# 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

## Implementing the Repository Pattern

To make our application more maintainable and scalable, we will now apply the **Repository Pattern**. This **design pattern** allows us to **separate the logic that retrieves data from the database from the business logic** that acts on that data. It also helps **avoid repeating the same code when performing basic database operations**.

In this section, we will implement a **Generic Repository** that **can be used with any entity in the application**.

### What is a Generic Repository?

A Generic Repository is a reusable class that provides basic **CRUD** (Create, Read, Update, Delete) operations for any type of entity. Instead of writing the same logic for every entity, we write it once and reuse it everywhere.

### Creating the Interface IRepository<TType,TId>

We begin by defining a simple and reusable interface that declares the **most commonly used operations** for any entity in our application.

A screenshot of a computer

AI-generated content may be incorrect.

We will now define a reusable interface that declares all the **essential operations** for working with any entity in our application. This interface includes basic CRUD operations and commonly used query methods.



This interface is designed to cover **all common needs** for working with a database:

|  |  |
| --- | --- |
| Method | Description |
| GetById(TId id) | Synchronously fetches a single entity by ID |
| GetByIdAsync(TId id) | Asynchronously fetches a single entity by ID |
| FirstOrDefault(predicate) | Finds the first match using a synchronous filter |
| FirstOrDefaultAsync(predicate) | Finds the first match using a LINQ expression (queryable in EF Core) |
| FindByConditionsAsync(predicate) | Similar to FirstOrDefaultAsync but used when filtering logic is more complex |
| GetAll() | Returns all entities as a list (sync) |
| GetAllAsync() | Returns all entities asynchronously |
| GetAllAttached() | Returns a queryable collection that is tracked by EF Core |
| Add(item) | Adds a new entity (sync) |
| AddAsync(item) | Adds a new entity asynchronously |
| AddRange(items) | Adds multiple entities (sync) |
| AddRangeAsync(items) | Adds multiple entities asynchronously |
| Delete(entity) | Removes a single entity (sync) |
| DeleteAsync(entity) | Removes a single entity asynchronously |
| Update(item) | Marks the entity as modified (sync) |
| UpdateAsync(item) | Marks the entity as modified asynchronously |
| SaveChangesAsync() | Commits all pending changes to the database |

### Implementing the Base Repository Class

Now that we have defined our IRepository<TType, TId> interface, we will create a class that implements it. This class will provide actual logic for querying the database using **Entity Framework Core**.

The base repository will be **generic**, which means it can work with any entity and any ID type.

In your CinemaApp.Data project, create a new file in the **Repository folder** and name it BaseRepository.cs

A computer screen with text

AI-generated content may be incorrect.

Let’s understand what’s happening inside this class:

* CinemaAppDbContext context – our database connection and tracking mechanism.
* DbSet<TType> dbSet – gives access to the correct table in the database depending on which entity we are working with.
* Every method defined in the interface is now implemented.



A screenshot of a computer program

AI-generated content may be incorrect.

### Creating Custom Entity Repositories (Movie and Watchlist)

Although our BaseRepository<TType, TId> provides **generic CRUD operations for all entities**, in many cases we need to implement **custom logic** specific to an entity (e.g. filtering movies, loading watchlist by user, etc.).

To follow this pattern, we create **entity-specific repositories** like MovieRepository and WatchlistRepository. These repositories will inherit from the base repository and allow us to add any movie-specific or watchlist-specific logic later.

A screenshot of a computer screen

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

Now that we’ve defined the custom repository interfaces for Movie and UserMovie, we will implement the actual classes that provide the functionality.

A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

### Registering Custom Repositories in the Dependency Injection Container

Now that we have created the MovieRepository and WatchlistRepository classes and their corresponding interfaces, we need to **register them in the DI container** so that they can be used by services or controllers.

This step connects the interface (what we depend on) to the concrete implementation (what actually performs the work).

Open the Program.cs file located in the **CinemaApp.Web** project.

Inside the builder.Services configuration block, locate the section where services are registered and add the following code:

A screen shot of a computer

AI-generated content may be incorrect.

### Refactoring the Services to Use Repositories

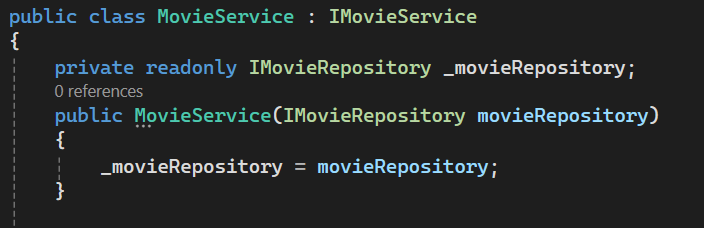
Until now, our services may have worked directly with the CinemaAppDbContext. In this step, we will refactor our MovieService and WatchlistService to follow **clean architecture principles** by working through the repositories we just implemented.

#### Refactoring MovieService to Use the IMovieRepository

Refactor the MovieService class so that it no longer works directly with the CinemaAppDbContext. Instead, it will use the IMovieRepository, which encapsulates all database access logic.

A screen shot of a computer program

AI-generated content may be incorrect.



#### Refactoring WatchlistService to Use the IWatchlistRepository

Refactor the WatchlistService class so that it no longer works directly with the CinemaAppDbContext. Instead, it will use the IWatchlistRepository, which encapsulates all database access logic.

A screen shot of a computer

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

#### Updating Method Bodies to Use Repositories

After replacing the constructors in both MovieService and WatchlistService to use the injected repositories (IMovieRepository and IWatchlistRepository), the next step is to update each method inside these services to **replace direct usage of the DbContext** with repository calls.

* Example: Refactoring AddToWatchlistAsync

This is how the method originally worked with the DbContext directly:

A screen shot of a computer program

AI-generated content may be incorrect.

Now we update the method to work with the injected repository field

A screen shot of a computer program

AI-generated content may be incorrect.

With this change, your method is now fully aligned with the repository pattern. You no longer depend directly on **CinemaAppDbContext**, which improves testability and structure.

Now that you've seen how to refactor individual methods like AddToWatchlistAsync in WatchlistService, your final task in this section is to **complete the refactoring of all remaining methods** in both services.

## Expanding the Database – Adding Missing Models

As we continue to build the CinemaApp project, it’s time to **expand our database schema** to support more advanced features that will be implemented later in the course.

Until now, our project has been focused on:

* Movies
* Watchlists
* Basic user interactions

What’s Missing?

To fully implement **all the functionalities expected** in this project, we will need to track:

* **Cinemas** – different locations where movies are shown
* **Tickets** – users purchasing tickets for a movie in a cinema

What Should We Do Next:

1. Create all **missing model classes** in the CinemaApp.Data.Models folder
2. Define **relationships** between them (one-to-many, many-to-many)
3. Update the DbContext (CinemaAppDbContext.cs) to include them
4. Create and apply a new **EF Core migration** to expand our database schema

### Creating the Cinema Model

The Cinema class represents a movie theater that offers one or more movie projections. It includes basic information such as name, location, and deletion status. It also manages relationships to the movies it plays and the tickets it issues:

Cinema

* Has Id – a Guid**, Primary Key**
* Has Name – a string with **min** length **2** and **max** length **80** (**required**)
* Has Location – string with **min** length **2** and **max** length **50** (**required**)
* Has IsDeleted – bool **(default value == false)**
* Has **CinameMovies** – a collection of type **CinemaMovie**
* Has Tickets – a collection of type Ticket

### Creating the CinemaMovie Model

The CinemaMovie class represents the **many-to-many relationship** between a Cinema and a Movie, enriched with additional properties like available tickets and showtime settings.

CinemaMovie

* Has MovieId – a Guid, foreign key (required)
* Has Movie – Movie (required)
* Has CinemaId – a Guid, foreign key (required)
* Has Cinema – Cinema (required)
* Has AvailableTickets – an int (required)
* Has IsDeleted – a bool (default: false)
* Has Showtimes – a string with column type varchar(5) (**default: "00000"**)

### Creating the Ticket Model

The Ticket entity represents a purchased movie ticket. It connects the user, the cinema, and the movie for which the ticket is valid.

Ticket

* Has Id – a unique Guid, **Primary Key**
* Has Price – a decimal (required)
* Has CinemaId – a Guid, foreign key (required)
* Has Cinema – Cinema (required)
* Has MovieId – a Guid, foreign key (required)
* Has Movie – Movie (required)
* Has UserId – a string, foreign key (required)
* Has User – IdentityUser (required)

### Creating the UserTicket Model

The UserTicket entity serves as a **mapping table** between users and tickets. It is particularly useful when a user can hold multiple tickets and each ticket can be linked to a user with additional metadata (if needed in future).

UserTicket

* Has UserId – a string, **Primary Key**, **Foreign Key** (required)
* Has User – IdentityUser (required)
* Has TicketId – a Guid, **Primary Key**, **Foreign Key** (required)
* Has Ticket – Ticket (required)

## Configuring the Models in the Application

### Add DbSet Properties

A screen shot of a computer program

AI-generated content may be incorrect.

### Configure Composite Keys and Relations with Fluent API

#### Create Configuration Classes

A computer screen shot of a program

AI-generated content may be incorrect.

A computer screen shot of a program

AI-generated content may be incorrect.

A computer screen shot of a program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

### Updating the Movie Entity – Adding Collection Properties

Now that we have created the CinemaMovie and Ticket entities, it’s time to update the Movie class to reflect its relationships.

A screenshot of a computer program

AI-generated content may be incorrect.

### Creating and Applying the Migration

Now that we’ve created and **configured all the new entity models**, it’s time to **update our database schema**.

A screen shot of a computer

AI-generated content may be incorrect.

