Assignment 2
Command Line Interface

In this assignment you will expand your application with a Command Line Interface (aka **CLI**). This is a simple text based interface in your console/terminal. You read commands from the terminal, and you output data to the terminal.

You will create some kind of functioning user interface to support the requirements specified last time.

Must -have requirements:

- Create new user (user name, password, etc)
- Create new post (title, body, user id)
- Add comment to existing post (body, user id, post id)
- View posts overview (just display [title, id] for each post)
- View specific post (see title and body, and comments on the post)

You have some freedom in how you structure the output, and how you read the input.

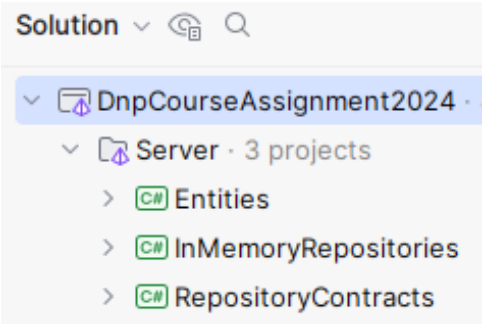
You are of course welcome to add more functionality to your CLI application:

- Manage users:
  - Create new.
  - Update existing.
  - Delete user.
  - See all users.
- Manage posts:
  - Create new.
  - Update existing.
  - Delete post.
  - See overview of posts, e.g. just id and title.
  - View single post.
- ”CRUD” operations on the other entities.
- When viewing a list of some entity, consider adding filtering options:
  - See all posts by a specific user id.
  - See all comments a specific user has made.
  - See all users with some specific word in their username.
  - ...
- See previous assignment for other examples.

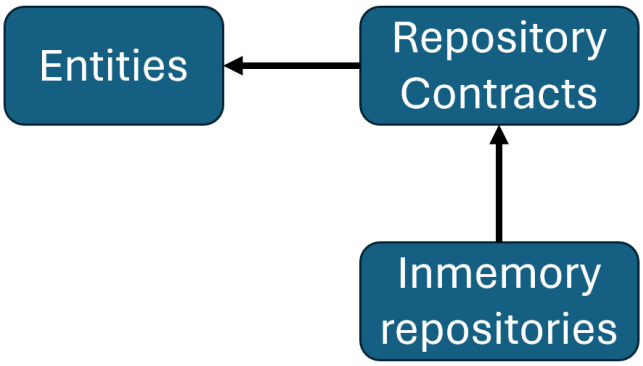
Status

Last time you implemented the domain model, repository interfaces, and some simple in-memory repository implementations, which just used a list to keep track of the entities.

Your application (hopefully) looks like this:



Or as a *component diagram*, we can show it like this, each box is a project:



The arrows indicate dependencies between projects.

The repository interfaces, in the RepositoryContracts project, manage the Entities. And your repository implementations implement the repository interfaces.

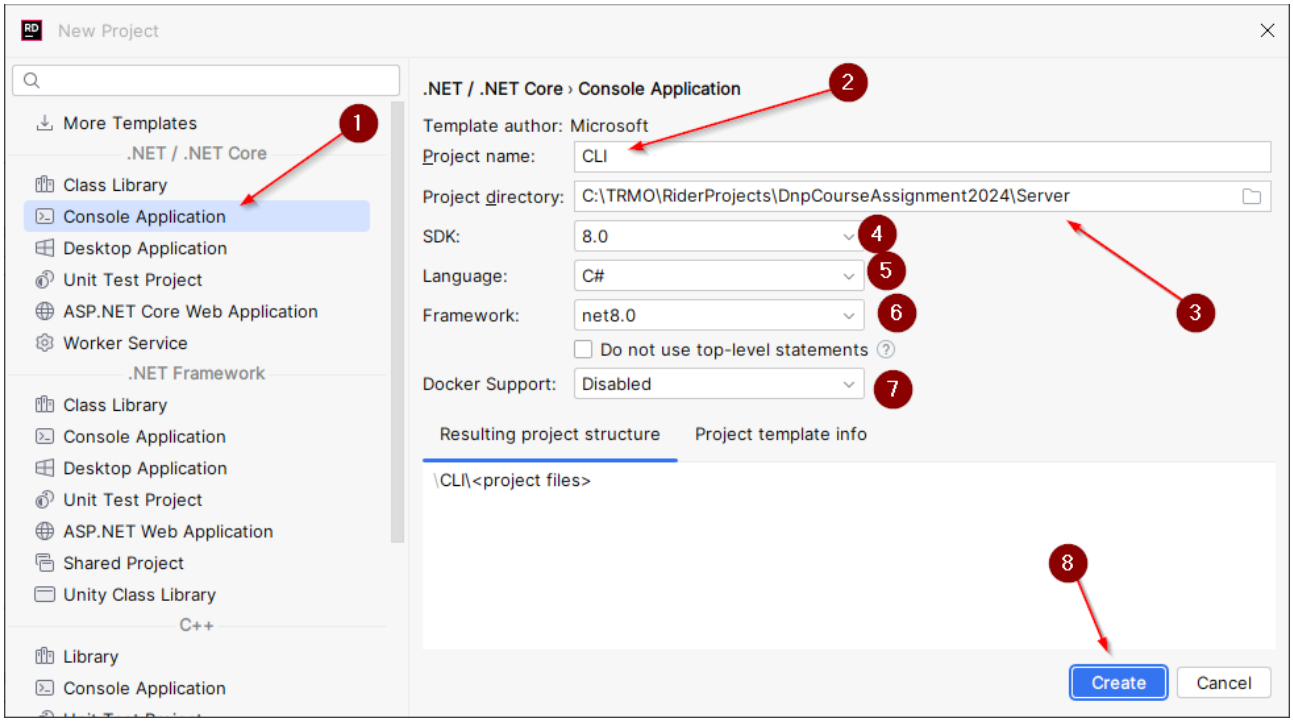
Setup

First, we need a new project for the Command Line Interface.

Create a Console Application project, on the Server side:

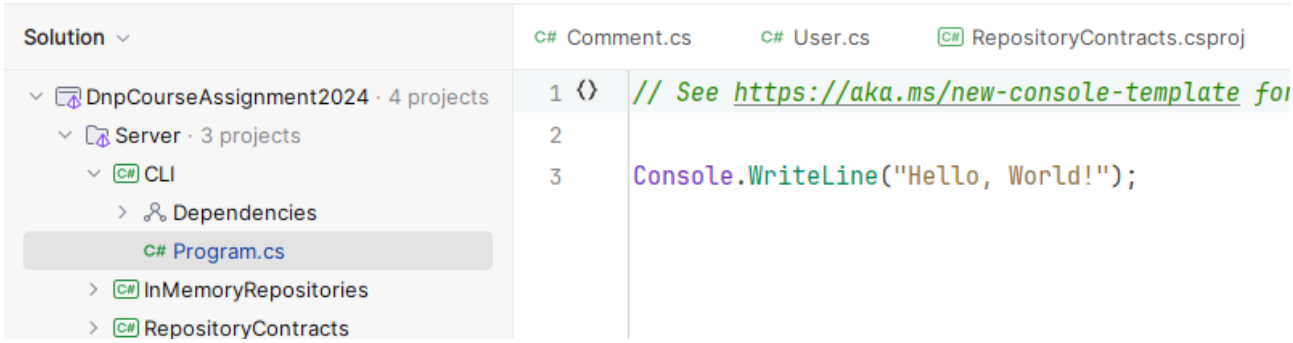
1) Select Console Application. This will be a runnable project, i.e. it has a main method (though implicit), so we can actually start the application.
2) Give the project a name.
3) Make sure the project is located in the Server folder
4) Pick SDK, if possible.
5) Language is obviously C#.
6) And latest framework.

7) We don’t need Docker.
8) And create the project.

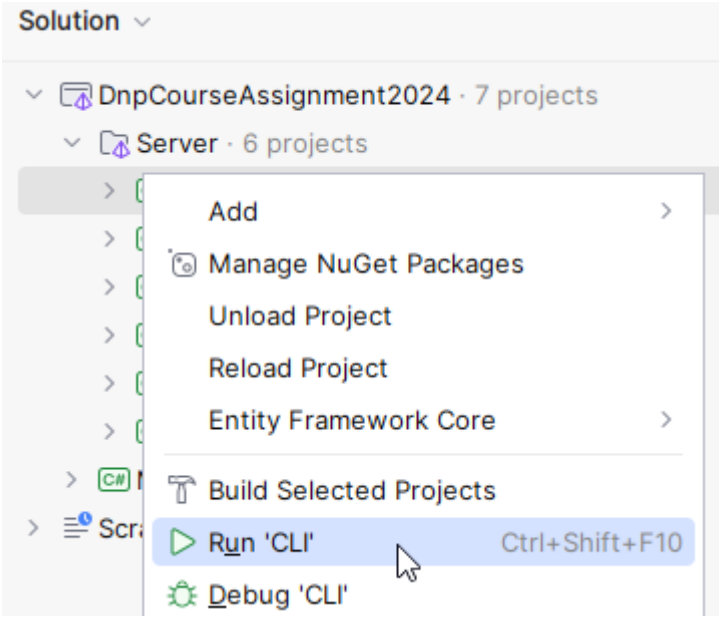


You then get this. There’s the CLI project, with a Program.cs class, which has an ”implicit” main-method. I.e. there is no method at all.

You should just think of the method signature as invisible, and the file contains the actual method body.



You can run this main method by right-clicking on the CLI project:



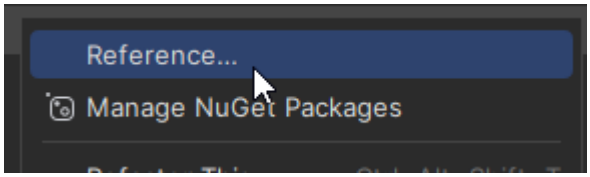
Give it a go, and verify ”Hello, World!” is printed to the console.

Dependencies

Your new CLI project needs to have dependencies to:

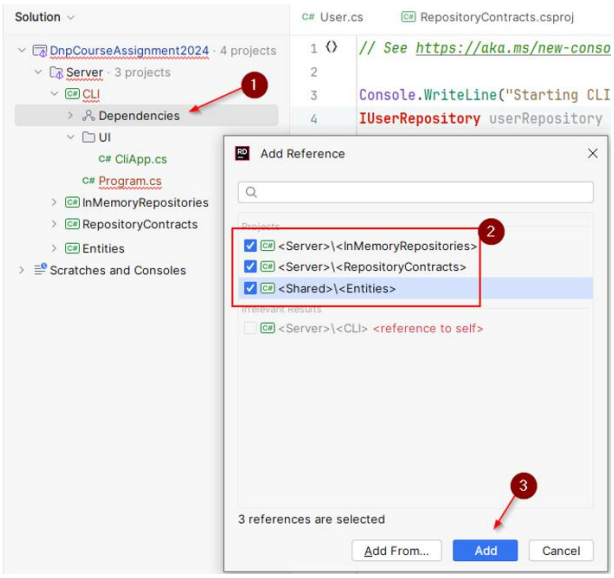
- Entities
- RepositoryContracts
- InMemoryRepositories

(1) Right click, and select ”reference”

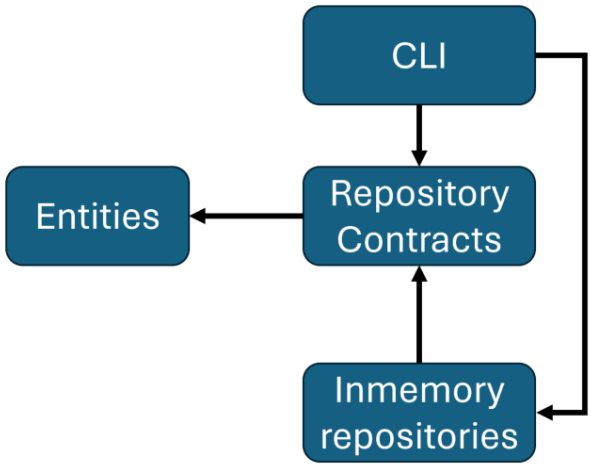


(2) Select projects to reference.

(3) Click add



Now the component diagram looks like this:



Dependencies are *transient* (remember DBS?). For example in the above we have the following dependencies:

RepositoryContracts -> Entities

CLI -> RepositoryContracts

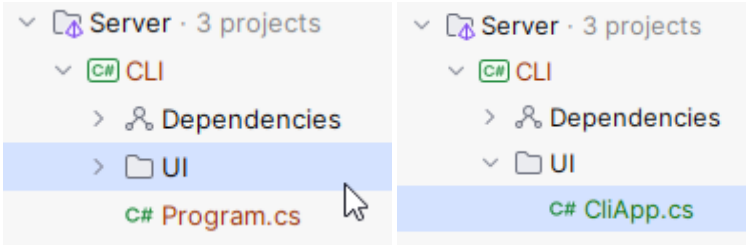
Which also means:

CLI -> Entities

UI class

We probably don't want all the UI logic in the Program file. The Program file is just used to create an instance of your UI class, and pass in repository instances. And then "start" the UI.

So, Create a new directory, call it e.g. "UI". And inside this directory, create the class which will manage your CLI UI. Something like this:



Program.cs

This class is now supposed to just create the necessary classes, and start the UI.

It could look something like this:

The main point is:

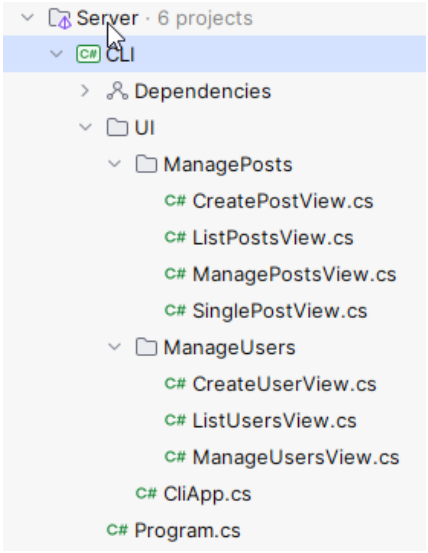
- Program.cs initially instantiates whatever needs to be created. Probably primarily repositories.
- They are passed to the CliApp.
- Then the CliApp is started. This call is await'ed. When you start using asynchronous programming, your entire app is quickly infiltrated with "Async" and Tasks. In the app, eventually an async method on a repository is called, and async methods can only be called by other async methods. "It's turtles all the way down", as they say.

You may rework this structure as you see fit. I strongly recommend splitting your UI logic into separate classes, and then you could potentially instead pass the repositories to the specific classes. This will make parallel team-work easier, too.

You could consider which "views" are necessary, and create a class responsible for a specific view.

The overall organization of your classes is left to you.

Here is my initial attempt, which can currently just create a new User and view a list of Users:



The CliApp may create a new ManageUserView instance. This is then responsible for instantiating CreateUserView and ListUsersView, when needed.

Implement CLI application

You must implement the CLI application, so that it fulfills the stated minimum requirements. You must print to the console, and read from the console, as needed.

Dependency injection

You ***MUST!*** ***only*** instantiate the repositories in the Program.cs class! This is the only place, where you have "IPostRepository postRepo = new InMemoryPostRepository();".

You then pass this variable, "postRepo", around as needed, through the constructors of other UI classes. If you have a class to handle creation of a new Post, this class should receive an IPostRepository as an argument in its constructor.

Otherwise, your classes might use different repository instances, meaning if you create a post and later want to view it, this may not be possible. If all UI classes share the same instance, it will be the same list of entities you have access to.

It will also be easier in assignment 3 to swap out the repository implementations.

This is my Program.cs file, notice instantiation of repositories:

```
using CLI.UI;
using InMemoryRepositories;
using RepositoryContracts;

Console.WriteLine("Starting CLI app...");
IUserRepository userRepository = new UserInMemoryRepository();
ICommentRepository commentRepository = new CommentInMemoryRepository();
IPostRepository postRepository = new PostInMemoryRepository();

CliApp cliApp = new CliApp(userRepository, commentRepository, postRepository);
await cliApp.StartAsync();
```

And this, for example, is part of my UI class handling creation of posts:

```
public class CreatePostView
{
    private readonly IPostRepository postRepository;

    1 usage  TRMO
    public CreatePostView(IPostRepository postRepository)
    {
        this.postRepository = postRepository;
    }
```

The repository is *injected* through the constructor. Notice the field variable type is the interface, ***not*** the concrete implementation.

With this approach future updates to your application is much simpler. This is the Dependency Inversion Principle from SOLID, in action.

Next assignment, you will implement new versions of your repositories, and you will (hopefully) only have to update the Program.cs class of your CLI app, in order to swap out implementations.

If you have done it correctly.

Optional: Business logic

Consider implementing some minimal business logic:

- Is the user input correct?
- If adding a comment to a post by a user, does both post and user Ids actually exist?
- When creating a user, is the username already taken?
- Etc.

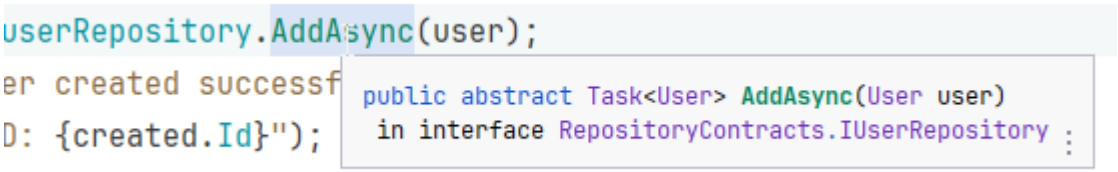
Asynchronous programming

As mentioned previously, once you start using asynchronous programming, most of your methods quickly become asynchronous. All the way up to the main method.

The general approach is, whenever you call an asynchronous method, the calling method will await the call. Here's an example. We are adding a User to the IUserRepository, this is a call to an asynchronous method:

```
private async Task AddUserAsync(string name, string password)
{
    // …
    User created = await userRepository.AddAsync(user);
    // …
}
```

We can identify the asynchronous method by the suffixed "Async". Or that it returns a Task:



We put `await` in front of the call. This will yield the code execution here, until the Task is finished, and then extract the contained object within the Task. In this case the User.

In my case, I print out the generated Id of the new User (this part is not shown here).

When we want to use the await keyword, we must make the containing method async too, and make it return Task instead of void, or return a Task<Something> instead of returning just Something.

So, my AddUserAsync method is marked as async, returning a Task.

This escalates upward, so the method calling AddUserAsync must also await the method call, and that method must itself be async. And so on. All the way out to the main method. Turtles all the way down. Or up. Depending on where you view it from.

Initial dummy data

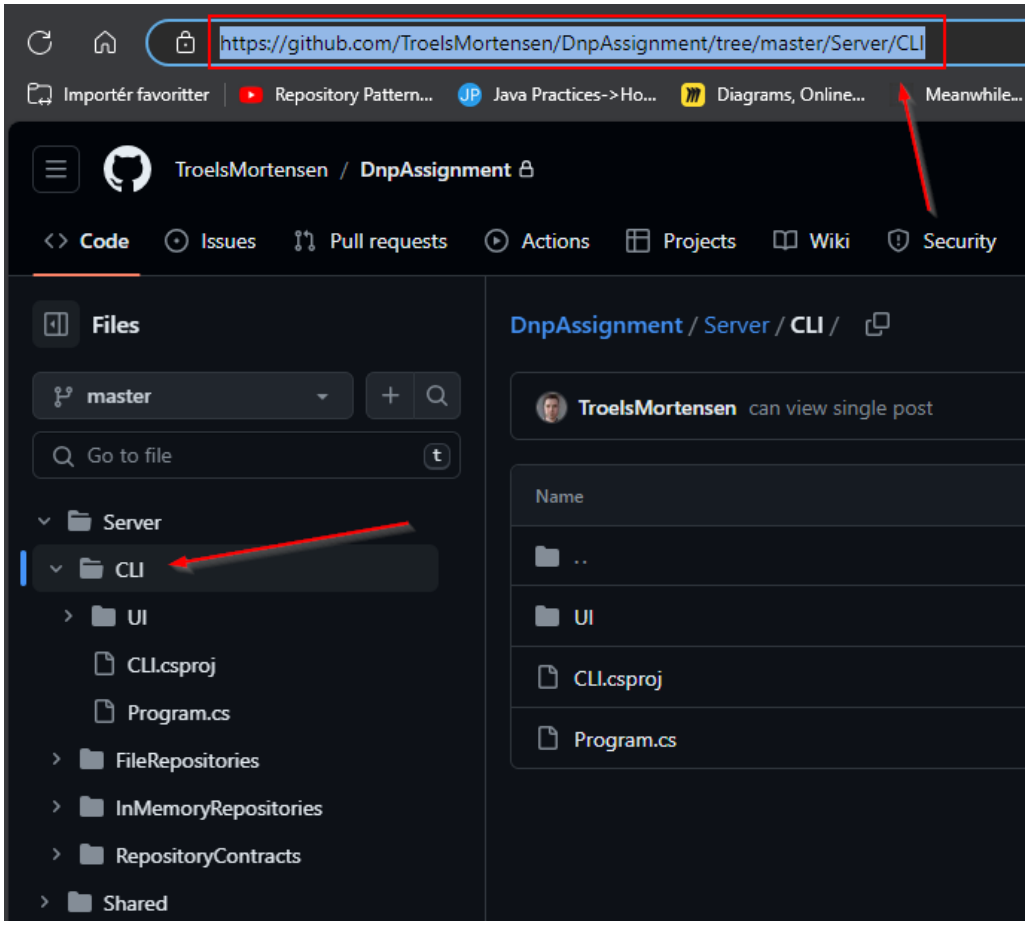
Each of your repositories must create some initial dummy data, say 3-5 entities, just so you have a few posts, users, comments, etc, when the application starts. It will be easier to test things, when you have this initial data.

Formalities

You may work on this assignment in groups.

You must have your assignment on github.

You will hand in a link to the new part on your GitHub repository on itslearning. That means on GitHub you navigate into your CLI project, and copy the link from there:



Deadline can be found on itslearning.