Assignment 1  
Entities & Repositories

In this assignment series you will build out a small forum app. For each assignment you will add something new to the project, and at the end of the semester you will have a fully working forum app. Hopefully. It is very much inspired by Reddit.

In the end it will consist of a simple CRUD focused Web API, with Entity Framework Core and a SQLite database to store data. For the front-end you will have a Blazor web app.

In this first assignment you will create the entities for your domain model, and you will define repository interfaces (repository is explained later).

Because your app will evolve over time - different parts will be added and swapped out - we need to design the system with modularity in mind. We do this by creating multiple projects, each responsible for something specific. It will be a simplified monolithic application.

The Web API (i.e. server) consists of two layers: network and a persistence. Often you will have a business logic layer in between, but we are skipping that and simplifying the server to focus on the .NET tools rather than sound architecture design. You should probably have this extra layer in your semester project.

There is some initial setup, which is best done on one computer, then shared to others through GitHub. I recommend reading through the entire document before actually starting on anything.

Let’s begin.

# Features

This assignment is open-ended, meaning we provide you with a few minimum requirements, which *must* be completed. We also have suggestions on how to expand upon the system, should you wish to. Or you can come up with your own ideas.

### Feature description

We need a User, having at least a username and a password. It needs an Id of type int.

We need a Post. It’s written by a User. It contains a Title and a Body. It also needs an Id of type int.

A User can also write a Comment on a Post. A Comment contains a Body, and an Id of type int.

All entities must have an Id of type int.

The way we create relationships between the Entities is described in detail further below. In short, we use foreign keys, rather than associations.

### Optional features

Further feature suggestions, if you are brave:

* A user can like/dislike a post.
* A user can like/dislike a comment.
* A user can create a subforum, meaning a post now belongs to a specific subforum.
* A user can comment on a comment.
* A user becomes moderator of a subforum they created, meaning they can delete comments and posts.
* Feel free to add further ideas yourself.

# Create Solution

First, you need a new, fresh solution. You may work in groups, so only one group member does the initial setup.

Create a new Solution in Rider:

A computer screen shot of a computer screen

Description automatically generated

And then:

1. Select Empty Solution
2. Give your solution a name
3. Place it somewhere
4. Create a new directory, this is probably checked already
5. Check this to create a git repository. You must have your assignment on GitHub, and hand in a link to your repository. It’s easier if this is checked initially, then you get some extra setup prepared for you, like a git ignore file.

A screenshot of a computer

Description automatically generated

Now you should have a new, fresh, empty solution.

# GitHub

You must use Git and GitHub for your assignments. You may use whatever branching strategy, you like. Each hand-in on itslearning is typically a link to your main branch.

A good start is to now commit your \*.sln file. E.g.: DnpAssignmentXYZ.sln. This file ”binds” your entire application together.

# Domain Model

Based on the above requirements, you must create a domain model diagram, where we can see:

* The entities of the system
* The properties (attributes) on the entities
* The relationships between entities, e.g. Post is written by a User. Remember multiplicities at both ends, like you were taught for the Entity Relationship Diagram in DBS, or the Domain Model in SWE.

# Entities

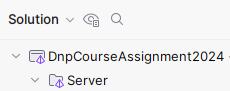
It’s time to implement the domain model, i.e. the entities as classes. These will just be data container classes, using properties.

First, create a new Solution Folder, call it Server. This folder will contain all server related code:

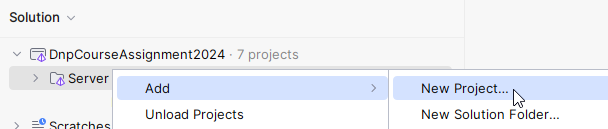
A screenshot of a computer

Description automatically generated

Now, your solution looks like this:

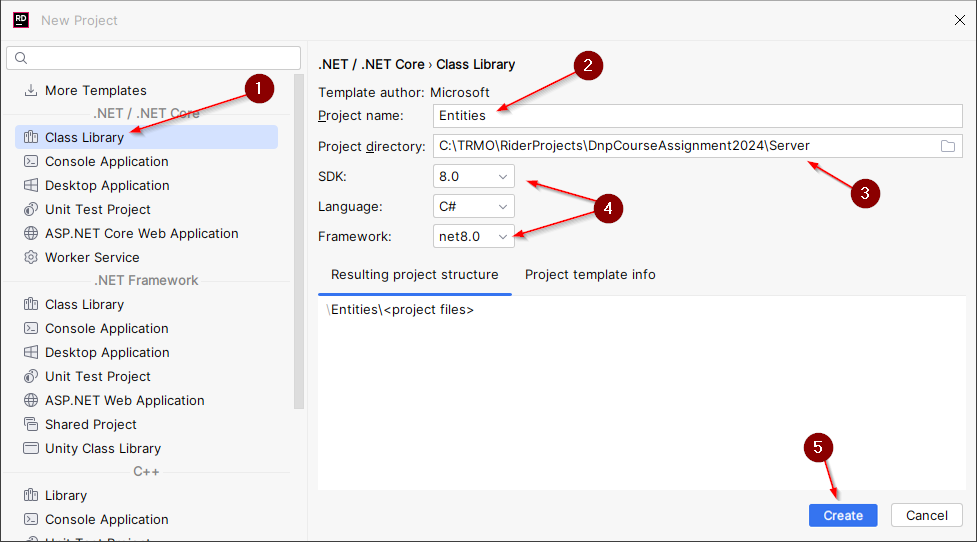


Then, create a new class library to contain your domain entities, inside the Server folder:

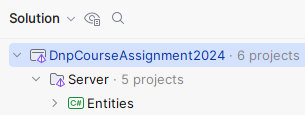


Then, setup the project as follows:

1. Select Class Library
2. Give the project a name
3. Specify the directory, i.e. put this project into Server
4. Select SDK and Framework, if available. If you only have one version installed, these are probably greyed out, or not there at all.
5. Select Create

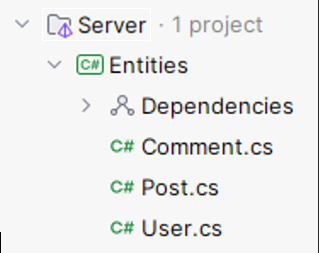


Now your solution looks like this:



You can safely delete the Class1.cs file inside Entities.

Now, you can create your entities, e.g. Post, User, etc, in this project:



Remember, all your classes should be under git version control. But you must also tell git about your projects.

This is done by adding and committing the \*.csproj file for each project.

Initially, a new project will have its name in red font, meaning git does not know about it. E.g.:

A screenshot of a computer

Description automatically generated

You must then find the \*.csproj file and add it. First, change the view from ”Solution” to ”File system”:

A screenshot of a computer

Description automatically generated

Then add the .csproj file in the project to git:

A screenshot of a computer

Description automatically generated

Do this for your Entities library project.

### Entity Relationships Modelling

You are used to model relationships with associations, e.g. a Post has many Comments. Then the Post would have a List of Comments.

We don’t initially do this. It comes later in the course.

DON’T DO ASSOCIATIONS NOW

We use the approach of a relational database, and model our entity-relationships using foreign keys.

This means a Comment will have a property called PostId, which references the post that the Comment belongs to.

A Post might have a SubForumId, which references the SubForum it belongs to, if you make this feature.

A Post will also have a UserId property, referencing the User, which wrote the post. Similar for the Comment, and so on.

Here is what your Post might initially look like. We use auto-properties instead of field variables! This is important!

A screenshot of a computer code

Description automatically generated

You may add constructor(s) if you find this needed. It will be later, and it will then be explained how to do it. Or you can just use the object initializer approach.

# Implement your Entities

Implement all your entities now.

# Repositories

Repository pattern is similar to Data Access Pattern, if you are familiar with that.

A Repository is an abstraction of data-access for a single entity. This means a Repository is an interface, and the implementation manages how to persist a specific entity-type. We will initially use a List (i.e. no actual persistence), then a file, and then a database, as we progress through the course.

The Repository interface comes in different shapes, but it will generally provide some standard methods, usually like Create, Read, Update, Delete (i.e. CRUD for short).

The Command Line Interface (assignment 2) and Web API (assignment 4) will use a Repository to save incoming data, or extract requested data, etc.

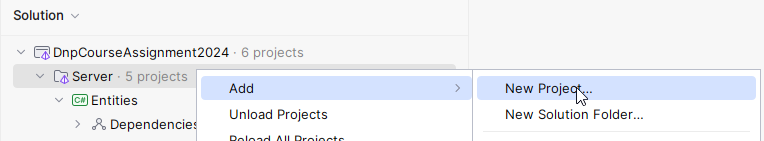
Both the ”presentation/network” layer, and the persistence layer will change over time, but the Repository-interfaces remain the same. This means they must live in their own respective project. It becomes similar to the layered architecture you have seen on second semester, and each project in this case becomes a layer.

# Repository Contracts Project

The Repositories live on the server side, and is only used on the server side.

We need a project to contain the Repository interfaces, aka ”contracts”. The actual implementations will go elsewhere, later.

Create a new Class Library Project, first:



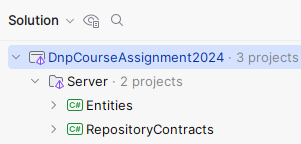
And then setup the project:

1. Select type
2. Input project name
3. Update the path, so the project is located in a ”Server” folder.
4. Select versions
5. Select to create

A screenshot of a project

Description automatically generated

And finally the new solution structure:



You can, again, delete Class1.cs. And put the RepositoryContracts.csproj file under version control, like before with the Entities project.

### Add Reference

The new RepositoryContracts project need to work with your entities. These are located in a separate project, and so we need to add a reference from RepositoryContracts to Entities.

Right click on the ”Dependencies” inside RepositoryContracts.

A screenshot of a computer

Description automatically generated

And then select the project, you want to referece, in our case the Entities:

A screenshot of a computer

Description automatically generated

Finally, select [Add].

***Note***: In the image above, it says <Shared>. In yours it says <Server>.

### Create Repository Interfaces

In this project you put Repository interfaces. One for each entity. And you will have at least 3 entities: Post, User, Comment. Maybe more, if you decide to expand your application with more features.

Our repository interfaces will define 5 standard methods:

1. Add – Used to store a new entity
2. Update – Used to update an existing entity.
3. Delete – Delete an existing entity.
4. GetSingle – Get a single entity.
5. GetMany – Get multiple entities.

Here is an example for the IPostRepository interface. The others look almost identical:

public interface IPostRepository  
{  
 Task<Post> AddAsync(Post post);  
 Task UpdateAsync(Post post);  
 Task DeleteAsync(int id);  
 Task<Post> GetSingleAsync(int id);  
 IQueryable<Post> GetMany();  
}

For now, ignore the Task return types, and the ”Async” method name suffixes. This will be explained later.

Method descriptions:

**Add** takes a Post and returns the created Post. This is because the server sets some data on the Post, e.g. the ID, and this should be returned to the client. The client might need this ID for something. This is common.

**Update** takes a Post (with ID) and replaces the existing Post. If no existing Post is found, an exception is thrown to indicate the error.

**Delete** will remove the Post with a given ID. If no matching Post is found, an exception is thrown.

**GetSingle** will return the Post matching the given ID. If no Post is found, an exception is thrown.

**GetMany** will return an IQueryable. This is an interface which can looped over in a for-each loop to extract the relevant entities. Or we can use LINQ, which we will see later in the course.   
This makes filtering the Post by some criteria easier (using predicates, later).   
Maybe we want to fetch all Posts with a specific substring in the title. Or some other property on the Post.  
The method is not async, the reason of which we will get back to, when we add a database.

You will define such an I\*Repository interface per entity!

### What is Task and Async?

We will come back to this in session 2. It’s asynchronous programming. It allows for different tasks/jobs/pieces of code to execute concurrently.

It requires us to return a Task, sometimes wrapping the actual value, e.g. Task<Post>. This is a Task which will eventually return a Post. We can then extract the Post from the Task, when the Task is finished.

By convention asynchronous methods have their method names suffixed with ”Async”, to indicate they are used differently from normal methods.

Again, we will come back to this later.

# In-memory Repository Implementations

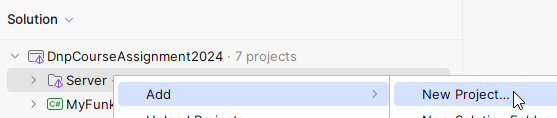
We will create initial implementations for the repository interfaces. We start by just keeping the data in a list, i.e. nothing is actually saved to disk or database, this comes later. But it lets us create a ”complete” system by assignment 2, which we can actually play around with.

### New project

Again, we need a new project. We are going to use his approach a lot, creating new projects to organize our code by technical responsibility.

The project goes into the Server solution folder, it is a class library and is named InMemoryRepositories.

Create a new class library like this, by right-clicking on the Server folder:



And then:

1. Select Class Library
2. Choose project name. Meaningful names are important.
3. Make sure it goes into the Server folder
4. Pick versions, if possible.
5. Select Create.

A screenshot of a computer

Description automatically generated

Delete Class1.cs.

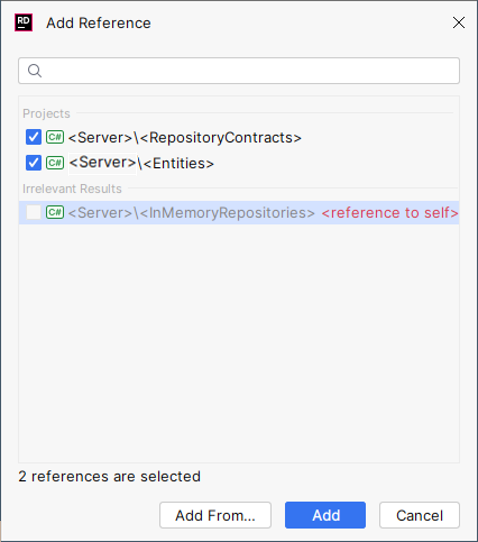
Put the project under version control, as previously.

### Dependency

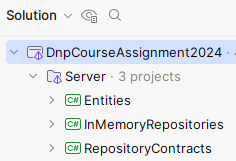
You need to add a dependency from InMemoryRepositories project to the Entities project, as you have done before.

This is so your repository implementations can know about the entities they manage.

You will also need a dependency from InMemoryRepositories project to the RepositoryContracts project, because this is where the interfaces are located:



Now your solution structure looks like this:



### Implementations

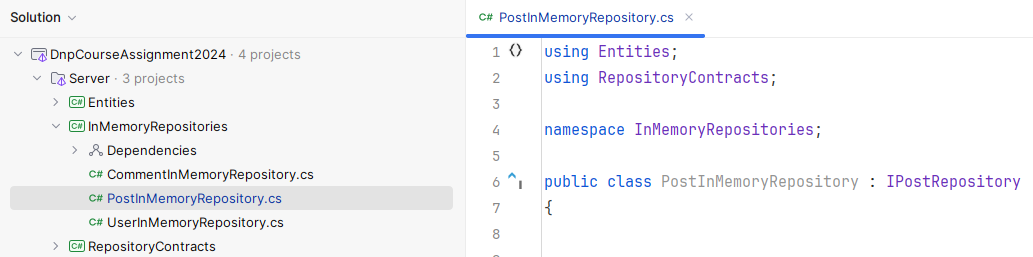
You will inside your new project create an implementation per repository interface.

The repository implementation just uses a list to store entities in.

Below is a description of the PostInMemoryRepository, step by step. You will then repeat the process for the other repository implementations. You could easily divide the work amongst group members here.

#### Post Repository class

First, you need a class, which implements the IPostRepository:



Notice the location of the class, and the class definition in the file.

#### List to contain the entities

You must add a field variable of type List<Post>. I have called it ”posts”, you can call it whatever makes sense to you. This will contain our entities.

#### Add method

This is the AddAsync method:

public Task<Post> AddAsync(Post post)  
{  
 post.Id = posts.Any()   
 ? posts.Max(p => p.Id) + 1  
 : 1;  
 posts.Add(post);  
 return Task.FromResult(post);  
}

It takes a Post as parameter, and returns a Post (inside a Task).

We first set the Id of the post. This is done by finding the current maximum Id of all the posts, and incrementing that value. Better and smarter approaches exist, but we will eventually use a database to generate the Id. I use a [ternary operator](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/conditional-operator) (click me) to check if there are any posts in the list, and if so, use the current maximum Id plus 1. Otherwise just use Id 1. It’s a short-hand if-else.

The Post is added to the list, i.e. the field variable you just created.

This method is not doing anything *asynchronous*. This will come later. So, to return a Task containing a Post, we have to explicitly wrap our post into a Task. That happens in the last line.

#### Update method

This is the update method:

public Task UpdateAsync(Post post)  
{  
 Post? existingPost = posts.SingleOrDefault(p => p.Id == post.Id);  
 if (existingPost is null)  
 {  
 throw new InvalidOperationException(

$"Post with ID '{post.Id}' not found");  
 }  
  
 posts.Remove(existingPost);  
 posts.Add(post);  
  
 return Task.CompletedTask;  
}

It receives a post, and returns nothing, i.e. a Task not containing an object.

First, we look for an existing post with the Id of the incoming argument-post. The SingleOrDefault method will loop through the posts-list, and find a post which matches the predicate.  
If none is found, null is returned. As indicated by the variable type of ”Post?”. I.e. the question mark indicates the variable ”existingPost” may be null.

We check if the existingPost is null, in which case there is nothing to update, and an exception is thrown. This exception can then be handled elsewhere, and an error can be returned to the client.  
Alternatively, you could create a new Post. That’s up to you, if you want that.

The easiest way to overwrite an existing post is just to delete the existing, and insert a new, with the same Id. So that is what we do.

In the end, we have to return a Task, but it does not contain anything, so we just return a ”completed task”.

#### Delete method

Here is the delete method:

public Task DeleteAsync(int id)  
{  
 Post? postToRemove = posts.SingleOrDefault(p => p.Id == id);  
 if (postToRemove is null)  
 {  
 throw new InvalidOperationException(  
 $"Post with ID '{id}' not found");  
 }  
  
 posts.Remove(postToRemove);  
 return Task.CompletedTask;  
}

This time we just need an Id to remove a Post, so that is the parameter.

Again, we look for an existing post. If none is found, we throw an exception.

Then the found post is removed from the list.

And a completed task is returned at the end.

#### GetSingle

Here is *part of* the method to return a single entity by Id:

public Task<Post> GetSingleAsync(int id)  
{  
 // Do implementation

return Task.FromResult(post);  
}

This time, you do the implementation yourself. It is very similar to the Delete method. You must look for a post. If none exist, throw an exception.

At the end the post is returned, wrapped in a Task.

#### GetMany

Finally, we need to be able to get many Posts, and apply some filtering, or ordering:

public IQueryable<Post> GetManyAsync()  
{  
 return posts.AsQueryable();  
}

A list, or most types of collections, can be converted to an IQueryable. This is an interface, which just provides the option to loop through it, e.g. with a for-each loop. Or LINQ, which we will get back to.

So, we convert the list of posts to an IQueryable<Post>.

### Other repositories

You must now implement your other repositories. They will look *very* similar!

You might even consider some optimization, so you don’t have so much duplicate code. If you are brave.

## Optional: Initial dummy data

Each of your repositories could create some initial dummy data, just so you have a few posts, users, comments, etc, when the application starts. Next time it will be easier to test things, when you have this initial data.

You can e.g. create this data in each of the Repository implementation constructors. You may benefit from having a specific method for this, which the constructor could call.

# Formalities

You may work on this assignment in groups.

You must have your assignment on GitHub.

You will hand in a link to your *GitHub repository* on itslearning.

Deadline can be found on itslearning.