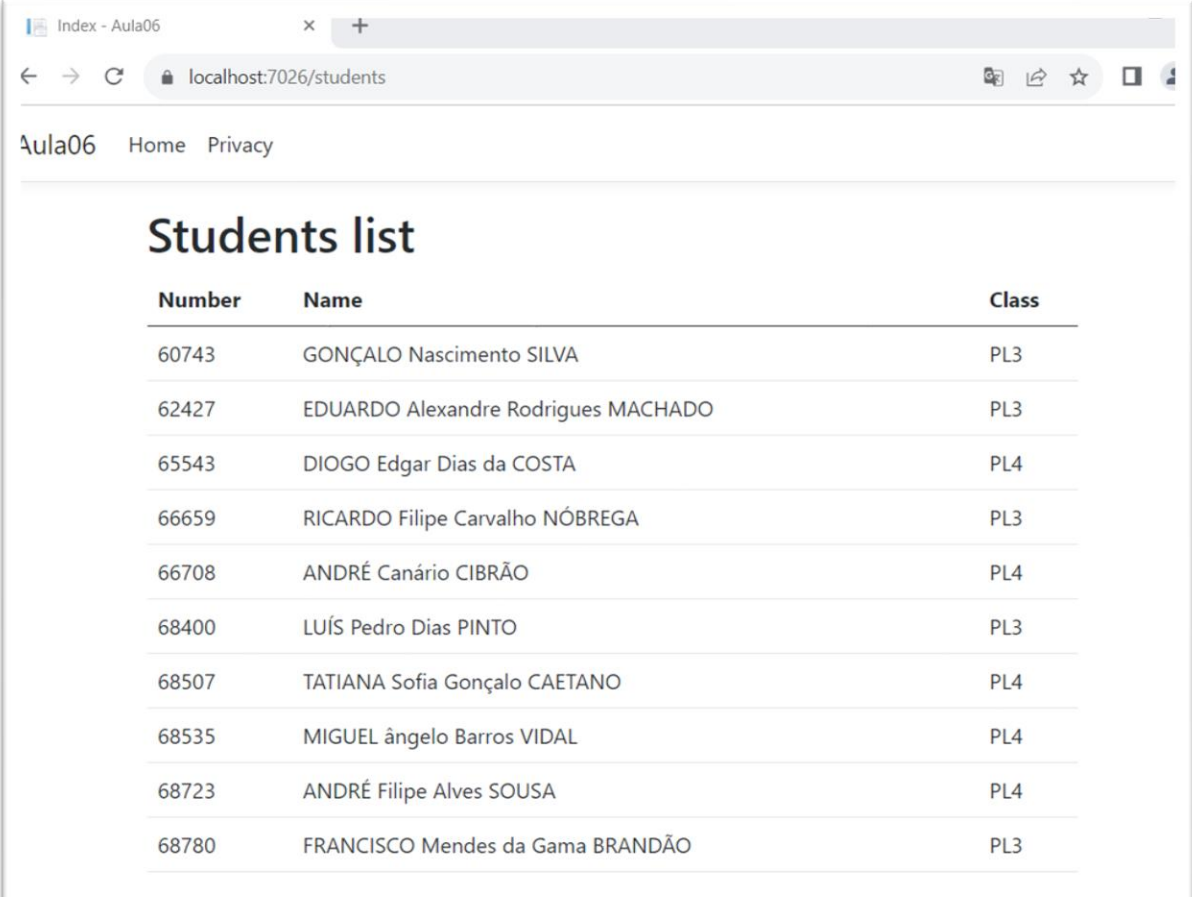
Add   Cancel

- Adapt the code to show the student number (primary key of the tables are usually omitted from the listings) and to remove all CRUD action links.

```
@model IEnumerable<Aula06.Models.Student>

<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Number)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Name)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Class)
            </th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Number)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Name)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Class.Name)
                </td>
            </tr>
        }
    </tbody>
</table>
```

Try the application on the implemented feature. The result obtained must match the
following image:

- Create a new resource (Index2) that allows you to list students using a filter based on the 1st letter of the student's name.

```
public async Task<IActionResult> Index2(string letter)
{
    if (!string.IsNullOrEmpty(letter))
    {
        return View(await _context.Students.Where(x => x.Name.StartsWith(letter)).Include(c => c.Class).ToListAsync());
    }
    else
        return View(await _context.Students.Include(c => c.Class).ToListAsync());
}
```

- Add an Index2 file view equal with same code as the Index view.

Try the resource by passing the letter information in a query string (for example **https://localhost:?????/students/index2?letter=A**). The query string is necessary to send information to the action (string letter) that is not provided in the URL.

We can configure the application to accept the "letter" in the URL itself (instead of using the query string). For this we add endpoints to the application.

- Add the following endpoint to the **Program.cs** file.

```
app.MapControllerRoute(
    name: "filtered",
    pattern: "Filter/{letter?}",
    defaults: new {Controller="Students", Action="Index2"},
    constraints: new {letter=@"[A-ZÂÁÉÓ]"}); //regular expression to validate "letter" field

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```
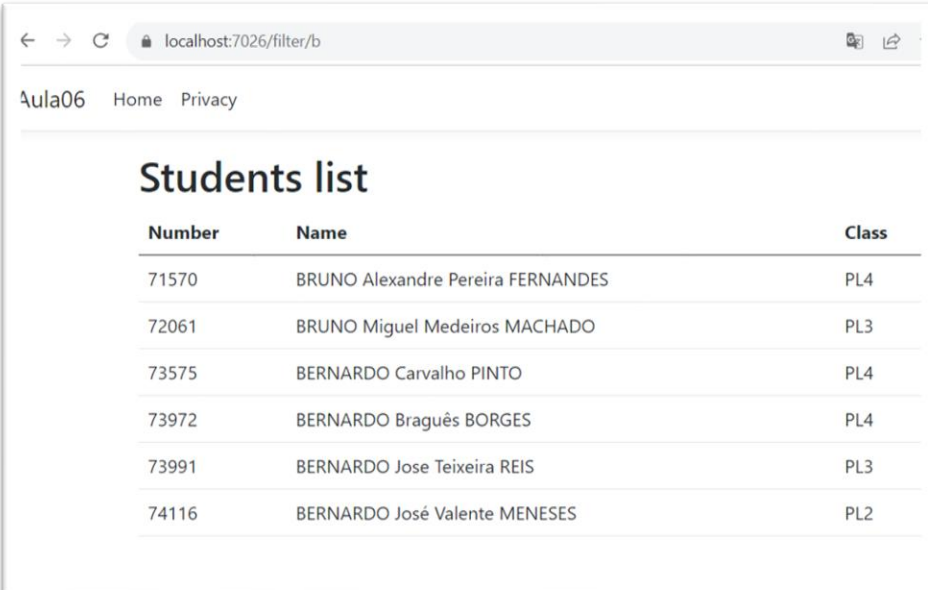
Base endpoint, which allows the name of a controller to be accepted in the URI (default is **Home**) followed by the name of an action (default is **Index**) and can also be followed by an **id** value (used, for example, in Edit, Details and Delete operations)

The new endpoint contains a pattern indicating how the URL can be written. In this case, we indicate that the base of the URL is always the word **Filter** followed optionally by a field (identifier between {}) named **letter** (? means optional). If the URL is validated at this endpoint, it will forward to the execution of the **Index2** action of the **Students** controller.

URL fields can have constraints defined. In this case, there are restrictions on the value of the **letter** parameter, through a regular expression that restricts the valid characters to be recognized (1 character from the following set [A-ZÂÁÉÓ]).

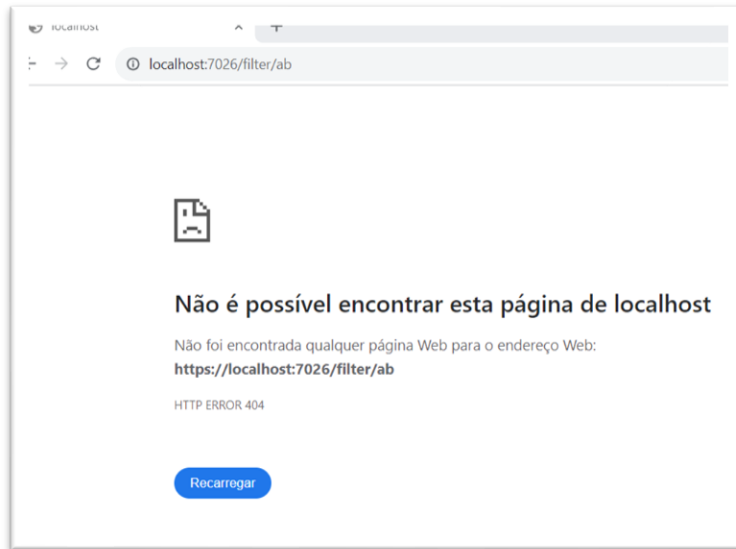Try de endpoint by accessing **https://localhost:?????/filter/B**.

If a resource that does not respect the restrictions is requested, another endpoint will be used to validate the requested URL and... it may result in an error (in the example

shown below, the request is forwarded to the **filter** controller and to the **ab** action which do not exist).

For more information on routing:

https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/routing?view=aspnetcore-8.0

To complement the exercise, three different and alternative ways of applying the restrictions to the values of the URI fields are presented (these variants must not be defined simultaneously with the previous one because they overlap in the pattern):

The first example uses "Inline Constraints" to restrict the letter to one alphabetic character.

```
/"Inline Constraint" constraint for "Checking Data Types"
/ See more at https://www.tektutorialshub.com/asp-net-core/asp-net-core-route-constraints/
.pp.MapControllerRoute(
    name: "filtered2",
    pattern: "Filter/{letter:alpha:length(1)?}",
    defaults: new { Controller = "Students", Action = "Index2" });
```

The second example uses an external custom class to implement the field constraints. The restrictions are defined in a class named **MyRulesConstraint** which implements **IRouteConstraint** interface.

```
app.MapControllerRoute(
    name: "filtered3",
    pattern: "Filter/{letter?}",
    defaults: new { Controller = "Students", Action = "Index2" },
    constraints: new { letter = new MyRulesConstraint() }); // custom class to validate "letter" field
```

Add the new class file to the project root.

```
       @ _ViewImports.cshtml
       @ _ViewStart.cshtml
    ▷  {} appsettings.json
       🗊 Aula06ScriptDB.sql
    ▷  C# MyRulesConstraint.cs
    ▷  C# Program.cs
```
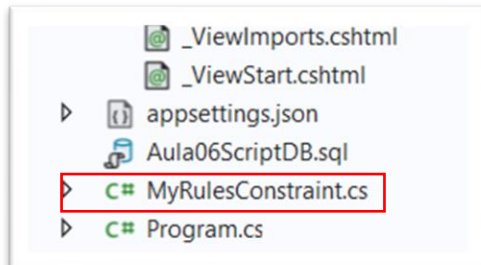
Use the following code in the class. The implementation of **IRouteConstraint** only needs
to code the **Match** method. This method should return a boolean value representing the
validation of the field "letter". In this case the validation is gathered by using a regular
expression. Other logic can be used to get the correct result.

```
public class MyRulesConstraint : IRouteConstraint
{
    public bool Match(HttpContext? httpContext, IRouter? route, string routeKey, RouteValueDictionary values, RouteDirection routeDirection)
    {
        if(routeKey!="letter")
            return false; // reject any other than "letter"

        // regular expression to validade one char of the presented set
        return values[routeKey] is null || Regex.IsMatch((values[routeKey] as string), @"^[a-zA-ZÂÁÉÓ]$");
    }
}
```

As the <u>third example</u>, we also can use this same class as an "Inline Constraint".

```
app.MapControllerRoute(
    name: "filtered4",
    pattern: "Filter/{letter:myrule?}", // fourth versio using the same class in inline format
    defaults: new { Controller = "Students", Action = "Index2" });
```

For that, we first need to register the class as a **RouteOption** in **Program.cs** file. The key
used to register the class will be the name used as inline constraint (**myRule**).

```
builder.Services.AddTransient<DbInitializer>();

builder.Services.AddRouting(options=>
options.ConstraintMap.Add("myRule",typeof(MyRulesConstraint)));

var app = builder.Build();
```

Third step –Decorate the view with a Navigation menu

Adapt the Students list view to offer a navigation menu between the various letter options. The menu will have all alphabet letters so the user can choose one of them by clicking on it.

It's necessary to use the **ViewBag** structure to send the chosen **letter** information to the view to help format the menu.

```
public async Task<IActionResult> Index2(string letter)
{
    ViewBag.letter = letter;

    if (!string.IsNullOrEmpty(letter))
    {
        return View(await _context.Students.Where(x => x.Name.StartsWith(letter)).Include(c => c.Class).ToListAsync());
    }
    else
        return View(await _context.Students.Include(c => c.Class).ToListAsync());
}
```

In the **Index2** view file, build the list of letters of the alphabet as navigation options with the next code. In this example, the "filtered" endpoint is used to build the links.

```
@model IEnumerable<Aula06.Models.Student>

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

<h1>Students list</h1>
@if (String.IsNullOrEmpty(ViewBag.letter) == false)
{
    <p>Names started by '@ViewBag.letter'</p>
}
else
{
    <p>All names</p>
}
@if (String.IsNullOrEmpty(ViewBag.letter))
{
    <span style="color:red; font-weight: bold;">All  | </span>
}
else
{
    <span><a asp-route="filtered" asp-route-letter="">All</a> | </span>
}

@for(char c= 'A'; c<='Z'; c++)
{
    if(ViewBag.letter == "" + c)
    {
        <span style="color:red; font-weight:bold;">@c   | </span>
    }
    else
    {
        <span><a asp-route="filtered" asp-route-letter="@c">@c</a> | </span>
    }
}
<partial name="Listing" model="Model" />
```

Fourth step – Homework

Implement a new resource in Students controller, with name **Index3** to present the students list sorted in ascending or descending order.

The ordering can be based on two criteria: <u>student number</u> and/or <u>student name</u>.

Use a new endpoint to mask orders as follows:

https://localhost:?????/order/byNumber/Descending

https://localhost:?????/order/byNumber/Ascending

https://localhost:?????/order/byName/Descending

https://localhost:?????/order/byName/Ascending

The application should look like the next example: