

ASP.NET Core Databases

Working with Entity Framework Core

Core

SoftUni Team
Technical Trainers



SoftUni



Software University

<https://softuni.bg>

Table of Content

1. Entity Framework Core
 - Code First Approach
2. EF Core Components
3. EF Core Configuration
 - Fluent API
4. CRUD Operations
5. Database Migrations

sli.do

#csharp-web



Code First Approach

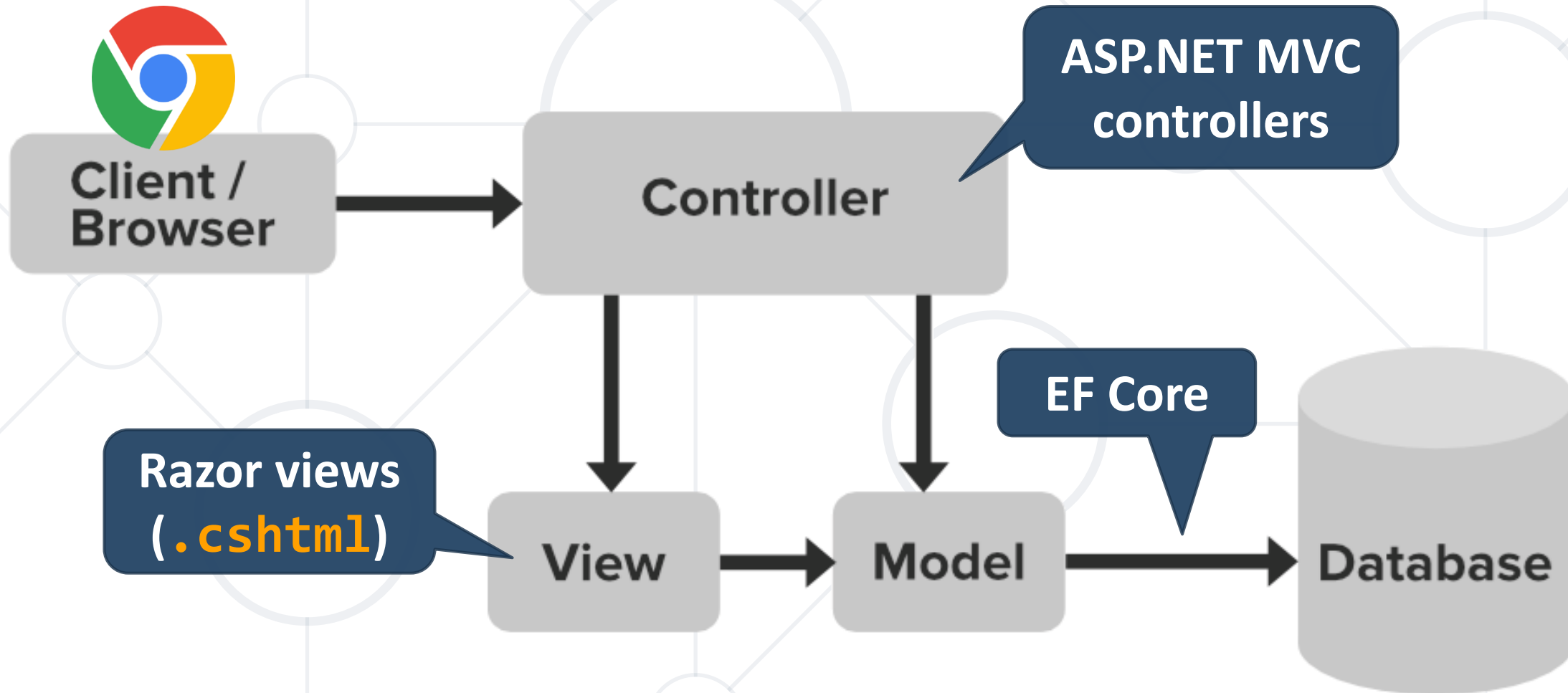
Entity Framework Core

Entity Framework Core: Overview

- The standard **ORM framework** for **.NET** and **.NET Core**
- Provides LINQ-based data queries and **CRUD** operations
- Automatic **change tracking** of in-memory objects
- Works with many relational databases (with different providers)
- Open source with independent release cycle



ASP.NET Core MVC + Entity Framework

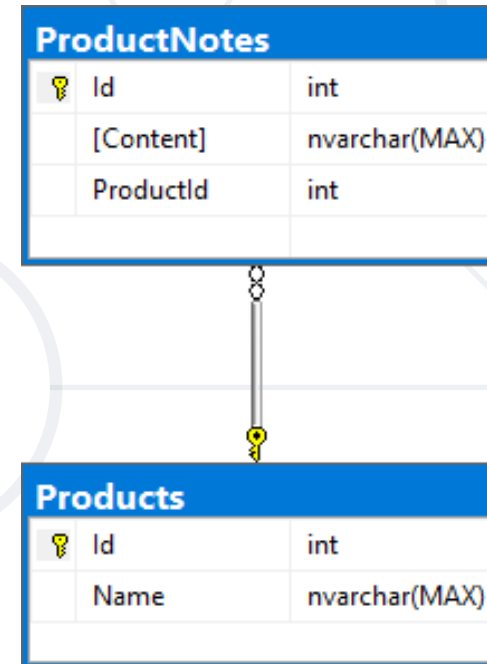


What is the Code First Approach?

- **Code First** means to write the .NET classes and let EF Core create the **database** from the **mappings**

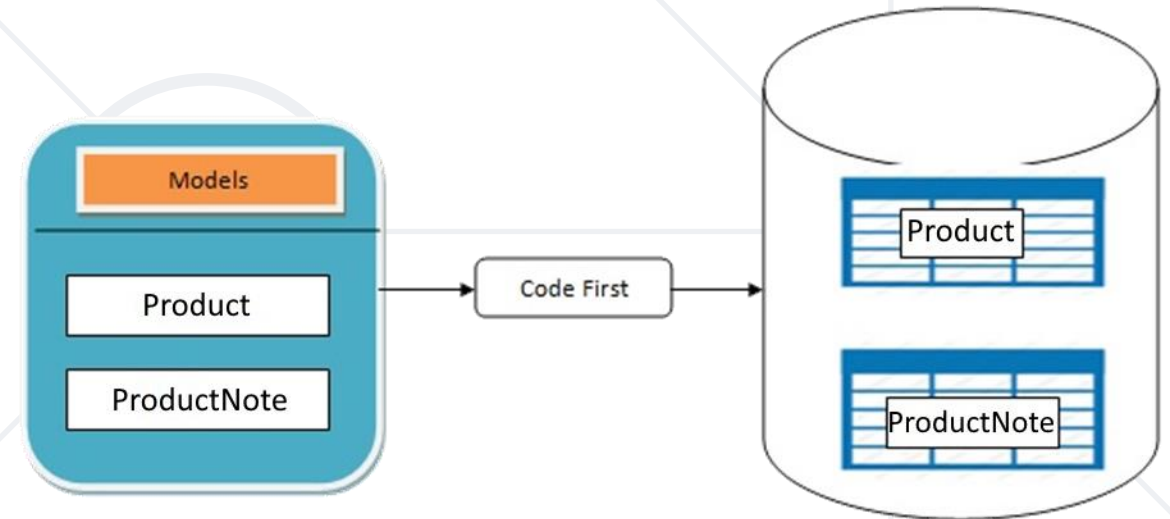


```
Models
├── C# Product.cs
│   ├── Product
│   │   ├── Id : int
│   │   └── Name : string
│   └── C# ProductNote.cs
│       ├── ProductNote
│       │   ├── Id : int
│       │   ├── Content : string
│       │   ├── ProductId : int
│       │   └── Product : Product
```




Why Use Code First?


- Write code **without** having to define **mappings** in XML or **create** database **tables**
- Define objects in **C# format**
- Enables database persistence with no configuration
- Changes to code can be **reflected** (migrated) in the schema
- **Data Annotations** or **Fluent API** describe properties
 - **Key, Required, MinLength**, etc.



Code First Basic Workflow (1)

1. Define the data model (**Code First** or **Scaffold from DB**)

Products		
	Id	int
	Name	nvarchar(MAX)

ProductNotes		
	Id	int
	[Content]	nvarchar(MAX)
	ProductId	int

2. Write & execute query over **IQueryable**

```
var products = this.data
    .Products
    .Select(p => new ProductViewModel()
    {
        Id = p.Id,
        Name = p.Name,
    })
    .ToList();
return View(products);
```

3. EF generates & executes an **SQL query** in the **DB**

```
SELECT [p].[Id], [p].[Name]
FROM [Products] as [p]
```

Code First Basic Workflow (2)

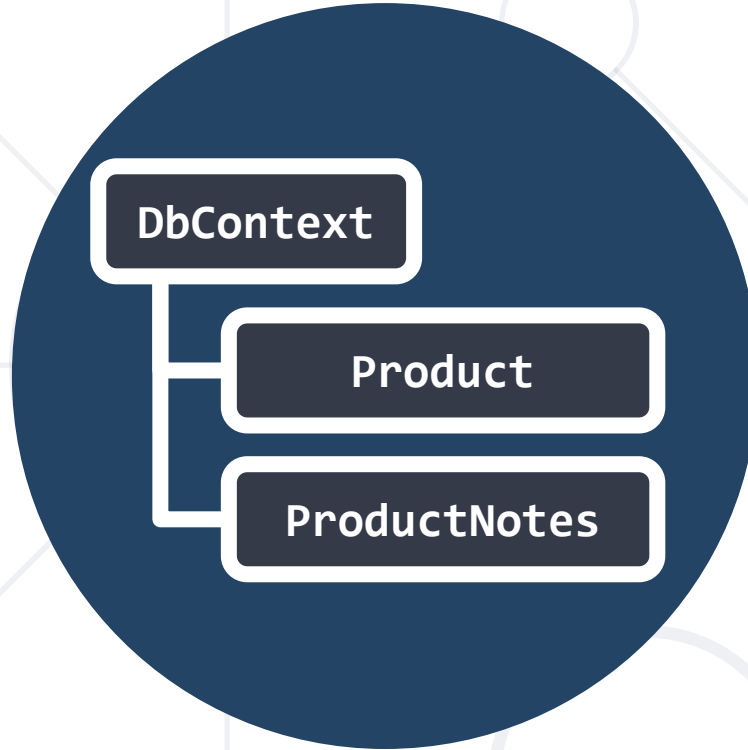
4. EF transforms the query results into .NET objects

5. Modify data with C# code and call "**SaveChanges()**"

```
var product = _data.Products.Find(id);  
product.Name = model.Name;  
  
_data.SaveChanges();
```

6. Entity Framework generates & executes SQL command to modify the DB

```
exec sp_executesql N'SET NOCOUNT ON;  
UPDATE [Products] SET [Content] = @p0  
WHERE [Id] = @p1;  
SELECT @@ROWCOUNT;  
' , N'@p1 int,@p0 nvarchar(4000)',  
@p1=1, @p0=N'Post 1 Changed'
```



EF Core Components

Overview of System Objects

Domain Classes (Models) (1)

- Bunch of normal C# classes (POCO)
 - May contain **navigation properties** for **table relationships**

```
public class ProductNote
{
    public int Id { get; set; }
    public string Content { get; set; }
    public int ProductId { get; set; }
    public Product Product { get; set; }
}
```

Primary key

Foreign key

Navigation property

- Recommended to be in a **separate class library**

Domain Classes (Models) (2)

- Another example of a domain class (model)

```
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public IList<ProductNote> ProductNotes { get; set; }
    = new List<ProductNote>();
}
```

One-to-many
relationship

- Usually named after the database, e.g., **ShoppingListDbContext**

```
public class ShoppingListDbContext : DbContext
```

Inherits the
DbContext class

- Manages model classes using **DbSet<T>** type
- Easily navigate through **table relations**
- Managing database **creation/deletion/migration**
- Executing **LINQ queries** as native **SQL queries**
- **DbContext** properties
 - **Database** – **EnsureCreated/Deleted** methods, DB Connection
 - **ChangeTracker** – holds info about the **automatic change tracker**

Defining DbContext Class

```
using Microsoft.EntityFrameworkCore;
```

EF Reference

```
public class ShoppingListDbContext : DbContext
```

Accepts **options** through the constructor

```
{  
    public ShoppingListDbContext  
        (DbContextOptions<ShoppingListDbContext> options)  
        : base(options)  
        => this.Database.EnsureCreated();
```

Collections of entities

```
    public DbSet<Product> Products { get; set; }  
    public DbSet<ProductNote> ProductNotes { get; set; }
```

```
    protected override void OnModelCreating(ModelBuilder builder)
```

```
    {  
        builder.Entity<Product>()  
            .HasMany(p => p.)  
            .WithOne(r => r.ProductNotes);
```

Use the Fluent API to describe our **table relations** to EF Core

```
    }  
}
```



EF Core Configuration

NuGet Packages, Configuration

- To add EF Core support to a project in Visual Studio

- Install it from **Package Manager Console**

```
Install-Package Microsoft.EntityFrameworkCore
```

- Or using **.NET Core CLI**

```
dotnet add package Microsoft.EntityFrameworkCore
```

- EF Core is modular – any **data providers** must be installed too

```
Microsoft.EntityFrameworkCore.SqlServer
```

- To use the Entity Framework Core **CLI tools**, install also

```
Microsoft.EntityFrameworkCore.Design
```

How to Connect to SQL Server? (1)

- In ASP.NET Core **connection string** is in the **appsettings.json** file and has the following **properties**

```
"ConnectionStrings": {  
  "DefaultConnection": "Server=(localdb)\\mssqllocaldb;  
Database=ShoppingList;Trusted_Connection=True;  
MultipleActiveResultSets=true"}
```

How to Connect to SQL Server? (2)

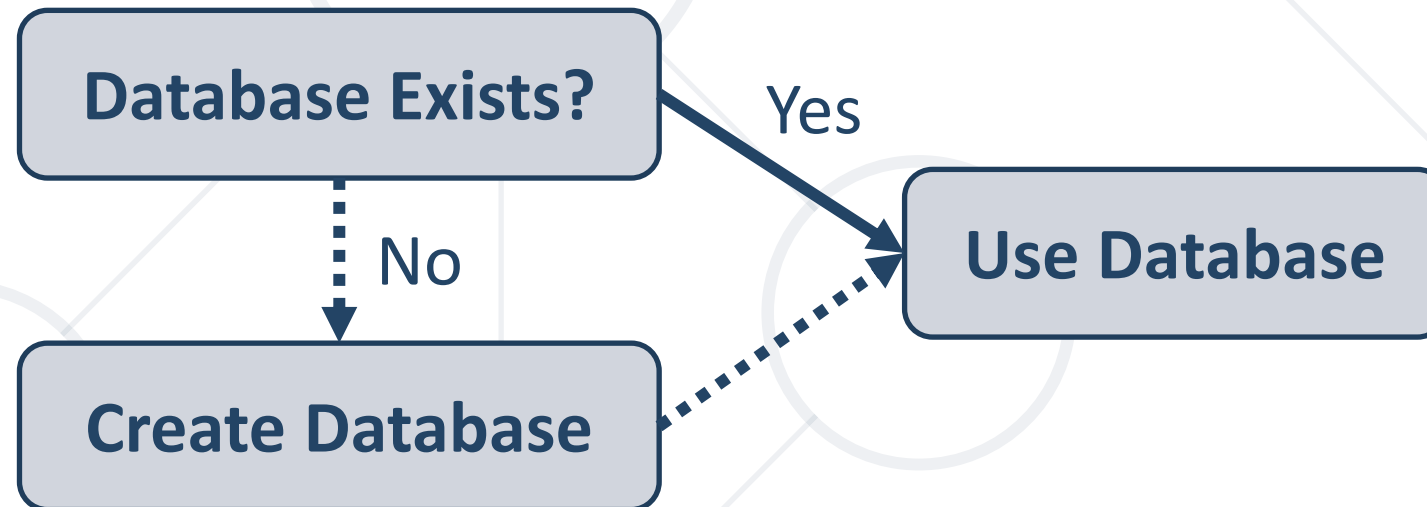
- Use the **DbContext** and tell it to use SQL with the **connection string** in the **Program** class

```
var connectionString = builder
    .Configuration
    .GetConnectionString("DefaultConnection");

builder
    .Services
    .AddDbContext<ShoppingListDbContext>(
        x => x.UseSqlServer(connectionString));
```

Database.EnsureCreated()

- When you create the DB context, you can call **Database.EnsureCreated()**
- This will **create the DB + schema**, when the DB is missing



- **EnsureCreated()** does not use migrations → you should drop the entire DB when you change the DB schema

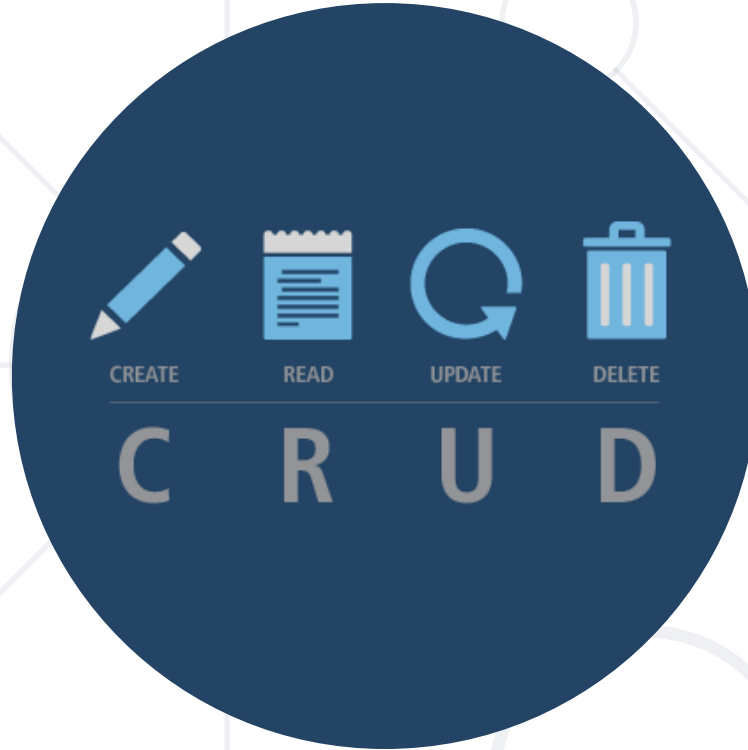
Database.EnsureCreated() – Example

```
public class ShoppingListDbContext : DbContext
{
    public ShoppingListDbContext(
        DbContextOptions<ShoppingListDbContext> options)
        : base(options)
        => Database.EnsureCreated();
    // This will create the DB schema if the DB does not exist

    // Any change in the data entities will not change the DB

    // You should update the DB by hand
    // or drop and re-create the DB after each entity change!

    ...
}
```

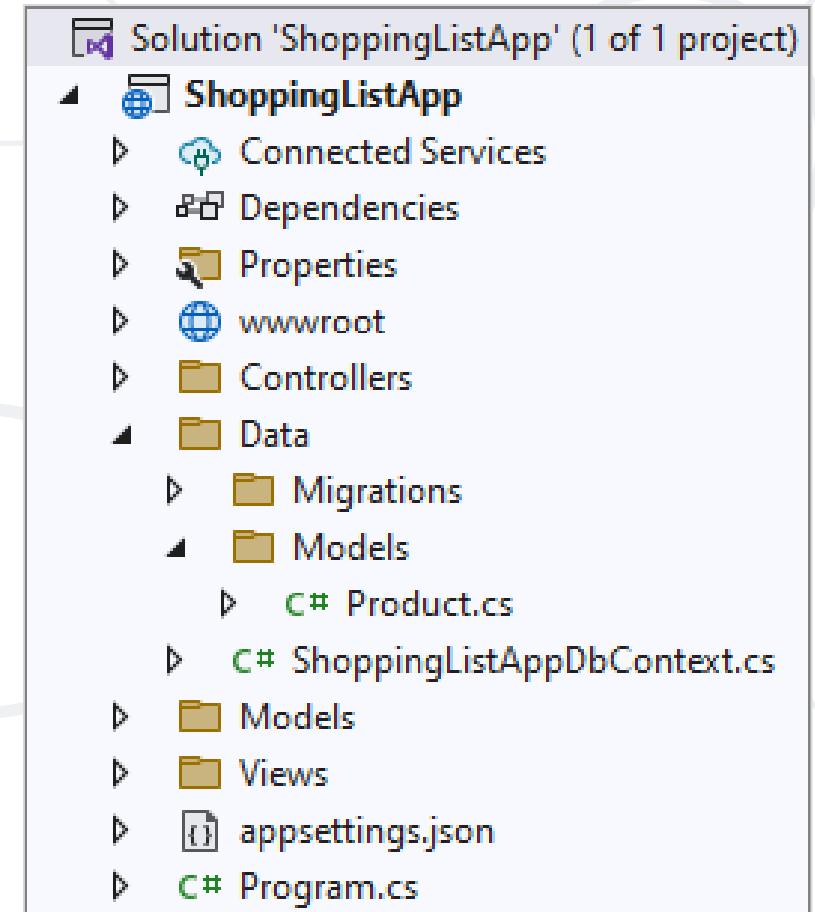


CRUD in ASP.NET Core MVC with EF

The "ShoppingList" App

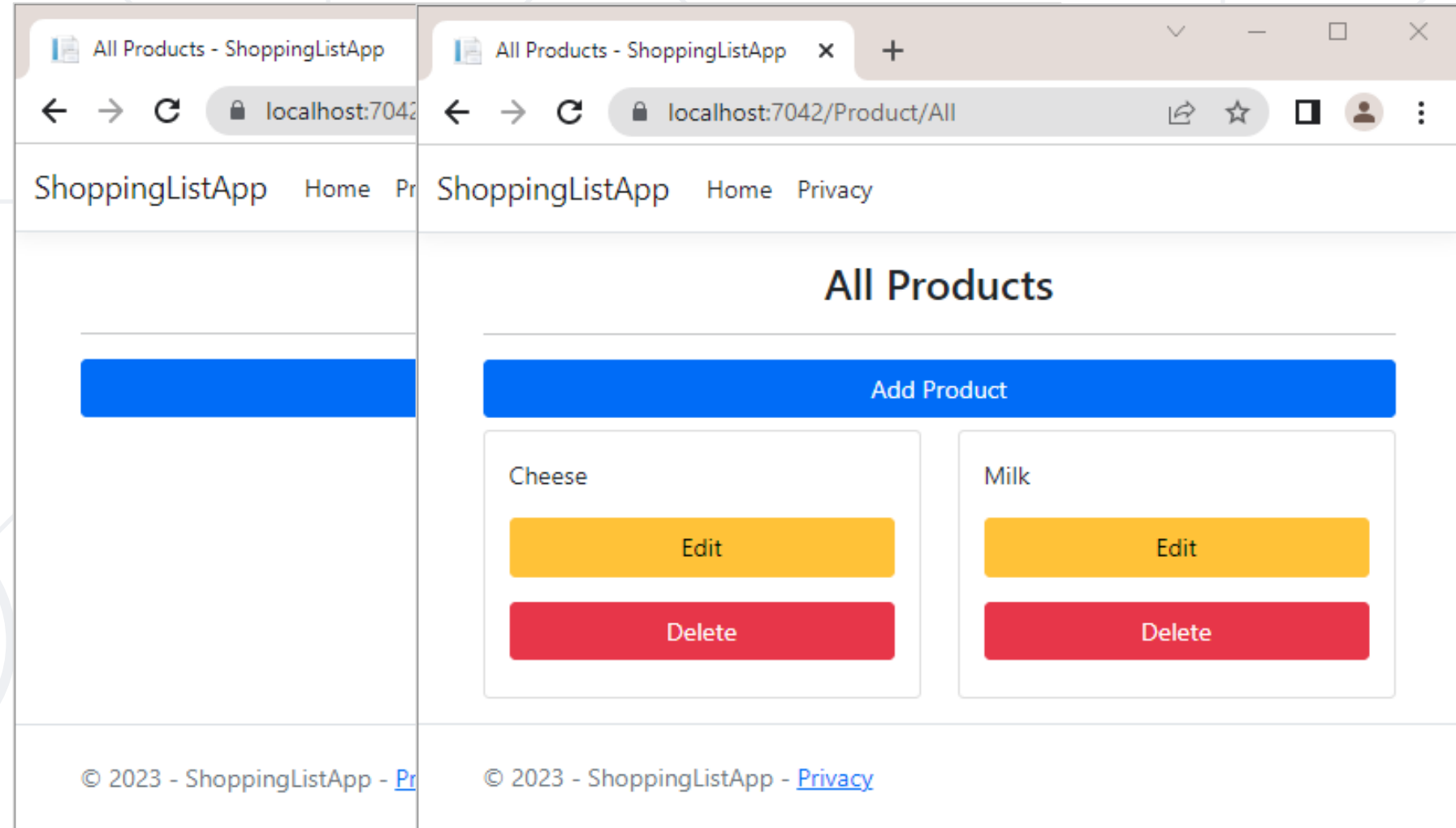
The "ShoppingList" App

- Create an MVC app with the **models** and **db context** from the previous slides
- Perform **CRUD operations** on the database to create the following functionalities:
 - Display all products
 - Add a product
 - Edit a product
 - Delete a product



The "All Products" Page (Reading Data)

- It should display **all added products** with their **content** + **[Edit]** and **[Delete]** buttons + **[Add Post]** button

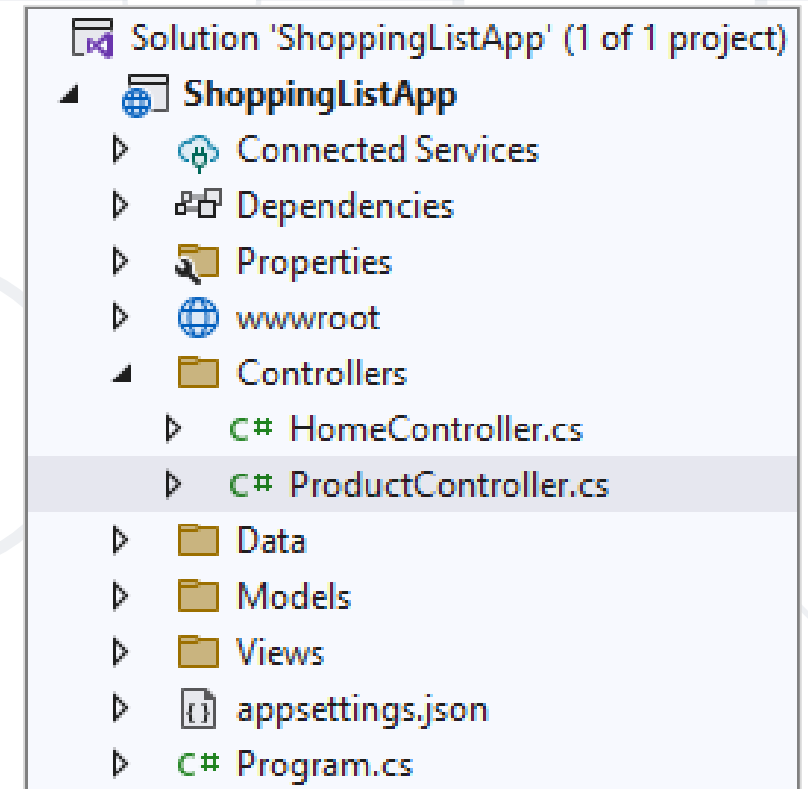


ProductsController Class

- Create a new **ProductController** in the "Controllers" folder
- Inject the **ShoppingListAppDbContext** through the **constructor**
 - And assign it to a **variable** to use it

```
public class ProductController : Controller
{
    private readonly ShoppingListAppDbContext _data;

    0 references
    public ProductController(ShoppingListAppDbContext data)
    => _data = data;
```



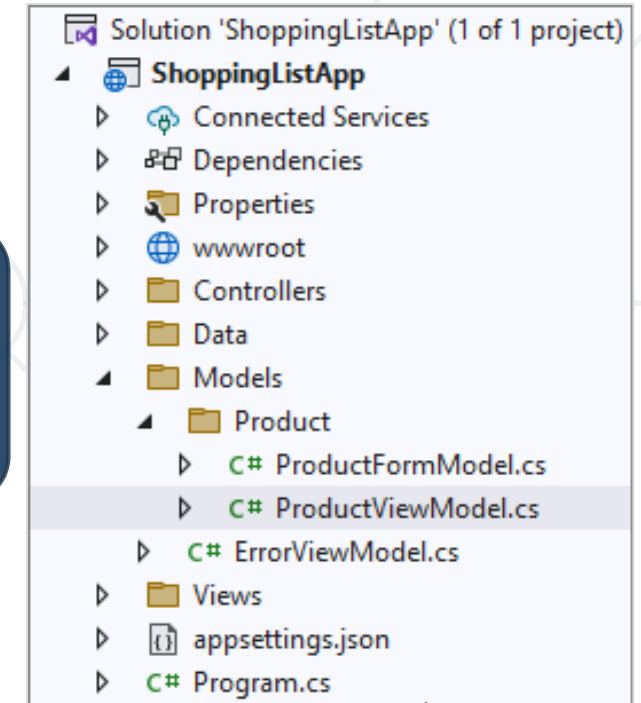
Reading Data (Controller + Model)

```
public class ProductController : Controller
{
    0 references
    public IActionResult All()
    {
        var products = _data
            .Products
            .Select(p => new ProductViewModel()
            {
                Id = p.Id,
                Name = p.Name,
            })
            .ToList();
        return View(products);
    }
}
```

Extract the
products from
the **database**

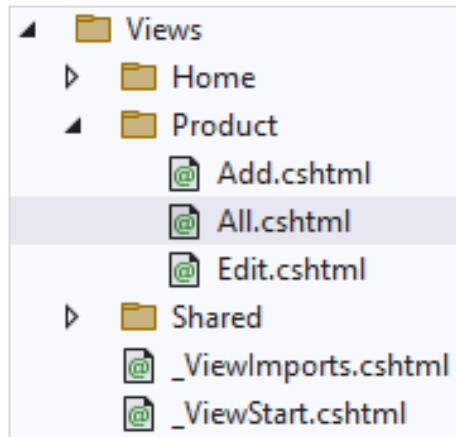
Project products
to a **model
collection**

Passes the model
collection to a **view**



```
public class ProductViewModel
{
    3 references
    public int Id { get; set; }
    2 references
    public string Name { get; set; } = null!;
}
```

Reading Data (View)

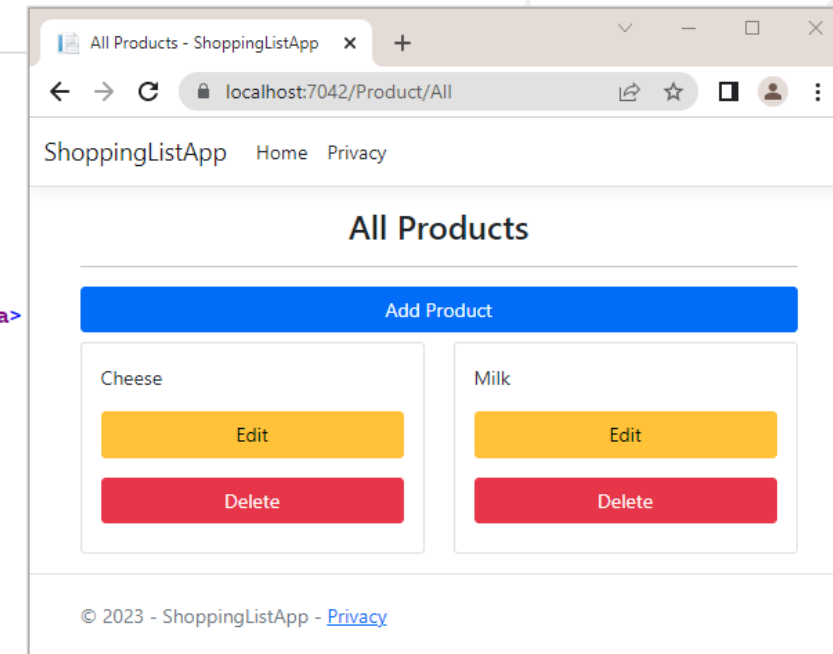


```
@model List<ProductViewModel>

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

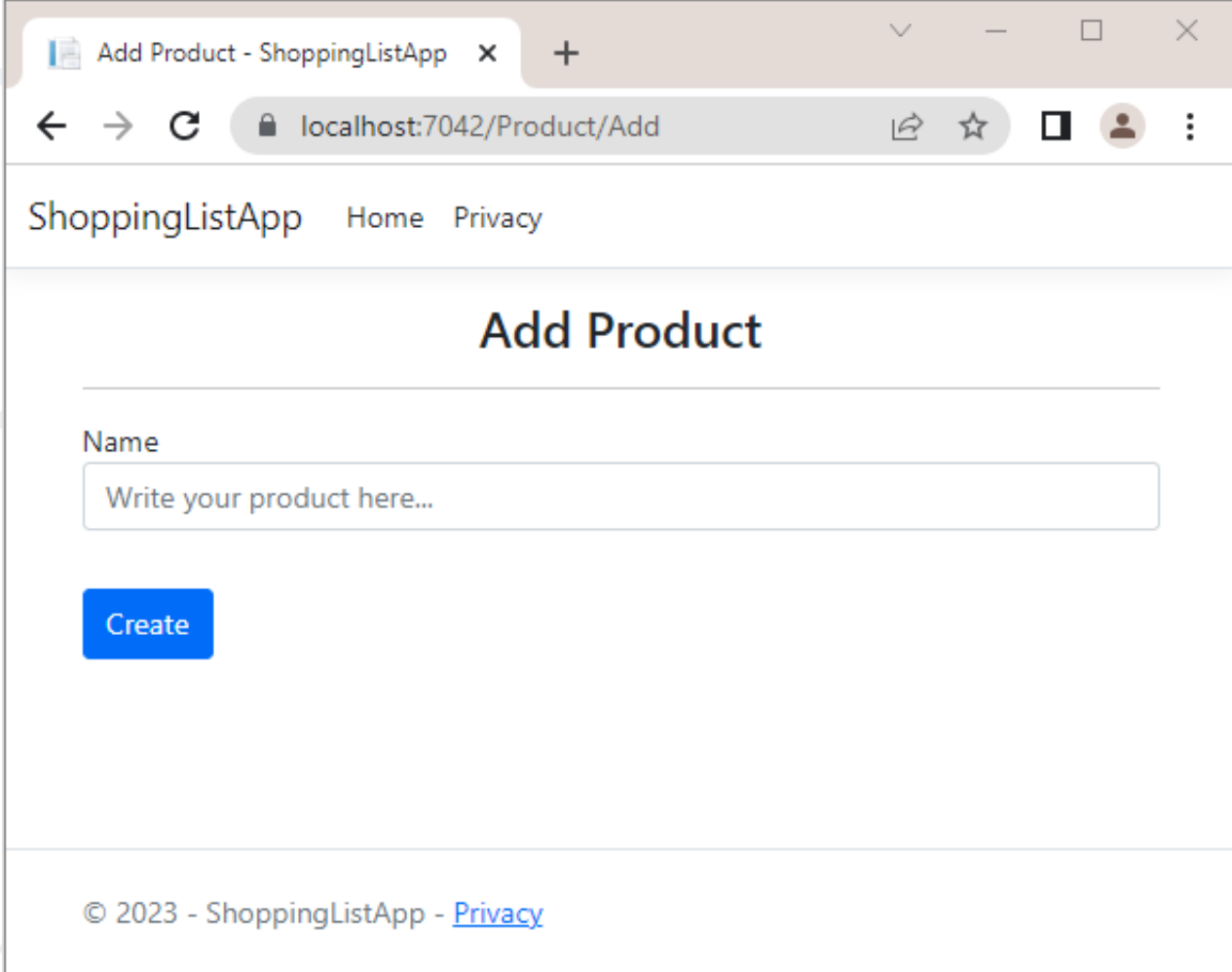
<h2 class="text-center">@ViewBag.Title</h2>
<hr />
<div class="d-grid gap-2 mb-2">
    <a asp-controller="Products" asp-action="Add" class="btn btn-primary">Add Product</a>
</div>

@if (Model.Count() > 0)
{
    <div class="row">
        @foreach (var product in Model)
        {
            <div class="col-sm-6">
                <div class="card">
                    <div class="card-body">
                        <p class="card-text">@product.Name</p>
                        <div class="d-grid gap-2">
                            <a asp-controller="Products" asp-action="Edit" asp-route-id="@product.Id" class="btn btn-warning">Edit</a>
                            <form class="mt-2" asp-controller="Products" asp-action="Delete" asp-route-id="@product.Id">
                                <div class="d-grid gap-2">
                                    <input type="submit" value="Delete" class="btn btn-danger mb-2" />
                                </div>
                            </form>
                        </div>
                    </div>
                </div>
            </div>
        }
    </div>
}
else
{
    <p class="text-center">No products yet!</p>
}
```



The "Add Product" Page (Creating New Data)

- It should display a **form** for adding a product



Add Product - ShoppingListApp

localhost:7042/Product/Add

ShoppingListApp Home Privacy

Add Product

Name

Create

© 2023 - ShoppingListApp - [Privacy](#)

Creating New Data (Controller + Model)

```
public class ProductController : Controller
{
    0 references
    public IActionResult Add()
        => View();

    [HttpPost]
    0 references
    public IActionResult Add(ProductFormModel model)
    {
        var product = new Product()
        {
            Name = model.Name
        };

        _data.Products.Add(product);
        _data.SaveChanges();

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

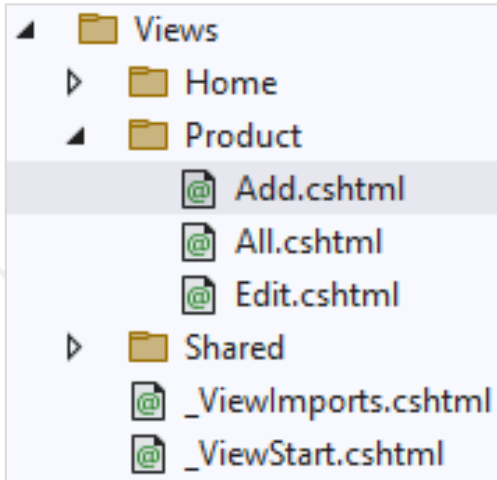
```
public class ProductViewModel
{
    3 references
    public int Id { get; set; }
    2 references
    public string Name { get; set; } = null!;
}
```

Create a new **Product** object

Add the object to the **DbSet**

Execute SQL statements

Creating New Data (View)

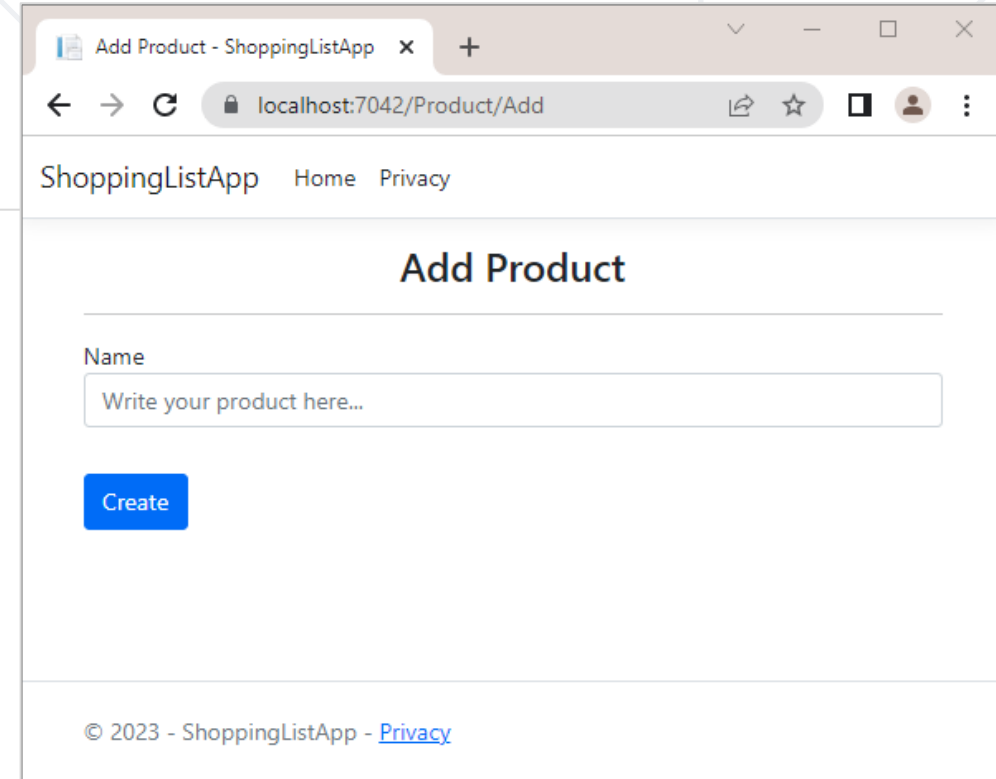


```
@model ProductFormModel

@{
    ViewBag.Title = "Add Product";
}

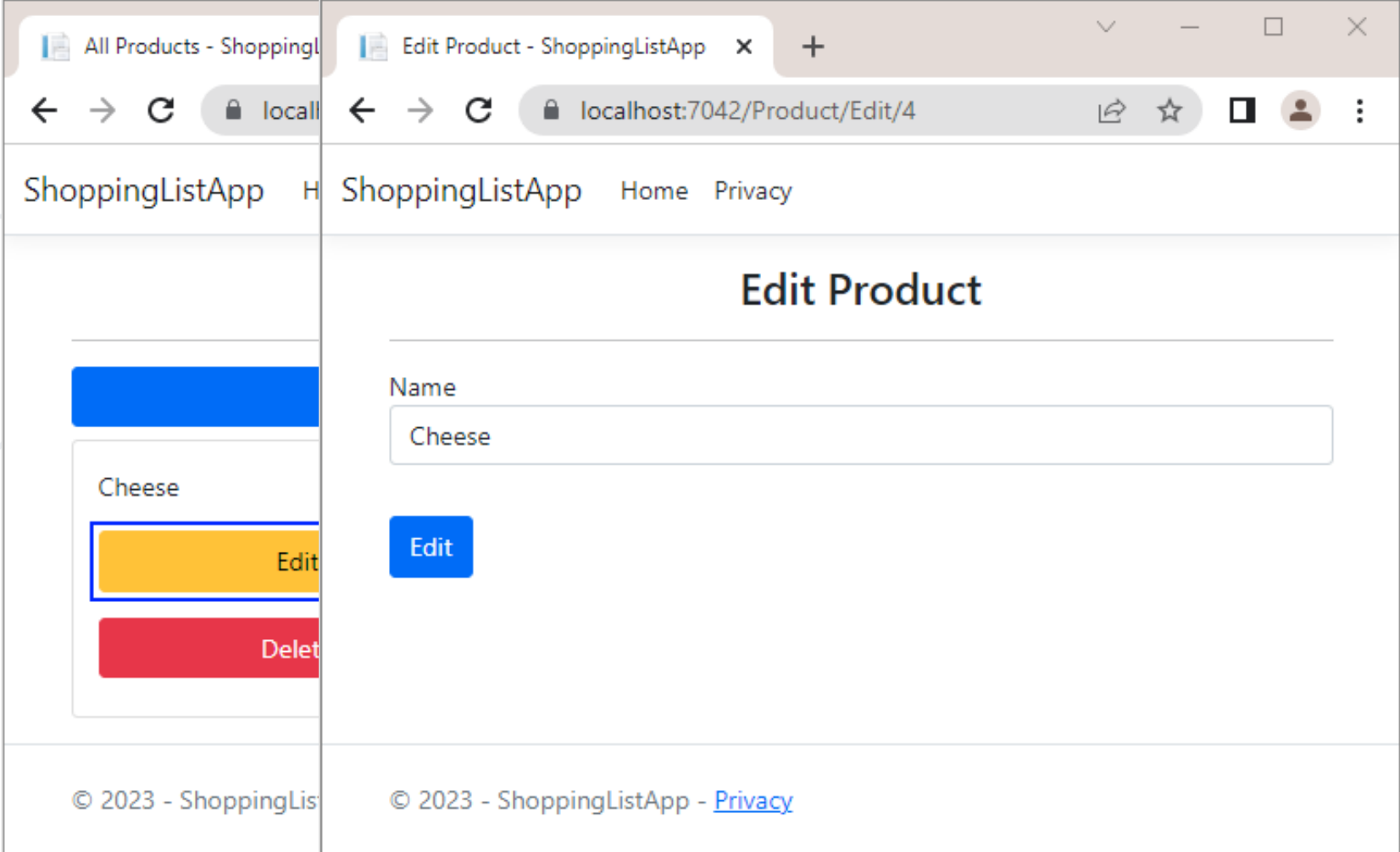
<h2 class="text-center">@ViewBag.Title</h2>
<hr />

<div class="row">
    <form method="post">
        <div class="form-group">
            <div class="mb-3">
                <label asp-for="Name"></label>
                <input asp-for="Name" class="form-control" placeholder="Write your product here...">
                <span asp-validation-for="Name" class="small text-danger"></span>
            </div>
        </div>
        <input class="btn btn-primary mt-3" type="submit" value="Create" />
    </form>
</div>
```



The "Edit Product" Page (Updating Existing Data)

- To **edit a product**, click on its **[Edit]** button
- It should display a **form for editing a product** with the product data in the fields



The screenshot displays the 'Edit Product' page of the ShoppingListApp. The browser's address bar shows the URL 'localhost:7042/Product/Edit/4'. The page features a navigation bar with 'ShoppingListApp', 'Home', and 'Privacy' links. The main content area is titled 'Edit Product' and contains a form with a 'Name' input field filled with 'Cheese' and a blue 'Edit' button. The footer includes the copyright notice '© 2023 - ShoppingListApp' and a link to the 'Privacy' page.

Updating Existing Data (Controller)

```
public class ProductController : Controller
{
    0 references
    public IActionResult Edit(int id)
    {
        var product = _data.Products.Find(id);

        return View(new ProductFormModel()
        {
            Name = product.Name
        });
    }
}
```

HTTP **GET** → display
the edit form

Pass a product
model to a **view**

Execute an
SQL UPDATE

HTTP **POST** →
update the DB

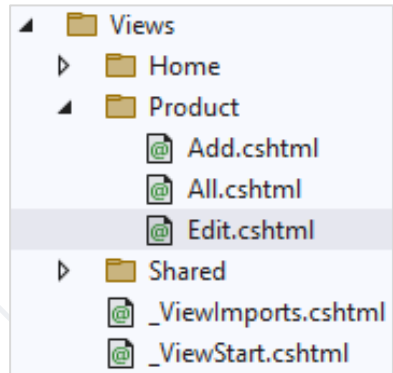
SELECT the
product for
update

```
[HttpPost]
0 references
public IActionResult Edit(int id, Product model)
{
    var product = _data.Products.Find(id);
    product.Name = model.Name;

    _data.SaveChanges();

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


Updating Existing Data (View)

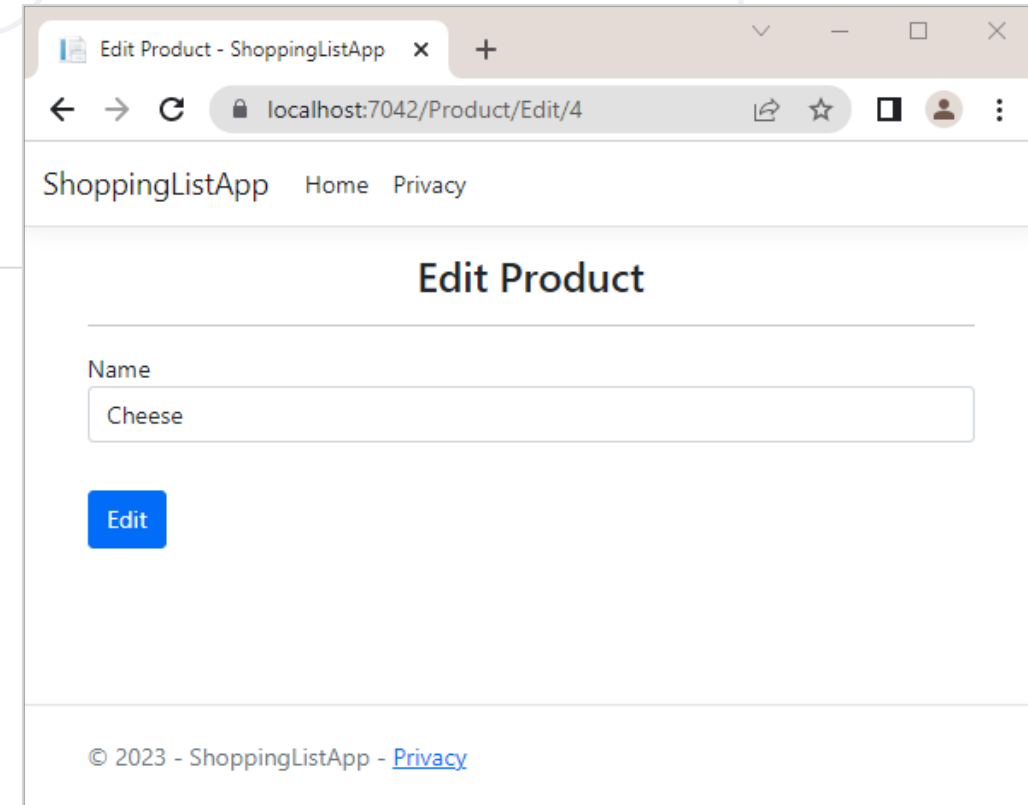


```
@model ProductFormModel

@{
    ViewBag.Title = "Edit Product";
}

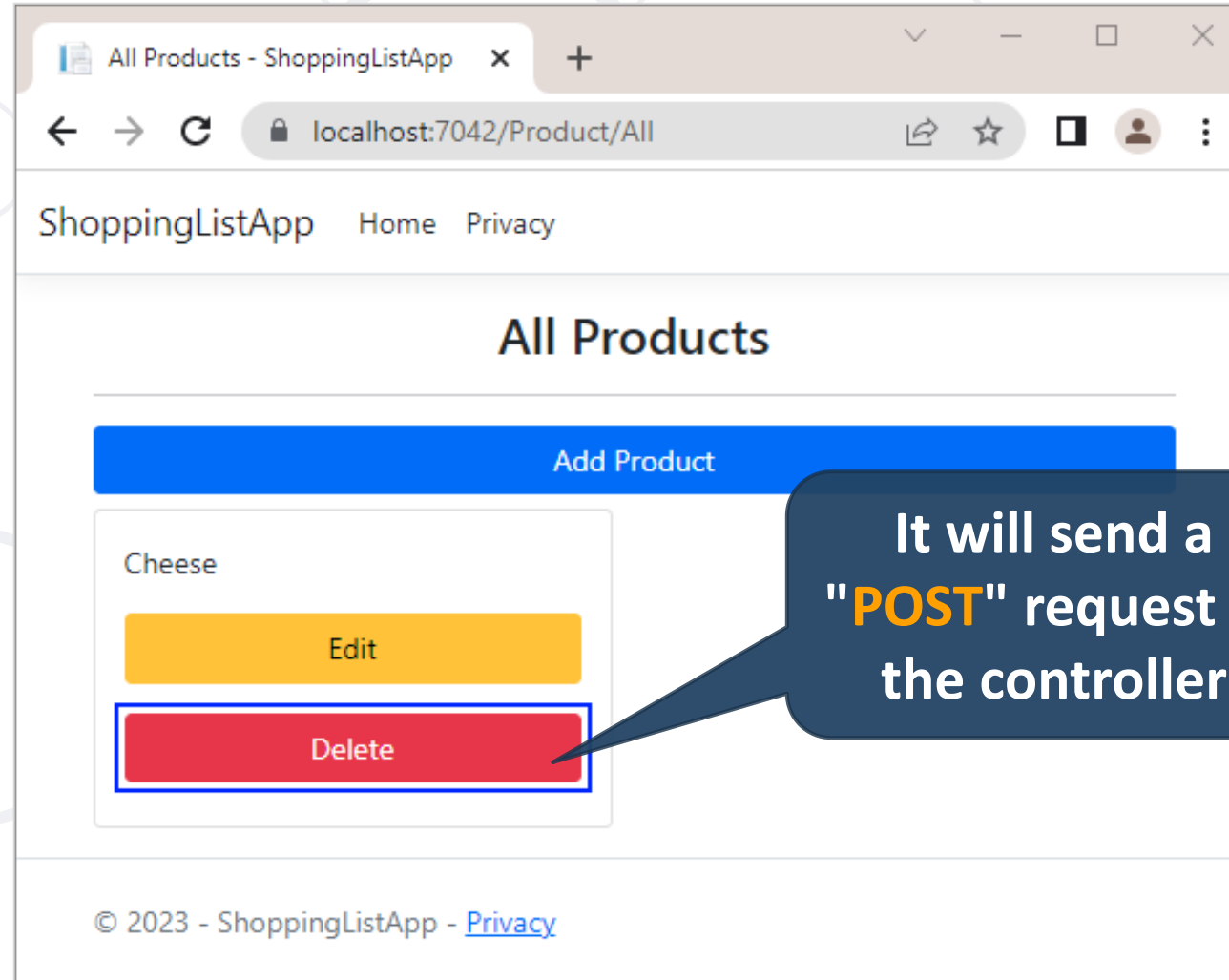
<h2 class="text-center">@ViewBag.Title</h2>
<hr />

<div class="row">
    <form method="post">
        <div class="form-group">
            <div class="mb-3">
                <label asp-for="Name"></label>
                <input asp-for="Name" class="form-control" placeholder="Write your product here...">
                <span asp-validation-for="Name" class="small text-danger"></span>
            </div>
        </div>
        <input class="btn btn-primary mt-3" type="submit" value="Edit" />
    </form>
</div>
```



Deleting Existing Data

- To delete a product, press its **[Delete]** button



Deleting Existing Data (Controller)

```
[HttpPost]
```

```
0 references
```

```
public IActionResult Delete(int id)
{
    var product = _data.Products.Find(id);

    _data.Products.Remove(product);
    _data.SaveChanges();

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

Execute the **SQL DELETE** command

Mark the entity for deleting at the next save



Database Migrations

Scripts for Modifying Table Structure in the DB

What Are Database Migrations?

- Updating database schema **without losing data**
 - Adding/dropping tables, columns, etc.
- **Migrations** in EF Core keep their **history**
 - Entity Classes, DB Context versions are all **preserved**
- **Automatically** generated by certain EF tools



```
└─ Migrations
   ├── C# 20230426142205_Initial.cs
   └── C# ShoppingListAppDbContextModelSnapshot.cs
```

- To **add a migration** in EF Core

- Use the **EF CLI Tools** `dotnet ef migrations add {MigrationName}`

- Use the **Package Manager Console** `Add-Migration {MigrationName}`

- To **undo a migration**, use one of the two ways

- `dotnet ef migrations remove {MigrationName}` **Remove-Migration**

Removes the last migration

- **Commit changes** to the database

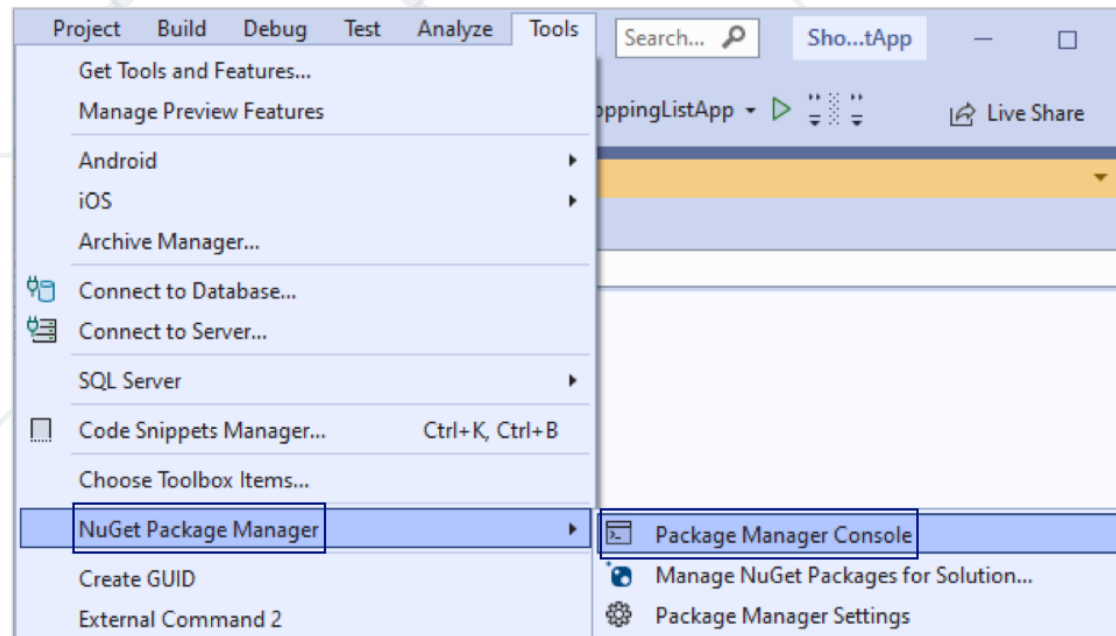
- `dotnet ef database update` **Update-Database**

- `db.Database.Migrate(); // Auto migrate at start`

Migrates any DB changes on startup

Migrate the "ShoppingListDemo" App (1)

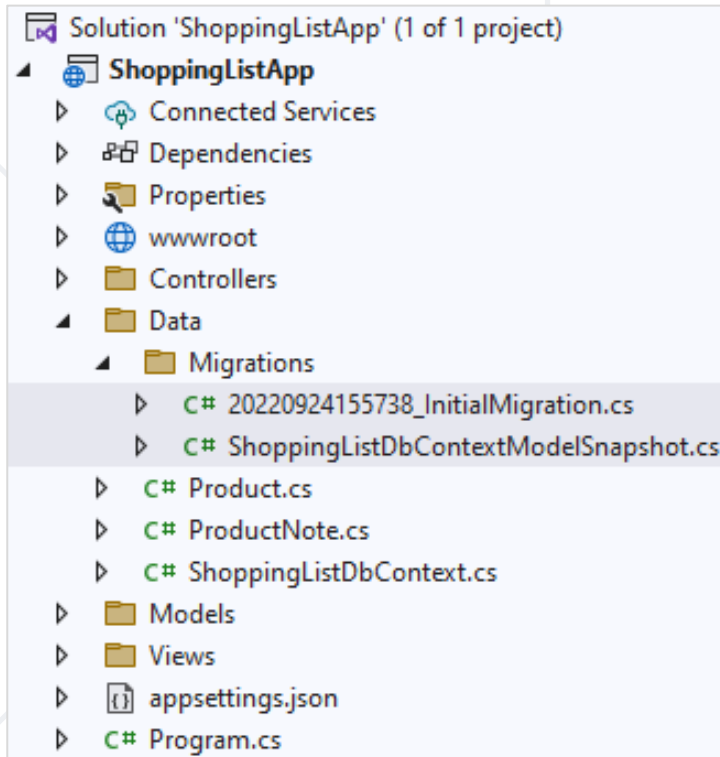
- Install the `Microsoft.EntityFrameworkCore.Tools` package
- Open the **Package Manager Console**
- Create a **migration**



```
Package Manager Console
Package source: All
Default project: ForumDemoApp
PM> Add-Migration InitialMigration -o Data/Migrations
```

Migration will be created in the "**Data/Migrations**" folder

Migrate the "ShoppingListDemo" App (2)



```
using Microsoft.EntityFrameworkCore.Migrations;

#nullable disable

namespace ShoppingListApp.Data.Migrations
{
    1 reference
    public partial class InitialMigration : Migration
    {
        0 references
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.CreateTable(
                name: "Products",
                columns: table => new
                {
                    Id = table.Column<int>(type: "int", nullable: false)
                        .Annotation("SqlServer:Identity", "1, 1"),
                    Name = table.Column<string>(type: "nvarchar(max)", nullable: false),
                    TestColumn = table.Column<string>(type: "nvarchar(max)", nullable: false)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_Products", x => x.Id);
                });

            migrationBuilder.CreateTable(
                name: "ProductNotes",
                columns: table => new
```


Auto Run Migration Scripts at Startup

```
public class ShoppingListDbContext : DbContext
{
    public ShoppingListDbContext(
        DbContextOptions<ShoppingListDbContext> options) :
        base(options)
        => Database.Migrate();
}
```

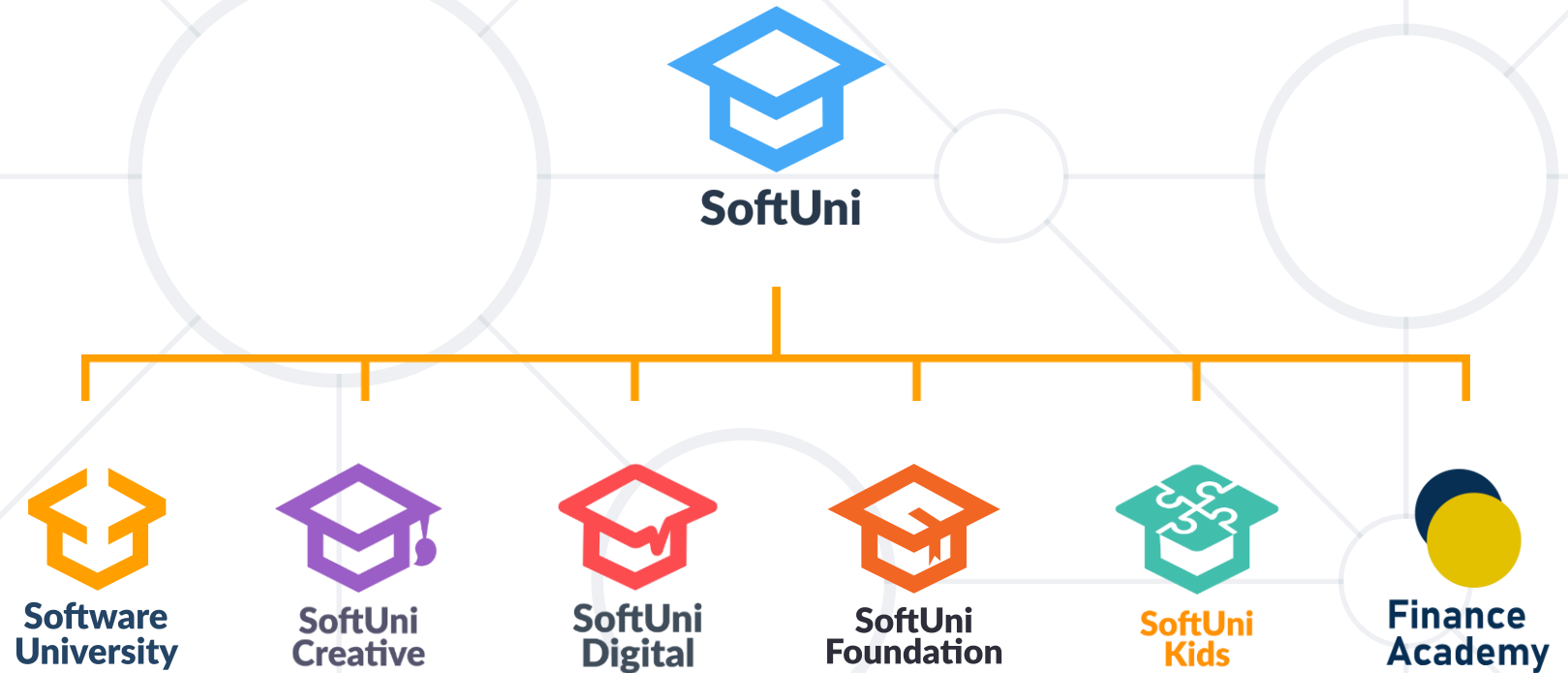
- This will **apply the migration scripts** (if not yet applied)
- Simple, but **can cause problems** → not recommended in production
- Recommended approach: migrate the database by hand

dotnet ef database update

- **EF Core** maps database objects to database schema
- **Code First approach** creates a database, based on C# classes that we create
- LINQ can be used to **query the DB** through the DB context
- **Database migrating** updates the database schema to match app data models



Questions?



SoftUni Diamond Partners

**SUPER
HOSTING
.BG**



**Coca-Cola HBC
Bulgaria**



POKERSTARS
POKER | CASINO | SPORTS
a Flutter International brand

INDEAVR
Serving the high achievers



AMBITIONED

 **DRAFT
KINGS**



**SOFTWARE
GROUP**

createX



Postbank
Решения за твоето утре

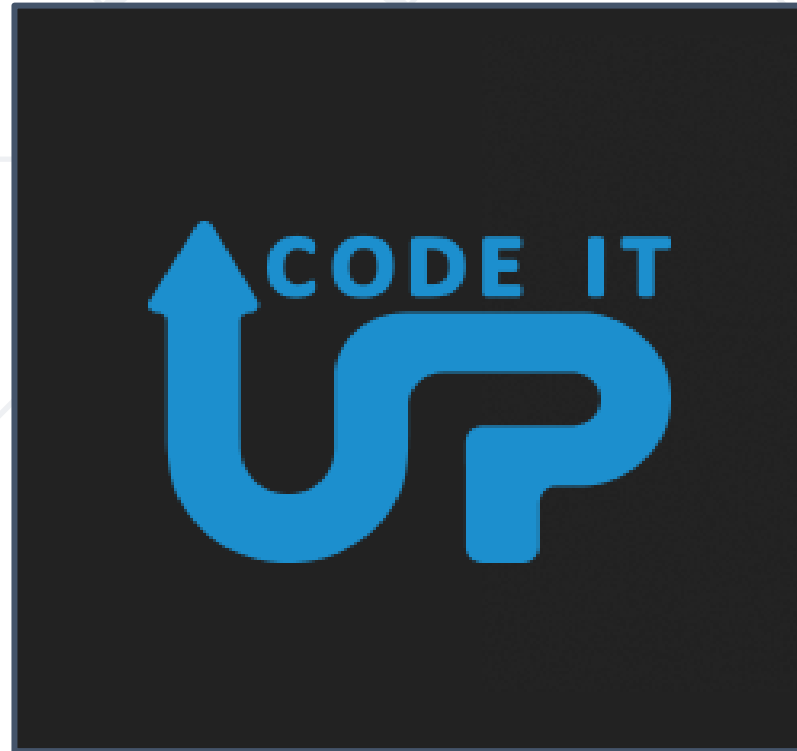


BOSCH

DXC
TECHNOLOGY



SmartIT



- Software University – High-Quality Education, Profession and Job for Software Developers

- softuni.bg, about.softuni.bg

- Software University Foundation

- softuni.foundation

- Software University @ Facebook

- facebook.com/SoftwareUniversity

- Software University Forums

- forum.softuni.bg



- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg/>
- © Software University – <https://softuni.bg>

