TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Maksim Nessin 179589IADB

Creation of a Test Environment for the Fintech Application Developed by Polybius Tech OÜ

Bachelor's thesis

Supervisors: Nadežda Furs-Nižnikova
MBA
Vadim Gerassimov
MSc

Tallinn 2020

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Maksim Nessin 179589IADB

Testkeskkonna loomine Polybius Tech OÜ poolt loodud *fintech* rakenduse jaoks

Bakalaureusetöö

Juhendajad: Nadežda Furs-Nižnikova
MBA
Vadim Gerassimov
MSc

Tallinn 2020

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Maksim Nessin

21.11.2020

Abstract

The aim of this thesis is to implement a test environment for the fintech application developed by Polybius Tech OÜ. The solution is made for the developers and testers of the company to increase the quality of the product and optimize resources invested in testing.

The solution is divided into two parts: virtualization of third-party services and data generator. Virtualization helps to avoid interacting with other services by emulating of their behaviour. The aim of the data generator is to save time and money while developing or testing the application.

The project has drastically minified the resources of the company required for testing and it has become a helpful solution among the developers.

This thesis is written in English and is 59 pages long, including 7 chapters, 14 figures and 5 tables.

Annotatsioon

Testkeskkonna loomine Polybius Tech OÜ poolt loodud *fintech* rakenduse jaoks

Käesoleva bakalaureusetöö eesmärk on luua testkeskkond Polybius Tech OÜ poolt loodud *fintech* rakenduse jaoks. See lahendus on valmistatud firma arendajatele ja testijatele rakenduse kvaliteedi tõstmiseks ja testimise jaoks mõeldud ressursside optimeerimiseks.

Lahendus koosneb kahest osast: kolmandate osapoolte teenuste virtualiseerija ja andmete generaator. Virtualiseerimise abil saab vältida suhtlust kolmandate osapooltega, emuleerides nende funktsioone. Andmete generaatori eesmärgiks on raha ja aja säästmine rakenduse testimisel või arendamisel.

Projekt on märgatavalt vähendanud testimiseks kasutatud ressursse ja on saanud kasulikuks lahenduseks firma arendajatele.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 59 leheküljel, 7 peatükki, 14 joonist, 5 tabelit.

List of abbreviations and terms

AI	Artificial Intelligence
API	Application Programming Interface
Bitcoin	The first cryptocurrency based on the blockchain technology
Blockchain	Series of immutable data validated by a cluster of computers
BSON	Binary JavaScript Object Notation
Crypto asset	Digital analogue of traditional money
Crypto Autopilot	The name of the product developed in Polybius Tech OÜ that uses a third-party trading algorithm
Cryptocurrency	Digital analogue of traditional money
DTO	Data Transfer Object
ERD	Entity-Relationship diagram
Ethereum	The competitor of Bitcoin that includes a network of other crypto assets
Fiat	Traditional currency
Fintech	Financial technology – a new vision of traditional finances
HTTP	HyperText Transfer Protocol
JDBC	Java DataBase Connectivity
JSON	JavaScript Object Notation
KYC	Know Your Customer – user's verification procedure
Liquidity provider	Service that can store and exchange crypto or fiat assets
NPM	Node.js package manager
PSD2	Revised Payment Service Directive
REST	Representational State Transfer – the architecture of communication between web services
RTS	Regulatory Technical Standards
SQL	Structured Query Language
UI	User Interface
VPN	Virtual Private Network
XML	eXtensible Markup Language

Table of contents

List of figures

List of tables

1 Introduction

1.1 Background

“The banking industry is ripe for change with the rise of fintech startups, the growing popularity of blockchain technology, and the dominance of millennials.” [1]

The Fintech industry shows new ways of improving traditional finances, banking, stocks and other financial services. This is a new ideology in the financial world that brings people Internet banking, expense managers, blockchain technologies, cryptocurrencies, trading algorithms, stock markets and many more. The traditional financial services are becoming old and weak allowing new projects to turn up on the scene. Information technology and mobile connectivity open the road for different applications that are going to reinvent the finances, making them easier to use, fast and understandable for everyone. Contactless payments, face-to-face transactions, digital cards, E-commerce - these are functions that we use today and there is no doubt that many more are on the way. Smart devices can be integrated into the so-called Internet of Things, a network of devices and services to communicate through. [2, p. 2]

However, the services must have proper security. A growing number of start-ups increase the attempts of breaking the system. A solid and secure service should be created to be commercially viable and ready for the real market that includes hackers and fraud. [2, p. 3]

This approach requires thorough tests before opening the project to the world. Mistakes can be very costly, as fintech deals with real money and the reputation of one company is easy to lose but hard to rebuild.

1.2 Problem

Polybius Tech OÜ specializes in creating a service for cryptocurrency and fiat management. The application works with third-party services and provides its own

features in the fintech sphere as well. The current development and testing processes of the company are complicated, take much time for preparation and require using real money for testing the application. It leads to slow delivery, loss of company money and poor quality of the product. The developers and testers need a solution that will help them to test, observe and fix issues before delivering features to the live environment.

1.3 Goal

The main purpose of the thesis is to create a test environment for the fintech application developed by Polybius Tech OÜ that will emulate the behaviour of the third-party services used by this application. Additionally, this environment will allow inserting different data presets to improve the testing of scenarios.

The advantages of using a separate development environment are the following:

- The environment serves as a platform for testing new code, functions and different scenarios. [3, p. 3]
- It closely represents the production environment which in turn allows performing regression testing, patch verification and problem determination before delivering new features. [3, p. 3]

1.4 Methodology

In order to solve the problem, it was decided to analyse the most time and money consuming steps of development and testing of the application. It is dedicated to understanding the correlation between those steps and the quality of the application. The solution should simplify these processes to increase the speed of development and raise the involvement in testing among the developers and testers.

Third-party solutions for similar problems are reviewed and compared. The benefits of those solutions are taken into account while creating a test environment for the fintech application of Polybius Tech OÜ.

As this solution is going to be used by the developers and testers, it is planned to create documentation containing database ERD-scheme and API documentation. The following

step is to choose a suitable framework based on the requirements and implement the solution. The final step is to access the solution, describe how the development process has changed and suggest possible future improvements for the test environment.

2 Problem overview

2.1 The current process of development and testing

The most time and money consuming parts of the development and testing of the fintech application are analyzed in the following chapters.

2.1.1 Cryptocurrency wallets and blockchain transactions

Currently, the application supports the creation of personal Bitcoin or Ethereum wallets. Each wallet has a public address (or hash) and a private key that is used to sign transactions made from this public address. A transaction is written into a blockchain and verified by other computers connected to the blockchain network. In case the signature for the transaction is incorrect or the cryptocurrency wallet does not have enough coins, the transaction is dropped out of the block [4, p. 53]. For Bitcoin network, six confirmations are widely considered to be safe and secure enough to prove your transaction will be valid and permanent [5]. Ethereum will likely require ten confirmations (~three minutes) to achieve a similar degree of security [6]. The average time spent to confirm a block in a Bitcoin network is 10 minutes [2, p. 150]. Ethereum block confirmation time depends on the chosen fee for the transaction but the average transaction confirmation time is 17 seconds [6].

The fintech application stores private keys and signs transactions on a different platform. The microservices do not have direct access to those keys, however, they request the third-party service for creating new cryptocurrency wallets and signing the transactions. The problem is that every time the functionality requires testing, new cryptocurrency wallets should be created and stored on the third-party service. Testing of transactions requires having actual cryptocurrency coins on a wallet and transaction fee to perform the transaction. Each transaction needs to be confirmed to ensure its authenticity and irreversibleness which takes time depending on the cryptocurrency type. Only users with verified identity can have crypto wallets in the application, so a KYC procedure should

be passed when a new test user is created. The full procedure of testing of the crypto wallet can be seen in Table 1.

Table 1. The full procedure of testing of crypto wallet functionality.

Action	**Time expenses**	**Money expenses**	**Notes**
User verification (KYC)	15 minutes	3.19$ [7]	Requires real documents (passport, driver's license). Provided by Veriff OÜ. The price is for the minimal plan if monthly verifications exceeded.
Crypto wallet creation			Stored on Microsoft Azure server.
Bitcoin transaction	60 minutes	5€ [8]	Transaction fee is approximate and may change.
Ethereum transaction	3 minutes	2€ [9]	Transaction fee is approximate and may change.

As a result, an integration test of the crypto wallet will require 78 minutes and nearly 10€ taken from the company's account.

2.1.2 Fiat and cryptocurrency exchange

The application provides functionality for exchanging fiat currencies to cryptocurrencies and vice versa. The current system supports Euro, Bitcoin and Ethereum to be exchanged. There is a fee of 1.5% for every exchange operation. The problem is that there's no option to avoid this fee as every financial operation should be reflected in the company's annual reports. Certainly, these operations may be overwritten as test operations by separating true and test data and manually changing financial reports, however, it is not considered as a long-term solution as it takes accountants' time. The full procedure of testing of exchange functionality for a new user can last up to 1 day and can be seen in Table 2.

Table 2. The full procedure of testing of fiat and cryptocurrency exchange.

Action	**Time expenses**	**Money expenses**	**Notes**
User verification (KYC)	15 minutes	3.19$ [7]	Requires real documents (passport, driver's license). Provided by Veriff OÜ. The price is for the minimal plan if monthly verifications exceeded.
Initiating a deposit via bank	Up to 1 day	0.38€ [10]	A real bank transaction is executed.
Exchange operation		1.5% fee	
Blockchain transaction	3-60 minutes	2-5€ [8]	Fee and time estimation depends on the chosen cryptocurrency.

An average test of fiat and cryptocurrency exchange can take more than 1 full day and approximately 10€. The money expenses can be higher as they depend on the chosen cryptocurrency and the amount to be exchanged.

2.1.3 Banking services

On 25 November 2015, the European Parliament and the council of the European Union issued a directive 2015/2366/EC (PSD2) on payment services in the internal market. [11]

This directive is dedicated to make the European payments market more integrated and efficient, simplify business for new payment service providers and make payments safe and secure. This directive also enforces EU banks to provide some of the functionality via API. It creates an opportunity to use banking functionality in different fintech applications. [12]

At this time, Polybius Tech OÜ and its affiliates are not licensed to operate with the European banks. However, to continue the development process of banking functionality it is required to create a system that will emulate API responses from banks. This solution will help to implement the functionality of adding a new bank account, transferring money and viewing transaction history. Unless the company is licensed to operate with

the European banks, the test environment will be used instead. In case the company becomes licensed, the emulated bank services will be used in the development and testing stages, while the actual integration with banks will be implemented in the production environment. Using the test environment while developing the banking functionality will allow switching to the real bank API implementation faster as most of the logic will have been already implemented.

2.1.4 Services that use AI trading algorithm

In the fintech industry, the AI algorithm is also called *Robo advisor*, but it does not always give advice or may not be robotic. It executes automated portfolio rebalancing based on passive investments and diversification strategies. Individuals with crypto assets find it useful as it can be used as a passive investment tool. [13, p. 152]

The fintech application uses an AI trading algorithm provided by a third-party. It makes decisions about exchanging Bitcoin to other cryptocurrencies and tries to increase the user's holdings. Despite its core and the algorithm are not described in this thesis, it is still possible to describe the emulated version of it for operating in the test environment. In this thesis, the term ‘liquidity provider’ is used for the service that performs trades with cryptocurrencies. The name of the product that uses this algorithm is *Crypto Autopilot*.

The functionality of the product is based on cryptocurrency transactions and exchange. It means that the testing process of *Crypto Autopilot* takes similar time and money expenses as the previously described features. The full process of testing of this product can be seen in Table 3.

Table 3. The full procedure of testing *Crypto Autopilot*.

Action	**Time expenses**	**Money expenses**	**Notes**
User verification (KYC)	15 minutes	3.19$ [7]	Requires real documents (passport, driver's license). Provided by Veriff OÜ. The price is for the minimal plan if monthly verifications exceeded.
Initiating a deposit via bank	Up to 1 day	0.38€ [10]	A real bank transaction is executed.
Exchange operation		1.5% fee	
Bitcoin transaction	60 minutes	5€ [8]	Transaction fee is approximate and may change.
Trading operation		0.2% fee	Executed on the liquidity provider side.
Bitcoin transaction	60 minutes	5€ [8]	Transaction fee is approximate and may change.

A procedure of testing the functionality of *Crypto Autopilot* can last for more than one full day and require more than 13€. The final expenses depend on the amounts to be exchanged and traded.

2.2 Reasons for the poor quality of the application

During 2 years of development of the application without a test environment, a lot of bugs and issues were found. Currently, developers do not have an opportunity to test some of the functionality without risking their money. Some new features cannot be checked fast for technical reasons. For instance, processing of card payments or cryptocurrency transactions are done outside of the company and therefore require more than 1 hour to be confirmed. If developers do not want to spend their money, the company will provide the money for testing the application. In this case, every action must be recorded by the

developer, increasing the time of development. This is the reason why some of the integration testing processes are usually ignored by developers.

Quality Assurance specialists also lack a test environment. Plenty of scenarios such as functionality for different user types, users without KYC, accounts with empty portfolios require proper testing. Creation or generation of users with different settings is done manually in the database that in turn takes much time. These actions could have been automated by using a test environment and the remaining time may be used to cover more cases with tests.

The Product Department also needs a test environment for viewing new features while they are being developed. It would be useful for Product Managers to have a picture of the feature before it is published. In this case, corrections in the user interface, change of the flow or some other proposals could be done in advance. It will exclude cases when a feature is published but the quality of it is not appropriate for a Product Manager.

2.3 Possible solutions from third parties

It was decided to look for a possible solution that has the required functionality for a reasonable price. Fortunately, there are plenty of projects that provide functionality for emulating API requests available on the Internet.

2.3.1 Traffic parrot

Traffic Parrot Ltd (founded in 2014) provides a solution for API mocking, stubbing and service virtualization. It supports dynamic responses, Maven and Gradle plugins, HTTP/2.0 protocol, Docker, Swagger and other modern technologies. However, the price for it depends on the number of floating licenses and features, protocols and technologies used. [14]

For this moment, this solution may seem to be too expensive and complicated as it has a lot more features than our company needs.

2.3.2 MockLab

MockLab solution is a platform for manual and automated testing. It supports Swagger documentation, the ability to create delays and faults in API responses, recording queries and playing it back, dynamically templating responses and simulation of system

behaviour. According to the pricing, the free plan allows only 5 mock API requests and 1000 requests per month that is not enough for our company. [15]

The most suitable plan is very expensive and therefore the company is not interested in this solution.

2.3.3 API Simulator

The latest version of API Simulator (1.7) provides mostly all technologies for creating a solid test environment that can be used by the developers and testers. This solution is free and is constantly updated. However, it does not support Swagger out-of-the-box. It also supports dynamic responses configured in YAML programming language. The biggest disadvantage of this solution is that it requires a lot of manual configuration to set it up. [16]

2.3.4 Summary of third party solutions

The third-party solutions can solve the problems, but the potential resources invested in setting them up are not justified. Some of the solutions have paid plans whereas the suggested free solution is difficult to set up, falls outside the architecture used in the company and requires constant support. It would be more convenient if the solution would be closer to the current architecture and be known to the employees.

2.4 Scope

This thesis deals with the creation of a test environment dedicated only to the fintech application developed by Polybius Tech OÜ. The main purposes of this environment are assistance with the continuous development of the application and implementation of a platform for pre-release integration testing. It is based on common practices of creation of similar environments or service emulators.

The solution is divided into two parts: data generator and service virtualizer.

Data generator provides the functionality of inserting test data, setting up the scenarios and generating historical data. It replaces the manual insertion of entities to the database. This data is to be used by other microservices of the fintech application which are not described in this thesis.

Another part, service virtualizer, deals with third-parties' API emulation. The solution provides dynamic API responses and emulates the logic of other services outside of the fintech application. Specifically, it emulates the logic of creation of the Bitcoin and Ethereum wallet and blockchain transactions in these networks. Additionally, it emulates the logic of connecting a bank account, viewing bank transactions and sending bank payments in accordance with 2015/2366/EC (PSD2) [11] directive. Moreover, a trading algorithm based on cryptocurrencies provided by third-party service is emulated in this solution. The algorithm itself remains unrevealed, however, base endpoints and documentation are provided in this thesis.

2.5 Role of the author

The author of the thesis analyzed the problems that existed in the process of development and testing of the application. The author gave the exact amount of resources invested in the testing and calculated approximate company's losses based on the public resources. Third-party solutions were analyzed, and it was explained why they are not suitable for achieving the goal of the thesis.

The author explored scientific resources that describe similar solutions and prepared the requirements for the test environment based on the common approaches. The framework, additional software and plugins were chosen according to the author's skills and the company's needs. The author proposed the design of the test environment including the ERD scheme of the database, projects structure and API documentation.

The implementation of the solution was done in a cooperation work with one senior developer of the company. The author participated in the development of all the parts of the test environment described in the scope of the thesis.

3 Solution analysis

3.1 Requirements

The requirements are divided into groups. The common requirements are based on the currently used architecture in the company, convenience of usage and requests from the developers and testers. The other recommendations are mostly justified by the business logic of the fintech application.

3.1.1 Common requirements

The solution should have the functionality of data generation and the virtualization of third-party services. It should be monolithic and runnable with one Docker container locally on a developer's machine. Common practices of creating similar solutions should be taken into account while creating the test environment for the fintech application. One of the most important requirements is that this solution has to be capable of subsequent improvements and easily integrable with the current architecture of Polybius Tech OÜ (see Figure 1).

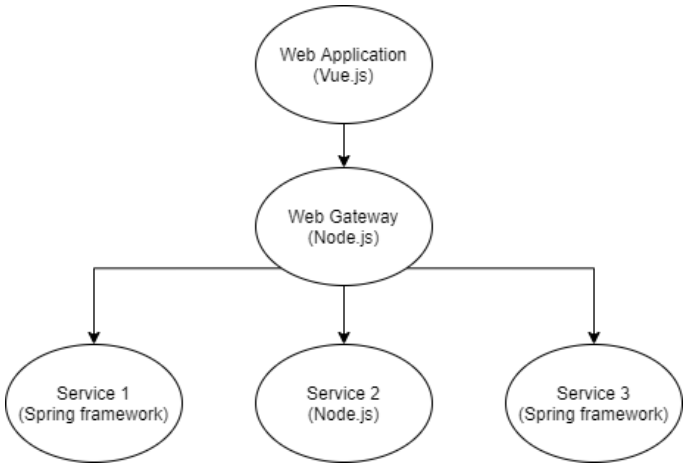


Figure 1. Scheme of the microservice architecture of the company.

The requirements are justified by the fact that the solution should have a solid architecture and at the same time be produced in the shortest timeframe. Besides, the code should be understandable for the employees of the company.

3.1.2 Requirements for data generation functionality

Data generation should be done by using a separate database source for each microservice that is observed within the scope of the thesis. It should be accessible via a browser and contain proper documentation to be understandable for non-technical users. The microservices that fall out of the scope or are inessential to the development and testing processes should not be covered with data generation functionality.

The solution should support the generation of the following data:

- User and KYC
- Cryptocurrency wallets and transactions
- User balance
- Bank accounts and transactions
- Historical data of *Crypto Autopilot*

3.1.3 Requirements for virtualization of third-party services

The virtualization of third-party services should be based on conventional architecture. The book “Testing Microservices with Mountebank” describes the basic principles of service virtualizations (see Figure 2). A system that is being tested uses virtual dependencies (or third-party services) instead of addressing real services. [17, p. 12]

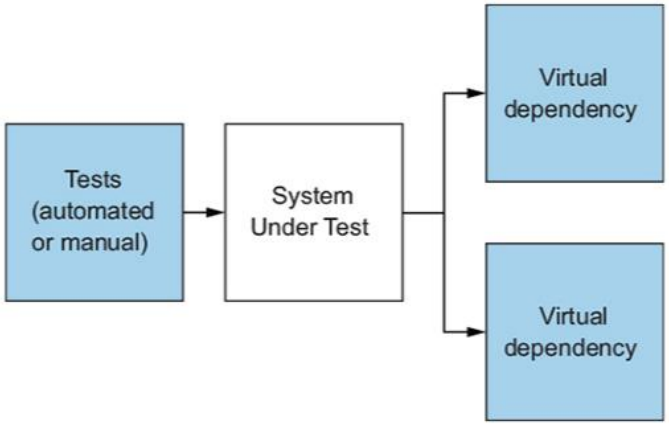


Figure 2. Testing using service virtualization. [17, p. 12]

Another book, “Service virtualization, 2nd IBM Limited Edition”, provides a similar approach to using service virtualization. From Figure 3, it can be seen that one service is unavailable due to some reasons. In the observed fintech application, it can be either a Bitcoin transaction that takes a fee and needs much time for confirmation or a bank service that is unavailable due to lack of license. This service can be virtualized in the test environment to save money and time or to make the development process possible even without real integration with the service.

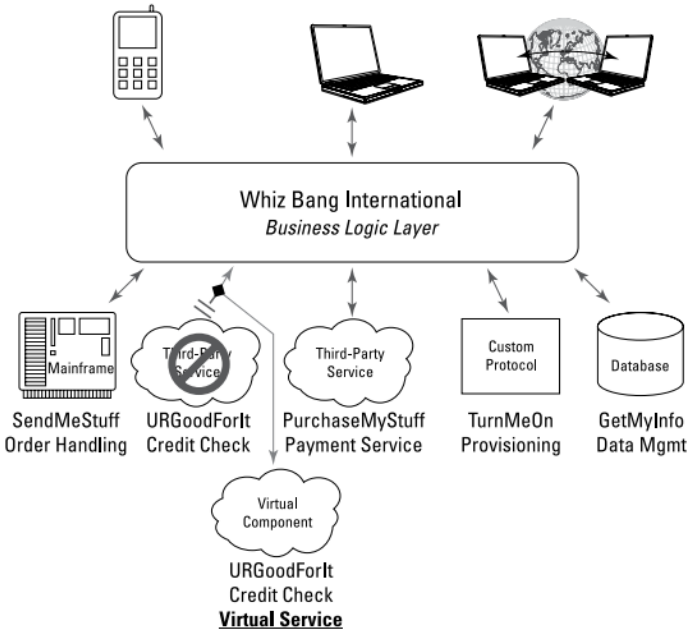


Figure 3. Service virtualization makes unavailable available. [18, p. 6]

This approach should be done by using a separate profile in the application properties of the microservices. Within this profile, API queries for third-party services have to be redirected to the test environment. This environment should handle the queries and return dynamic or static answers depending on the requested action. In case the action result is mutable, the response should reflect these mutations (e.g. balance change after a successful bank transaction). If the action result is not necessary or does not affect the logic of the microservice, the response may be static and have successful HTTP status (e.g. entity creation response with a static ID that is not used in the business logic).

3.1.4 Requirements for virtualization of banking services

The solution should rely on the Open Banking standard for PSD2 applications. The Standard is designed to assist any European account providers in meeting their PSD2 and RTS requirements as well as supporting their application for an exemption from the contingency mechanism. This market-enabling Standard is built in an optional modular format to most effectively meet consumer and market needs. [19]

As the directive 2015/2366/EC [11] does not specifically describe the technical details about providing banking services, any European bank has a right to introduce their own implementation of the directive. However, Open Banking has, to some extent, become an accepted standard, building the logic based on which will allow integrating with most of the European banks.

3.2 Choice of a framework and building tool

There are three popular web application frameworks that support REST API: Spring Framework, ASP.NET and Node.js. The frameworks provide similar functionality, use OOP programming languages and rely on layered architecture patterns. The main idea is to select the framework that will be easy to work with as other developers of the company will also participate in the development and the future improvements of this solution.

3.2.1 ASP.NET and NuGet

ASP.NET is a web application framework marketed by Microsoft that programmers can use to build dynamic web sites, web applications and XML web services. This is a free framework that is used in nearly 3% [20] of web applications on the entire web. ASP.NET

is supported by a lot of contributors and has plenty of libraries in the NuGet package manager [21].

This framework is dedicated to web applications with a huge user base. It can handle many web queries simultaneously and can be easily scaled if the number of users grows. Big projects with solid architecture and code structure are often based on this framework. [22]

As the solution that is proposed in this thesis is a small and independent application to be used locally, the vast functionality of ASP.NET will remain unused. This framework also falls outside of the microservice architecture used in the company as there are no ASP.NET web applications, which in turn will require additional knowledge about it among other developers. The main programming language of this framework, C#, would have become the third one used by backend developers. It will lead to inconsistency in the code of different services. The NuGet package manager contains a vast variety of libraries but the choice is less dedicated to fintech and cryptocurrencies. For instance, there is no such library that unifies the most popular cryptocurrency exchanges or banking providers under one interface.

3.2.2 Spring framework and Gradle

Spring framework provides comprehensive infrastructure support for developing Java applications. It has a Dependency injection feature and supports other modules and extensions such as Spring JDBC and Spring Boot [23]. With Spring Boot the test environment can be set up to use Gradle dependency manager and have a separate profile to work in test mode. Spring framework is not that popular as ASP.NET [24], but more open source libraries are written on Java programming language. *xChange* [25] and *Ibanity* [26] libraries unify most popular crypto exchanges and the European banks under one interface which in turn makes the development process faster. The openness of Java libraries is one of the reasons why most of the microservices in the Polybius Tech OÜ company use Spring framework for the application in the rapidly growing fintech industry.

Usage of Spring framework for implementing the test environment will allow utilizing the dependencies of other microservices of the company (e.g. database entities, base classes and interfaces). It will also be easy to update as other developers will be already

familiar with this technology. Building a solution based on already used infrastructure will take less time as well.

3.2.3 Node.js and NPM

Node.js is one of the recent technologies for building backend services. It uses either JavaScript or TypeScript, uses a non-blocking I/O model, making it efficient and lightweight [27]. The Express.js extension allows implementing a small web service that uses REST API. There is a vast majority of extensions and libraries available across NPM as a lot of developers contribute into modern libraries for the fintech industry.

As Node.js is lightweight and suitable for small projects, it could be used for the proposed solution. However, the base classes and interfaces need to be rewritten. JavaScript and TypeScript are not primary programming languages in the Polybius Tech OÜ company, therefore it will create inconsistency in the code.

3.2.4 Summary of frameworks and building tools

The selected framework should correspond to the current architecture and be capable of improvements in a short period of time. The frameworks potentially suitable for the solution are compared in Table 4.

Table 4. Backend frameworks comparison.

Framework	**Author's experience**	**Share in the project architecture**	**Learning complexity**
ASP.NET	Good	-	Middle [28]
Spring framework	Very good	High	High [29]
Node.js	Middle	Low	Low [30]

Referring the provided analysis, the best choice of framework is Spring framework as it can be easily integrated into the current architecture, is known by the author and other developers of the company and has a variety of libraries dedicated for the fintech industry.

3.3 Choice of database

There are different types of databases such as centralized, distributed, graph, relational, Non-SQL and others. For projects where there are lots of unstructured data, graph and

Non-SQL types of databases may be applied for increasing the performance of search and insertion. Another example, MongoDb, is a document-oriented type of databases that allows storing as many JSON, BSON or XML formatted documents in one record as possible. [31]

However, the solution neither has to deal with big data nor store unstructured data. Relational databases are popular among developers, easy to build and use. This type of database will be used in the solution. Relational databases require an ERD-diagram that reflects the structure of data stored and used in the solution.

Oracle database is mostly dedicated to huge enterprises where speeds and security are the main priority [32]. The prices for the database are high and therefore it is not rational to use it in the solution.

MySQL is an open-source relational database which provides free plans for non-commercial use. It is free and widely used. Although price list for enterprises is applied with ranges of 2000-10000$. [33]

PostgreSQL is a free and open-source solution for web applications. It was one of the first database management systems to be developed, and it allows users to manage both structured and unstructured JSON data [32]. It can handle large data structures and at the same time be useful for small applications as well. PostgreSQL is used in every microservice of the Polybius Tech OÜ company.

SQLite is a compact serverless solution for small applications. It behaves as a plain relational database, supports SQL language, and can store a large amount of data in a file on a disk drive. This database is free to use and easy to build in into nearly any system.

Summing up, PostgreSQL and SQLite are both suitable for the solution. SQLite would have been given an advantage if the test environment was a finalized program that is run only locally. The solution will be improved in the future, allowing testing of the fintech application not only locally but in a pre-production environment. It will require a server-side database that can be based on PostgreSQL. Furthermore, this database technology is already used across all the microservices in the company.

3.4 Design of the solution

The chapter covers the patterns used for the development of the solution. The application architecture is based on the division of application layers. The database is used as data storage and created in accordance with the ERD scheme.

3.4.1 Application architecture

The solution uses three application layers: presentation, business and data layer. This approach is described in the book 'Learn Microservices with Spring Boot: A Practical Approach to RESTful Services using RabbitMQ, Eureka, Ribbon, Zuul and Cucumber'. The main application layers proposed in the book can be seen in Figure 4.

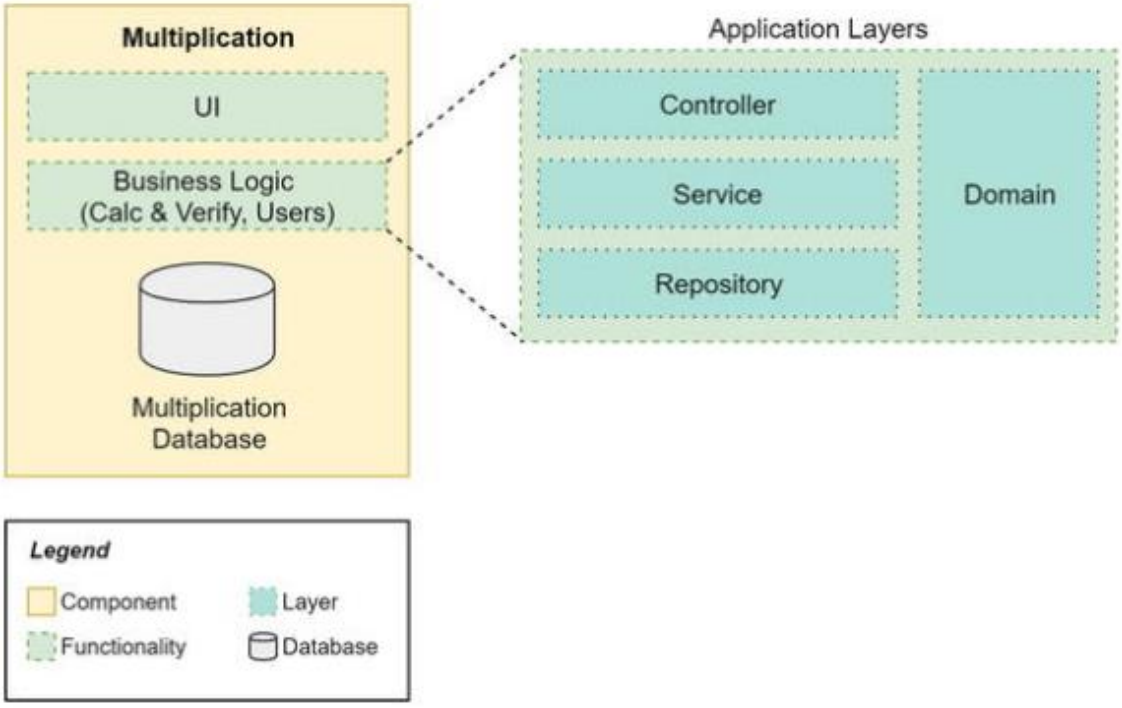


Figure 4. The application's architecture pattern. [34, p. 25]

The architecture pattern consists of 3 layers:

- Presentation layer is responsible for REST API and mapping business objects to Data-Transfer-Objects. In Spring Boot it is represented with *@Controller* annotated classes. [34, p. 24]

- Business layer contains the actual logic of the application. It uses domain entities and services. The services are represented with *@Service* annotation. [34, p. 24]

- Data layer is responsible for persisting entities in a database. It translates domain objects to the database entities and performs actions on the database side. Its classes are represented with *@Repository* annotation. [34, p. 24]

3.4.2 Database ERD scheme

The database scheme describes only the structure and entities required for the test environment. Other microservices used by the fintech application are based on different database structures and fall out of the scope. The database should store information that is needed to emulate Bitcoin and Ethereum network wallet balances, blockchain transactions, banking functionality, and liquidity provider's behaviour. The ERD scheme is presented in Figure 5.

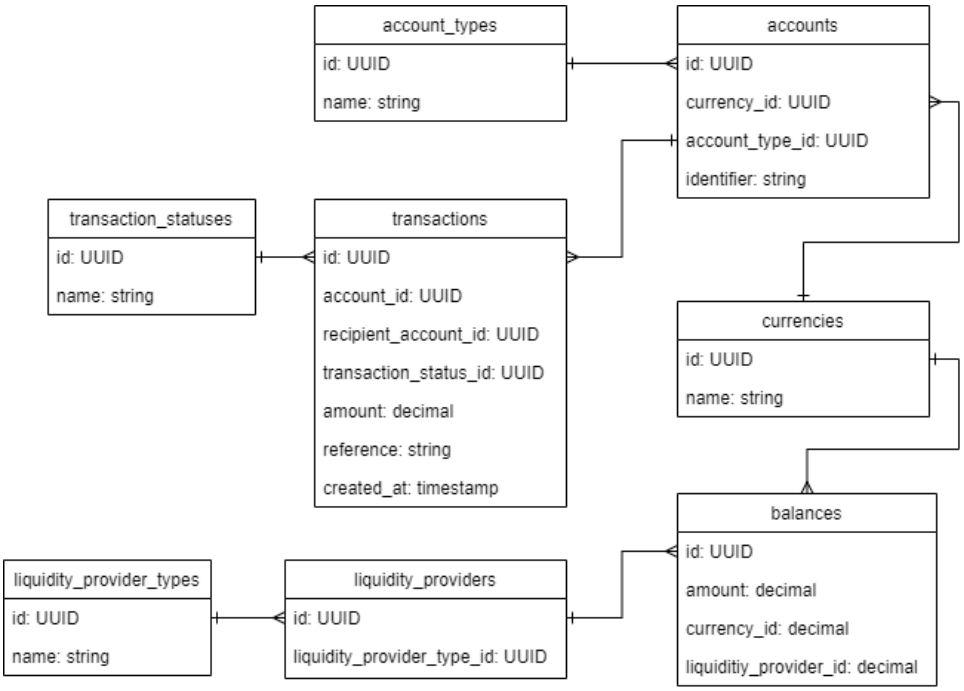


Figure 5. ERD scheme.

The ERD scheme consists of 8 entities. Bank and cryptocurrency transactions are unified under one interface by having different account types. The solution will have different types of liquidity providers with their balances. The entity of liquidity provider does not have properties, however, it was decided to include it in the scheme in order to

provide a possibility to improve the solution in the future releases. Trading will be emulated by changing the balances of a specified liquidity provider.

3.5 Analysis summary

The analysis has covered the requirements for the test environment and frameworks that can be used for implementing it. It has also specified the database that will be used for the implementation of the solution.

Spring framework has been chosen as the main framework of the solution. A common practice of building Spring applications has been presented in this analysis. The analysis has brought out the ERD scheme which the chosen PostgreSQL database will be based on. The application is ready for further improvements as it is based on modern technologies that the developers of the Polybius Tech OÜ company are familiar with.

4 Implementation

The solution implementation is divided into three parts: development of the service, configuring the microservices and testing. The solution does not have UI except for the auto-generated Swagger page.

4.1 Preparation

The service is based on Spring Framework architecture and uses Java as the main programming language. The solution consists of two main blocks: data generator and third-party services virtualizer. It uses PostgreSQL database for storing entities that are necessary for the virtualizer to emulate the behaviour of third-party services (e.g. by storing cryptocurrency transactions).

4.1.1 Creation of the Spring Boot project

The official Spring framework page provides a tool Spring Initializr for creating a base Spring Boot project (see Figure 6). Spring Boot is an extension of the Spring framework which eliminated the boilerplate configurations required for setting up a Spring application [23].

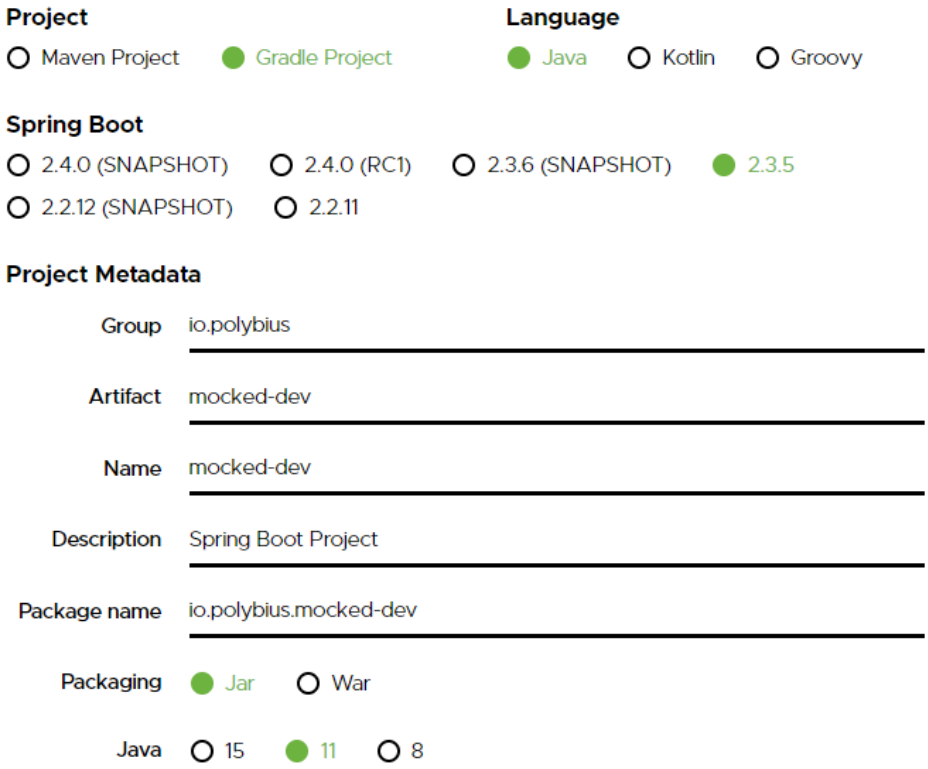


Figure 6. Initial configuration of the solution.

Java 11 was chosen to match the version with other projects of the company. The group of artifacts starts with *io.polybius* as this solution belongs to the group of applications developed by Polybius Tech OÜ.

4.1.2 Gradle configuration

The configuration file *build.gradle* stored in the root directory contains information about plugins, repositories and dependencies used in the project. The configuration file is generated by default, however, more dependencies are added to the *dependencies* section (see Figure 7).

```
dependencies {
   implementation 'org.springframework.boot:spring-boot-starter-web'
   implementation 'org.springframework.boot:spring-boot-starter-
webflux'
   implementation 'org.springframework.boot:spring-boot-starter-jdbc'
   implementation 'org.springframework.boot:spring-boot-starter-jdbc'
   implementation 'org.springframework.boot:spring-boot-starter-
data-jpa'
   implementation 'io.springfox:springfox-swagger2'
   implementation 'io.springfox:springfox-spring-web'
   implementation 'io.springfox:springfox-swagger-ui'
   implementation 'org.postgresql:postgresql:42.2.5'
   implementation 'org.liquibase:liquibase-core:3.8.8'

   testImplementation('org.springframework.boot:spring-boot-
starter-test') {
      exclude group: 'org.junit.vintage', module: 'junit-
vintage-engine'
   }
}
```

Figure 7. Dependencies section of the Gradle configuration.

The dependencies will allow handling API requests, performing prepared SQL statements and creating database migrations. Tests are handled by a default Spring testing library.

4.1.3 Project structure

By default, Spring application projects start with 2 folders: project folder and resources. The inner folders may vary depending on the project needs. The chosen approach containing 3 application layers should be reflected in the structure as well. The structure of the solution can be seen in Figure 8.

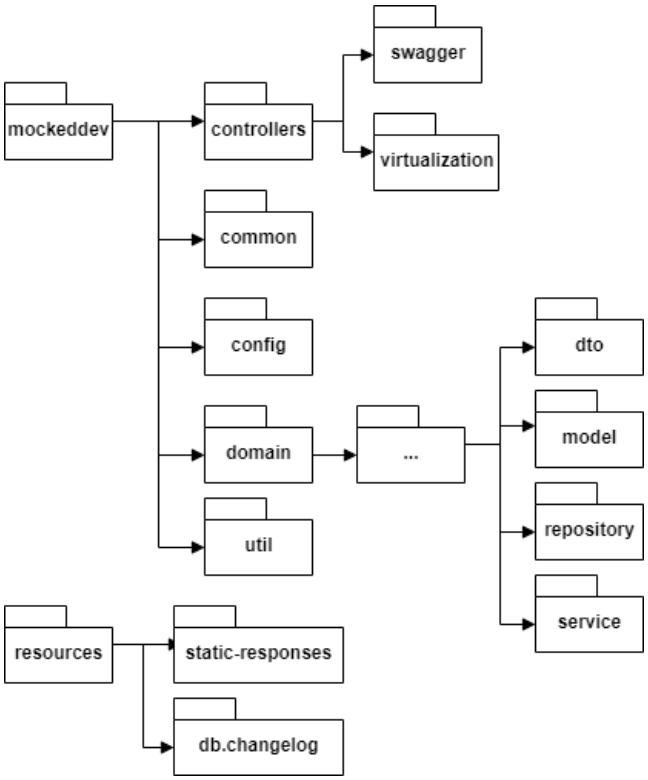


Figure 8. Project structure.

Resources of this solution contain a database migration file, application properties for different Spring profiles and static responses in JSON format. The static response files are used for virtualized services' API queries, the responses of which do not change (e.g. the ID of trade is static as it is not used in the business logic).

The *controllers* package consists of two types of controllers. Swagger controllers are used for data generation via the user interface. Virtualization controllers are used for accepting third-party requests, handling them and responding with dynamic or static answers.

Common package contains base classes, interfaces and converters that are used in the business logic.

Spring annotated *@Configuration* files are stored in the *config* package. These files introduce Spring *beans* and *values*. In Spring, a *bean* is an object that is instantiated, assembled, and managed by a Spring IoC container [35]. Whereas *@Value* annotated object can be instantiated from the application configuration file [35].

Util package consists of helper classes that simplify code.

Each domain in the *domain* package contains Data-Transfer objects (DTO), model, repository and service packages:

1. DTO – In this solution: classes that are mapped from the request body or to response body.

2. Model – represent database entities.

3. Repository – classes that are responsible for data access. Annotated with *@Repository*.

4. Service – classes that are responsible for business logic. Annotated with *@Service*.

4.2 Data generation

The following chapters will cover setting up data sources, insertion of test data and UI for the simplification of the process.

4.2.1 Setting up data sources

Coming from the requirements, the solution should be able to insert database entities to the databases used by other microservices of the company. It is done by using separate data sources in the *application.properties* file. An example of overriding the data source for the identity microservice can be seen in Figure 9.

```
databases.identity.datasource.dbName=identity
databases.identity.datasource.jdbcUrl=${postgres.basePath}/${databases
.identity.datasource.dbName}
databases.identity.datasource.driverClassName=org.postgresql.Driver
databases.identity.datasource.username=${postgres.user}
databases.identity.datasource.password=${postgres.password}
```

Figure 9. Identity microservice data source configuration example.

The configuration file contains *beans* of the data source. It can be used by the base repository of each microservice to insert data into a specified database. The example of data source instantiation is shown in Figure 10.

```
@Configuration
public class DatasourcesConfig {
  @Bean(name = "identityDb")
  @ConfigurationProperties(prefix="databases.identity.datasource")
  public DataSource identity() {
    return DataSourceBuilder.create().build();
  }
}
```

Figure 10. Example of Spring configuration file with data source *bean* of identity microservice.

As the fintech application uses different microservices, the prefix is used to distinguish the properties of the specified microservice. The test environment connects to all databases of the microservices and manipulates their data. To avoid conflicts with the live environment data, a separate schema is built on the actual database structure.

4.2.2 Insertion of test data

Database entities of each microservice are moved to private packages and downloaded via Gradle from the private repository. Base repository class can be inherited only for those entities which implement the interface. The base repository supports insertion, update, deletion and querying for the database entity specified in the generic clause.

Each database entity has its own repository which extends the base repository. This repository is used in services to perform manipulations with data. The simplest example of data generation can be the following:

1) An API endpoint accepts parameters of a test user to be inserted into the database.
2) The API controller transfers the parameters to the service. The parameters can be either registration date, verification status etc.
3) The service creates a database user entity with the specified fields and inserts it using the repository of the microservice the entity belongs to.

4.2.3 Swagger configuration

The solution uses Swagger to simplify the communication between a developer and the solution. It is configured in a separate class by using *@Configuration* and *@EnableSwagger2* annotations (see Figure 11).

```
@Configuration
@EnableSwagger2
public class SwaggerConfig {
  @Bean
  public Docket api(TypeResolver resolver) {
    Pattern publicApiPattern = Pattern.compile("/dev-api/.*");
    return new Docket(DocumentationType.SWAGGER_2)
        .select()
        .paths(path -> publicApiPattern.matcher(path).matches())
        .build()
  }
```

Figure 11. Swagger configuration.

This approach will allow generating test data with the user interface. To distinguish data generation endpoints (see **Appendix 6**) from service virtualization endpoints, the pattern is used in the Swagger configuration

4.3 Virtualization of third-party services

The following chapters will describe the emulation of the third-party services within the scope of the thesis. The implementations are described in the text, however, the examples of source code can be found in the appendices as well.

4.3.1 Emulation of cryptocurrency address creation

The Bitcoin and Ethereum network addresses in the test environment are saved as a random sequence of numbers and letters. These addresses may not be valid for the actual blockchain networks, however, they are unique and usable in the test environment. A Bitcoin address contains 34 symbols (example: **3LoJFcGiBgCzy235poxmq8uZGFGSK3ZbJN** [36]). An Ethereum address starts with *0x* and the remaining part contains 40 symbols (example: **0x501906Ce564be7bA80Eb55A29EE31ECfaE41b6f2** [36]). The addresses can be generated using a simple random string generator (see Figure 12) and saved to the database.

```
public String generateString(int length) {
  String symbols =
"ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789abcdefghijklmnopqrstuvxyz";
  StringBuilder stringBuilder = new StringBuilder();
  IntStream.range(0, length).forEach(s ->
stringBuilder.append(symbols.charAt(randomService.nextInt(symbols.l
ength() - 1))));
  return stringBuilder.toString();
}
```

Figure 12. Method for generation of a random string with a specified length.

The method takes a random character from the permitted character sequence and appends it to the string builder. This action is repeated the number of times specified in the argument of the method.

4.3.2 Emulation of cryptocurrency blockchain transactions

When a cryptocurrency transaction is created in the test environment, the random transaction identifier is generated with the above method (see Figure 12). In the live environment, blockchain transactions are confirmed during a certain period. In the test environment, it can be emulated using Spring *@Scheduled* annotation. The delay for the endpoint is set to 10 seconds to reflect blockchain confirmation procedures. When the scheduled method is called, it updates the statuses of all the current cryptocurrency transactions in the database. The current database entity structure allows making transfers to another cryptocurrency address within the system. It can be used for transferring crypto assets to different test users.

4.3.3 Emulation of cryptocurrency exchange rates

The emulation of currency exchange rates is described in the book “Spring Boot Messaging”. A similar approach is used in the solution for streaming dynamic cryptocurrency rates for Bitcoin and Ethereum. It was decided to take the base price of Bitcoin as 10000 EUR and Ethereum 200 EUR. The solution uses Reactor from the Spring Framework to handle streaming API queries. Reactor is a fully mature and non-blocking reactive programming foundation for the JVM that implements the Reactive Extension specifications and Java 8 functional APIs [37, p. 170].

The implementation of the query can be found in **Appendix 2**.

4.3.4 Emulation of banking services

Coming from the requirements for the solution, banking services should be emulated in accordance with the Open Banking standard [38]. However, it was decided to simplify requests as some of the parameters are not required in the test environment (see endpoints of the banking service in **Appendix 6**). Furthermore, any client is considered authenticated and Euro is chosen as the default currency.

A bank account balance is calculated by summing up all incoming transactions and subtracting all outcoming transactions. It is implemented in SQL and done on the database side (see Figure 13).

```
select sum(case when recipient_account_id = :account_id then
amount else -amount end) as balance from transactions t join
transaction_statuses ts on t.transaction_status_id = ts.id where
ts.name = 'DONE'
```

Figure 13. SQL query for retrieving the bank account balance.

Payments are initiated by inserting a transaction entity with pending status to the database. The same approach of changing transaction status is used as with the cryptocurrency blockchain transactions. *@Scheduled* annotated method changes the status of a transaction. After the transaction is considered fulfilled, the account balance will change. A test user can see the changes by requesting balances or transactions via corresponding endpoints.

4.3.5 Emulation of trading algorithm and liquidity provider

The application uses a third-party algorithm that provides information about recommended trades to be performed. When the response is received, the orders are sent to the liquidity provider that buys or sells the recommended assets. As a result, a user's portfolio is changed. The communication between the services can be seen in Figure 14.

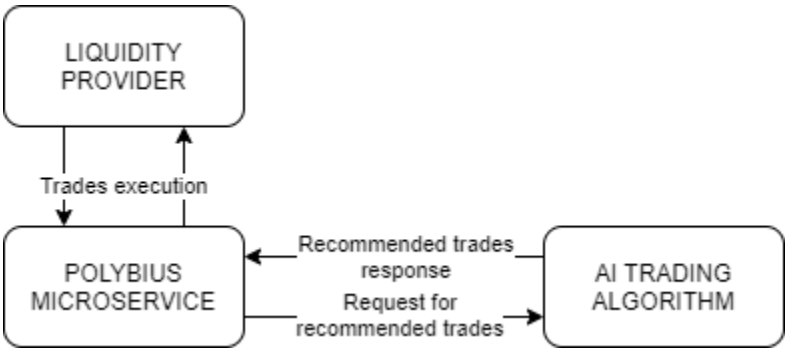


Figure 14. Communication of the company's microservice with third-party services.

In the test environment, the trading algorithm is simplified in a way that it responds with only one recommendation per request. It chooses a random amount of Bitcoin or Ethereum to be sold or bought. As cryptocurrency rates are constantly changed, this will imitate portfolio changes.

The liquidity provider accepts trade requests and changes the balance of it in the database. The exchange rate is taken from the rating service described above. The response is not used in the business logic, however, the library for communication with the liquidity provider requires a response. In the test environment, it was decided to use a static response. The liquidity provider uses *x-www-form-urlencoded* content type. In order to get the parameters, it was decided to use *ServerWebExchange* interface of the Spring Web-Flux library.

The code for interaction with the liquidity providers can be found in **Appendix 3**.

4.4 Testing

With the *@SpringBootTest* annotation, Spring Boot provides a convenient way to start up an application context to be used in a test [39]. It allows overriding Spring *beans*, accepting test configurations and connecting to a test database. Within the thesis, a separate database for tests is used to avoid mixing of data. The database source is configured in the *application-integration-test.properties* file and the base test class is annotated with *@ActiveProfiles("integration-test")* annotation.

The data generator provides only insertion functionality and can be tested trivially. Every endpoint of the data generator is called with specific GET parameters. After that, a

database request is made to get the actual data inserted to the database. All the fields are compared to the ones that were initially provided.

Crypto address generation is made with randomness which will lead to unpredictable test results. However, it can be solved by replacing the service with function for generating random value to the mocked *bean* analogue. In Spring framework there is a *@MockBean* annotation that creates a mocked instance of the service that can be configured to return specified values when a method of the service is called. The example of the test can be found in **Appendix 4.**

Cryptocurrency exchange rates are provided with the asynchronous method which is hard to test. Nevertheless, Reactor Test library can be used for testing the reactive streams. Its component, StepVerifier, provides a declarative way of creating verifiable steps for async publisher sequence by expressing expectations about the set of events that will eventually happen upon subscription [40]. The test also requires *@MockBean* annotation for the service that returns random numbers. The example of the test is provided in **Appendix 4**.

Third-party services used in the solution can be mocked and set up to respond with pre-defined answers. This will guarantee that the test result is always the same. For instance, when a third-party is asked to provide a recommended trade, a static response is returned. Then, the response is redirected to the liquidity provider side, that in turn handles it and responds with the pre-defined answer. In the last part of the test, all parameters sent to the third parties are verified.

5 Assessment of the created solution

The solution has been made in accordance with all the requirements. It provides a full test environment for the fintech application developed by Polybius Tech OÜ and can be used by the developers and testers inside the company. The most valuable functions of it simplify the process of development, emulation of services and test data generation. This project has become a very helpful solution for saving time and money invested in the testing.

5.1 Comparison with the testing process before and after

With the creation of the test environment, all resources invested in testing were incredibly optimized. The current approach drastically decreases the time of development and testing of the application (see Table 5). The remaining resources can be directed to improving the quality of the fintech application, code refactoring and implementation of new features.

Table 5. The current expenses after introducing the test environment.

Action	**New approach**	**Time expenses**	**Money expenses**	**Notes**
User verification (KYC)	Verification status can be set via Swagger UI	-	-	Does not require real documents
Crypto wallet creation	Created via Swagger UI	-	-	No third-party services needed
Bitcoin transaction	Emulated	10 seconds	-	
Ethereum transaction	Emulated	10 seconds	-	
Initiating a deposit via bank	EUR amount can be set via Swagger UI	-	-	No interaction with real banks
Exchange operation	Emulated	-	-	
Trading operation	Emulated	-		Liquidity provider's behaviour is emulated
Withdrawal	Emulated (same as *Bitcoin* transaction)	10 seconds	-	No fees are paid to third parties

The processes that required time and money expenses are simplified by using the test environment. This solution introduces test money that can be added, transferred and invested into the trading algorithm without the need for using real bank accounts and credit cards. With the current approach, identity verification and transaction confirmations can be shortened to seconds. Moreover, there is no need for actual interaction with third-party services which may take fees.

The example of usage of the test environment can be found in **Appendix 5**.

6 Possible future improvements

The current version of the solution solves the problems of development and testing processes, however, it can have even more features. One of the most helpful features could be deploying the test environment to Google Cloud and serving it on a separate domain. The access could be restricted to the employees of the company only by using VPN.

As the test environment emulates the state of the production environment, it will allow the testers performing more advanced test scenarios. For instance, employees of the company could create test users and transfer test money or crypto assets between them. This improvement will also be useful for those employees who do not participate in the development process and therefore lack computer skills, but still want to test the functionality. For example, Product Manager can test the flow of unreleased features in advance. This environment could be possibly used for presentations of the product as well.

The fintech application is continuously updated having more and more services to be integrated with. This test environment will allow implementing emulations of credit/debit card payment processors, stock trading services and more. Any new service appearing in the fintech world can be integrated into the solution.

7 Summary

The thesis has surveyed the problems that occurred during the development of the fintech application and described the way of testing the application before the solution was found. The problems that significantly made the development process difficult were: time expenses for user verification and cryptocurrency transactions, impossible implementation of banking services due to the lack of license, money expenses on fees to third-party services and bank transactions. The unoptimized way of development caused meaningful drawdown in the quality of the application.

The thesis has introduced a test environment for the fintech application developed by Polybius Tech OÜ that minimized the resources invested in the development and testing of the application. It has been designed in accordance with the scientific resources and the current back-end architecture of the company. The main prerequisite for developing the solution was a lack of suitable third-party solutions that would be free and maintainable. The solution is scalable and can be improved in the future.

The introduced test environment has drastically decreased the time expenses for testing and developing the application. As it was based on the emulation principle, all processes that needed additional fees and time expenses are currently emulated and require no expenses. This solution has become a very useful tool among the employees.

References

[1] F. Sorrentino, “Forbes,” Forbes, 20 11 2015. [Online]. Available: https://www.forbes.com/sites/franksorrentino/2015/11/20/heard-at-the-2015-aba-national-convention/. [Accessed 21 11 2020].

[2] D. K. C. Lee and R. H. Deng, Handbook of blockchain, digital finance, and inclusion, London: Academic Press, 2018.

[3] R. Tretau, WebSphere Application Server: Test Environment Guide, IBM Redbooks, 2002.

[4] J. Hill, Fintech and the Remaking of Financial Institutions, San Diego: Elsevier Science & Technology, 2018.

[5] “Bitcoins.net,” [Online]. Available: http://bitcoins.net/guides/bitcoin-confirmations. [Accessed 21 11 2020].

[6] V. Buterin, “Ethereum Blog,” Ethereum, 14 9 2015. [Online]. Available: https://blog.ethereum.org/2015/09/14/on-slow-and-fast-block-times/. [Accessed 21 11 2020].

[7] “Pricing,” Veriff OÜ, [Online]. Available: https://www.veriff.com/pricing. [Accessed 30 12 2020].

[8] “YCharts,” 21 11 2020. [Online]. Available: https://ycharts.com/indicators/bitcoin_average_transaction_fee. [Accessed 21 11 2020].

[9] “YCharts,” 21 11 2020. [Online]. Available: https://ycharts.com/indicators/ethereum_average_transaction_fee. [Accessed 21 11 2020].

[10] Swedbank, “Hinnakiri,” Swedbank, 21 11 2020. [Online]. Available: https://www.swedbank.ee/private/home/more/pricesrates. [Accessed 21 11 2020].

[11] E. Parliament, “EUR-Lex,” 25 11 2015. [Online]. Available: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=celex%3A32015L2366. [Accessed 21 11 2020].

[12] “PSD2 and strong customer authentication,” Ravelin Insights, [Online]. Available: https://www.ravelin.com/insights/ultimate-guide-psd2-strong-customer-authentication. [Accessed 21 11 2020].

[13] S. Chishti and J. Barberis, The FinTech Book, 2016.

[14] Traffic Parrot Ltd, [Online]. Available: https://trafficparrot.com/. [Accessed 21 11 2020].

[15] MockLab, “Getting Started,” [Online]. Available: https://www.mocklab.io/docs/getting-started/. [Accessed 21 11 2020].

[16] A. Simulator, “Features,” [Online]. Available: https://apisimulator.io/api-simulator-features/. [Accessed 21 11 2020].

[17] B. Byars, Testing Microservices with Mountebank, Manning Publications, 2019.

[18] J. Hurwitz, Service Virtualization For Dummies, 2nd IBM Limited Edition, Hoboken: John Wiley & Sons, Inc., 2017.

[19] “Welcome to the Open Banking Standard,” Open Banking Limited, 2020. [Online]. Available: https://standards.openbanking.org.uk/. [Accessed 22 11 2020].

[20] “Asp .Net VS NodeJs,” Similar Tech, 2020. [Online]. Available: https://www.similartech.com/compare/asp-net-vs-nodejs. [Accessed 22 11 2020].

[21] “An introduction to NuGet,” Microsoft, 24 5 2019. [Online]. Available: https://docs.microsoft.com/en-us/nuget/what-is-nuget. [Accessed 16 12 2020].

[22] M. t. Wierik, “Building Horizontal Scalable Stateful Applications With ASP.NET Core,” 21 9 2020. [Online]. Available: https://medium.com/swlh/building-horizontal-scalable-stateful-applications-with-asp-net-core-1db270d24646. [Accessed 16 12 2020].

[23] Baeldung, “A Comparison Between Spring and Spring Boot,” 10 10 2020. [Online]. Available: https://www.baeldung.com/spring-vs-spring-boot. [Accessed 21 11 2020].

[24] “Asp .Net VS Spring,” SimilarTech, 16 12 2020. [Online]. Available: https://www.similartech.com/compare/asp-net-vs-spring. [Accessed 16 12 2020].

[25] knowm, “xChange,” 2020. [Online]. Available: https://github.com/knowm/XChange. [Accessed 21 11 2020].

[26] “Meet Ibanity,” Isabel Group, 2020. [Online]. Available: https://documentation.ibanity.com/. [Accessed 22 11 2020].

[27] R. Gleason, “Node.js vs. Spring Boot — Which Should You Choose?,” 8 1 2020. [Online]. Available: https://medium.com/better-programming/node-js-vs-spring-boot-which-should-you-choose-2366c2f76587. [Accessed 21 11 2020].

[28] S. B. Meher, “Is .NET easy to learn?,” 13 2 2017. [Online]. Available: https://medium.com/@sambhajimeher/is-net-easy-to-learn-74b13987dc23. [Accessed 16 12 2020].

[29] T. Watson, “Java Spring framework – pros, cons, common mistakes,” Skywell software, 11 1 2019. [Online]. Available: https://skywell.software/blog/java-spring-framework-pros-cons-mistakes/. [Accessed 16 12 2020].

[30] “How long does it take to learn Node.js?,” BetterStack, 8 7 2019. [Online]. Available: https://betterstack.dev/blog/how-long-does-it-take-to-learn-nodejs/. [Accessed 16 12 2020].

[31] “Различные типы баз данных, что вы должны знать,” Best Programmer, [Online]. Available: https://bestprogrammer.ru/baza-dannyh/razlichnye-tipy-baz-dannyh. [Accessed 22 11 2020].

[32] C. Arsenault, “The Pros and Cons of 8 Popular Databases,” Keycdn, 20 4 2017. [Online]. Available: https://www.keycdn.com/blog/popular-databases. [Accessed 22 11 2020].

[33] “MySQL Products,” MySQL, [Online]. Available: https://www.mysql.com/products/. [Accessed 22 11 2020].

[34] M. Macero, Learn Microservices with Spring Boot: A Practical Approach to RESTful Services using RabbitMQ, Eureka, Ribbon, Zuul and Cucumber, Apress, 2017.

[35] “Introduction to the Spring IoC Container and Beans,” Spring, 18 11 2020. [Online]. Available: https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-introduction. [Accessed 23 11 2020].

[36] “What cryptocurrency address formats are used on Bitpanda?,” Bitpanda Support, 23 11 2020. [Online]. Available: https://support.bitpanda.com/hc/en-us/articles/360001657820-What-cryptocurrency-address-formats-are-used-on-Bitpanda. [Accessed 23 11 2020].

[37] F. Gutierrez, Spring Boot Messaging, New Mexico: Apress, 2017.

[38] “Open Banking Read-Write API Profile - v3.1.6,” [Online]. Available: https://openbankinguk.github.io/read-write-api-site3/v3.1.6/profiles/read-write-data-api-profile.html. [Accessed 23 11 2020].

[39] T. Hombergs, “Integration Tests with Spring Boot and @SpringBootTest,” [Online]. Available: https://reflectoring.io/spring-boot-test/. [Accessed 23 11 2020].

[40] A. Nandan, “Testing Reactive Microservice in Spring Boot — Unit Testing,” 3 7 2020. [Online]. Available: https://medium.com/@nandan.abhi10/testing-reactive-microservice-in-spring-boot-unit-testing-fe453887ffa1. [Accessed 23 11 2020].

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

I Maksim Nessin

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis “Creation of a Test Environment for the Fintech Application Developed by Polybius Tech OÜ”, supervised by Nadežda Furs-Nižnikova and Vadim Gerassimov
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

23.11.2020

1 The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 – Cryptocurrency rates endpoint

```
@RestController
public class RateController {
 private final ExchangeService exchangeService;

 public RateController(ExchangeService exchangeService) {
  this.exchangeService = exchangeService;
 }

 @GetMapping(value = "/exchange/rates", produces = TEXT_EVENT_STREAM_VALUE)
 public Flux<String> rates(@RequestParam String currencyPair) {
  return exchangeService.rate(currencyPair);
 }
}
```

```
@Service
public class ExchangeService {
 private final RandomService randomService;
 private Map<String, Double> rates = Map.of("BTC/EUR", 10000D, "ETH/EUR", 200D);

 public ExchangeService(RandomService randomService) {
  this.randomService = randomService;
 }

 public Flux<String> rate(String currencyPair) {
  return Flux.interval(Duration.ofSeconds(1))
          .map(s -> String.valueOf(randomRate(currencyPair)));
 }

 public Double randomRate(String currencyPair) {
  Double factor = (randomService.nextDouble() * 2 - 1) / 1000D + 1;
  return rates.get(currencyPair) * factor;
 }
}
```

Appendix 3 – Interaction with liquidity provider

```
@Configuration
public class LiquidityProviderConfiguration {
  public final String assetDetails;
  public final String exchangeInfo;
  public final String time;
  public final String order;
  public final String subAccountTransfer;

  public LiquidityProviderConfiguration(@Value("classpath:liq/asset-details.json") Resource
assetDetails,
                        @Value("classpath:liq/exchange-info.json") Resource exchangeInfo,
                        @Value("classpath:liq/time.json") Resource time,
                        @Value("classpath:liq/order-response.json") Resource order,
                        @Value("classpath:liq/sub-account-transfer.json") Resource
subAccountTransfer) {
    this.assetDetails = asString(assetDetails);
    this.exchangeInfo = asString(exchangeInfo);
    this.time = asString(time);
    this.order = asString(order);
    this.subAccountTransfer = asString(subAccountTransfer);
  }
}

@RestController
public class LiquidityProviderApiController {
  private final LiquidityProviderService liquidityProviderService;
  private final LiquidityProviderConfiguration liquidityProviderConfiguration;

  public LiquidityProviderApiController(LiquidityProviderService liquidityProviderService,
                        LiquidityProviderConfiguration liquidityProviderConfiguration) {
    this.liquidityProviderService = liquidityProviderService;
    this.liquidityProviderConfiguration = liquidityProviderConfiguration;
  }

  @PostMapping(value = "/liquidity-provider/order")
  public Mono<String> handleOrder(ServerWebExchange exchange) {
    return exchange.getFormData().flatMap(a -> {
      Map<String, String> params = a.toSingleValueMap();
      String symbol = params.get("symbol");
      String baseCurrency = symbol.replace("EUR", "");
      String side = params.get("side");
      BigDecimal amount = new BigDecimal(params.get("quantity"));
      liquidityProviderService.trade(baseCurrency, side, amount);
      return just(liquidityProviderConfiguration.order);
    });
  }
}
```

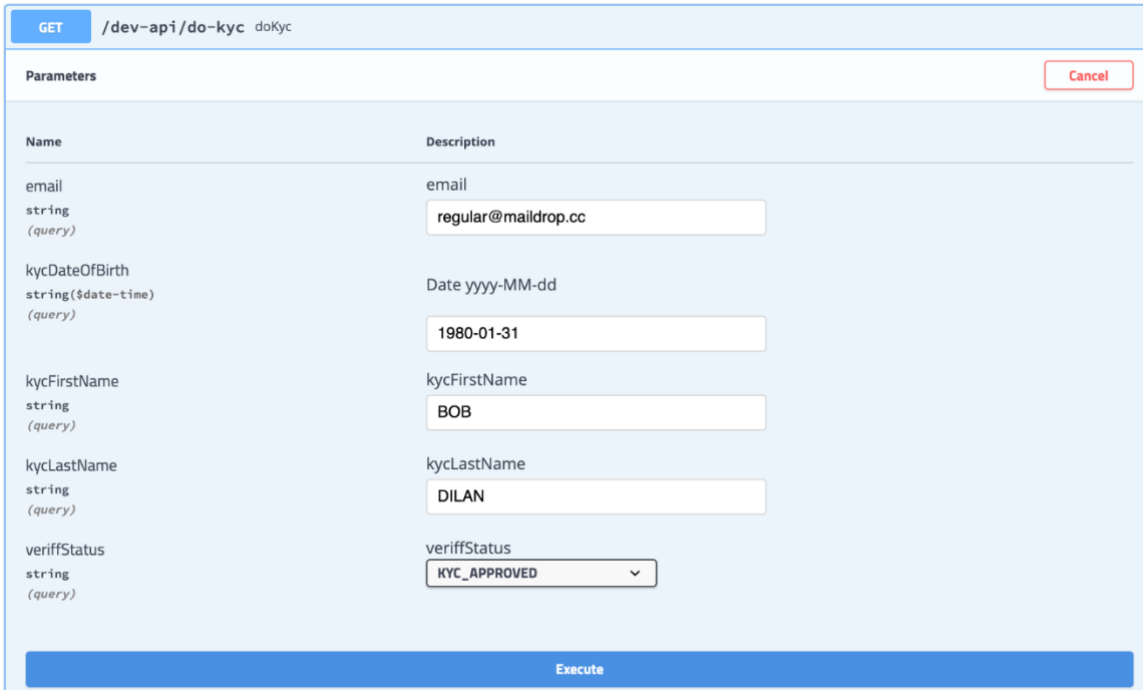
```
@Service
public class LiquidityProviderService {
  private final LiquidityProviderRepository liquidityProviderRepository;
  private final ExchangeService exchangeService;

  public LiquidityProviderService(LiquidityProviderRepository liquidityProviderRepository,
                  ExchangeService exchangeService) {
    this.liquidityProviderRepository = liquidityProviderRepository;
    this.exchangeService = exchangeService;
  }

  public void trade(String baseCurrency, String side, BigDecimal amount) {
    BigDecimal price = new BigDecimal(exchangeService.randomRate(baseCurrency + "/EUR"));
    boolean isBuyTrade = side.equals("BUY");

    BigDecimal baseCurrencyAmountAfterTrade = isBuyTrade ? amount : amount.negate();
    BigDecimal eurAmountAfterTrade = isBuyTrade ? amount.multiply(price).negate() :
amount.multiply(price);

    subAccountTransfer("EUR", eurAmountAfterTrade);
    subAccountTransfer(baseCurrency, baseCurrencyAmountAfterTrade);
  }

  private void subAccountTransfer(String asset, BigDecimal amount) {
    liquidityProviderRepository.provider(Type.TRADING).ifPresent(provider -> {
      BigDecimal currentAccountBalance = provider.balances.getOrDefault(asset, ZERO);
      BigDecimal newBalance = currentAccountBalance.add(amount);
      provider.balances.put(asset, newBalance);

      liquidityProviderRepository.save(provider);
    });
  }
```

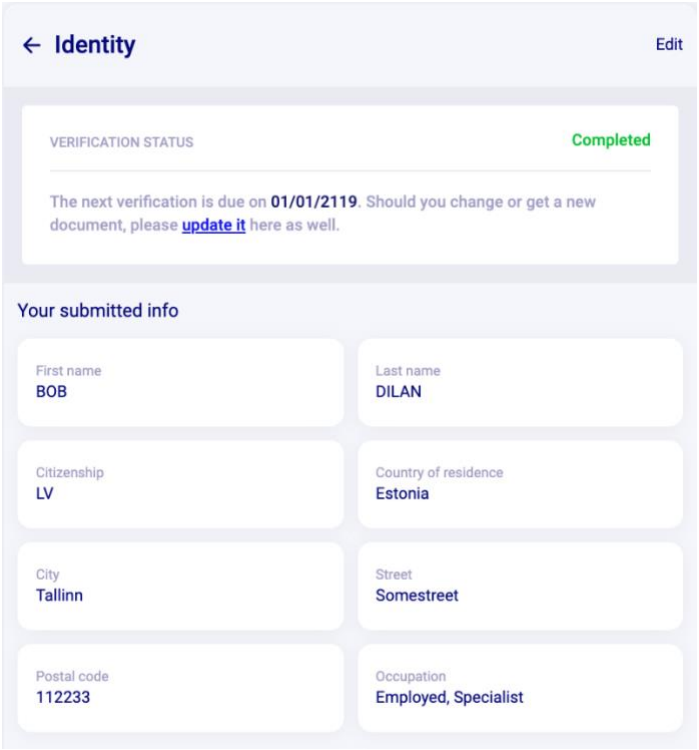
Appendix 4 – Example of tests

```
@SpringBootTest
class MockedDevApplicationTests {
 @MockBean
 private RandomService randomService;
 @Autowired
 private Controller controller;

 @Test
 void rates() {
  when(randomService.nextDouble()).thenReturn(0.432D).thenReturn(0.124D);

  Flux<String> flux = controller.rates("BTC/EUR");

  // factor = (X * 2 - 1) / 1000 + 1;
  // BTC/EUR price is 10000
  // ((0.432 * 2 - 1)/1000 + 1) * 10000 = 9998.64
  StepVerifier.create(flux)
      .expectNext("9998.64")
      .expectNext("9992.48")
      .thenCancel()
      .verify();
 }

 @Test
 void generateAddress() {
  when(randomService.nextInt(anyInt())).thenReturn(5, 9, 3, 10);

  String actual = controller.generateAddress(4);

  // 1 2 3 4 5 6 7 8 9 10
  // A B C D E F G H I J KLMN...
  assertThat(actual).isEqualTo("FJDK");
 }
}
```

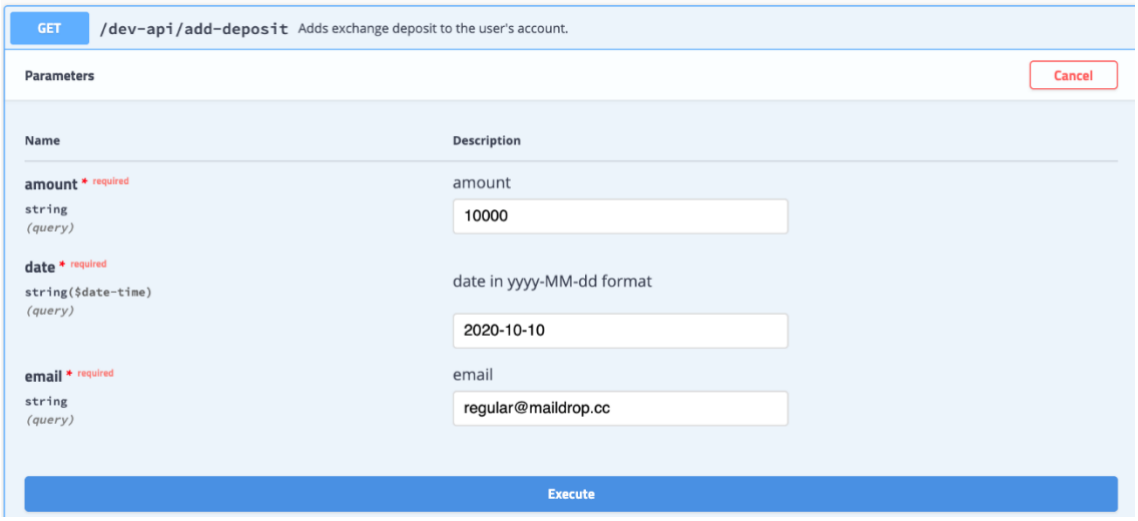
Appendix 5 – Usage of the test environment

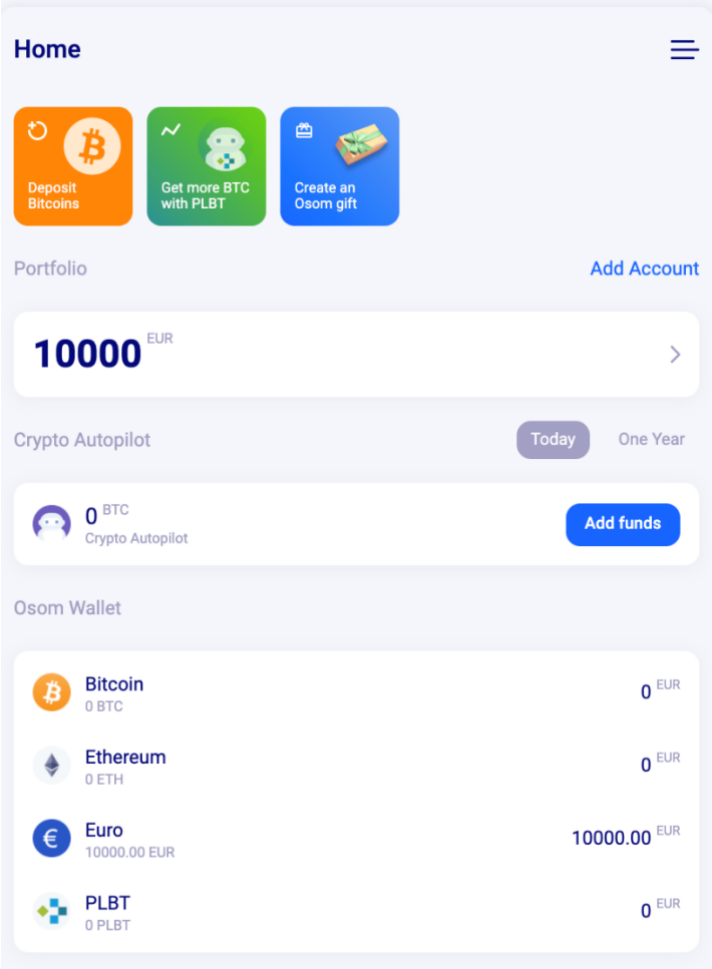
1) Adding verification status





2) Adding test money to the user





Appendix 6 – API endpoints documentation

API endpoints of the data generator

Base URL is `/dev-api` followed by the name of microservice. Optional parameters are written in *italics*.

Request	Parameters	Description
POST /identity/users	authenticationType, email, *referralCode*, *password*	Creates an unverified test user with specified email, authentication type and optional parameters. Referral code is not set by default. Password is set to "password" by default.
POST /identity/users/kycs	email, dateOfBirth, firstName, lastName, verificationStatus, *randomizeName*	Initiates a KYC for the specified user with identity data. Name can be randomized to avoid the system's check for similar names of different users.
POST /wallet/crypto-addresses	email, currencyCode	Creates a fake cryptocurrency address for the specified cryptocurrency and assigns it to the user with the email. The address is randomized and valid only within the test environment.
POST /wallet/crypto-addresses/{address}/balance	address, amount	Adds the amount of crypto asset to the specified crypto address.
POST /wallet/crypto-addresses/transactions	fromAddress, toAddress, amount, status, *transactionHash*	Creates a transaction with the specified sender's and recipient's addresses. Depending on the status the transaction may or may not be confirmed. Transaction hash is randomized if not set.
POST /exchange/deposits	email, amount, date	Adds a deposit to be used for cryptocurrency exchange.
POST /bank/accounts	email, currencyCode, *iban*	Creates a fake bank account with the specified currency and assings it to the user. IBAN is generated if not set.

POST /bank/accounts/{iban}/balance	iban, amount	Adds the amount of money to the specified bank amount.
POST /bank/accounts/transactions	fromIban, toIban, amount, status, *transactionHash*	Creates a fake bank transaction with the specified sender's and recipient's IBANs. Depending on the status the transaction may or may not be confirmed. Transaction hash is randomized if not set.
POST /autopilot/deposits	email, amount, date	Creates a deposit in the *Crypto Autopilot* service. The deposit can be created in the past to emulate historical data and charts in the UI.
POST /autopilot/withdrawals	email, amount, date	Creates a withdrawal in the *Crypto Autopilot* service. The withdrawal can be created in the past to emulate historical data and charts in the UI.
POST /liquidity-provider/history		Randomize a history of crypto asset movements and trades on the liquidity provider side.

API endpoints of the service virtualizer

Request parameters can be presented in JSON or HTTP parameters depending on each endpoint. For simplicity of the table, JSON parameters are put into brackets.

Request	**Request parameters**	**Response body**	**Description**
GET /banking/accounts/{accountId}/balances		[amount, currency]	Returns a list of balances with currencies for the requested bank account.
GET /banking/accounts/{accountId}/transactions		[transactionId, transactionReference, status, amount, currency]	Returns a list of bank transactions for the requested bank account.
POST /domestic-payments	{amount, currency, creditorAccount}	[domesticPaymentId, status, creationDateTime]	Initiates a fake domestic (local) bank payment.

Scheduled: 10 seconds POST /scheduler/update-transaction-statuses			Emulates the process of confirming cryptocurrency and bank transactions. Periodically updates the current transaction statuses in the database of the test environment.
GET /exchange/rates	currencyPair	text-event-stream-data:string	Periodically returns a randomized rate for the specitifed currency pair. The HTTP request is formed within the Server-sent events (SSE) specification.
POST /liquidity-provider/order	symbol, side, quantity	{symbol, orderId, timestamp, status}	Initiates an exchange on the liquidity provider side.
GET /trading-algorithm/request		[symbol, side, quantity]	Responds with randomized cryptocurrencies and recommended amounts to be exchanged. Emulates a real tranding algorithm.