Simple Bank App Documentation

Overview

The simple bank app is, as its names suggests, a simple app that emulates in the most basic way possible the two main actions that can be done with a bank account: deposit and withdraw money. The purpose of the app is to demonstrate the thought process behind designing and implementing a web API.

Use Cases

The app will have two main modules, authentication and bank account. Each module has its own use cases. For the purposes of this application, the authentication module is not included in this documentation since authentication is one of the most common modules in any application. Though it is important to consider is that all the bank account use cases will only work for logged in users

The main uses cases this app covers are:

Create a bank account

The user can create one or many bank accounts with a starting balance. He/she can only create bank accounts that belong to himself/herself.

Deposit In a bank account

The user can deposit money in any of his accounts.

Withdraw from a bank account

The user can withdraw money from any of his accounts. He/she cannot withdraw more money than there is available in each account.

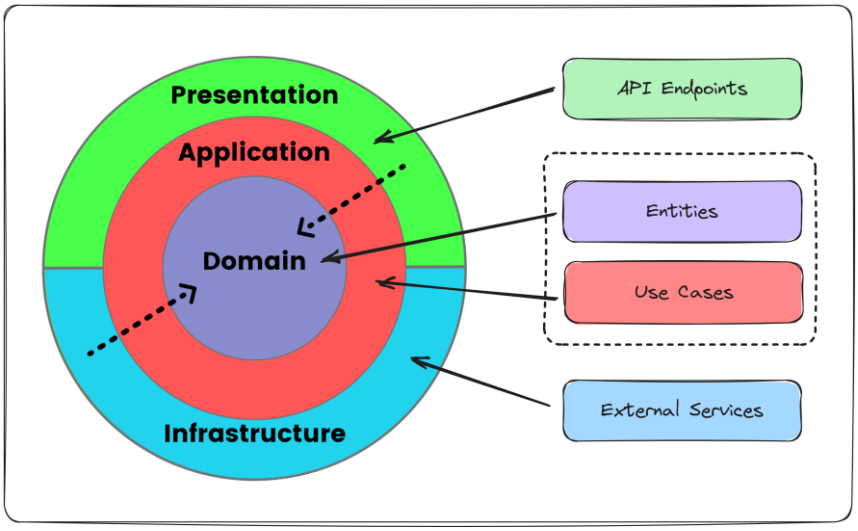
Delete a bank account

The user can delete any of his bank accounts. He/she can only delete an account without any available funds. If he/she wants to delete an account that has money in it, he/she first needs to withdraw all the money.

Design

Clean Architecture

The app was designed following the clean architecture principles. It is divided in the following 4 layers:



The main goal of this architecture is to achieve separation of concerns, with each layer having a distinct responsibility and making sure they are as loosely couple as possible. This design facilitates testing by isolating the core logic from external dependencies, it makes the system more scalable and more maintainable in the long run.

Test Driven Development

The uses cases described before where developed using TDD, which helps write reliable code by writing the tests first. This approach helps the developer to have a clearer picture of the functionality before writing the actual implementation, which prevents bugs and results in more maintainable and continuously tested code.

Now, writing tests is an art. Each test should have the right balance between flexibility and functionality. Writing too much tests becomes a problem because the cost of modifying the application increases, but writing few tests translates into a higher risk of bugs.

In my opinion, a good way to start writing tests is by identifying the critical logic of our app. This is the section of our that is essential for the business. In this particular app, depositing and withdrawing money is the most important action a user can perform, so they definitely need to have one or more unit tests; and they should check for edge cases, like the user trying to withdraw more money than it has available in an account, or the user trying to manipulate accounts that do not belong to him. Creating and deleting accounts is also important for the business so they should have their tests as well.

Finally, the tests should also validate that the app returns the correct errors. The more we are clear about the errors that can occur in an application, the more stable the application is.

CQRS

In modern apps, usually read requests happen more frequently than write requests. And, usually, read requests require different models to return the data in the various ways a user may need it. For this reason, it makes sense to logically separate the read actions from the write actions. This is the idea behind CQRS. This application implements these principles using the idea of commands and queries. Commands are actions that manipulate the data in some manner, and queries are actions that only retrieve information. At the code level, each command/query has its own independent implementation, facilitating testing and isolation from other parts of the system. This is especially useful for the use cases that are critical for the business. Now, obviously CQRS is not a silver bullet that solves everything. There are certain scenarios where a most traditional approaches, like using services, is not only enough but it is even better than CQRS. In this application, the authentication module is implemented using services for two main reasons: It is a module that we can be relatively certain that it won't be change that much in the future, and it's not so easy to categorize the login action. It is technically a query but it can also be a command, since usually logging in involves saving data in the db and changing the user or the application state. For these scenarios, where a particular action could be both a command or a query, its probably not the best idea to use CQRS. But for other scenarios, like the bank account use cases, it makes total sense.

Database

For this app, I decided to use a relation database (SQL Server). The design is simple, we only have two tables: User and BankAccount. User can have zero or many bank accounts.

Implementation

Architecture Overview

A .csproj was created per each layer. One of the main goals of the clean architecture approach is to keep things well organized and improve readability. Ideally, a new developer should be able to get a good idea of the application use cases just by looking at the folder and file names. Because of this, in the application layer, for example, there are two folders: Authentication and BankAccount. Each folder contains the logic for these modules. This gives us the advantage that if a new dev wants to understand the BankAccount logic in detail, he can just go to that folder and he will find all the use cases clearly defined. This way he can easily know where and how to add new code.

The same idea applies for the other layers. In the presentation layer, since there is not any core logic implemented but rather the purpose of the layer is to receive requests, the contracts that define those requests are separated in Authentication and BankAccount folders (and each module has its own controller). So similarly to the application layer, just by having a quick look at it, a new dev that sees the code for the first time can quickly start getting a clear picture of where to add new code.

Finally, other things like validators, errors, mappings, etc. are also organized in folders. There is a common folder as well in each project for shared files between modules. Again, the idea is to be as organized as possible. Having a lot of folders may seem unnecessary but, in the long run, it helps to easily scale a system.

Dependency Injection

In order to follow, as best as possible, the clean architecture principle of separation of concerns, each layer has its own dependency injection file. This way, each layer is responsible of its dependencies. This prevents the Program.cs file from becoming too big as the project grows.

Error Handling

There are many ways to handle errors in an application. Personally, I prefer the approach of writing custom errors rather than custom exceptions. Custom errors allow us to better standardize the error response strategy and it keeps a clear separation between controlled errors and unhandled exceptions. A key part of having a stable system is making sure we are able to quickly identify and fix unhandled exceptions.

For this reason, I used a library called ErrorOr for error handling. This library allows a method to return either whatever the method needs to return, or a list of customizable errors. Each method is then responsible of identifying the scenarios where an error should be returned.

I'm also taking advantage of the Problems() object response of the .NET controllers. This provides a quick and easy way to correctly return the metadata of an error. Finally, unhandled exceptions are re-routed to a /error endpoint, where it can be logged to a file or a db table specific for unhandled errors.

Validations

Is it crucial to make sure the data that we receive in each endpoint has the correct format and the necessary values. For this reason, I decided to use FluentValidations library, which is a library that provides a clear and easy way of writing validations. Also, its always preferable to keep the validations logic separated from the model, since its way more scalable. This makes using a library like FluentValidations a better approach than built-in features like data annotations.

Mapping

Another key part of any system is object mapping which is basically converting an object from one class to another. In architectures like Clean Architecture it becomes even more important, since a way to making sure the separation of concerns principle is enforced across layers is making sure each layer has its own models. This usually will cause multiple models with duplicated attributes, but having everything as loosely couple as possible is always worth it.

Mapster is the library I decided to use for mapping. AutoMapper is usually the go to library for this, but Mapster is 4 times faster. Mapster works with config files to define the mapping strategies. In the app, each layer manages its own mapping config files and add them in their own dependency injection file. Again, each layer should be as loosely couple from each other as possible.

ORM

One of the restrictions of this project was not to use Entity Framework or Dapper, which are usually the way to go when choosing an ORM for .NET. For this reason, I decided to use Linq2DB, which is a light and easy to use ORM. It has the downside that it auto generates a file with the entity classes, making it so the Domain models (which should have the entity definition according to the clean architecture and domain driven design principles) have to be mapped to this generated classes and vice versa. Also, mocking the dbContext of linq2Db for testing purposes required and extra interface (ICommonRepository), which just added more unnecessary code.

For this reason, generally speaking, for a relational database, Entity Framework or Dapper should be the way to go.

Test Driven Development

One really important thing when writing tests is the naming convention. Personally, I think that naming the tests like this: Class_Scenario_ExpectedReturn is the best approach. This is the naming convention that can be found in this app. This is easy to understand and easy to follow.

Another important thing is how the tests are organized. For this app I decided to create just one test project and then subdivided each layer in folders, but it's probably better to have one test project per layer, to keep in sync with the separation of concerns principle. On the other hand, in my opinion, it's important to separate the test files per use case. The idea is that a new developer should be able to just go and read the unit tests and get a good understanding of all the use cases of the app, and what are the edge cases and most common errors for each use case; and this can only be achieved if each use case has individual test files.

Obviously, this cannot be achieved perfectly, specially with the infrastructure layer, since the repositories tend to be shared by multiple use cases. But at least the application tests should be organized like this, that way the organization of the tests matches the organization of the code itself, making it easier to know where to add the code to fix a bug when a test fails.

Docker

Docker was implemented using a DockerFile to dockerize the api and a docker-compose file to run the api and the db server simultaneously. Keep in mind that every time the docker-compose file is run the db is re-created. The ports for both the api and the db are configurable in the docker-compose file.

Considerations

- As mentioned at the beginning of this document, the authentication module details are not included in this documentation. TDD was not followed when implementing the register and the login, mostly for time limitations. In a real development environment, those features should be developed using TDD.
- There is no GET endpoint in the app. This is only because of time limitations. The repository method to get an account was developed, but I couldn't add the whole endpoint.

- Another restriction for the application was not to use the MediatR library. Again because of time limitations, I was not able to implement an alternative library to implement the mediator pattern. Based on my research, MediatR appears to be the best library out there for this purpose.
  For this reason, the command handlers are injected individually in the BankAccountController, which is not a good practice when implementing Command/Queries with handlers.

If there is any question about this application, please do not hesitate to contact me:
ron.cg95@gmail.com