

Tutorial for Class number 4

This exercise intends to implement an application to manage the information of an entity model in a database system. The application will allow to perform CRUD operations without having to write SQL. The database system operations will be supported by **Entity Framework Core** and generated by the process **Code First**.

This tutorial follows the “ASP.NET Core MVC with EF Core - tutorial series”

<https://learn.microsoft.com/en-us/aspnet/core/data/ef-mvc/?view=aspnetcore-7.0>

First step

- add a class named **Category** to **Models** folder in project with the next content.

```
namespace Aula04.Models
{
    public class Category
    {
        [Key]
        public int Id { get; set; }

        [Required(ErrorMessage = "Required field")]
        [StringLength(50, MinimumLength = 3, ErrorMessage = "{0} length must be between {2} and {1}")]
        public string? Name { get; set; }

        [Required(ErrorMessage = "Required field")]
        [MaxLength(256, ErrorMessage = "{0} length can not exceed {1} characters")]
        public string? Description { get; set; }

        public Boolean State { get; set; } = true; //default value
    }
}
```

This class intends to represent a Category entity in a Database system. For this reason, a specific field is added to the class in order to guarantee the data identity.

Normally, this field can be defined in three alternative ways:

- with the **Id** identifier;
- with the identifier equal to the **class name and ending with Id** (for example: **CategoryId**);
- with any identifier and with the data annotation **Key**;

In class Category we use first and third option... it was not necessary. The third option could not be used.

All of these options represent fields of information that in the database defines primary keys. If the field is integer type, then it will have self-generated values in the database - Identity (1,1). If this functionality is not desired, use the following data annotation:

```
[DatabaseGenerated(DatabaseGeneratedOption.None)]
```

It's not the case in this exercise.

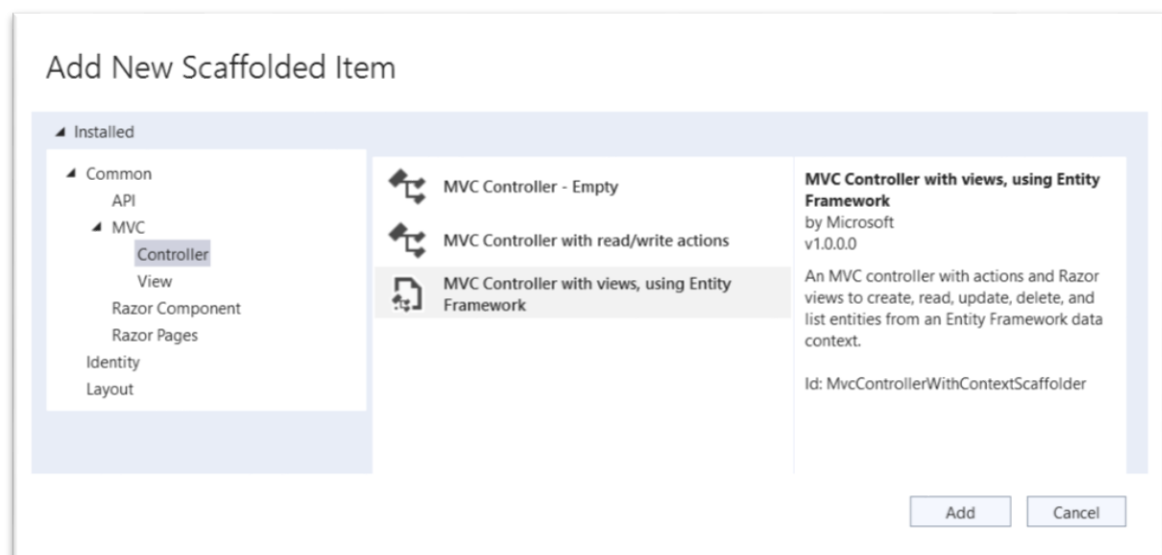
Now we need to add some important information that will be needed to manage information in a database system:

- Data Context : contains all data sets of information, based on model classes, to be used in database system;
- Connection String : contains the information that the provider need to know to be able to establish a connection to the database;
- Registering the connection string in application configuration : so the application can use the information in code;

In this process, all these three elements will be generated by the scaffold operations when execute the next operation (Second Step).

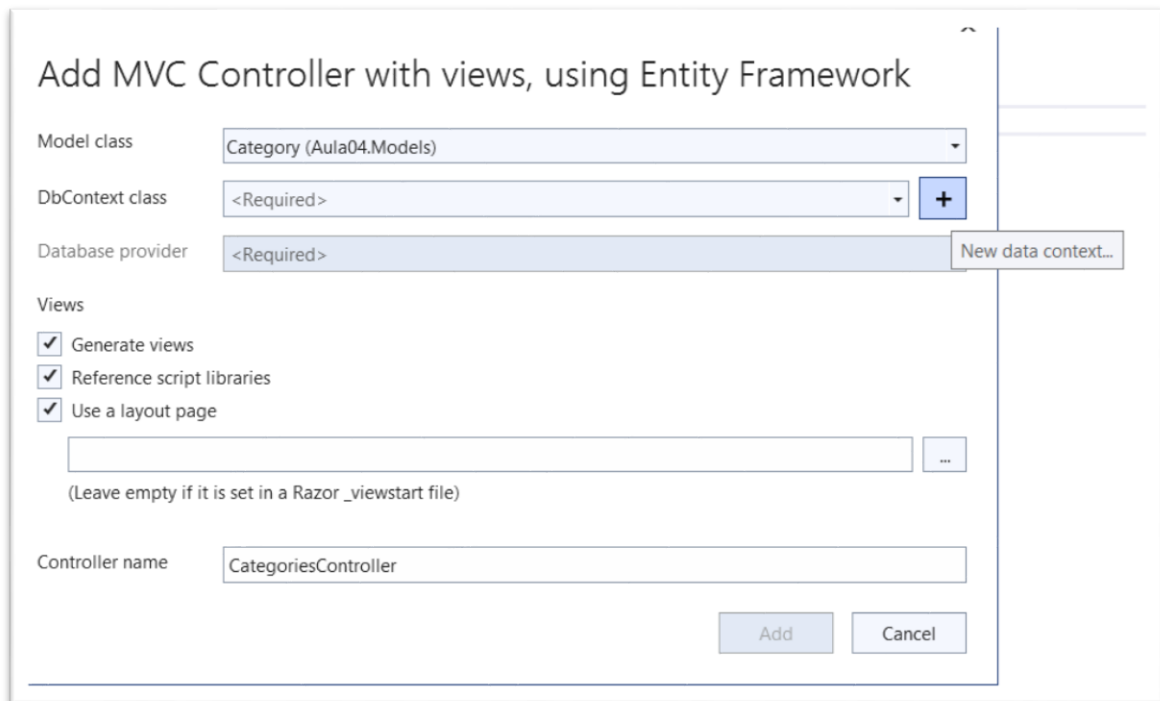
Second Step

- Add a controller with the template “MVC Controller with views, using Entity Framework”. This will add to project one controller class and all views code for CRUD operations over the model data.



- Choose the Category model class choose an existing Data Context class or create a new one.

As we still don't have any Data Context in project, we need to create a new one. The project can have multiple Data Context classes but we also can add a new model set to an existing data context (when the new data set is intended for the same database).



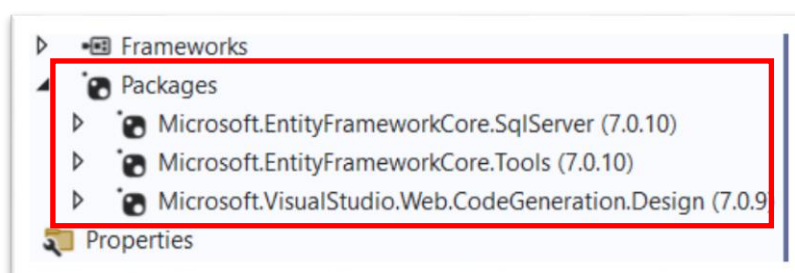
- In button '+' add a data context name as suggested.



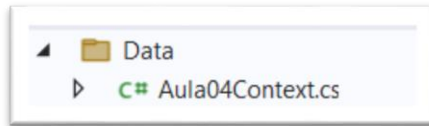
- Accept or modify the suggested name for the controller (**CategoriesController**).

After this operation with success you must confirm the existence these elements in project:

- ✓ The packages for Entity Framework installed:



- ✓ The Data Context Class



In this class we must have a DbSet for each entity model class (in this case we have one). In our example we have the Category DbSet.

```
public class Aula04Context : DbContext
{
    public Aula04Context (DbContextOptions<Aula04Context> options)
        : base(options)
    {
    }

    public DbSet<Aula04.Models.Category> Category { get; set; } = default!;
}
```

- ✓ The Categories controller Class with all CRUD operations defined.

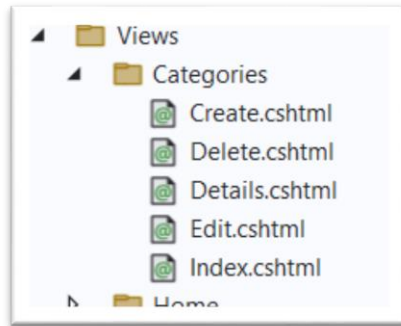


Here we can analyse all generated code and see how database operations are executed, as for example, in **Delete** action.

```
// POST: Categories/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    if (_context.Category == null)
    {
        return Problem("Entity set 'Aula04Context.Category' is null.");
    }
    var category = await _context.Category.FindAsync(id);
    if (category != null)
    {
        _context.Category.Remove(category);
    }

    await _context.SaveChangesAsync();
    return RedirectToAction(nameof(Index));
}
```

- ✓ And all View files of the Categories controller actions



- ✓ Beside this, we also have the connection string defined in file **appsettings.json**.

```
"ConnectionStrings": {  
  "Aula04Context": "Server=(localdb)\\mssqllocaldb;Database=Aula04Context-ae7b8666-1178-471..."  
}
```

In this configuration, the "Class04Context" key contains a value that consists of the location of the SQL server **(localdb)\\mssqllocaldb** and the name of the **Database**, among others. The database name can be modified before the creation of the database (we do not do it in this exercise).

NOTE: the SQL server location may depend on your PC settings. You can also modify this value.

- ✓ At last, we have the file **Program.cs** where it is register Class04Context as a service in **CreateBuilder** process.

```
var builder = WebApplication.CreateBuilder(args);  
builder.Services.AddDbContext<Aula04Context>(options =>  
    options.UseSqlServer(builder.Configuration.GetConnectionString("Aula04Context")) ?? throw new I...
```

Services (such as the EF database context) are registered with dependency injection during application startup. Components that require these services (such as MVC controllers) are provided these services via constructor parameters. You can see the controller **Categories** constructor code that receives a context instance.

```
private readonly Aula04Context _context;  
  
public CategoriesController(Aula04Context context)  
{  
    _context = context;  
}
```

Now we are ready to prepare the database with some commands executed in “Package Manager Console”.

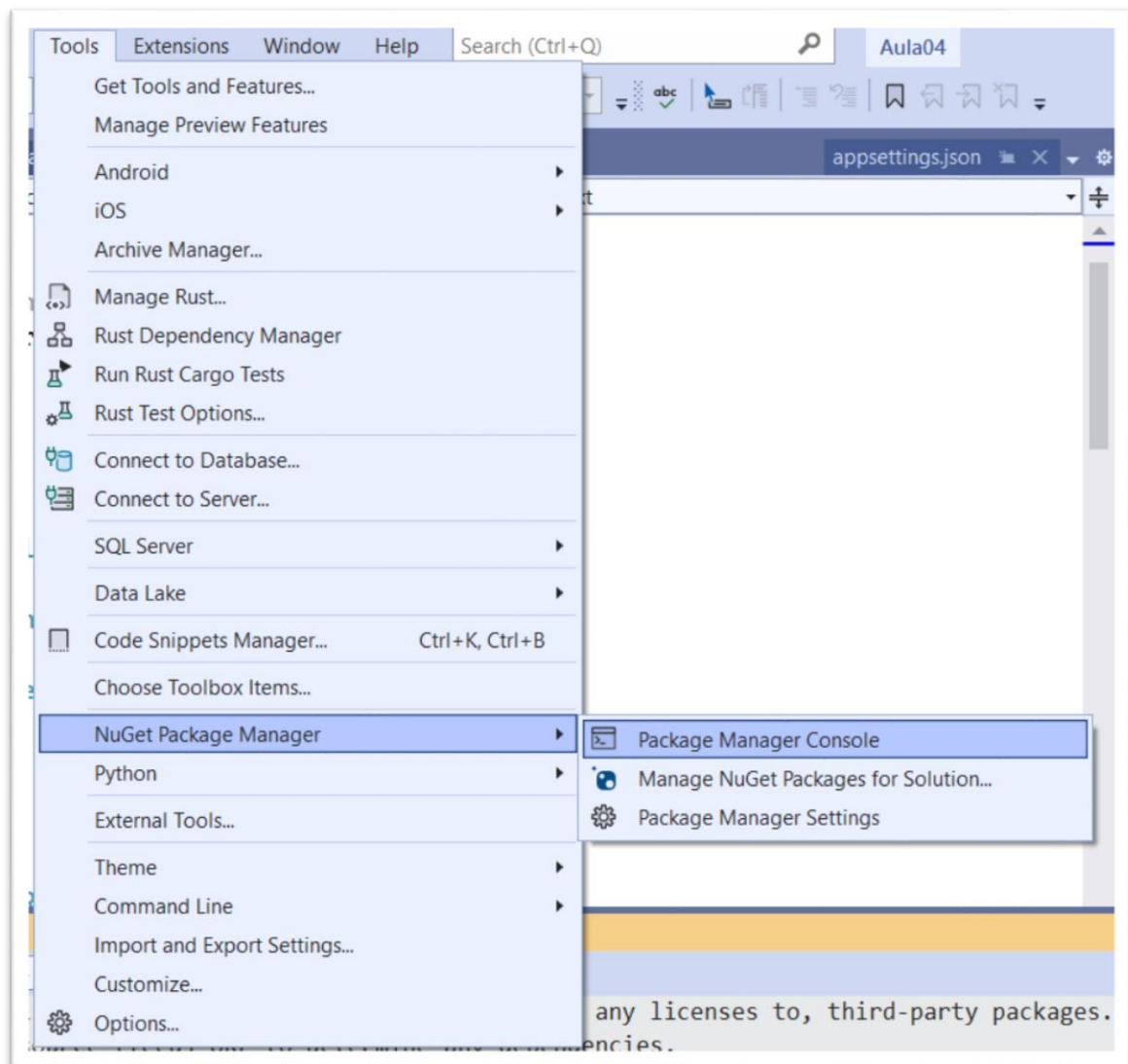


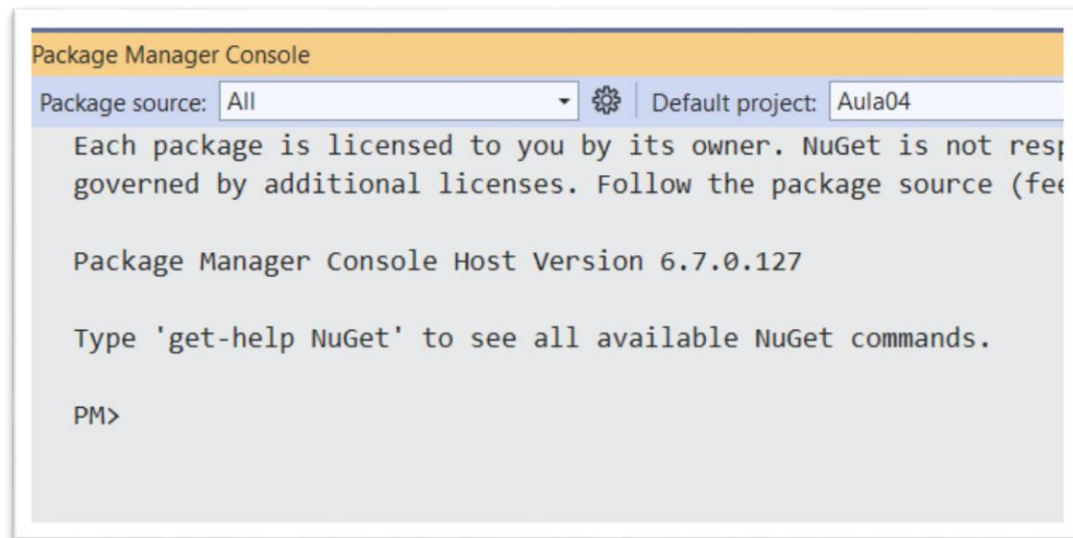
For Migrations definition see

<https://learn.microsoft.com/en-us/aspnet/core/data/ef-mvc/migrations?view=aspnetcore-7.0>

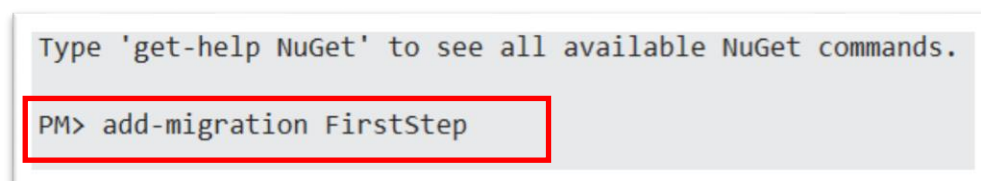
Third step

Open “**Package Manager Console**”

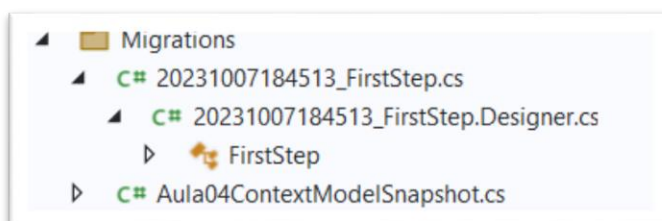




We start by creating a migration class by executing the command **add-migration <name>** using **FirstStep** as the migration name.



In Solution Explorer we found a new folder (Migrations) where two files are important to see. The **Migration** file and the **Snapshot** file. In the first we can see some code to create the entity table in database (and also to destroy it). The second one is a file where each migration creates a snapshot of the current database schema. When you add a migration, EF determines what changed by comparing the data model to the snapshot file. The differences found determines the migration content and the snapshot is updated with the modifications caused by migration.



Part of the content of the **FirstStep** class file is the next one.


```
public partial class FirstStep : Migration
{
    /// <inheritdoc />
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "Category",
            columns: table => new
            {
                Id = table.Column<int>(type: "int", nullable: false)
                    .Annotation("SqlServer:Identity", "1, 1"),
                Name = table.Column<string>(type: "nvarchar(50)", maxLength: 50, nullable: false),
                Description = table.Column<string>(type: "nvarchar(256)", maxLength: 256, nullable: false),
                State = table.Column<bool>(type: "bit", nullable: false)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_Category", x => x.Id);
            });
    }
}
```

Part of the content of the **Class04ContextModelSnapshot** class file is the next one.

```
[DbContext(typeof(Aula04Context))]
partial class Aula04ContextModelSnapshot : ModelSnapshot
{
    protected override void BuildModel(ModelBuilder modelBuilder)
    {
        #pragma warning disable 612, 618
        modelBuilder
            .HasAnnotation("ProductVersion", "7.0.10")
            .HasAnnotation("Relational:MaxIdentifierLength", 128);

        SqlServerModelBuilderExtensions.UseIdentityColumns(modelBuilder);

        modelBuilder.Entity("Aula04.Models.Category", b =>
        {
            b.Property<int>("Id")
                .ValueGeneratedOnAdd()
                .HasColumnType("int");

            SqlServerPropertyBuilderExtensions.UseIdentityColumn(b.Property<int>("Id"));

            b.Property<string>("Description")
                .IsRequired()
                .HasMaxLength(256)
                .HasColumnType("nvarchar(256)");

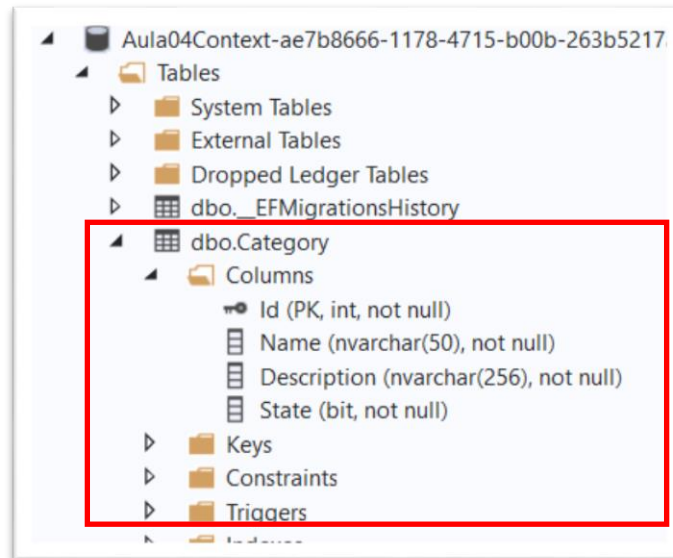
            b.Property<string>("Name")
                .IsRequired()
                .HasMaxLength(50)
                .HasColumnType("nvarchar(50)");
        });
    }
}
```

Next we will create the database by executing the command **update-database**.

```
To undo this action, use Remove-Migration.
PM> update-database
```

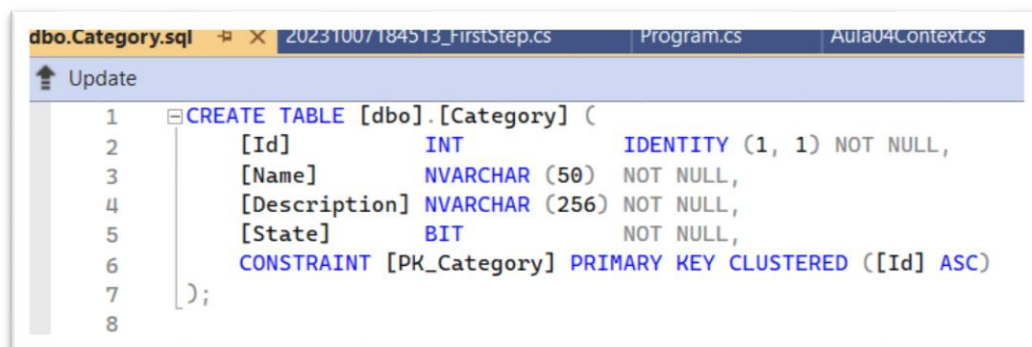

The result of this operation is the update of database according to migration contents. In this particular case it was created the database and with a Categories table.

See in **SQL Server Object Explorer** the table and database created.



We can verify the database created and also the Category table.

Using the mouse right button over the **dbo.Category** we can use the View Code option to see the SQL code that generated the table. Here we can confront the SQL restrictions (NOT NULL, NVARCHAR(?)) with some of the data annotations used in **Category** class model (Required, StringLength).



For running the application for the first time it is important to have some data in database.

Fourth Step

Initialize DB with test data. The Entity Framework will create an empty database for the application. To resolve this, we can write a method that is called after the database is created in order to populate it with test data.

Here you'll use the **EnsureCreated** method to automatically create the database.

- In the **Data** folder, create a new class file named **DbInitializer.cs** and replace the template code with the following code, which causes a database to be created when needed and loads test data into the new database.

```
public class DbInitializer
{
    private Aula04Context _context;

    public DbInitializer(Aula04Context context)
    {
        _context = context;
    }

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

        if (!_context.Category.Any())
        {
            return;
        }

        var categorias = new Category[] {
            new Category { Name = "Programming", Description = "Algoritms and programming area courses" },
            new Category { Name = "Administration", Description = "Public administration and business management courses" },
            new Category { Name = "Communication", Description = "Business and institutional communication course" }
        };

        //_context.Category.AddRange(categorias);
        foreach (var c in categorias)
        {
            _context.Category.Add(c);
        }
        _context.SaveChanges();
    }
}
```

The code checks if there are any category data in the database, and if not, it assumes the database is new and needs to be seeded with test data.

Now we should call this code in application launch inside **Program.cs** file.

- Modify the code in Main method to be like the next one.

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

// Add service to the new class DbInitializer
builder.Services.AddTransient<DbInitializer>();

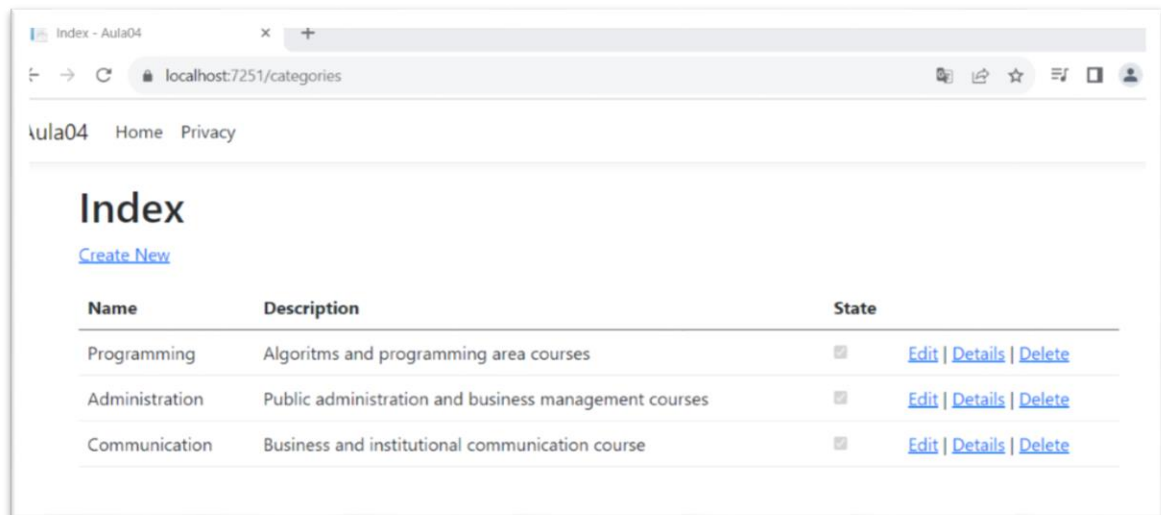
var app = builder.Build();

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

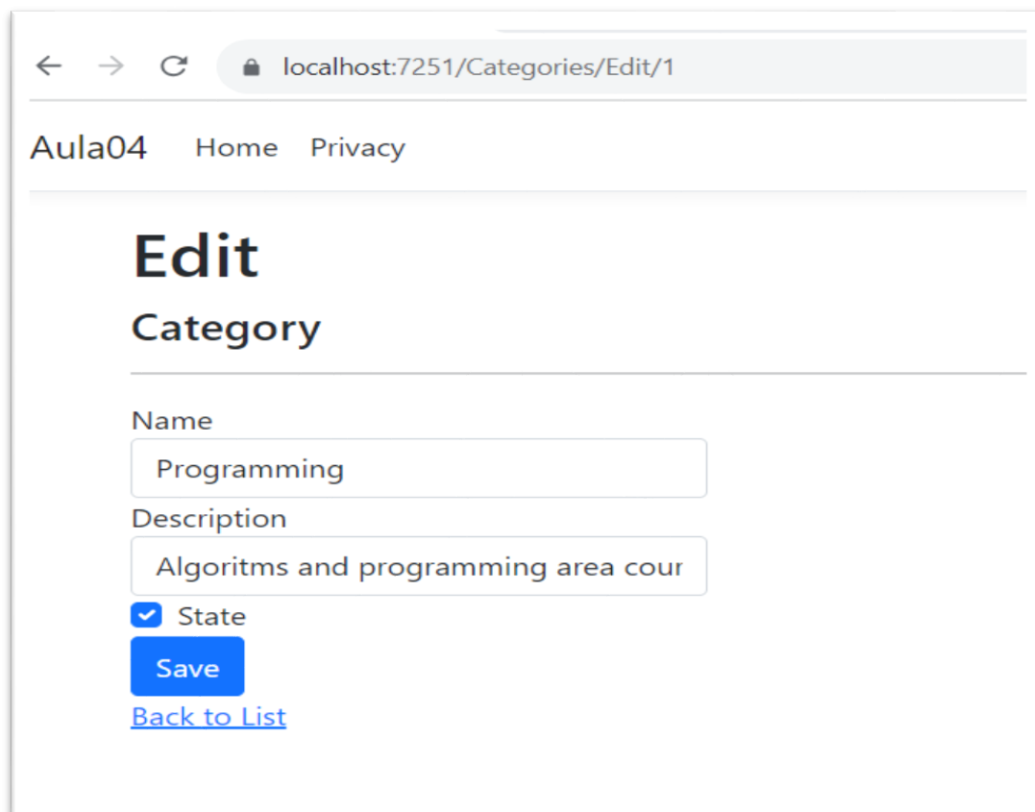
var initializer = services.GetRequiredService<DbInitializer>();
//execute the method Run from the class DbInitializer
initializer.Run();
```

And we are ready to test the application 🚧💣

- run on **https://localhost:????/categories/index** to see all categories listed.



- or editing one of the elements listed in **https://localhost:????/Categories/Edit/1** (by clicking Edit link)



You should test all CRUD operations over category elements. 😊

Homework:

- Add a new field named **Date**, of type **DateTime**, to Category class model and define his default value as **DateTime.Now**.

```
[DisplayName("Creation Date")]  
public DateTime Date { get; set; } = DateTime.Now;
```

This default value will be available only in the web application code. Whenever an instance of the Category class is created, if the **Date** value is not specified it will be used the **DateTime.Now** value for it. This rule will not be transferred to database.

- Add a new Migration to the project by executing the command **add-migration SecondStep** in the “Package Manager Console”;
- Update the database structure by executing the command **update-database** in the “Package Manager Console”;
- Adjust the data initialization in **DbInitializer.cs** class by adding the new field value (after that you should clean all data in database table for the initialization to work);
- Add the new field to the information presented in the following views:
 - Index (listing)
 - Details
 - Delete
- Add the new field to the **Edit** operation. For this, adjust the formulary in the view file and the **Bind** operation in the corresponding action for POST method.



For more information about Model Binding please consult

<https://docs.microsoft.com/en-us/aspnet/core/mvc/models/model-binding?view=aspnetcore-7.0>