Assignment of bachelor's thesis

Title:	ExpenseTracker - personal finance manager
Student:	Jamaladdin Azizov
Supervisor:	Ing. Zdeněk Rybola, Ph.D.
Study program:	Informatics
Branch / specialization:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2021/2022

Instructions

The goal of the thesis is the creation of an application for managing personal finances. The key features include:
- managing of multiple personal accounts
- tracking transactions on the accounts with advanced categorization
- support for recurring and future payments
- statistics and reports

The thesis should consist of:
- analysis of the processes and requirements
- research of existing applications and their comparison
- technology and architectural design of a custom solution
- implementation and testing of the solution
- documentation of the solution, including a user guide and deployment guide

Electronically approved by Ing. Michal Valenta, Ph.D. on 31 March 2020 in Prague.

Bachelor's Thesis

EXPENSETRACKER - PERSONAL FINANCE MANAGER

Jamaladdin Azizov

Fakulta informačních technologií ČVUT v Praze
Software Engineering
Vedoucí: Ing. Zdeněk Rybola, Ph.D.
June 27, 2021

Odkaz na tuto práci: Jamaladdin Azizov. *ExpenseTracker - Personal Finance Manager*. Bachelor's Thesis. České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Contents

List of Figures

I would like to thank my supervisor Ing. Zdeněk Rybola, Ph.D. for supporting me with the topic and for all his valuable advice during the creation of the thesis. Above all, I would like to thank my family for all the support during my studies.

Declaration

Abstrakt

Cílem práce je vytvoření aplikačního programovacího rozhraní (API) pro správu osobních financí. To zahrnuje analýzu aplikace XpenseTracker po předmětu BIE-SP2, analýzu a další návrhy vylepšení, realizaci vylepšení a testování vylepšeného řešení.

Keywords osobní finance, rozpočtování, API

Abstract

The goal of the thesis is the creation of an application programming interface (API) for managing personal finances. This includes the analysis of the XpenseTracker application after the BIE-SP2 subject, analysis and further suggestion of improvements, realization of improvements, and testing of the improved solution.

Keywords personal finances, budgeting, API

Acronyms

API	Application Programming Interface
CRUD	Create, Read, Update, Delete
UML	Unified Modeling Language
HTML	Hypertext Markup Language
CSS	Cascading Style Sheets
OOP	Object Oriented Programming
FP	Functional Programming
FRP	Functional Reactive Programming
CLI	Command Line Interface
ORM	Object-Relational Mapping
RDBMS	Relational Database Management System

Chapter 1 Introduction

Since money is essential these days, there are plenty of good reasons why you should have a personal expense tracker. Managing money is not an easy task – it takes effort and a sound system. However, once you set it up, you can result in long-term success by staying on top of your financial situation.

Knowing your financial situation can help you make better decisions with your money. It can be as simple as having a budget and sticking to it.

When it comes to tracking expenses or incomes, there are plenty of great ways you can do it—from excel templates to applications that you can use.

There's no reason people should be running up debt or be unable to pay for essentials because of their spending habits. There is a lot of unnecessary spending in people's everyday lives that can be stopped. Even small cheap purchases can add up. Logging all expenses can help to prevent that.

Another reason to keep track of monthly expenses is that it allows one to look into the future with a clearer idea of what funds are available. Budgeting also lets us know how much we could save and thus how much we can put away into different funds and accounts per month. So keeping track of monthly expenses gives a better overview of the budget, which can be very helpful.

In the end, realizing you have little money is stressful, but knowing that and not knowing what you are spending it on is much worse. So while making sure you have enough to meet your lifestyle and basic needs is one stressor, worrying about tracking does not have to be.

Considering all the details above, it was decided to create a system that will help manage your finances.

1.1 Goal of the thesis

The goal of the thesis is the creation of an API for managing personal finances with the following main features:

- Managing multiple personal accounts
- Tracking transactions on the accounts with advanced categorization
- Support for budgeting
- Sharing personal accounts with other users
- Support for recurring payments

The frontend part of the application will be created by Amir Qamili as his bachelor thesis using this API.

The initial version of the application has already been created for two semesters under the guidance of Ing. Zdeněk Rybola, PhD, in subjects BIE-SP1 and BIE-SP2 by the group of 6 students: Amir Qamili, Beksultan Baatyrbekov, Ilya Ryabukin, Jamaladdin Azizov, Nilay Baranwal, Rishabh Malik. The application is used to track the expenses and incomes of the users using several payment accounts and categorization with a simple web interface.

1.2 Structure

The first chapter of this work deals with an acquaintance with the current version of the application, an analysis of the current and the new state of the application. This part is followed by the design part chapter, where a new application and new requirements are presented. The last chapter is the implementation chapter, which deals with the development of the application itself into its final form.

Chapter 2

Analysis

In this chapter, an analysis of the current version of the ExpenseTracker application will be performed. The chapter will help to understand better and present the current version of the application, what state it is in, what it can do and can not do. The result of this analysis will serve as a starting point for the next part of this chapter, which will describe new application features. In addition, this chapter includes a comparison with existing solutions in the market.

2.1 Current version

In this section, the current version of the ExpenseTracker application is discussed. In addition, it contains the requirements that were already implemented, the domain model and issues of the current version.

2.1.1 Requirements and current application functions

The current version of the application does not satisfy the specifications that were planned at the start of the project. The followings requirements were implemented:

- **User registration and login**

  The application provides user profiles. Registration of a user is done via email and password. After logging in, the user has access to all the features of the application.

- **Payment accounts**

  A payment account is an account that you use to deposit funds or to spend money. Each user can create several payment accounts. Basic CRUD operations are implemented for accounts.

- **Categories**

  Categories help organize past and new records into aggregated structures - both for incomes and expenses. Each category can also have a subcategory. Basic CRUD operations are implemented for categories.

- **Transactions**

  Transactions are the basis of the whole application. Each account consists of transactions. Each transaction can be either income or expense. Basic CRUD operations are implemented for transactions.

2.2 Domain model and known issues

Another important part of the analysis is the domain model. This model is often illustrated using the UML class diagram. It does not contain any implementation details, but will serve as a good starting point for modeling the database model and the design model of classes. On the [1] is shown the Domain model of the current version.

As it can be seen from the model, the current version of the application does not have all the functionalities which were planned at the start of the project. Moreover, The model is missing several important attributes. The main entity in this model is Transaction, which is composition of Account and Category. It is possible to create two types of transactions: income and expense. But, the model is missing a transfer type, which is important if the user wants to transfer money from one payment account to another. Moreover, it is not possible to split transactions into several subtransactions.

Account have transactions, but it is missing balance attribute to see overall balance of an account.

Finally, there is no possibility to create repeating transactions and share payment account with another users. Both of these functionalities were planned in the initial requirements of the application. Below is a short description of each entity in the model:

2.2.1 User

This class represents registered users. In order to use the application user has to register. User entity is associated with Account and Access entities.

2.2.2 Account

Each Account consist of a name, type, currency and icon. Accounts may have only one user or could be shared with another users. Each account will have their own transactions.

2.2.3 Transaction

The main entity in this model Transaction. Transaction has to have an Account and a Category. In addition, Transaction has an amount, date, description, state and type. Transaction could be type of Income or Expense.

2.2.4 Category

Category is used to categorize transactions. Each category has to have a name. Moreover, each category may have a subcategory.

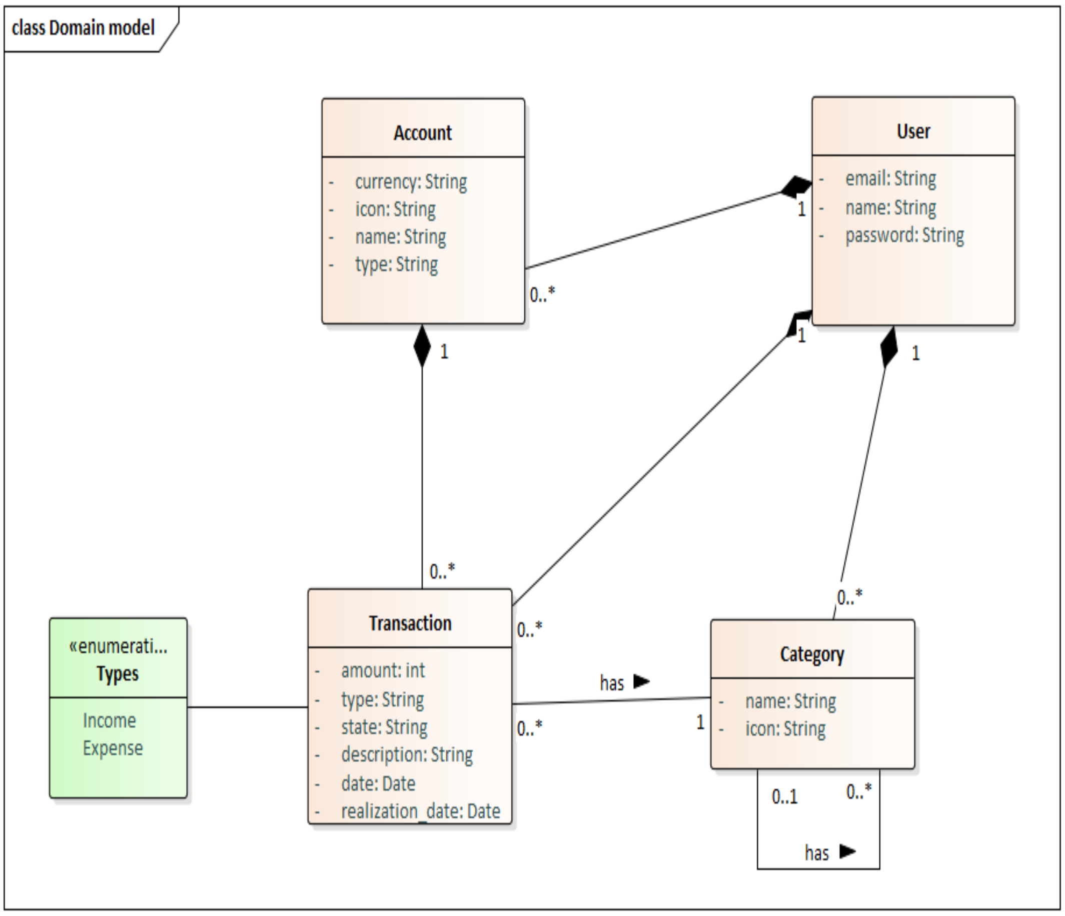


Figure 1 Current domain model

2.3 Current implementation

This section describes the current architecture of the application.

2.3.1 Backend

The entire backend part of the application, which takes care of working with data, was programmed in PHP with Laravel framework. The application is based on a three-layer architecture. The layers are divided into:

- **Presentation layer** - Takes care of displaying information to the user, often using HTML pages, which are accompanied by CSS styles. This layer can control user input and take care of changes in the business layer or in view.
- **Business layer** - Takes care of the application's logic and is independent of the presentation and data layers. It will provide for methods of accessing and maintaining data, and it might accommodate the movement of work from one function to another.
- **Data layer** - Enables a communication between the application and the data storage.

2.3.2 Frontend

The application's front end uses the Blade templating engine for PHP, created specifically for the Laravel framework. The styling of the website is complemented by styles using the CSS language and the Bootstrap package. JavaScript with the jQuery library takes care of the dynamics of the site.

2.4 New Requirements

The new requirements need to be implemented to achieve the pursued goals. First of all, the missing requirements that were set at the initial stage:

- **FR1: Sharing payment account with another user**

  This functionality should enable sharing a user's payment account with other users to keep track of incomes/expenses in a common account. Each of the shared users may have different permissions.

- **FR2: Creating repeating transactions**

  Most of the users will have some transactions that will repeat on certain days. In the current implementation, the users will have to create them manually every time. In order to automate this process, the application should be able to create repeating transactions with custom recurrence patterns.

In the current version of the application, a user needs to create two separate 'income' and 'expense' transactions with two different accounts to transfer money between them. In addition, the user needs to assign a category to them, which complicates the process. To ease the process, the application should have a transaction of type 'transfer'. This type of transaction could have several use cases, such as withdrawing money from a bank account to your wallet:

- **FR3: Transfer between payment accounts**

  Add a new 'transfer' transaction type

The application should be able to show the overall balance of each payment account. When a user creates a transaction, the balance should change according to the transaction type. Also, it should be possible to set the initial balance when creating an account::

- **FR4: Calculate balance of the payment accounts**

  Add balance property to account. The balance should be updated after each transaction in this account.

If you're like me, nearly every Amazon order may go to various budget categories. It is not possible to assign several categories with different amounts to the transactions in the current version. Having this feature, users should be able to split a transaction into multiple sub-transactions. Each of the subtransactions may have a their own category and amount spent:

- **FR5: Create/Show/Update/Delete Sub-transactions**

  Functionality to split a transaction into several subtransactions.

One of the goals of the application should be providing a way for users to eliminate debt, save money, and reach their financial goals faster. Adding budgeting functionality can help with that:

- **FR6: Budgeting**

  Users can create daily/weekly/monthly/yearly budgets for specific categories to track their spending.

In addition to all of the above:

- **FR7: Create/Show/Update/Delete Project**

  Projects may be used to unite transactions into one group. They may be helpful if you're going on a trip or another special event.

- **FR8: Refresh token**

  In order not to force users to log in every hour, refresh token may be used by the web or mobile apps.

2.5 New domain model

The domain model of the initial state of the application was shown in the figure [1]. Now an extended version of the domain model is introduced in the figure [2]. The new version of the domain model now covers all the requirements of the applications described above. Only new and updated entities are described in this section.

2.5.1 Updated entities

2.5.1.1 Account

Account now has balance and starting balance attributes. Starting balance represents the initial balance when the account was created. Balance is then calculated by adding starting balance and adding or subtracting each account's transactions—also, currency and type of an account represented by enum types.

2.5.1.2 Transaction

The Transaction entity now has a new type, transfer, to represent transaction between two accounts. In addition, now transactions may be a part of a Project. Moreover, the enum type is used now to show the state of a transaction in the model. Finally, Transactions may have sub-transactions.

2.5.2 New entities

2.5.2.1 Transaction template

Transaction template is a new type of transactions which are repeating transactions. User can create a template with recurrence patterns when a transaction is going to be repeated. Moreover, users may choose if repeating transactions are going to be created automatically or manually confirm to create them.

2.5.2.2 Access

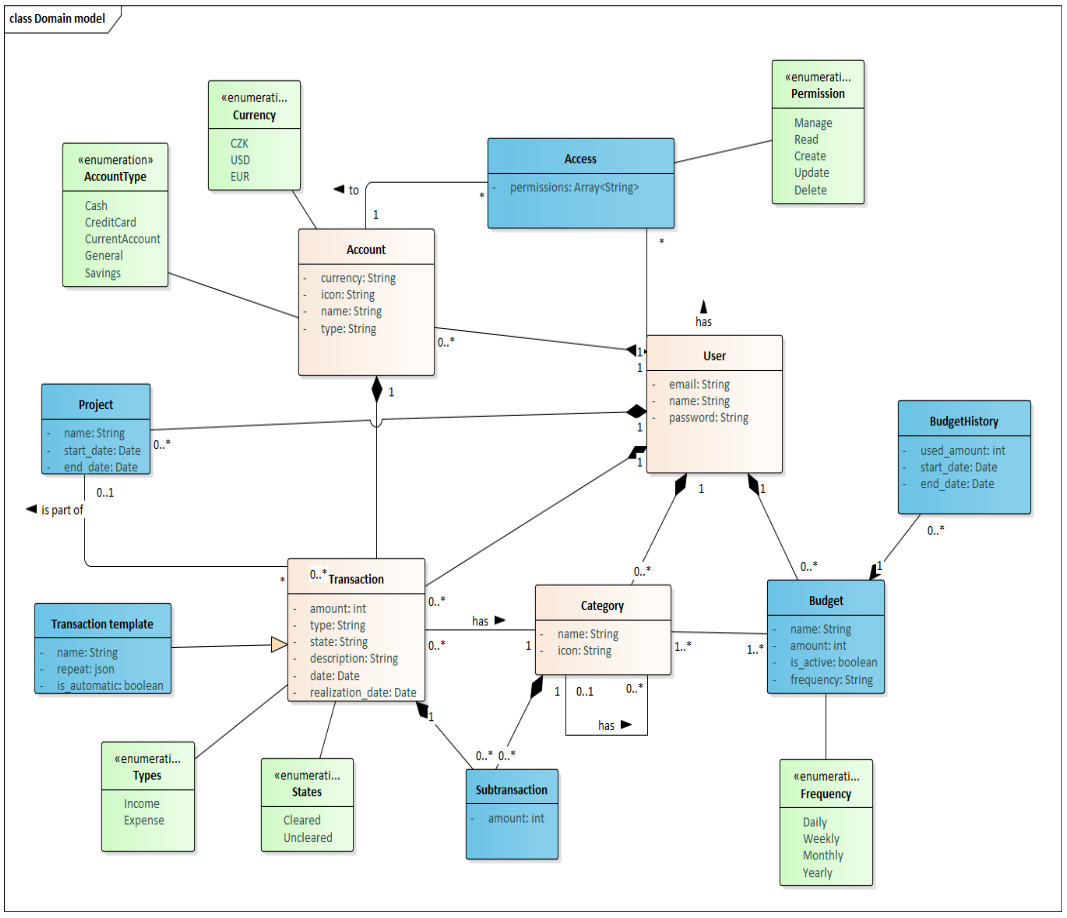
Access entity represents accesses to shared accounts. It consists of permissions for shared users. The owner of an account may choose which permissions to give to other users when sharing the account.

2.5.2.3 Budget

The Budget entity will be used to create repeating budget plans. Budgets consist of a name, a repeat frequency, an amount the user wants to spend.

2.5.2.4 Project

Projects are used to unite transactions into one group. Projects consist of a name, start and end dates.



■ **Figure 2** New domain model

2.6 Existing Solutions

As this is not a new idea, several applications used for similar or the same purposes already exist. This section will introduce some competing applications. Information about competing applications was obtained directly from the applications' website.

1. **Mint** – is one of the popular personal finance apps that provides your complete financial picture in one place. Once you link your bank accounts, Mint pulls your transactions, categorizes them, and adds them to the application. After that, you can keep track of your bills and spendings and create a budget if needed. You can also add transactions manually if you do not have one of your accounts connected but want to count its spending towards your budget. The majority of Mint's services are free to customers. However, Mint has monetized its free product by including advertisements on various parts of its website and app. In addition, Mint also generates revenue through the sale and distribution of aggregate consumer data. Premium version costs $16.99 per month and offers ad-free experience and credit monitoring service feature. [16]

2. **Spendee** – Spendee allows you to create shared wallets with other users that you can use to manage shared expenses. You can import your bank transactions and let the application categorize them for you. You can also add expenses manually. In addition, the application offers budgeting for each spending category.

   The free version allows only manual tracking with one payment account and one budget. Plus version costs $1.99 per month and removes the limit for the number of accounts and budgets. Premium version costs $2.99 monthly and offers sharing wallets with others and automatic bank accounts synchronization. [23]

3. **Wallet by BudgetBakers** – Wallet is one of the most popular personal finance apps in the Apple App Store and Google Play Store. It also can be used through its web interface. However, the Android version of the app is the complete one, while the iOS and web versions lack some of its useful features. [3]

   Wallet's free features on Android include:

   - Categorize spending. Each category has a list of subcategories and can be personalized
   - Saving goals and budgeting. You can set a budget for a specific account or category and Wallet will notify you when you go beyond it
   - Planned payments

   Here's what Wallet's premium option adds to its free version:

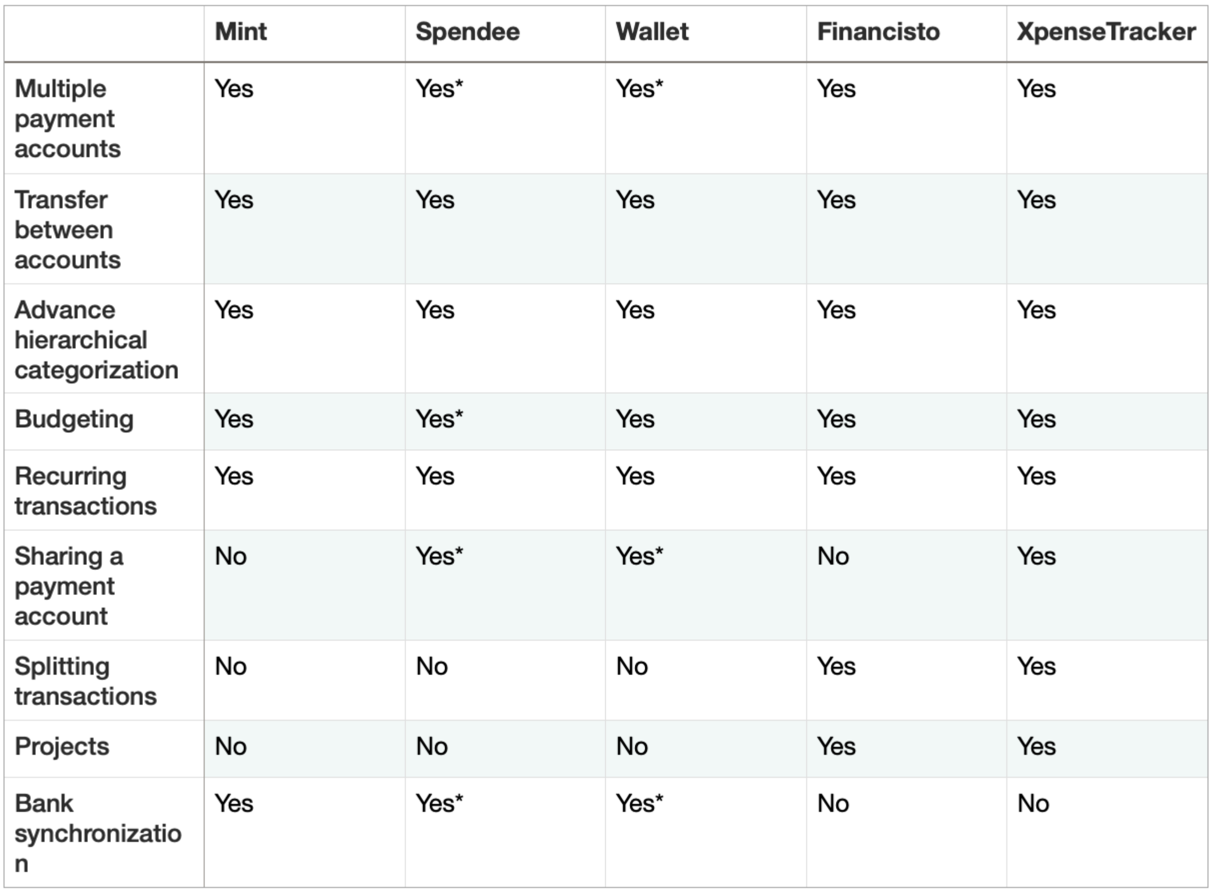
   - Bank synchronization—Transactions will automatically be added to Wallet and categorized
   - Unlimited accounts
   - Multi-user sharing

4. **Financisto** – is a free, open-source personal finance manager for the Android platform. It has a lot of features, but the last version of the application was released two years ago. [22] It has following features:

   - Multiple accounts
   - Scheduled and recurring transactions
   - Transfers
   - Splitting transactions
   - Hierarchical categories with custom attributes
   - Projects to organize transactions
   - Recurring budgets

The result of this analysis is that the ExpenseTracker application has all the main features of other applications. Unfortunately, the application is missing a bank synchronization feature, but even though getting API access to the banks is difficult, it may be implemented in the feature. In a bright sight, ExpenseTracker has additional feature compared to most of the above-mentioned applications, such as projects functionality for one time events and splitting a transaction into subtransactions.

The following table overview in the [3] contains a comparison of the application with the applications mentioned above. Paid features are marked with the character "*".



	Mint	Spendee	Wallet	Financisto	XpenseTracker
Multiple payment accounts	Yes	Yes*	Yes*	Yes	Yes
Transfer between accounts	Yes	Yes	Yes	Yes	Yes
Advance hierarchical categorization	Yes	Yes	Yes	Yes	Yes
Budgeting	Yes	Yes*	Yes	Yes	Yes
Recurring transactions	Yes	Yes	Yes	Yes	Yes
Sharing a payment account	No	Yes*	Yes*	No	Yes
Splitting transactions	No	No	No	Yes	Yes
Projects	No	No	No	Yes	Yes
Bank synchronizatio n	Yes	Yes*	Yes*	No	No

Figure 3 Comparison table

Chapter 3
Design

3.1 Technologies

This section contains the technologies used during the development process and why exactly these technologies are chosen.

3.1.1 Programming language and Run-time environment

The Backend of the initial version of the application was developed using PHP, mostly synchronous language and hardly scalable. In addition, PHP does not offer strict type checking. Moreover, the application itself had many bugs, and most of the code had to be refactored.

As I was interested in learning Node.js - which solves all the above issues of PHP , I decided to use the TypeScript programming language with Node.js runtime environment.

3.1.1.1 Node.js

Node.js is an open-source, cross-platform, JavaScript runtime environment built on Chrome's V8 JavaScript engine that executes JavaScript code outside of a browser. It was created in 2009, and it is used for building fast and scalable applications. [18] Advantages of Node.js:

- **Non-blocking code**

  Due to its asynchronous, non-blocking input/output processing and event-driven nature, Node.js is a suitable choice for building modern solutions based on WebSockets, microservices, event queues, and jobs. [18]

- **Faster performance**

  As I mentioned above, it has a non-blocking input/output model. So that makes processing the requests very fast. [18]

- **Community**

  Node.js has the most significant open-source community and the largest repository of libraries and packages. [18]

3.1.1.2 TypeScript

TypeScript is an open-source language that builds on JavaScript, one of the world's most used languages. [26] In addition to above-mentioned advantages of JavaScript, TypeScript adds additional features on top of those:

- **Transpiler**

  TypeScript has a source-to-source compiler that converts TypeScript code into JavaScript code, allowing early detection of code errors. [26]

- **Type checking**

  TypeScript allows for static type checking during compilation, so locating code errors becomes much more manageable. Types provide a way to describe an object's shape, provide better documentation, and allow TypeScript to validate that your code is working correctly. [26]

3.1.2 Frameworks

This section describes frameworks and technologies which were used in the development of ExpenseTracker application:

3.1.2.1 Nest.js

Nest is a framework for building practical, scalable Node.js server-side applications. It created with TypeScript, preserves compatibility with pure JavaScript, and combines OOP, FP, and FRP elements. Nest aims to provide an application architecture out of the box which allows for effortless creation of highly testable, scalable, loosely coupled and easily maintainable applications. Recently, the NestJS framework has been gaining extreme popularity due to its incredible features. [17] Some of them are:

- Easy to use, learn and master.
- Powerful CLI to boost productivity and ease of development.
- Detailed and well-maintained documentation.
- Active codebase development and maintenance.
- Open-source .
- Support for dozens of nest-specific modules that help you easily integrate with common technologies and concepts like ORM, GraphQL, Logging, Validation, Caching, WebSockets and much more.
- Easy unit-testing applications.

3.1.2.2 MikroORM

MikroORM is a free, open-source ORM library for TypeScript for Node.js. ORMs work as querying and object creation runtime between client code and relational database. They provide multiple powerful features like object and query caching, concurrency control, object-oriented query languages and much more. MikroORM itself was built with TypeScript and based on Data Mapper, Unit of Work and Identity Map patterns. The first and most important implication of having a Unit of Work is that it automatically handles transactions. In addition, thanks to Identity Map, the application will always have only one instance of a given entity in one context. That allows skipping loading already loaded entities, as well as comparison by identity. All of that simplifies the implementation of the data layer; that is why this framework was chosen. [2]

3.1.3 Databases

3.1.3.1 Maria.db

MariaDB is a freely available open-source RDBMS that uses Structured Query Language. MariaDB is a forked version of MySQL and is a general-purpose DBMS engineered with extensible architecture to support a broad set of use cases via pluggable storage engines. Many features contribute to MariaDB's popularity as a database system. Its speed is one of its most prominent features. MariaDB is also remarkably scalable and can handle tens of thousands of tables and billions of rows of data. It can also manage small amounts of data quickly and smoothly, making it convenient for small businesses or personal projects. Another feature that sets it apart from its predecessors is its focus on security. These are the reasons why it is used in the development of this application. [15]

3.1.3.2 Redis

Redis is an open-source, in-memory data structure store used as a database, cache, and message broker. It supports data structures such as lists, strings, hashes, sets, bitmaps, and geospatial indexes with radius queries. Because of its versatility, Redis was used as a database cache and as a job queue in this application. [21]

3.1.4 Platform as a Service

PaaS, or Platform-as-a-Service, is a cloud computing model that provides customers with a complete platform—hardware, software, and infrastructure—for developing, running, and managing applications without the cost, complexity, and inflexibility of building and maintaining that platform. With PaaS, developers only have to worry about managing the applications or software they develop; the PaaS provider handles everything else, including any platform maintenance, development tools, and database management. As a result, they can save the time and money needed to maintain, upgrade, or replace systems and software. In this application, this technology was used for the deployment. [7]

3.1.5 Database as a Service

DBaaS or Database-as-a-Service is a cloud computing service model that provides users with access to a database without the need for setting up physical hardware or installing software. All of the organisational tasks and maintenance are taken care of by the service providers so that all the user needs to do is use the database. The DBaaS is used in the application as a cloud database that connects to the PaaS where the application is deployed. [25]

3.2 Architecture

The ExpenseTracker application consists of a central database and the Node.js web server. The application and the databases can run on different servers or share the same. Also, the application and database can be deployed on PaaS. The possible deployment model of the application is shown in the figure [4].

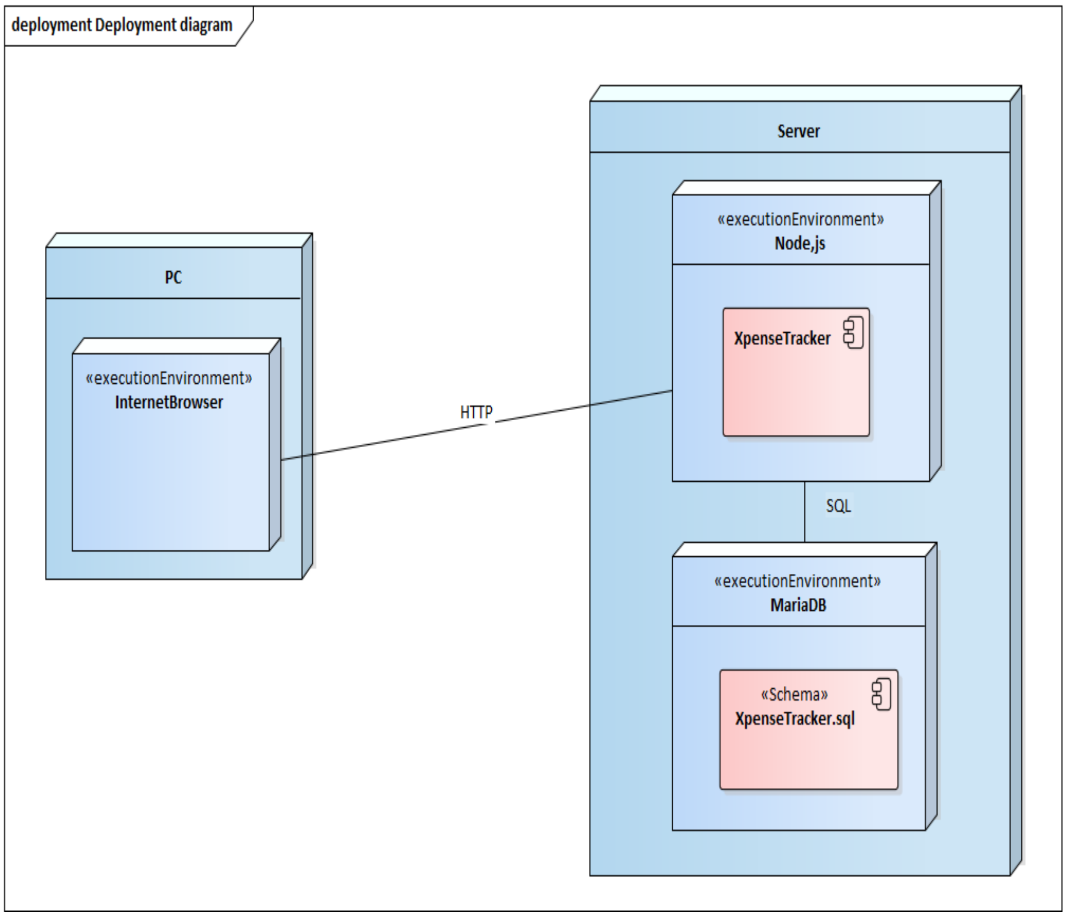


Figure 4 Deployment diagram

The application architecture is designed using the principle of Monolithic architecture. Monolithic applications are a codebase that performs as a single unit. Developers do not need to think about the communication between the various application's components. Thus, they require less time for building such apps. In addition, it's faster to deploy such applications because of a single unified codebase. Monolithic architectural style also has some weaknesses. It is hard to scale the application. When an app's component requires more resources, it is difficult to isolate such a component for independent scaling. Also, it might be more difficult to keep adding more features to the application as the size and complexity grows. [4]

The logical architecture of the application is shown in the figure [5].The application consists of three basic layers.

- Controller — provides the REST API and handles client requests,
- Service — contains business logic
- Repository — abstracts access to the database

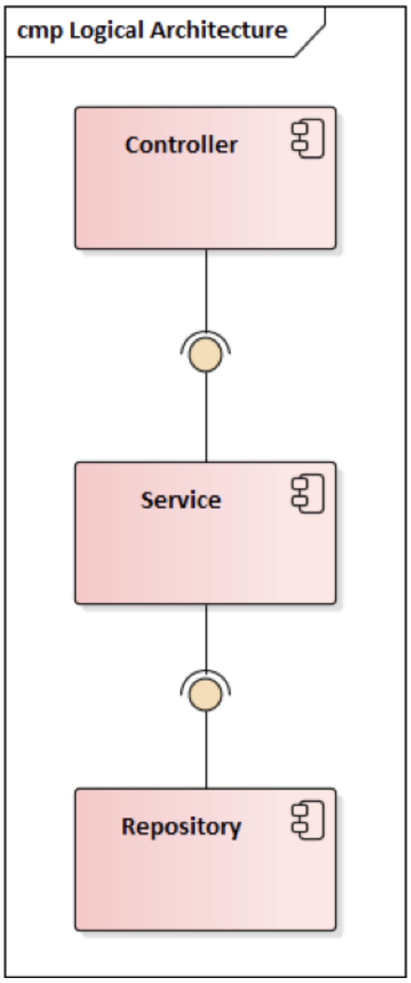


Figure 5 Logical architecture

3.3 Database

The database model, like the domain model, is displayed using a UML diagram. The database models differ from the domain model in that it contains specific information about attributes, relationships, foreign keys, and database implementation. The model helps programmers to better orient themselves in the given application structure. In the diagram [6], the new database schema is shown.

- The **users** table represents the registered user in the system. It consists of a name, an email, a hashed password and a token version to revoke old tokens.
- The **accounts** table defines payment accounts. It contains account type and currency, starting and current balance, and an icon name. In addition, accounts belong to a user. Users and accounts tables are joined via user id foreign key.
- Shared accounts between users are realized by **accesses** table. The table has a shared account id, the ids of the users with whom the account is shared, and permissions.
- The **categories** table contains a name and an icon. A self-relation implements the subcategories — defined by `parent_category_id` foreign key.
- Budgeting functionality implemented through **budgets** and **budget_history** tables. The budgets table has general information about a budget: a recurrence interval and the amount user wants to spend in the given interval. Budget instances for each recurrent interval is saved in the **budget_history** table. This table contains the amount spent and the budget id. Finally, categories defined for budgets represented by **categories_budgets**.
- The **projects** table represents a list of projects. A project has a name, start and end date. In addition, a column is_active may be used to filter active and archived projects.
- **Planned_transaction_templates** and **planned_transactions** tables describe repeating transactions. The template of the repeating transaction is saved in **planned_transaction_templates** table with necessary fields to create a transaction in the **transactions** table. All the future dates when the transaction will occur is then saved in **planned_transactions** table. **is_automatic** column in **planned_transaction_templates** table denotes he information if future transactions should be created automatically or confirmed manually by the user in order to create them.
- **T**ransactions created manually by the user and repeating transactions created from the template are saved in transactions table. The table consists of the amount of a transaction, short description, transaction type and state. Also, it has the transaction date when the transaction has occurred and the realization date. Every transaction should have an account and a category (except when the transaction type is transfer). The information about the transfer transaction is saved in the **transfers** table. In addition, a transaction may belong to a project. Finally, the **transactions** table has a one-to-many relationship with the **subtransactions** table. The tables are joined via the **transaction_id** foreign key in the **subtransactions** table.

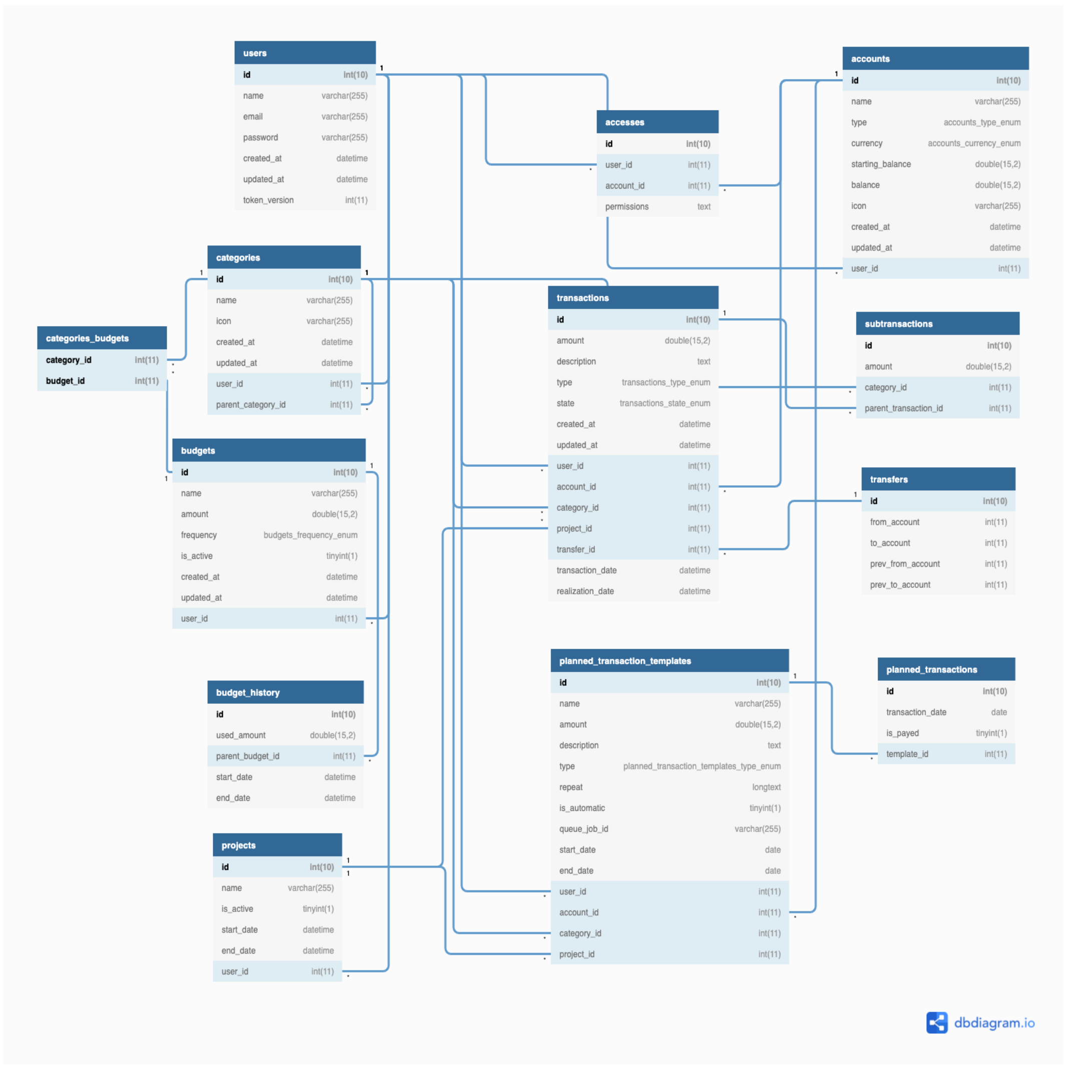


Figure 6 Database diagram

3.4 Class model

In this section will be shown examples of classes using class models.

3.4.1 Planned Transaction Service Class

As it was discussed before, the application consists of three layers. As the controller layer provides the REST API and handles client requests, all the business logic contains in the service layer. The figure [7] shows the Transaction Service Class, which was decided to be shown as an example as one of the most nontrivial classes. All the service classes have a similar structure.

The class implements all the functionality related to creating transactions. Usage of repository pattern allows isolating the business layer from the data layer. This allows the service just to handle the managing of repeating transaction templates and not overwhelm the class with additional functionalities.

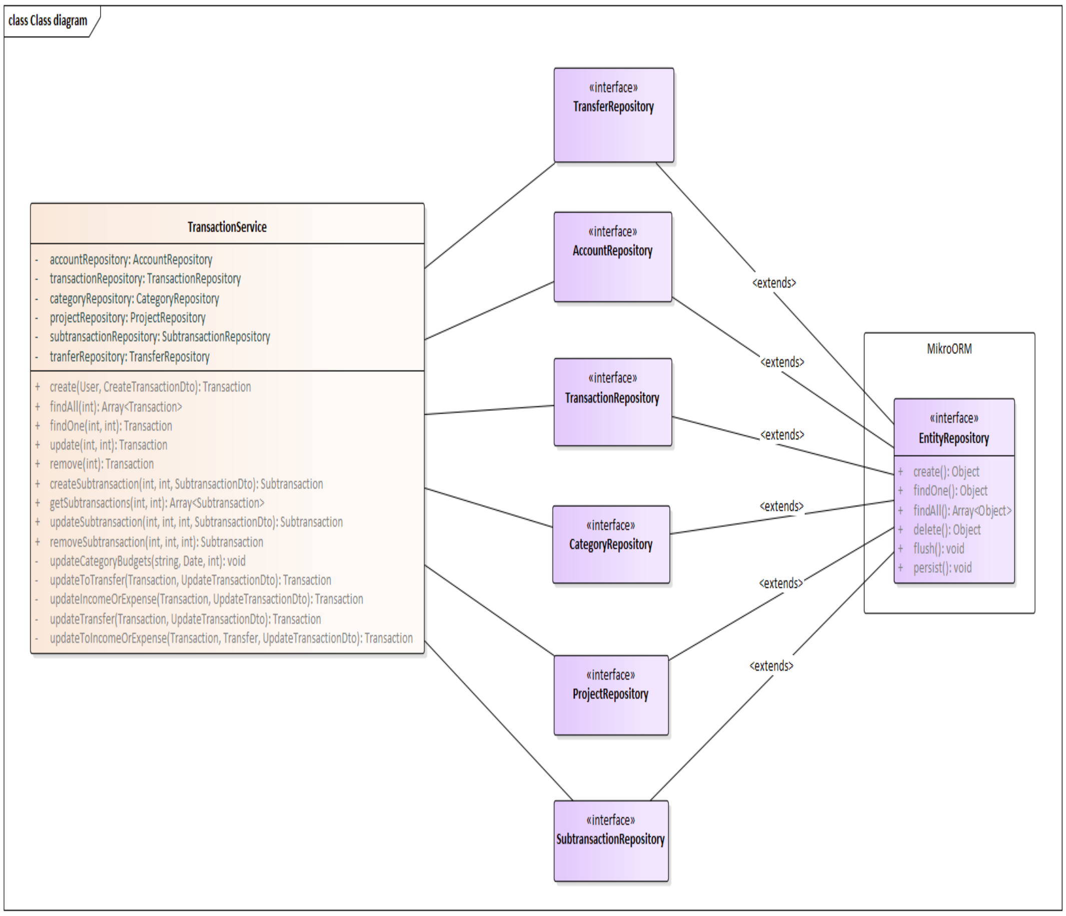

class Class diagram
«interface»
TransferRepository
TransactionService
- accountRepository: AccountRepository
- transactionRepository: TransactionRepository
- categoryRepository: CategoryRepository
- projectRepository: ProjectRepository
- subtransactionRepository: SubtransactionRepository
- tranferRepository: TransferRepository
+ create(User, CreateTransactionDto): Transaction
+ findAll(int): Array<Transaction>
+ findOne(int, int): Transaction
+ update(int, int): Transaction
+ remove(int): Transaction
+ createSubtransaction(int, int, SubtransactionDto): Subtransaction
+ getSubtransactions(int, int): Array<Subtransaction>
+ updateSubtransaction(int, int, int, SubtransactionDto): Subtransaction
+ removeSubtransaction(int, int, int): Subtransaction
- updateCategoryBudgets(string, Date, int): void
- updateToTransfer(Transaction, UpdateTransactionDto): Transaction
- updateIncomeOrExpense(Transaction, UpdateTransactionDto): Transaction
- updateTransfer(Transaction, UpdateTransactionDto): Transaction
- updateToIncomeOrExpense(Transaction, Transfer, UpdateTransactionDto): Transaction
«interface»
AccountRepository
«interface»
TransactionRepository
«interface»
CategoryRepository
«interface»
ProjectRepository
«interface»
SubtransactionRepository
<extends>
<extends>
<extends>
<extends>
<extends>
<extends>
MikroORM
«interface»
EntityRepository
+ create(): Object
+ findOne(): Object
+ findAll(): Array<Object>
+ delete(): Object
+ flush(): void
+ persist(): void

Figure 7 Class model diagram

3.5 Sequence model

This section contains the sequence diagrams for a sampled use case scenario. As an example Create Repeating Transaction scenario is chosen.

3.5.1 Create Repeating Transaction

A user wants to add a new repeating transaction to the system. This scenario starts when the user has already logged in. First, the user should fill in all the information about the repeating transaction he wants to add. It contains the name, account, category, amount, short description, transaction type, recurrence pattern and the check if the transactions should be created automatically. All the information is collected and sent to the '/api/templates' endpoint via a POST request. Then the PlannedTransactionController validates the request and sends the request body to the PlannedTransactionService. After that, PlannedTransactionService creates a new TransactionTemplate instance and asks TransactionTemplateRepository to save the provided entity to the database. After successfully saving new data to the database, a new template entity is returned in the response. Meanwhile, the application asynchronously sends a message to PlannedTransactionProccessor to calculate future transaction dates. The diagram [8] shows the sequence diagram of this process.

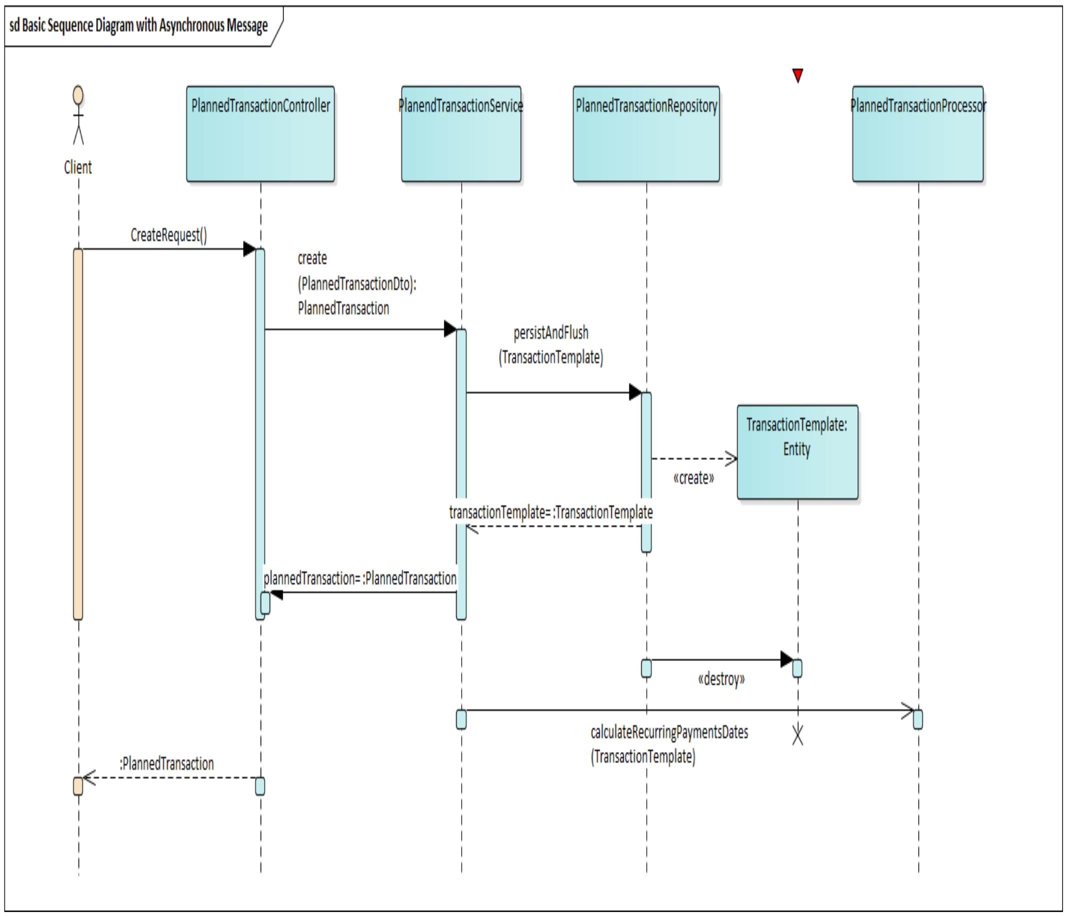

sd Basic Sequence Diagram with Asynchronous Message
Client
PlannedTransactionController
PlanendTransactionService
PlannedTransactionRepository
PlannedTransactionProcessor
CreateRequest()
create
(PlannedTransactionDto):
PlannedTransaction
persistAndFlush
(TransactionTemplate)
TransactionTemplate:
Entity
«create»
transactionTemplate=:TransactionTemplate
plannedTransaction=:PlannedTransaction
«destroy»
calculateRecurringPaymentsDates
(TransactionTemplate)
:PlannedTransaction

Figure 8 Class model diagram

Chapter 4
Implementation

In this chapter, implementation details, testing of the application and the list of tools used during development are discussed.

4.1 Implementation Details

This section goes into detail regarding the implementation of some parts of the application.

4.1.1 Structure

The code structure of the application is divided into 3 main parts:

- **Modules** – they are used to organize the code and split features into logical reusable units. Grouped TypeScript files are decorated with "@Module" decorator, which provides metadata that Nest makes use of to organize the application structure.
- **Services** – also called providers, which are designed to abstract any form of complexity and logic. Services can be created and injected into controllers or other services.
- **Controllers** – they are responsible for handling incoming requests and returning appropriate responses to the client-side of the application.

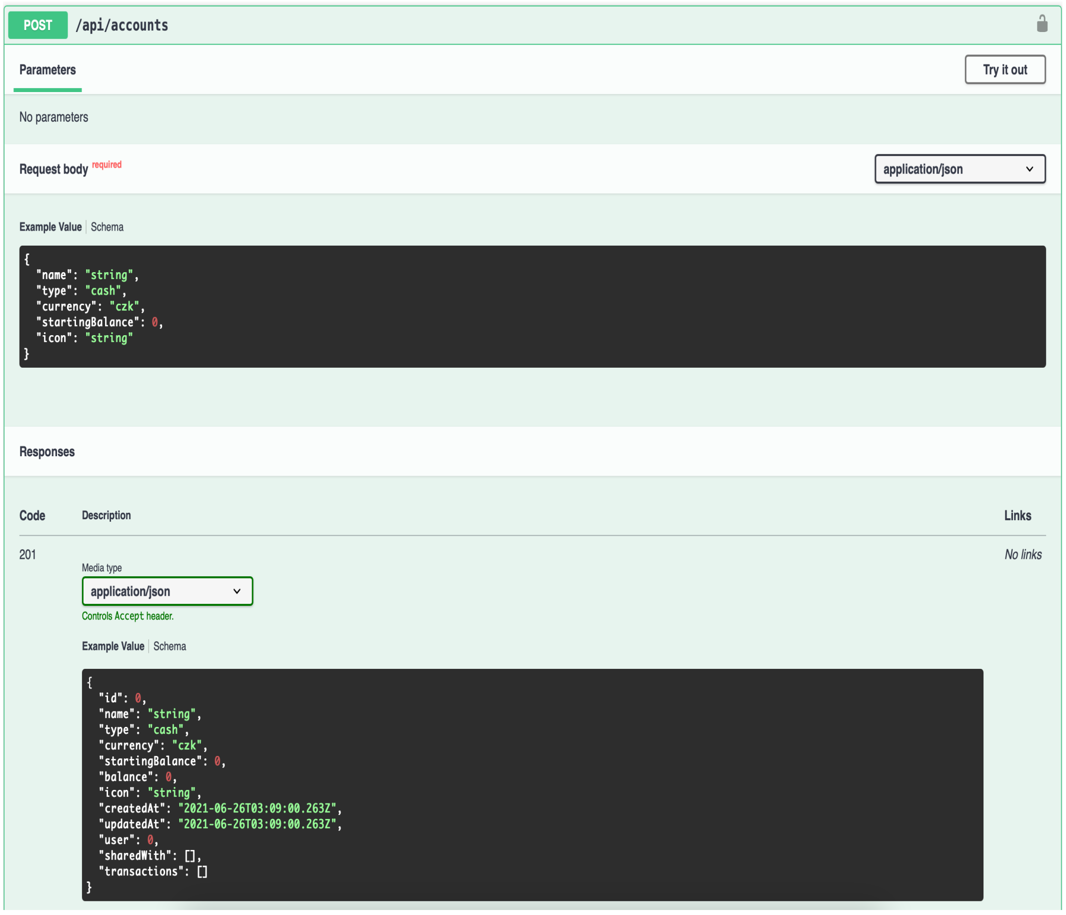
4.1.2 API documentation

As this application provides REST API, the consumers of this API need to know how the model looks like and which operations are available to be called. Furthermore, client applications can be implemented in different technologies. One of the platform-independent solutions is OpenAPI specification. The OpenAPI Specification defines a standard, language-agnostic interface to RESTful APIs, which allows both humans and computers to discover and understand the service's capabilities without access to source code or documentation. When properly defined, a consumer can understand and interact with the remote service with minimal implementation logic. [24]

The OpenAPI specification is usually defined in JSON or YAML file. The specification file permits software developers to define their API's essentials, including:

- Present endpoints and each endpoint's operations
- The input and output operation parameters
- Authentication techniques

To visualize and interact with the API's resources, Swagger UI is used. It's automatically generated from OpenAPI Specification, with the visual documentation making it easy for client-side consumption. In the diagram [9], documentation of one of the endpoints is shown. The full version of ExpenseTracker API documentation can be found on `lhttps://xpensetracker-api.herokuapp.com/api/`.



```
{
  "name": "string",
  "type": "cash",
  "currency": "czk",
  "startingBalance": 0,
  "icon": "string"
}
```

```
{
  "id": 0,
  "name": "string",
  "type": "cash",
  "currency": "czk",
  "startingBalance": 0,
  "balance": 0,
  "icon": "string",
  "createdAt": "2021-06-26T03:09:00.263Z",
  "updatedAt": "2021-06-26T03:09:00.263Z",
  "user": 0,
  "sharedWith": [],
  "transactions": []
}
```

Figure 9 Swagger UI

4.1.3 Authentication

The application uses JSON Web Tokens to authenticate users. JWT find their applications in various authentication mechanisms. These are typically passed in the Authorization header when a user submits a request to the client. It is a JSON encoded representation of a claim that can be transferred between two parties. Then, the claim is digitally signed by the issuer of the token, and the party receiving this token can later use this digital signature to prove the ownership on the claim. [10]

JSON Web Tokens can be broken down into three parts: header, payload, and signature.

4.1.3.1 Header

The information contained in the header describes the algorithm used to generate the signature. The decoded version of the header from may look like:

```
    {
    "alg": "HS256",
    "typ": "JWT"
    }
```

HS256 is the hashing algorithm HMAC SHA-256 used to generate the signature in the above example.

4.1.3.2 Payload

All the claims within JWT authentication are stored in this part. Claims are used to provide authentication to the party receiving the token. The decoded version of the payload from the JWT example may look like:

```
{
    "userId": "12345",
}
```

The ‘userId’ field is used to identify the user to whom the token was issued to.

4.1.3.3 Signature

The signature part of a JWT is derived from the header and payload fields. The steps involved in creating this signature are described below:

- Combine the base64url encoded representations of header and payload with a dot
- Hash the above data with a secret-key only known to the server issuing the token. The hashing algorithm is the one described inside the header
- Base64Url encode the hash value obtained from the step above

Because the secret key is only known to the server, only it can issue new tokens with a valid signature. Users can not forge tokens as producing a valid signature for the token requires the knowledge of the secret key. But how to invalidate a single token? A no-effort solution is to change the server’s secret key, which invalidates all tokens. Not really nice for users that should not have their token expired for no reason. In order to solve this problem, a token version property was added to the user object in the database to reference the current version of the token, and it is passed in the payload of a token. When users change or reset their password, the token version attribute is incremented, which invalidates all previous token issued to the user

4.1.4 Caching

Caching is a great and simple technique that helps improve applications' performance. It acts as a temporary data store providing high-performance data access. In the ExpenseTracker application, all the database queries are cached. The strategy implemented works with constants, which specify the duration after which data are considered to be ”expired” and required to be fetched from the database again.

4.1.5 Account balance calculation

Account balance calculation is one of the new requirements. Initially, it was implemented in the business layer of the application. Every time a user created a new transaction, the transaction's amount was added to or subtracted from the account of the transaction. As all the database queries were cached in the application, the request to get the account details could be the outdated version. In order to make this feature work, I decided to use SQL Triggers. The trigger is a statement that a system executes automatically when there is any modification to the database. In a trigger, we first specify when the trigger is to be executed and then the action to be performed when the trigger executes. [8] So after each change in the ‘transactions‘ table, the trigger is executed, which updates the balance of an account. Below is an example of one of the triggers in the transactions table:

```
begin

    SELECT balance
    INTO @account_balance
    FROM accounts
    WHERE id = NEW.account_id;

    IF NEW.type = 'income' THEN
        SET @account_balance = @account_balance + NEW.amount;
        UPDATE accounts set balance = @account_balance
        WHERE id = NEW.account_id;
    END IF;

    IF NEW.type = 'expense' THEN
        SET @account_balance = @account_balance - NEW.amount;
        UPDATE accounts set balance = @account_balance
        WHERE id = NEW.account_id;
    END IF;

    IF NEW.type = 'transfer' THEN

        SELECT from_account
        INTO @fromAccount
        FROM transfers
        WHERE id = NEW.transfer_id;

        IF NEW.account_id = @fromAccount THEN
            SET @account_balance = @account_balance - NEW.amount;
            UPDATE accounts SET balance = @account_balance
            WHERE id = NEW.account_id;
        END IF;

        IF NEW.account_id != @fromAccount THEN
            SET @account_balance = @account_balance + NEW.amount;
            UPDATE accounts set balance = @account_balance
            WHERE id = NEW.account_id;
        END IF;
    END IF;

end
```

4.1.6 Repeating transactions

In order to create a repeating transaction, predefined structure of recurrence pattern has to be sent by the clients. The repeating transaction can be repeated by frequencies:

- Daily
- Weekly
- Monthly
- Yearly

In addition to above:

- If transaction is weekly, it is possible to define weekdays where the recurrence will be applied.
- If transaction is monthly, it is possible to define month days.
- If transaction is yearly, it is possible to define month days as well as months.

Finally, it is possible to defines intervals to the frequencies. For example, when using the weekly rule, an interval of two means once every two weeks.

After receiving the recurrence pattern, the application calculates the dates when a transaction will occur and saves these days to the database. Currently, dates in 10 years advance are calculated. Finally, to create transactions automatically on each of the estimated dates, the CRON job is used. A cron job is a Linux command used for scheduling tasks to be executed sometime in the future. It is usually used to schedule a job that is executed periodically. [12] The repeat pattern of this CRON job will be similar to the recurrence pattern of the repeating transaction. This cron job will run a function, which will create a transaction.

One of the issues working on the implementation of this feature was timezone support. Currently, the application supports only Central European Timezone, but it will be possible to add additional time zones support in the future.

4.1.7 Sharing accounts

One of the problems during the development was the implementation sharing an account with other users. The correct implementation has to be chosen, or otherwise, the application could have serious security problems. A claim based authorization is used to solve this problem. When an identity is created, it may be assigned one or more claims issued by a trusted party. A claim is a name-value pair representing what the subject can do, not what the subject is. When sharing an account, the user invites a new user via email and assigns claims. The following claims are available:

- Manage – Complete access to the account. Invited users can create, update and delete transactions of the account.
- Read – It is the default assigned permission. Invited user can see the transactions.
- Update – The transactions of the shared account can only be updated by the invited user.
- Delete – Invited users are only able to delete the transactions of the shared account.

Using the example below, it is possible to check which action on a specified account is allowed for the user. As you probably guessed, can and cannot accept the same arguments but has different meanings, can allow to do an action on the specified subject and cannot forbids.

```
async createForUser(userId: number) {
  const { can, build } = new AbilityBuilder<Ability<[Action, Subjects]>>
    (Ability as AbilityClass<AppAbility>)

  can(Action.Manage, Account, { user: userId });

  const accesses = await this.accessRepository.find({ user: userId }, [
    'account',
  ]);

  accesses.forEach((access) => {
    access.permissions.forEach((permission: Action) => {
      can(permission, Account, { id: access.account.id });
    });
  });

  return build({
    detectSubjectType: (item) =>
      item.constructor as ExtractSubjectType<Subjects>,
  });
}
```

4.2 Testing

In this section, the unit testing and manual user testing of the ExpenseTracker application are discussed.

4.2.1 Unit tests

One of the reasons why the initial version of the ExpenseTracker application contained a number of problems is that the code was not covered with unit tests. With the new functionalities being implemented, the already existed ones were not tested properly to confirm that the changes did not cause any unexpected behaviour. That is why the main functions of the application are now covered with the unit tests. Although, some parts of the application was difficult to unit test. For example, create a repeating transaction. Such functionalities were only tested manually. The unit testing is done with the help of Jest, a JavaScript unit testing framework. All the tests can be run during the build phase of the application or run manually. This is an example of the test of create transfer method of class TransactionService:

```
        const { balance: accountToBefore } = (
            await DI.em
                .getRepository(Account)
                .createQueryBuilder()
                .select('*')
                .where({ id: defaultData.toAccount.id })
                .limit(1)
                .getResult()
        )[0];
        const { balance: accountFromBefore } = (
            await DI.em
                .getRepository(Account)
                .createQueryBuilder()
                .select('*')
                .where({ id: defaultData.fromAccount.id })
                .limit(1)
                .getResult()
        )[0];
        const response = await api
            .post('/api/transactions/')
            .auth(defaultData.token, { type: 'bearer' })
            .send(transferNew);
        expect(response.status).toBe(201);
        const { balance: accountTo } = (
            await DI.em
                .getRepository(Account)
                .createQueryBuilder()
                .select('*')
                .where({ id: defaultData.toAccount.id })
                .limit(1)
                .getResult()
        )[0];
        const { balance: accountFrom } = (
            await DI.em
                .getRepository(Account)
                .createQueryBuilder()
                .select('*')
                .where({ id: defaultData.fromAccount.id })
                .limit(1)
                .getResult()
        )[0];
        expect(accountFrom).toBe(
            accountFromBefore - response.body.data[0].amount
        );
        expect(accountTo).toBe(
            accountToBefore + response.body.data[1].amount
        );
    });
```

4.2.2 User testing

Even though all the parts of an application could be covered by tests, there still will be some bugs not discovered by them. This is why User testing is important. As mentioned in the Introduction chapter, the frontend of the application is being developed by another student, Amir Qamili. Using the application during his development process, he reported some bugs, which were fixed. Current functionalities that were tested by him while developing a user interface for them:

- User authentication
- Accounts
- Categories
- Transactions and subtransactions

4.3 Tools

This section contains the list of tools used during the implementation process.

4.3.1 Node Package Manager

NPM – or "Node Package Manager" – is the default package manager for Node.js. NPM consists of two main parts:

- Online repository that hosts JavaScript packages
- Command-line utility for interacting with said repository that aids in package installation, version management, and dependency management [19]

4.3.2 Jest

Jest an open source project maintained by Facebook, and it's especially well suited for JavaScript code testing, although not limited to that: it can test any TypeScript code as well. Its strengths are:

- It is fast and safe – By ensuring your tests have a unique global state, Jest can reliably run tests in parallel. To make things quick, Jest runs previously failed tests first and re-organizes runs based on how long test files take.
- Provides easy mocking – Jest uses a custom resolver for imports in your tests, making it simple to mock any object outside of a test's scope. [14]

4.3.3 Gitlab

Git is a source code versioning system that lets you locally track changes and push or pull changes from remote resources. To keep track of the changes during the software development process, GitLab was used as a Git repository hosting service. [9]

4.3.4 Heroku

During the development, the new changes to the application are deployed to Heroku. Heroku provides a platform as a service. Here are five reasons why Heroku was chosen as a PaaS:

- Simplicity – Heroku hides all of the complexity of the servers themselves behind a friendly web-based user interface. Once your app is up and running on the platform, deployments are just a click away.
- Stability – Heroku's global platform is backed by Amazon Web Services, by far the largest cloud infrastructure provider.
- Security – By abstracting away the servers, Heroku takes on the responsibility of making sure they are secure.
- Scalability – Heroku's platform runs your code inside something it calls a dyno. A dyno is an isolated container that bundles up computing resources with a copy of the application code and its dependencies. [11]

4.3.5 JawsDB

JawsDB is a Database-as-a-Service (DBaaS) provider supplying a fully functional, fully managed, relational database for use with your application. JawsDB provides easy delivery and management of a relational database in the cloud. JawsDB provides integration with Heroku, which is one of the reasons why it is chosen as a DBaaS provider. [13]

4.3.6 Visual Studio Code

Visual Studio Code is a free and open-source code editor developed by Microsoft for Windows, Linux and macOS. It is super fast and lightweight which can be used to view, edit, run and debug source code for applications. It has great JavaScript and TypeScript support which makes it a great tool to develop Node.js applications. [5]

4.3.7 Sequel Ace

Sequel Ace is a fast, easy-to-use, open=source Mac database management application for working with MySQL and MariaDB databases. It is used as a database administration tool during the development of the application. [1]

4.3.8 Paw

Paw is a full-featured HTTP Mac client app. The application lets you test and describe the APIs you build or consume. [20]

4.3.9 Docker

Docker is a containerization platform that enables you to create, deploy, and run applications conveniently with the help of containers. Containerization is a lightweight alternative to full machine virtualization that involves encapsulating an application in a container with its own operating environment. This provides many of the benefits of loading an application onto a virtual machine, as the application can be run on any suitable physical machine without any worries about dependencies. [6]

Chapter 5
Conclusion

In this thesis, I analysed the initial version of the ExpenseTracker application. The analysis of the implemented functionalities and problems discovered during this process allowed to make a list of requirements that had to be implemented to accomplish the goals assigned to the application. Moreover, the new domain model of the application was described. Based on the new domain model and functionalities, I completed the overall picture of the application. Doing that allowed me to analyse and compare the ExpenseTracker application to the existing solutions and see the weak and strong sides of the application.

I discussed the architecture of the new version of the application and the technologies used during the development process. Moreover, class and sequence models and the new database model were also described. All of that helped to create the final version of the ExpenseTracker application

The final version of the application completely fulfils the goal of this thesis. The application provides functionalities to manage multiple personal accounts, track transactions on the accounts with advanced categorization, budgeting, and recurring payments support, and share personal accounts with other users. The application itself is deployed on Heroku on the following url: `xpensetracker-api.herokuapp.com`.

5.1 Future work

While the created application is a fully-functional, there are ways in which it could be further improved. The system works with the Central European Timezone only, but support for other time zones something that should be included. That goes hand in hand with the fact that week might start of different days in different countries. In addition, while the application has several types of transaction, it is not possible to create a one-time planned transaction. It will also be good to have more currency support and automatic currency conversion in the application. The main feature of the existing solutions on the market is Bank synchronization. Some solutions exist that provide the functionality to connect with end-customers bank accounts from across the globe, such as SaltEdge and Plaid. The application could use these solutions to offer such functionality. Finally, offering additional ways to register using third-party services such as Google or Apple will be very useful.

Bibliography

[1] Sequel Ace. Mariadb/mysql database management for macos. https://github.com/Sequel-Ace/Sequel-Ace.

[2] Martin Adámek. Typescript orm for node.js. https://mikro-orm.io.

[3] BudgetBakers. Wallet by budgetbakers. https://budgetbakers.com.

[4] Oleksandr Bushkovskyi. Things to know about monoliithic and microservices architecture. https://theappsolutions.com/blog/development/monolithic-vs-microservices/.

[5] Visual Studio Code. Code editing. redefined. https://code.visualstudio.com.

[6] Docker. Empowering app development. https://www.docker.com.

[7] IBM Cloud Educations. What is pas. https://www.ibm.com/cloud/learn/paas.

[8] GeeksForGeeks. Sql triggers. https://www.geeksforgeeks.org/sql-triggers/.

[9] GitLab. Iterate faster, innovate together. https://about.gitlab.com.

[10] Olivia Harris. Jwt authentication. https://www.softwaresecured.com/security-issues-jwt-authentication/.

[11] Heroku. Cloud application platform. https://www.heroku.com.

[12] Hivelocity. What is cron job. https://www.hivelocity.net/kb/what-is-cron-job/.

[13] JawsDB. Database-as-a-service. https://www.jawsdb.com.

[14] Jest. Delightful javascript testing. https://jestjs.io.

[15] MariaDB. Enterprice open source. https://mariadb.com.

[16] Mint. Budget tracker and planner. https://mint.intuit.com.

[17] NestJS. A progressive node.js framework. https://nestjs.com.

[18] Node.js. Run javascript everywhere. https://nodejs.dev.

[19] Node.js. What is npm? https://nodejs.org/en/knowledge/getting-started/npm/what-is-npm/.

[20] Paw. The most advanced api tool. https://paw.cloud.

[21] Redis. Redis. https://redis.io.

[22] Denis Solonenko. Open source personal finance tracker for android. https://github.com/dsolonenko/financisto.

[23] Spendee. Money manager and budget planer. https://www.spendee.com.

[24] Swagger. Openapi specification. https://swagger.io/specification/.

[25] Techopedia. What is database as a service. https://www.techopedia.com/definition/29431/database-as-a-service-dbaas.

[26] TypeScript. Typed javascript at any scale. https://www.typescriptlang.org.

Contents of enclosed CD

```
 readme.txt ........................ Description of the contents of enclosed CD
 documentations .............................................. Documentation
   models.eap ........................................ Enterprise Architect models
   api.yaml ............................................... API documentation
 scripts ......................................................... SQL scripts
   create.sql ..................................................... Create script
 src .............................................................. Source codes
   impl ........................................... Implementation source code
   thesis .......................... Source code of the thesis in the format of LaTeX
 text ................................................ The thesis text directory
   thesis.pdf ............................... Text of the work in the PDF format
```