Crypto Avenue
An Interactive Platform for Crypto Trading

Candidat: Cezar BUNA

Coordinator: Prof. Razvan CIOARGA

Session: June 2024

1. Introduction

1.1 Context

CryptoAvenue is an interactive web application designed to assist users in managing their cryptocurrency portfolios. Following the rise of Bitcoin in 2009 along with blockchain technology, the financial landscape has seen a significant transformation. Cryptocurrencies are decentralized solutions to traditional banking that offer transparency, security and complete control over one's own money, without the involvement of banks. This project represents the final project of my Computer Science bachelor's degree and it reflects my passion for cryptocurrency and blockchain, which started ever since I first discovered the exciting world of blockchain technology and cryptocurrencies.

1.2 Motivation

I first became enthusiastic about cryptocurrencies and blockchain technology when I first discovered them, back in 2015. Decentralized finance is a pretty big game changer in the financial industry that has fascinated me for a long time and made me want to dive deeper into it. The motivation behind this project is to provide the crypto community with a very simple, easy-to-use tool that eliminates the trouble of managing your cryptocurrency investments and to help people make clever investment decisions. The idea behind CryptoAvenue is to solve all these problems through an intuitive UI that aids the user in tracking their portfolio's performance over time and making decisions based on real data.

The driving force behind CryptoAvenue is personal. Part of its growth is driven by an understanding of what could happen if cryptocurrencies, and blockchain in general, were adopted on a much larger scale. The world of digital currencies is still at a very early stage, with the notion of helpful tools such as this one not even existing just a few years ago. This platform works to simplify the landscape of cryptocurrency in general, while making it easily accessible to most of the population, which will lead to a higher adoption rate among citizens of the world and a global adoption of digital currencies.

1.3 Problem Statement

Because of the high volatility nature of the cryptocurrency market, investors find it challenging to make well-informed decisions on what they are going to invest their hard-earned money in. The abundance of assets and the fact that they are dispersed across multiple platforms makes it a challenge for users to manage their diversified portfolios. By providing a simple, user-friendly platform that retrieves real-time data from various trustworthy sources (API’s) and gives users a very detailed overview of their assets, CryptoAvenue seeks to solve the problems above.

Furthermore, the application uses AI in order to provide users with custom recommendations and advice, therefore assisting them in making optimal decisions when choosing how to invest their money. Additionally, the application also allows users to revert previous transactions that they executed, in case something went wrong.

2. Theoretical presentation

2.1 Overview

My app, CryptoAvenue, is a fullstack desktop application which uses an array of different technologies for the front-end, back-end, as well as the database side, each one serving a different purpose in the scope of the application.

The **back-end** , which represents the back-bone of the application, is built using the .NET Core framework, which is a cross-plaform framework used by developers for creating and maintaining modern, high-performance applications, and written in **C#,** the primary language used with .NET Core. C# is known for its safety, performance and various number of libraries and extensions. The back-end has been written and compiled using Microsoft’s Visual Studio, an integrated development environment (IDE), which is a fantastic tool for debugging and developing applications, especially those written in C#, which is the main language the IDE was supposed to support. Developers at Microsoft regularly maintain and improve Visual Studio IDE and the .NET Core framework, which makes it a lot easier for developers to do their work as fast and optimally as possible.

The **data-base**, where all the data which the application uses is stored is managed and maintained through Microsoft’s SQL Server Management Studio. SSMS is a powerful tool for managing SQL Server databases. It provides an easy-to-use interface for writing queries, designing databases and managing the data processed by the application. SQL Server is a reliable platform for data storage. It supports structure query language(SQL) by allowing for custom, complex queries and efficient data retrieval. This is crucial for handling large volumes of data used by big-scale applications, such as crypto-trading applications, because they include information like historical data of different cryptocurrencies, transactions and market information.

The **front-end** is the most challenging part of the application. It revolves around several technologies, tools and programming languages.

The main development environment for the front-end of this application is JetBrains's WebStorm IDE. It provides a very friendly and comprehensive development space, with features such as very good code completion, error detection and automatization of different processes, such as importing modules into typescript files. I prefer it over Microsoft's Visual Studio Code(which is the tool most front-end developers use) because it has much better code assistance and integration with version control tools, like github or gitlab. Visual Studio code can also be heavily customized and potentially perform even better than Webstorm, although in order to achieve this you would have to find and install various plugins and extensions, and many of those come as standard in JetBrain's Webstorm IDE.

The framework used for developing the front-end of CryptoAvenue is Angular (specifically version 17.0.9) and Angular CLI. Angular is a framework for building single-page applications using HTML and Typescript. Its architecture is based on components, which are pieces of code, each one containing and HTML element(.HTML file), a styling element(.CSS or .SCSS file), a typescript file which holds all the logic of the component(.TS file) and a test file (.SPEC.TS file). Angular also incorporates the concept of modules. These modules help organize an application into different coherent blocks of functionalities. Modules are decorated with the '@NgModule' decorator. The starting point of an angular application is the root module, which is usually called 'AppModule'.

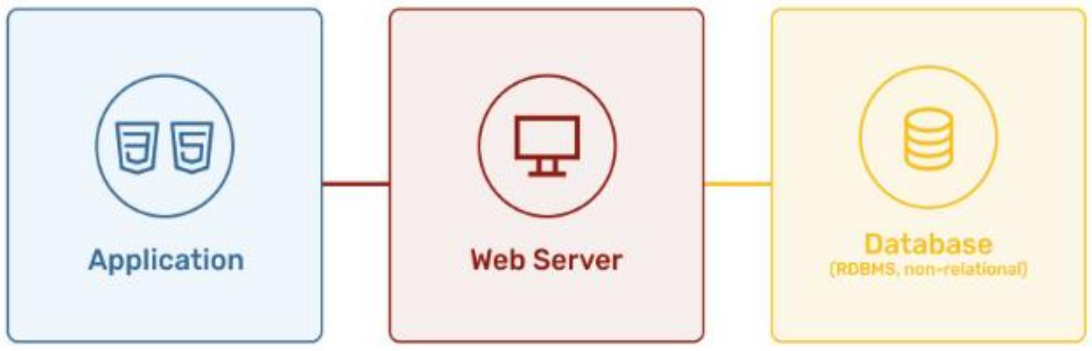
Angular CLI (Command Line Interface) is a very powerful tool that helps the developer by simplifying the process of development and maintenance of Angular applications. It offers different functionalities, such as being able to create an entire component, service or module directly from the command line, instead of having to create each file by hand.

Typescript is a statically superset of Javascript developed by Microsoft. It adds optional typing, allowing the users to define different types for variables, function parameters and return types. This helps catch some type-related errors that were otherwise pretty difficult to solve in a vanilla Javascript application. This feature of typescript greatly improves its integration with different IDEs, allowing for features such as code completion, real-time type checking and the option to refactor code documents. All these make for a much more interactive and user-friendly experience for the developer.

HTML(HyperText Markup Language) is the standard language used for creating web pages and it is processed by all the available browsers.

These layers communicate between each other through the HTTP(HyperText Transfer Protocol) along with all its methods (GET, POST, PUT, PATCH, DELETE). These types of requests are usually made for CRUD (Create-Read-Update-Delete) operations, like retrieving a user's information from the database or creating a new account

The .NET Core back-end interacts with the SQL Server database through EF Core (Entity Framework core), which is a popular ORM (Object-relational Mapper) that simplifies database operations by allowing the users to create database queries directly by using .NET objects through LINQ (Lanquage-Integrated Query).



2.2 Back-end Technologies

2.2.1. ASP.NET Core Web API

ASP.NET Core is a high-performance framework that developers use to create modern web applications. It allows programmers to design microservices, online apps and , most importantly, web APIs. It also has a notable speed advantage and easier configuration over other similar frameworks.

In this subchapter I am going to present the ASP.NET Core Web API project type and how it is used for creating fullstack applications and integrated into Microsoft's Visual Studio IDE. This project type is ideal for building API's that can serve data to different clients, such as desktop apps, mobile apps or even IoT devices.

At the base of the ASP.NET Core Web API there is the **Program.cs** file, which serves as the starting point of the application.Before 2019,nside this file we could find the **Main()** method. This method was the first that was being called whenever the application was executed. However, since 2019, starting with .NET 6 , the developers at Microsoft have changed the program.cs file in order to simplify the building process of the application by deleting the Main() method.

In this Program.cs file the WebApplication Builder is initialized, which sets up the host and the necessary configurations for the project.

```
var builder = WebApplication.CreateBuilder(args);
```

Next, the class contains the registration of various services with the dependency injection container. Methods such as AddEdnpointsApiExplorer() and AddControllers() set up the MVC controllers and AddSwaggerGen() adds the Swagger component for API documentation purposes. After this, users can also register custom services, HTTP clients and repositories. Here is an example of a few services registrated into the application through the Program.cs file:

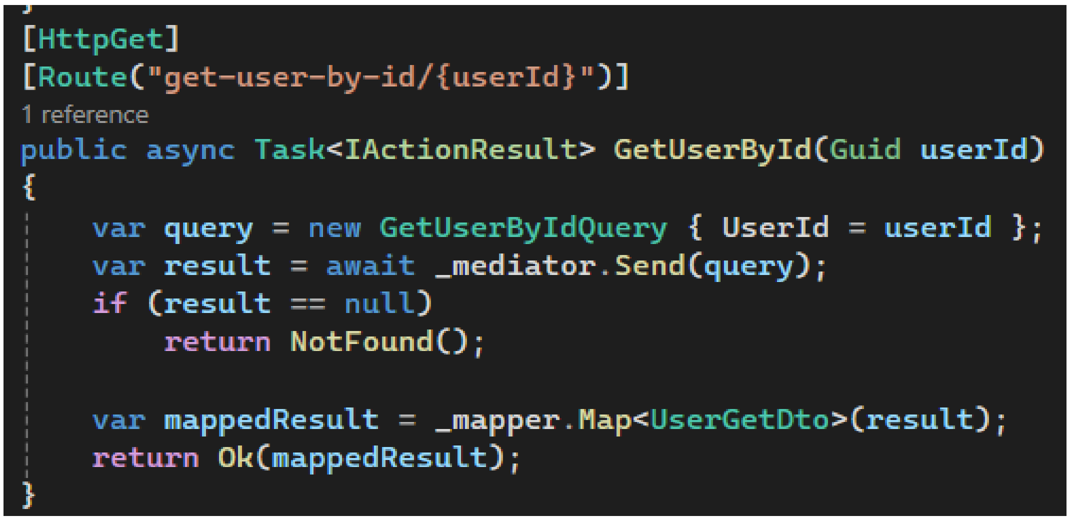
```
builder.Services.AddControllers();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
```

After these the Program.cs file can be used to register several other functionalities in the scope of the application, such as Controllers, JSON Options, **MediatR**, **AutoMapper,** database context and whatever else is necessary for the application to function according to plan.

In ASP.NET Core Web API projects, controllers are very important components that are used for handling HTTP request but also generating HTTP responses. Basically, they function as entry points for the client applications (mobile applications, desktop applications) to interact with the logic on the server side.

A very important part of a controller is the **routing**. Controllers define HTTP requests to certain action methods. There are 3 possible types of routing: attribute-based, convention-based or a combination of both. Attribute routers can be decorated in order to specify routes directly on the action methods. Action methods can handle specific HTTP verbs, such as GET, POST, PUT, PATCH, DELETE. Each action method corresponds to one of the endpoints and it executes the logic that needs to be executed in order to process the request.

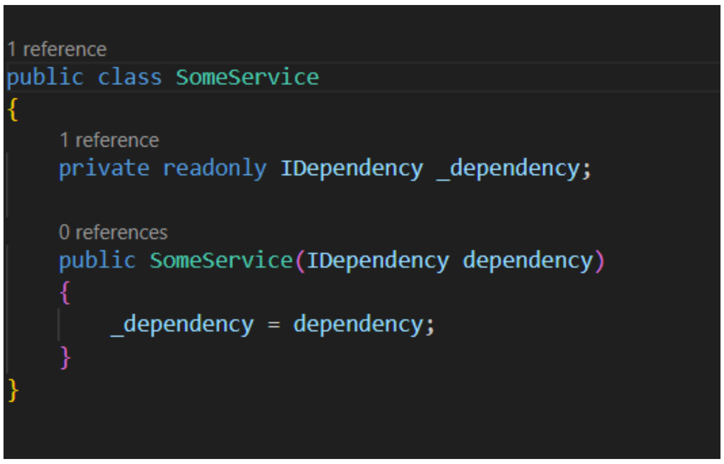
Below I will show you an example of a method inside one of my controllers that contains both an HTTP verb and routing, specifically an HTTPGET method:



```
[HttpGet]
[Route("get-user-by-id/{userId}")]
1 reference
public async Task<IActionResult> GetUserById(Guid userId)
{
    var query = new GetUserByIdQuery { UserId = userId };
    var result = await _mediator.Send(query);
    if (result == null)
        return NotFound();

    var mappedResult = _mapper.Map<UserGetDto>(result);
    return Ok(mappedResult);
}
```

HTTP requests are used to communicate between clients (browsers, mobile apps) and servers. Each one of these methods has a specific purpose. The GET request is used to fetch data from the server, the POST request is used to submit data to the server(for purposes such as adding or updating entries in the database), the PUT request is usually used to update or create a specific resource, the PATCH request is used to apply minor modifications to a resource (this is more often used when you are 100% sure that the purpose of the request is to update a resource and not to create one), and finally, the DELETE request is used to remove a resource from the server (such as deleting a database entry). There are other requests (HEAD and OPTIONS), but they are very rarely used in scope of APIs, especially for the .NET Core framework.

The ASP.NET Core framework has built-in support for dependency injection. It is a fundamental part of most .NET applications. The used services are usually registered in the Program.cs class, therefore the class which has such a service injected into it will know where to take the data from. Such services are injected into classes through constructor injection, but method injection and property injection are used regularly as well. Below I will show you an example of how a dependency can be injected into a class in the ASP.NET Core framework(specifically, constructor injection):


```
1 reference
public class SomeService
{
    1 reference
    private readonly IDependency _dependency;

    0 references
    public SomeService(IDependency dependency)
    {
        _dependency = dependency;
    }
}
```

Another very important part of .NET Core applications is the **NuGet Package Manager**. NuGet is free package-manager for the .NET environment that offers resources for building, sharing and using different packages. It makes adding and modifying libraries in the .NET framework much easier.

Some important elements of the NuGet Package Manager are:

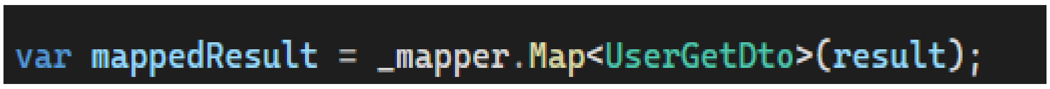
-package creation: libraries and tools can be included in packages that developers can make and publish.

-dependency management: NuGet maintains all the different dependencies between packages, making sure that they are compatible with each other

-versioning: it gives the developers the ability to choose between all the existing versions of a package, with the restriction that the version they choose must be compatible with the version of .NET Core they are using.

Using NuGet in Microsoft's Visual Studio is fairly easy: users can either use the Package Manager Console, which is a terminal-like interface where they can install different packages or maintain the existing ones or they can right click the project's solution and select 'Manage NuGet Packages', after which they will be prompted to a screen where they can search and install whichever NuGet packages they please.

The last part of this subchapter worth mentioning is the **AutoMapper** NuGet package, which is essential for simplifying the process of mapping one entity to another. Example: let's say you have an endpoint that retrieves a user based on their ID. In the User class you would typically have some sensitive information, such as their password, and you wouldn't want to send such private information back to the client. Therefore, you create a new class (UserGetDto), which will hold most of the user's information, except for the password, and then you map the User class to the new UserGetDto class. Example of how the AutoMapper is used:


```
var mappedResult = _mapper.Map<UserGetDto>(result);
```

2.2.2. Data Access Layer (DAL)

This subchapter will cover how Data Access Layer subprojects are usually constructed in the .NET Core framework and how **Entity Framework core** is integrated into them.

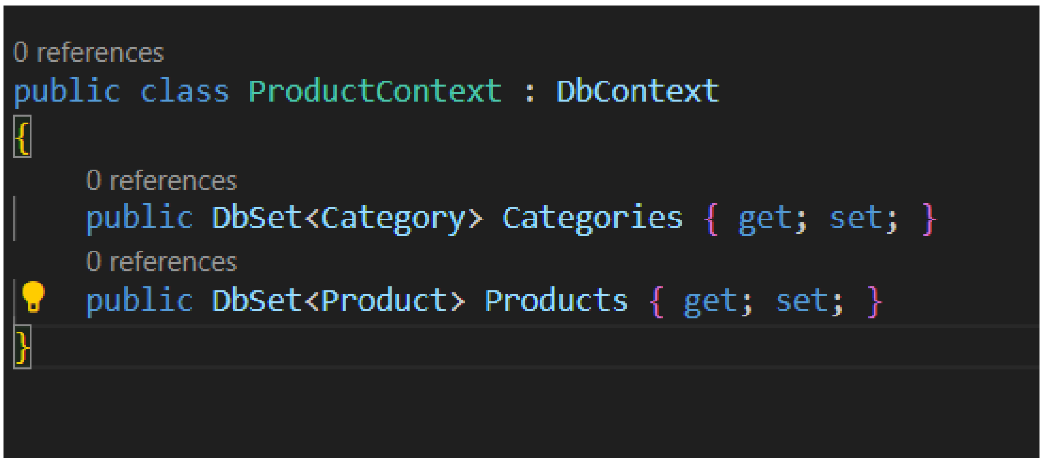
Data Access Layer(.Dal) subprojects are usually in the form of a class library and they are used for defining the database and establishing the supply chain or connection between the application side of the project and the database layer.

Entity Framework Core is an open-source, lightweight and extensible version of the Entity Framework (a popular Object-Relational Mapping – ORM for .NET Core applications). Entity Framework Core allows developers to communicate with the database directly by using .NET objects, so they wouldn't need to write a lot of unnecessary data-access code. This framework also support Microsoft's Language Integrated Query (or LINQ).

For such projects, a class named 'Application_name_DbContext) must be created. This class inherits from the DbContext class (System.Data.Entity.DbContext).

An essential part of the Entity Framework Core is the DbContext class, which serves as a connection between your domain(entity classes) and the database. It deals with the entity object management during runtime, which includes database retrieval and change tracking. It also also responsible for opening or closing the connections do the database, whenever such actions are necessary. It also configures the database schema through the **OnModelCreating()** method, which is used to set up constraints and relationships between different entities from the database (such as foreign key constraints).

Another important property of the Entity Framework Core and the DbContext class are the properties defined as DbSet<T>, where T represents an entity type. These entity sets each correspond to one of the models of the application, meaning a database table. Here, you can see an example from Microsoft's documentation where 2 different DbSet<T>s are used in the scope of the DbContext class:

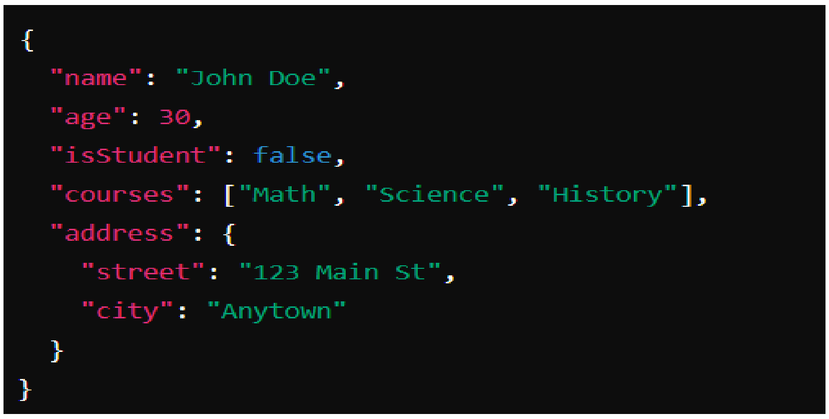


```
0 references
public class ProductContext : DbContext
{
    0 references
    public DbSet<Category> Categories { get; set; }
    0 references
    public DbSet<Product> Products { get; set; }
}
```

2.2.3 .NET Core Configuration Files

Before we dive into this interesting subchapter I would like to point out the meaning of JSON(short for JavaScript Object Notation) files. A JSON file stores simple forms of data structures in a text format. Similarly to dictionaries, it uses key-value pairs. JSON is language-independent, but uses similar features which should be easy to understand for developers that are familiar with programming languages such as C,C++, Java or Python.

A JSON file consists of the following components: objects, defined by curly braces '{}' with key-value pairs, arrays, which are an ordered list of values enclosed in square brackets '[]' and values, which can take many forms like strings, arrays, objects, Booleans etc. If you look below this line you will see an example of how a JSON object is defined. In this case, a JSON object is created to store the information of a person, very similar to an object:

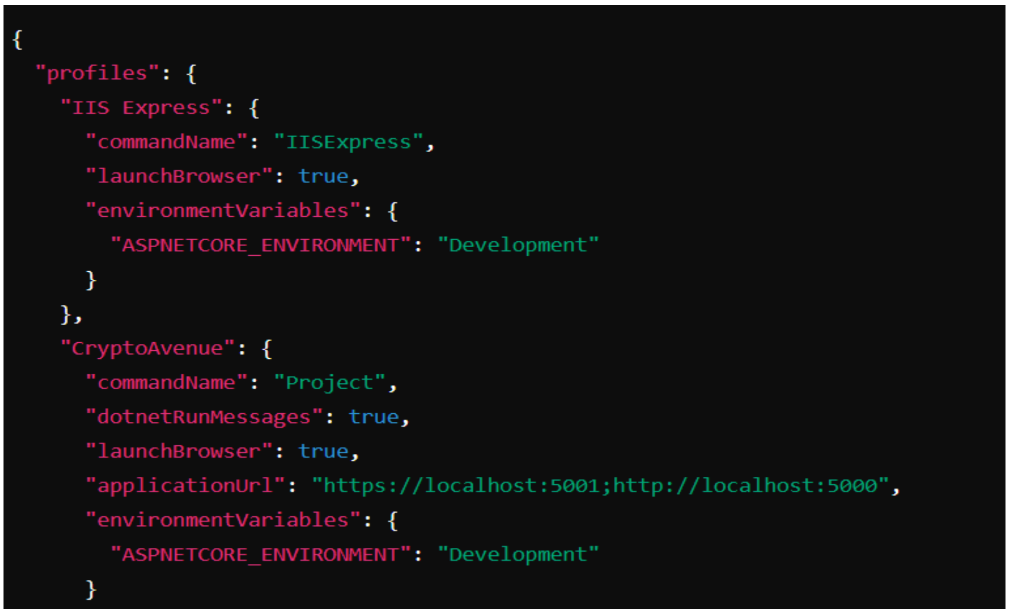


```json
{
  "name": "John Doe",
  "age": 30,
  "isStudent": false,
  "courses": ["Math", "Science", "History"],
  "address": {
    "street": "123 Main St",
    "city": "Anytown"
  }
}
```

.NET Core applications use several configuration files to handle environment specific setups and settings. Out of all these files, the 2 most important ones are the "appSettings.json" and the "launchSettings.json".

The "appSettings.json" file is used to store the main configurations of a .NET Core application. It is a very convenient way to store information such as development, testing and production configurations all into one place. The settings in this file are all loaded at runtime and can be accessed throughout the application. The appSettings file is loaded automatically in the Program.cs whenever a new project is created.

The "launchSettings.json" is used to configured how the application is launched. It contains information related to the deployment environment, such as the server URL and environment variables. This file is mainly used during the development stage and it is not deployed with the application. Below you can find an example of how the deployment configurations are written in the "launchSettings.json" file:



```
{
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "CryptoAvenue": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "https://localhost:5001;http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
```

2.3. Front-end Technologies

In this subchapter I will go into more detail about the technologies and design techniques I have used for the purpose of building my application.

2.3.1. "tsconfig.json" File in Angular

This file is a configuration file found in Angular projects that defines how the compiler should transform the TypeScript code into vanilla JavaScript. It allows the users to change the compilation process by specifying compiler options or other settings, like the target, module or root directory of the application.

2.3.2. "package.json" File in Angular

This file is a very important component of Angular based applications, since is stores the metadata relevant to the project. It contains crucial information about the project, such as the name, version, scripts, dependencies, devDependencies, the entry point of the application, the author and the license under which the project is distributed.

2.3.3. .ICO Files in Angular

.ico files are often used to set the favicon of an application, which represents the small image found in the address bar of the browser or in the bookmarks.

2.3.4. .PNG and .JPG Files in Angular

These types of files are generally used for storing static or background images which are going to be utilized for styling the applications. They can usually be found in the root folder of the application under “assets/resources”.

2.3.5. “SPEC.TS” File in Angular

Files of type .spec.ts have the purpose of holding the testing specifications of Angular components. They are written in TypeScript and their role is to ensure the correct functionality of their parent component based on tests written by developers.

2.3.6. PRIMENG

PrimeNG is one of the main UI component libraries meant to be used by Angular applications, along with AngularMaterial. It offers a vast range of features that enhance the development of modern, responsive and dynamic web applications.It makes life easier for developers by providing them with a rich set of pre-built and easily-customizable UI elements. Think of it as an enhanced version of vanilla HTML.

This vast library offers more than 80 different UI components, covering use cases such as: forms, input fields, menus, charts and many more. This means that developers won’t need to waste their time on building these complicated UIs from scratch, like they would in a normal Angular application.

All the PrimeNg elements are responsive, meaning that they will look good on any type of device, whether it is a mobile, tablet or desktop.

PrimeNg elements are also highly customizable and themable. Developers can modify each components styles in order to match the theme or color scheme of their application. Also, PrimeNg offers developers certain pre-defined themes they can purchase to have a good starting point for their UI.

In my opinion, the biggest benefit of using PrimeNg is the great documentation you can find on it. This vast documentation provides multiple examples for each use case of every single UI component, along with guides and API references. It also has a very good community backing it and a variety of support options.

Here is a code snippet that shows an example of a PrimeNg element called ‘p-table’ and how it is being used.

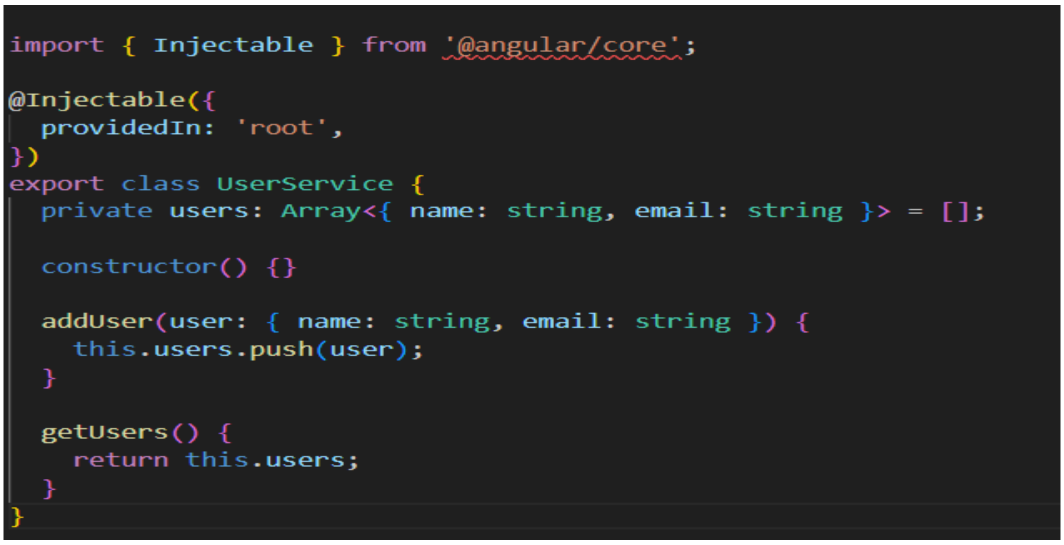
```
<p-table [value]="users" [paginator]="true" [rows]="10" [responsiveLayout]="'scroll'">
    <ng-template pTemplate="header">
        <tr>
            <th>Name</th>
            <th>Email</th>
            <th>Age</th>
        </tr>
    </ng-template>
    <ng-template pTemplate="body" let-user>
        <tr>
            <td>{{ user.name }}</td>
            <td>{{ user.email }}</td>
            <td>{{ user.age }}</td>
        </tr>
    </ng-template>
</p-table>
```

This example showcases a table containing the values of an array “users”, with additional properties like [paginator] and [rows] that configure the pagination of the table. The “ng-template”s used define the header and body sections of the table.

2.3.7. Services in Angular

Services are essential building blocks of Angular applications. They allow developers to define functionalities which can be then incorporated into other components of the application. They can be used to handle tasks such as fetching data, business logic, state management and communication between 2 or more different components inside the application.

Services can be generated through the Angular CLI by using the command “ng generate service service_name”. They must import the “Injectable” module from “@angular/core” and they require to be decorated with the @Injectable decorator. Here is a basic example of an Angular service that adds or retrieves users from an array:



```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root',
})
export class UserService {
  private users: Array<{ name: string, email: string }> = [];

  constructor() {}

  addUser(user: { name: string, email: string }) {
    this.users.push(user);
  }

  getUsers() {
    return this.users;
  }
}
```

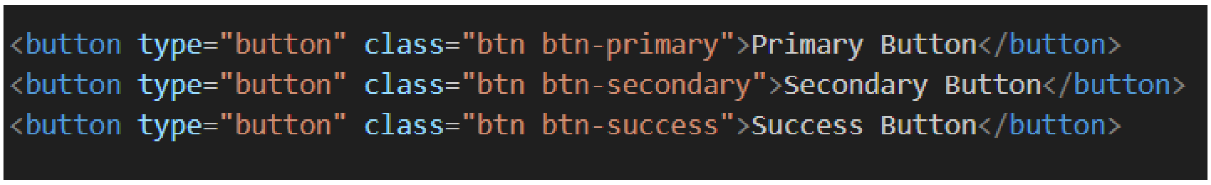
2.3.8. Dependency Injection in Angular

In Angular, services are decorated with the "@Injectable" decorator, which makes them available for injection. After this, other components can utilize those services by adding them in the constructor of the class.

2.3.9. BOOTSTRAP

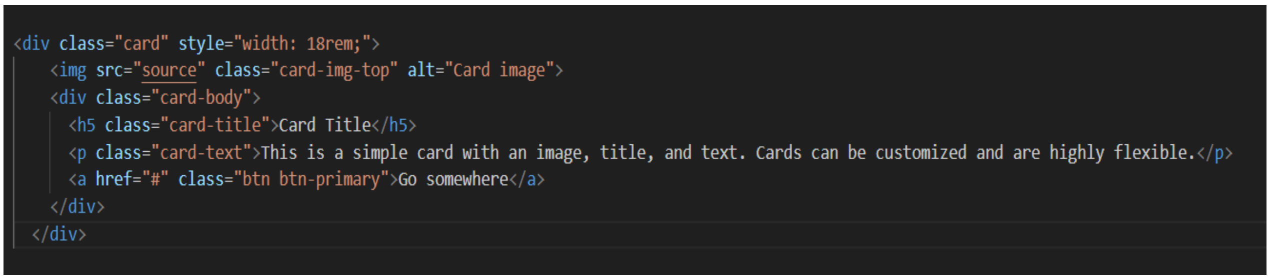
Bootstrap is the final UI library I have used in my application. It is very popular among web development and design because it offers a set of tools which help create a consistent and modern design for any web application. In order to install it you would have to type in the following command in your terminal: "npm install bootstrap". Here are some basic examples of bootstrap usages:

-buttons:



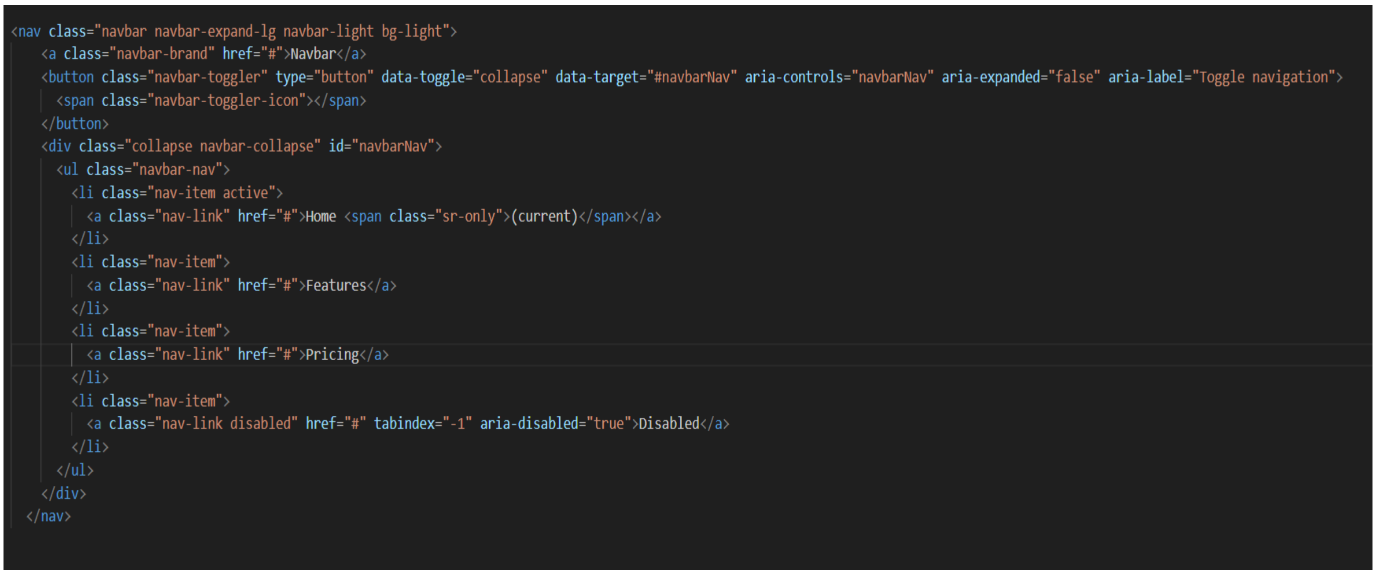
```
<button type="button" class="btn btn-primary">Primary Button</button>
<button type="button" class="btn btn-secondary">Secondary Button</button>
<button type="button" class="btn btn-success">Success Button</button>
```

-cards:



```
<div class="card" style="width: 18rem;">
    <img src="source" class="card-img-top" alt="Card image">
    <div class="card-body">
      <h5 class="card-title">Card Title</h5>
      <p class="card-text">This is a simple card with an image, title, and text. Cards can be customized and are highly flexible.</p>
      <a href="#" class="btn btn-primary">Go somewhere</a>
    </div>
  </div>
```

-navigation bars



```html
<nav class="navbar navbar-expand-lg navbar-light bg-light">
    <a class="navbar-brand" href="#">Navbar</a>
    <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarNav" aria-controls="navbarNav" aria-expanded="false" aria-label="Toggle navigation">
      <span class="navbar-toggler-icon"></span>
    </button>
    <div class="collapse navbar-collapse" id="navbarNav">
      <ul class="navbar-nav">
        <li class="nav-item active">
          <a class="nav-link" href="#">Home <span class="sr-only">(current)</span></a>
        </li>
        <li class="nav-item">
          <a class="nav-link" href="#">Features</a>
        </li>
        <li class="nav-item">
          <a class="nav-link" href="#">Pricing</a>
        </li>
        <li class="nav-item">
          <a class="nav-link disabled" href="#" tabindex="-1" aria-disabled="true">Disabled</a>
        </li>
      </ul>
    </div>
  </nav>
```

3. My Implementation

In this chapter I will present various parts of my codebase and a step-by-step explanation of the entire development process of my application, starting with the database layer, followed by the back-end layer and finally, the front-end layer.

3.1. The Database

The database I modelled for the purpose of this application is fairly simplistic. I used 5 database tables: Users, Coins, Wallets, Transactions and WalletCoins. Each of these tables is based on a model entity, which contains the 'Id' field, or the unique identifier. It is the primary key for every table. The 'Id' is of GUID format. A GUID is a 128-bit value followed by 3 groups of 4 hexadecimal digits each, followed by one group of 12 hexadecimal digits. Example: 6B29FC40-CA47-1067-B31D-00DD010662DA.

The **Users** table contains all the information related to the clients that register an account into my application. Its fields are: Username, Email and Password.

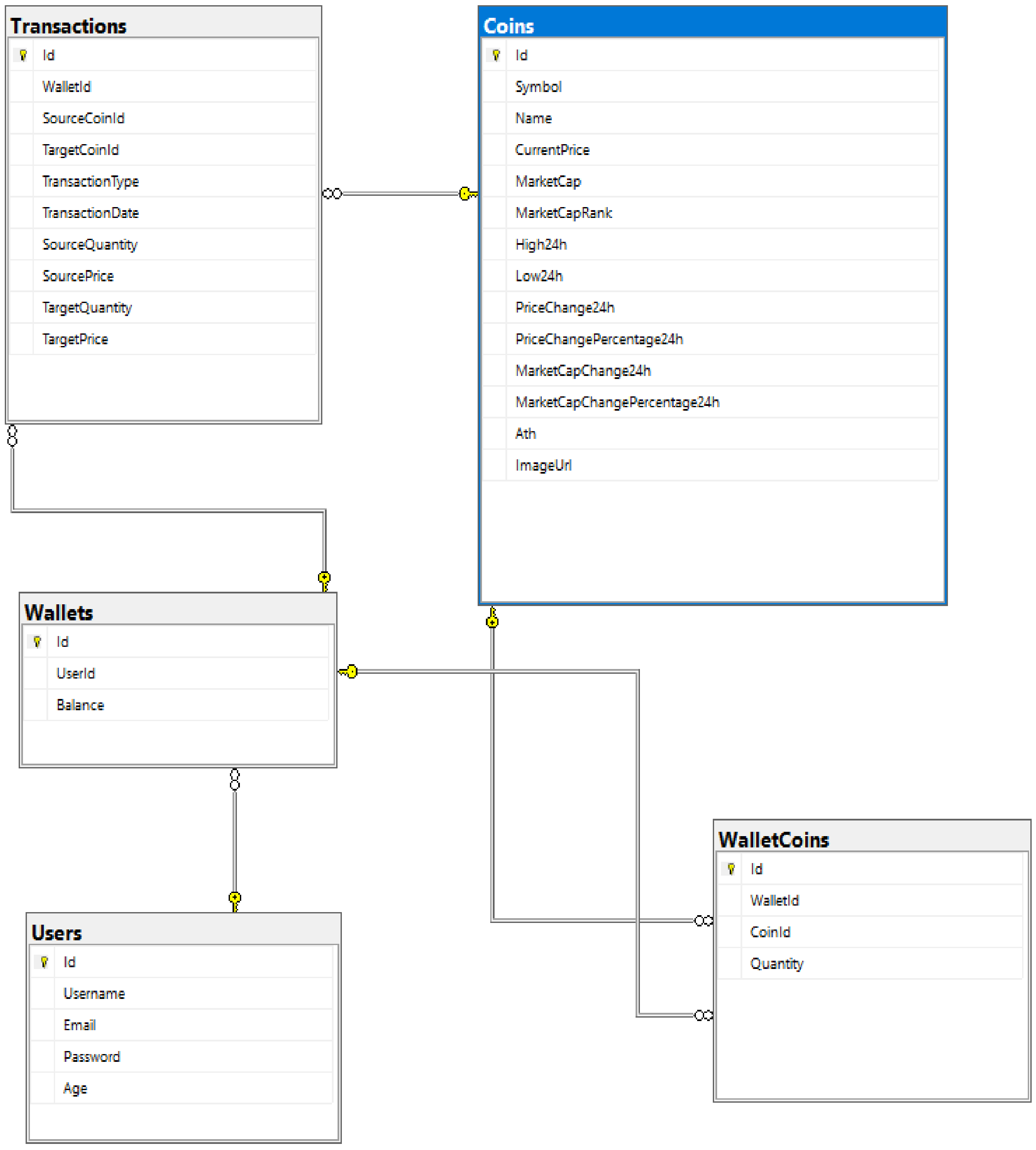
The Coins table is based on the output of the /coins/markets/list endpoint from the CoinGecko API. It contains relevant information about cryptocurrencies, such as Name, Symbol, CurrentPrice, MarketCap, MarketCapChange24h etc.

The Wallets table has only 2 fields, other than the standard Id, which are the UserId – a foreign key pointing to the Users table and the Balance field – a sum of the values of all the cryptocurrencies owned by the user specified by the UserId.

The WalletCoins table was designed for managing the many-to-many relationship between the Wallets and the Coins tables. Besides the Id, it contains 3 other fields: WalletId – foreign key pointing to the Wallets table, CoinId- foreign key pointing to the Coins table and Quantity, which signifies the amount of the cryptocurrency specified by the CoinId the user possesses.

Finally, the Transactions table is the most comprehensive table. Each time a user buys, sells or trades cryptocurrencies inside CryptoAvenue, a new transaction will be created and stored in the database. This table contains multiple fields: WalletId – points to the Wallet of the user who made the transaction, SourceCoinId – points to the Id of the coin that was sold, TargetCoinId – points to the Id of the coins that was bought, TransactionType – can either be 'BUY', 'SELL' or 'TRADE', SourceQuantity – the quantity of the sold coin, SourcePrice – the price of the coin that was sold, TargetQuantity – the quantity of the coin that was purchased, TargetPrice – the price of the coin that was purchased.

Below this paragraph I have inserted a diagram of my database, provided by Microsoft's SQL Server Management Studio:


Transactions
Id
WalletId
SourceCoinId
TargetCoinId
TransactionType
TransactionDate
SourceQuantity
SourcePrice
TargetQuantity
TargetPrice
Coins
Id
Symbol
Name
CurrentPrice
MarketCap
MarketCapRank
High24h
Low24h
PriceChange24h
PriceChangePercentage24h
MarketCapChange24h
MarketCapChangePercentage24h
Ath
ImageUrl
Wallets
Id
UserId
Balance
WalletCoins
Id
WalletId
CoinId
Quantity
Users
Id
Username
Email
Password
Age

3.2. The Back-End

In this subchapter I am going to present the back-end layer of my application, built in ASP.NET Core using Microsoft's Visual Studio IDE.

3.1. Structure

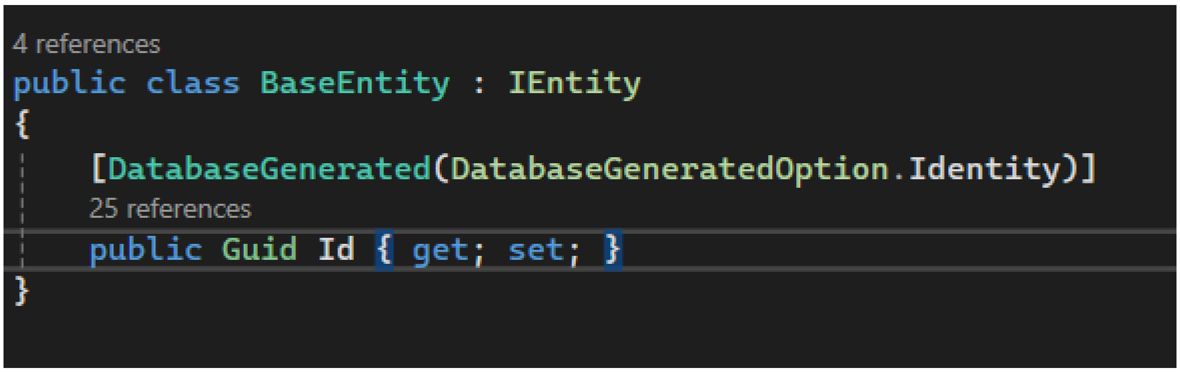
In CryptoAvenue, the back-end is separated into 4 different subprojects. Each one of these projects has a specific purposes and distinct functionalities: the **API** layer, the **Application** layer, the **Domain** layer and the **Data Access** layer. This separation is built based on a well-known design pattern called **Clean Architecture.** This pattern promotes a well-defined separation of the functionalities inside the application and it allows for easier maintability.

The API layer is of type ASP.NET Core Web API serves as the entry point for the application. It handles HTTP requests (GET, POST, PUT, PATCH, DELETE) and uses the mediator design pattern to route them to their corresponding methods, after which it returns the HTTP response. This layer contains classes like the controllers, Data transfer objects, middleware, profiles, services and the Program.cs file, which is the main starting point of the entire application. In this file the developer can register various settings of the app, like services, controllers and repositories.

The Application Layer is of type Class Library and contains the main business logic of the application. It manages the data flow between the API and the domain or database layer. It defines the most complex functionalities of the application. This layer contains objects such as command, queries and their respective handlers.

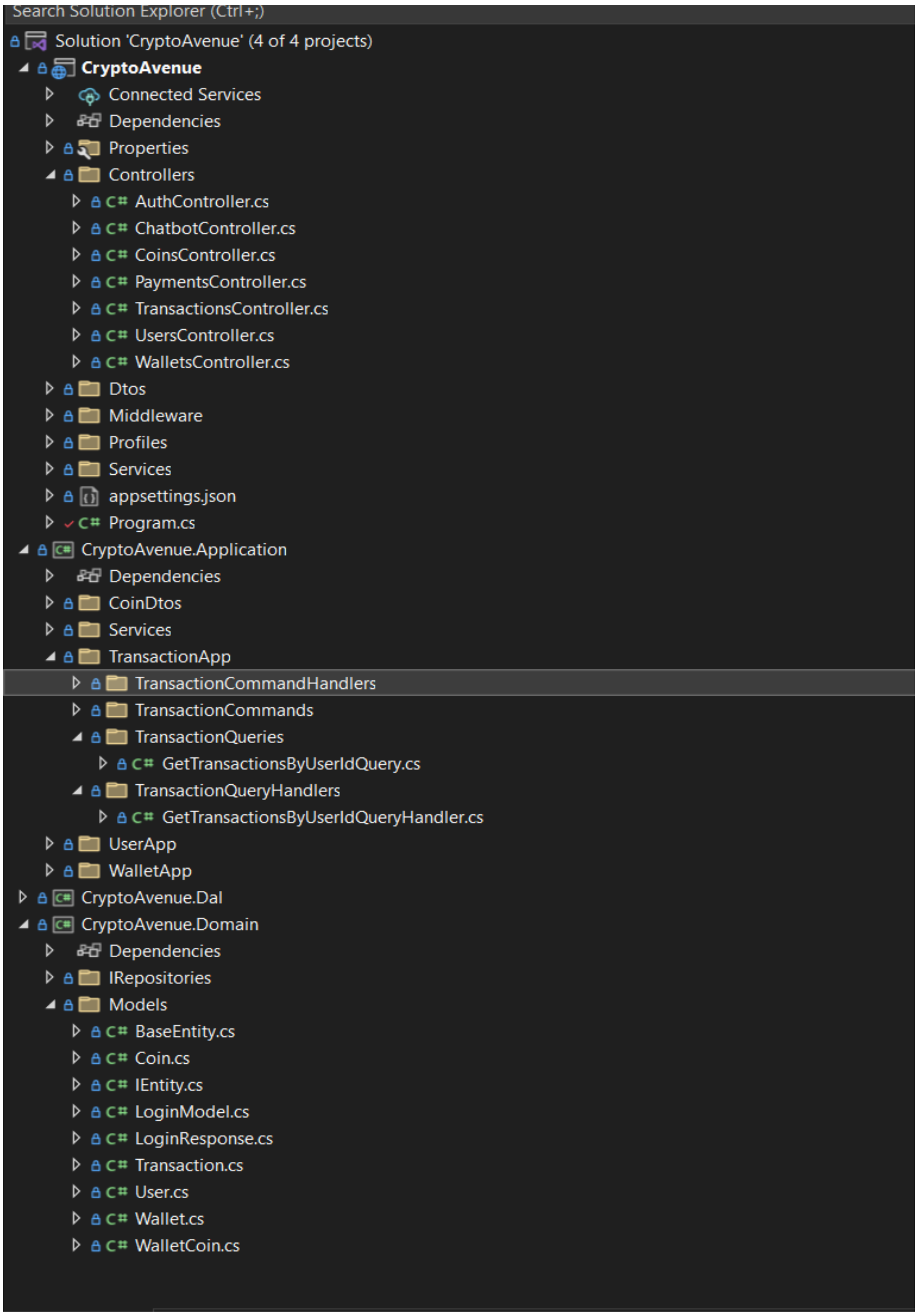
The Data Access Layer (DAL) is of type Class Library and handles the interaction between the application and the database. It abstracts the data storage and retrieval, making it easier and cleaner for the rest of the application to access it. This layer contains objects such as the DbContext and repositories for each type of entity from the Domain layer.

The final layer is the Domain Layer, also a Class Library, represents the entities used in the application and also the interfaces for each repository. Each one of the entities inherits from an entity called 'BaseEntity', which contains the Guid property – the primary key of every entity in my database. The base entity look like this:



```
4 references
public class BaseEntity : IEntity
{
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    25 references
    public Guid Id { get; set; }
}
```

Below I will attach a screenshot that shows the structure of my back-end projects, extracted from the file explorer in Microsoft's Visual Studio:



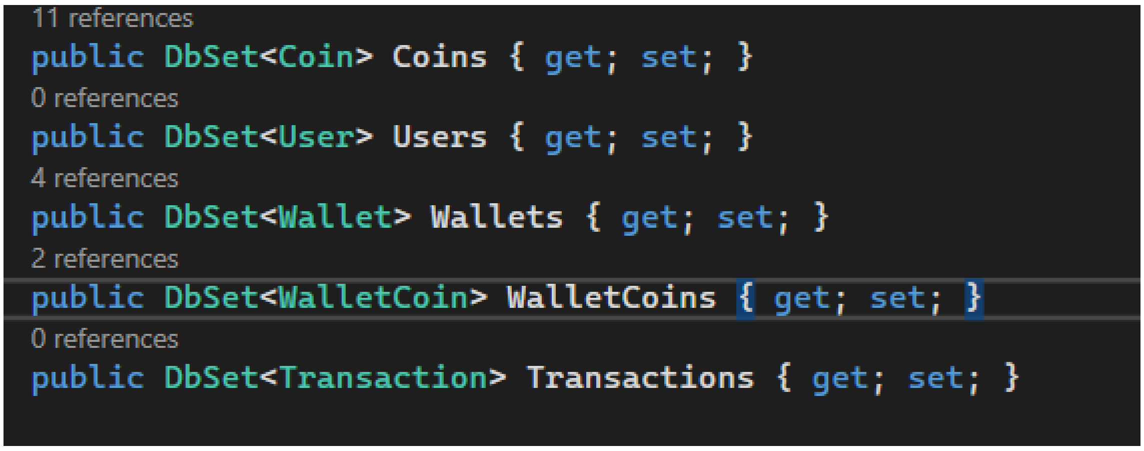
```
Search Solution Explorer (Ctrl+;)
Solution 'CryptoAvenue' (4 of 4 projects)
  CryptoAvenue
    Connected Services
    Dependencies
    Properties
    Controllers
      AuthController.cs
      ChatbotController.cs
      CoinsController.cs
      PaymentsController.cs
      TransactionsController.cs
      UsersController.cs
      WalletsController.cs
    Dtos
    Middleware
    Profiles
    Services
    appsettings.json
    Program.cs
  CryptoAvenue.Application
    Dependencies
    CoinDtos
    Services
    TransactionApp
      TransactionCommandHandlers
      TransactionCommands
      TransactionQueries
        GetTransactionsByUserIdQuery.cs
      TransactionQueryHandlers
        GetTransactionsByUserIdQueryHandler.cs
    UserApp
    WalletApp
  CryptoAvenue.Dal
  CryptoAvenue.Domain
    Dependencies
    IRepositories
    Models
      BaseEntity.cs
      Coin.cs
      IEntity.cs
      LoginModel.cs
      LoginResponse.cs
      Transaction.cs
      User.cs
      Wallet.cs
      WalletCoin.cs
```

3.2. The DbContext

In this subchapter I will talk about the **DbContext** class of my application, specifically CryptoAvenueDbContext.

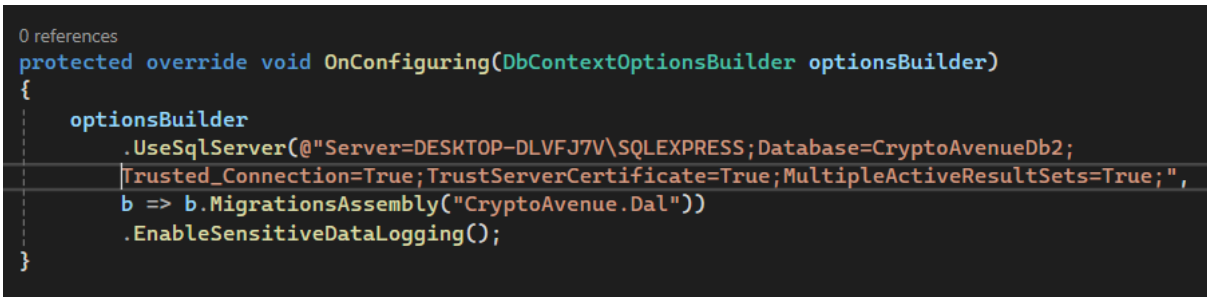
This class plays a crucial role in the context of the application as it defines the main interaction with the database. It inherites the ‘DbContext’ class provided by Entity Framework Core, which is an Object Relational Manager (ORM) that makes data access easier in .NET Core applications by mapping the entities in the domain to the database tables. The constructor of this class accepts ‘DbContextOptions’ as a parameter.

The properties inside this class are of type DbSet<T>, where T represents an entity from my domain layer. Each one of these sets corresponds to a database table. Here are all the DbSets I used in my DbContext:



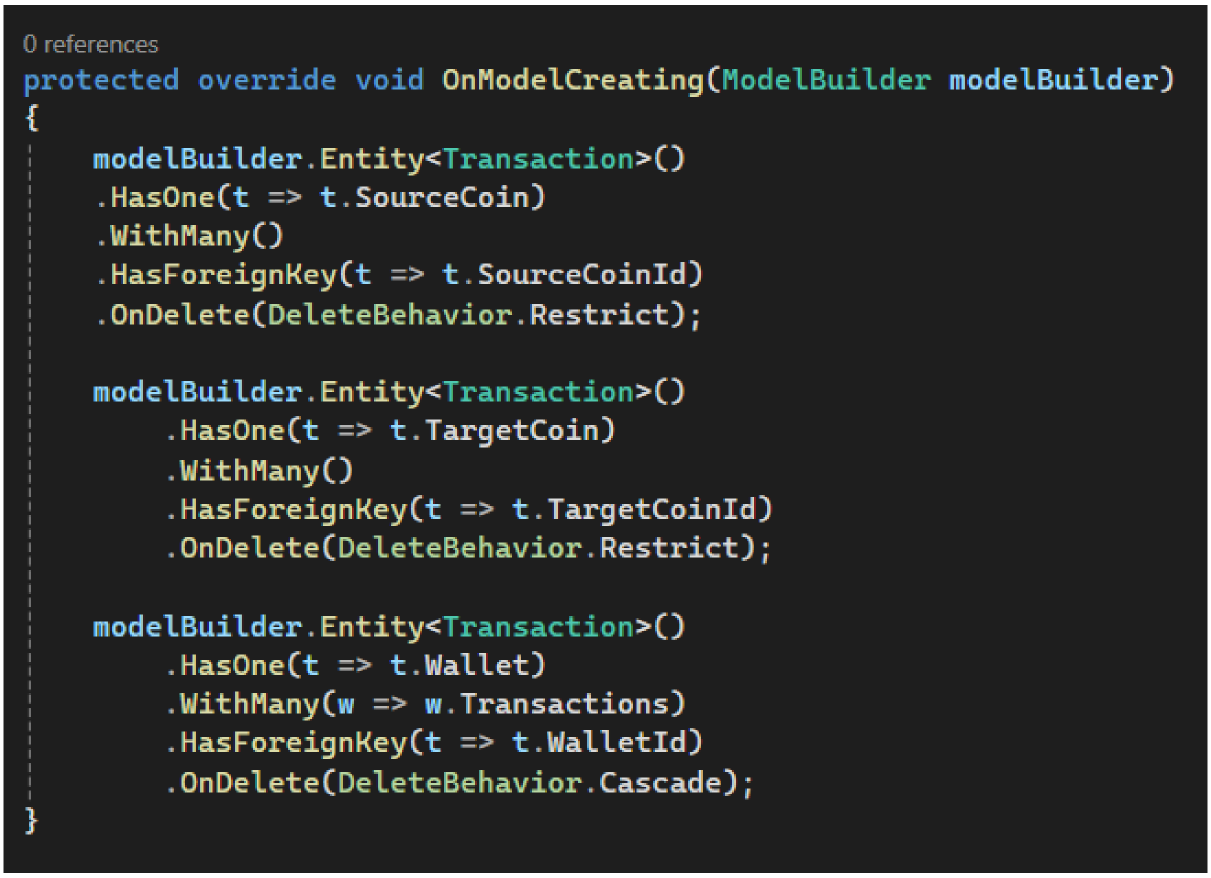
```
11 references
public DbSet<Coin> Coins { get; set; }
0 references
public DbSet<User> Users { get; set; }
4 references
public DbSet<Wallet> Wallets { get; set; }
2 references
public DbSet<WalletCoin> WalletCoins { get; set; }
0 references
public DbSet<Transaction> Transactions { get; set; }
```

The **OnConfiguring()** method method of the DbContext configures the SQL Server database intended to be used by the application through the connection string. This string also provides other parameters like the ‘MigrationsAssembly’ that specifices where the migrations will be stored. Each time the developer updates the entities in the domain (therefore the database tables) by adding another property/field or deleting one entirely, he must open the NuGet Package Manager Console and type in the command ‘add-migration migration_name’. After this, a new migration class will be added to the ‘Migrations’ folder. This migration file specifies the exact changes that were made to the structure of the database. After creating the migration, provided that no errors occurred, the developer should use the command ‘update-database’, which will take the information from the last migration available and update the database tables accordingly. My OnConfiguring() method:



```
0 references
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder
        .UseSqlServer(@"Server=DESKTOP-DLVFJ7V\SQLEXPRESS;Database=CryptoAvenueDb2;
        Trusted_Connection=True;TrustServerCertificate=True;MultipleActiveResultSets=True;",
        b => b.MigrationsAssembly("CryptoAvenue.Dal"))
        .EnableSensitiveDataLogging();
}
```

Another important method in the DbContext class is the OnModelCreating() method. This one is used for configuring entities and the relationships between them in order to avoid SQL specific issues such as CASCADE Deletions. In my own case, I used this method to configure the relationships between the Transaction, Coin and Wallet entities. Specifically, I added the restriction that each transaction has one ‘sourceCoin’, with a foreign key ‘SourceCoinId’, so deleting a SourceCoin won’t automatically delete its associated Transactions because of DeleteBehaviour.Restrict. I did the same exact thing for the TargetCoin inside the Transaction Entity. The final configuration I did in this method is specifying that by deleting a ‘Wallet’ its associated Transactions will be deleted alongside it, due to ‘DeleteBehaviour.Cascade’. Here is my OnModelCreating() method:



```
0 references
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Transaction>()
    .HasOne(t => t.SourceCoin)
    .WithMany()
    .HasForeignKey(t => t.SourceCoinId)
    .OnDelete(DeleteBehavior.Restrict);

    modelBuilder.Entity<Transaction>()
        .HasOne(t => t.TargetCoin)
        .WithMany()
        .HasForeignKey(t => t.TargetCoinId)
        .OnDelete(DeleteBehavior.Restrict);

    modelBuilder.Entity<Transaction>()
        .HasOne(t => t.Wallet)
        .WithMany(w => w.Transactions)
        .HasForeignKey(t => t.WalletId)
        .OnDelete(DeleteBehavior.Cascade);
}
```

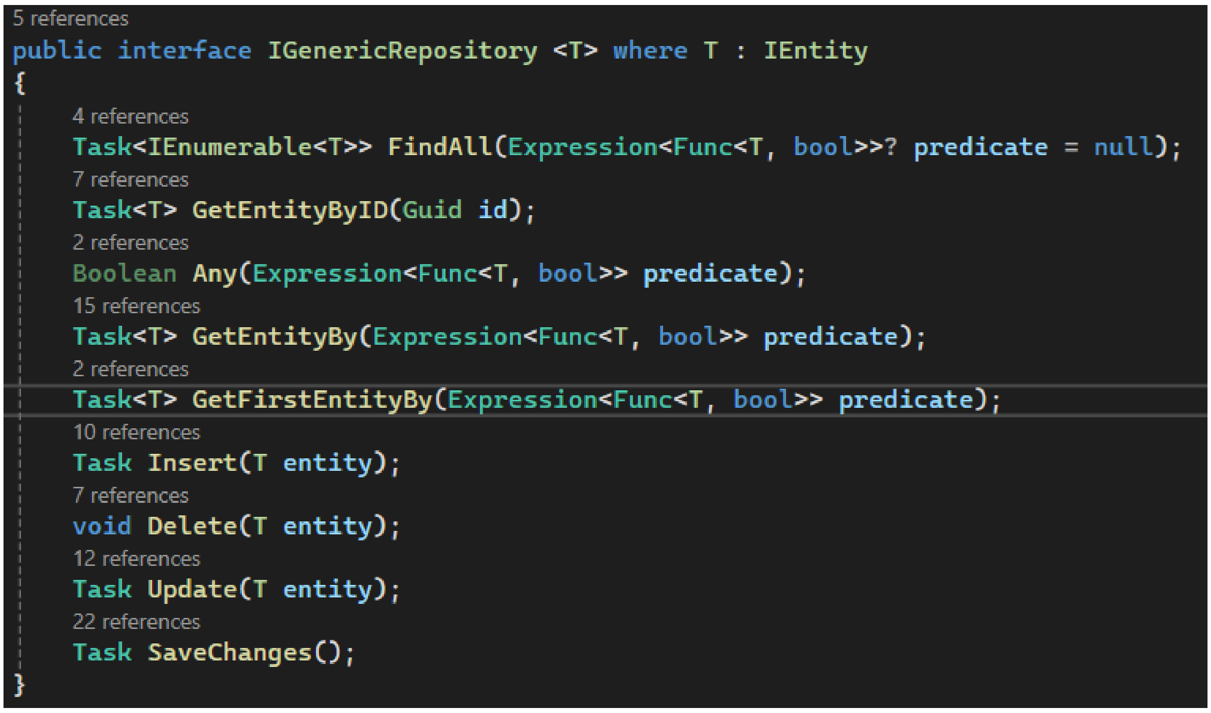
3.3. Generic Repository Pattern

In this subchapter I will explain how I used the generic repository pattern for easier access to data inside my application. I defined the interfaces for the repositories in the CryptoAvenue.Domain subproject and I added the implementations of each repository in the CryptoAvenue.Dal subproject. Adding them in the same subproject was impossible since a project reference exist from the .Dal to the .Domain (where the entities are defined), therefore I couldn't add a reverse project reference (from .Domain to .Dal) since ASP.NET Core doesn't offer support for two-way project referencing.

The Generic Repository Pattern is a software design pattern used in development for easier data abstraction and access. This specific pattern allows developers to create a set of common CRUD operations, such as create, update, delete or save changes that apply to any type of entity from the domain layer, leading to easier access and avoidance of code reusability and redundancy.

In CryptoAvenue, the generic repository pattern is implemented in order to manage data operations for entities like : Coin, User, Wallet, WalletCoin and Transaction.

The main interface for the generic repository is found in the 'IRepositories' folder inside of the CryptoAvenue.Domain subproject. The 'IGenericRepository' interface defines the signatures of all the methods that will be implemented by the repository classes.

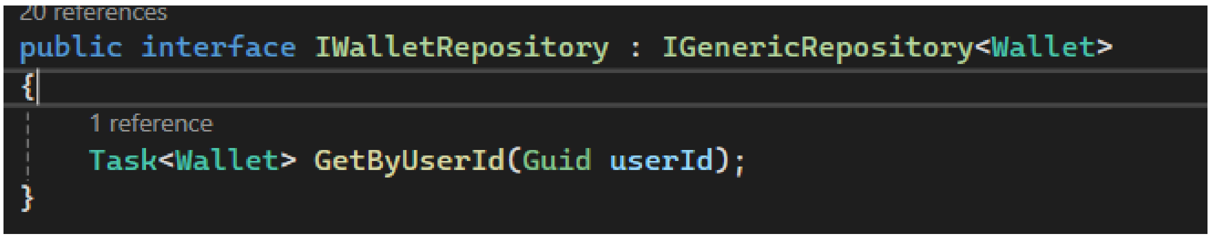


```
5 references
public interface IGenericRepository <T> where T : IEntity
{
    4 references
    Task<IEnumerable<T>> FindAll(Expression<Func<T, bool>>? predicate = null);
    7 references
    Task<T> GetEntityByID(Guid id);
    2 references
    Boolean Any(Expression<Func<T, bool>> predicate);
    15 references
    Task<T> GetEntityBy(Expression<Func<T, bool>> predicate);
    2 references
    Task<T> GetFirstEntityBy(Expression<Func<T, bool>> predicate);
    10 references
    Task Insert(T entity);
    7 references
    void Delete(T entity);
    12 references
    Task Update(T entity);
    22 references
    Task SaveChanges();
}
```

As you can see in the attached image above, the 'IGenericRepository' interface has a suite of methods used for data access and data modification. For example, the method 'FindAll()' is of type 'Task<IEnumerable<T>>'. Type Task<T> is often used in this application to promote the usage of asynchronous development (I will talk more about this later on). 'Task<T>' represents an operation that returns a value of type 'T'.

Operations of type ‘Task’ can be awaited, allowing for them to run without blocking the main execution of the program. Therefore, our ‘FindAll()’ method returns an operation with a return type of ‘<IEnumerable<T>>’. It takes as a parameter a predicate, which can be also null. This predicate specifices a LINQ querying parameter (condition) that will be used to navigate and search a list of elements. Another very important method is the ‘SaveChanges()’ method that is used for saving changes to the database after a modification has been made.

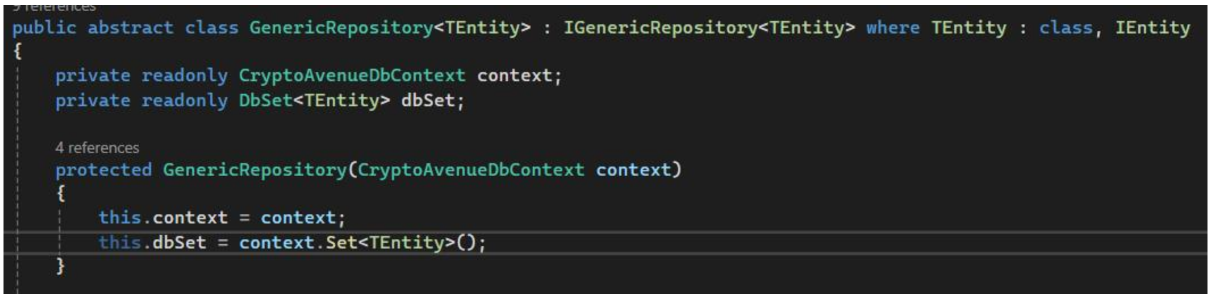
In the same folder we can also find an ‘IRepository’ interface for each of the domain entities, like ‘IWalletRepository’ or ‘IUserRepository’. Each one of these implement the ‘iGenericRepository’ interface. In these interfaces we can add other methods apart from the ones in the ‘IGenericRepository’, specific to the entity type. For example, below you will see a screenshot of the IWalletRepository, which implement the IGenericRepository interface with the specified type ‘Wallet’:



```
20 references
public interface IWalletRepository : IGenericRepository<Wallet>
{
    1 reference
    Task<Wallet> GetByUserId(Guid userId);
}
```

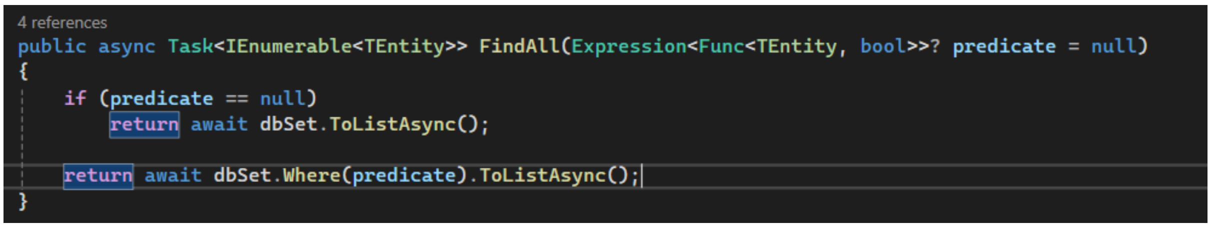
As you can see, the method ‘GetByUserId(Guid userId)’ has been added to this repository as it is only meant to be used by entities of type Wallet.

Moving on to the implementation of the GenericRepository implementations. These are found in the CryptoAvenue.Dal subproject (Data Access Layer), under the folder ‘Repositories’. Here is the definition of the GenericRepository class:



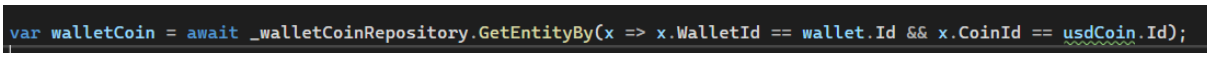
```
public abstract class GenericRepository<TEntity> : IGenericRepository<TEntity> where TEntity : class, IEntity
{
    private readonly CryptoAvenueDbContext context;
    private readonly DbSet<TEntity> dbSet;

    4 references
    protected GenericRepository(CryptoAvenueDbContext context)
    {
        this.context = context;
        this.dbSet = context.Set<TEntity>();
    }
```

This class is injected with the DbContext, which contains all the entity sets related to the domain entities. In this class we can find the implementations of the methods in the interface, such as the ‘FindAll()’ method that is responsible for search the entity set defined by the ‘TEntity’ type:

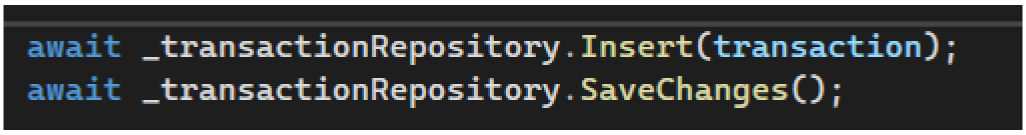


```
4 references
public async Task<IEnumerable<TEntity>> FindAll(Expression<Func<TEntity, bool>>? predicate = null)
{
    if (predicate == null)
        return await dbSet.ToListAsync();

    return await dbSet.Where(predicate).ToListAsync();
}
```

These repositories are used all throughout the application layer of the app. They are injected into the constructor of each command or query and used inside the handler method. Here are a few examples:



```
var walletCoin = await _walletCoinRepository.GetEntityBy(x => x.WalletId == wallet.Id && x.CoinId == usdCoin.Id);
```



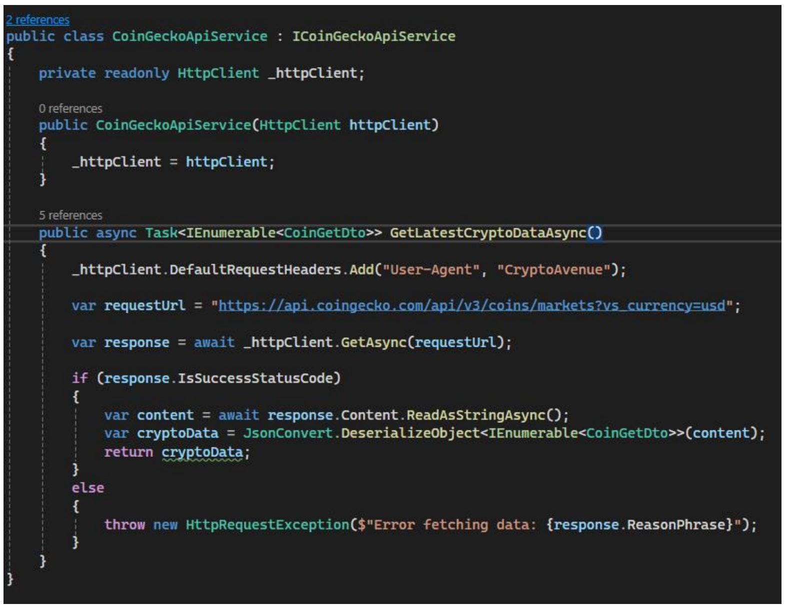
```
await _transactionRepository.Insert(transaction);
await _transactionRepository.SaveChanges();
```

3.4. CoinGecko API Service

In this subchapter I will showcase the process through which I retrieve data from the CoinGecko API and update my Coins database table based on the data received every 30 seconds, so users of my app can utilize the latest available data from the cryptocurrency markets.

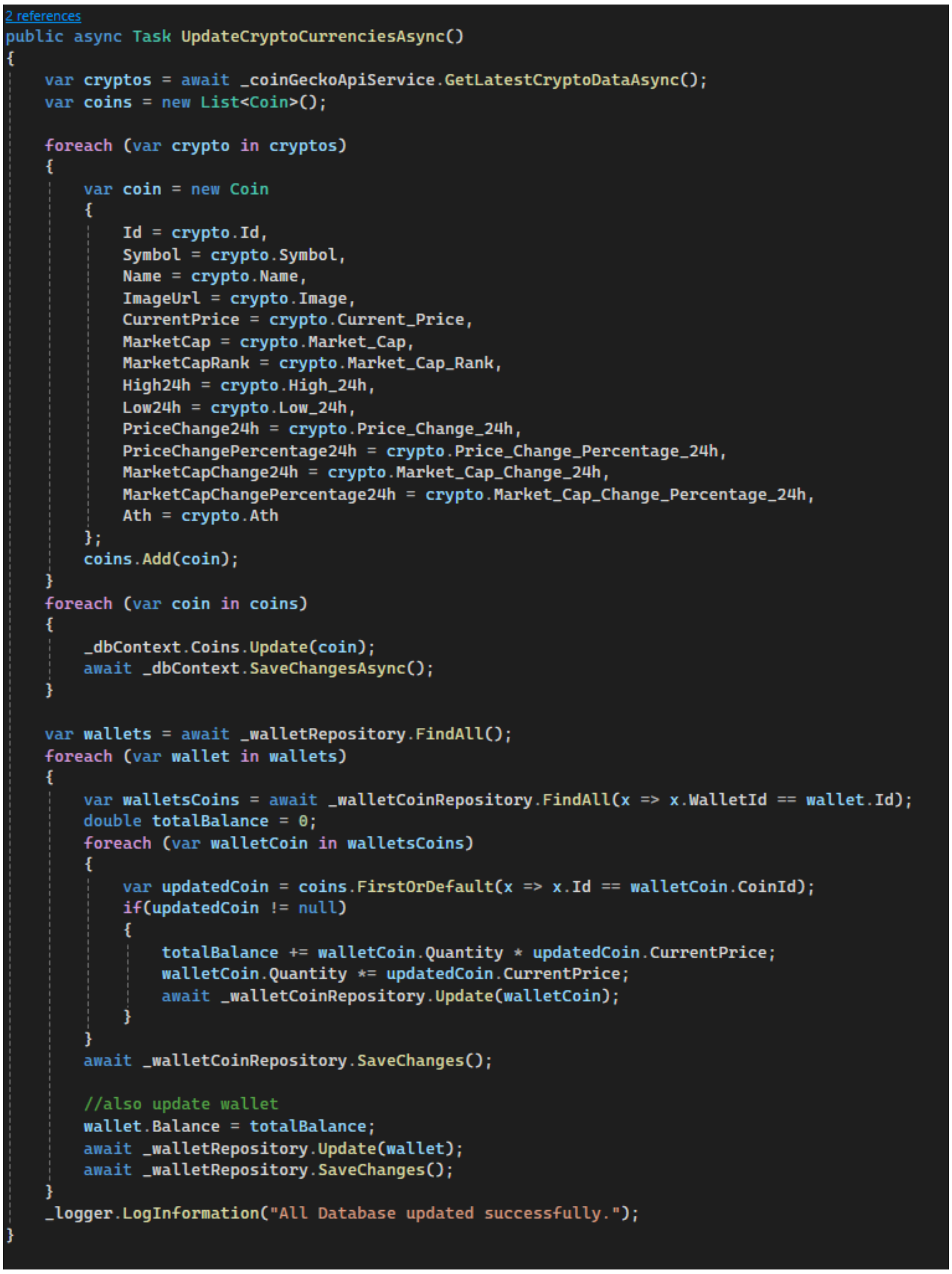
Retrieval of data from the CoinGecko API involves several steps. It is handled through a series of services that integrate with the API and update the database accordingly. The first step of the process is defining the ICoinGeckoApiService. This interface contains the signature of the GetLatestCryptoDataAsync() method, which returns a list of ‘CoinGetDto’s. The ‘CoinGetDto’ is a class I have defined that is mapped exactly to the response of the ‘/coins/markets’ endpoint of the CoinGecko API.

Next up is the creating the class ‘CoinGeckoApiService’ which implements the ‘ICoinGeckoApiService’ interface. This class is injected with an httpClient used to send the HTTP request and receive HTTP response from the CoinGecko API. After receiving the data from the API I used the ‘JsonConvert.DeserializeObject<IEnumerable<CoinGetDto>>()’ method to map the response from the API to my CoinGetDto entity, which I then return. Here is the full implementation of the class:



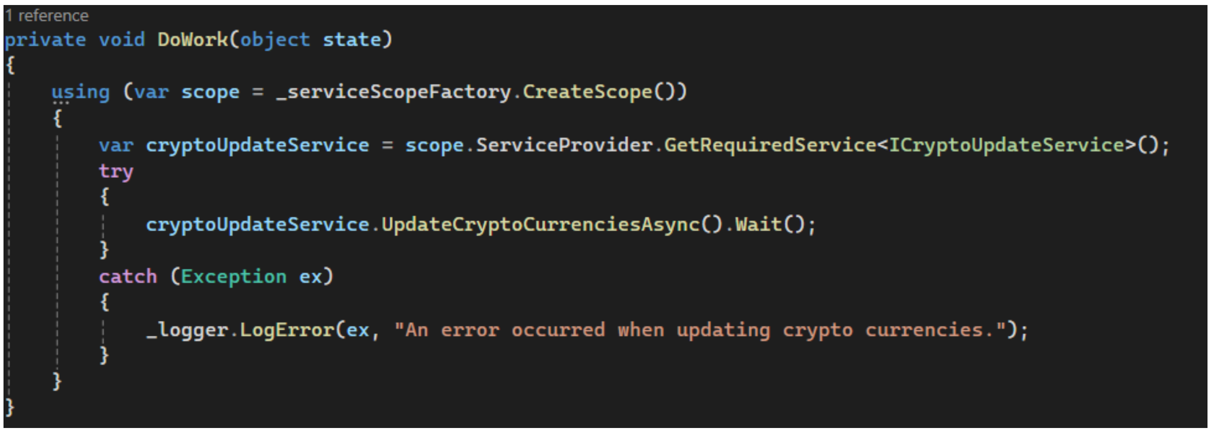
```
2 references
public class CoinGeckoApiService : ICoinGeckoApiService
{
    private readonly HttpClient _httpClient;

    0 references
    public CoinGeckoApiService(HttpClient httpClient)
    {
        _httpClient = httpClient;
    }

    5 references
    public async Task<IEnumerable<CoinGetDto>> GetLatestCryptoDataAsync()
    {
        _httpClient.DefaultRequestHeaders.Add("User-Agent", "CryptoAvenue");

        var requestUrl = "https://api.coingecko.com/api/v3/coins/markets?vs_currency=usd";

        var response = await _httpClient.GetAsync(requestUrl);

        if (response.IsSuccessStatusCode)
        {
            var content = await response.Content.ReadAsStringAsync();
            var cryptoData = JsonConvert.DeserializeObject<IEnumerable<CoinGetDto>>(content);
            return cryptoData;
        }
        else
        {
            throw new HttpRequestException($"Error fetching data: {response.ReasonPhrase}");
        }
    }
}
```

This service is then used CryptoUpdateService class, where I injected it into the constructor, alongside all the repositories for the entities which I want to update based on the information I receive from the service. In the main method of this service I first call the ‘GetLatestCryptoData()’ async method of the CoinGeckoApiService, which returns a list of CoinGetDtos. I then create a new List<Coin> for mapping the list from CoinGetDtos to Coins. After this is completed, I loop over the new List<Coin> and use the ‘_dbContext.Coins’ to update the cryptocurrencies in the database accordingly. After this, the other entities related to the Coin Ids found in the array are also updated, such as the entries in WalletCoins and Wallets. Here is the source code of the CryptoUpdateService class:



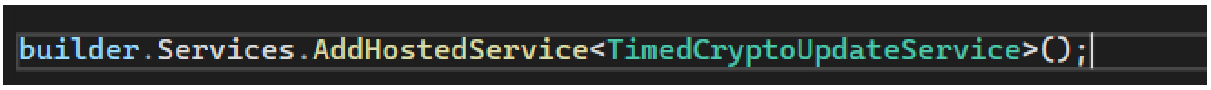
```
2 references
public async Task UpdateCryptoCurrenciesAsync()
{
    var cryptos = await _coinGeckoApiService.GetLatestCryptoDataAsync();
    var coins = new List<Coin>();

    foreach (var crypto in cryptos)
    {
        var coin = new Coin
        {
            Id = crypto.Id,
            Symbol = crypto.Symbol,
            Name = crypto.Name,
            ImageUrl = crypto.Image,
            CurrentPrice = crypto.Current_Price,
            MarketCap = crypto.Market_Cap,
            MarketCapRank = crypto.Market_Cap_Rank,
            High24h = crypto.High_24h,
            Low24h = crypto.Low_24h,
            PriceChange24h = crypto.Price_Change_24h,
            PriceChangePercentage24h = crypto.Price_Change_Percentage_24h,
            MarketCapChange24h = crypto.Market_Cap_Change_24h,
            MarketCapChangePercentage24h = crypto.Market_Cap_Change_Percentage_24h,
            Ath = crypto.Ath
        };
        coins.Add(coin);
    }
    foreach (var coin in coins)
    {
        _dbContext.Coins.Update(coin);
        await _dbContext.SaveChangesAsync();
    }

    var wallets = await _walletRepository.FindAll();
    foreach (var wallet in wallets)
    {
        var walletsCoins = await _walletCoinRepository.FindAll(x => x.WalletId == wallet.Id);
        double totalBalance = 0;
        foreach (var walletCoin in walletsCoins)
        {
            var updatedCoin = coins.FirstOrDefault(x => x.Id == walletCoin.CoinId);
            if(updatedCoin != null)
            {
                totalBalance += walletCoin.Quantity * updatedCoin.CurrentPrice;
                walletCoin.Quantity *= updatedCoin.CurrentPrice;
                await _walletCoinRepository.Update(walletCoin);
            }
        }
        await _walletCoinRepository.SaveChanges();

        //also update wallet
        wallet.Balance = totalBalance;
        await _walletRepository.Update(wallet);
        await _walletRepository.SaveChanges();
    }
    _logger.LogInformation("All Database updated successfully.");
}
```

The final step of this process is creating the 'TimedCryptoUpdateService' class that is responsible for setting up the 'CryptoUpdateService' to run every 30 seconds. It implements the 'IHostedService' and 'IDisposable' interfaces. "IHostedService' defines the methods for starting and stopping the background task and 'IServiceScopeFactory' provides a way for dependency injection in order to ensure that the services used in the update processes have the correct lifetime. The main functionalities are inside the DoWork() method which retrieves the 'CryptoUpdateService' from the provider and calls the UpdateCryptoCurrenciesAsync() method:



```
1 reference
private void DoWork(object state)
{
    using (var scope = _serviceScopeFactory.CreateScope())
    {
        var cryptoUpdateService = scope.ServiceProvider.GetRequiredService<ICryptoUpdateService>();
        try
        {
            cryptoUpdateService.UpdateCryptoCurrenciesAsync().Wait();
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "An error occurred when updating crypto currencies.");
        }
    }
}
```

After all these requirements are met, this timed updating service must be registered inside the 'Program.cs' file, like so:



```
builder.Services.AddHostedService<TimedCryptoUpdateService>();
```

3.5. Asynchronous Development in .NET Core

Asynchronous Development is a very good tool to use in most applications since it makes the app work much faster and more efficiently. Using the 'async' and 'await' operators in .NET is a great way to handle time-consuming actions, like input-output operations or database operations. This enables developers to create code that can execute this operation without causing the main thread to stall, therefore greatly increasing the time efficiency and responsiveness of the application.

The '**async'** keyword marks a method as asynchronous. It allows the use of the await operator inside of it but it requires the return type to be 'Task' or 'Task<T>'.

The '**await'** operator pauses the execution of the method in use until all the previously awaited tasks are completed.

The advantages of asynchronous programming include improved responsiveness, scalability and simplicity. Another positive factor of async programming is the fact that even though asynchronous methods throw exceptions, they are usually caught in the 'Task' object returned by the method they are used in.

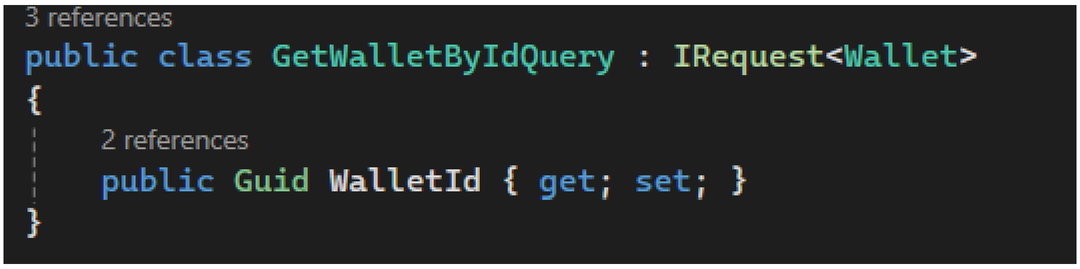
3.6. CQRS Pattern

In this subchapter I will talk about the Command Query Responsibility Segregation design pattern and how I implemented it into my application.

The CQRS pattern is an architectural design pattern that promotes the separation of operations that alter data from the operations that read data. The operations that read data are called queries, while the ones that modify data are called commands. This separation is designed to optimize performance, scalability and data security. Commands refer to the intention of executing a change in the system, such as deletes, updates or inserts, while queries are tailored to retrieving data without any changes in the system. This clean approach can simplify the architecture of complex applications, where it is very important to separate the business logic for data retrieval versus data modification.

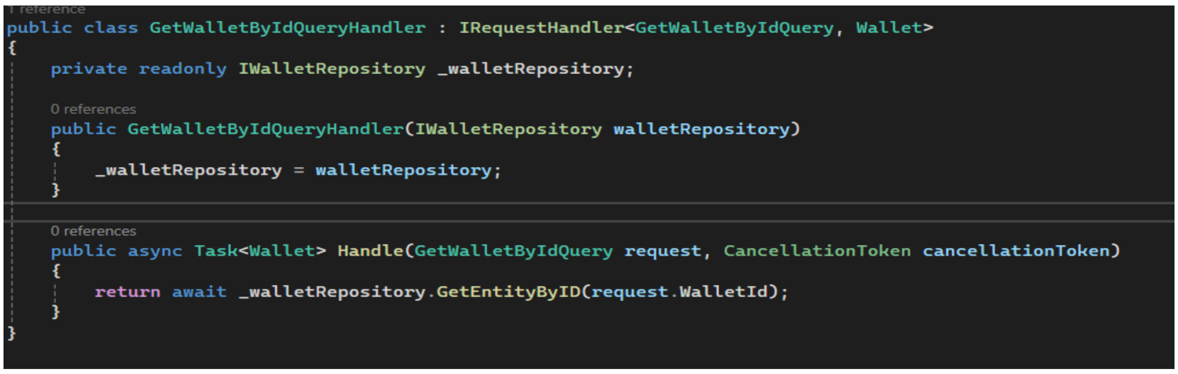
In .NET Core applications, the Command Query Responsibility Segregation design pattern is usually implemented through separate entities that handle commands and queries independently.

Both commands and queries are encapsulated in classes that inherit that implement the 'IRequest' interface. This interface is has an optional type <T>, which specifies the return type of the query or command. Below I will show you an example of a simple query from my application that returns a wallet from the database based on a WalletId provided as a property in the query class:



```
3 references
public class GetWalletByIdQuery : IRequest<Wallet>
{
    2 references
    public Guid WalletId { get; set; }
}
```

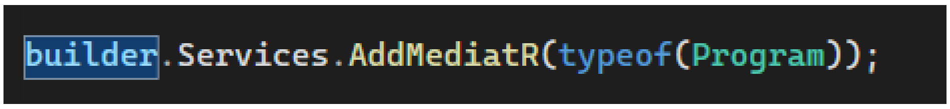
Each command or query has an attached handler. These handlers implement the 'IRequestHandler' interface of type <command&query_name, return_type>. These handlers contain the logic for retrieving or updating data based on the input command or query. They usually have a repository or database context injected into them. This is the handler of the 'GetWalletByIdQuery' from my codebase:



```
1 reference
public class GetWalletByIdQueryHandler : IRequestHandler<GetWalletByIdQuery, Wallet>
{
    private readonly IWalletRepository _walletRepository;

    0 references
    public GetWalletByIdQueryHandler(IWalletRepository walletRepository)
    {
        _walletRepository = walletRepository;
    }

    0 references
    public async Task<Wallet> Handle(GetWalletByIdQuery request, CancellationToken cancellationToken)
    {
        return await _walletRepository.GetEntityByID(request.WalletId);
    }
}
```

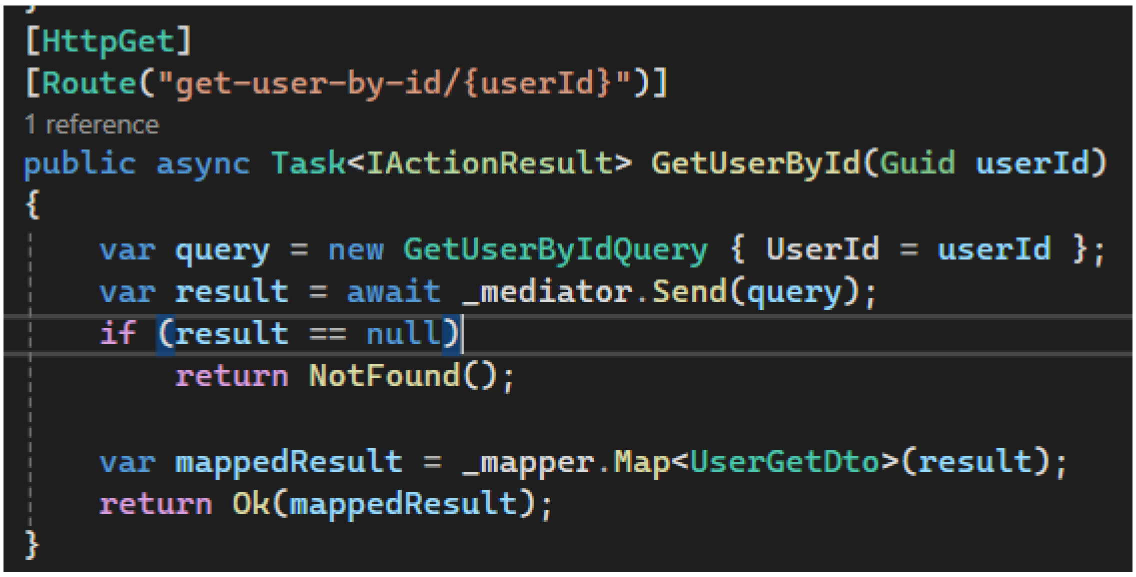
These commands and queries are then used in the controllers inside the Web API. In order for the controller to know which handler corresponds to the command or query it wants to use it is necessary to implement the MediatR pattern. MediatR is a popular library in .NET Core that simplifies the implementation of the CQRS pattern by acting as a mediator between command or queries and their respective handlers. This implies a much cleaner separation of the business logic from the request processing logic.

MediatR is setup in .NET Core projects by registering it in the ‘Program.cs’ file:



```
builder.Services.AddMediatR(typeof(Program));
```

The IMediator interface is then used in services and controllers to send commands or queries. I will provide you with an example of one of the methods in my ‘UsersController’ class from my application that uses the IMediator interface alongside a query and returns the result:



```
[HttpGet]
[Route("get-user-by-id/{userId}")]
1 reference
public async Task<IActionResult> GetUserById(Guid userId)
{
    var query = new GetUserByIdQuery { UserId = userId };
    var result = await _mediator.Send(query);
    if (result == null)
        return NotFound();

    var mappedResult = _mapper.Map<UserGetDto>(result);
    return Ok(mappedResult);
}
```

As you can clearly see, the ‘GetUserByIdQuery’ query object in instantianed by providing it with a user ID. Afterwards, the mediator sends the query asynchronously, which will lead to the handler of that specific query. The query then returns a result (in the form of a ‘User’ instance), which is then mapped to the ‘UserGetDto’ entity and returned with a status 200 (OK).

3.7. JWT Integration in .NET Core

Jason Web Tokens (JWT) represent a safe means of representing claims between 2 parties. Claims can be digitally signed and secured since they are encoded are JSON objects.

Whenever a certain person or device needs to be approved or verified, JWTs are one of the most popular ways to do it. They can be used in web applications to secure the interactions between the users and the server. Typically, JWTs consist of 3 main components:

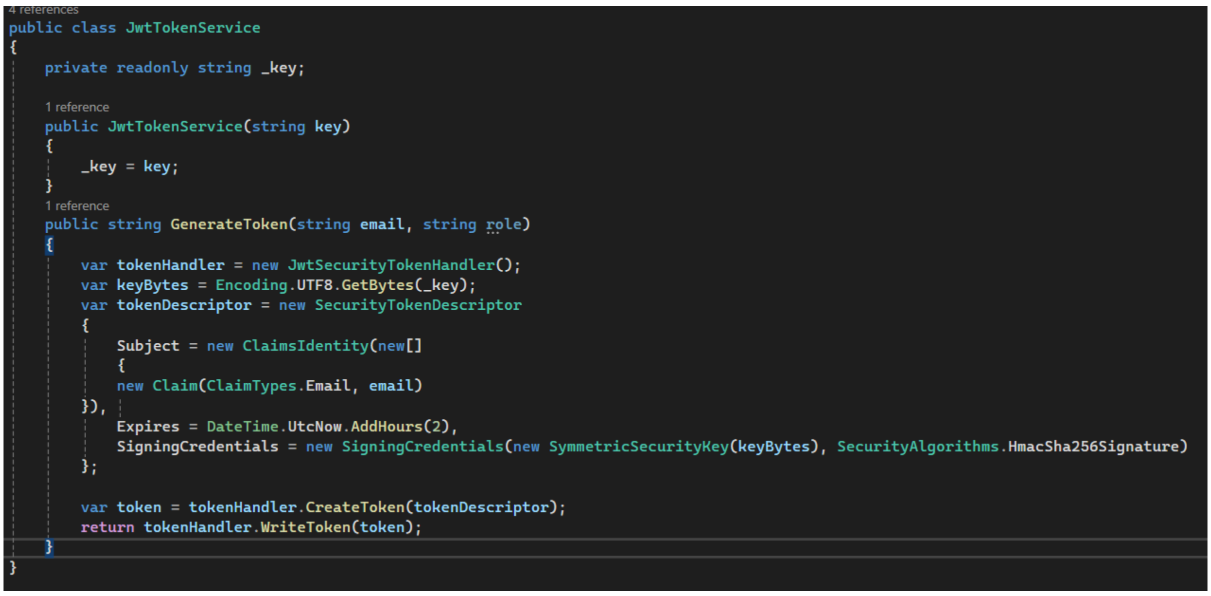
-header: containing the token type and the algorithm that is used for signing and encrypting (example: RSA, HMAC SHA256)

-payload: contains the claims of the request. These claims typically contain information such as user's data (email, password) and additional information like the role they posses in the scope of the application

-signature: used to verify that the sender who sends the request is in fact who he is and that the message hasn't undergone any changes along the way

In .NET Core, JWTs are most often used for securing Web APIs. .NET provides support for JWT implementation and configuration through its middleware components, allowing developers to easily implement these features.

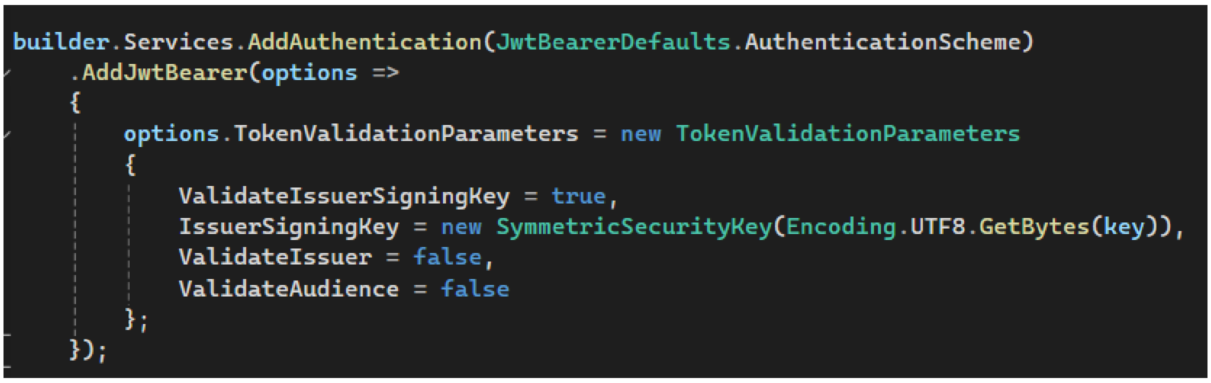
In my application, I have created a 'JwtTokenService' class that is responsible for generating the tokens. My service takes a user's email address and creates a new token with an expiration time (2 hours in this case) and a signature. Here is the source code of my service:



```
4 references
public class JwtTokenService
{
    private readonly string _key;

    1 reference
    public JwtTokenService(string key)
    {
        _key = key;
    }
    1 reference
    public string GenerateToken(string email, string role)
    {
        var tokenHandler = new JwtSecurityTokenHandler();
        var keyBytes = Encoding.UTF8.GetBytes(_key);
        var tokenDescriptor = new SecurityTokenDescriptor
        {
            Subject = new ClaimsIdentity(new[]
            {
            new Claim(ClaimTypes.Email, email)
        }),
            Expires = DateTime.UtcNow.AddHours(2),
            SigningCredentials = new SigningCredentials(new SymmetricSecurityKey(keyBytes), SecurityAlgorithms.HmacSha256Signature)
        };

        var token = tokenHandler.CreateToken(tokenDescriptor);
        return tokenHandler.WriteToken(token);
    }
}
```

The 'ClaimsIdentity' object includes the user's email address inside the token, after which the token is set to expire after a period of 2 hours and then it is signed using the HMAC SHA256 algorithm to secure its integrity.

After this, the ‘JwtTokenService’ is register in the ‘Program.cs’ file. Also, in this file I specified how the incoming tokens should be verified, using the ‘AddJwtBearer’:

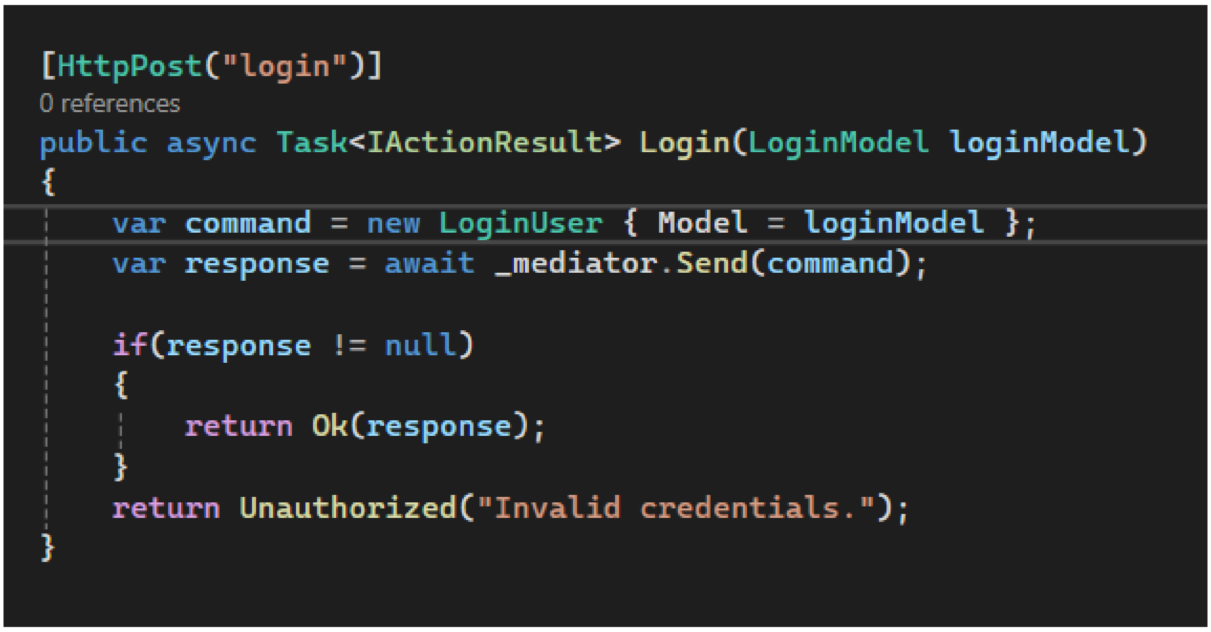


```
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuerSigningKey = true,
            IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(key)),
            ValidateIssuer = false,
            ValidateAudience = false
        };
    });
```

In the method above you can see that the token’s signature is validated using a symmetric security key and the audience and issuer validation options are set to false, since there is no need to validate the intended recipients.

Next, I have created a ‘LoginModel’ class, which represent a data transfer object for the purpose of authenticating users. This class contains only 2 properties, specifically the email and the password, since these are the only 2 credentials a user needs to log in to CryptoAvenue.

The authentication process is then handled in the ‘AuthController’ class, where a new ‘LoginUser’ command is created, that takes as parameter a login model, which is of type ‘LoginModel’. The MediatR is then used to send the command and fetch the response. If the response is good then it will be sent to the front-end. The response contains the user’s email address and the token key. Source code of the ‘LoginUser()’ method inside of the ‘AuthController’:



```
[HttpPost("login")]
0 references
public async Task<IActionResult> Login(LoginModel loginModel)
{
    var command = new LoginUser { Model = loginModel };
    var response = await _mediator.Send(command);

    if(response != null)
    {
        return Ok(response);
    }
    return Unauthorized("Invalid credentials.");
}
```