# Exercises: ASP.NET Core - Separation of Concerns, Expanding Database, Extending Models

Problems for exercises for the ["ASP.NET Core Advanced" course @ SoftUni](https://softuni.bg/trainings/4708/asp-net-advanced-october-2024)

A popcorn and film reels and a movie ticket

Description automatically generated with medium confidence

### Refactoring the CinemaWebApp with Layered Architecture

#### Introduction to Layered Architecture

In this chapter, we will **refactor** the **CinemaWebApp** by adopting a **layered architecture approach**. This approach separates the application into distinct **layers**, each **responsible for handling a specific concern**. By doing so, we improve **maintainability**, **scalability**, and **testability** of the application.

#### Benefits of Separation

* **Maintainability**

Easier to maintain and modify parts of the system without affecting others.

* **Testability**

You can easily write unit tests for services and controllers as they are decoupled from database logic.

* **Scalability**

The layered architecture makes it easier to scale and add new features.

A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated

#### The Layers

* **Common Layer**

Provides centralized management for shared constants, validation messages, and other reusable components across the project. This layer reduces redundancy and ensures that any changes are propagated throughout the entire application easily

* **Data Layer**

The Data layer in the architecture is responsible for handling database interactions, migrations, and configurations

* **Data.Models Layer**

Represents the core domain entities of the application. This layer defines the database structure and the relationships between different entities. By keeping this layer separate, we ensure that all the business logic is cleanly decoupled from the data definitions.

* **Web Layer**the topmost layer of the application, responsible for handling HTTP requests and presenting the UI to the user. It interacts with the **Service Layer** to process data and present it to the user via Razor views.
* **Web.Infrastructure Layer**  
  It contains extension methods that simplify the setup and configuration of services or middleware
* **Web.ViewModels Layer**The ViewModels layer serves as the intermediary between your application’s model layer and the user interface (UI). ViewModels shape how data is delivered to views and provide additional properties that may not exist directly in your domain models

#### Step-by-Step Guide to Creating a New Layer in Your ASP.NET Core Application

To create a new layer in your ASP.NET Core project, you will follow a process similar to creating a new class library project. Each layer will contain specific responsibilities and components.

##### Create a New Class Library Project

* Open your solution in **Visual Studio**
* Right-click the solution in **Solution Explorer** and select **Add** -> **New Project**
* Choose **Class Library** as the project template
* Name the new project based on the layer you want to create, for example:
  + **CinemaWebApp.Data** for the Data Access Layer
  + **CinemaWebApp.Services** for the Service Layer (Business Logic)
  + **CinemaWebApp.Common** for shared constants, utilities, or configurations
* Set the **Target Framework**
  + Ensure that the new class library targets the same .NET version as your main web application

A screenshot of a project

Description automatically generated

A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated

* **Add** Required **Folders and Files**
  + After creating the new layer, you can organize it by adding folders for better structure.

A screenshot of a computer

Description automatically generated

* **Add References Between Projects**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

#### Install the Required Packages

To set up the necessary packages for each project in your solution, you need to ensure that every layer has the correct dependencies.

* **CinemaWebApp.Common**
  + This layer is typically used for shared constants and validation logic, so it may not need any additional dependencies, but if you have any specific validation or utility libraries you need, you can install them
* **CinemaWebApp.Data -** This project handles **database context** and **migrations**
  + **Microsoft.EntityFrameworkCore**: To manage the interaction between the data model and the database
  + **Microsoft.EntityFrameworkCore.SqlServer**: If you're using SQL Server
  + **Microsoft.EntityFrameworkCore.Tools**: For migrations and database updates
* **CinemaWebApp.Data.Models**
  + **Microsoft.EntityFrameworkCore**: To ensure that your models can interact with the database
* **CinemaApp.Web.Infrastructure**
  + This layer can contain middleware or extensions. Common packages might include:
    - **Microsoft.AspNetCore.Http.Abstractions** - For HTTP context-related classes if you work with middleware
    - **Microsoft.Extensions.Logging** - If you are handling logging or diagnostics
* **CinemaWebApp.ViewModels**
  + For the ViewModels project, there might not be any specific package dependencies unless you are using specific validation or data transfer libraries

##### Restore Dependencies

After installing the necessary packages for each project, ensure you:

* **Right-click on the Solution** in Solution Explorer and choose **Restore NuGet Packages** to ensure all dependencies are restored and ready

##### Build the Solution

* Once all the packages are installed, **build the solution** to ensure everything is set up correctly

### Expanding the Database

In this section, we will expand the database by adding a new model called **Ticket**. The Ticket model represents the relationship between a cinema, a movie, and a user. It stores information about a user's ticket for a movie screening at a specific cinema, including the ticket price.

#### Implementing Ticket Model

The Ticket model will contain the following properties:

* **Id**
* **Price**
* **CinemaId**
* **MovieId**
* **UserId**

Here is how you can define the Ticket model in the Models folder:

A screenshot of a computer program

Description automatically generated

##### Extending Cinema and Movie Models

It is a good idea to **extend the Cinema and Movie models** to **include collections of Ticket entities**. This will ensure that the **relationships are fully navigable from both sides**, and it will also allow **easier access to related Tickets** for each Cinema and Movie

A computer code with text

Description automatically generated with medium confidence

A screenshot of a computer program

Description automatically generated

##### Extenting IdentityUser

The **IdentityUser class does not necessarily need a collection of tickets** because it already serves as a general-purpose user entity in ASP.NET Core Identity and is not directly tailored to your application’s specific data needs.

However, if you frequently need to access all tickets associated with a particular user, you could create a custom application user class that extends IdentityUser and adds a collection of tickets. This way, you can access user-related tickets more conveniently.

##### Creating a Custom ApplicationUser with a Collection of Tickets

In **ASP.NET Core**, the **IdentityUser class is used by default** to represent users in the application. However, to add **custom properties**, like a collection of tickets, we need to **extend this class**. We'll create a new class called **ApplicationUser** that **inherits from IdentityUser**. This allows us to:

* **Add custom properties** that are unique to our application's needs
* **Set up relationships** with other entities, like Tickets

A close-up of a computer screen

Description automatically generated

##### Working with Application User

* We should modify the Ticket class to **associate tickets with ApplicationUser A screenshot of a computer program

  Description automatically generated**
* Update the **UserMovie** Class   
  A screenshot of a computer program

  Description automatically generated

##### Updating Program.cs to Use ApplicationUser

Now that we have created the **ApplicationUser** class, we need to ensure that the rest of our **application is configured to use it instead of the default IdentityUser**. This includes **updating the Program.cs** file to make sure Identity works with our **custom user class and our database.**

A computer screen shot of a code

Description automatically generated

##### Updating the AppDbContext

* When we add the **Ticket** entity and **modify the AppDbContext** to **use the ApplicationUser** class **instead of the default IdentityUser**, we need to **update both the inheritance of AppDbContext and the Fluent API configuration** in the OnModelCreating method.
  + We need add to our **AppDbContext** a **new DbSet<Ticket>** for tickets so that **Entity Framework knows about the new entity**.
  + Adding **Fluent API Configuration** for the Ticket Entity

A screenshot of a computer program

Description automatically generated

A computer screen shot of a program

Description automatically generated

#### Creating and Applying a Migration

Now that we have **updated our AppDbContext** to use **ApplicationUser** and have **added the Ticket entity** with its **relationships**, the next step is to **update the database**. In ASP.NET Core, we do this by **creating and   
applying a migration**.

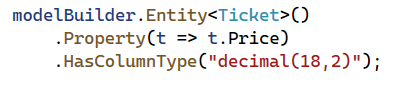
A screenshot of a computer

Description automatically generated

* **Step-by-Step Guide to Fix the Decimal Precision Warning**

A screenshot of a computer code

Description automatically generated

* + Modify the OnModelCreating Method  
    
  + Apply the Migration

A red and white text

Description automatically generated

##### Migration Issues

If you're **running into issues with migrations**, or just **want to ensure Entity Framework can create the DbContext during design time**, we will create the **AppDbContextFactory**

* Create **AppDbContextFactory.cs**
* Required Packages:
  + **Microsoft.Extensions.Configuration**
  + **Microsoft.Extensions.Configuration.Json**
  + **Microsoft.Extensions.Configuration.FileExtensions**

**A screenshot of a computer program

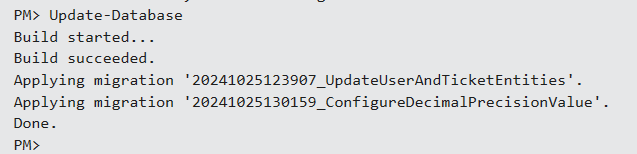
Description automatically generated**

**A screenshot of a computer

Description automatically generated**

##### Update-database

The next step is to **apply the migration to update the database schema**. This will create the necessary tables and relationships based on your models.



A screenshot of a computer

Description automatically generated

**Testing the Application**

* **Run the Application**
* **Verify Basic Functionality**
* **Check for Data Consistency**
  + **Look for Data Changes in the Database**
* **Test User Features**

**A screenshot of a computer

Description automatically generated**

* **The error shown in the screenshot is caused by the fact that your Razor view or controller is trying to use UserManager<IdentityUser>, but your application is configured to use ApplicationUser as the identity model**
* **You need to replace all instances where UserManager<IdentityUser> and SignInManager<IdentityUser> are used with UserManager<ApplicationUser> and SignInManager<ApplicationUser>.**

**A screenshot of a computer program

Description automatically generated**

* **If you are injecting UserManager<IdentityUser> or SignInManager<IdentityUser> in your controllers, make sure to replace them with UserManager<ApplicationUser> and SignInManager<ApplicationUser>**

A screenshot of a computer program

Description automatically generated

##### Editing the Register and Login Models

The **RegisterModel** handles the user registration process in our application. By default, it references IdentityUser, so we need to update it to use our ApplicationUser.

A screenshot of a computer

Description automatically generated

Similarly, we need to update the **Login Model** and **Logout Model** to use ApplicationUser instead of IdentityUser

A screenshot of a computer

Description automatically generated

A white background with text

Description automatically generated

**By following these steps, you have ensured that the RegisterModel, LoginModel, and other Identity-related models work with ApplicationUser instead of IdentityUser.**

**REBUILD THE PROJECT**

### Repository Pattern

The **Repository Pattern** is a design pattern that abstracts the data access layer from the business logic of an application. It acts as **an intermediary between the data source** (such as a database) **and the business logic layer**, providing a standardized way to perform operations like fetching, adding, updating, and deleting data.

By using the repository pattern, the code for data access is centralized, reusable, and easier to maintain. It also helps **decouple the data access logic from the rest of the application**, making it more flexible and testable, as the underlying data source can be swapped out (e.g., from a database to an API) without affecting the higher-level code.

#### Implement the Generic Repository

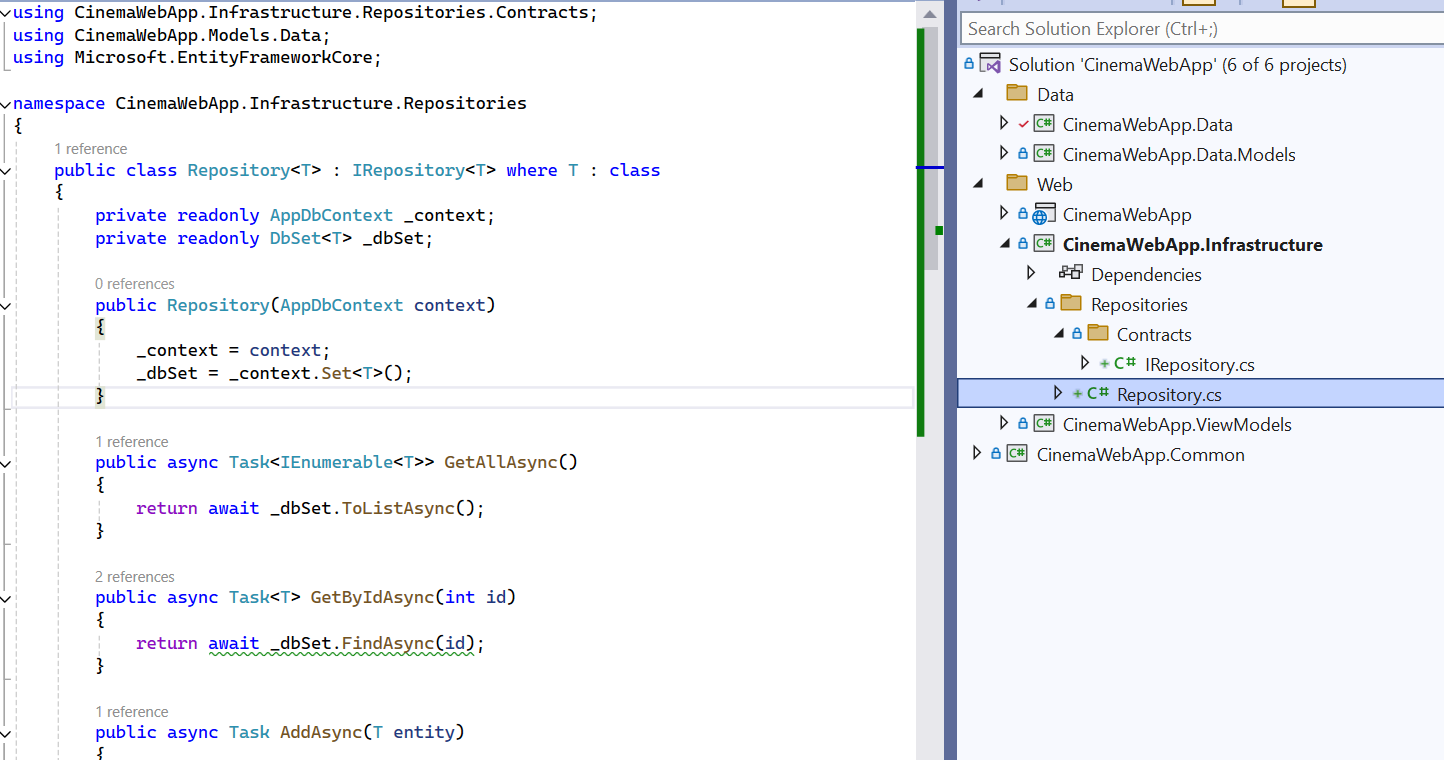
We'll create a **base repository class** that **implements the IRepository<T> interface**. This class will contain the common logic for interacting with the database **via AppDbContext.**

In your **CinemaWebApp** project, you should implement the **repository pattern** in the **Infrastructure** layer. This is where data access logic typically resides in a layered or clean architecture.

#### Structure for Repositories

* **CinemaWebApp.Infrastructure** -> **Repositories** (folder)(subfolder for interfaces - **Contracts**)
  + **IRepository<T>.cs** (Generic interface)  
    A screenshot of a computer program

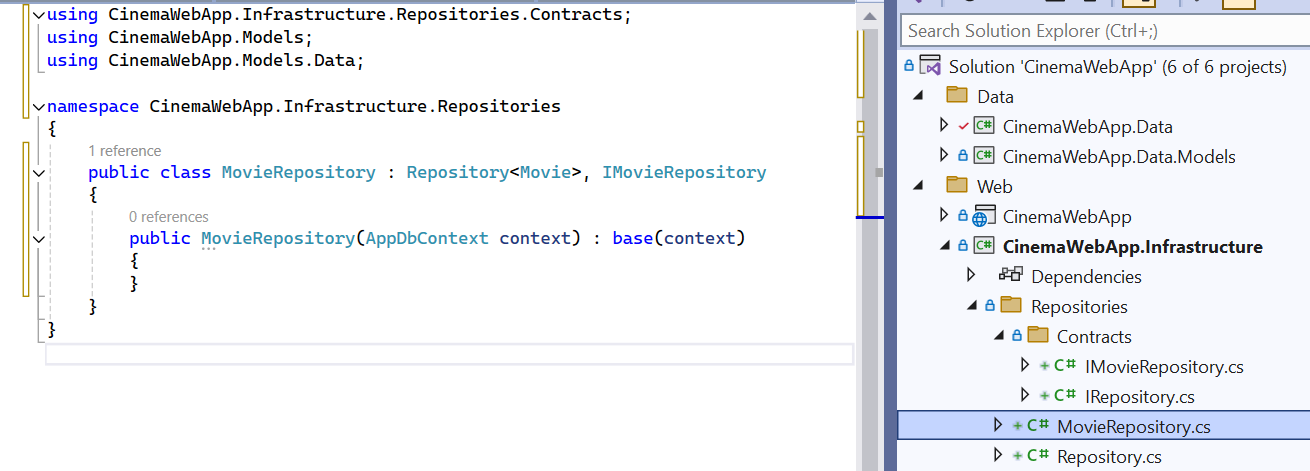
    Description automatically generated  
    
  + **Repository<T>.cs** (Generic implementation)


#### Create Specific Repositories for Each Entity

A screenshot of a computer

Description automatically generated



**Repeat** for other entities as needed - **CinemaRepository**, **TicketRepository**

#### Register Repositories in Dependency Injection

We have the **generic** and **specific repositories**, they **need to be registered in the DI container** to be accessible throughout the application.

* **Open Program.cs** in the **CinemaWebApp** project
* Add the following registrations for the repositories  
  A screen shot of a computer code

  Description automatically generated

### Services

#### Create Service Interfaces and Implementations

**Define interfaces** for your services, then **implement these interfaces in classes** that handle the business logic and interact with repositories.

#### Register the Service in Dependency Injection

**Register the services in the DI container** (inside **Program.cs**) so that they can be injected into controllers or other parts of the application.

#### Use the Service in Controllers

**Inject the service into your controllers** and **use it to interact with the business logic** instead of directly interacting with repositories or **AppDbContext**