CZECH TECHNICAL UNIVERSITY IN PRAGUE

Faculty of Electrical Engineering

Finance Management Application
(Backend)

Author: Ivan Sedakov
Study programme: Open Informatics
Branch of study / Specialization: (BPOI318) Software

Thesis supervisor: Ing. Šebek Jiří

May 2022

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení:	**Sedakov**	Jméno: **Ivan**	Osobní číslo: **492282**
Fakulta/ústav:	**Fakulta elektrotechnická**		
Zadávající katedra/ústav:	**Katedra počítačů**		
Studijní program:	**Otevřená informatika**		
Specializace:	**Software**		

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Správa osobních financí

Název bakalářské práce anglicky:

Personal finance management

Pokyny pro vypracování:

Create the backend support for the web application, that will help people solve their finance problems and make money management easier.
1) Analyze existing similar applications and problems with financial management in the modern world
2) Analyze technologies, platforms and programming languages for backend web development
3) Create Authentication and Authorization system with JWT tokens
4) Focus on application security
5) Focus on the use of modern technologies
6) Separate application logic and implementation into services that can be run in parallel
7) Design and implement backend support for a web application
8) Test the application with benchmarks, unit tests and process tests.

Seznam doporučené literatury:

1. Pressman R. S.: Software Engineering
2. John Skeet.: C# in Depth
3. Robert C. Martin: Clean Code. A Handbook of Agile Software Craftsmanship
4. Erich G., John V., Richard H., Ralph J.: Design Patterns: Elements of Reusable Object-Oriented Software

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Jiří Šebek   kabinet výuky informatiky   FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **11.02.2022**   Termín odevzdání bakalářské práce: **20.05.2022**

Platnost zadání bakalářské práce: **30.09.2023**

Ing. Jiří Šebek podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

.

Datum převzetí zadání   Podpis studenta

Author statement for undergraduate thesis

I declare that the presented wort was developed independently and that I have listed all sources of information used within it in accordance with the methodical instruction for observing the ethical principles in the preparation of university theses.

Prague, date …………………………..   Signature …………………………

Abstract

The main goal of the work is to implement backend support for a web application that helps manage finances. At first, were analyzed existing applications for similar purposes and were chosen the best features and options. Then were represented application functions, implementation, and technologies. Finally, the result of the work and gained experience were analyzed.

Keywords: web application, backend app, .net, C#, finances, income, expense

Abstrakt

Hlavním cílem práce je implementace backendové podpory webové aplikace, která pomáhá spravovat finance. Nejprve byly analyzovány existující aplikace pro podobné účely a byly vybrány nejlepší vlastnosti a možnosti. Dále byly zastoupeny aplikační funkce, implementace a technologie. Nakonec byl analyzován výsledek práce a získané zkušenosti.

Keywords: webová aplikace, backendová aplikace, .net, C#, finance, příjmy, výdaje

Contents

1. Introduction

1.1 Motivation

Financial management has always been one of the most important things in human life. [1, 13] Failure to pay enough attention to own finances leads to disastrous consequences and can even destroy the future. Most people need to control incoming and outgoing finances, have an idea of their debts, monetary goals, and statistics, plan for the future, and manage the present. A financial management application can help deal with all these troubles and make people's lives easier.

Of course, such applications already exist in our world, but programming has no boundaries and limits for improvement. Humanity goes forward and always needs modernization. Developers update user interfaces, improve the performance and quality of resources, and apply new features and technologies. Our application has been developed using the most modern technologies and methodologies and includes the best features and capabilities of existing financial management applications.

1.2 Aims

The main part of such an application is the implementation of the backend since it contains the main logic of the program and connects all the parts and services of the application. Therefore, it is this part that contains the most bugs, errors, and dark places and most of all needs detailed analysis, a clear structure, strict rules, competent documentation, and should have the best performance. [5]. Once correctly implemented, the backend can be used with almost all types of modern applications such as web, mobile, and desktop applications.

This work is devoted to the backend part of the application. The following chapters provide an analysis of existing financial management applications, the requirements and use cases for our application, the implementation, and a list of technologies used.

2. Research

The chapter contains an analysis of existing projects on the market with similar functions and purposes, as well as an analysis of world backend technologies, databases, and a comparison of various types of architectures.

2.1 Analysis of similar existing apps

Three similar applications were analyzed for their functionality, user interfaces, and main features.

2.1.1. Spendee

The Spendee [16] is a finance management application, which was developed by a Czech startup company Spendee Ltd. Suitable functions for our application:

- Ingoing and outgoing transactions.
- Statistics and Graphs.
- Transaction Categories
- Accounts with different currencies.
- Import/Export of transactions.
- Budgets (constant payments during periods).

Premium Subscription functions:

- Auto Categories.
- Connection to the bank account.
- An unlimited number of accounts.

The Spendee has a user-friendly interface, and mobile and desktop applications, but a small list of features. Without a premium subscription, the app is too trivial.

2.1.2. CoinKeeper

The CoinKeeper [15] is a finance management application. It was developed by a Russian IT company. Suitable functions for our application:

- Ingoing and outgoing transactions.
- Accounts with different currencies.
- Transaction Categories.
- Statistics and Graphs.
- Import/Export of transactions.

Premium Subscription functions:

- Joint finance management.
- Disable ads.
- Extended Statistics and Categories

In comparison to Spendee, all the transactions should be created manually. The Spendee has a “Connection to the bank account” function in the premium version. The user interface is not bad, but some sections were made completely non-obvious and don’t have windows help. Also, without a Premium subscription, there is the ad, which is very inconvenient for this type of application.

2.1.3. Wallet

The Wallet [14] is a finance management application, which was developed by a Czech startup company BudgetBakers. Suitable functions for our application:

- Ingoing and outgoing transactions.
- Accounts with different currencies.
- Transaction Categories
- Budgets (constant payments during periods).
- Shopping lists.
- Debts.
- Goals.
- Import/Export of transactions.

Premium Subscription functions:

- Connection to the bank account
- Insightful reports.
- Advanced budgets, Categories.
- An unlimited number of accounts.

In comparison to CoinKeeper and The Spendee, Wallet has many more features in both versions. The interface is user-friendly and very obvious.

Feature	Spendee	CoinKeeper	Wallet
Accounts	+	+	+
Transactions and Categories	+	+	+
Statistics and Graphs	+	+	+
Bank synchronization	+ (Premium)	-	+ (Premium)
Goals	-	-	+
Debts	-	-	+
Without Ads	+	+ (Premium)	+
Rating	2	3	1

Table 1: Comparison Table

After analyzing all applications, The Wallet was recognized as the best, because it has a huge number of interesting features and has everything that users might need in the premium version. The Spendee is a balanced choice, it has all the basic options and a user-friendly and practical interface. CoinKeeper was chosen as the worst because it doesn't have any interesting features and has an ad.

2.2 Analysis of technologies

The most interesting frameworks and technologies for me were analyzed for choosing the best one for this project.

2.2.1 Backend

- Gin

  Gin is a high-performance micro-framework that can be used to build web applications and microservices. It contains a set of commonly used functionalities, e.g. routing, middleware support, rendering, that reduce boilerplate code and make writing web applications simpler [17]. Gin with Golang language is a modern and new technology compared to Spring Boot and ASP.NET. But it already has an active developer community, lots of tutorials, and comprehensive tools. Go is very fast and comparable in performance to C++, making it the fastest language and a great choice for modern web development.
- Spring Boot

  Spring Boot is an open-source micro-framework used to build Spring applications with the help of microservices [18]. Spring Boot is based on Java and is the oldest framework on our list. It has a huge community and lots of material and courses. Java is the most supported language on the CVUT and one of the most used languages in the world from our list.

- .NET

  .NET is a developer platform made up of tools, programming languages, and libraries for building many different types of applications. ASP.NET extends the .NET developer platform with tools and libraries specifically for building web apps [19]. .Net was developed by Microsoft and integrates easily with services such as Azure. The framework uses C# programming language, which has an exceptionally expressive syntax and provides good security and performance. .Net has a huge community, is very readable and minimizes version conflicts, is fast enough for web development, and is backed by one of the biggest and strongest companies in the world.

.NET and Spring Boot are both popular web frameworks that have everything a developer might need. And I think the main reason why someone prefers one or the other depends on the language of that framework. Those who prefer C# will choose .NET, lovers of Java or Kotlin will use Spring Boot.

I prefer C# and I feel uncomfortable with Java. The main reason is syntax and used libraries. I prefer attributes over annotations, Entity Framework with LINQ over Spring Data JPA, and other things that are specific for everyone. The next important point is performance (request per second, RAM usage), where Java has always been the worst choice. Performance Tests results are shown in Figure 1 and Figure 2.

Gin with Golang looks like a very interesting and great choice, but I have no experience with it and my project doesn't need such good performance as Golang provides. I often use C# and should choose a language that I love and know for this project.

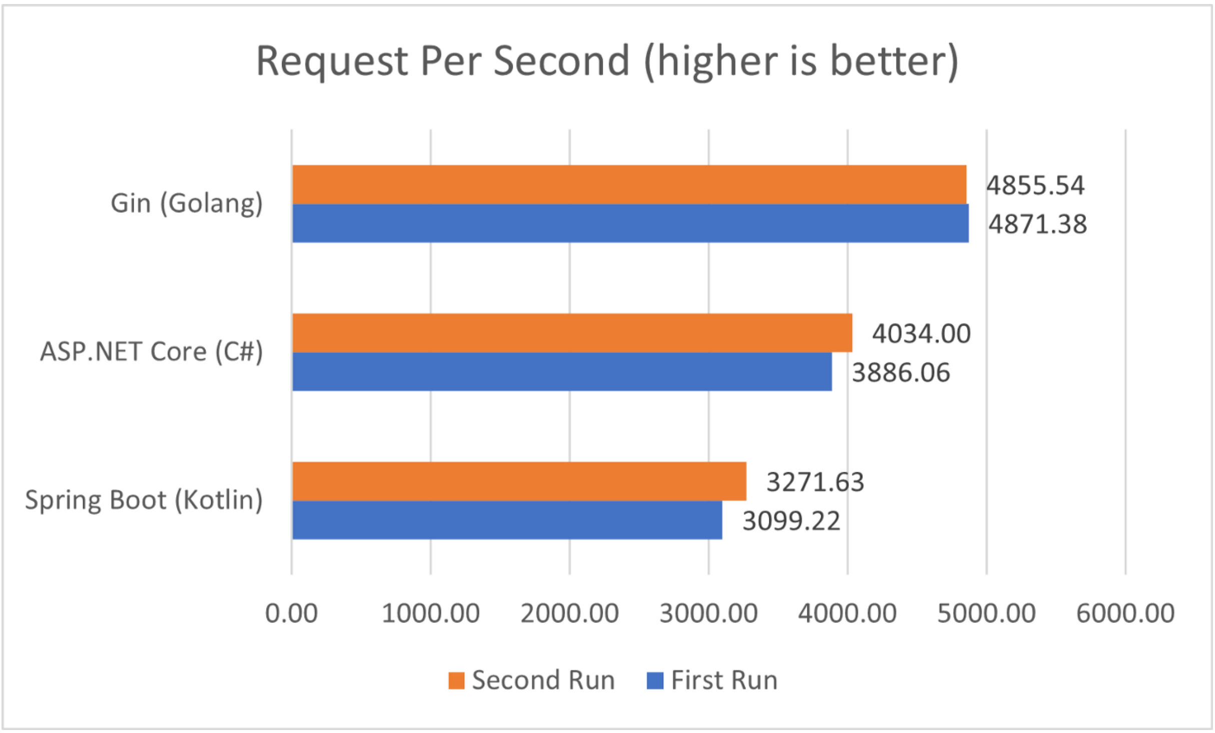


Figure 1: Request Per Second, [20]

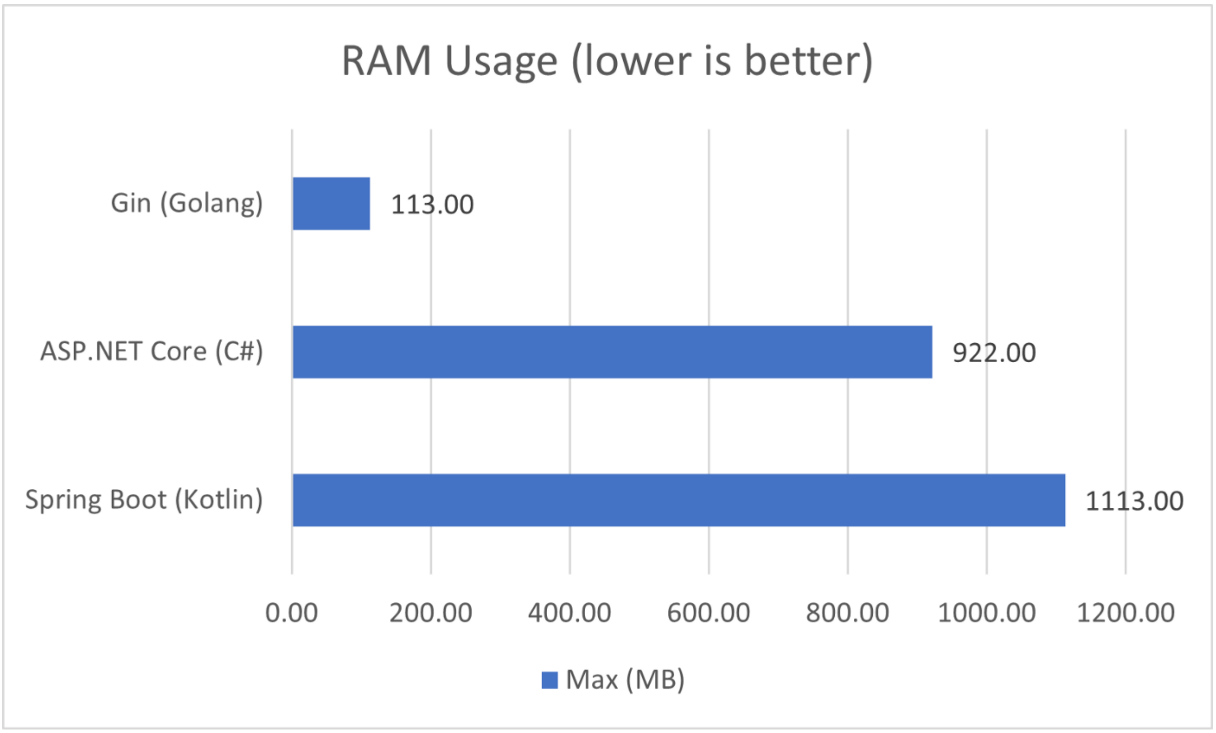


Figure 2: RAM Usage, [20]

2.2.2 Database

Difference between SQL and NoSQL

SQL databases are primarily called as Relational Databases (RDBMS); whereas NoSQL databases are primarily called as non-relational or distributed databases [21].

SQL	NoSQL
These databases have fixed or static or predefined schema	They have a dynamic schema
These databases are not suited for hierarchical data storage.	These databases are best suited for hierarchical data storage.
These databases are best suited for complex queries	These databases are not so good for complex queries
Vertically Scalable	Horizontally scalable
Follows ACID property	Follows CAP

Table 2: SQL VS NoSQL, [21]

The App uses a relational SQL database because of:

- Data is easily structured into categories.
- Data is consistent in input, and meaning, and easy to navigate.
- Relationships can be easily defined between data points.
- Complex Queries: update transactions inside a group by checking group rights and user permissions.
- Vertical Scalable: increasing the power of CPU and RAM to an existing machine instead of adding more machine resources is more appropriate.
- ACID property.

Relational Databases

The most widely implemented relational databases are MySQL, PostgreSQL, and SQLite [37]. They were compared on the advantages and disadvantages of choosing the best one for your project and goals. [Table 3] shows the most important pros and cons of these databases.

PostgreSQL looks like the best choice for our project. It has free hosting, which is necessary because the frontend part is implemented independently. We do not need very high speed, we store data in the cache and use cache preloaders as much as possible. The application will have many users writing data to the database at the same time, use complex operations, and deal with large amounts of data.

	MySQL	PostgreSQL	SQLite
Advantages	Security	SQL compliance	Testing
	Speed	Data integrity is important	
		Complex operations	
		Free Hosting	
Disadvantages	Slowed development	Speed	Working with lots of data
	Licensing and proprietary features		High write volumes
	Temporary free hosting		Temporary free hosting

Table 3: Comparison of relational databases

2.2.3. Architecture

- Monolithic Architecture

  A monolithic architecture is the traditional unified model for the design of a software program. Monolithic, in this context, means composed all in one piece. Monolithic software is designed to be self-contained; components of the program are interconnected and interdependent rather than loosely coupled as is the case with modular software programs. In a tightly-coupled architecture, each component and its associated components must be present in order for code to be executed or compiled. Monolithic programs typically have better throughput than modular approaches, such as the microservice architecture and they can be easier to test and debug because with fewer elements there are fewer variables that come into play [24].

- Microservices Architecture

  Microservices allow a large application to be separated into smaller independent parts, with each part having its own realm of responsibility. To serve a single user request, a microservices-based application can call on many internal microservices to compose its response. Typically, microservices are used to speed up application development. [25]. Modular architectures reduce the risk that a change made within one element will create unanticipated changes within other elements, because modules are relatively independent. Modular programs also lend themselves to iterative processes more readily than monolithic programs. [24]. The main advantage over Monolithic Architecture is that the modified program component does not affect other parts of the program.

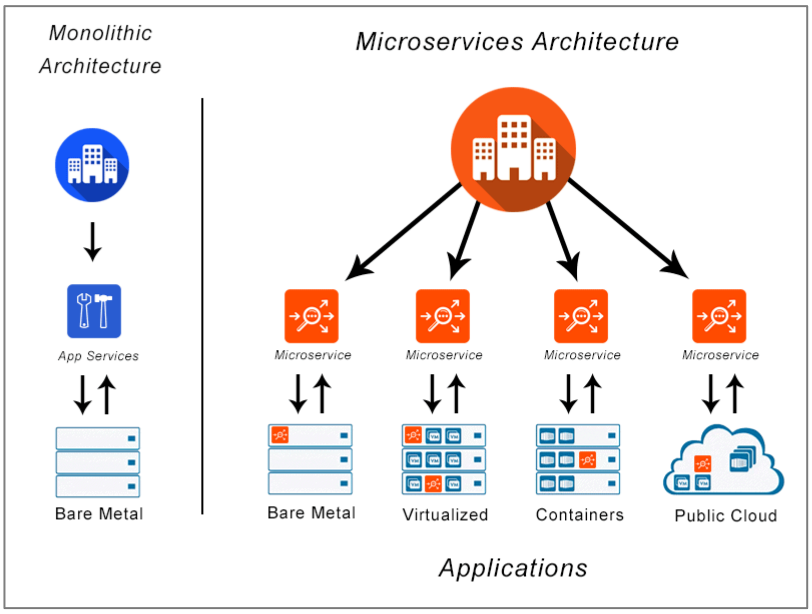


Figure 3: Monolithic Architecture vs Microservices Architecture, [23]

The Application uses Monolithic Architecture, which is the best choice for the following reasons [22]:

- Small team
- The simple application
- No microservices expertise
- Quick launch (quick demonstration)

3. Analysis

The chapter defines functional and non-functional requirements and contains information about application use cases. It was determined after analyzing existing applications and feedback about them from their users.

3.1. Requirements

3.1.1 Functional

FR-1: Swagger API Documentation

FR-2: Authentication, Authorization, Registration

FR-3: Accounts

FR-4: Transactions, Subscriptions, Debts, Goals

FR-5: Groups, Roles, Rights

FR-6: Notifications about Subscriptions and Debts Expired

FR-7: Dashboard

FR-8: Statistics and Graphs

3.1.2 Non-Functional

NP-1: Backend: .Net 6, C#, PostgreSQL

NP-2: REST API

NP-3: Monolithic Architecture

NP-4: Easy Vue.js Client App connection for Monolithic Architecture

NP-5: Repository and Unit of Work Patterns for Data Access Layer.

NP-6: Cache as much as possible, Cache Preloader

NP-7: Message Service

3.2. Use Cases

The use cases were created after a detailed analysis of an existing application and reading comments from their users about improving the functionality of the application. Apart from the initial options, it also contains enhancements such as subscriptions, goals, and debts. These use cases provide the application with all the necessary functionality that a user might need to manage their finances.

UC-1: Transaction, Subscription, Debt, Goal entities

- CRUD Operations
- GET/DELETE by User Id
- History Mode for Debts, Subscriptions, Goals (shows old and not active)
- Pagination support for Transactions and History Mode

UC-2: User Registration and Authorization logic

UC-3: Group Operations

- Users can create Group with rights and roles
- Users with required group rights can add/delete other users to the Group
- Users with required group rights can change other users rights and roles
- Users with required group rights can add/delete transactions and subscriptions from their own accounts to the group’s accounts

UC-4: Settings

- Users can change their personal data

UC-5: Notifications

- Users can GET their own notifications Count and all their notifications
- Users can change the read status of their own notifications and delete them.

UC-6: Categories

- Users can create/delete own categories
- User can add/update/delete category from transaction

UC-7: Dashboard

- Get Last Transactions, Subscriptions, Debts, Goals
- Statistic Data

[Figure 4-13] contains Use Case Models.

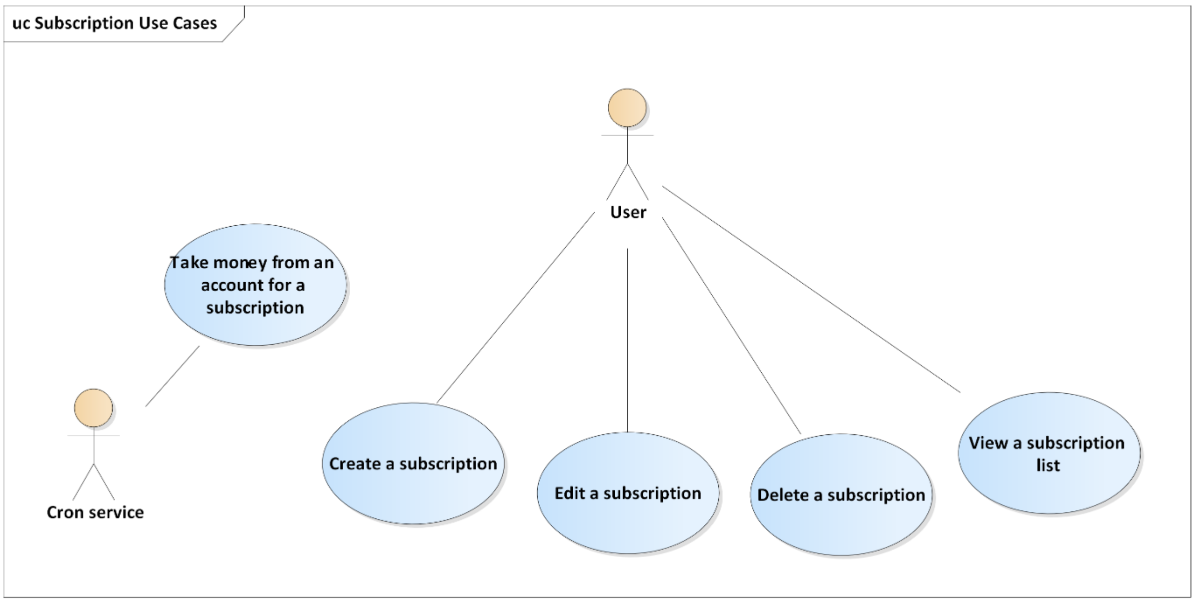


Figure 4: Subscription Use Cases [Tikhon Zaikin]

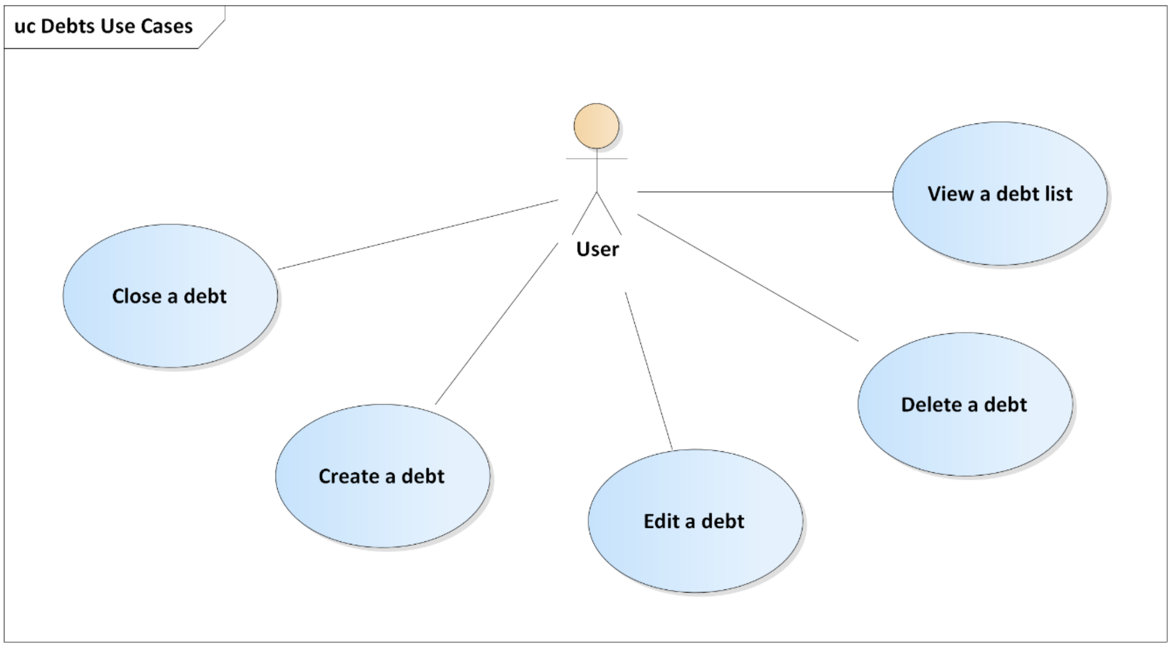


Figure 5: Debts Use Cases [Tikhon Zaikin]

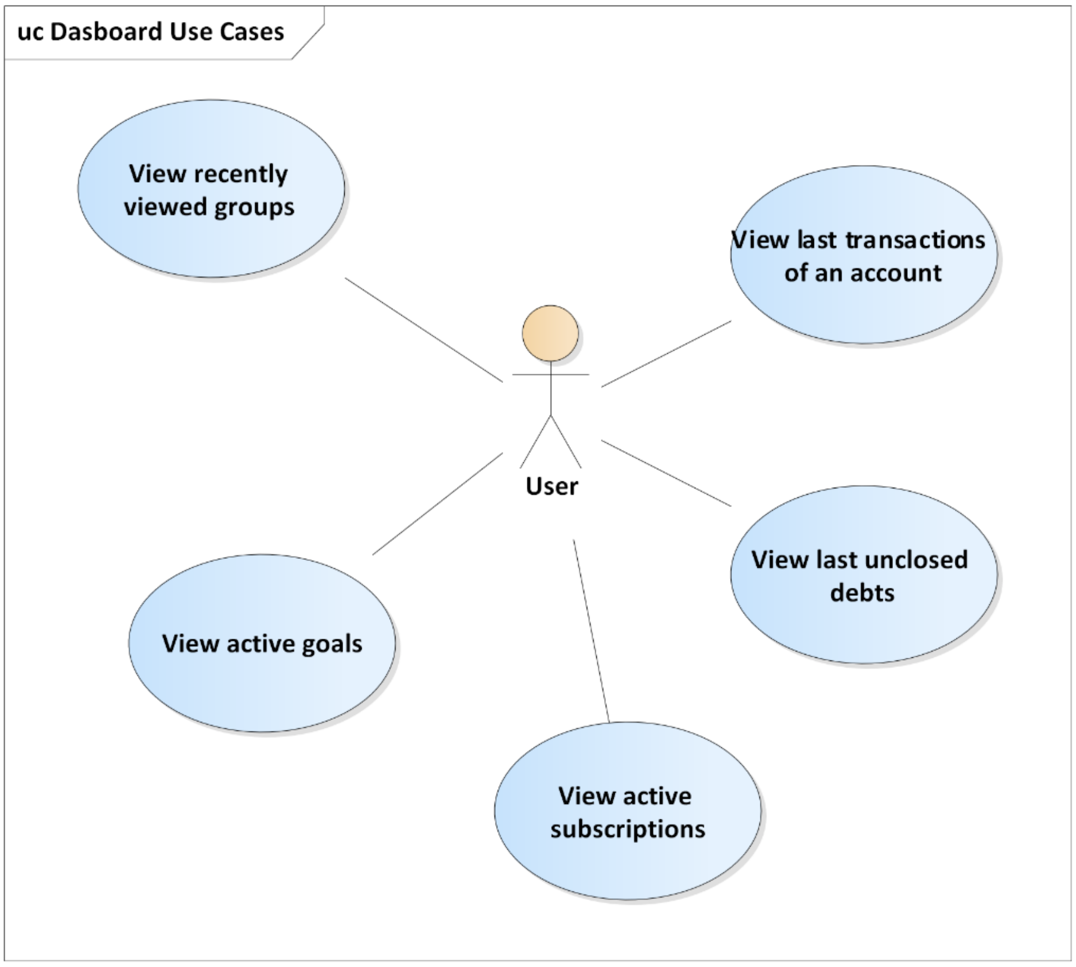

uc Dasboard Use Cases
View recently viewed groups
View last transactions of an account
User
View last unclosed debts
View active goals
View active subscriptions

Figure 6: Dashboard Use Cases [Tikhon Zaikin]

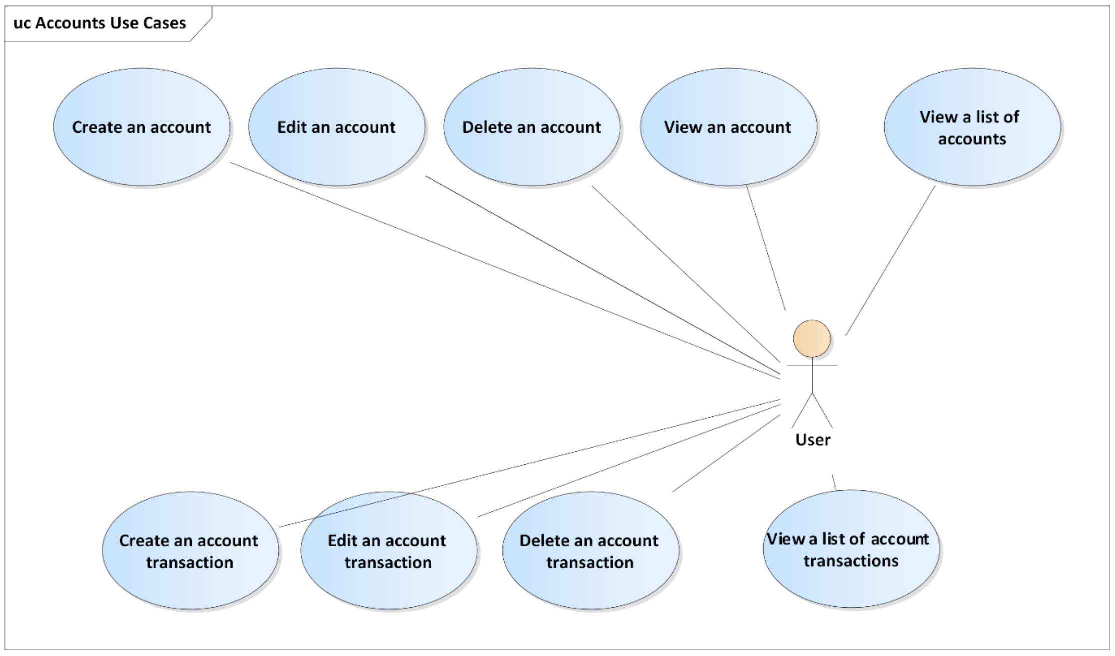

uc Accounts Use Cases
Create an account
Edit an account
Delete an account
View an account
View a list of accounts
User
Create an account transaction
Edit an account transaction
Delete an account transaction
View a list of account transactions

Figure 7: Account Use Cases [Tikhon Zaikin]

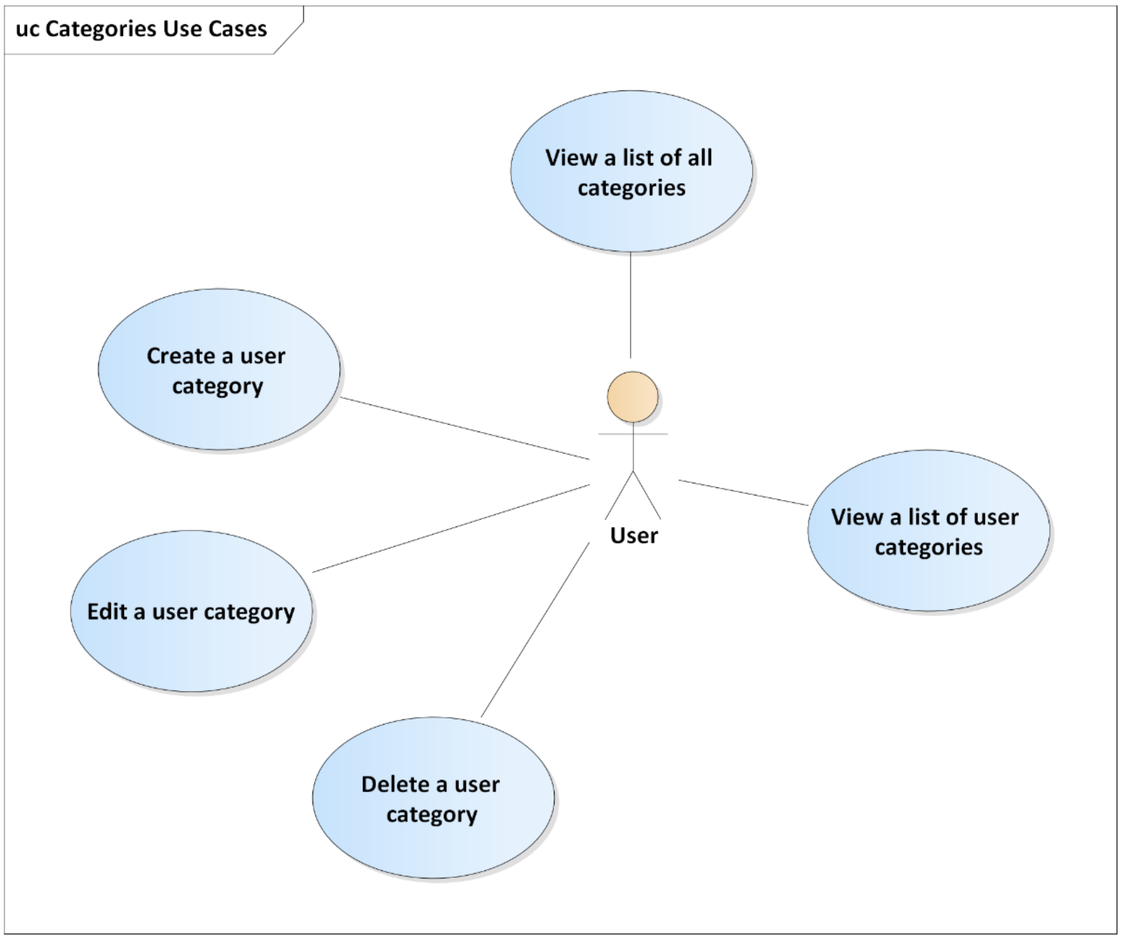


Figure 8: Categories Use Cases [Tikhon Zaikin]

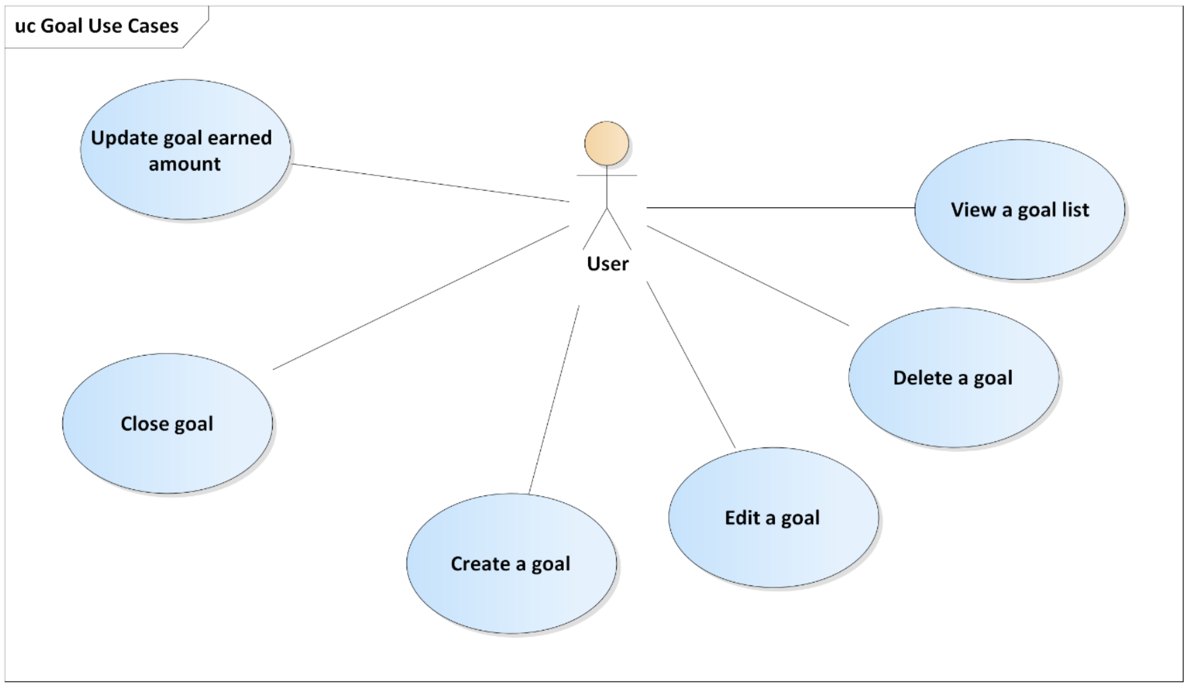


Figure 9: Goal Use Cases [Tikhon Zaikin]

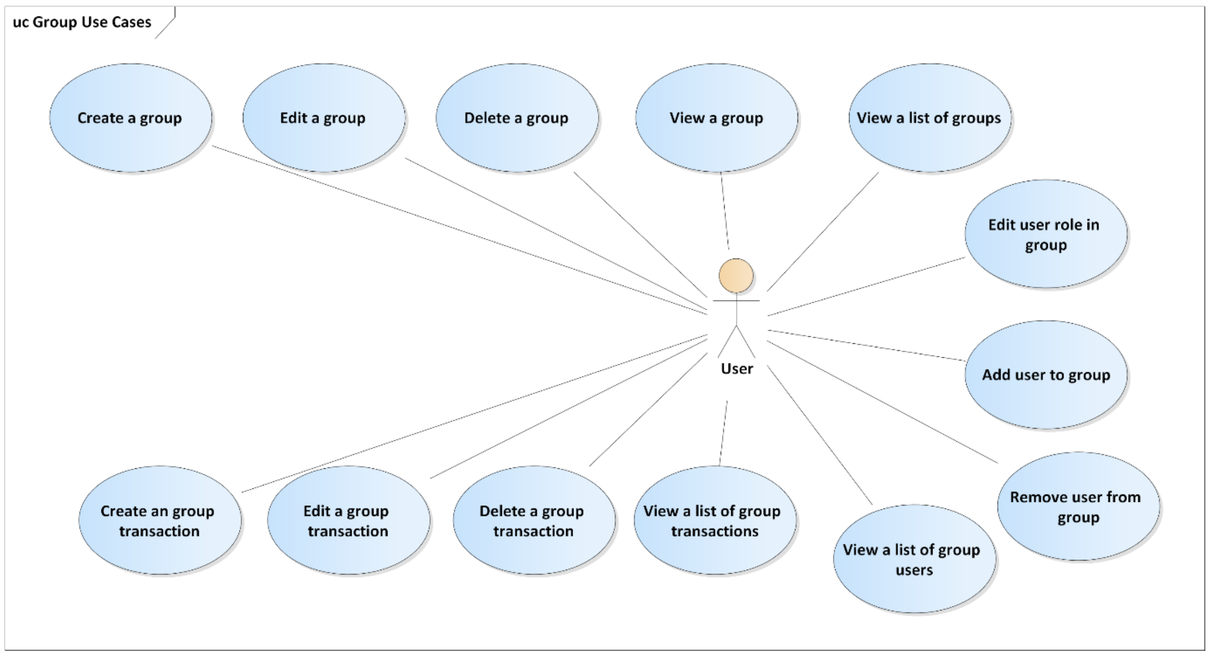


Figure 11: Group Use Cases [Tikhon Zaikin]

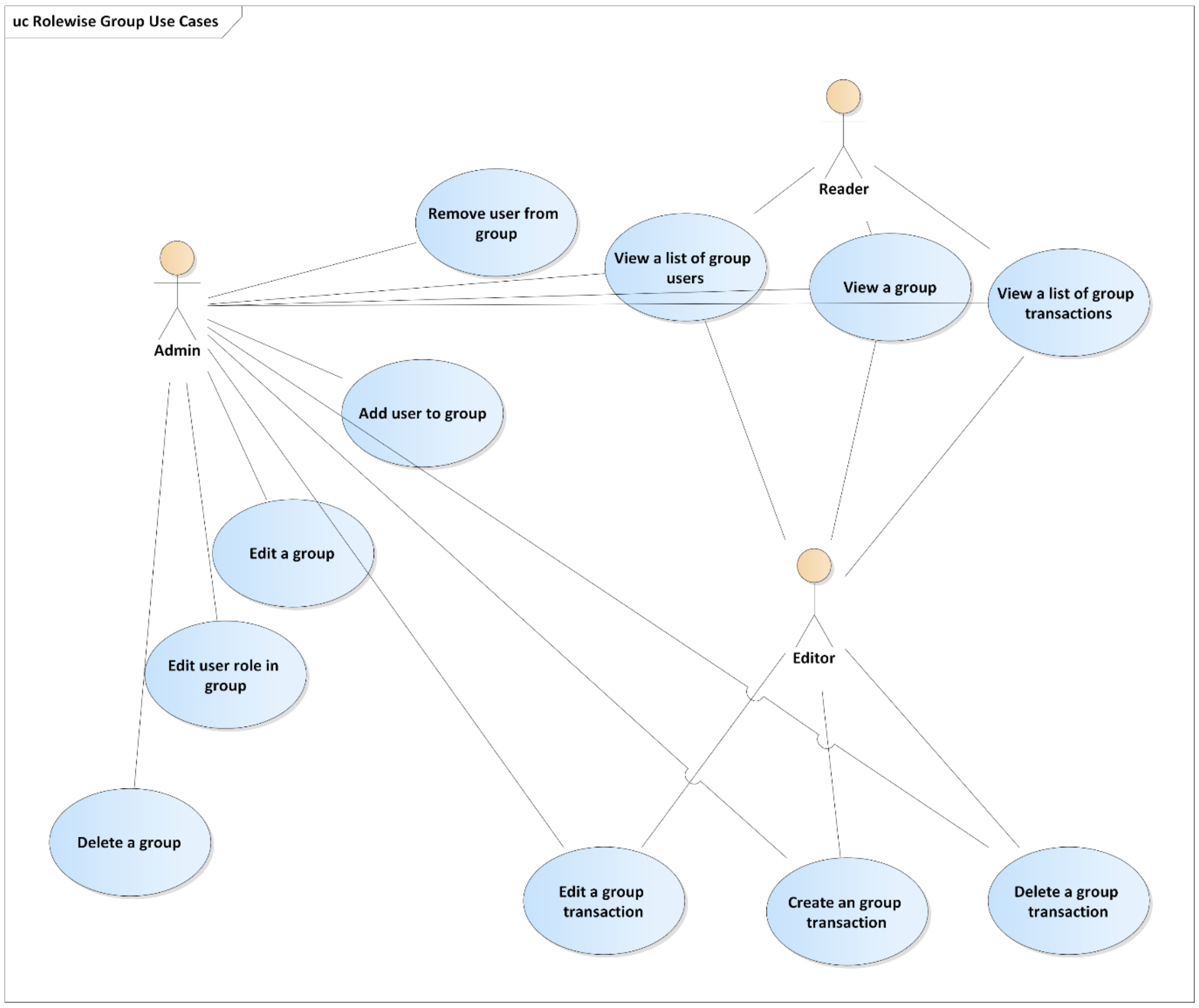


Figure 10: Rolewise Group Use Cases [Tikhon Zaikin]

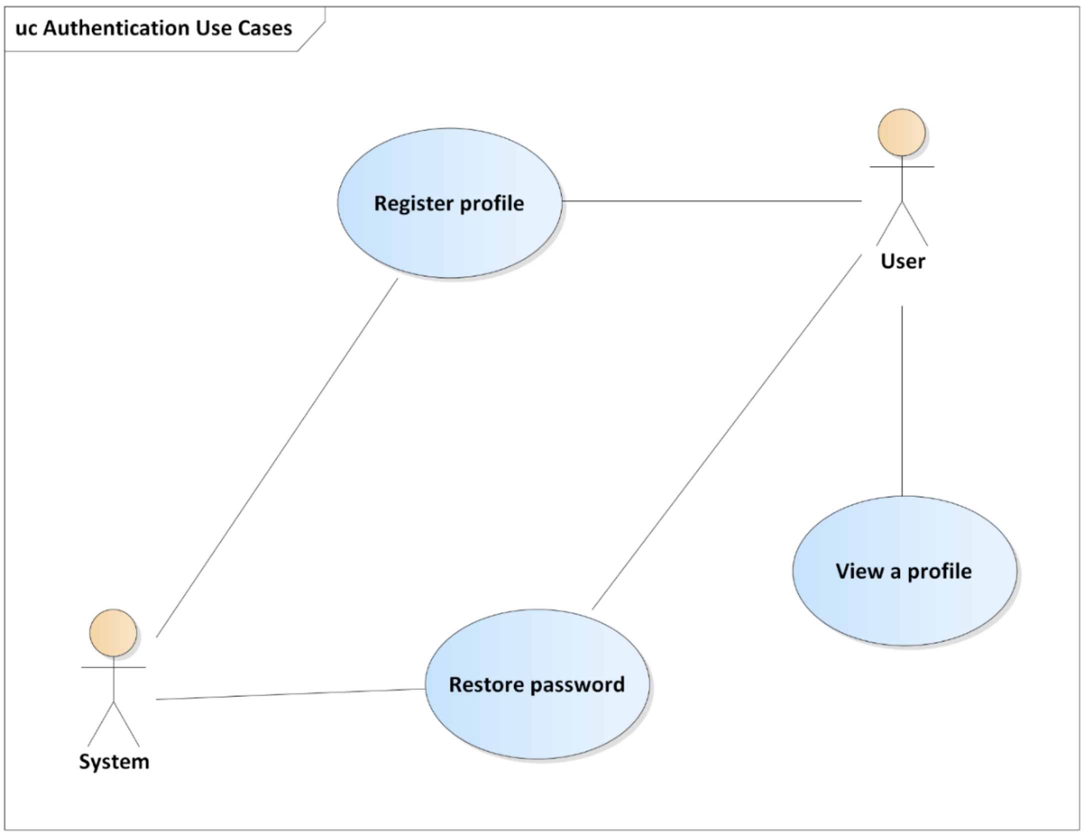


Figure 12: Systemwide Use Cases [Tikhon Zaikin]

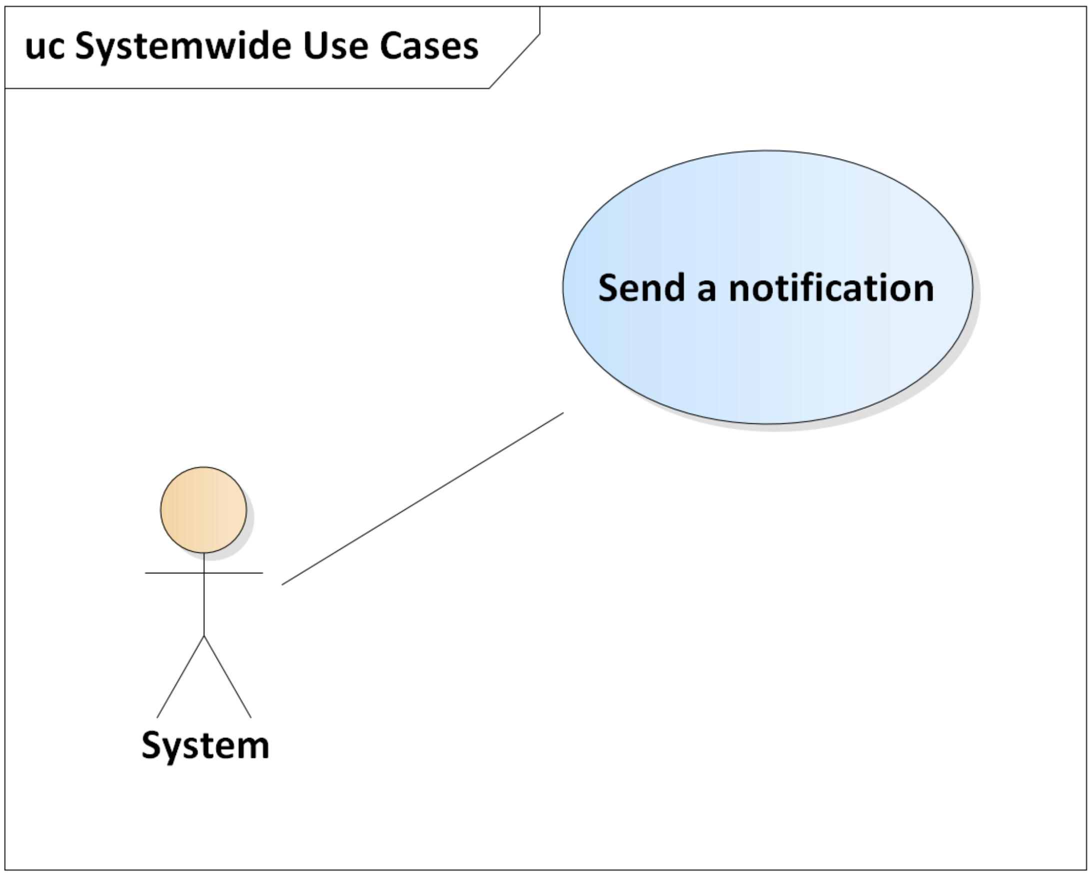


Figure 13: Authentication Use Cases [Tikhon Zaikin]

4. Design

The chapter provides information about the application architecture, solution structure, and entity models. It contains a component diagram and a business domain diagram, talks about each layer of the application, and shows the request path from the API controller to the database.

4.1. Architecture

The application has a multitier architecture that means a client-server architecture in which data management logic and application processing are physically separated. Web application architecture defines the interactions between applications, middleware systems, and databases to ensure multiple applications can work together. [45]. The application has three main layers, each of which is responsible for certain work and logic. There are also helper layers for Operation and Repository layers. The Helper and Mapper layers help to separate the logic and make the code easier to understand and keep it clean. The data access layer helps the Repository interact with the Database and provides all the necessary operations. [Figure 14].

This system allows to divide the entire application into precisely defined layers, where each layer has its own responsibility and provides certain logic and functions. This gives a clear idea of where some method should be and what it does.

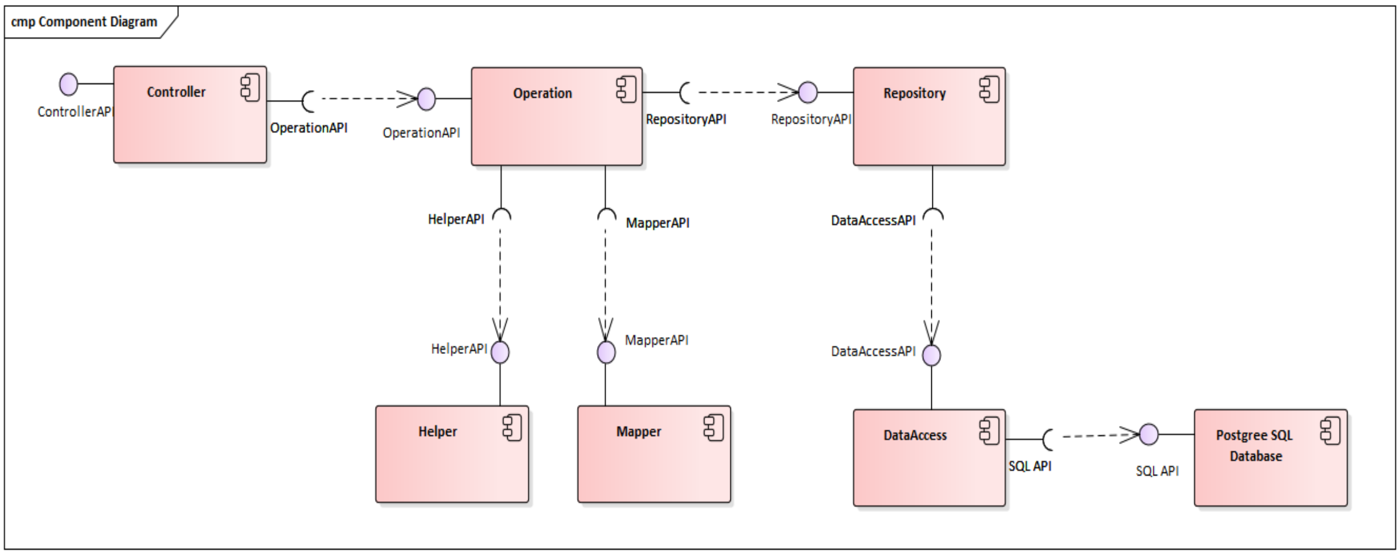


Figure 14: Component Diagram

4.1.1. Controller Layer

The Controller layer is the conductor of operations for a request. It controls the transaction scope and manages the session related information for the request. The controller first dispatches to a command and then calls the appropriate view processing logic to render the response. [29] The Controller checks authentication tokens, users' rights, and roles via Attributes. The Controller provides Endpoints for API Requests.

4.1.2. Operation Layer

The Operation class is a layer between Repository and Controller. It provides the main logic of the application: checks if parameters are correct (Guard Library), Maps Model to DTO and vice versa, uses internal and external libraries, manipulates with data via Repository, and prepares response by request from Controller.

4.1.3. Repository Layer

Repositories are classes that encapsulate the logic required to access data sources. They centralize common data access functionality, providing better maintainability and decoupling the infrastructure or technology used to access databases from the domain model layer. With using Entity Framework and Dapper, the code that must be implemented is simplified, thanks to LINQ and strong typing. This lets to focus on the data persistence logic rather than on data access plumbing. [28]

4.1.4. Services

The Service is an independent application that runs parallel to the main application. It is used for background processing, iterative tasks, and logic separation. The main app communicates with the Service via database or API. In the database case, the Main App can create a new row with task info in a table which is tracked by the Service App

[Figure 15] shows the process of the user request handling which goes through Controller, Operation, and Repository layers.

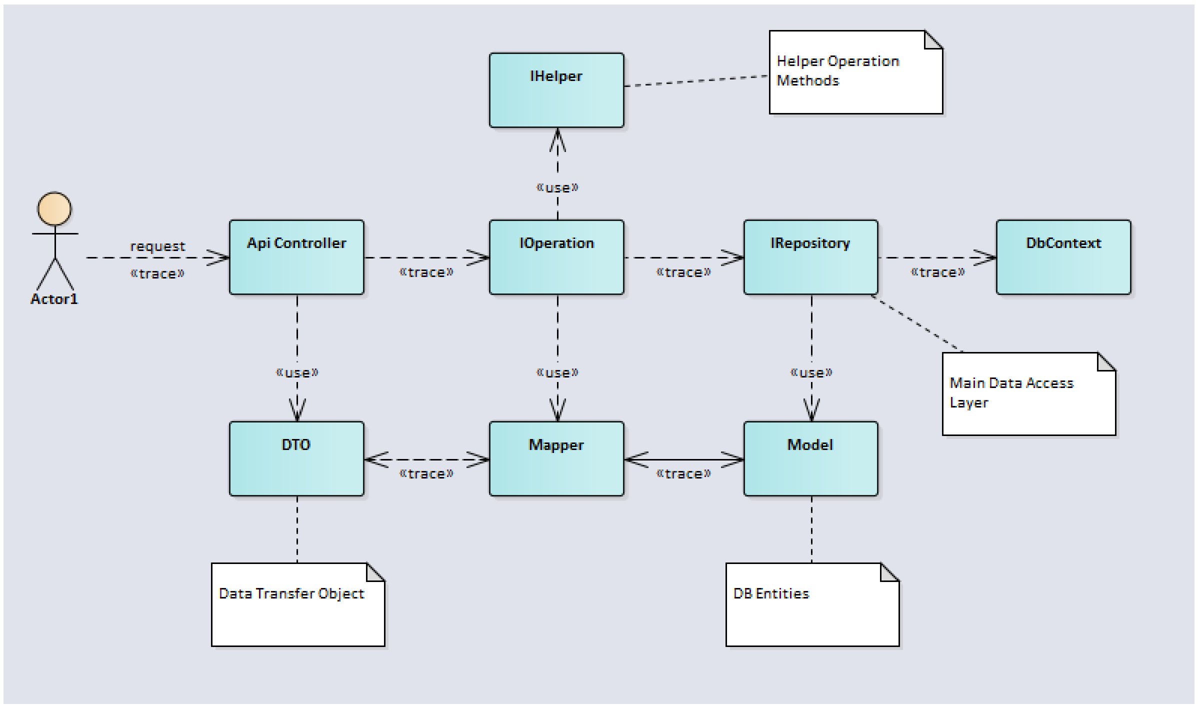


Figure 15: Process of user request handling
API Controller takes request via Endpoint, sends it to IOperation, IOperation gets Data Model from DB via Repository, converts it to DTO via Mapper, can use some Helper class for other logic executing.

4.1.5. Models

Database Context and Class Models are auto-generated from database tables by update-models-from-db.bat scripts inside FinanceManagement.Infrastructure. [Figure 16] shows the structure and organization of the models.

[Id] is a Primary Key for a table and is used as a Foreign Key for every table that has references to that table. A many-to-many relationship is represented as a separate table (for example, [UserGroupRole]) that has two Foreign Keys and additional information about the relationship.

Enum is stored as a table with [Id] and [Name] columns and [enum] schema. This helps to keep a single integer instead of a string for all data and also makes it easy to refactor (such as changing the priority). [GroupRole] and [NotificationType] are examples of Enum tables.

The [Message] table is used for the MessageService which sends emails and SMS and does not depend on other objects. It is used as a communication medium between the main application that adds data and the service that reads that data and processes it.

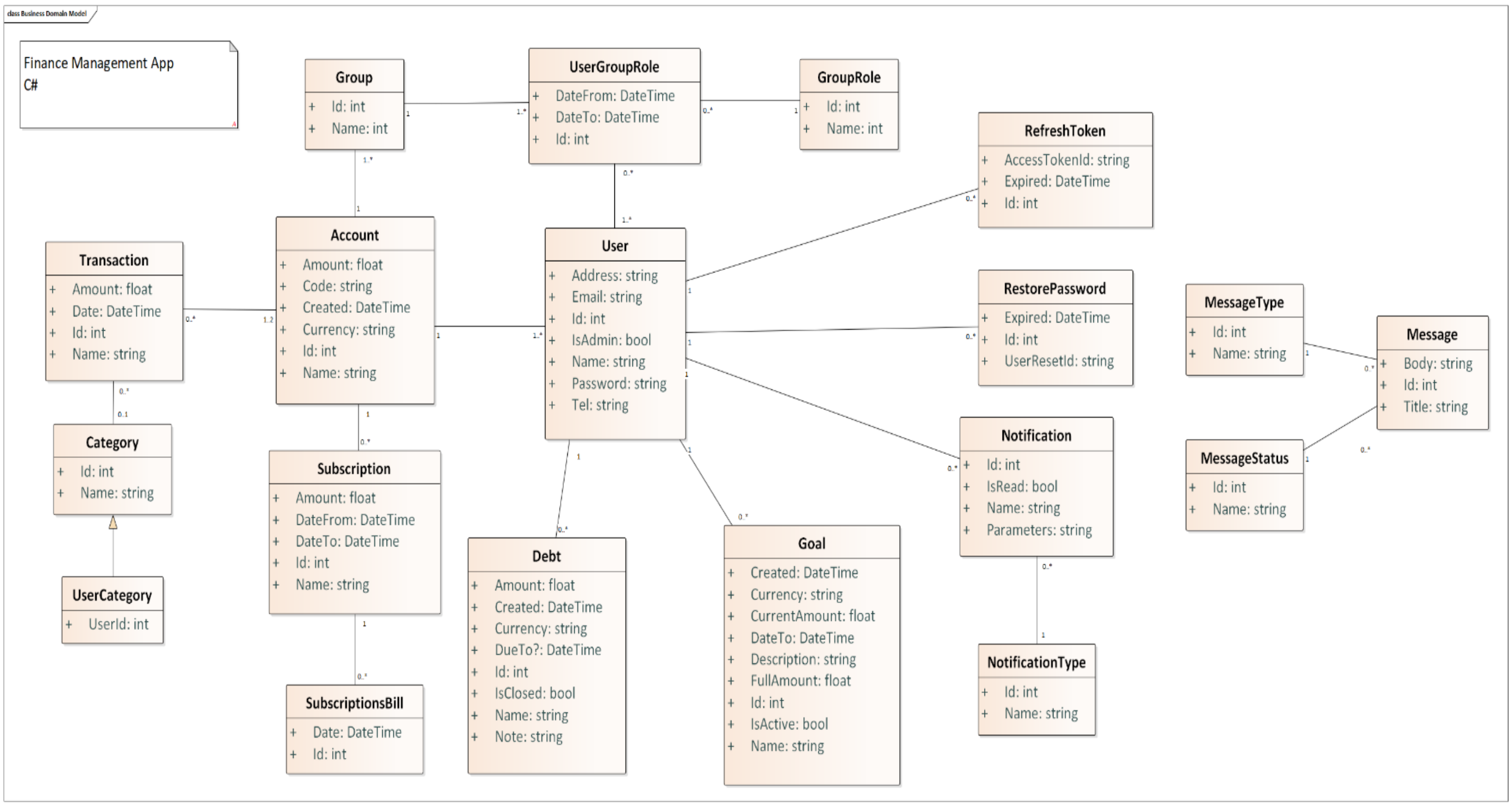
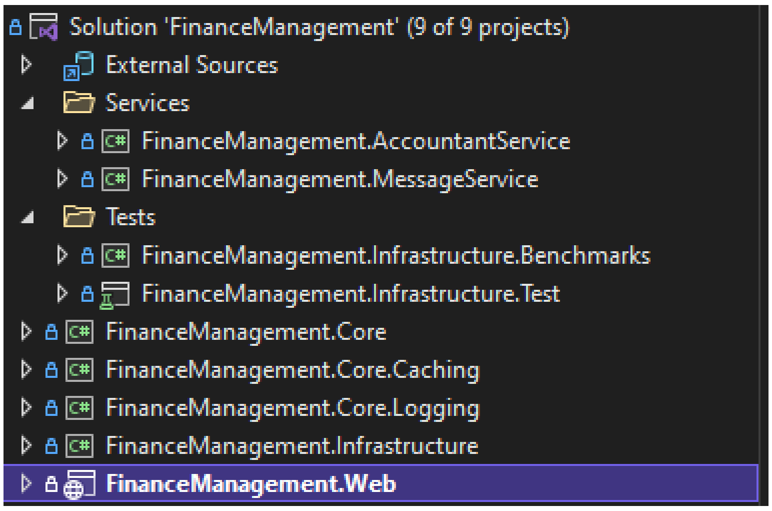


Figure 16: Business Domain Model

4.2. Solution and Files Structure

[Figure 17] shows us the structure of the solution. FinanceManagement.Web is the main project which has a starting point of the application and contains web technologies such as controllers, Vue apps, and HTML Templates. The Core libraries are used for logical separation and contain the specific part and implementation. For example FinanceManagement.Core.Caching provides caching support (cache modules, dependencies, invalidation, providers). FinanceManagement.Infrastracture is the basic library for FinanceManagement.Web with Business Access Layer and Data Access Layer. This is where operations, helpers, DTO, models, repositories, factories, configurations, etc. are stored. Service projects have their own starting point and run in parallel with the main app. Test projects provide all kinds of tests like Unit Tests, Benchmarks, and Functional Tests.



```
Solution 'FinanceManagement' (9 of 9 projects)
  External Sources
  Services
    FinanceManagement.AccountantService
    FinanceManagement.MessageService
  Tests
    FinanceManagement.Infrastructure.Benchmarks
    FinanceManagement.Infrastructure.Test
  FinanceManagement.Core
  FinanceManagement.Core.Caching
  FinanceManagement.Core.Logging
  FinanceManagement.Infrastructure
  FinanceManagement.Web
```

Figure 17: Solution Explorer

5. Implementation

The chapter contains information about the implementation of the application. It analyzes all used technologies and external libraries and explains why they were used and what it brings to the application.

5.1. Configuration

To configure the differences between development, staging, and production environments is used the Database Table with values and keys. Connection strings, common constants, and external library settings are stored in appsettings.json file inside the project. ASP.NET Core configures app behavior based on the runtime environment using an environment variable. [30]

5.2. Logging and Profiling

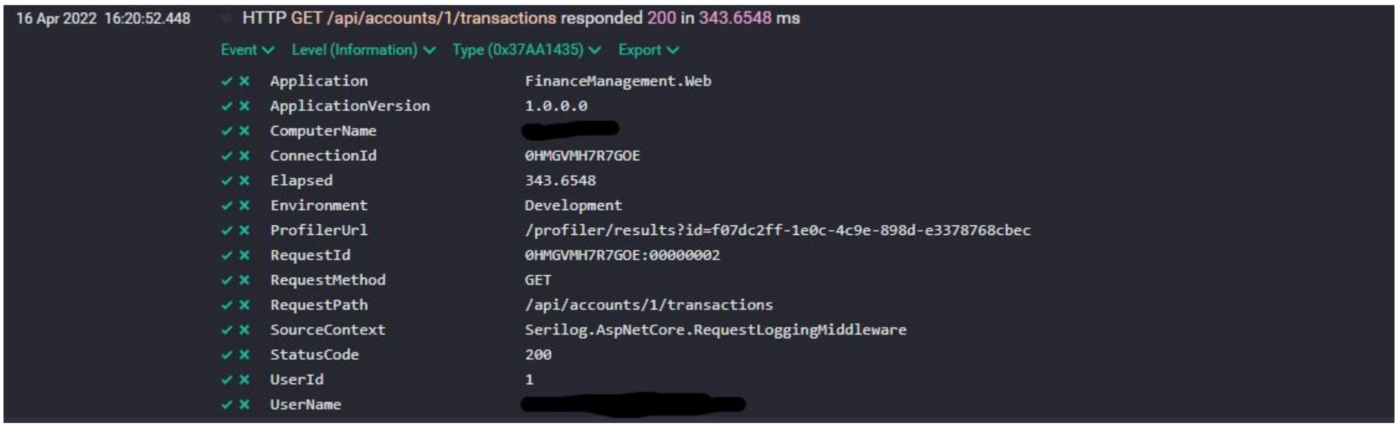
5.2.1. Serilog

Serilog is a diagnostic logging library for .NET applications. It is easy to set up, has a clean API, and runs on all recent .NET platforms. While it's useful even in the simplest applications, Serilog's support for structured logging shines when instrumenting complex, distributed, and asynchronous applications and systems. Serilog provides diagnostic logging to files, the console, and many other outputs. Unlike other logging libraries, Serilog is built from the ground up to record structured event data. [31]

Features [32]

1) Community-backed and actively developed
2) Format-based logging API with familiar levels like Debug, Information, Warning, Error, and so-on
3) Best-in-class .NET Core support, including rich integration with ASP.NET Core
4) Support for a comprehensive range of sinks, including files, the console, on-premises and cloud-based log servers, databases, and message queues
5) Sophisticated enrichment of log events with contextual information, including scoped (LogContext) properties, thread and process identifiers, and domain-specific correlation ids such as HttpRequestId
6) Zero-shared-state Logger objects, with an optional global static Log class

The Serilog Row [Figure 9] contains a lot of useful information and may be sorted by properties. The Environment, ComputerName, UserName, UserId, ProfilerUrl (contains the URL of the MiniProfiler Page with detailed query analysis [Figure 18]), Application and ApplicationVersion properties are set by the application itself. The ConnectionId, Elapsed, RequestId (trace identifier value on the HttpContext) RequestMethod, RequestPath, SourceContext, StatusCode are default



```
16 Apr 2022  16:20:52.448     HTTP GET /api/accounts/1/transactions responded 200 in 343.6548 ms
                              Event ∨  Level (Information) ∨  Type (0x37AA1435) ∨  Export ∨
                              ✓ ✕  Application            FinanceManagement.Web
                              ✓ ✕  ApplicationVersion     1.0.0.0
                              ✓ ✕  ComputerName           [redacted]
                              ✓ ✕  ConnectionId           0HMGVMH7R7GOE
                              ✓ ✕  Elapsed                343.6548
                              ✓ ✕  Environment            Development
                              ✓ ✕  ProfilerUrl            /profiler/results?id=f07dc2ff-1e0c-4c9e-898d-e3378768cbec
                              ✓ ✕  RequestId              0HMGVMH7R7GOE:00000002
                              ✓ ✕  RequestMethod          GET
                              ✓ ✕  RequestPath            /api/accounts/1/transactions
                              ✓ ✕  SourceContext          Serilog.AspNetCore.RequestLoggingMiddleware
                              ✓ ✕  StatusCode             200
                              ✓ ✕  UserId                 1
                              ✓ ✕  UserName               [redacted]
```

Figure 18: Serilog Row

Serilog properties.

Serilog is one of the most popular logging packages and it contains all the features that we might need. We can also log into the internal database and create our own control panel or use an external dashboard such as Kibana. But Serilog provides a modern, elegant, simple solution without subsequent implementations and labor costs. Thus, it is considered the best choice for this application.

5.2.2. MiniProfile

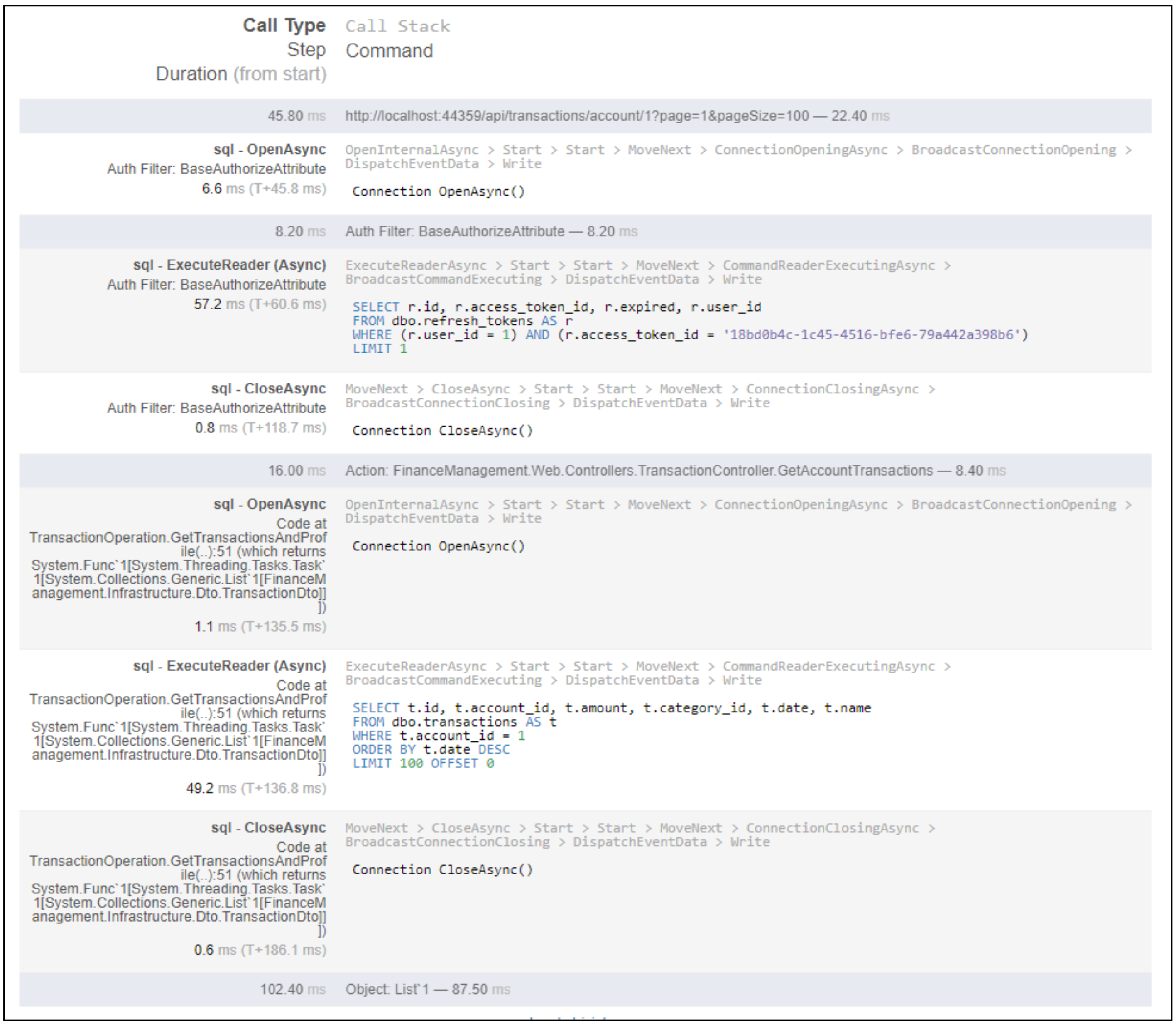
An ADO.NET profiler, capable of profiling calls on raw ADO.NET (SQL Server, Oracle, etc), LINQ-to-SQL, Entity Framework (including Code First and EF Core), and a range of other data access scenarios [32]. Profiling is used for complex operations and database requests. This gives us a clear idea of the project's performance and weaknesses in the code. The parent class of all Operation classes has a protected method that accepts any function and runs it via Profiler [Code 1]. This gives us a complete understanding of the function's steps and how it works [Figure 90]. MiniProfile sends profiling results to Serilog [Code 2].

```
await ExecuteAndProfile(() => Func());


protected virtual TResult ExecuteAndProfile<TResult>(Func<TResult> function)
{
            using (MiniProfiler.Current.Step(codeBlockDescription))
     {
                        try
            {
                                    return function.Invoke();
            }
            catch (Exception exception)
            {
                        throw new BaseException(CODE_DESCRIPTION_PROFILE, exception);
            }
     }
}
```

Code 1: MiniProfile usage

```
app.UseSerilogRequestLogging(options =>
{
            options.EnrichDiagnosticContext = (diagnosticContext, httpContext) =>
     {
                        diagnosticContext.Set("ProfilerUrl",
                        ProfilerConstants.GetUrlToProfilationDetail((Guid)httpContext.Items["ProfilerId"]));
            };
});
```

Code 2: MiniProfile Url to Seq Logging Connection



Call Type Call Stack
Step Command
Duration (from start)

45.80 ms http://localhost:44359/api/transactions/account/1?page=1&pageSize=100 — 22.40 ms

sql - OpenAsync
Auth Filter: BaseAuthorizeAttribute
6.6 ms (T+45.8 ms)
OpenInternalAsync > Start > Start > MoveNext > ConnectionOpeningAsync > BroadcastConnectionOpening > DispatchEventData > Write

```
Connection OpenAsync()
```

8.20 ms Auth Filter: BaseAuthorizeAttribute — 8.20 ms

sql - ExecuteReader (Async)
Auth Filter: BaseAuthorizeAttribute
57.2 ms (T+60.6 ms)
ExecuteReaderAsync > Start > Start > MoveNext > CommandReaderExecutingAsync > BroadcastCommandExecuting > DispatchEventData > Write

```
SELECT r.id, r.access_token_id, r.expired, r.user_id
FROM dbo.refresh_tokens AS r
WHERE (r.user_id = 1) AND (r.access_token_id = '18bd0b4c-1c45-4516-bfe6-79a442a398b6')
LIMIT 1
```

sql - CloseAsync
Auth Filter: BaseAuthorizeAttribute
0.8 ms (T+118.7 ms)
MoveNext > CloseAsync > Start > Start > MoveNext > ConnectionClosingAsync > BroadcastConnectionClosing > DispatchEventData > Write

```
Connection CloseAsync()
```

16.00 ms Action: FinanceManagement.Web.Controllers.TransactionController.GetAccountTransactions — 8.40 ms

sql - OpenAsync
Code at TransactionOperation.GetTransactionsAndProfile(..):51 (which returns System.Func`1[System.Threading.Tasks.Task`1[System.Collections.Generic.List`1[FinanceManagement.Infrastructure.Dto.TransactionDto]]])
1.1 ms (T+135.5 ms)
OpenInternalAsync > Start > Start > MoveNext > ConnectionOpeningAsync > BroadcastConnectionOpening > DispatchEventData > Write

```
Connection OpenAsync()
```

sql - ExecuteReader (Async)
Code at TransactionOperation.GetTransactionsAndProfile(..):51 (which returns System.Func`1[System.Threading.Tasks.Task`1[System.Collections.Generic.List`1[FinanceManagement.Infrastructure.Dto.TransactionDto]]])
49.2 ms (T+136.8 ms)
ExecuteReaderAsync > Start > Start > MoveNext > CommandReaderExecutingAsync > BroadcastCommandExecuting > DispatchEventData > Write

```
SELECT t.id, t.account_id, t.amount, t.category_id, t.date, t.name
FROM dbo.transactions AS t
WHERE t.account_id = 1
ORDER BY t.date DESC
LIMIT 100 OFFSET 0
```

sql - CloseAsync
Code at TransactionOperation.GetTransactionsAndProfile(..):51 (which returns System.Func`1[System.Threading.Tasks.Task`1[System.Collections.Generic.List`1[FinanceManagement.Infrastructure.Dto.TransactionDto]]])
0.6 ms (T+186.1 ms)
MoveNext > CloseAsync > Start > Start > MoveNext > ConnectionClosingAsync > BroadcastConnectionClosing > DispatchEventData > Write

```
Connection CloseAsync()
```

102.40 ms Object: List`1 — 87.50 ms

Figure 19: MiniProfile Result

5.3. API Controllers

Other applications can communicate with this backend app via API Controllers.

An API gateway is an API management tool that sits between a client and a collection of backend services. API management refers to the processes for distributing, controlling, and analyzing the APIs that connect applications and data across the enterprise and across clouds. An API gateway is one part of an API management system. The API gateway intercepts all incoming requests and sends them through the API management system, which handles a variety of necessary functions [4].

Exactly what the API gateway does will vary from one implementation to another. Some common functions include authentication, routing, rate limiting, billing, monitoring, analytics, policies, alerts, and security. An API gateway is a way to decouple the client interface from backend implementation. When a client makes a request, the API gateway breaks it into multiple requests, routes them to the right places, produces a response, and keeps track of everything. An API gateway acts as a reverse proxy to accept all application programming interface (API) calls, aggregate the various services required to fulfill them, and return the appropriate result [4].

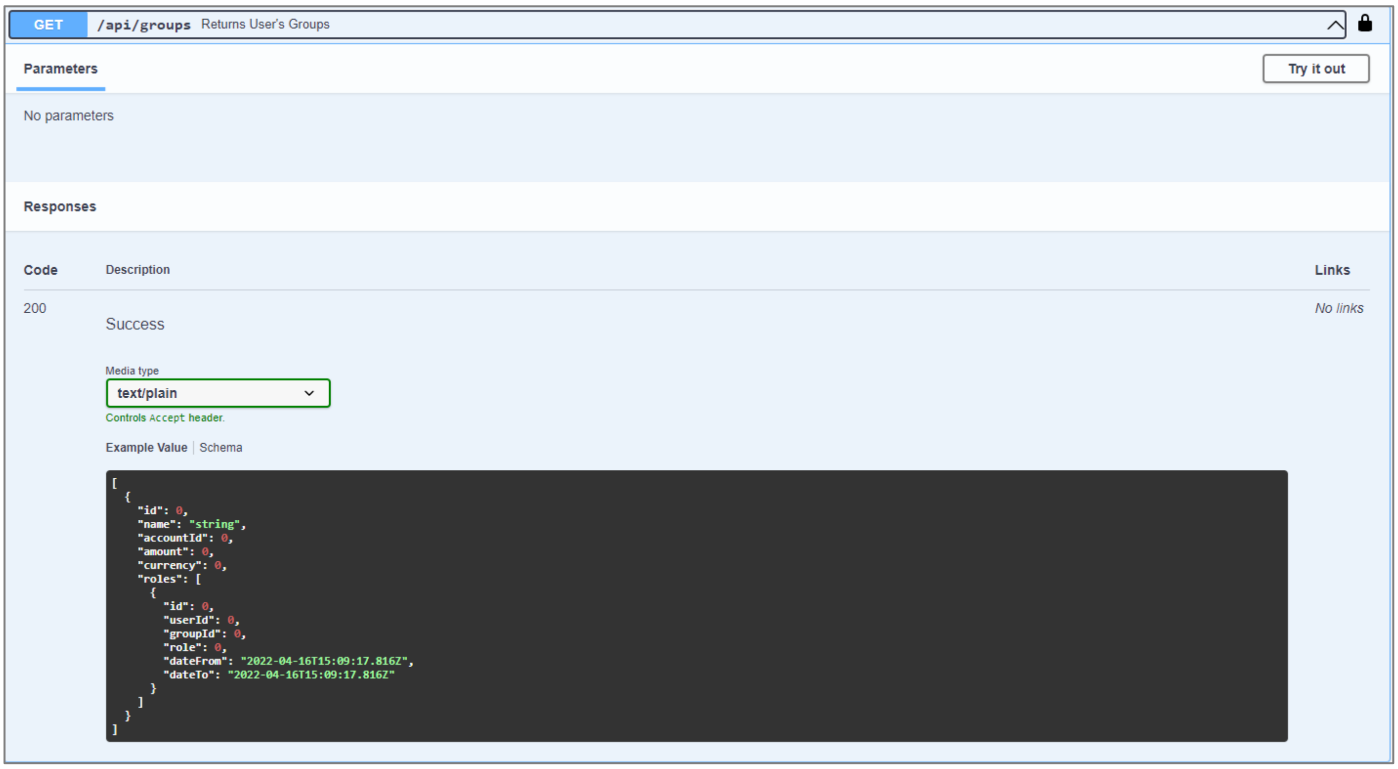
Api Controlers use REST API. REST is a set of architectural constraints, not a protocol or a standard. REST is an acronym for **RE**presentational **S**tate **T**ransfer. In REST, the primary data representation is called resource. Having a consistent and robust REST resource naming strategy – will prove one of the best design decisions in the long term. REST APIs use Uniform Resource Identifiers (URIs) to address resources. REST API designers should create URIs that convey a REST API’s resource model to the potential clients of the API. When resources are named well, an API is intuitive and easy to use. If done poorly, that same API can be challenging to use and understand. [43]. The application API uses best practices and guidelines written in [43].

API Controllers use Attributes inherited from System.Attribute to different verifications and operations before executing the endpoint’s logic. The attribute may forbid the next execution. For example, BaseAuthorizeAttribute: Authorize will forbid the next execution if the user's authorization fails validation.

Some requests may wait on the response a long time, therefore FE will send with the request a Cancellation Token, which will be checked each time after some process on BE. If FE wants to cancel the request (web page closing, user action) it will close the cancelation token. BE will catch this action and will stop its own process.

Every data access route starts with '/api'. This helps the frontend application recognize when it should send an API request or switch a page. In development mode, the frontend and backend work in parallel, the frontend requires a proxy to the development server, and for all routers that start with "/api" it can use the server's URL instead own. When the route does not start with '/api', Frontend Router System just switches its own pages.

Swagger [2] is used for API documentation [Figure 21]. Swagger is a language-agnostic specification for describing REST APIs. It allows both computers and humans to understand the capabilities of a REST API without direct access to the source code. [44]. It has a clear, concise user interface separated by controllers and provides all information about the endpoints (request method and path), as well as the ability to try to call each endpoint and shows information about the input parameters and the returned object. [Figure 20].


GET /api/groups Returns User's Groups

Parameters

Try it out

No parameters

Responses

Code	Description	Links
200	Success	No links

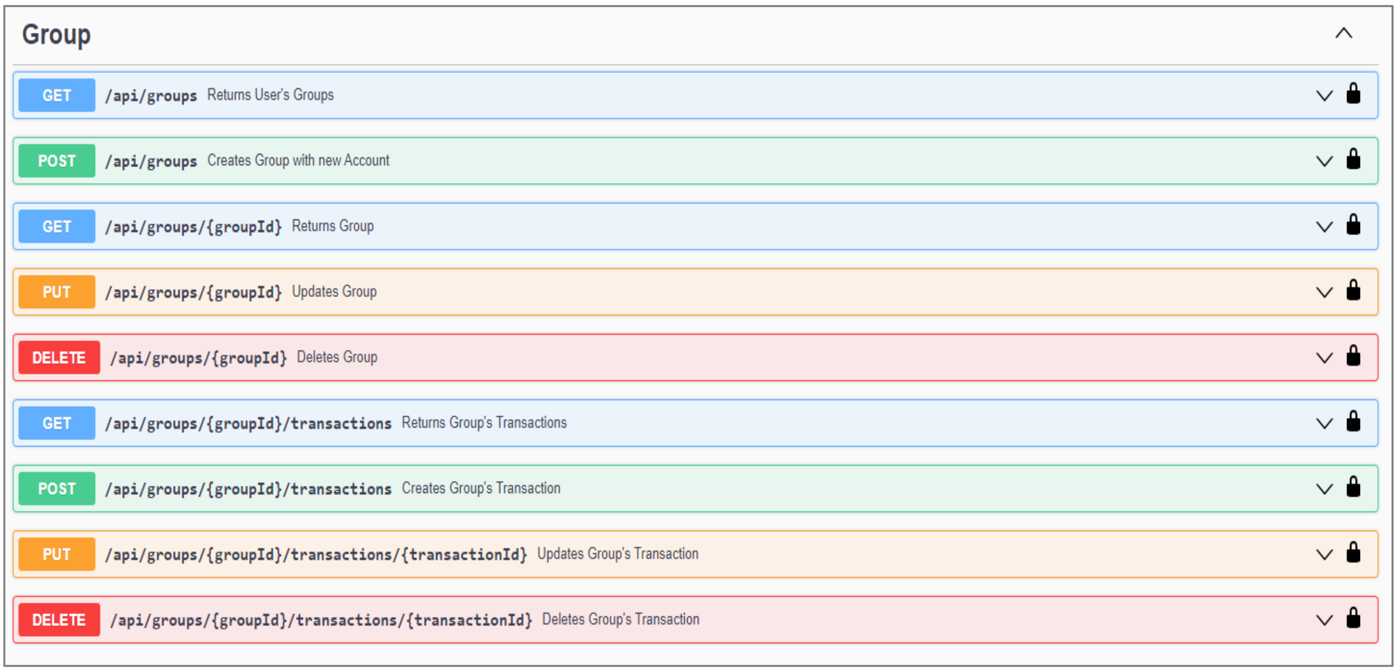
Media type: text/plain

Controls Accept header.

Example Value | Schema

```
[
  {
    "id": 0,
    "name": "string",
    "accountId": 0,
    "amount": 0,
    "currency": 0,
    "roles": [
      {
        "id": 0,
        "userId": 0,
        "groupId": 0,
        "role": 0,
        "dateFrom": "2022-04-16T15:09:17.816Z",
        "dateTo": "2022-04-16T15:09:17.816Z"
      }
    ]
  }
]
```

Figure 20: Swagger Endpoint


Group

GET /api/groups Returns User's Groups

POST /api/groups Creates Group with new Account

GET /api/groups/{groupId} Returns Group

PUT /api/groups/{groupId} Updates Group

DELETE /api/groups/{groupId} Deletes Group

GET /api/groups/{groupId}/transactions Returns Group's Transactions

POST /api/groups/{groupId}/transactions Creates Group's Transaction

PUT /api/groups/{groupId}/transactions/{transactionId} Updates Group's Transaction

DELETE /api/groups/{groupId}/transactions/{transactionId} Deletes Group's Transaction

Figure 21: Swagger API

5.4. WebSocket and SignalR

5.4.1. WebSocket

WebSocket is a computer communications protocol, providing full-duplex communication channels over a single TCP connection. It's used in apps that benefit from fast, real-time communication. [10].

5.4.2. SignalR

ASP.NET SignalR is a library for ASP.NET that makes it incredibly simple to add real-time web functionality applications. SignalR will use WebSockets under the covers when it's available, and gracefully fallback to other techniques and technologies when it isn't, while your application code stays the same. SignalR also provides a very simple, high-level API for doing server to client RPC (Remote procedure call, call JavaScript functions in clients' browsers from server-side .NET code), as well as adding useful hooks for connection management, e.g. connect/disconnect events, grouping connections, authorization [33]. The Application uses SignalR for notification handling [Code 5], changes in groups, and requests for dashboard data. Each SignalR Hub inherits from BaseHub which is responsible for the authentication process and the creation of users groups. In the OnConnected event, BaseHub creates a user group by ConnectedId and UserID from the HttpContext [Code 4]. This makes it possible to send some information to all user sessions (multiply open browser tabs) [Code 3].

```
public override async Task OnConnectedAsync()
{
        await Groups.AddToGroupAsync(Context.ConnectionId, userId);
        await base.OnConnectedAsync();
        Log.Information("Client connected {UserId} {ConnectionId}", userId, ConnectionId);
}
```

Code 3: SignalR Hub, OnConnected

```
public interface INotificationHub
{
        Task SendNotification(NotificationDto notification);
}


IHubContext<NotificationHub, INotificationHub> notificationHubContext;


await notificationHubContext.Clients.Group(userId.ToString()).SendNotification(new NotificationDto());
```

Code 4: SignalR NotificationHub

5.5. App Services

The Application can use different services for distributing work and logic. It will bring a lot of benefits.

- Flexibility
- Distributing Processing
- Independent Processes
- Easier understanding

FinanceManagement. AccountantService helps to application check the validity of subscriptions, debts, etc, and also bills subscriptions. The Service communicates with the main app via database. FinanceManagement.MessageService is responsible for sending emails and SMS to users from the system. The Service communicates with the main app via database.

5.6. Mappers

- Mappers Adapt Database Model to DTO and vice versa
- AutoMapper uses in almost all cases. MapperConfiguration.cs sets AutoMapper Rules [Code 5].
- Static Mappers uses in cases when we need to rewrite most properties.

```
config.NewConfig<Notification, NotificationDto>()
 .Map(s => s.NotificationType, m => (NotificationTypeEnum) m.NotificationTypeId);
```

Code 3: Setting AutoMapper Rule to Convert Model Notification to DTO inside MapperConfiguration.cs

5.7. Authorization and Authentication

- API uses a JWT (JSON Web Token) [9] as authentication.
- HTTP Authorization header using the Bearer schema.
- Access tokens [7] carry the necessary information to access a resource directly [34]. Access tokens are stored on the client side in local storage.
- Refresh Tokens [8] are credentials used to get access tokens. Refresh Tokens are stored in the Database and Cached. A user doesn't have access to it.

The API controller can secure each endpoint with an authentication attribute that verifies the access token. The frontend can obtain an access token via a login request and must send it with every request where authentication is required. The authentication attribute converts an access token from the header into claims (identifier, roles, user data, token data), validates it, and compares it with the refresh token stored in the database. If all verifications are successful, the attribute grants permissions to a request and an endpoint executes its logic.

There are no other independent projects that need this authentication process (token generation, data caching). We don't need a separate service that can authenticate users for multiple different sites. Thus, the authentication logic is implemented inside the main application.

5.8. Data Context

5.8.1. Entity Framework Core

Entity Framework Core is a modern object-database mapper for .NET. It supports LINQ queries, change tracking, updates, and schema migrations. EF Core works with many databases, including SQL Database (on-premises and Azure), SQLite, MySQL, PostgreSQL, and Azure Cosmos DB [27]. Entity Framework is used for almost all requests because it is very simple and convenient. Entity Framework allows you to write SQL queries in C# and returns prepared result as C# object [Code 6]. Entity Framework is an object-relational mapper (O/RM) that enables .NET developers to work with a database using .NET objects. It eliminates the need for most of the data-access code that developers usually need to write [38].

```
return await this.Entities.OrderBy(s => s.Id).Select(s => s.Id)
.Skip(skip).Take(count).ToListAsync();
```

Code 4: Entity Framework Query

5.8.2. Dapper

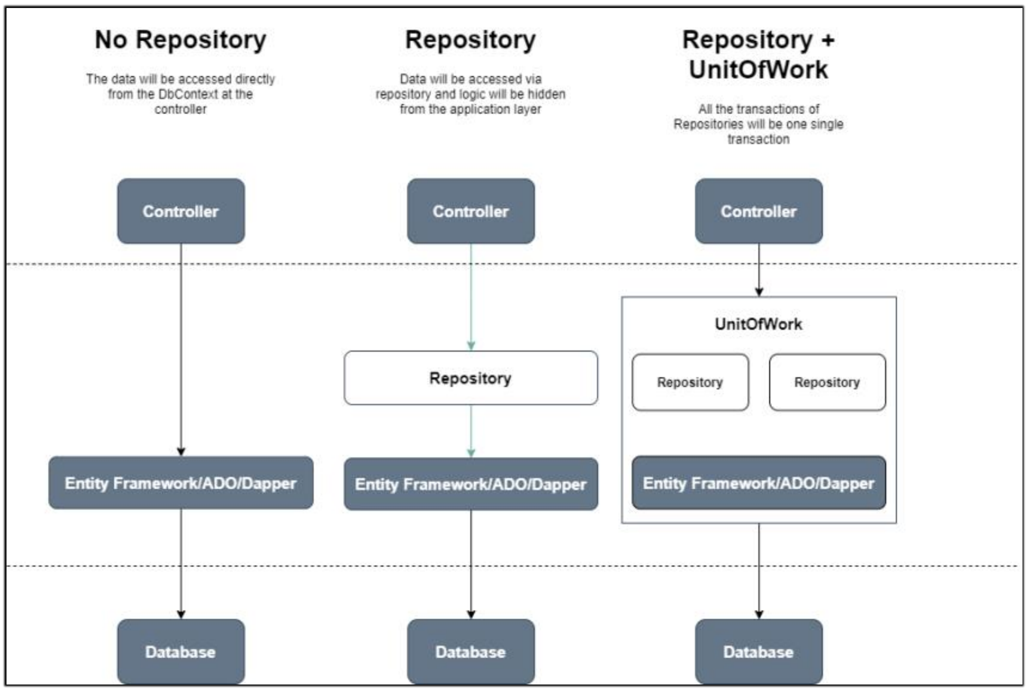
Dapper is a popular simple object mapping tool. It is designed primarily to be used in scenarios where you want to work with data in a strongly typed fashion - as business objects in a .NET application, but don't want to spend hours writing code to map query results from ADO.NET data readers to instances of those objects. [35]. Dapper is used in the application for calling and writing Stored Procedures [Code 7].

```
DynamicParameters sqlParameter = new DynamicParameters();

sqlParameter.Add("messages_count", limit);
sqlParameter.Add("message_type", messageTypeId);

return (await DbConnection.QueryAsync<Message>("dbo.get_created_messages", sqlParameter, commandType:
System.Data.CommandType.StoredProcedure)).ToList();
```

Code 5: Calling Stored Procedure via Dapper

5.8.3. Repository

DataAccess.cs provides Repository and Unit of work patterns, Cached Repository, and DbContext. It is responsible for the connection and middleware between .NET and the database [Code 8]. [Figure 22] shows the difference between projects with and without these patterns. The repository is nothing but a class defined for an entity, with all the operations possible on that specific entity. Unit of Work is referred to as a single transaction that involves multiple operations of insert/update/delete and so on kinds. To say it in simple words, it means that for a specific user action (say registration on a website), all the transactions like insert/update/delete and so on are done in one single transaction, rather then doing multiple database transactions. This means, one unit of work here involves insert/update/delete operations, all in one single transaction [26]. DbContextFactory.cs provides Factory for different DbContextes.


No Repository
The data will be accessed directly from the DbContext at the controller
Repository
Data will be accessed via repository and logic will be hidden from the application layer
Repository + UnitOfWork
All the transactions of Repositories will be one single transaction
Controller
Controller
Controller
UnitOfWork
Repository
Repository
Repository
Entity Framework/ADO/Dapper
Entity Framework/ADO/Dapper
Entity Framework/ADO/Dapper
Database
Database
Database
Figure 22: Repository and UOW Patterns, [26]

```
public interface IDataAccess : IDisposable
{
        T Repository<T>() where T : IRepository;
        GeneratedDbContext DbContext { get; }
        Task SaveDbContext();
}
```

Code 6: IDataAccess Interface

5.8.4. Indexes

A SQL index is used to retrieve data from a database very fast. Indexing a table or view is, without a doubt, one of the best ways to improve the performance of queries and applications. [36]

The Database Tables use indexes for primary keys; foreign keys; properties, which uses in SQL queries [Code 9].

```
Entities.Where(e => e.Id == accountId && e.UserId == userId && !e.IsDeleted);


CREATE UNIQUE INDEX ui_account_id_user_id_is_deleted
ON dbo.accounts (id, user_id, is_deleted);
```

Code 7: Entity Framework query and index creation to efficiently execute that query

5.9. Caching

One of the most important features of an application is data caching. Each entity that the user wants to access is cached and has its own type of cache invalidation. There are no external packages, the project itself implements caching. The application uses its own memory to store cached data. There is no need to use Redis [42] yet.

IMemoryCache [Code 10] provides all the necessary operations. It has a provider that is responsible for getting, adding, and deleting cache data. Dependencies are used to invalidate the cache by data type. The Invalidate method invalidates the parameter cache data. If the application is running on multiple servers, this method can provide a notification system between them to reload the cache on another machine.

```
public interface IMemoryCache
{
        ICacheProvider Provider { get; }
        ICacheDependencyManager Dependencies { get; }
        bool Enabled { get; set; }
        void Invalidate(DataCacheNotification invalidateMessage);
}
```

Code 8: IMemoryCache Interface

After invalidating the cache, you need to query the database for up-to-date data. Cache modules [Code 11] implement this logic. They process invalidation requests and reload all data by invalidation type. They are also used for periodic reloading of data and at application startup.

```
public interface ICacheModule
{
   void Run();

   ......

   bool HandleDataCacheNotification(DataCacheNotification notification);
}
```

Code 9: ICacheModule Interface

The base class of each class of operations has methods for storing and retrieving cached results. These methods try to get data from the cache by the given key. If there is no requested data, it calls callback functions that return entities from the database, after which this data is stored in the cache and returned from the function.

```
protected virtual async Task<TResult> ExecuteMemoryCachedAsync<TResult>(
            Func<Task<TResult>> function,
            string cacheKey,
            bool ignoreCachedValue,
            bool cacheNullValue,
            params CacheDependency[] dependencies);
```

Code 10: ExecuteMemoryCachedAsync method

5.10. Project Installation

Thanks to the monolithic architecture, launching the application is very easy. It has three different configurations: Debug, Debug Backend Only, and Release. Debug and Debug Backend Only are used for development and differ in that Debug also runs a frontend development server while Debug Backend Only supports only the backend part. The Release configuration is used for building and publishing [Code 13]. It does not build the frontend and does not run the frontend development server but uses the prepared assembly in the FinanceManagement.Web/wwwroot folder.

There are also different launch profiles with their own settings [Code 14]. It helps to set environment variables, application URL, a command to execute, browser open status, and so on.

The application can be run with any configuration and profile using IDEA or the dotnet run command [Code 15].

```
dotnet build --configuration Release
```

Code 11: dotnet build command

```
"profiles": {
        "IIS Express": {
            "commandName": "IISExpress",
            "launchBrowser": false,
            "environmentVariables": {
              "ASPNETCORE_ENVIRONMENT": "Development"
            }
        },
          "dotnet run": {
            "commandName": "Project",
            "environmentVariables": {
              "ASPNETCORE_ENVIRONMENT": "Development"
            },
            "applicationUrl": "http://localhost:44359"
        }
}
```

Code 13: Launch profiles

```
dotnet run --configuration %configuration% --launch-profile "dotnet run"
```

Code 12: dotnet run command

5.11. Design Patterns

5.11.1. Repository and Unit of Work

The repository and unit of work patterns [Figure 23] are intended to create an abstraction layer between the data access layer and the business logic layer of an application. Implementing these patterns can help insulate your application from changes in the data store and can facilitate automated unit testing or test-driven development (TDD). [11]. Usage in DataAccess.cs.

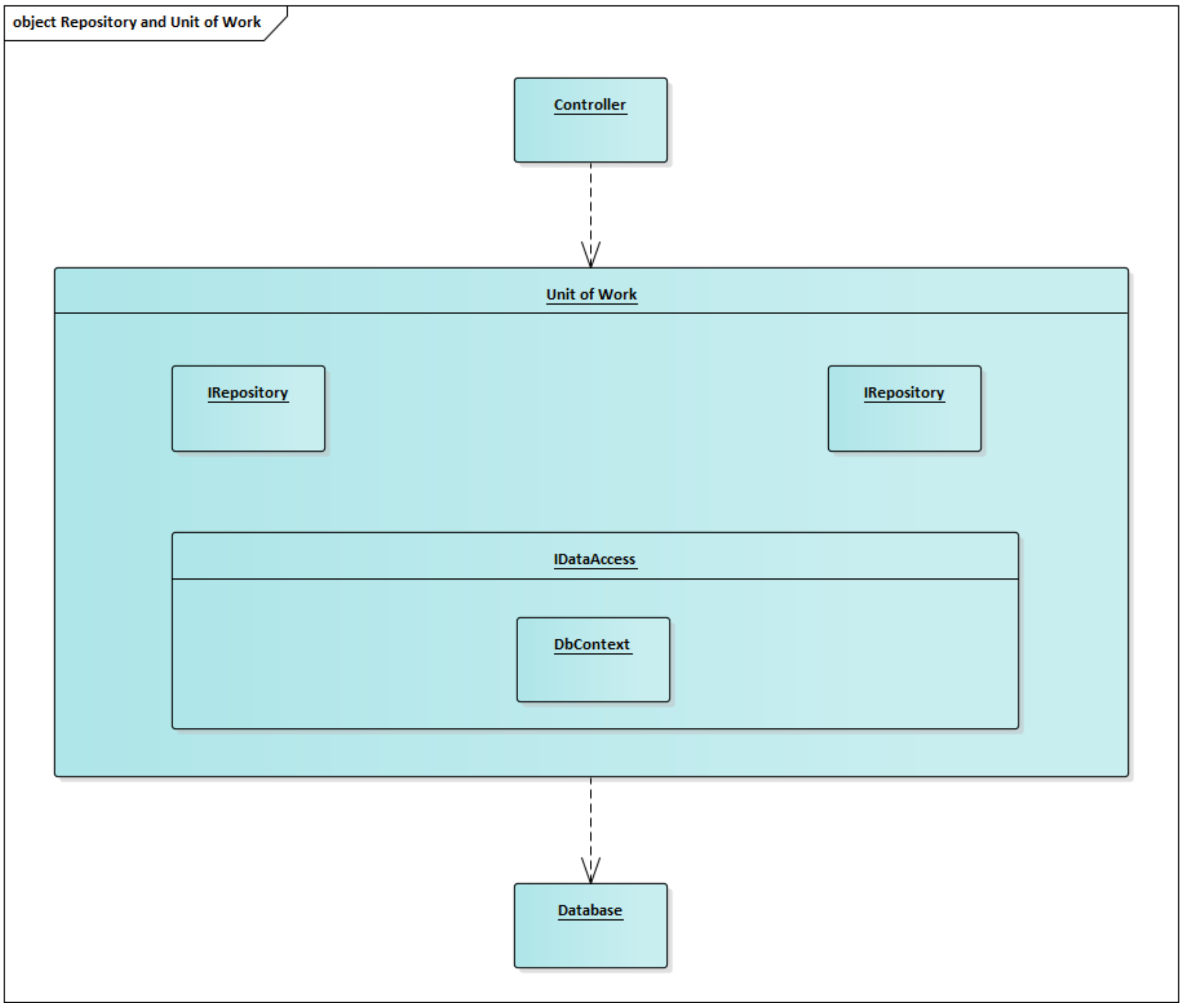


Figure 23: Repository and Unit of Work DP

5.11.2. Dependency Injection

Dependency Injection (DI) is a design pattern used to implement IoC. It allows the creation of dependent objects outside of a class and provides those objects to a class through different ways. Using DI, we move the creation and binding of the dependent objects outside of the class that depends on them. [47]

Dependency Injection manages the lifecycle of the services. [47]

- Transient: creates a new instance of the service, every time you request it.
- Scoped: creates a new instance for every scope. (Each request is a Scope). Within the scope, it reuses the existing service.
- Singleton: Creates a new Service only once during the application lifetime, and uses it everywhere

Dependency Injection implements Lazy Initialization [Code 16]. Lazy Initialization is a tactic to delay the creation of an object, the calculation of a value, or any other computationally expensive process until the time when it is to be used for the first time. [46].

Dependency Injection also has a scan option that adds all dependencies on the implemented class/interface. This is very useful, for example, for Operation or Repository classes that implement the same interface [Code 17].

```
services.AddTransient(typeof(Lazy<>), typeof(Lazier<>));

internal class Lazier<T> : Lazy<T> where T : class
{
        public Lazier(IServiceProvider provider) : base(() =>
            provider.GetRequiredService<T>()){}
}
```

Code 14: Lazier

```
services.AddFactory<IDataAccess, DataAccess>();

services.Scan(scan => scan.FromAssemblyOf<InfrastructureServiceInstaller>()
            .AddClasses(classes => classes.AssignableTo<ITransientOperation>())
            .AsImplementedInterfaces()
     .WithTransientLifetime()
);
```

Code 15: Dependency Injection: Adding dependencies

5.11.3. Facade

The implementation has a wide range of objects with their own logic and structure. We have to disguise it with a simple and clear interface for better understanding and hiding implementation details. To do this, we can use the Facade Design Patter. Facade is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes. [48]. Application-wide use [Figure 24], configuration is in InfrastructureServiceInstaller.cs.

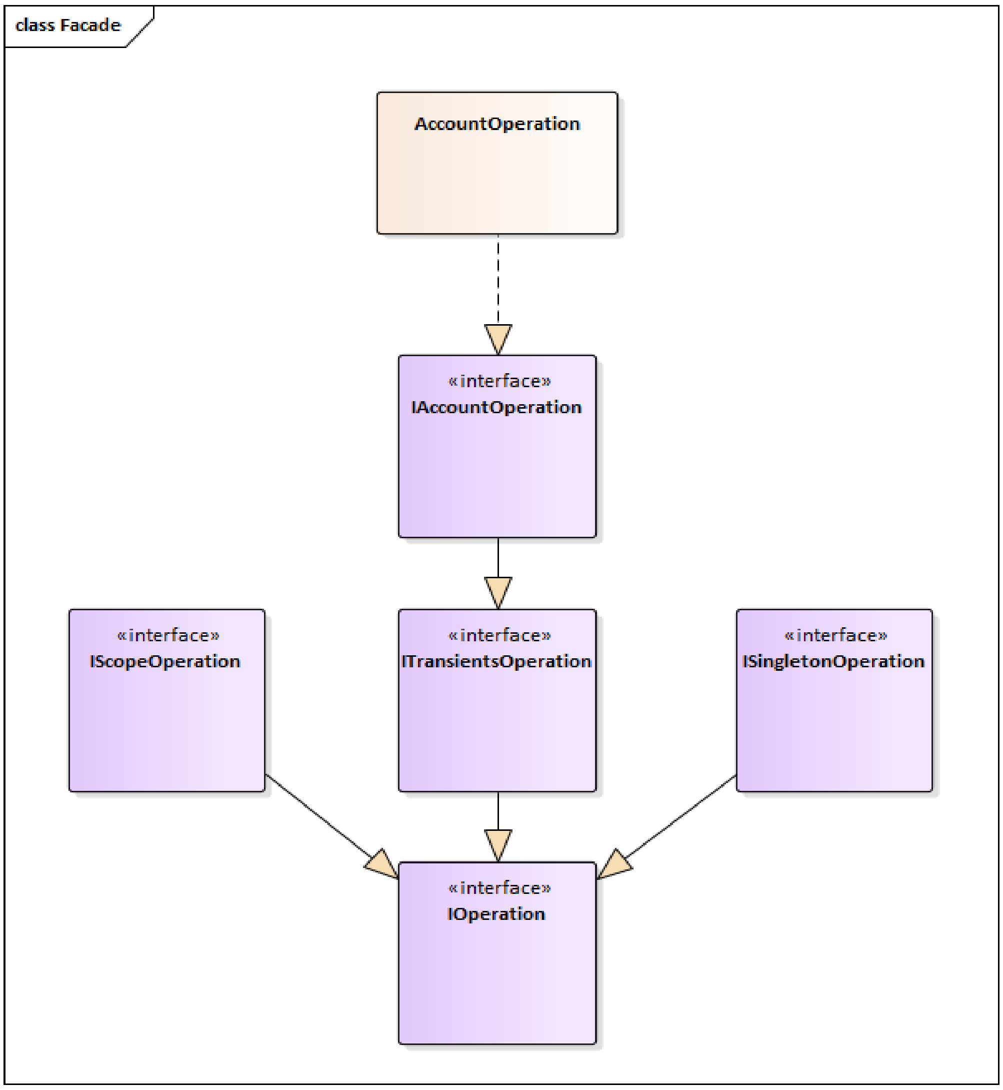

class Facade
AccountOperation
«interface» IAccountOperation
«interface» IScopeOperation
«interface» ITransientsOperation
«interface» ISingletonOperation
«interface» IOperation

Figure 24: Facade Design Pattern

5.11.4. Observer

The client should be notified about some background tasks (group changes, subscription expiration, etc.). It can send constant requests to the server, for example every minute, to check if there are new notifications on the server. But this leads to a large overload of requests and a load on the system. Most of these requests will be negative and will not return any information. Instead, we can notify the client from the server when something new has happened. It perfectly implements the Observer Design Pattern. The observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing. [49].

Usage in BaseHub.cs [Figure 25].

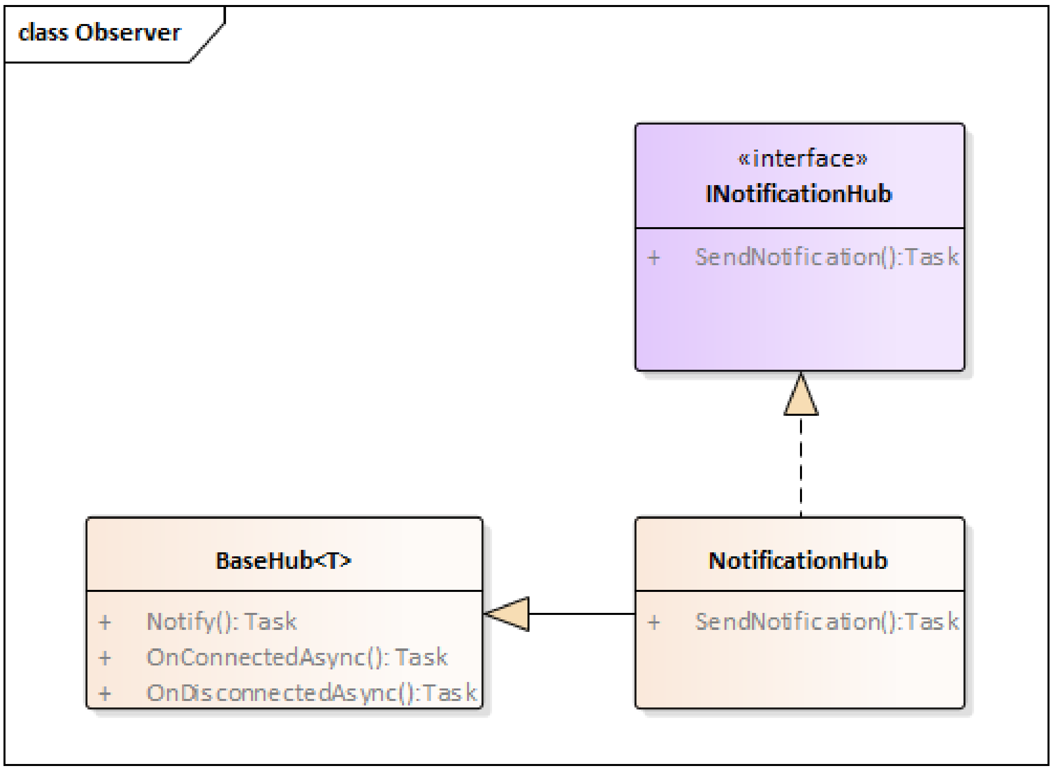

class Observer
«interface»
INotificationHub
+ SendNotification():Task
BaseHub<T>
+ Notify(): Task
+ OnConnectedAsync(): Task
+ OnDisconnectedAsync():Task
NotificationHub
+ SendNotification():Task

Figure 25: Observer Design Pattern

5.11.5. Strategy

Strategy is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable. [50]. It helped us create a service process with multiple processors, where each processor inherits from the base processor with a specific interface and has a different implementation inside.

Usage in FinanceManagement.AccountantService.BaseProccessor.cs [Figure 26].

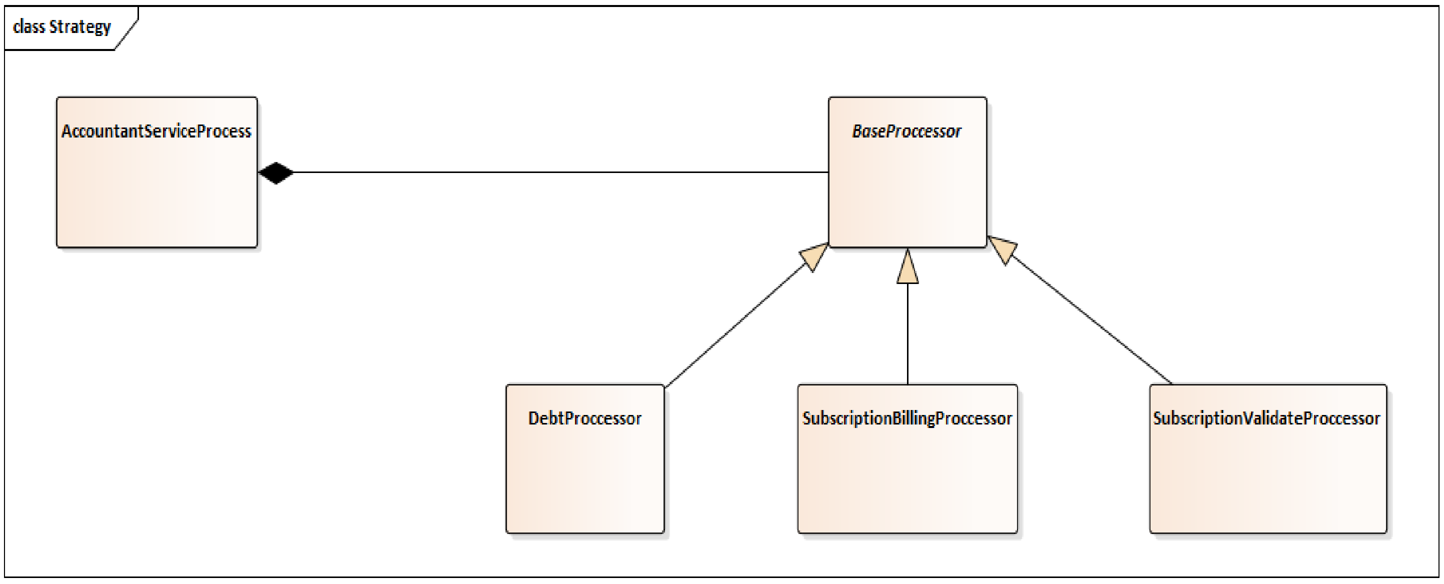


Figure 26: Strategy Design pattern

5.11.6. Builder

Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code. [51]. It made it easy for us to set up a web hosting instance with our own customizations and extensions.

Application-wide use, FinanceManagement.Web.Program.cs [Code 18].

```
Host.CreateDefaultBuilder(args)
.UseSerilog(…)
.ConfigureWebHostDefaults(…)
.UseWindowsService()
.Build()
.Run();
```

Code 16: Builder Design Pattern

6. Testing

One of the most important tasks for the backend is to check how fast the program is and how much memory it uses. For these purposes, Benchmarks were used. They call functions with the same input many times and show the average time and memory usage results.

The next type of test is security testing. For this, Unit and Functional tests were used. They call the same functions with different user roles and check if any role without the necessary rights can access some logic or not. For example, an unauthorized user cannot create a Subscription or any authorized user cannot obtain another user's Transactions.

It's also very important to know if functions do exactly what they're supposed to. For example, if a function needs to create an object in a database, the test will check if the database will have this new object after the function is called. These tests should also check for exceptions and errors. For example, a function that updates an object in the database and receives as a parameter an object ID that does not exist should throw an exception.

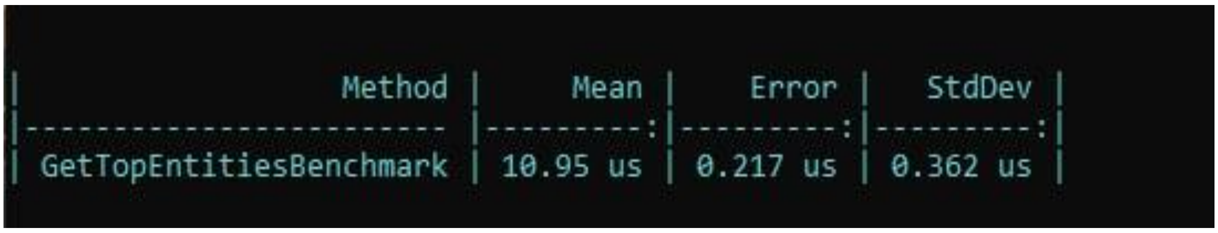
6.1 Benchmarks

Benchmark testing compares performance testing results against performance metrics that are agreed upon in the organization based on different industry standards. It helps determine the quality standards of every software application that belonged to an organization. Benchmark testing covers software, hardware, and network performance. The goal for benchmark testing is to test all the current and future releases of an application to maintain high-quality standards [39].

Dashboard Operations are the most complex and difficult operations. Here, Benchmarks are used to test charts and statistical data receiving. The dashboard also has a request GetTopEntities, it allows testing for cache misses [Code 19]. Benchmark Result [Figure 27] shows Mean (arithmetic mean of all measurements), Error (half of 99.9% confidence interval), StdDev (standard deviation of all measurements). Benchmarks will be very helpful when an application will have many rows in a database table to determine if the application is still fast.

```
Summary summary = BenchmarkRunner.Run<DashboardBenchmark>();

[Benchmark]
public async Task GetTopEntitiesBenchmark()

[Benchmark]
public async Task GetTransactionsAmountInTimeChartData()
```

Code 17: Dashboard Benchmark



```
|                 Method |     Mean |     Error |    StdDev |
|----------------------- |---------:|----------:|----------:|
| GetTopEntitiesBenchmark | 10.95 us | 0.217 us | 0.362 us |
```

Figure 27: Benchmark Result

6.2 Unit Tests

Unit Testing is a type of software testing where individual units or components of a software are tested. The purpose is to validate that each unit of the software code performs as expected. Unit Testing is done during the development (coding phase) of an application by the developers. Unit Tests isolate a section of code and verify its correctness. A unit may be an individual function, method, procedure, module, or object [40].

One of the most important parts of an application is cache hits. Unit Tests can help test operations that need to store certain entities for caching. Such tests call a function that must create a cache entry, check if that cache entry actually exists, and compare it to the object that should be stored by this cache key [Code 20].

```
var user = await userOperation.GetUserByIdCached(userId);
Assert.True(user != null); // user found

var cacheItem = Cache.Current.Provider.Get<UserDto>(cachekey);

Assert.True(cacheItem.HasValue && !cacheItem.IsNull); // cache Item exists
Assert.Equal(user.Id, cacheItem.Value.Id); // cache Item is the same as function result Item
```

Code 18: GetUserByIdCachedTest

The unit tests found no cache misses and no redundant database queries, which means the app can work fine with the cache-store, invalidate and reload its data.

6.3 Functional Tests

Functional Testing is a type of software testing that validates the software system against the functional requirements/specifications. The purpose of Functional tests is to test each function of the software application, by providing appropriate input, and verifying the output against the Functional requirements [41].

Functional tests are used to verify that functions actually do what they are supposed to. For example TransactionOperationTest.CreateUpdateDeleteTransactionTest() creates, updates, deletes Transaction and checks its properties after each operation. It also checks if the account’s amount has been updated. These tests cover all important operations with transactions, subscriptions, accounts, debts, and goals.

Functional testing has shown that the application logic executes correctly and doesn't do anything else that it shouldn't. This means that any external application can be connected to this server application and use its API.

6.4 Conclusion

The tests cover the most important features of the application. They check authentication, user roles, database queries, and function logic, measure complex queries and help us change some functions without possible errors in the future.

7. Conclusion

7.1 Summary

This project provides a good foundation for a finance management application with basic and essential functions that were chosen after a detailed analysis of existing similar apps. Of course, it may be extended with other important features and options that will help the application to be as useful as possible to users. But already now this backend app could be helpful and has great potential. This is proven by the described options, structure, and implementation.

The most modern technologies and platforms were used which allows the application to compete with others in the next 5 years. It is easy to support due to chosen programming language and rigorous project structure. It may be connected with all frontend frameworks that are popular now. It uses long-term support for external libraries and does not strictly depend on them. In addition, the code is well commented, each endpoint has a summary and description, and each public function has either a friendly name or a comment about what it does. All this proves that the main goals of each backend project have been achieved.

7.2 Future

The implemented application works and meets the functional and non-functional requirements, but also has room for improvement. For example, premium status and premium user role logic. In addition to income, it also limits the number of entities that one user can have, which gives us more free disk space. The next possible improvements are the following authorization possibilities, such as authorization via Facebook, Twitter, and Google.

Bibliography

[1] Why is Financial Management important in life: https://www.mymoneysouq.com/financial-blog/why-is-financial-management-important-in-life, 2022, [online]

[2] Swagger: https://swagger.io/, 2021, [online]

[3] What is a REST API *https://www.redhat.com/en/topics/api/what-is-a-rest-api*, 2020, [online]

[4]: What does an API gateway do: https://www.redhat.com/en/topics/api/what-does-an-api-gateway-do, 2019, [online]

[5]: The Critical Importance of Frontend and Backend Website Maintenance: https://norakramerdesigns.com/the-critical-importance-of-frontend-and-backend-website-maintenance, 2021, [online]

[6] SignalR: https://dotnet.microsoft.com/en-us/apps/aspnet/signalr, 2022, [online]

[7] Authorization Access tokens: https://www.okta.com/identity-101/access-token, 2020, [online]

[8] Authorization Refresh tokens: https://www.oclc.org/developer/news/2013/authentication-and-authorization-refresh-tokens.en.html, 2013, [online]

[9] Authorization JWT: https://fusionauth.io/docs/v1/tech/apis/jwt, 2021, [online]

[10] Web Sockets: https://medium.com/easyread/websocket-connection-handsake-under-the-hood-560ab1ceaff5, 2020, [online, Jonathan Natanael Siahaan]

[11] Implementing the Repository and Unit of Work Patterns *https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application*, 2021, [online]

[12] Difference between ASP.NET and Sring Boot: *https://medium.com/@putuprema/spring-boot-vs-asp-net-core-a-showdown-1d38b89c6c2d,* 2021, [online]

[13]: Why personal finance is crucial in everyday life: https://www.financialexpress.com/money/why-personal-finance-is-crucial-in-everyday-life, 2021, [online]

[14]: Wallet: https://budgetbakers.com/, 2022, [online]

[15]: CoinKepper: https://about.coinkeeper.me/eng, 2022, [online]

[16]: Spendee: https://www.spendee.com/, 2022, [online]

[17]: Gin: https://semaphoreci.com/community/tutorials/building-go-web-applications-and-microservices-using-gin, 2022, [online]

[18]: Spring Boot: https://scand.com/company/blog/pros-and-cons-of-using-spring-boot/, 2022, [online]

[19]: .Net: https://dotnet.microsoft.com/en-us/apps/aspnet, 2022, [online]

[20]: Spring Boot vs ASP.NET Core: https://medium.com/@putuprema/spring-boot-vs-asp-net-core-a-showdown-1d38b89c6c2d, 2021, [online, Putu Prema]

[21]: Difference between SQL and NoSQL: https://www.geeksforgeeks.org/difference-between-sql-and-nosql, 2021, [online]

[22]: Microservices vs Monolithic architecture: https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business, 2018, [online, Romana Gnatyk]

[23]: Microservices: https://avinetworks.com/glossary/microservice, 2021, [online]

[24]: Monolithic architecture: https://whatis.techtarget.com/definition/monolithic-architecture, 2016, [online, Ivy Wigmore]

[25]: What is microservices architecture: https://cloud.google.com/learn/what-is-microservices-architecture, 2022, [online]

[26]: Unit of Work in Repository Pattern: https://www.c-sharpcorner.com/UploadFile/b1df45/unit-of-work-in-repository-pattern, 2018, [online, Jasminder Singh]

[27]: Entity Framework: https://docs.microsoft.com/en-us/ef, 2022, [online]

[28]: Design the infrastructure persistence layer: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design, 2021, [online]

[29]: Controller layer: https://help.hcltechsw.com/commerce/9.0.0/developer/concepts/csdcontrollerbase.html, 2021, [online]

[30]: Use multiple environments in ASP.NET Core: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/environments?view=aspnetcore-6.0, 2022, [online]

[31]: Serilog: https://serilog.net/, 2022, [online]

[32]: MiniProfiler: https://miniprofiler.com/, 2022, [online]

[32]: Serilog GitHub: https://github.com/serilog/serilog/blob/dev/README.md, 2022, [online]

[33]: ASP.NET SignalR: https://dotnetfoundation.org/projects/asp.net-signalr, 2022, [online]

[34]: How to: Use API Access Token for API: https://kb.vmware.com/s/article/79543, 2020, [online]

[35]: Dapper: https://www.learndapper.com/, 2019, [online]

[36]: SQL index overview and strategy: https://www.sqlshack.com/sql-index-overview-and-strategy/, 2018, [online, Bojan Petrovic]

[37]: SQLite vs MySQL vs PostgreSQL: A Comparison Of Relational Database Management Systems: https://www.digitalocean.com/community/tutorials/sqlite-vs-mysql-vs-postgresql-a-comparison-of-relational-database-management-systems, 2014, [online]

[38]: What is Entity Framework?: https://www.entityframeworktutorial.net/what-is-entityframework.aspx, [online]

[39]: Performance Testing: Baseline and Benchmark Testing: https://www.loadview-testing.com/blog/performance-testing-baseline-and-benchmark-testing/, 2021, [online, Glenn Lee]

[40]: Unit Testing Tutorial: https://www.guru99.com/unit-testing-guide.html, 2022, [online, Thomas Hamilton]

[41]: What is Functional Testing: https://www.guru99.com/functional-testing.html, 2022, [online, Thomas Hamilton]

[42]: Redis Cache: https://redis.io, 2022, [online]

[43]: REST naming: https://restfulapi.net/resource-naming/, 2021, [online, Lokesh Gupta]

[44]: ASP.NET Core web API documentation with Swagger: https://docs.microsoft.com/en-us/aspnet/core/tutorials/web-api-help-pages-using-swagger, 2022, [online]

[45]: What is Web Application Architecture: https://stackify.com/web-application-architecture/, 2017, [online, Angela Stringfellow]

[46]: Lazy Initialization: https://medium.com/geekculture/software-design-pattern-4-lazy-initialization-35f606f1ddf3, 2021, [online]

[47]: Dependency Injection: https://www.tutorialsteacher.com/ioc/dependency-injection, 2022, [online]

[48]: Facade Design Pattern: https://refactoring.guru/design-patterns/facade, 2022, [online]

[49]: Observer Design Pattern: https://refactoring.guru/design-patterns/observer, 2022, [online]

[50]: Strategy Design Pattern: https://refactoring.guru/design-patterns/strategy, 2022, [online]

[51]: Builder Design Pattern: https://refactoring.guru/design-patterns/builder, 2022, [online]

[52]: Clean Code, 2008, [Robert Cecil Martin]

[53]: Clean Architecture: A Craftsman's Guide to Software Structure and Design, 2017, [Robert Cecil Martin]

[54]: C# in Depth, 2019, [Jon Skeet]

[55]: C# 10 and .NET 6 – Modern Cross-Platform Development: Build apps, websites, and services with ASP.NET Core 6, Blazor, and EF Core 6 using Visual Studio 2022 and Visual Studio Code, 6th Edition 6th ed. Edition, 2021, [Mark J. Price]

Supplement A. Abbreviations List

JWT	Jason Web Token
DB	Database
SQL	Structured Query Language
HTTP	Hypertext Transfer Protocol
API	Application Programming Interface
REST	Representational State Transfer
CRUD	Create, Read, Update, Delete
DAO	Data Access Object
DTO	Data Transfer Object
URL	Uniform Resource Locator
CAP	Consistency, Availability & Partition Tolerance
RPC	Remote procedure call
DP	Design Patterns

Supplement B. Diagrams

B.2. Deployment Diagram

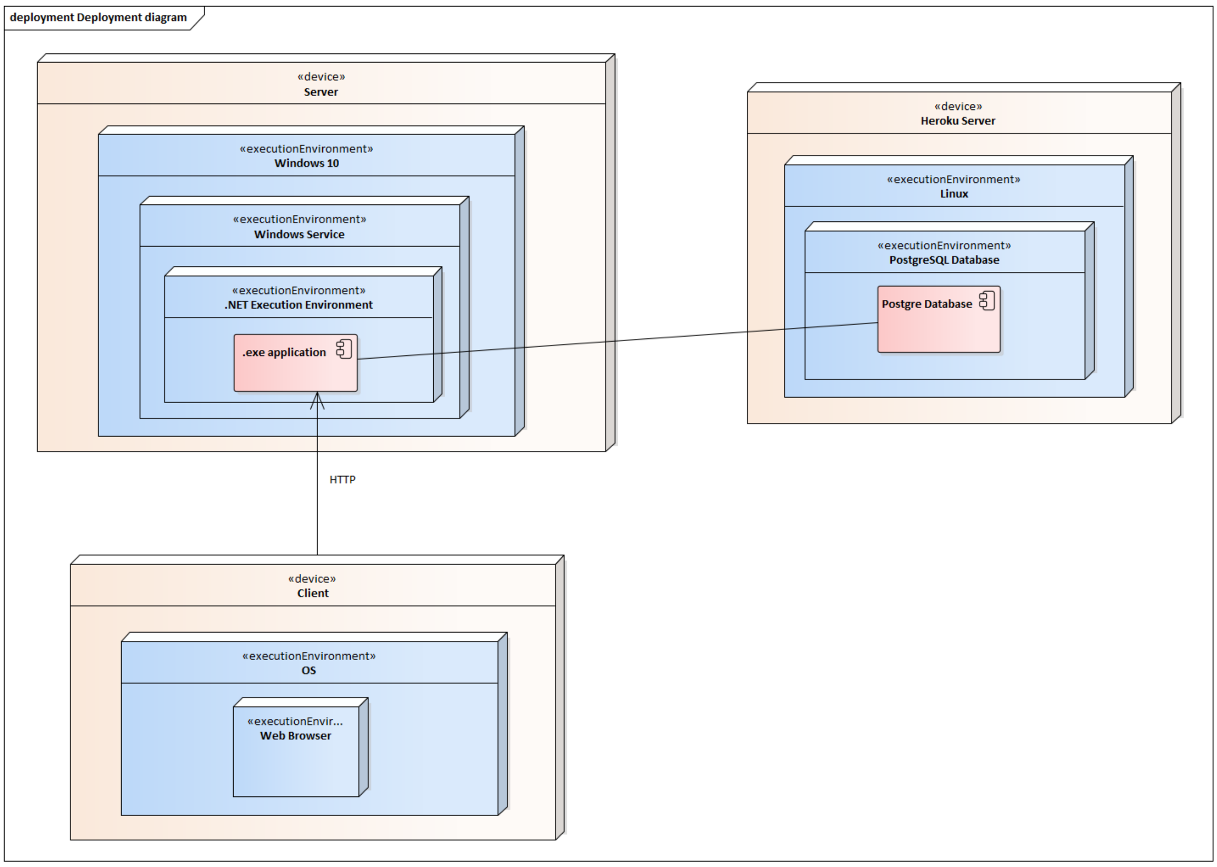


Figure 28: Deployment Diagram