Table of Contents

Demo: Shop Database, CRUD

Author: Baifan Zhou
Some explanations are created with the help of ChatGPT.

Introduction

This demo covers asynchronous database communication, expanding the database schema using migrations, and implementing lazy loading for efficient data retrieval. Here's an overview of what you will learn:

- **Asynchronous Database Communication**:
  - Transition all database operations in the `ItemController` to asynchronous methods, improving application responsiveness and allowing tasks to run concurrently without blocking the main thread.
  - Update methods to use `async` and `await` keywords, and use asynchronous Entity Framework Core methods such as `ToListAsync()`, `FirstOrDefaultAsync()`, `FindAsync()`, and `SaveChangesAsync()`.

- **Expanding the Database with Migrations**:
  - Shift from using `Database.EnsureCreated()` to Entity Framework Core migrations, which provide a robust way to handle schema changes and database updates over time.
  - Install the `EntityFrameworkCore.Design` package and use the `dotnet ef` command-line tool to create and apply migrations, enabling you to manage schema changes and extend your database with new classes such as `Customer`, `Order`, and `OrderItem`.

- **Lazy Loading**:
  - Implement lazy loading to defer the loading of related entities until they are actually needed, which can improve performance and reduce initial data load.
  - Modify your model classes to include `virtual` navigation properties and configure lazy loading proxies by installing the relevant NuGet package and updating the `ItemDbContext` configuration.

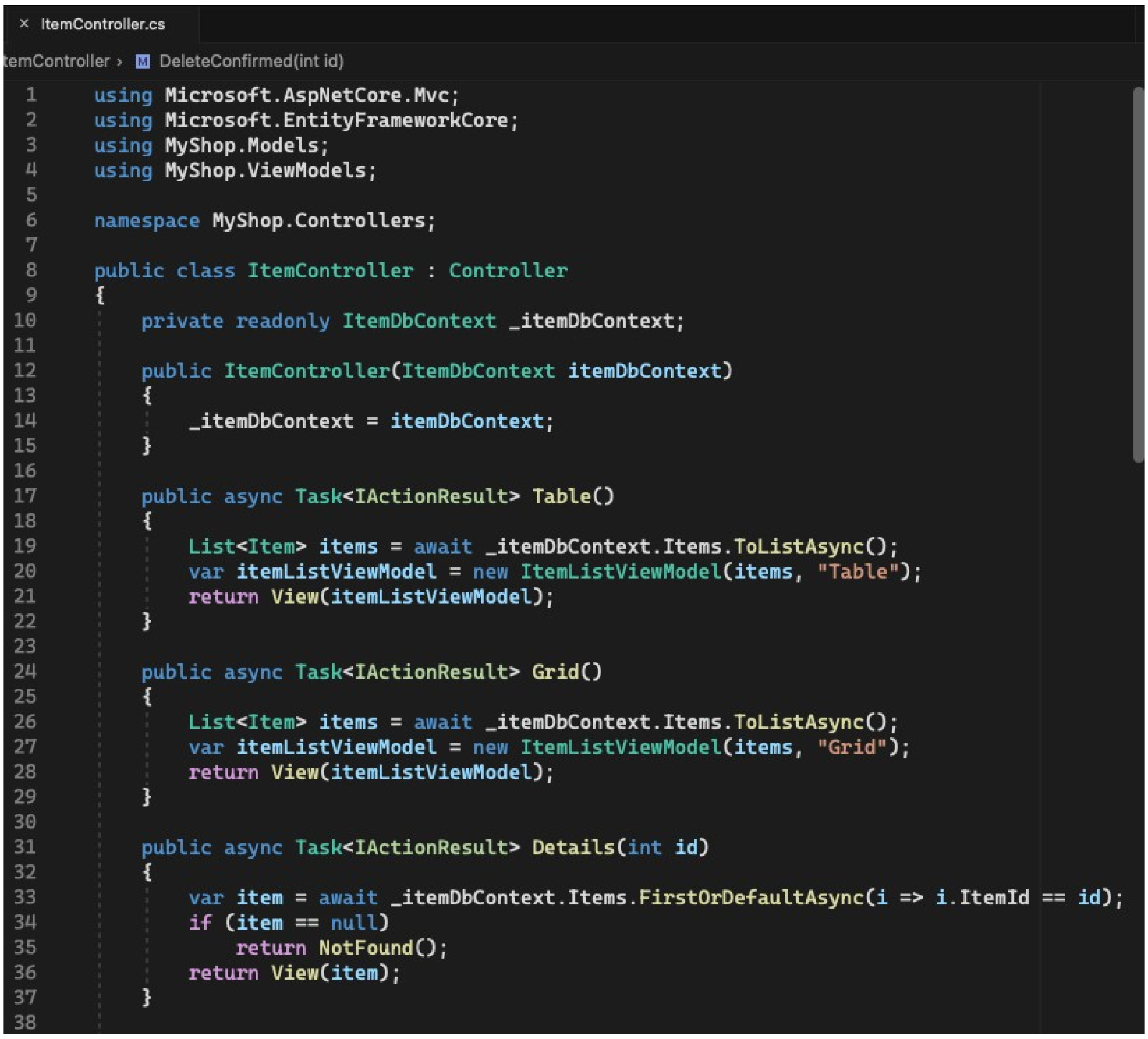
Happy coding!

Asynchronous Database Communication

Async methods allow your application to be more responsive. They allow the application to perform multiple tasks concurrently without blocking the main thread.
In traditional synchronous programming, when a time-consuming task is executed, the entire program execution is halted until that task is completed. During this time, the user interface becomes unresponsive, and the application appears to freeze, leading to a poor user experience.
When an asynchronous task is initiated, the program continues to execute other tasks or respond to user input without waiting for the completion of the time-consuming operation. The asynchronous task runs in the background, and when it is finished, it can notify the application to handle the results.

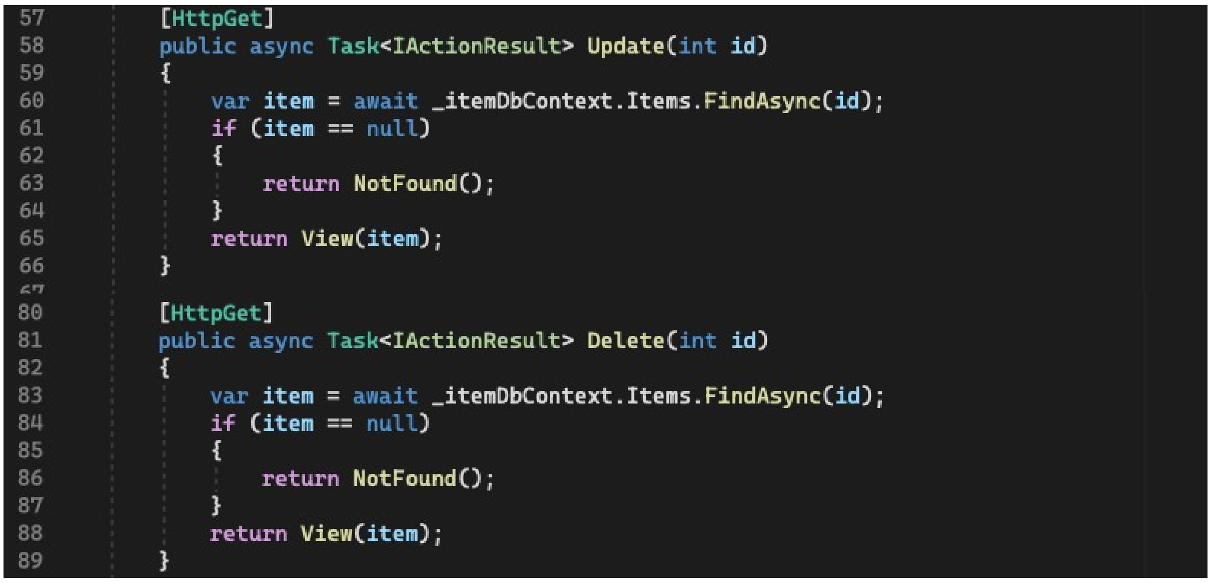
Now we change our communication with the database to asynchronous. Essentially, we need to change all methods that communicate with the database.

In the Controllers/ItemController.cs, update the methods to be async by adding the async keyword and returning Task<T>:
[HttpGet]: Table(), Grid(), Details(), [HttpGet]Update(), [HttpGet]Delete()
[HtttpPost]: [HtttpPost]Create(), [HtttPost]Update(), [HtttpPost]DeleteConfirmed()
Note: we need to import the EntityFrameworkCore (Line2) to use the async methods



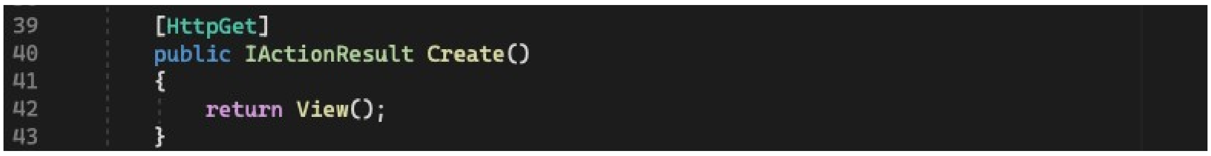
```
1   using Microsoft.AspNetCore.Mvc;
2   using Microsoft.EntityFrameworkCore;
3   using MyShop.Models;
4   using MyShop.ViewModels;
5
6   namespace MyShop.Controllers;
7
8   public class ItemController : Controller
9   {
10      private readonly ItemDbContext _itemDbContext;
11
12      public ItemController(ItemDbContext itemDbContext)
13      {
14          _itemDbContext = itemDbContext;
15      }
16
17      public async Task<IActionResult> Table()
18      {
19          List<Item> items = await _itemDbContext.Items.ToListAsync();
20          var itemListViewModel = new ItemListViewModel(items, "Table");
21          return View(itemListViewModel);
22      }
23
24      public async Task<IActionResult> Grid()
25      {
26          List<Item> items = await _itemDbContext.Items.ToListAsync();
27          var itemListViewModel = new ItemListViewModel(items, "Grid");
28          return View(itemListViewModel);
29      }
30
31      public async Task<IActionResult> Details(int id)
32      {
33          var item = await _itemDbContext.Items.FirstOrDefaultAsync(i => i.ItemId == id);
34          if (item == null)
35              return NotFound();
36          return View(item);
37      }
38
```

Continue to change the all HttpGet:



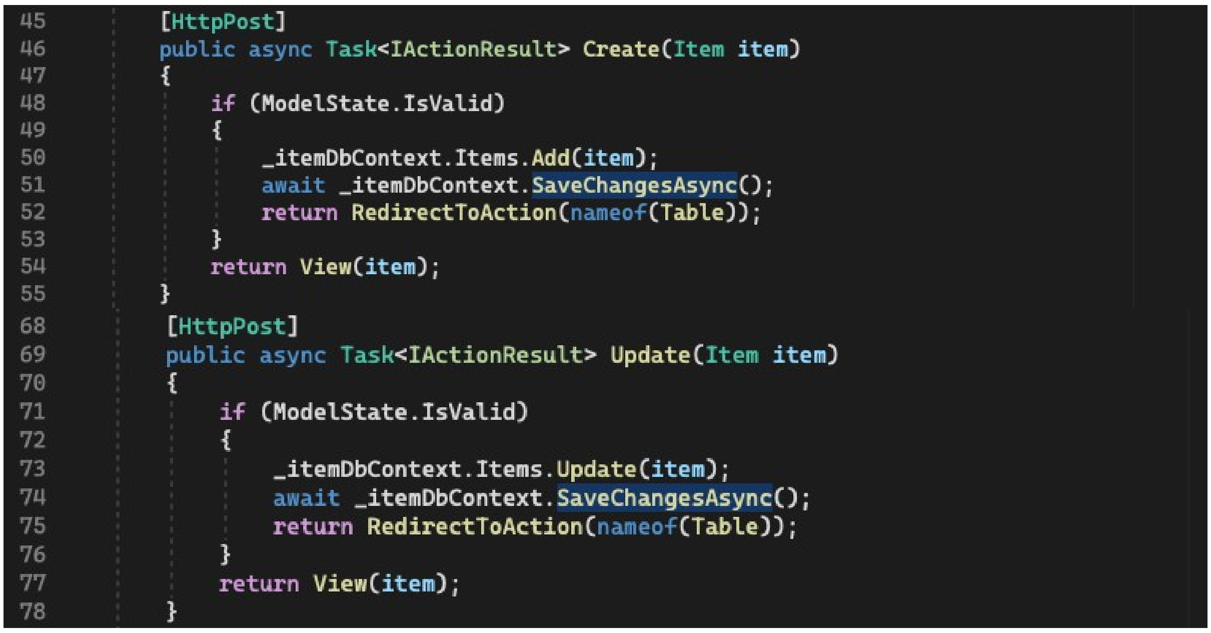
```
        [HttpGet]
        public async Task<IActionResult> Update(int id)
        {
            var item = await _itemDbContext.Items.FindAsync(id);
            if (item == null)
            {
                return NotFound();
            }
            return View(item);
        }

        [HttpGet]
        public async Task<IActionResult> Delete(int id)
        {
            var item = await _itemDbContext.Items.FindAsync(id);
            if (item == null)
            {
                return NotFound();
            }
            return View(item);
        }
```

Note: The HttpGet version of Create() does not communicate with the database and does not need the async method



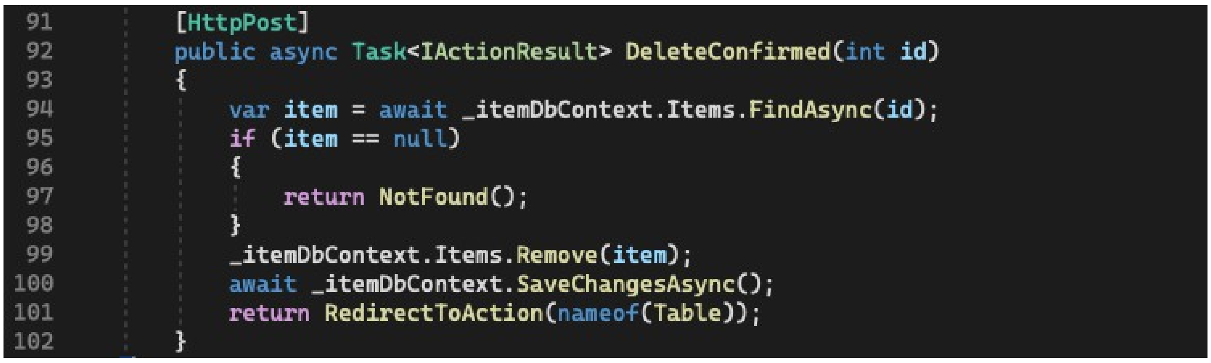
```
        [HttpGet]
        public IActionResult Create()
        {
            return View();
        }
```

Continue to change the all HttpPost:



```
        [HttpPost]
        public async Task<IActionResult> Create(Item item)
        {
            if (ModelState.IsValid)
            {
                _itemDbContext.Items.Add(item);
                await _itemDbContext.SaveChangesAsync();
                return RedirectToAction(nameof(Table));
            }
            return View(item);
        }
        [HttpPost]
        public async Task<IActionResult> Update(Item item)
        {
            if (ModelState.IsValid)
            {
                _itemDbContext.Items.Update(item);
                await _itemDbContext.SaveChangesAsync();
                return RedirectToAction(nameof(Table));
            }
            return View(item);
        }
```

Note:
The Add() method only marks the entity as added in the context and does not immediately persist it to the database. It is therefore not asynchronous. Update() is likewise.
The SaveChangesAsync() method is used to persist all the changes made in the context to the database asynchronously.

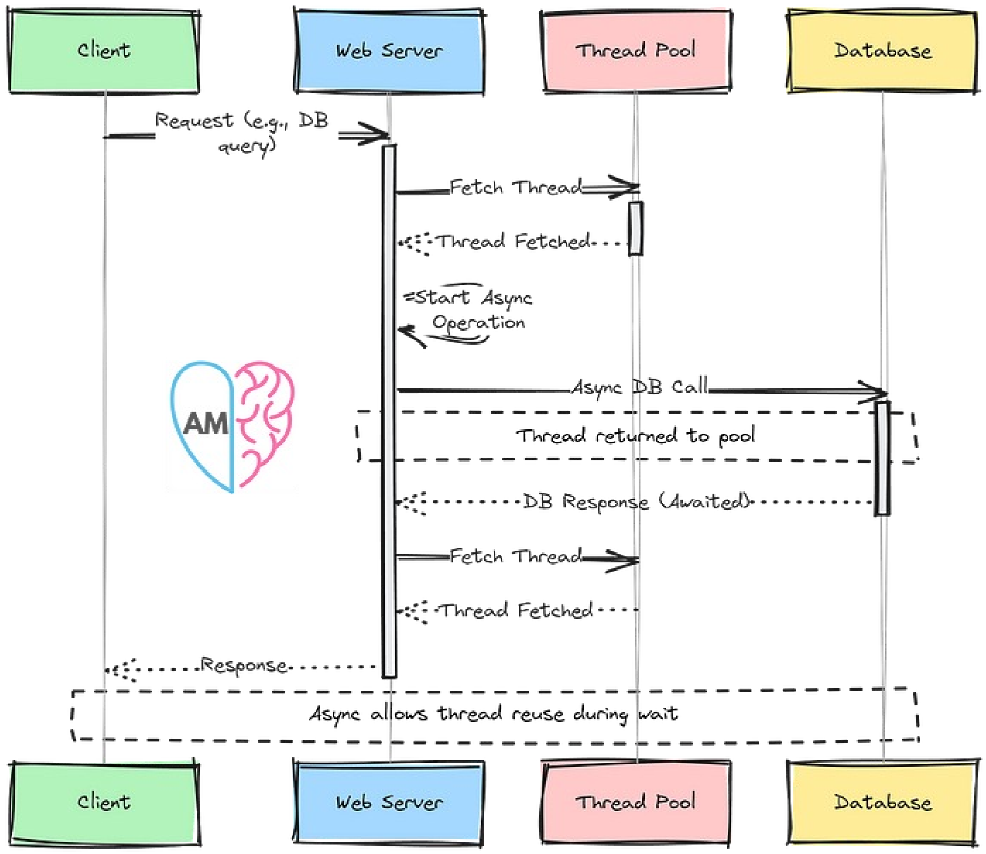


```
[HttpPost]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    var item = await _itemDbContext.Items.FindAsync(id);
    if (item == null)
    {
        return NotFound();
    }
    _itemDbContext.Items.Remove(item);
    await _itemDbContext.SaveChangesAsync();
    return RedirectToAction(nameof(Table));
}
```

All the methods that perform database operations are added an “await” keyword and changed to their async version. These methods include:

- ToListAsync()
- FirstOrDefaultAsync()
- FindAsync()
- SaveChangesAsync()

Read more:
https://medium.com/@vitormoschetta/understanding-synchronous-and-asynchronous-communication-in-software-development-db914a8d7947
https://admirmujkic.medium.com/essential-strategies-and-practices-for-entity-framework-core-595cb7d57782 (Source of the figure below, accessed 30-07-2024)
https://geekbot.com/blog/7-examples-of-asynchronous-communication-at-work-how-to-best-use-them/ (extended reading :)



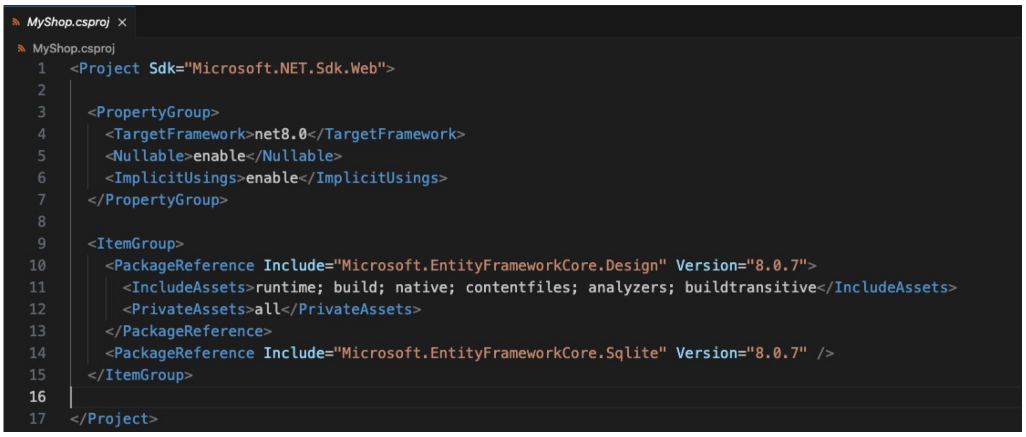
Expand the Database with Migrations

For a shop management system, we need more classes, such as customers, order, etc.
You can image that this update of new classes and properties of classes may also happen in the future.
Previously, we used Database.EnsureCreated() to create a database. It is a simple and convenient way to create a database directly from the DbContext.
It is mainly intended for development and testing scenarios when you want to quickly create a database. It only creates the database schema based on the current model defined in your DbContext. As a result, it doesn't allow for database schema updates or data migrations in future changes to your entities.

Now we want to adopt Migrations, a more robust and scalable way to manage database schema changes over time. Migrations enable you to evolve your database schema and data across multiple application versions.
To do so, we first need to install a NuGet package, EntityFrameworkCore.Design:

```
dotnet add package Microsoft.EntityFrameworkCore.Design --version="8.*"
```

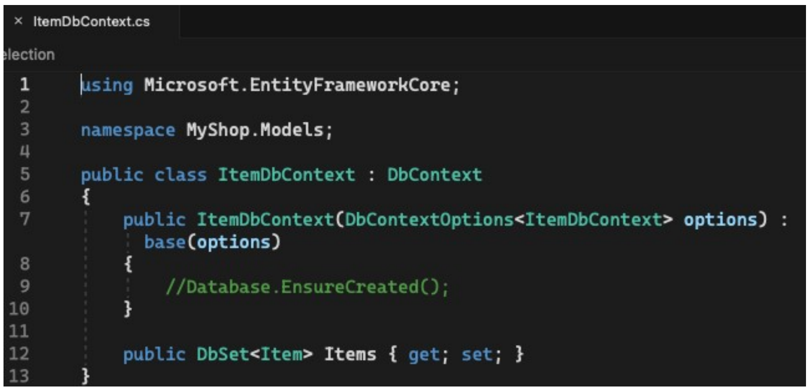
The new NuGet package should appear in MyShop.csproj



```xml
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="8.0.7">
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
      <PrivateAssets>all</PrivateAssets>
    </PackageReference>
    <PackageReference Include="Microsoft.EntityFrameworkCore.Sqlite" Version="8.0.7" />
  </ItemGroup>

</Project>
```

Now **backup** the previous database “ItemDatabase.db” and **DELETE** it from the project.

Comment out the Database.EnsureCreated(); (Line9) in ItemDbContext.cs
This is important! Otherwise, you will get: SQLite Error 1: 'table "Items" already exists'.
This issue typically arises during database migrations or when running code that attempts to create tables without first checking if they already exist.



```csharp
1   using Microsoft.EntityFrameworkCore;
2
3   namespace MyShop.Models;
4
5   public class ItemDbContext : DbContext
6   {
7       public ItemDbContext(DbContextOptions<ItemDbContext> options) :
          base(options)
8       {
9           //Database.EnsureCreated();
10      }
11
12      public DbSet<Item> Items { get; set; }
13  }
```

In the next we will create ItemDatabase.db with Migrations.
Open terminal from the project, install the dotnet ef tool by running:

```
dotnet tool install --global dotnet-ef
```

If you encounter such messages

```
Tools directory '/Users/[UserName]/.dotnet/tools' is not currently on the
PATH environment variable.
```

You can first add the tool directory to your PATH.
For Mac zsh: run

```
cat << \EOF >> ~/.zprofile
# Add .NET Core SDK tools
export PATH="$PATH:/Users/baifanz/.dotnet/tools"
EOF
```

And run `zsh -l` to make it available for the current session.

For bash, run

```
echo 'export PATH="$PATH:/Users/baifanz/.dotnet/tools"' >> ~/.bash_profile
source ~/.bash_profile
```

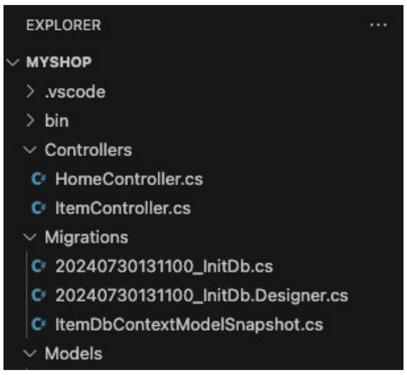
Verify ef installation, run:

```
dotnet ef –version
```

Then create the initial migrations: (if it fails, first rebuild the project, then run the command)

```
dotnet ef migrations add InitDb
```

A folder called Migrations should appear:

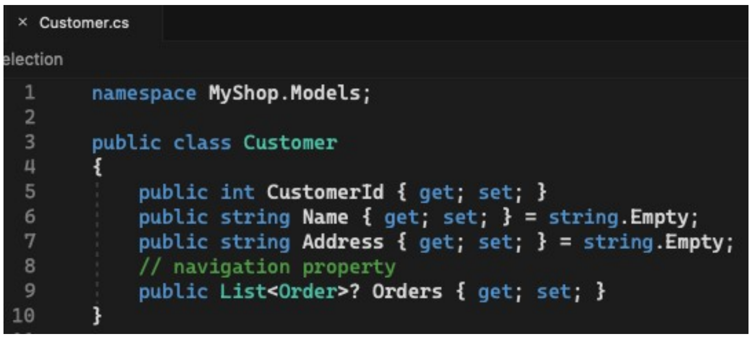


Then create the ItemDatabase.db with the following terminal command:

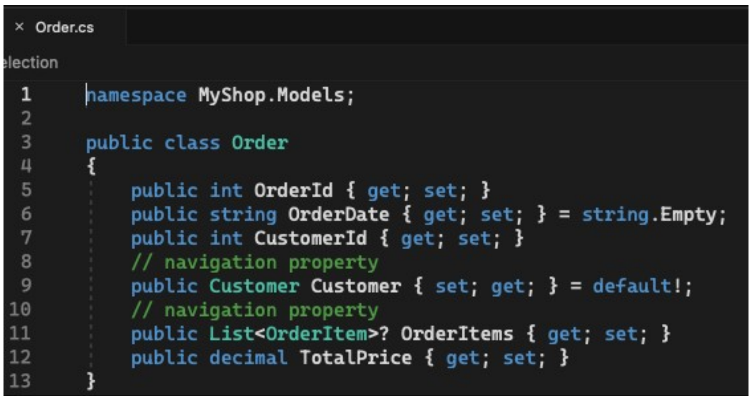
```
dotnet ef database update
```

Now the same ItemDatabase.db should appear. Different from the previous version, this one we can expand with more classes.

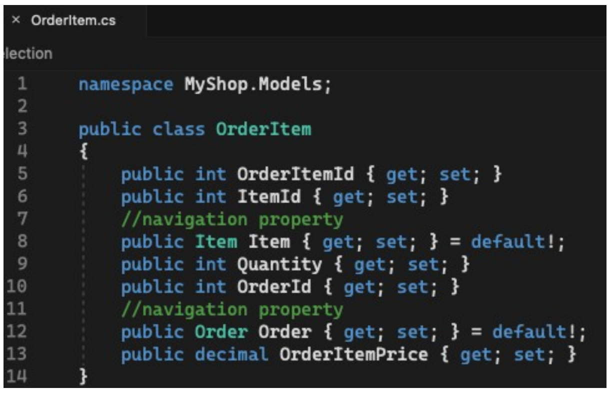
We now create some classes: Customer, Order, OrderItem, where Customer can have many Orders, one Order consists of multiple OrderItems. One OrderItem specifies which Item is in the Order and its quantity.
Create new classes under Models/ (Pay attention to the non-nullable constraints, string.Empty for strings, and default! For classes.)


× Customer.cs

```csharp
namespace MyShop.Models;

public class Customer
{
    public int CustomerId { get; set; }
    public string Name { get; set; } = string.Empty;
    public string Address { get; set; } = string.Empty;
    // navigation property
    public List<Order>? Orders { get; set; }
}
```

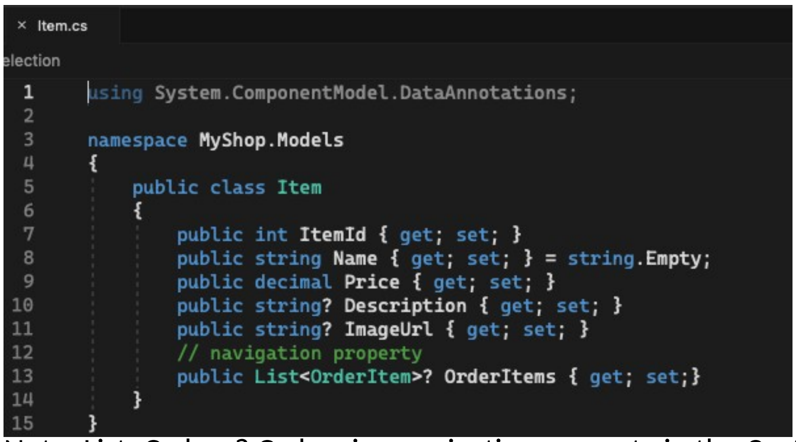

× Order.cs

```csharp
namespace MyShop.Models;

public class Order
{
    public int OrderId { get; set; }
    public string OrderDate { get; set; } = string.Empty;
    public int CustomerId { get; set; }
    // navigation property
    public Customer Customer { set; get; } = default!;
    // navigation property
    public List<OrderItem>? OrderItems { get; set; }
    public decimal TotalPrice { get; set; }
}
```


× OrderItem.cs

```csharp
namespace MyShop.Models;

public class OrderItem
{
    public int OrderItemId { get; set; }
    public int ItemId { get; set; }
    //navigation property
    public Item Item { get; set; } = default!;
    public int Quantity { get; set; }
    public int OrderId { get; set; }
    //navigation property
    public Order Order { get; set; } = default!;
    public decimal OrderItemPrice { get; set; }
}
```

Modify the Item.cs:


× Item.cs

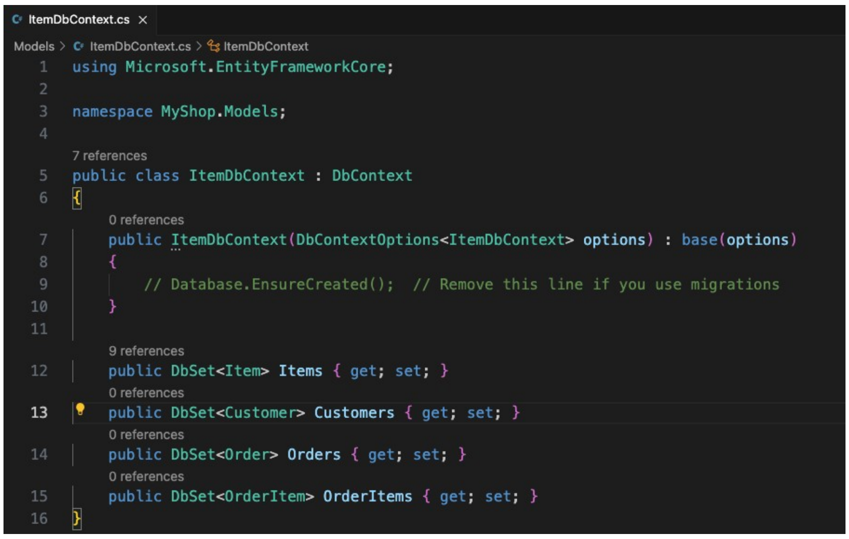
```csharp
using System.ComponentModel.DataAnnotations;

namespace MyShop.Models
{
    public class Item
    {
        public int ItemId { get; set; }
        public string Name { get; set; } = string.Empty;
        public decimal Price { get; set; }
        public string? Description { get; set; }
        public string? ImageUrl { get; set; }
        // navigation property
        public List<OrderItem>? OrderItems { get; set;}
    }
}
```

Note: List<Order>? Orders is a navigation property in the Customer class. A navigation property in an entity class represents a relationship between entities and allows you to navigate from one entity to related entities.

In this case, the Orders property is a navigation property that represents the relationship between a Customer and their associated Order entities. By having the Orders property in the Customer class, you can easily access all the orders that belong to a specific customer.

Similarly, these navigation properties exist in Order (Customer, OrderItem), OrderItem(Item, Order), Item(OrderItem).

Modify the Models/ItemDbContext.cs to include the newly added classes into the database:

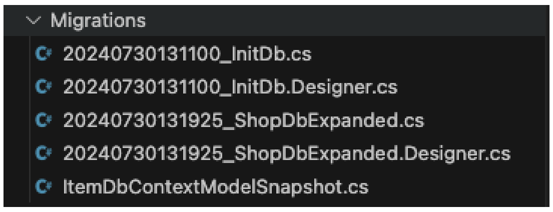


```
using Microsoft.EntityFrameworkCore;

namespace MyShop.Models;

7 references
public class ItemDbContext : DbContext
{
    0 references
    public ItemDbContext(DbContextOptions<ItemDbContext> options) : base(options)
    {
        // Database.EnsureCreated();  // Remove this line if you use migrations
    }

    9 references
    public DbSet<Item> Items { get; set; }
    0 references
    public DbSet<Customer> Customers { get; set; }
    0 references
    public DbSet<Order> Orders { get; set; }
    0 references
    public DbSet<OrderItem> OrderItems { get; set; }
}
```

Run the command in Terminal: (rebuild with Command/Control + B in case it fails)

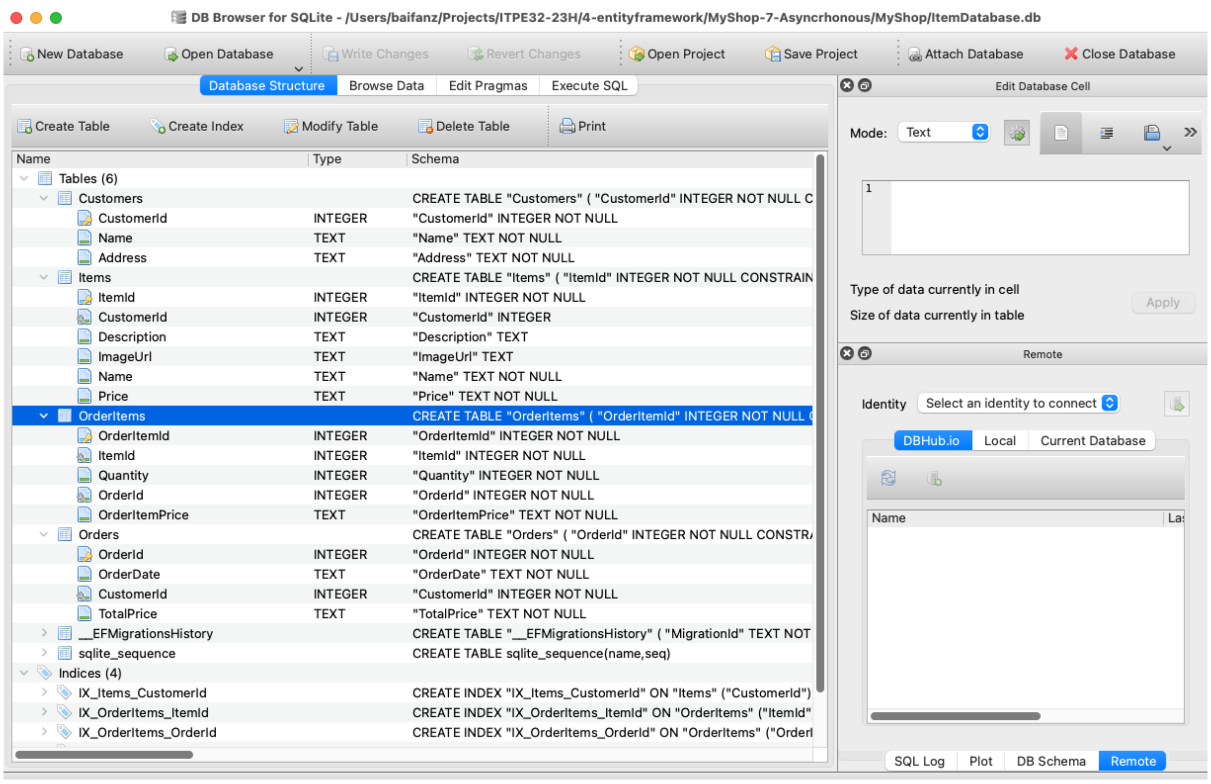
```
dotnet ef migrations add ShopDbExpanded
dotnet ef database update
```

These commands add a new version of Migrations and update the database with the new classes



```
Migrations
  20240730131100_InitDb.cs
  20240730131100_InitDb.Designer.cs
  20240730131925_ShopDbExpanded.cs
  20240730131925_ShopDbExpanded.Designer.cs
  ItemDbContextModelSnapshot.cs
```

Open the new ItemDatabase.db with DB Browser for SQLite, you will see the expanded data structure (no data content now because we have not added them)



DB Browser for SQLite - /Users/baifanz/Projects/ITPE32-23H/4-entityframework/MyShop-7-Asyncrhonous/MyShop/ItemDatabase.db

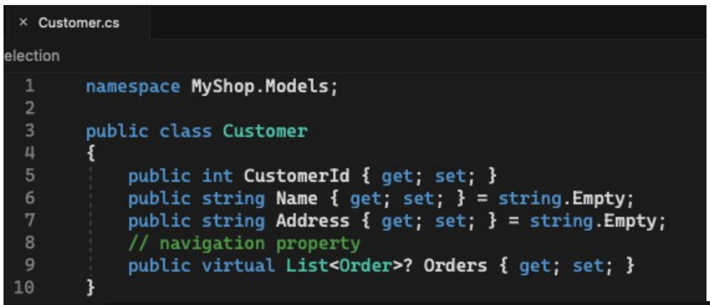
Mode: Text

Name	Type	Schema
Tables (6)		
Customers		CREATE TABLE "Customers" ("CustomerId" INTEGER NOT NULL C
CustomerId	INTEGER	"CustomerId" INTEGER NOT NULL
Name	TEXT	"Name" TEXT NOT NULL
Address	TEXT	"Address" TEXT NOT NULL
Items		CREATE TABLE "Items" ("ItemId" INTEGER NOT NULL CONSTRAIN
ItemId	INTEGER	"ItemId" INTEGER NOT NULL
CustomerId	INTEGER	"CustomerId" INTEGER
Description	TEXT	"Description" TEXT
ImageUrl	TEXT	"ImageUrl" TEXT
Name	TEXT	"Name" TEXT NOT NULL
Price	TEXT	"Price" TEXT NOT NULL
OrderItems		CREATE TABLE "OrderItems" ("OrderItemId" INTEGER NOT NULL C
OrderItemId	INTEGER	"OrderItemId" INTEGER NOT NULL
ItemId	INTEGER	"ItemId" INTEGER NOT NULL
Quantity	INTEGER	"Quantity" INTEGER NOT NULL
OrderId	INTEGER	"OrderId" INTEGER NOT NULL
OrderItemPrice	TEXT	"OrderItemPrice" TEXT NOT NULL
Orders		CREATE TABLE "Orders" ("OrderId" INTEGER NOT NULL CONSTRA
OrderId	INTEGER	"OrderId" INTEGER NOT NULL
OrderDate	TEXT	"OrderDate" TEXT NOT NULL
CustomerId	INTEGER	"CustomerId" INTEGER NOT NULL
TotalPrice	TEXT	"TotalPrice" TEXT NOT NULL
__EFMigrationsHistory		CREATE TABLE "__EFMigrationsHistory" ("MigrationId" TEXT NOT
sqlite_sequence		CREATE TABLE sqlite_sequence(name,seq)
Indices (4)		
IX_Items_CustomerId		CREATE INDEX "IX_Items_CustomerId" ON "Items" ("CustomerId")
IX_OrderItems_ItemId		CREATE INDEX "IX_OrderItems_ItemId" ON "OrderItems" ("ItemId"
IX_OrderItems_OrderId		CREATE INDEX "IX_OrderItems_OrderId" ON "OrderItems" ("OrderI

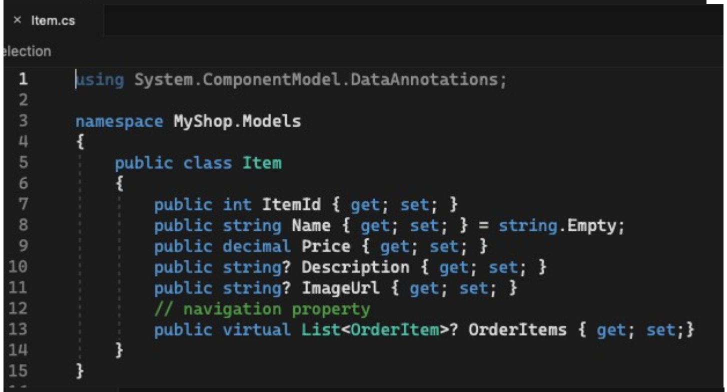
Lazy Loading

In ORM (Object-Relational Mapping) frameworks like Entity Framework Core (EF Core), a technique called lazy loading can defer the loading of related entities until they are accessed. It allows you to load related data from the database on-demand, rather than loading all related data eagerly upfront. This can lead to more efficient data retrieval and improved performance, as only the required data is fetched when needed.

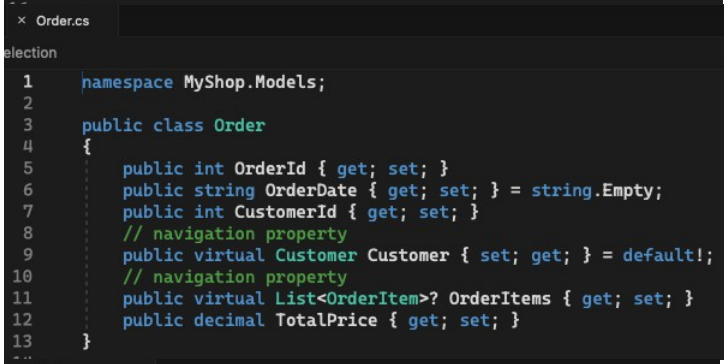
To enable lazy loading, first we add the keyword “virtual” before all the navigation properties: Customer(List<Order>), Item(List<OrderItem>), Order(Customer, List<OrderItem), OrderItem(Item, Order).



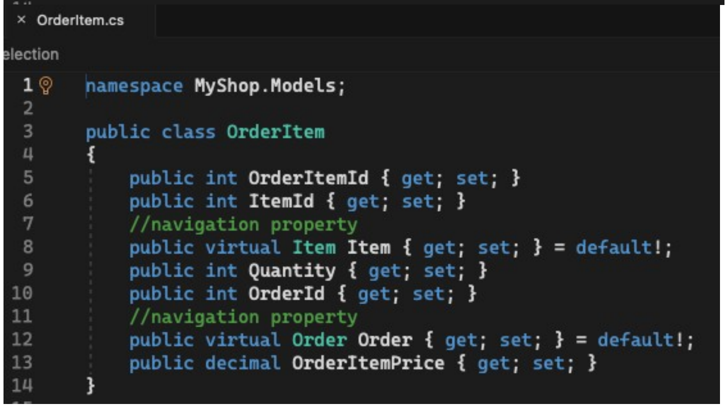
```
namespace MyShop.Models;

public class Customer
{
    public int CustomerId { get; set; }
    public string Name { get; set; } = string.Empty;
    public string Address { get; set; } = string.Empty;
    // navigation property
    public virtual List<Order>? Orders { get; set; }
}
```



```
using System.ComponentModel.DataAnnotations;

namespace MyShop.Models
{
    public class Item
    {
        public int ItemId { get; set; }
        public string Name { get; set; } = string.Empty;
        public decimal Price { get; set; }
        public string? Description { get; set; }
        public string? ImageUrl { get; set; }
        // navigation property
        public virtual List<OrderItem>? OrderItems { get; set;}
    }
}
```



```
namespace MyShop.Models;

public class Order
{
    public int OrderId { get; set; }
    public string OrderDate { get; set; } = string.Empty;
    public int CustomerId { get; set; }
    // navigation property
    public virtual Customer Customer { set; get; } = default!;
    // navigation property
    public virtual List<OrderItem>? OrderItems { get; set; }
    public decimal TotalPrice { get; set; }
}
```

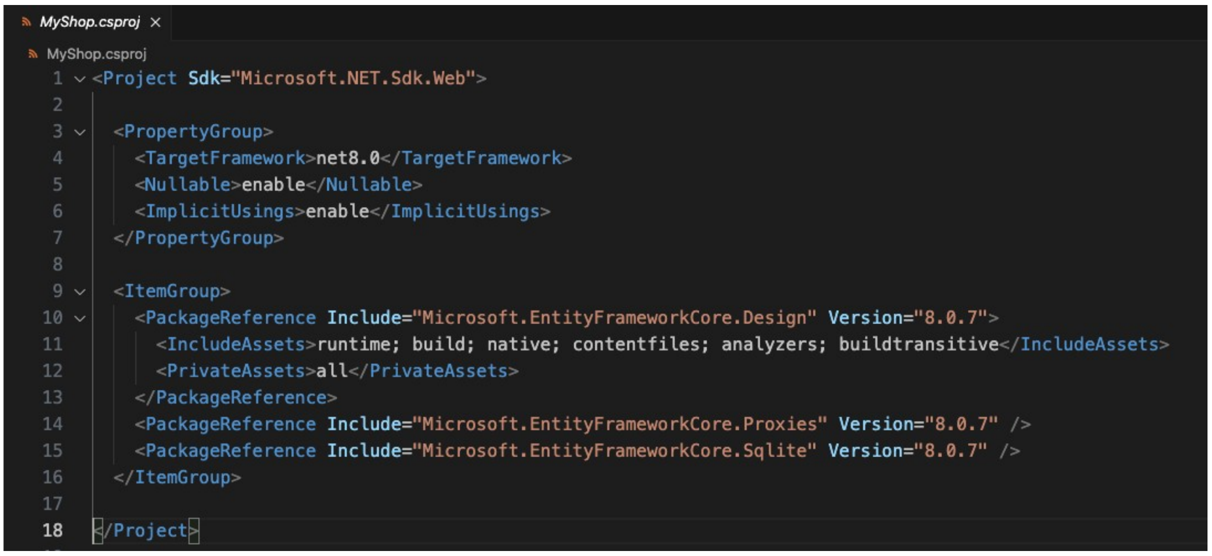


```
namespace MyShop.Models;

public class OrderItem
{
    public int OrderItemId { get; set; }
    public int ItemId { get; set; }
    //navigation property
    public virtual Item Item { get; set; } = default!;
    public int Quantity { get; set; }
    public int OrderId { get; set; }
    //navigation property
    public virtual Order Order { get; set; } = default!;
    public decimal OrderItemPrice { get; set; }
}
```

Then we install the NuGet package proxies by running:

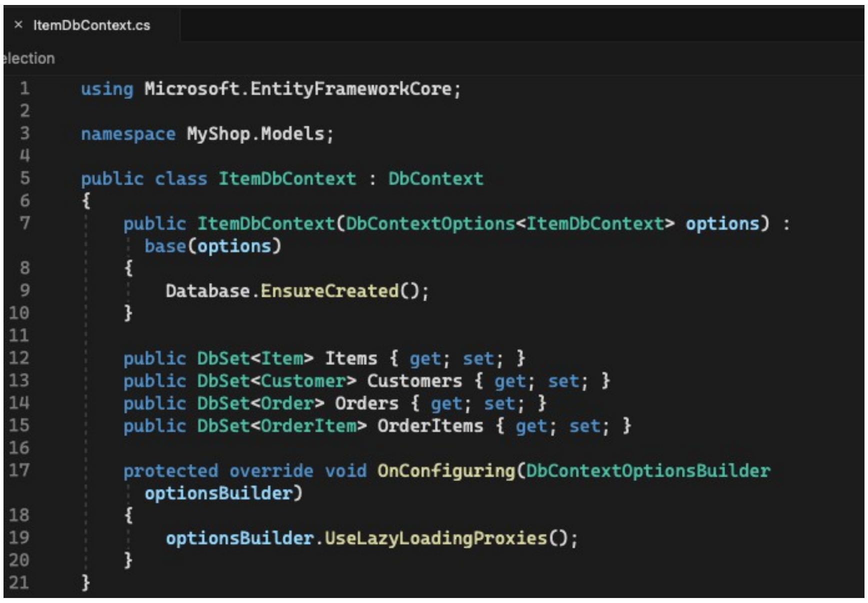
```
dotnet add package Microsoft.EntityFrameworkCore.Proxies --version="8.*"
```

This will appear in MyShop.csproj



```xml
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="8.0.7">
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
      <PrivateAssets>all</PrivateAssets>
    </PackageReference>
    <PackageReference Include="Microsoft.EntityFrameworkCore.Proxies" Version="8.0.7" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Sqlite" Version="8.0.7" />
  </ItemGroup>

</Project>
```

Now add the Line17 – Line19 to ItemDbContext to enable lazy loading (this requires the new NuGet package “proxies”).



```csharp
1      using Microsoft.EntityFrameworkCore;
2
3      namespace MyShop.Models;
4
5      public class ItemDbContext : DbContext
6      {
7          public ItemDbContext(DbContextOptions<ItemDbContext> options) :
             base(options)
8          {
9              Database.EnsureCreated();
10         }
11
12         public DbSet<Item> Items { get; set; }
13         public DbSet<Customer> Customers { get; set; }
14         public DbSet<Order> Orders { get; set; }
15         public DbSet<OrderItem> OrderItems { get; set; }
16
17         protected override void OnConfiguring(DbContextOptionsBuilder
             optionsBuilder)
18         {
19             optionsBuilder.UseLazyLoadingProxies();
20         }
21     }
```

By calling UseLazyLoadingProxies() in the OnConfiguring method, EF Core will generate proxies for your entity classes, allowing lazy loading of related entities when accessed.

With lazy loading enabled, when you access a navigation property like OrderItems of the Order class, EF Core will automatically load the related OrderItem entities from the database on-demand, without explicitly loading the entire tree of dependent objects connected by the navigation properties.

It's important to note that enabling lazy loading comes with certain considerations, such as the potential for increased database queries if navigation properties are accessed in a loop. Be cautious about using lazy loading in performance-critical scenarios and consider using eager loading (.Include) or explicit loading (DbContext.Entry) when you need to fetch related data in one go.

Read More:
https://learn.microsoft.com/en-us/ef/core/querying/related-data/

Now Run the project. Everything works! ☺ But nothing is to see! No worries, we will verify the proxies and add massive data via database initialisation in the next demo.