BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

SPECIALIZATION COMPUTER SCIENCE IN ENGLISH

DIPLOMA THESIS

A Simplified Version of Modern Payment Systems

Supervisor
Camelia Chisăliţă-Creţu
Lecturer, PhD

Author
Vîrgă Tudor-Valentin

2020

UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA

FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ ENGLEZĂ

LUCRARE DE LICENŢĂ

O Versiune Simplificată a Sistemelor de Plăţi Moderne

Conducător ştiinţific
Lector dr. Camelia Chisăliţă-Creţu

Absolvent
Vîrgă Tudor-Valentin

2020

Abstract

The object of this paper is the payment system, including the history of money and payment methods, modern payment systems, existing implementations, and the development of a payment system reduced to its core functionalities. The role of each chapter is delimited and the first chapters start with a history lesson to give a better understanding of the notion of payment system, then we review the theoretical notions and technologies used for developing the application. The final chapter deals with implementation details with the software development process including analysis, testing, and using tools to monitor and improve performance.

The first chapter serves as a short introduction, describing what payment systems are, their purpose, and the reasoning behind the application and its motivation.

The second chapter aims to explain the needs such a system tries to fulfill utilizing a brief history lesson and how current and past implementations managed, or not, to meet those needs, but it also specifies the functional requirements the application should meet.

The third chapter analyzes the application's design goals by enumerating the functional and non-functional requirements, and also describing it’s architecture.

The fourth chapter revisions the technologies used for back-end, front-end, and database. It provides technical background for why this technology stack is a good fit for the application.

The fifth chapter describes the applicable theoretical notions suitable for the application such as concurrent and parallel programming, multithreading, and STP (straight-through processing) and explains why their use is a necessity in such systems.

The sixth chapter is concerned with the development flow of the application, including both functional and non-functional testing. Also, it presents profiling tools and how these tools can help developers improve the performance of such a system. It also deals with the package design, class design, and the use-cases along with the use case diagram.

The seventh chapter has the objective of drawing a conclusion and proposing future upgrades and improvements of the application.

This work is the result of my own activity. I have neither given nor received unauthorized assistance on this work.

Name: **Signature:**
Vîrgă Tudor-Valentin

[illegible]

Contents

List of Figures

Chapter 1 - Introduction

Payment is the exchange of a form of goods, be it services, assets, or funds between two or more parties, which have previously agreed on some acceptable terms [1]. Nowadays, due to the evolution of the monetary system, most payments are made through currency. The appearance of currency simplified payments because it provides a safe and convenient medium for the parties involved and it can easily be stored. For example, in the past, if a dairy farmer needed shoes, he would need to find a shoemaker who would be willing to take milk as payment for shoes. In this case, if a suitable shoemaker weren't found in time, not only would the farmer not get his shoes, but his milk would spoil, becoming worthless. The currency, on the other hand, has no expiration date and it's always exchangeable for other goods.

1.1. Motivation

A payment system is a set of processes and technologies that transfer monetary value from one entity or person to another. The core principle of a payment system is that it uses cash substitutes, such as checks or electronic messages, to create the debits and credits that transfer value [2]. Therefore, a tech approach sounds inevitable due to the heavy processing and accounting such a system requires. Given the current economic landscape, we can easily assume that hundreds of millions of payments have to be processed worldwide each day, a system based on manual accounting and processing would require a great human workforce. Also, the human factor implies that calculus errors might occur and those errors might be affecting business and people who depend on the correctness of the results of those calculations. Another important aspect of payment is processing time.

A hundred years ago, if you wanted to make payment across the country, you had to go to the bank, complete a form and give the money to the cashier, and maybe in one or two weeks, your money would reach their final destination. Moreover, there are security and trust issues. The money or the check you sent might never reach the destination due to meteorological conditions, thieves, or other external factors. Just imagine a world where all payments should be done via this method and also consider that there are international payments to be done, it's just not acceptable given the current technological context. Wouldn't it be easier if all you had to do is write an account number and a sum of money from the comfort of your personal computer, click OK and everything would be solved the same day or next day in the worst case? Well, this is how things happen nowadays.

1.2. Brief Description of the Application

The modern payment systems aim to resolve all of the aforementioned issues. However, due to the complexity of such systems, the application is a simplified version. Nonetheless, it could easily be used by an institution where managing and transferring virtual funds is a necessity.

It comes with an easy to use and intuitive user interface for moving funds between its participants, also providing an option for processing and initiating a large number of payments.

Besides, it deals with security issues like initiating unauthorized payments by implementing a four-eyes check system where two different administrators of the system have to approve the payment. The user interface is web-based so it can be accessed from any type of electronic device which supports web browsers.

There are two types of participants, regular ones and administrators. Besides creating accounts for themselves, editing their user account, and initiating transactions, administrators have the right to perform these actions on other user accounts, but it’s also their responsibility to verify and approve incoming transactions. Another security measure of the system is an audit function that tracks every action performed by users such as editing their profile, creating a new virtual account, entering new payments, or approving a transaction in case of administrators. In the interest of tracking the performance of the system when under heavy processing and accounting load, a text file containing precise figures regarding the performance is generated and a summary of the file is also logged in the console by the system.

1.3. The Layout of the Paper

The paper is organized as follows:

- The first chapter serves as a short introduction, describing what payment systems are, their purpose, and the reasoning behind the application and its motivation.

- The second chapter aims to explain the needs such a system tries to fulfill utilizing a brief history lesson and how current and past implementations managed, or not, to meet those needs. It also specifies the functional requirements the application should meet and comes with a less technical analysis of the application.

- The third chapter analyzes the application's design goals by enumerating the functional and non-functional requirements, and also describing it’s architecture.

- The fourth chapter revisions the technologies used for back-end, front-end, and database. It provides technical background for why this technology stack is a good fit for the application.

- The fifth chapter describes the applicable theoretical notions suitable for the application such as concurrent and parallel programming, multithreading, and STP (straight-through processing) and explains why their use is a necessity in such systems.

- The sixth chapter is concerned with the development flow of the application, including both functional and non-functional testing. Also, it presents profiling tools and how these tools can help developers improve the performance of such a system. It also deals with the package design, class design, and the use-cases along with the use case diagram.

- The seventh chapter has the objective of drawing a conclusion and proposing future upgrades and improvements of the application.

Chapter 2 – Payment Systems

2.1. A World without Money

Money, in some form, has been part of human history for at least the last 3,000 years. Before that time, it is assumed that a system of bartering was likely used. Bartering is a direct trade of goods and services, but such arrangements take time because not everyone is willing to trade their goods or services for someone else's. One of the great achievements of money was increasing the speed at which business could be done. Slowly, a type of prehistoric currency involving easily traded goals like animal skins, salt, and weapons developed over the centuries [3]. The system of barter and trade spread across the world, and it still survives today in some parts of the globe.

Sometime around 770 B.C, the Chinese moved from using actual tools and weapons as a medium of exchange to using miniature replicas of the same tools cast in bronze. Nobody wants to reach into their pocket and impale their hand on a sharp arrow so, over time, these tiny daggers, spades, and hoes were abandoned for the less prickly shape of a circle, which became some of the first coins [4]. Although China was the first country to use recognizable coins, the first minted coins were created not too far away in Lydia (now western Turkey).

In 600 B.C., Lydia's King minted the first official currency, it was made from a mixture of silver and gold that occurs naturally, and stamped with pictures that acted as denominations. Lydia's currency helped the country increase both it's internal and external trade, making it one of the richest empires in Asia Minor. Just when it looked like Lidya was taking the lead in currency developments, around 700 B.C., the Chinese moved from coins to paper money. By the time Marco Polo in 1271 A.D., the emperor had a good handle on both money supply and various denominations. In the place where the American bills say, "In God We Trust", the Chinese inscription warned, "Those who are counterfeiting will be decapitated" [5]. Europeans were still using coins up to the 16th century, helped along by acquisitions of precious metals from colonies to keep minting more and more cash. Eventually, the banks started using bank notes for depositors and borrowers to carry around instead of coins. Those notes could be taken to the bank at any time and exchanged for their face values in silver or gold coins. This paper money could be used to buy goods and operated much like currency today, but it was issued by banks and private institutions, not the government, which is now responsible for issuing currency in most countries [6]. The first paper currency issued by European governments was issued by colonial governments in North America. Because shipments between Europe and the colonies took so long, the colonists often ran out of cash as operations expanded.

The shift to paper money in Europe increased the amount of international trade that could happen. Banks and the ruling classes started buying currencies from other nations and created the first currency market. The stability of a particular monarchy or government affected the value of the country's currency and the ability for that country to trade on an increasingly international market. The competition between countries often led to currency wars, where competing countries would try to affect the value of the competitor's currency by driving it up and making the enemy's goods too expensive, by driving it down and reducing the enemy's buying power, ability to pay for a war, or by eliminating the currency [3].

The 21st century gave rise to two disruptive forms of currency: mobile payments and virtual currency. Mobile payments are money rendered for a product or service through portable electronic devices such as a cell phone, smartphone, or tablet. Mobile payment technology can also be used to send money to friends or family members. Increasingly, services like Apple Pay and Samsung Pay are vying for retailers to accept their platforms for point-of-sale payments. Bitcoin, released in 2009 by the pseudonymous Satoshi Nakamoto, became the gold standard, so to speak, for virtual currencies. Virtual currencies have no physical coinage [7]. The appeal of virtual currency is it offers the promise of lower transaction fees than traditional online payment mechanisms and is operated by a decentralized authority, unlike government-issued currencies.

2.2. Payment Channels

Payment initiators and their processers can use different channels to make a payment and each has different operating characteristics, rules, and settlement mechanisms [2]. Generally speaking, all payment systems fall into one of the following five payment channels:

- Paper-based systems such as checks or drafts. Payments are initiated when one party writes an instruction on paper to pay another. These systems are one of the oldest forms of no cash payment system [2]. Checks are a common paper-based channel and while in decline are still widely used in the United States and a few other countries.

- RTGS (Real Time Gross Settlement) and other high-value payments; called wire transfers by most people. Wires came into being in the late 1800s with the invention of the telegraph but did not become widely used until the early 1900s.

- Low-Value Batch Systems and Automated Clearing House (ACH) batch payments were introduced in the early 1970s and were designed to replace checks with electronic payments. Unlike wires, which are processed individually, ACH payments are processed in batches and were originally intended for small payments under $100,000 such as payroll and consumer transactions [2].

- Cards are a payment channel that includes credit, debit, and stored-value cards. They are a heavily used and fast-growing segment of the methods for making and receiving payments. The card channel also often provides the rails or settlement systems supporting some of the newer e-commerce payment systems such as mobile wallets.

Figure 1. Visa and MasterCard Logos (youtube.com)

- The explosion of mobile applications, cyber currencies, and other payment alternatives has created several payment methods that do not nearly fit in the preceding four channels.

While the rules of settlement and value transfer are not suspended for this channel they are considered distinct.

Figure 2. Bitcoin representative image (entrepreneur.com)

2.3. Existing Implementations

There are many types of payment systems that serve different purposes, some of them are used for intra-banking payments, others are used for inter-banking payments and there are also a few that deal with international payments. These payment systems usually interact with each other via messages which are regulated by SWIFT. Since there is such a multitude of payments systems, this paper will only present the most important ones:

- Real-time gross settlement (RTGS) is the continuous process of setting payments on an individual order basis without netting debits with credits across the books of a central bank (e.g., bundling transactions). Once completed, real-time gross settlement payments are final and irrevocable. It is generally employed for large-value interbank funds transfers. These types of systems are increasingly used by central banks worldwide and can help minimize the risk of high-value payment settlements among financial institutions [8]. RTGS can often incur a higher charge than processes that bundle and net payments.

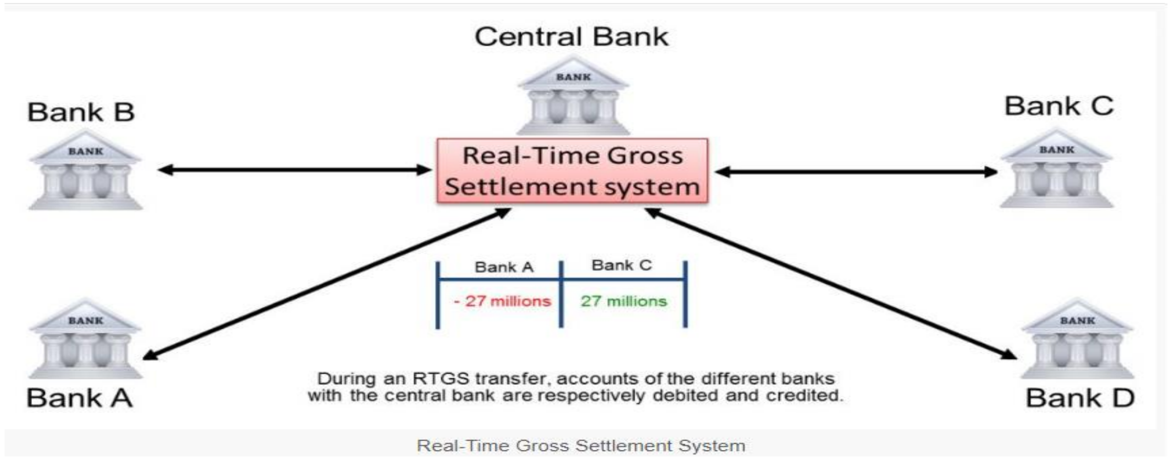

Central Bank
BANK
Bank B
BANK
Real-Time Gross Settlement system
Bank C
BANK
Bank A
Bank C
- 27 millions
27 millions
BANK
Bank A
During an RTGS transfer, accounts of the different banks with the central bank are respectively debited and credited.
BANK
Bank D
Real-Time Gross Settlement System

Figure 3. RTGS Fund Transfer [8]

- Central securities depository (CSD) is a specialist financial organization holding securities such as shares either in the certified or uncertificated form so that ownership can be easily transferred through a book entry rather than the transfer of physical certificates. This allows brokers and financial companies to hold their securities at one location where they can be available for clearing and settlement. This is usually done electronically, making it much faster and easier than was traditionally the case where physical certificates had to be exchanged after trade had been completed [9]. In some cases, these organizations also carry out the centralized comparison, and transaction processing such as clearing and settlement of securities transfers, securities pledges, and securities freezes. Many countries have one domestic CSD that was traditionally associated with the national stock exchange. An international CSD settles trades in international securities such as Eurobonds although many also settle trades in various domestics securities, usually through direct or indirect links to local CSDs.

- The Automated Clearing House (ACH) is an electronic funds-transfer system that is used for payroll, direct deposit, tax refunds, consumer bills, tax payments, and many more payments services in a country. This network essentially acts as a financial hub and helps people and organizations move money from one bank account to another. ACH transactions consist of direct deposits and direct payments, including B2B transactions, government transactions, and consumer transactions [10].

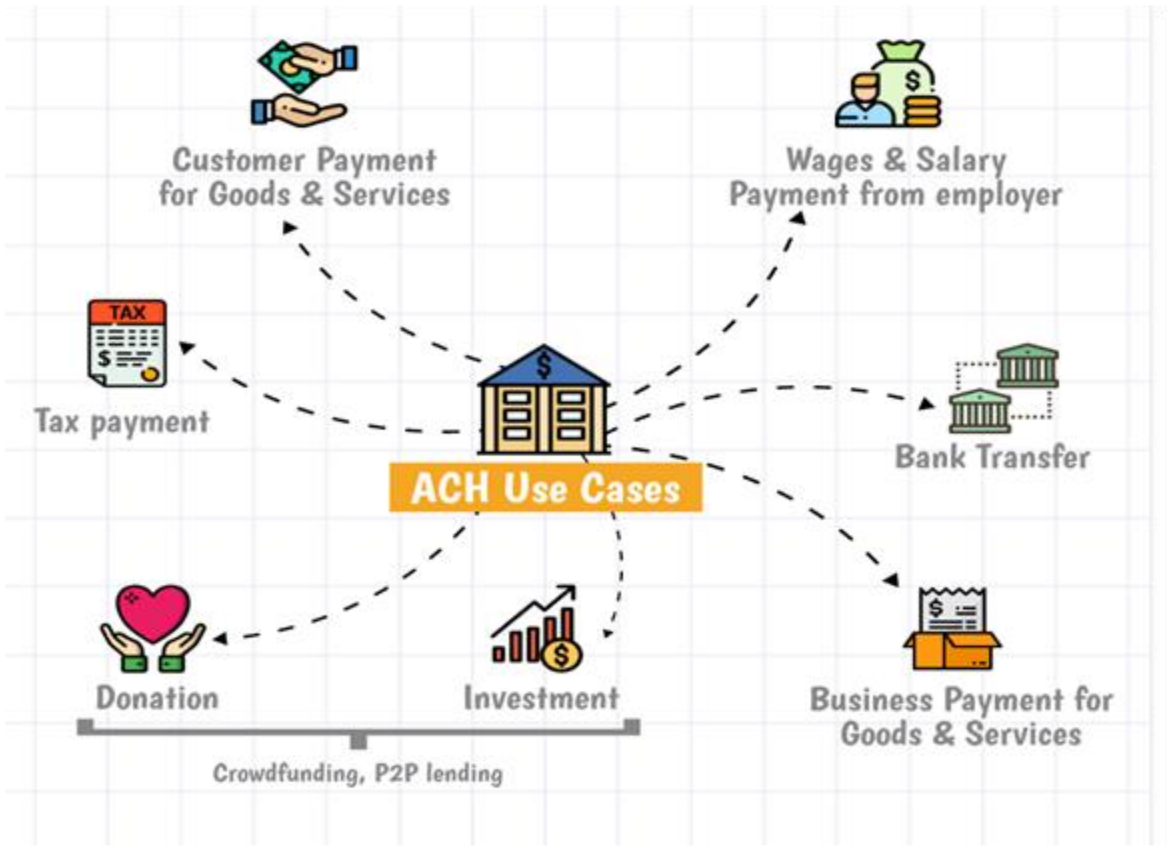

Customer Payment for Goods & Services
Wages & Salary Payment from employer
TAX
Tax payment
ACH Use Cases
Bank Transfer
Donation
Investment
Business Payment for Goods & Services
Crowdfunding, P2P lending

Figure 4. ACH Use Cases [10]

There are several companies on the market specializing in the payment systems field among which Montran (which also has a branch in Cluj, Romania), STT-Software, SIA, CMA, and Tosan. Usually, these companies only provide payment systems solutions due to the domain’s complexity and variety. The systems aforementioned are usually administrated by the central bank of a country and in some cases, these systems are built in-house by central banks.

Chapter 3 – Application Requirements and Design

The application tech stack was chosen considering the requirements such a system should meet. For the back-end, a language that provided multithreaded and concurrent programming along with a framework that made web development easier was needed so Java and its framework, Spring Boot, was a good choice. As for the database, a relational one was required due to the multiple links between users, accounts, and payments so I chose PostgreSQL. To make the communication between the server and the database, the use of an Object Relation Mapping (ORM) like Hibernate along with the support of Spring's Java Persistence API (JPA) Data is an efficient combo. For the front-end part of the application, I chose a server-side template engine for Java, called Thymeleaf, that has modules for the Spring Framework [11].

3.1. Functional Requirements

A functional requirement is the description of a software application or the description of one of its components. The functional requirements for this application are the following:

- For both **administrators** and **regular users**:
    - The user should have the option to log in using its credentials. There should be two types of users: normal users with limited authority and administrators.
    - Both regular users and administrators should have the ability to edit their profile.
    - Both regular users and administrators should be able to create accounts, regular users for themselves and administrators for them, and regular users.
    - Both regular users and administrators should have the ability to initiate transactions, the administrators can only initiate transactions for regular users.
    - Regular users should be able to search for a transaction, by one or more filters.
- For **administrators:**
    - Administrators should have separate screens for both steps of the four eyes check process. An administrator should only be able to approve one step of the process, but not both. There also should be a screen for the transactions which were rejected.
    - Administrators should have a screen for creating new users.
    - Administrators can search for users by their username. The same applies to accounts.
    - The application should provide a screen with the timeline of the transactions associated with a certain account, and how those transactions modified the account

balance. The user should be able to search for the transactions which were happened in a time interval by selecting the starting and the ending date.

- The application should provide a screen where administrators can see all transactions, ordered by date.

- Administrators can also edit regular user profiles.

3.2. Non-Functional Requirements

Non-Functional Requirements specify the quality attribute of a software system. They judge the software system based on responsiveness, usability, security, portability, and other non-functional standards that are critical to the success of the software system [12]. An example of non-functional requirements is how fast does a website loads. Failing to meet non-functional requirements can result in systems that fail to satisfy user needs. These requirements allow the developers to impose constraints or restrictions on the design of the system. The non-functional requirements for this application are the following:

- Concerning **security** and **authentication**:

    - The system should display an error message every time a user fails to log in.

    - The system should display an error message every time a regular user tries to access an unauthorized path.

    - The system should display an error message every time a user with already existing credentials is created.

    - The system should alert the user that creates or updates an entity if there is a missing or incorrect field.
    - The system should point to the user that an entity that they were searching for like a user, an account, or a transaction does not exist in the database.

    - The system should display an error page if an invalid URI is accessed.

    - The system should oversee the four eyes check process so that an administrator is not allowed to approve a transaction for both steps of the process.

    - The system should always give feedback to the user which performed an update, stating the result of the update, whether it was successful or not.

    - The application should provide an audit with every action performed on an entity such as an account or a user profile.

- Concerning the **batch transaction** feature:

    - The application should provide functionality for processing a batch of transactions. These transactions will come in a text format and will benefit from straight-through processing (STP), this means that no administrator will have to approve these transactions.

    - The system should process the payments correctly, especially the ones that come in batches, and there should be no inconsistency in the account's balances and that thing should be transparent when verifying the history of payments for an account.

    - The application should provide a text file as an output when the batch processing happens. It should also log the number of transactions, the speed at which they were processed, and the time elapsed every 10 seconds.

3.3. Architecture

This subchapter will deal with the architecture of the entire application and the data flow, low system design, and database design.

The high-level design is simplistic and the data flow is bidirectional. The input of the user is validated and sent to the back-end, filtered by the business rules, and stored in the database as shown in the below diagram. If everything was successful, the response passes through the business flow and gets displayed to the end-user.

The back-end of the application is multi-layered, more specifically, a classic (Model-View-Controller (MVC). The view consists of the Thymeleaf pages which are server-side rendered. The controller is concerned with the routing of the incoming requests and it also checks for invalid inputs. There are two middle-ware components, a service that is concerned with the business rules of the application. The business rules represent the logic of the application, what the application needs to do based on the input that comes from the controller. The other middle-ware component is the repository which deals with the database and retrieving information based on queries coming from the service. The model layer represents the multiple entities such as an account, a user, a payment that is obvious to the end-user, less obvious entities such as the balance of the account, or an audit entry for every action of the user. Besides, there are also entities concerned with the internals of the application such as a role for every user or a base entity that every other entity is based on.

The database design is quite complex since there are lots of connections between transactions, accounts, and users. For example, a transaction needs a link to the user that initiated, a link to the creditor's account, one to the debtor's account, and another two for the four eyes check process. There is no point in enumerating every foreign key, but the main tables are the ones regarding transactions, accounts, and users.

Chapter 4 – Technology Stack Analysis

This chapter will briefly analyze each component used in the application development, the database, the back-end, and the front-end part.

4.1. Java and the Spring Framework

Figure 5. Java Logo

Java is a programming language and computing platform first released by Sun Microsystems in 1995 [13], where James Gosling led a team of researchers to create a new language that would allow consumer electronic devices to communicate with each other. Work on the language began in 1991, and before long the team's focus changed to a new niche, the World Wide Web. Java's ability to provide interactivity and multimedia showed that it was particularly well suited for the web [14]. The difference between the way Java and other programming languages worked was revolutionary for those times.

Code in other languages was first translated by a compiler into instructions for a specific type of computer. The Java compiler instead turns code into something called bytecode, which is then interpreted by a software called Java Runtime Environment (JRE). This acts as a virtual computer that interprets bytecode and translates it for the host computer. Because of this, Java code can be written the same way for any platform, which helped lead to its popularity for use on the Internet, where many different types of computers may retrieve the same web page. Java's motto is "write once, run anywhere". There are many places where Java is used in the real world, starting from e-commerce websites to android apps, from scientific application to financial applications like electronic trading systems, form games like Minecraft to desktop applications like Eclipse and IntelliJ.

Figure 6. Spring Logo

The Spring Framework provides a comprehensive programming and configuration model for modern Java-based enterprise applications, on any kind of deployment platform [15]. Spring is flexible and has a comprehensive set of extensions and third-party libraries that let developers develop almost any application. At its core, Spring Framework uses Inversion of Control (IoC) and Dependency Injection (DI) to provide the foundation for a wide-ranging set of features and functionalities. Spring also has contributions from all the big names in tech, including Alibaba, Amazon, Google, Microsoft, and more. It is a framework that helps developers wire different components together. It is most useful in cases where there are a lot of components and they need to be combined in different ways depending on different settings or environments. Due to dependency injection, which is a pattern that allows building very decoupled systems, unit testing becomes very easy since all classes have setters for the important dependencies and these can be easily mocked to provide the required behavior.

The main Spring module needed for developing the current situation is Spring MVC because it makes web development easier and provides a multi-layered base for the application. Another Spring modules that make the development of the current application are Spring Boot, which it makes it easier to create stand-alone applications without having to stress about deploying, Spring Data, which helps the developer map the relations between database tables and back-end entities and Spring Security which is a framework that enables the developer to highly customize the authentication and the authorization processes. Besides, Spring also provides modules for creating cloud applications, mobile development, web services, creating a command-line interface (CLI) applications, and many more.

4.2. MVC and Spring MVC

MVC is a software architecture that separates the application logic from the rest of the user interface. It does this by separating the application into three parts: the model, the view, and the controller. The model manages fundamental behaviors and data of the application. It can respond to requests for information, respond to instructions to change the state of its information, and even notify observers in event-driven systems when information changes. This could be a database or any number of data structures or storage systems. In short, it is the data and data-management of the application. The view effectively provides the user interface element of the application. It will render data from the model into a form that is suitable for the user interface. The controller receives user input and makes calls to model objects and the view to perform appropriate actions. All in all, these three components work together to create the separation of concerns and are the three basic components of the MVC architecture.

Spring MVC, like many other web frameworks, is designed around the front controller pattern where a central Servlet, the DispatcherServlet, provides a shared algorithm for request processing, while actual work is performed by configurable delegate components [16]. This model is flexible and supports diverse workflows. Among the Spring's web MVC module many unique web support features here are the most important: clear separation of roles, customizable and adaptable binding and validation, and also there is no need for duplication for the reusable business code.

As for the application, Spring MVC was the best choice because it provides the separation of concerns which is vital in such a complex context which also has to provide loose couples between the application's components. Beyond the current MVC components, the application will have two more layers, a repository concerned with the database information, and a service concerned with the business logic and flows of the application. The principal models of the application will be the user, the account, and the transactions between those accounts and users. Almost every model will also have its repository concerned with getting and inserting data into the database. The next layer is the service, almost one for every model, which will implement the flows for every request of the user of the application based on the model in the request. The controller will receive requests from the application's users and will redirect them to the specific services that need to handle those requests. Finally, the view is composed of the HTML pages together with the Thymeleaf specific templates which are server-rendered.

4.3. Security and Spring Security

Spring Security is a framework that provides authentication, authorization, and protection against common attacks. With first-class support for both imperative and reactive applications, it is the de-facto standard for securing Spring-based applications [17]. It also integrates well with frameworks like Spring Web MVC, as well as with standards like OAuth2, and it auto-generates login/logout pages and protects against common exploits like Cross-site request forgery (CSRF).

As for the application, it was used to provide the 2 types of users: regular users and administrators. It also takes care of the types of pages regular users have access to because the front-end might be similar for both administrators and normal users, but more information might be displayed for one of them, so there is no need for code duplication. This prevention of code duplication is also possible due to Thymeleaf's templates which display, or not, a part of the web page depending on the role of the user it is accessed by. It also provides a template for the login page of the application and takes care of the user session when switching between different web pages. Everything happens inside one configuration class and the code needed is minimal and these configurations are highly customizable.

It is also worth mentioning that bcrypt password encoder was used for encrypting the user passwords and storing them in the database. The bcrypt hashing function allows us to build a password security platform that scales with computation power and always hashes every password with a salt [18]. Bcrypt was designed based on the Blowfish cipher which is a symmetric-key block cipher, this is where the b comes from and the crypt is the name of the hashing function used by the UNIX password system. The configuration of the bcrypt password encoder is also done in the same class as the Spring Security one and is easy to implement.

4.4. Spring Data and Hibernate

Spring Data's role is to provide a familiar and consistent, Spring-based programming model for data access while still retaining the special traits of the underlying data store [19]. It makes it easy to use data access technologies, relational and non-relational databases, map-reduce frameworks, and cloud-based data services.

Figure 7. A hibernating bear [19]

Hibernate is an ORM solution for Java environments. ORM is the programming technique to map application domain model objects to the relational database tables. One more concept is also needed for understanding how Spring Data and Hibernate work together. JPA provides a specification for persisting, reading, and managing data from your Java object to relational tables in the database [20]. Spring Data JPA is not a JPA provider, but rather a library/framework that adds an extra layer of abstraction on the top of our JPA provider (like Hibernate in this situation). For the image to be complete, Spring Boot uses an opinionated algorithm to scan for and configure a data source, a PostgreSQL relational database in our case. This allows us to easily fully-configured data source implementation by default.

For all of the aforementioned to be properly used, a rather simple configuration is needed. In this file, for the database connection, the database driver, URL, username, and password have to be specified. Also, the database dialect has to be mentioned, because Spring JPA uses different dialects for different relational databases like MySQL or PostgreSQL. Hibernate configurations are written in this file as well. Among the Hibernate configurations which can affect the performance of the application, there are features for logging, batch inserts or updates, the quantity of the batch operations, and new ID generators.

Spring Data *JpaRepository* and *CrudRepository* are two interfaces that provide a template for out of the box methods in a relational database such as insert, update, delete, and search contact. The classes representing the repository layer are annotated with the *@Repository* annotation and implement the *JpaRepository* or the *CrudRepository* interfaces. There are well-predefined naming conventions, such that the expected query is created from the name of the method and its parameters. For example, *findByStatusNotLike(String status)* returns all accounts that have a different status from the one given as the status argument. For more complex or difficult to express queries, there is also a *@Query* annotation where the SQL query can be written explicitly and use the parameters of the annotated method. Additionally, the methods can return page objects where the page contains as many objects as needed for display on one page, so not all records are requested from the database, overloading the RAM. There is also an option for retrieving only some fields of an object from the database so that only the needed fields for displaying or application logic, this also helping with the reduction of RAM usage.

In conclusion, Spring Data JPA comes with a lot of useful tools that can help developers write less boilerplate code and also addresses complex problems met when working with such tools. However, using it might reduce the application's performance in some specific cases and could also confuse developers because of the layer of abstraction it adds to the application.

4.5. Relational Databases and PostgreSQL

A relational database is a data collection of items and relationships between them. These items are organized as a set of tables with columns and rows [21]. Information about objects represented in the database is held in tables. Each column in a table holds a certain kind of data and a filed stores the actual value of an attribute. The table rows represent a collection of the related values of a single object or entity. Each row in a table could be marked with a unique identifier called the primary key, and rows among multiple tables can be made related using foreign keys. This data can be accessed in many different ways without reorganizing the database tables themselves. A unique code called the primary key may be placed on each row in a table, and rows between several tables can be connected using foreign keys. Without rearranging the database tables themselves, these data can be used in various ways.

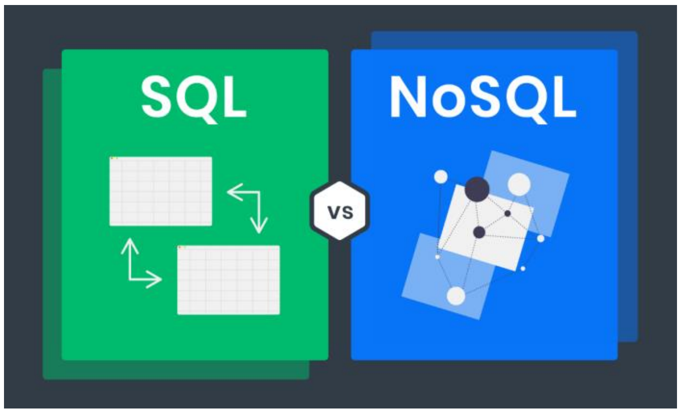


Figure 8. SQL vs NoSQL

Structured Query Language (SQL) is the main method used to interact with Relational Databases. SQL is used to create, update, or erase rows of data, access data subsets for transaction processing and analytics application, and handle other facets of the database. A database transaction is made of one or more SQL statements that are performed as a series of operations that constitute a

single logical unit of work [21]. Transactions include an “all-or-nothing” option, which implies either the whole transaction will be done as a single unit and written to the database, otherwise, none of the transaction’s different components should execute. The transaction ends in a COMMIT or ROLLBACK, according to database terms. Data integrity is the completeness, precision, and consistency of data.
NoSQL databases also have to be taken into account when building an application so a comparison between SQL and NoSQL is worth the time. In opposition to relational databases, NoSQL ones have unstructured data manged by dynamic schemas, and the data can also be stored in multiple ways. They can be column-oriented, document-oriented, graph-based, or organized as a key-value store [23]. This flexibility translates to the capability of creating documents without a well pre-defined structure and also each document can have a unique structure. The syntax used to operate on the database doesn’t have to be transmissible from one database to another and also fields can be added to the database as you go. Usually, SQL databases scale well vertically, that meaning that the load on a single server can be increased by adding SSD, RAM, CPU, or cache-memory. On the other hand, NoSQL databases scale better horizontally, that meaning that multiple servers are added. Usually, a payment system like the one the application emulates is hosted by a single capable machine, rather than more distributed servers. Also, because of the multiple links between accounts and transactions, there have to be strict relations in place between different tables of the database, so that’s why the data of this application is stored on a SQL database. **Table** consists of the main differences between SQL and NoSQL databases.

Criterion	**SQL**	**NoSQL**
Definition	Called primarily RDBMS or relational databases	Called primarily called non-relational or distributed databases
Designed for	Used for online analytical processing systems (OLAP), uses SQL syntax and queries to analyze and get data for further insights	Developed in response to the demands presented for the modern application development and consist of various kind of database technologies
Query Language	Structured Query Language (SQL)	No declarative query language
Type	Table based databases	Can be document based, key-value pairs, graph databases
Scalability	Vertically scalable	Horizontally scalable
Importance	Should be used when data validity is extremely important	Should be used when it’s more important to have fast data rather than correct data
Best option	Support needed for dynamic queries	Scale based on changing requirements
ACID vs BASE	Atomicity, Consistency, Isolation, and Durability (ACID) is a standard for RDBMS	Basically Available, Soft State, Eventually Consistent (BASE) is a model for many NoSQL systems
Top Companies Using	Hootsuite, CircleCI, Gauges	Airbnb, Uber, Kickstarter

Table 1. Comparison between SQL and NoSQL databases

Relational databases use a collection of restrictions to maintain database integrity. Among these constraints, there are primary and foreign keys, the not-null constraint, the unique and the default constraints, but also the check constraints. Such integrity restrictions help is helping to enforce business rules on data in the tables to maintain the quality and durability of the data. Besides these, most relation databases often enable the embedding of custom mode into triggers that execute depending on an action on the database. ACID is a standard that all database transactions must comply with. Atomicity involves either successful execution of a transaction as a whole, otherwise, if a portion of the transaction fails, invalidation of the entire transaction is required. Consistency requires that the data written into the database as part of the transaction must comply with all specified rules restrictions including constraints, cascades, and triggers. To achieve concurrency control and ensure that every single transaction is only independent of itself. Durability means that all modifications made to the database are irreversible from the moment when a transaction is completed successfully.

PostgreSQL is a powerful, open-source object-relational database system that uses and extends the SQL language combined with many features that safely store and scale the most complicated data workloads [22]. PostgreSQL's roots date back to 1986 as part of the POSTGRES project at Berkeley University Of California and its core platform is under active development for more than 30 years. Due to its proven architecture design, durability, data integrity, diverse feature set, extensibility, and commitment to the open-source community to continually deliver creative and effective solutions PostgreSQL has gained a strong reputation. PostgreSQL works on all major operating systems has been complying with the ACID standards since 2001, and has versatile add-ons, such as the popular PostGIS geospatial database extensor. Because of all of the reasons aforementioned, PostgreSQL has become the first choice for many individuals and organizations in terms of open-source relational databases. Besides being open source and free, PostgreSQL is also highly customizable and extensible. Some of the other advantages of using PostgreSQL presented on their official website are support for all needed datatypes and the possibility of creating new customizable types, support of data integrity, concurrency and performance through powerful types of basic/advanced indexing methods and parallelization, support for international character sets and full-text search and also spatial database extender. Another interesting perk is its ability to be transformed into a NoSQL database by using documents like JSON and XML. The content of these data types can be indexed, providing a lot of speed and data integrity so the user can get the best of both worlds, using an easy SQL syntax to query non-relational, but indexed data with the best speed [22].

As for the application, the web interface was very useful in the process of development because it is lightweight and it provides lots of functionality while being easy to use and intuitive. It also comes with a diagram functionality that builds the diagram of the database including all the foreign and private keys and how the tables are linked through them. As far as the database tuning goes, indexes and ID generators are also easy to create and there is a high degree of customizability. In conclusion, PostgreSQL was easy to integrate with the application and came with a lot of advantages that made development easier.

4.6. Thymeleaf

Thymeleaf is a modern server-side Java template engine for both web and standalone environments. The main aim of Thymeleaf is to introduce elegant natural templates into the development workflow, especially HTML that can be displayed properly in browsers and also act as static prototypes, resulting in better cooperation in development teams. Thymeleaf is perfect for modern HTML5 Java web development because of its modules for Spring Framework, host integration with many tools, and the ability to plug in custom functionality, although it can do much more than that [24].

Figure 9. Thymeleaf Logo

In light of the current boost in popularity of JavaScript frameworks such as React, Angular, and Vue it's worth drawing a comparison between server-side rendering and client-side rendering. Server-side rendering (SSR) is the conventional rendering approach, where the server essentially hosts all of the web page resources. When the page is requested, the HTML is sent to the browser and rendered, JS and CSS will also be downloaded, and the complete render will be provided to the user/bot [25]. Client-side rendering (CSR) is a more recent form of rendering technique, which relies on the browser to execute JavaScript through a JavaScript framework like the aforementioned ones. Practically, the client can first request the source code that has very little indexable HTML in it, then make a second request for the .js file that includes all the HTML in JavaScript as strings. Upon the initial request, server-side rendering can sometimes be a bit quicker, just because it does not imply as many round trips to the server. It does not stop here though, performance is also contingent on some additional specific factors. This can all contribute to vastly varying user experiences: the internet speed of the user who sent the request, how many concurrent users there are at a given time, the geographical localization of the server, the degree of optimization for speed page, and others. On the other hand, the client-side rendering's initial request is slower compared to the other because of the multiple back and forth trips to the server. Nevertheless, after these requests are done with, the client-side rendering will be delivering a lightning-quick experience through the JS framework.

Because the user interface is just a tool for initiating transactions and displaying data about those transactions and accounts a server-side framework was chosen. In the companies and institutions where these systems are used, there is usually a stable, reliable, and fast internet infrastructure which allows for a quick response and less waiting time. The application is also based on a monolithic architecture, rather than a microservice oriented one, so the front-end would better fit in this picture, rather than being a separate client application. However, there are both drawbacks and advantages in using either of the options, but even though important and the need for it being straight forward and easy to use, the user interface is not the most important in such systems.

Figure 10. Server-Side Rendering vs. Client-Side Rendering [25]

Thymeleaf templates are easy to use and integrate into basic HTML syntax. It also provides templates for loops, if statements, and iterating through data structures such as arrays and maps.

The data structures used on the back-end have to coincide with those used in Thymeleaf and the objects inside the data structures can be then easily iterated. The framework’s documentation is also covering every aspect of using it and it is the main source of information, and there is rarely a need to search in other sources other than the official documentation. It also has plug-ins for IntelliJ and Eclipse IDEs that can help with auto-completion and syntax correctness.

Chapter 5 – Concurrency, Parallelism, and Performance

This chapter will analyze the concepts of concurrency and parallelism in computer science and how they can help every component of the application increase performance.

5.1. Concepts Overview

Before jumping into the techniques that helped the application achieve a better performance, let's review the concepts of concurrency, parallelism, processes, threads, synchronism, and asynchronism.

Both concurrency and parallelism are referred to as computer architectures which focus on how tasks or computations are performed. Concurrency simply describes executing multiple tasks at the same time, but not necessarily simultaneously. In a concurrent application, two tasks can start, run, and complete in overlapping periods, a task can start even before another one gets completed. Consider you are given the task of singing and eating at the same time. At a given instance of time either you would sing or you would eat as in both cases your mouth is involved. So to do this, you would eat for some time and then sing and repeat this until your food is finished or song is over [26] as shown in **Figure 11**.

Figure 11. You can sing or eat at a time not simultaneously [26]

The way concurrency is achieved across various processors is different in the computer science world. In a single-core environment, concurrency is achieved via a process called context-switching. In a multi-core environment, however, concurrency can be achieved through parallelism.

Parallelism means performing two or more tasks simultaneously. Parallel computing in computer science refers to the process of performing multiple calculations simultaneously [26]. For example, you can cook and speak to one of your friends over the phone at the same time as shown in **Figure 12**.

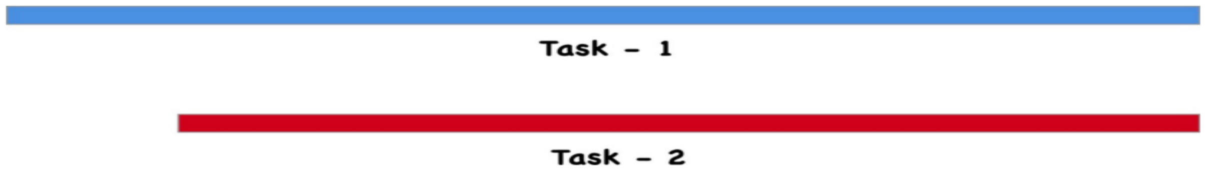


Figure 12. Two tasks being performed simultaneously over the same period [26]

Threads are a sequence of execution of code that can be executed independently of one another. It is the smallest unit of tasks that can be executed by an OS. A program can be single-threaded or multi-threaded. A process is an instance of a running program. A program can have multiple processes. A process usually starts with a single thread, but later down the line of execution, it can create multiple threads. The difference between threads and processes is shown in **Figure 13**.

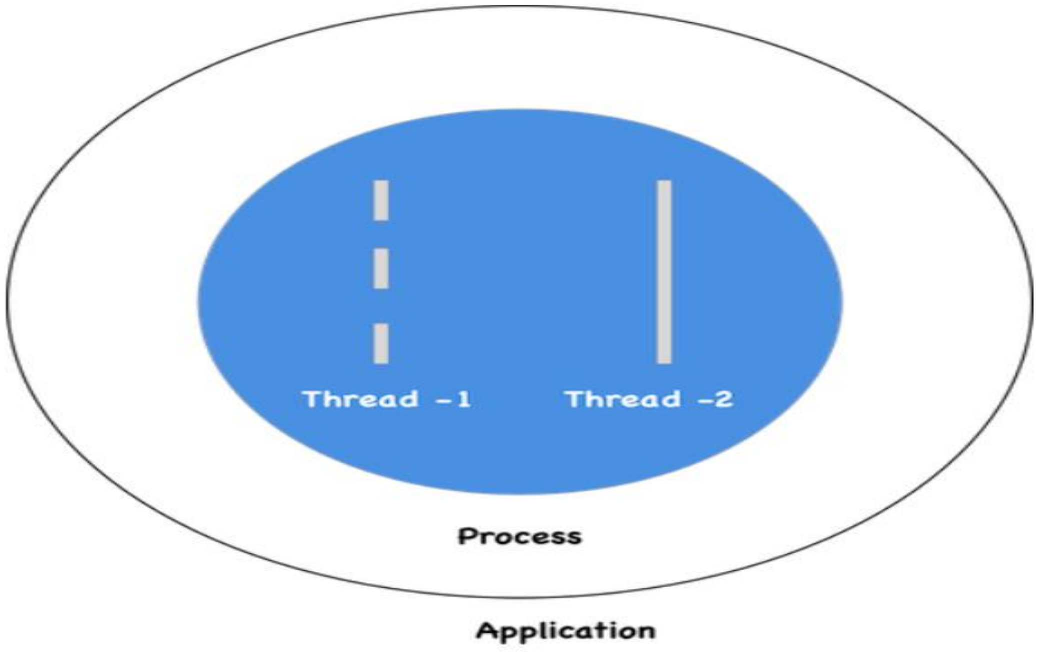


Figure 13. Distribution of Processes and Threads in an application [26]

In a synchronous programming model, tasks are executed one after another. Each task waits for the previous task to complete and then gets executed. In an asynchronous programming model, however, when one task gets executed, tasks can be switched without waiting for the previous to get completed.

The best scenario performance-wise is to use an asynchronous model in a multi-threaded environment. Multi-threaded is often better, but if a synchronous programming model is used, a thread has to wait for the previous one to finish so it makes no difference if one or multiple threads are used. In an asynchronous programming model, however, tasks get executed in different threads without waiting for any tasks and independently finish off their executions as shown in **Figure 14**.

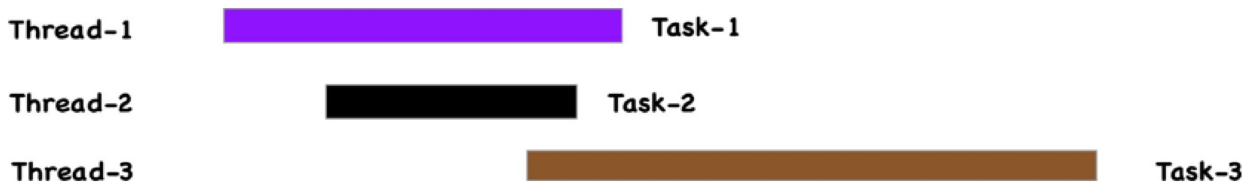


Figure 14. Asynchronous model in a multi-threaded environment [26]

5.2. Multithreading

Multithreading represents having multiple threads of execution inside the same program. A thread is similar to a separate CPU executing the programming. Therefore, a multithreaded program can be seen as an application that has multiple CPUs executing different parts of code at the same time. That being said, a thread is not the same as a CPU. Usually, a single CPU will share its execution time among multiple threads, switching between executing each of the threads for a given amount of time [27]. It is also possible to have threads of an application be executed by different CPUs as shown in **Figure 15**.

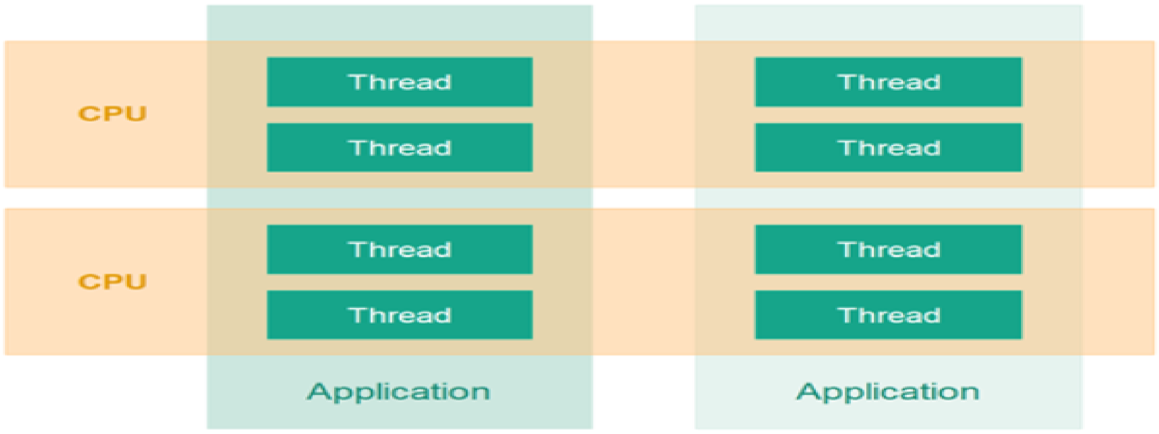


Figure 15. Threads being executed by different CPUs [27]

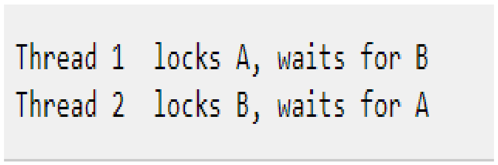
There are several reasons as to why one would use multithreading in an application. Some of the most common reasons for multithreading are a better utilization of a single CPU, multiple CPUs or CPU cores, and better user experience with regards to responsiveness and fairness.

Back in the old days a computer had a single CPU and was only capable of executing a single program at a time. However, many mainframe systems have been able to execute multiple programs at a time for many more years than personal computers. Later came multitasking which meant that computers could execute multiple programs or processes at the same time. However, not everything happened at the same time, because the single CPU was shared between the processes. The operating system would perform context switches as discussed in the concurrency part of this chapter. Along with multitasking came a new challenge for software developers because programs could no longer assume to have all the CPU time available, nor all memory or any other computer resources, so a program had to release all resources it was no longer using. Later yet came multithreading, which is similar to having multiple CPUs executing within the same process.

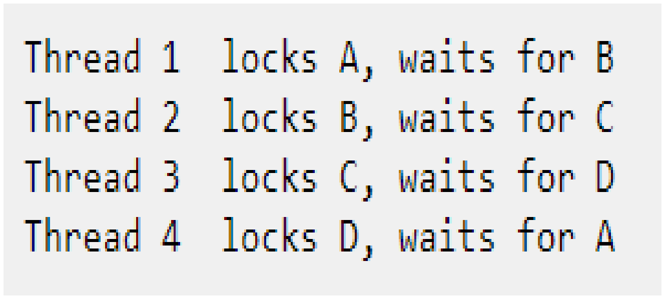
However, multithreading is not easy and is even more challenging than multitasking. The threads are executing within the same process and are hence reading and writing the same memory simultaneously. This can result in errors that are not usual in a single-threaded program.

Java was one of the first languages to make multithreading easily available to developers. The first Java concurrency model assumed that multiple threads executing within the same application would also share objects. This type of concurrency model is typically referred to as a shared state concurrency model [27]. However, this model causes a lot of concurrency problems which can be hard to solve elegantly. Therefore, an alternative concurrency model referred to as “shared nothing” or “separate state” has gained popularity. In the separate state concurrency model, the threads do not share any objects or data. This avoids a lot of the concurrent access problems of the shared state concurrency model. New functional programming parallelism has been introduced with the Fork and Join framework in Java 7, and the collection streams API in Java 8.

However, there are still issues that developers have to deal with constantly and the application also raised one of these well-known problems, the deadlock. A deadlock is when two or more threads are blocked waiting to obtain locks that some of the other threads in the deadlock are holding. It occurs when multiple threads need the same locks, at the same time, but obtain them in a different order. For instance, if thread 1 locks A, and tries to lock B, and thread 2 has already locked B, and tries to lock A, a deadlock arises. Thread 1 can never get B, and thread 2 can never get A. Besides, neither of them will ever know. This situation is a deadlock as shown in **Figure 16**.


```
Thread 1  locks A, waits for B
Thread 2  locks B, waits for A
```
Figure 16. Deadlock [27]

This situation can also occur when having more than 2 threads, and it also becomes increasingly more difficult to deal with when the number of threads is increased. Such an example can be seen in **Figure 17.** In this situation, thread 1 waits for thread 2, thread 2 waits for thread 3, thread 3 waits for thread 4, and thread 4 waits for thread 1.


```
Thread 1  locks A, waits for B
Thread 2  locks B, waits for C
Thread 3  locks C, waits for D
Thread 4  locks D, waits for A
```
Figure 17. 4 Threads deadlock [27]

A more complicated situation in which deadlocks can occur is during a database transaction. A database transaction may consist of many SQL update requests. When a record is updated during a transaction, that record is locked for updates from other transactions, until the first transaction completes. Each update request within the same transaction may, therefore, lock some records in the database. If multiple transactions are running at the same time that needs to update the same records, there is a risk of them ending up in a deadlock.

The deadlock could occur in the payment system when two simultaneous transactions take place. In one transaction account, A is debited, and B is debited, while in the second account B is debited, and account A is credited. The first transaction is executed by one thread, and the second by other thread and the deadlock situation presented in **Figure 17** occurs. To prevent this situation from happening, a possible solution is to have a predefined order in which the locks take place. Each account has an ID, and when a transaction takes place, the account with the larger id gets locked first, and only after that, the other account is locked. This way, if the ID of account A is larger than the ID of account B, the situation aforementioned would never occur.

Another important aspect of locking resources from other threads is to identify the shortest portion of code that has to be locked from the other threads. In the context of the debit and credit operations performed by the payment system, the are some operations that happen before the accounting, that can also be executed by other threads such that the time the accounts are locked is minimal. In this way, the accounts are locked only for subtracting funds from the debited account and adding funds to the credited account.

5.3. Thread Creation and Management in Java

Java's multithreading system is built upon the Thread class, its methods, and its companion interface, Runnable [28]. To create a new thread, the Thread class is extended or the Runnable interface implemented. Thread exists in 5 states, including a new state when the instance is created, running state, suspended state, and then can be resumed, blocked when waiting for a resource, and terminated which halts its execution immediately at any given time.

There is however a performance overhead associated with starting a new thread, and each thread is allocated some memory for its stack. Thread Pools are useful when there is a need for a limited number of threads in the application at the same time. Instead of starting a new thread for every task to executed concurrently, the task can be passed to a thread pool. As soon as the pool has any idle threads the task is assigned to one of them and executed. Internally the tasks are inserted into a blocking queue which the threads in the pool are dequeueing from. When a new task is inserted into the queue one of the idle threads will dequeue it successfully and execute it. The rest of the idle threads in the pool will be blocked waiting to dequeue tasks.

The Java ExecutorService interface represents an asynchronous execution mechanism that is capable of executing tasks concurrently in the background. Below, in **Figure 18,** is a diagram illustrating a thread delegating a task to an ExecutorService for asynchronous execution:

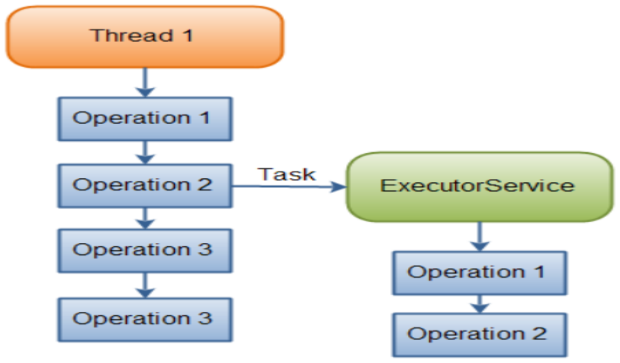


Figure 18. A thread delegating tasks to an ExecutorService

Once the thread has delegated the task to ExecutorService, the thread continues its execution independent of that task. The ExecutorService then executes the task concurrently, independently of the thread submitted the task. The Java ExecutorService is very similar to a thread pool. The implementation of the ExecutorService interface present in java.util.concurrent package is a thread pool implementation [29].

Another aspect to take into consideration is how many threads to use for executing several tasks because sometimes using more threads for the same number of tasks can improve performance over using fewer threads and vice-versa. Some things to take into consideration when diagnosing system performance are CPU bound (the thread needs lots of CPU resources), memory-bound (the thread needs lots of RAM resources), and I/O bound (the thread uses network and/or hard drive resources). All of these three resources are finite and any one of them can limit the performance of a system. A common recommendation is using n+1 threads, where n is the number CPU cores available. That way n threads can work the CPU while 1 thread is waiting for disk I/O. Having fewer threads would not fully utilize CPU resources (at some point there will always be I/O to wait for) and having more threads would cause threads fighting over the CPU resource. Also, threads are not free to use, but they come with overhead like context switches and data that has to be exchanged that comes with the necessity of various locking mechanisms. The only way to find the best solution is to experiment with various numbers of threads and use profiling tools that capture the use of CPU, RAM, and I/O resources over time.

Chapter 6 – Application Development

This chapter is concerned with an in-depth look at the application development phases, more precisely analysis, design, implementation, and testing. For a better understanding of the processes involved, diagrams along with thorough explications will be provided.

6.1. Application Analysis

Simply put in words, a use case is a set of actions or steps that define the interactions between an actor and a system for the user to perform an action on that system. The actor can be represented by a person or an external system. The relationship between use cases, actors, and the application is best represented through a UML use case diagram which is the primary form of representing and visualizing requirements in the earlier development stages of an application. Use case modeling is applied often for specifying the context of a system, capturing the requirements of a system, validating a systems architecture, but also for driving implementation and test case generation [30]. First, we will go through the test-cases for both regular users and administrators and then use a UML use case diagram for better visualization.

Use cases:

- Use case: **Login**
  Actor: Regular user, Administrator
  Preconditions: An account with the used credentials
  Main scenario:
    1. The user is provided with the login screen and enters his\her credentials;
    2. Authentication is successful and the user is logged in.

  Sub-variations:
    2a. The user credentials are not found by the application, and the user is prompted to provide his\her credentials again.

- Use case: **Logout**
  Actor: Regular user, Administrator
  Preconditions: The user is logged in
  Main scenario:
    1. The user clicks the Logout button which is displayed at the top of the page, above the main menu;
    2. The user is logged out of the system.

- Use Case: **Initiate Payment**
  Actor: Regular user, Administrator
  Preconditions: User is logged in
  Main scenario:
    1. The user goes to the *Transaction* menu and selects the *Make Payment* option;
    2. The user fills in the form. If the user is a regular one, he will be provided with his\her accounts to fill the *Debit Account* field;

3. After filling in the form, the user clicks the *Make Payment* button for the process to finish;

Extensions:

2a. If the debit account and the credit account have different currencies, the currency exchange will happen automatically.

Sub-variations:

2b. If during the form verification process errors occur, the user will be notified of what field caused the error and what that field should contain.

- Use Case: **List Own Accounts**
  Actor: Regular user, Administrator
  Preconditions: The user is logged in
  Main scenario:
  1. The user goes to the *Accounts* menu and selects the *My Accounts* option;
  2. A list with all the accounts belonging to that user is displayed.

  Sub-variations:

  2a. The user has no accounts, so instead of the list, the system will display a message stating that the user has no accounts.

- Use Case: **Account Details**
  Actor: Regular user, Administrator
  Preconditions: The user is logged in and listed the list of accounts
  Main scenario:
  1. The user clicks the symbol below the *Details* column for the account he wants to find more details about ;
  2. The details of the account are displayed to the user.

- Use Case: **Search Balance**
  Actor: Regular user, Administrator
  Preconditions: The user accessed the *Account Details* page for an account
  Main scenario:
  1. The user chooses the dates between which he wants to see the transactions;
  2. A list with all the transactions which occurred between the dates provided by the user is displayed.

  Sub-variations:

  2a. There are no transactions that occurred between those dates, and a message displays this information.

- Use Case: **Add User**
  Actor: Administrator
  Preconditions: The user is logged in as an administrator
  Main scenario:
  1. The user goes to the *Users* menu and selects the *Add User* option;
  2. A new page with containing a form is displayed and the user fills in the form;
  3. The user finishes the process by pressing the *Add User* button.

  Sub-variations:

  2a. If there are any errors, the form will be reloaded and the fields that caused the error to happen will have suggestions for what that field should contain.

- Use Case: **Edit User**
  Actor: Administrator
  Preconditions: The user is logged in and listed the list of users
  Main scenario:
    1. The user clicks the symbol below the *Edit* column for the user he wants to find more details about;
    2. The details of the user are displayed to the administrator and he can edit the editable fields;
    3. The user finishes the process by pressing *Update User.*

- Use Case: **Delete User**
  Actor: Administrator
  Preconditions: The user is logged in as an administrator and listed the list of users
  Main scenario:
    1. The user clicks the symbol below the *Delete* column for the user he wants to delete;
    2. The user is deleted from the list.

- Use Case: **List All Users**
  Actor: Administrator
  Preconditions: The user is logged in as an administrator
  Main scenario:
    1. The user goes to the *Users* menu and selects the *All Users* option;
    2. A list of all the users in the system is displayed.

- Use Case: **Search User**
  Actor: Administrator
  Preconditions: The user is logged in as an administrator
  Main scenario:
    1. The user goes to the *Users* menu and selects the *Search User* option;
    2. A new page containing a search field is displayed and the user fills in the field;
    3. The user finishes the process by pressing the *Search* button;
    4. The user with the username filled in the search field is displayed as an element of the list and further actions are possible.

  Sub-variations:
    4a. The user with the username filled in the search is not found and a message indicating this is displayed.

- Use Case: **Edit Account**
  Actor: Administrator
  Preconditions: The user is logged in and listed the list of accounts
  Main scenario:
    1. The user clicks the symbol below the *Edit* column for the account he wants to find more details about;
    2. The details of the account are displayed to the user and he can edit the editable fields;

3. The user finishes the process by pressing *Update Account*.

- Use Case: **Delete Account**
  Actor: Administrator
  Preconditions: The user is logged in as an administrator and listed the list of accounts
  Main scenario:
  1. The user clicks the symbol below the *Delete* column for the account he wants to delete;
  2. The account is deleted.

- Use Case: **List All Accounts**
  Actor: Administrator
  Preconditions: The user is logged in as an administrator
  Main scenario:
  1. The user goes to the *Accounts* menu and selects the *All Accounts* option;
  2. A list with all the accounts in the system is displayed.

  Sub-variations:
  2a. Thre are no accounts in the system, so instead of the list, the system will display a message stating that the user has no accounts.

- Use Case: **Generate Accounts for Load Testing**
  Actor: Administrator
  Preconditions: The user is logged in as an administrator
  Main scenario:
  1. The user goes to the *Load Testing* menu and selects the *Load Testing* option;
  2. A new page containing two buttons is displayed;
  3. The user finishes the process by pressing the *Generate Accounts* button.

- Use Case: **Start Load Testing**
  Actor: Administrator
  Preconditions: The user is logged in as an administrator
  Main scenario:
  1. The user goes to the *Load Testing* menu and selects the *Load Testing* option;
  2. A new page containing two buttons is displayed;
  3. The user finishes the process by pressing the *Start Load Testing* button.

- Use Case: **Verify Payment**
  Actor: Administrator
  Preconditions: The user is logged in as an administrator
  Main scenario:
  1. The user goes to the *Transactions* menu and selects the *Verification* option;
  2. A new page containing all the transactions which are ready to be verified is displayed;
  3. The user chooses the transaction to be verified by pressing on the symbol on the *Verify* column;
  4. The details of the transaction are displayed and the user finishes the process by pressing the *Verify Payment* button.

- Use Case: **Approve Payment**
  Actor: Administrator
  Preconditions: The user is logged in as an administrator
  Main scenario:
    1. The user goes to the *Transactions* menu and selects the *Approval* option;
    2. A new page containing all the transactions which are ready to be approved is displayed;
    3. The user chooses the transaction to be approved by pressing on the symbol on the *Approve* column;
    4. The details of the transaction are displayed and the user finishes the process by pressing the *Approve Payment* button.

- Use Case: **Authorize Payment**
  Actor: Administrator
  Preconditions: The user is logged in as an administrator
  Main scenario:
    1. The user goes to the *Transactions* menu and selects the *Authorization* option;
    2. A new page containing all the transactions which are ready to be authorized is displayed;
    3. The user chooses the transaction to be authorized by pressing on the symbol on the *Authorize* column;
    4. The details of the transaction are displayed and the user finishes the process by pressing the *Authorize Payment* button.

- Use Case: **List All Transactions**
  Actor: Administrator
  Preconditions: The user is logged in as an administrator
  Main scenario:
    1. The user goes to the *Transactions* menu and selects the *All Transactions* option;
    2. A new page containing all the transactions ordered by date is displayed and the user can navigate through the transactions by switching between pages of transactions.

- Use Case: **Filter Trasanctions by Status**
  Actor: Administrator
  Preconditions: The user is logged in as an administrator
  Main scenario:
    1. The user goes to the *Transactions* menu and selects the *Filter By Status* option;
    2. A new page with a dropdown field is displayed and the user has to choose a status from the list and press the *Filter* button;
    3. A new page containing all the transactions ordered by date that have the selected status is displayed and the user can navigate through the transactions by switching between pages of transactions.

All the use cases presented above can also be observed in **Figure 19**, the use-case diagram of the application.

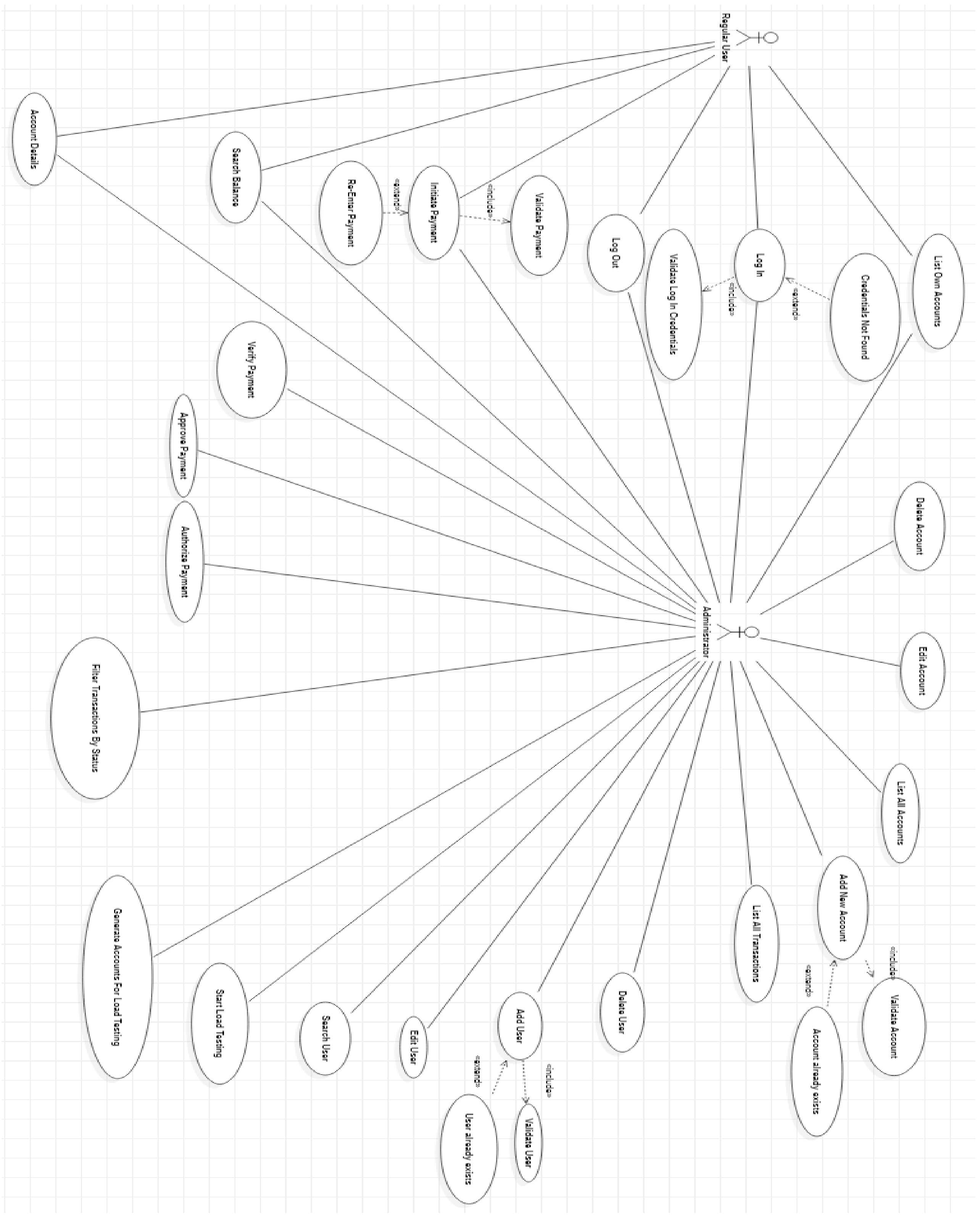
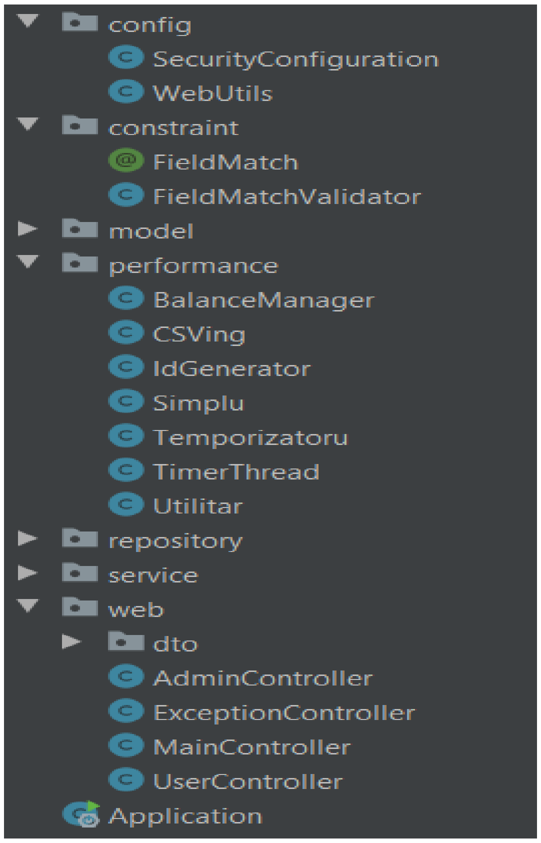


Figure 19. Use Case Diagram for A Simplified Version of Modern Payment Systems

6.2. Application Design

The application packages are distributed based on their functionality as presented in **Figure 20**. The first visible delimitation is the one from the MVC structure, so there are separate packages for models, repositories, services which are an extra layer and controllers. There is also a separate folder in the controller's package which is used for the DTOs (Data Transfer Object) classes which are used for validating input when entering a new user, account, or transaction in the system. The config package contains the class with all the configurations regarding what paths of the application can be accessed by different users and also the configuration for encrypting the password and maintaining user sessions, and also a class with some utility methods for user input. The constraint package contains 2 classes, both created to validate and match user input especially in the forms used for creating and editing users, accounts, and transactions. Finally, the performance package contains all the classes that back up the load testing functionality, including the ones used for working with CSV files, multi-threading, logging, and timing the progress of the application.



```
▼ config
      SecurityConfiguration
      WebUtils
▼ constraint
      FieldMatch
      FieldMatchValidator
► model
▼ performance
      BalanceManager
      CSVing
      IdGenerator
      Simplu
      Temporizatoru
      TimerThread
      Utilitar
► repository
► service
▼ web
   ► dto
      AdminController
      ExceptionController
      MainController
      UserController
  Application
```

Figure 20. Package Design

Due to the increased number of classes and methods of each class, only the most relevant packages and with the most class relationships will be presented in the following class diagrams. Figure **Figure 21** represents the class diagram for the *model* package. Half of the classes inherit from the *BaseEntity* class which only has an identifier to avoid code duplication. There are also classes used for comparing dates and payments that are used for displaying transactions in sorted order. *MyException* is the class used for all exceptions occurring in the system where a message has to be displayed to the user. Figure **Figure 22** displays the class diagram of the service layer where every class has an interface such that the declarations and the implementations of the methods are kept separate. Every class that is also displayed in a way or another to the user, has a repository to help with database data retrieval and a service to help with business logic. The controllers that can be observed in **Figure 20** are split such that there is a controller for the administrator, one for the regular user, one for pages both types of users can access, and an exception controller used for displaying errors. **Figure 23** represents the performance package and the classes used for the load testing feature. *BalanceManager*, *Utilitar*, and *Simplu* are used for thread creation, thread management, locking, and unlocking resources, while the *TimerThread* and *Temporizatoru* are used for timing every step of the process and the *CSVing* class is used for reading and creating CSV files. The class components are loosely coupled and there are not a lot of dependencies between classes or packages.

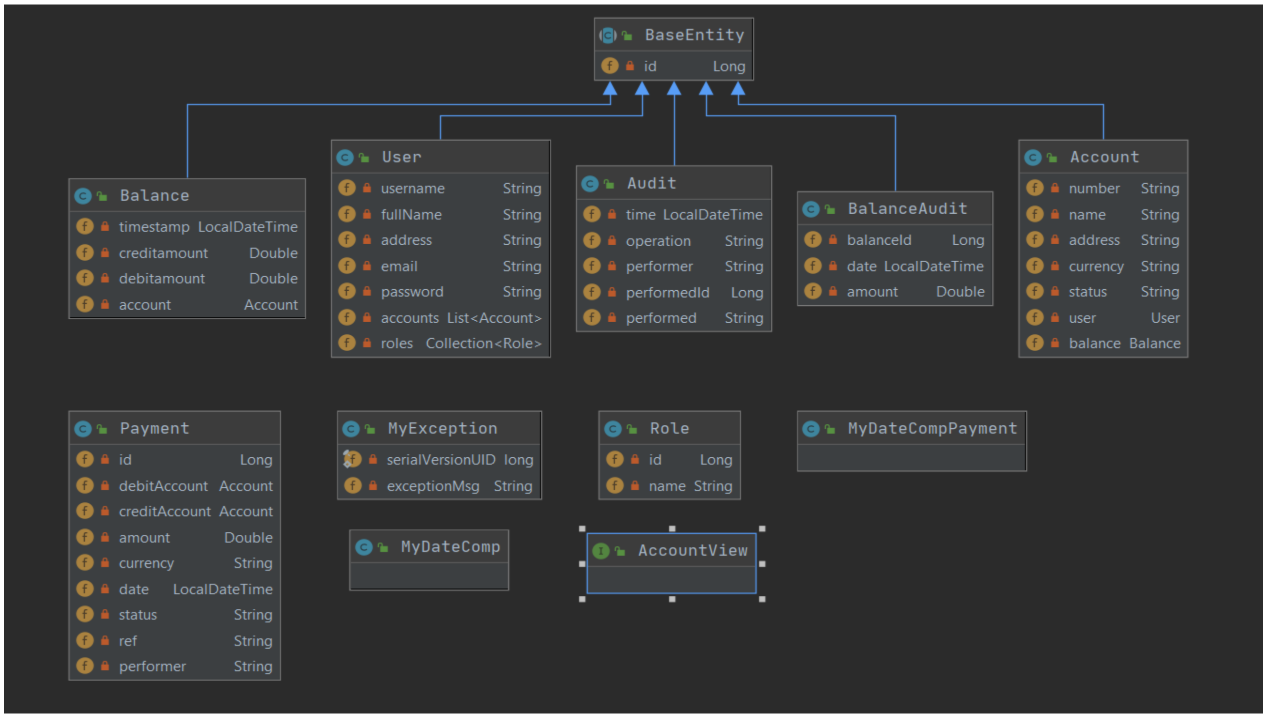


Figure 21. Model Package Diagram

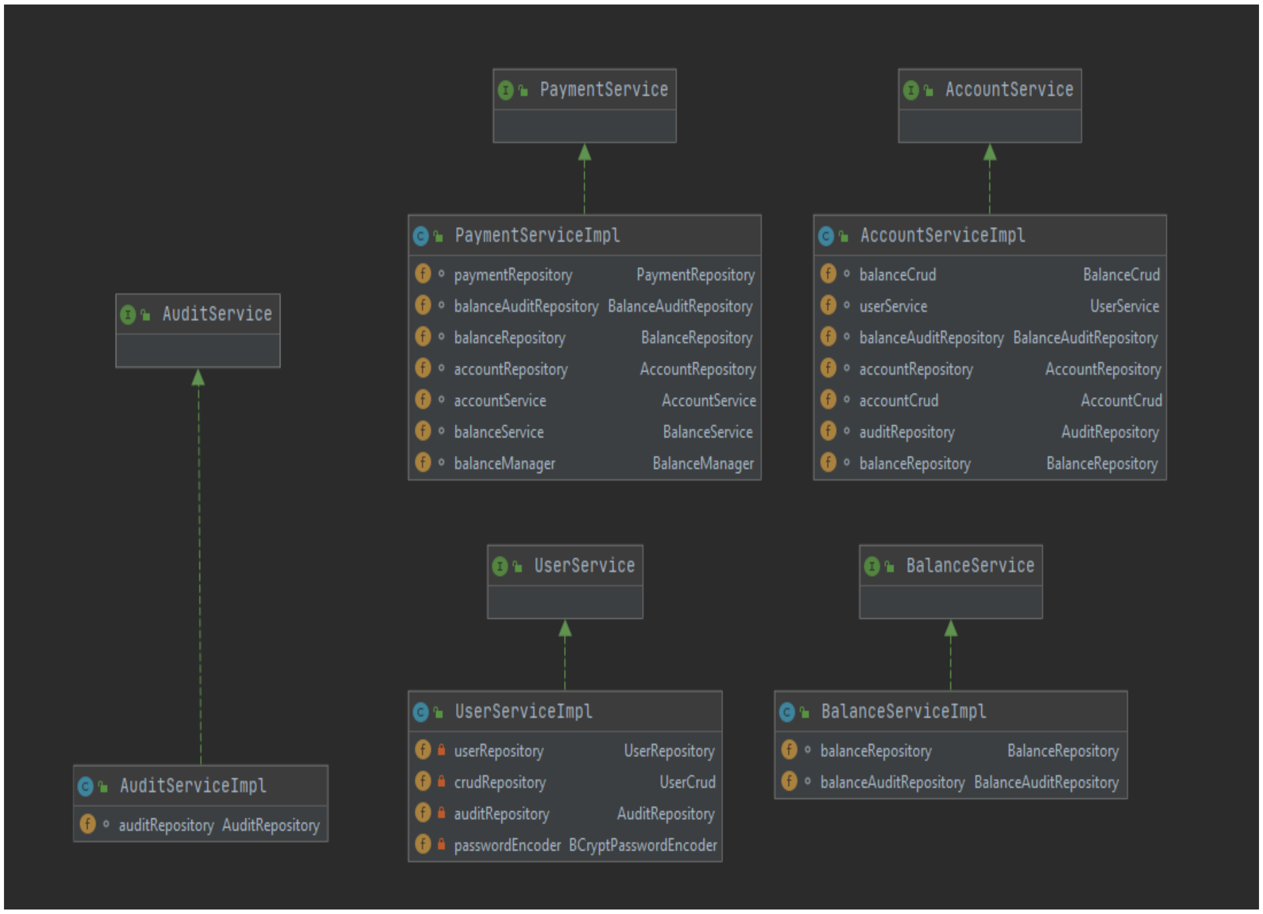

PaymentService
AccountService
PaymentServiceImpl
paymentRepository PaymentRepository
balanceAuditRepository BalanceAuditRepository
balanceRepository BalanceRepository
accountRepository AccountRepository
accountService AccountService
balanceService BalanceService
balanceManager BalanceManager
AccountServiceImpl
balanceCrud BalanceCrud
userService UserService
balanceAuditRepository BalanceAuditRepository
accountRepository AccountRepository
accountCrud AccountCrud
auditRepository AuditRepository
balanceRepository BalanceRepository
AuditService
UserService
BalanceService
UserServiceImpl
userRepository UserRepository
crudRepository UserCrud
auditRepository AuditRepository
passwordEncoder BCryptPasswordEncoder
BalanceServiceImpl
balanceRepository BalanceRepository
balanceAuditRepository BalanceAuditRepository
AuditServiceImpl
auditRepository AuditRepository

Figure 22. Service Package Diagram

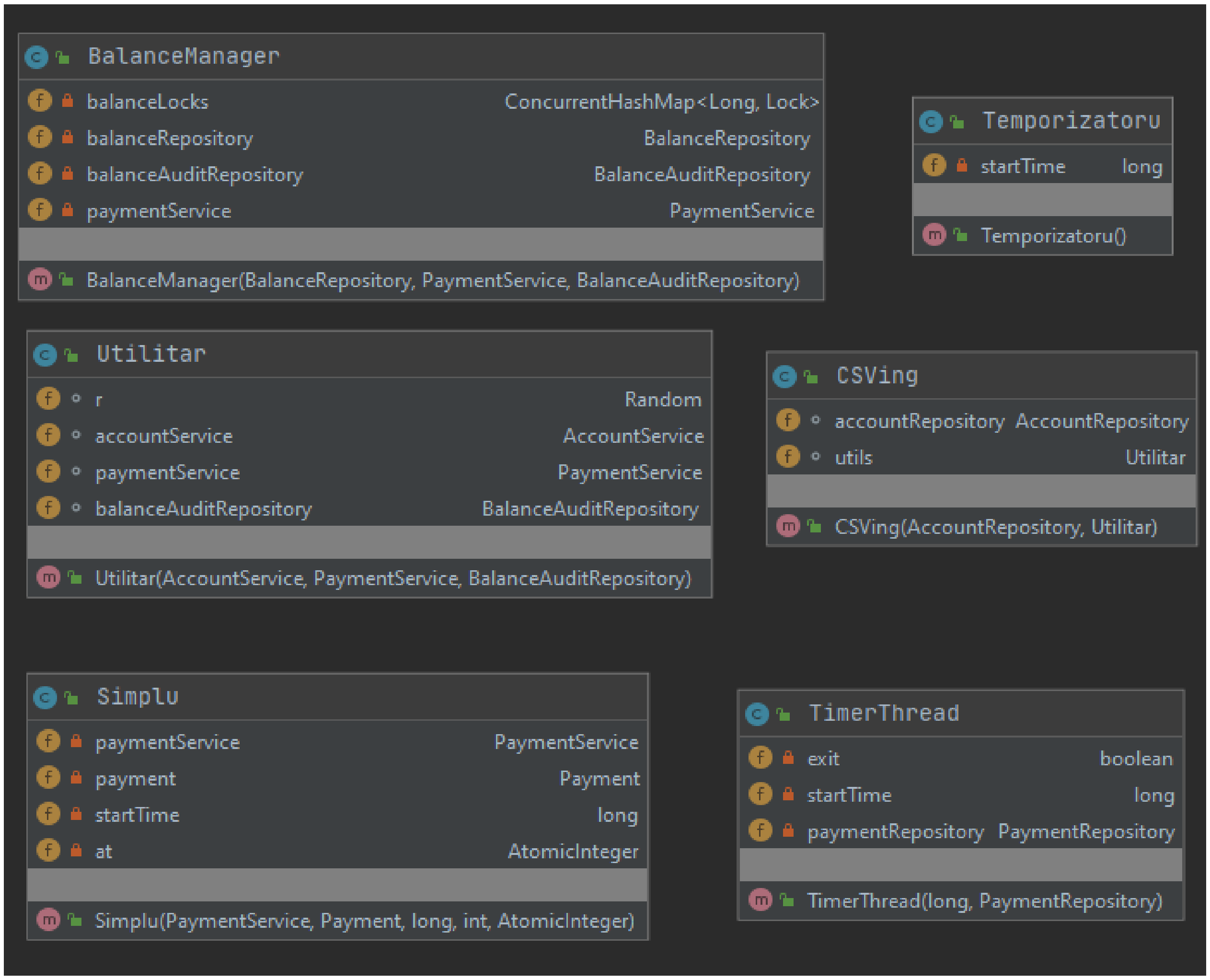

BalanceManager
balanceLocks ConcurrentHashMap<Long, Lock>
balanceRepository BalanceRepository
balanceAuditRepository BalanceAuditRepository
paymentService PaymentService
BalanceManager(BalanceRepository, PaymentService, BalanceAuditRepository)
Temporizatoru
startTime long
Temporizatoru()
Utilitar
r Random
accountService AccountService
paymentService PaymentService
balanceAuditRepository BalanceAuditRepository
Utilitar(AccountService, PaymentService, BalanceAuditRepository)
CSVing
accountRepository AccountRepository
utils Utilitar
CSVing(AccountRepository, Utilitar)
Simplu
paymentService PaymentService
payment Payment
startTime long
at AtomicInteger
Simplu(PaymentService, Payment, long, int, AtomicInteger)
TimerThread
exit boolean
startTime long
paymentRepository PaymentRepository
TimerThread(long, PaymentRepository)

Figure 23. Performance Package Diagram

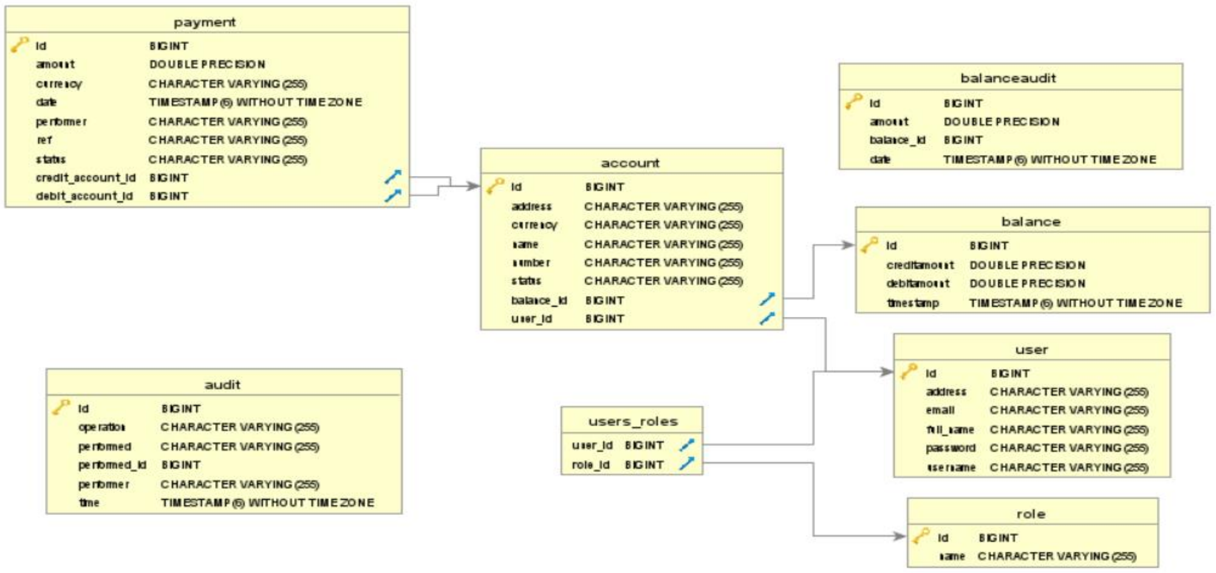

payment
id BIGINT
amount DOUBLE PRECISION
currency CHARACTER VARYING (255)
date TIMESTAMP (6) WITHOUT TIME ZONE
performer CHARACTER VARYING (255)
ref CHARACTER VARYING (255)
status CHARACTER VARYING (255)
credit_account_id BIGINT
debit_account_id BIGINT
balanceaudit
id BIGINT
amount DOUBLE PRECISION
balance_id BIGINT
date TIMESTAMP (6) WITHOUT TIME ZONE
account
id BIGINT
address CHARACTER VARYING (255)
currency CHARACTER VARYING (255)
name CHARACTER VARYING (255)
number CHARACTER VARYING (255)
status CHARACTER VARYING (255)
balance_id BIGINT
user_id BIGINT
balance
id BIGINT
creditamount DOUBLE PRECISION
debitamount DOUBLE PRECISION
timestamp TIMESTAMP (6) WITHOUT TIME ZONE
user
id BIGINT
address CHARACTER VARYING (255)
email CHARACTER VARYING (255)
full_name CHARACTER VARYING (255)
password CHARACTER VARYING (255)
username CHARACTER VARYING (255)
audit
id BIGINT
operation CHARACTER VARYING (255)
performed CHARACTER VARYING (255)
performed_id BIGINT
performer CHARACTER VARYING (255)
time TIMESTAMP (6) WITHOUT TIME ZONE
users_roles
user_id BIGINT
role_id BIGINT
role
id BIGINT
name CHARACTER VARYING (255)

Figure 24. Database Diagram

Figure 24 presents the database diagram. The main links between tables are the ones connecting accounts, users, and transactions. The *payment* table has two foreign keys, one for the ID of the debit account and one for the ID of the credit account. The *account* table has two foreign keys, one for connecting the accounts with its user and one for linking the account with its balance. There is a table for users, one for roles, and one that associates each user with a specific role, basically, a many-to-many relationship between two tables with a third table as intermediate. The *audit* table is used for registering actions performed on users and accounts and the *balanceaudit* table is used for registering the transactions performed on each account.

6.3. Application Implementation

Spring Framework was the most helpful tool involved in developing the application due to its annotation system and minimal configuration. The *@Table* and *@Entity* annotations were used for mapping the actual classes to database tables. Other helpful annotations used for creating one-to-one and one-to-many relationships were *@OneToOne* and *@OneToMany*, *@ManyToOne* respectively. Foreign keys were created with the help of the *@JoinColumn* annotations and the specification of the *@Id* annotation made it easy to state the primary key and use pre-defined incrementing algorithms for the identifier column. Another helpful feature was the *@Lazy* annotation and the *FetchType.LAZY* option, both allowing the application to fetch data from the database only if that data is needed so the RAM is not filled with useless information.

The *@Repository*, *@Service*, and *@Controller* annotations were used for delegating attributions to each class in the multi-layered design and there is also a *@ControllerAdvice* annotation which was used for developing a different controller for displaying errors and exceptions to the user. Spring also comes with annotations to help with the validation process of the user input such as *@Empty*, *@NonEmpty*, *@FieldMatch,* and *@Email* that indicates that a specific field must obey the annotation signification which is self-explanatory. Other useful annotations that were used are the *@Configuration* which indicates that the annotated class is used for configuring the paths of the application and the *@SpringBootApplication* which indicates the main class of the application.

For configuring the application and specifying the external resources used in developing the application, there is a *pom.xml* file where all the external dependencies and libraries for Spring, SpringBoot, Thyemeleaf, and OpenCSV are enumerated. All configurations for the database connection, database properties, and query logging can be found in the *application.properties* file.

6.4. Application Testing

To verify the application behaves how it should and there are no unexpected behaviors a testing phase was necessary. This phase is split between functional and non-functional testing.

Functional testing is a type of testing that translates to verifying that each function of the software application operates in conformance with the functional requirements specified in the

third chapter [31]. Every functionality of the system is tested by providing appropriate input, verifying
the output, and comparing the actual results with the expected results and can be done either manually or using automation, but in our case, it was manual due to the simple to use interface and the short time for testing every scenario since the application is not that complex. This functional manual testing was done by walking through every use case and verifying if the application responded accordingly to both valid and invalid input. Manual testing was carefully executed especially when implementing a new feature related to transactions which are the main focus of the application. By repeatedly manual testing the application every time a new feature is introduced we can ensure that our feature was integrated successfully and there are no unexpected behaviors of the application.

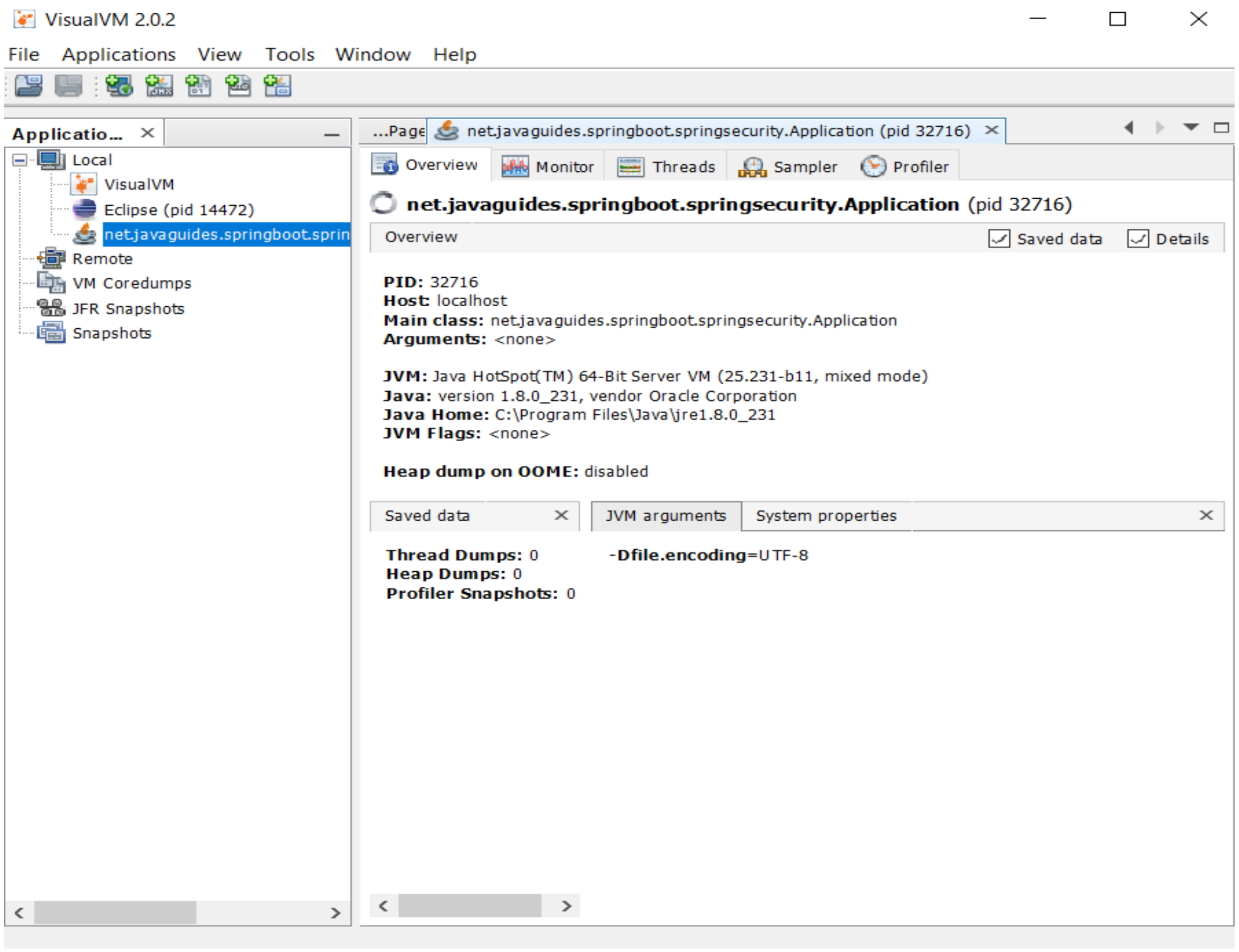
Non-functional testing is a type of testing that translates to verifying that each of the non-functional requirements is met and the application is intuitive and acts such that the end-user is satisfied with the application generally, without going into detail about specific features. This phase was also performed manually by setting some standards and criteria that the application should meet and then performing a walkthrough of the application. This walkthrough was done without focusing on the actual functionalities, but rather on how fast the application responds, checking if every part of the process is intuitive and how the application behaves when facing invalid input. An important part of the non-functional testing is also the load testing feature of the application where logging results and execution times, and resolving transactions without erroneous accounting was vital. The general walkthrough and more detailed walkthroughs of every menu of the application were performed multiple times during development and ensured that application is easy to use, intuitive, responsive, fast and nothing feels off.

In conclusion, testing was a really important task to perform as much as possible during the process of developing the application because the application, even though not so complex, has to have no unexpected behavior and be as intuitive and easy to use as possible since its purpose is to deal with transactions and ultimately people’s and business’s funds.

6.5. Application Performance

This subchapter is concerned with profiling and measuring the overall performance, RAM, and CPU consumption of the application by analyzing every layer, and what bottlenecks or problems might occur performance-wise. However, because the front-end of the application is server-side rendered, the main focus will be on the back-end and the database. The main subject will be the batch transaction feature of the application, but also later queries performed for checking balances after those transactions.

VisualVM is a tool that integrates command-line JDK tools and lightweight profiling capabilities and it can be used to monitor and troubleshoot Java applications. The main features are displaying local and remote Java processes configuration and environment, and monitoring process performance and memory. These features can also be observed in **Figure 25.**



net.javaguides.springboot.springsecurity.Application (pid 32716)

Overview

```
PID: 32716
Host: localhost
Main class: net.javaguides.springboot.springsecurity.Application
Arguments: <none>

JVM: Java HotSpot(TM) 64-Bit Server VM (25.231-b11, mixed mode)
Java: version 1.8.0_231, vendor Oracle Corporation
Java Home: C:\Program Files\Java\jre1.8.0_231
JVM Flags: <none>

Heap dump on OOME: disabled
```

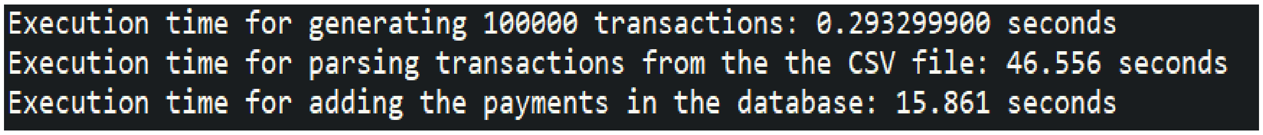
```
Thread Dumps: 0
Heap Dumps: 0
Profiler Snapshots: 0
```

```
-Dfile.encoding=UTF-8
```

Figure 25. VisualVM

The application provides a menu for creating the accounts and the transactions required for this load testing feature. The accounts are created in the database and the transactions are inserted in a CSV file from where it will be later processed. For a simpler interaction with the CSV file where the transactions are firstly written to and the read from into memory, a library called OpenCSV that provides an easy to use API for working with such documents was used. After this step is finished with, everything is ready and the load testing can start.

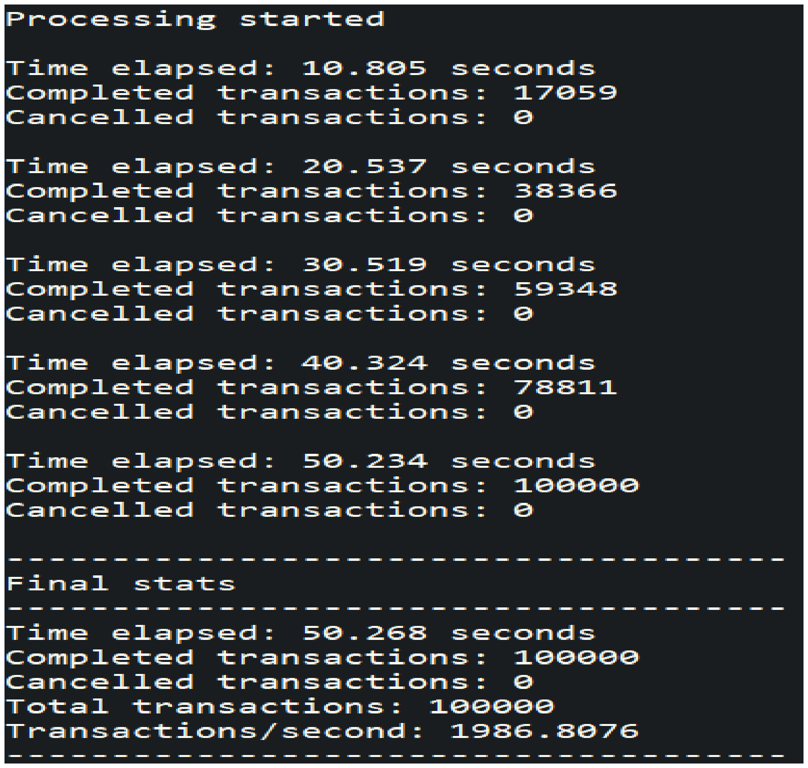
In the same menu where the accounts and transactions are generated, there is another button for starting the load testing. Every step regarding transaction processing is logged in the console with the name of the event and the execution time of that specific event. **Figure 26** presents the execution times of the pre-processing steps.



```
Execution time for generating 100000 transactions: 0.293299900 seconds
Execution time for parsing transactions from the the CSV file: 46.556 seconds
Execution time for adding the payments in the database: 15.861 seconds
```

Figure 26. Pre-processing times

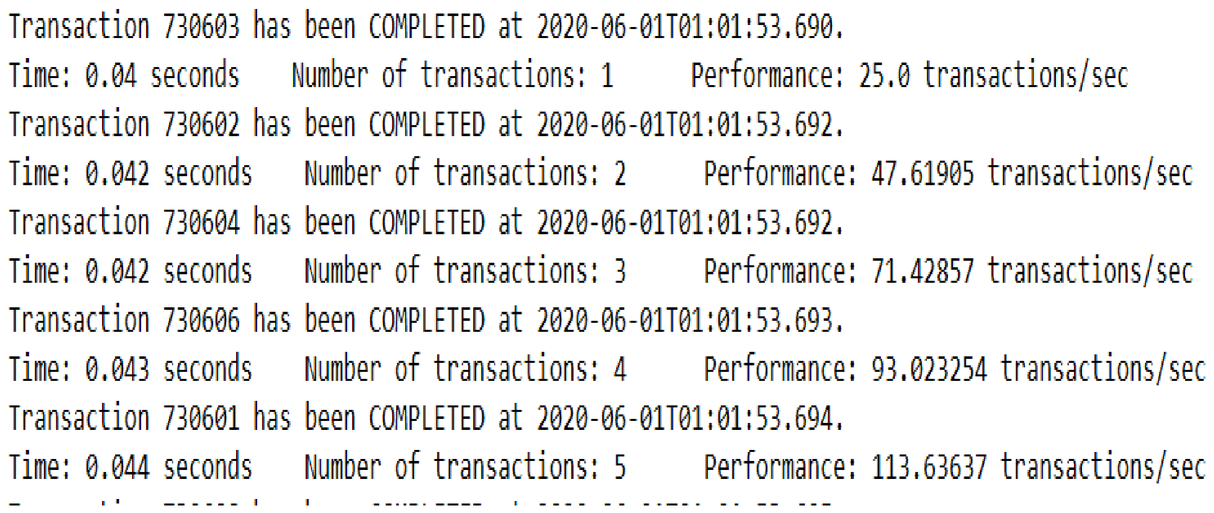
Figure 19 presents a log that is created every 10 seconds stating the elapsed time, and how many transactions were completed and canceled followed by the final stats of the load-testing process.



```
Processing started

Time elapsed: 10.805 seconds
Completed transactions: 17059
Cancelled transactions: 0

Time elapsed: 20.537 seconds
Completed transactions: 38366
Cancelled transactions: 0

Time elapsed: 30.519 seconds
Completed transactions: 59348
Cancelled transactions: 0

Time elapsed: 40.324 seconds
Completed transactions: 78811
Cancelled transactions: 0

Time elapsed: 50.234 seconds
Completed transactions: 100000
Cancelled transactions: 0

-------------------------------------
Final stats
-------------------------------------
Time elapsed: 50.268 seconds
Completed transactions: 100000
Cancelled transactions: 0
Total transactions: 100000
Transactions/second: 1986.8076
-------------------------------------
```

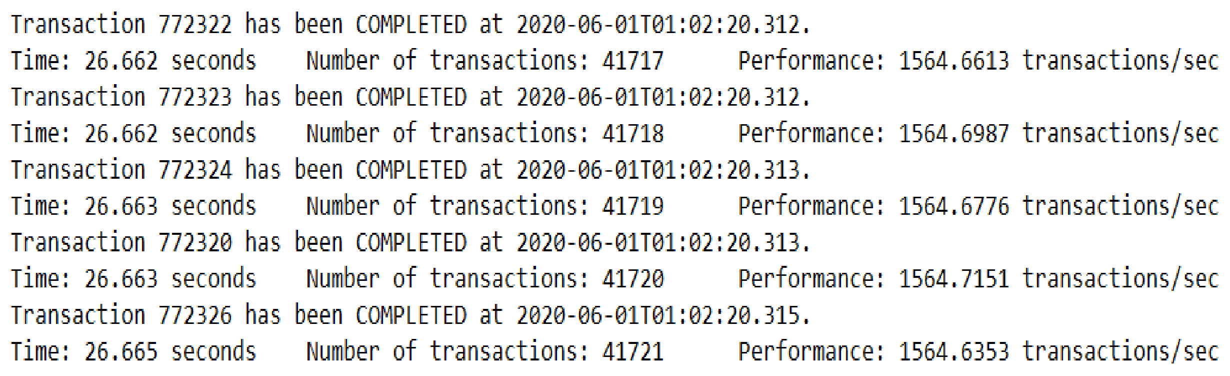
Figure 27. Processing times

Every transaction also produces a log in a text file stating its id, the time of the completion, the time elapsed since the processing began, the number of the transaction since the processing began, and also the performance until that point expressed by transactions per second. By taking a look at **Figure 28**, **Figure 29** and **Figure 30** we can conclude that overall performance increases overtime since at the 5th transaction, the performance was 113 transactions/second and by the 41721st transaction it grew up to 1564 transaction, and at the last transaction, it was around 1850 transactions/second.



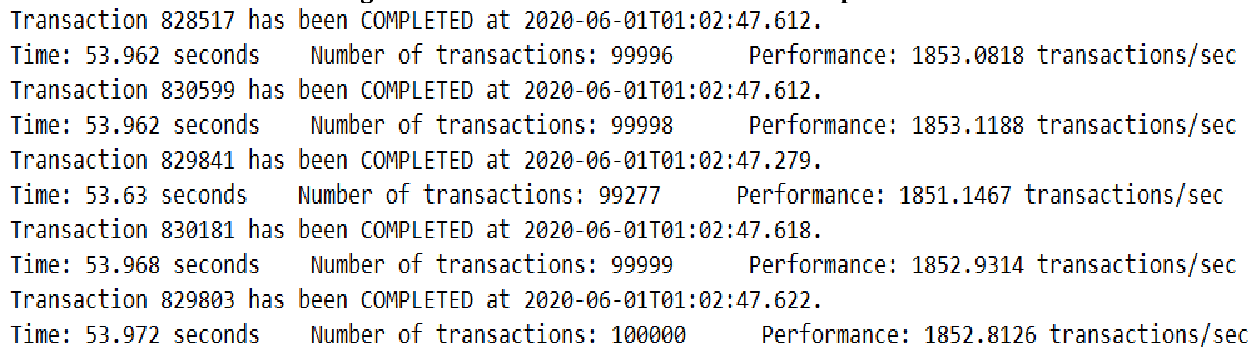
```
Transaction 730603 has been COMPLETED at 2020-06-01T01:01:53.690.
Time: 0.04 seconds    Number of transactions: 1     Performance: 25.0 transactions/sec
Transaction 730602 has been COMPLETED at 2020-06-01T01:01:53.692.
Time: 0.042 seconds    Number of transactions: 2     Performance: 47.61905 transactions/sec
Transaction 730604 has been COMPLETED at 2020-06-01T01:01:53.692.
Time: 0.042 seconds    Number of transactions: 3     Performance: 71.42857 transactions/sec
Transaction 730606 has been COMPLETED at 2020-06-01T01:01:53.693.
Time: 0.043 seconds    Number of transactions: 4     Performance: 93.023254 transactions/sec
Transaction 730601 has been COMPLETED at 2020-06-01T01:01:53.694.
Time: 0.044 seconds    Number of transactions: 5     Performance: 113.63637 transactions/sec
```

Figure 28. Performance at the beginning of the process



```
Transaction 772322 has been COMPLETED at 2020-06-01T01:02:20.312.
Time: 26.662 seconds    Number of transactions: 41717      Performance: 1564.6613 transactions/sec
Transaction 772323 has been COMPLETED at 2020-06-01T01:02:20.312.
Time: 26.662 seconds    Number of transactions: 41718      Performance: 1564.6987 transactions/sec
Transaction 772324 has been COMPLETED at 2020-06-01T01:02:20.313.
Time: 26.663 seconds    Number of transactions: 41719      Performance: 1564.6776 transactions/sec
Transaction 772320 has been COMPLETED at 2020-06-01T01:02:20.313.
Time: 26.663 seconds    Number of transactions: 41720      Performance: 1564.7151 transactions/sec
Transaction 772326 has been COMPLETED at 2020-06-01T01:02:20.315.
Time: 26.665 seconds    Number of transactions: 41721      Performance: 1564.6353 transactions/sec
```

Figure 29. Performance in the middle of the process



```
Transaction 828517 has been COMPLETED at 2020-06-01T01:02:47.612.
Time: 53.962 seconds    Number of transactions: 99996      Performance: 1853.0818 transactions/sec
Transaction 830599 has been COMPLETED at 2020-06-01T01:02:47.612.
Time: 53.962 seconds    Number of transactions: 99998      Performance: 1853.1188 transactions/sec
Transaction 829841 has been COMPLETED at 2020-06-01T01:02:47.279.
Time: 53.63 seconds    Number of transactions: 99277      Performance: 1851.1467 transactions/sec
Transaction 830181 has been COMPLETED at 2020-06-01T01:02:47.618.
Time: 53.968 seconds    Number of transactions: 99999      Performance: 1852.9314 transactions/sec
Transaction 829803 has been COMPLETED at 2020-06-01T01:02:47.622.
Time: 53.972 seconds    Number of transactions: 100000      Performance: 1852.8126 transactions/sec
```

Figure 30. Performance during final transactions

During the processing phase, we can use VisualVM's feature along with Windows's Task Manager to see how the CPU and the RAM behave during the processing phase. To tell if the multithreading of the application went well, the CPU consumption should be around 100% all the time during the processing phase because that translates to an efficient organization of the threads, the locks, and how the tasks are distributed between threads. The threads with their name starting with "pool-6" are the threads working on processing and as **Figure 32** reflects, a thread spends most of its time performing an action, rather than waiting for other resources to be unlocked. **Figure 31** presents the CPU consumption menu for 60 seconds, and we can observe how the consumption suddenly raises to 100%, that being the starting point of the processing phase. By observing these visual representations of how the CPU and the threads behave during the processing phase, we can conclude that the application does a good job at locking resources and sharing tasks between threads.

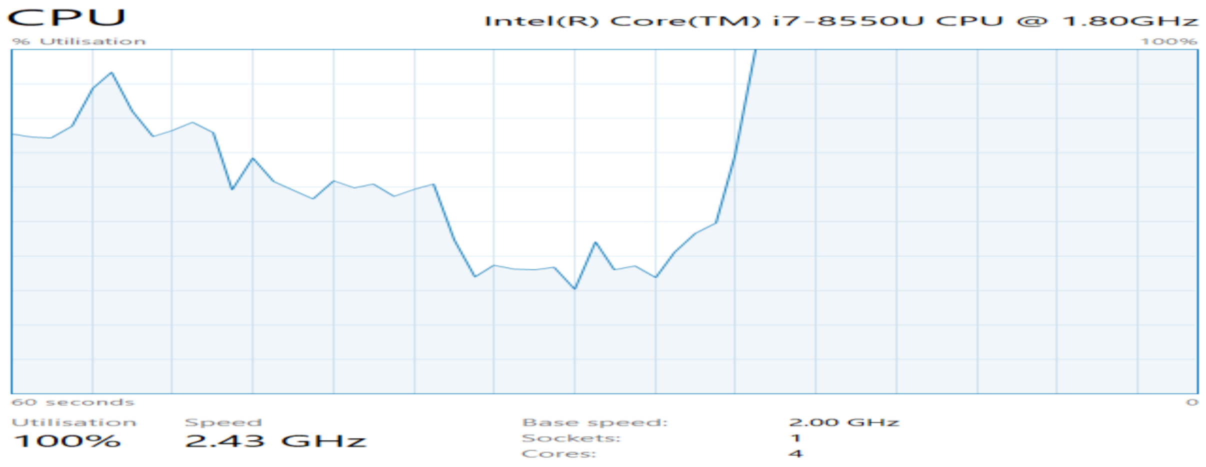
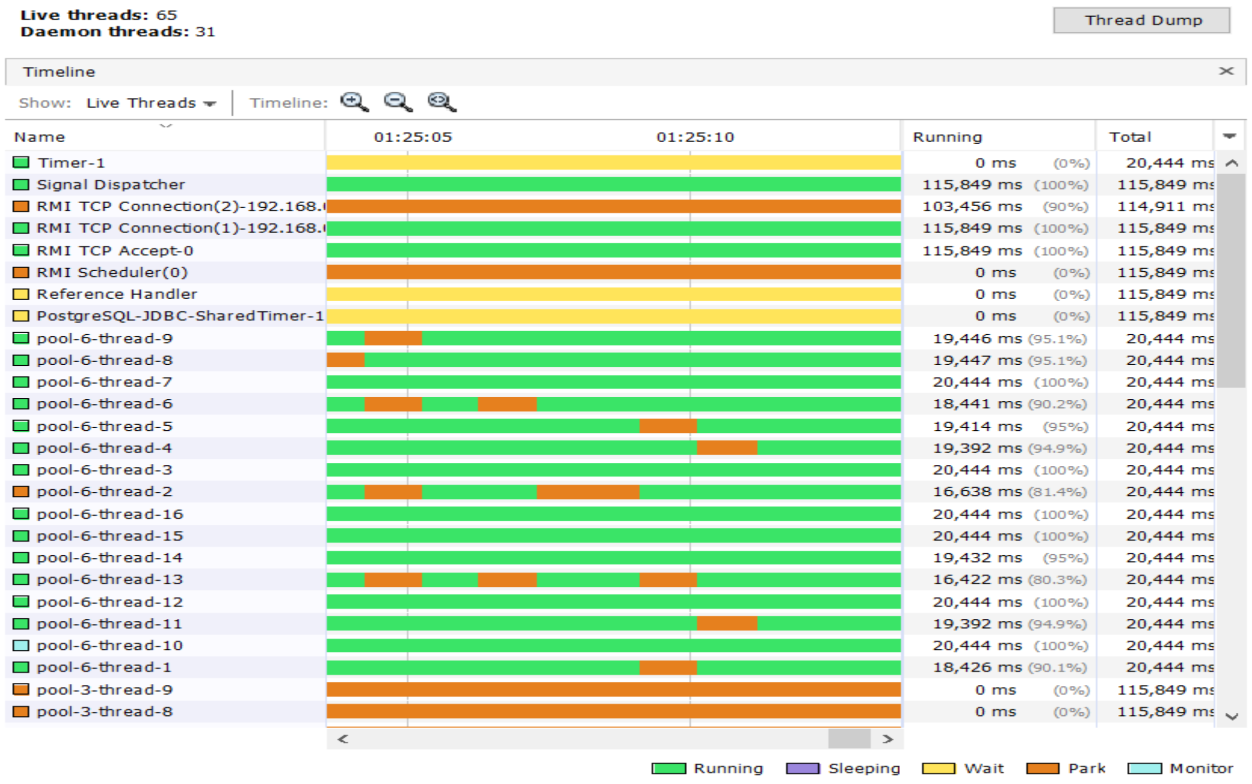


Figure 31. CPU Consumption during Processing

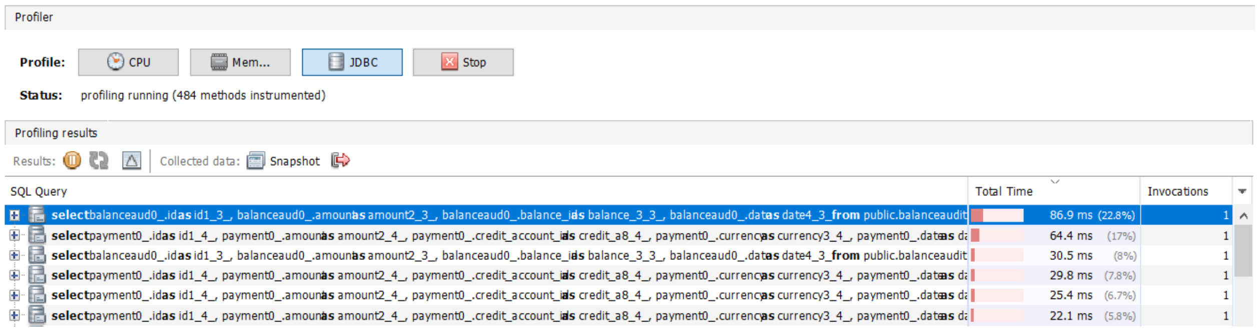

Live threads: 65
Daemon threads: 31

Name	01:25:05 01:25:10	Running		Total
🟩 Timer-1		0 ms	(0%)	20,444 ms
🟩 Signal Dispatcher		115,849 ms	(100%)	115,849 ms
🟧 RMI TCP Connection(2)-192.168.		103,456 ms	(90%)	114,911 ms
🟩 RMI TCP Connection(1)-192.168.		115,849 ms	(100%)	115,849 ms
🟩 RMI TCP Accept-0		115,849 ms	(100%)	115,849 ms
🟧 RMI Scheduler(0)		0 ms	(0%)	115,849 ms
🟨 Reference Handler		0 ms	(0%)	115,849 ms
🟨 PostgreSQL-JDBC-SharedTimer-1		0 ms	(0%)	115,849 ms
🟩 pool-6-thread-9		19,446 ms	(95.1%)	20,444 ms
🟩 pool-6-thread-8		19,447 ms	(95.1%)	20,444 ms
🟩 pool-6-thread-7		20,444 ms	(100%)	20,444 ms
🟩 pool-6-thread-6		18,441 ms	(90.2%)	20,444 ms
🟩 pool-6-thread-5		19,414 ms	(95%)	20,444 ms
🟩 pool-6-thread-4		19,392 ms	(94.9%)	20,444 ms
🟩 pool-6-thread-3		20,444 ms	(100%)	20,444 ms
🟧 pool-6-thread-2		16,638 ms	(81.4%)	20,444 ms
🟩 pool-6-thread-16		20,444 ms	(100%)	20,444 ms
🟩 pool-6-thread-15		20,444 ms	(100%)	20,444 ms
🟩 pool-6-thread-14		19,432 ms	(95%)	20,444 ms
🟩 pool-6-thread-13		16,422 ms	(80.3%)	20,444 ms
🟩 pool-6-thread-12		20,444 ms	(100%)	20,444 ms
🟩 pool-6-thread-11		19,392 ms	(94.9%)	20,444 ms
🟦 pool-6-thread-10		20,444 ms	(100%)	20,444 ms
🟩 pool-6-thread-1		18,426 ms	(90.1%)	20,444 ms
🟧 pool-3-thread-9		0 ms	(0%)	115,849 ms
🟧 pool-3-thread-8		0 ms	(0%)	115,849 ms

🟩 Running 🟪 Sleeping 🟨 Wait 🟧 Park 🟦 Monitor

Figure 32. VisualVM Thread Menu during Processing

Another feature of the VisualVM that could help us improve the application performance is the JDBC Profiler. This feature comes in handy when we try to find out which queries performed on the database are expensive time-wise as shown in **Figure 33**.



Status: profiling running (484 methods instrumented)

SQL Query	Total Time		Invocations
selectbalanceaud0_.id**as** id1_3_, balanceaud0_.amoun**as** amount2_3_, balanceaud0_.balance_i**ds** balance_3_3_, balanceaud0_.dat**as** date4_3_**from** public.balanceaudit	86.9 ms	(22.8%)	1
selectpayment0_.id**as** id1_4_, payment0_.amoun**as** amount2_4_, payment0_.credit_account_i**ds** credit_a8_4_, payment0_.currenc**as** currency3_4_, payment0_.dat**as** da	64.4 ms	(17%)	1
selectbalanceaud0_.id**as** id1_3_, balanceaud0_.amoun**as** amount2_3_, balanceaud0_.balance_i**ds** balance_3_3_, balanceaud0_.dat**as** date4_3_**from** public.balanceaudit	30.5 ms	(8%)	1
selectpayment0_.id**as** id1_4_, payment0_.amoun**as** amount2_4_, payment0_.credit_account_i**ds** credit_a8_4_, payment0_.currenc**as** currency3_4_, payment0_.dat**as** da	29.8 ms	(7.8%)	1
selectpayment0_.id**as** id1_4_, payment0_.amoun**as** amount2_4_, payment0_.credit_account_i**ds** credit_a8_4_, payment0_.currenc**as** currency3_4_, payment0_.dat**as** da	25.4 ms	(6.7%)	1
selectpayment0_.id**as** id1_4_, payment0_.amoun**as** amount2_4_, payment0_.credit_account_i**ds** credit_a8_4_, payment0_.currenc**as** currency3_4_, payment0_.dat**as** da	22.1 ms	(5.8%)	1

Figure 33. VisualVM's JDBC Profiler

By exploring the results, we can conclude which query was the most used and which queries are the most expensive and act accordingly by performing different database tuning techniques, such as adding indexes.

In conclusion, profiling and measuring performance tools give us solid knowledge about how our application behaves under different circumstances, so we have a better idea as to where improvements need to be done and we can then apply well-known tuning techniques.

Chapter 7 – Conclusions and Future Work

Summarizing the first chapters, we can conclude that trading goods and services through intermediate agents were a common practice dating back to the first agricultural civilizations. First came coins made out of different precious metals, and slowly society made the transition to money, and now a new decentralized system that uses virtual coins is rapidly growing along with the fintech industry represented by companies such as Revolut or Curve. Payment systems are a vital component in the modern banking and fintech industries, and their improvement through technology enables regular people, businesses, and governments to transfer funds almost instantly. Being such an important tool in such a sensible business domain, payment systems should excel in security, performance, reliability, and availability. Developing such a system is not an easy task, but a clear development trajectory and using Agile best practices would make it easier.

The application is a very simplified version of an actual payment system, but still has a level of complexity to it. Real payment systems can vary from around 100.000 lines of code to even 1.000.000 lines. However, the complexity varies depending on the specific requirements of such a payment system and on the end-user. We have seen the basic requirements and functionalities such a system has to meet and provide. Java is heavily used in developing such systems, or at least important modules of such systems, due to its popularity among other programming languages and its support for parallel and concurrency techniques. We have also introduced a possible tech stack for developing the application, discussing each technology along with its advantages and drawbacks in detail.

The latter chapters discussed the requirements such a system should meet and how we can achieve them. By making use of multithreading, and revising the theory behind it, we saw how multithreading makes better use of RAM and CPU resources to increase the performance. We also used existing tools for profiling, monitoring, testing, and improving the application. By providing functional and non-functional requirements, and using functional and non-functional testing we proved that these phases should not be missing from any development cycle.

Further improvements should include:

- Specializing the payment system for one specific purpose such that it becomes usable in certain domains, rather than being generic.

- Working on the application's networking capabilities. Usually, such systems work with messages which contain payment instructions. These messages are usually sent through secure networks due to the importance of the message and its content.

- Making the application more secure and immune to usual web-applications vulnerabilities such as SQL injection and cross-site scripting.

- Making the web interface more user friendly, and considering the use of client-side rendering modern JavaScript frameworks such as React or Angular as an alternative for Thyemeleaf and server-side rendering.

Bibliography

[1] L. Y. a. H. Yu, in *Chinese Coins: Money in History and Society*, Long River Press, p. 3.

[2] H. U. Vogel, in *Marco Polo Was in China: New Evidence from Currencies, Salts and Revenues*, p. Page 171–173.

[3] G. Abhisek, "Concurrency, Parallelism, Threads, Processes, Async and Sync — Related?," 29 December 2018. [Online]. Available: https://medium.com/swift-india/concurrency-parallelism-threads-processes-async-and-sync-related-39fd951bc61d. [Accessed 07 May 2020].

[4] J. Jenkov, "Java Concurrency and Multithreading Tutorial," 29 03 2020. [Online]. Available: http://tutorials.jenkov.com/java-concurrency/index.html#what-is-multithreading. [Accessed 12 05 2020].

[5] D. Grenville, "Coinage In Western Continental Europe, Africa, And The Byzantine Empire.," Britannica, [Online]. Available: https://www.britannica.com/topic/coin/Charlemagne-and-the-Carolingian-coinages. [Accessed 6 May 2020].

[6] S. Tuli, "Creating Threads and Multithreading in Java," 18 June 2018. [Online]. Available: https://dzone.com/articles/java-thread-tutorial-creating-threads-and-multithr. [Accessed 14 May 2020].

[7] J. Jenkov, "Java ExecutorService," 28 October 2018. [Online]. Available: http://tutorials.jenkov.com/java-util-concurrent/executorservice.html. [Accessed 14 May 2020].

[8] "Main Page," 29 October 2018. [Online]. Available: https://www.thymeleaf.org. [Accessed 15 May 2020].

[9] "Fundamentals of Global Payment Systems and Practices," 2018. [Online]. Available: https://www.treasuryalliance.com/assets/publications/payments/Fundamentals_of_Payment_Systems.pdf. [Accessed 19 April 2020].

[10] "Handbook of world stock, derivative and commodity exchanges website," [Online]. Available: http://www.exchange-handbook.co.uk/articles_story.cfm. [Accessed 20 April 2020].

[11] J. Redman, "Satoshi Nakamoto's Brilliant White Paper Turns 9-Years Old," Bitcoin, 31 October 2017. [Online]. Available: https://news.bitcoin.com/satoshi-nakamotos-brilliant-white-paper-turns-9-years-old/. [Accessed 21 April 2020].

[12] "Spring Data," [Online]. Available: https://spring.io/projects/spring-data. [Accessed 29 March 2020].

[13] "Spring Framework," [Online]. Available: https://spring.io/projects/spring-framework. [Accessed 5 April 2020].

[14] "Spring MVC," [Online]. Available: https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#mvc. [Accessed 5 May 2020].

[15] "Spring Security," [Online]. Available: https://docs.spring.io/spring-security/site/docs/current/reference/html5/#jc-method. [Accessed 9 May 2020].

[16] "Thymeleaf," [Online]. Available: https://www.thymeleaf.org/index.html. [Accessed 16 April 2020].

[17] "What is a Relational Database?," [Online]. Available: https://aws.amazon.com/relational-database/. [Accessed 7 May 2020].

[18] "What is Java?," [Online]. Available: https://java.com/en/download/faq/whatis_java.xml. [Accessed 5 April 2020].

[19] "What is Non-Functional Requirement? Types and Examples," [Online]. Available: https://www.guru99.com/non-functional-requirement-type-example.html. [Accessed 29 March 2020].

[20] "What is PostgreSQL?," [Online]. Available: https://www.postgresql.org/about/. [Accessed 2 May 2020].

[21] D. Arias, "Hashing in Action: Understanding bcrypt," 31 May 2018. [Online]. Available:

https://auth0.com/blog/hashing-in-action-understanding-bcrypt/. [Accessed 3 May 2020].

[22] A. Beattie, 13 May 2019. [Online]. Available: https://www.investopedia.com/articles/07/roots_of_money. [Accessed 5 May 2020].

[23] A. Augustyn, "Java, Computer Programming Language," [Online]. Available: https://www.britannica.com/technology/Java-computer-programming-language. [Accessed 20 April 2020].

[24] B. Burkholder, "JavaScript SEO: Server Side Rendering vs. Client Side Rendering," 6 July 2018. [Online]. Available: https://medium.com/@benjburkholder/javascript-seo-server-side-rendering-vs-client-side-rendering-bc06b8ca2383. [Accessed 25 April 2020].

[25] G. DAUGHERTY, "Real-Time Gross Settlement (RTGS)," 10 September 2019. [Online]. Available: https://www.investopedia.com/terms/r/rtgs. [Accessed 7 April 2020].

[26] R. Fadatare, "What Is the Difference Between Hibernate and Spring Data JPA?," 31 December 2018. [Online]. Available: https://dzone.com/articles/what-is-the-difference-between-hibernate-and-sprin-1. [Accessed 4 May 2020].

[27] W. Kenton, 9 June 2019. [Online]. Available: https://www.investopedia.com/terms/p/payment.asp. [Accessed 10 April 2020].

[28] W. KENTON, "Automated Clearing House (ACH)," 14 November 2019. [Online]. Available: https://www.investopedia.com/terms/a/ach.asp. [Accessed 20 April 2020].

[29] M. Smallcombe, "SQL vs. NoSQL: How Are They Different and What Are the Best SQL and NoSQL Database Systems?," 23 August 2019. [Online]. Available: https://www.xplenty.com/blog/the-sql-vs-nosql-difference/. [Accessed 8 May 2020].

[30] "Purpose of Use Case Diagram," VisualParadigm, [Online]. Available: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-use-case-diagram/. [Accessed 02 June 2020].

[31] "Functional Testing Vs Non-Functional Testing: What’s the Difference?," [Online]. Available: https://www.guru99.com/functional-testing-vs-non-functional-testing.html. [Accessed 04 June 2020].