TALLINN UNIVERISTY OF TECHNOLOGY
Faculty of Information Technology
Department of Informatics
Chair of Information Systems

Migrating the Monolith to a Microservices Architecture: the Case of TransferWise

Bachelor Thesis

Author: Erko Risthein
123869IAPB
Supervisor: Raul Liivrand

Tallinn
2015

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

---------- ----------

(*kuupäev*) (*allkiri*)

Annotatsioon

Käesoleva töö eesmärgiks on analüüsida monoliitse rakenduse migreerimisprotsessi mikroteenustel põhinevale arhitektuurile. Töö põhineb TransferWise'i näitel.

Töös käsitletud põhiprobleem seisneb selles, kuidas TransferWise'i kiire kasvutempo juures jätkusuutlikult skaleeruda, ilma produktiivsust kaotamata.

Töö olulisemateks tulemusteks on ühe näidismikroteenuse implementatsioon ja üldistatud metoodika järgmise mikroteenuse monoliidist eraldamiseks.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 46 leheküljel, 5 peatükki, 17 joonist ja 7 tabelit.

Abstract

The aim of this thesis is to analyze the migration process from a monolithic application to a microservices based architecture. It is a case study of TransferWise.

The main problem that is tackled in this paper is how to sustainably scale a tech startup company without compromising productivity.

The main result of this paper is a successfully implemented proof-of-concept microservice and a generalized methodology for extracting microservices from a monolithic application.

The thesis is in English and contains 46 pages of text, 5 chapters, 17 figures and 7 tables.

Glossary of Terms and Abbreviations

ABA	*American Bankers Association*
Android	*Mobile operating system based on Linux*
AngularJS	*Web application framework*
AOP	*Aspect-Oriented Programming*
API	*Application Programming Interface*
BDD	*Behavior Driven Development*
BIC	*Business Identifier Code*
CD	*Continuous Delivery*
Classpath	*Parameter that defines the location of classes*
CPU	*Central Processing Unit*
DI	*Dependency Injection*
Docker	*Deployment automation tool using software containers*
DTO	*Data Transfer Object*
Elasticsearch	*Search server based on Lucene*
Gradle	*Build automation tool*
Grails	*Web framework for the Java/Groovy platform*
Groovy	*Object-oriented programming language for the Java platform*
GUI	*Graphical User Interface*
H2	*In-memory relational database management system written in Java*

Hamcrest	*Library of matcher objects*
Hibernate	*ORM Framework*
HTTP	*Hypertext Transfer Protocol*
IBAN	*International Bank Account Number*
IDE	*Integrated Development Environment*
IoC	*Inversion of Control*
iOS	*Mobile operating system by Apple Inc.*
ISO	*International Organization for Standardization*
Jackson	*JSON processor for Java*
JAR	*Java Archive*
Java	*Object-oriented programming language*
Java Bytecode	*Instruction set for the JVM*
Java EE	*Java Enterprise Edition*
JDK	*Java Development Kit*
JPA	*Java Persistence API*
JSON	*JavaScript Object Notation*
JUnit	*Unit testing framework for Java*
JVM	*Java Virtual Machine*
Kibana	*UI for Elasticsearch*
KPI	*Key Performance Indicator*
Logback	*Logging library for Java*

Logstash	*Tool for managing logs*
Mockito	*Mocking framework for Java*
MVC	*Model-View-Controller*
MVP	*Minimum Viable Product*
MySQL	*Relational Database Management System*
OOP	*Object Oriented Programming*
ORM	*Object-Relational Mapping*
OS	*Operating system*
Quartz	*Job scheduling library*
REST	*Representational State Transfer*
Servlet	*Java program that runs within a web server*
Slf4j	*Simple Logging Facade for Java*
SOLID	*Five basic principles of OOP design: Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion*
Spock	*Testing framework*
Spring Framework	*Application framework and an IoC container*
SQL	*Structured Query Language*
SWIFT	*Society for Worldwide Interbank Financial Telecommunication*
Tomcat	*Web server and servlet container*
TransferWise	*Peer-to-peer money transfer service*

UI	*User Interface*
VM	*Virtual machine*
WAR	*Web application Archive*
XML	*Extensible Markup Language*

Table of Figures

Table of Tables

Table of Contents

1. Introduction

A year ago there were 60 people working at TransferWise. Today, there are over 300. How do you scale a startup company at such a growth rate without sacrificing productivity?

TransferWise has a flat organizational structure, which means that there are no middle managers between the staff and the executives. This essentially means that all the teams have to be self-managing. There is no boss to tell you what to do. The whole organization is built upon autonomous independent teams. [1]

People at TransferWise strongly believe that this is the only way to sustainably scale a business. Other more hierarchical organizational structures tend to encourage blind submission to superiors and repress individual creativity and freedom. This, in turn, means that people work for their bosses, not for the customers.

However, a horizontal organizational structure creates challenges at the same time. How should a company organize its development efforts when it comprises numerous five-to-seven-person teams? Over the past year, it has become evident that it is increasingly hard to arrange the work between the teams when everybody is working on a single monolithic application. Therefore, it became apparent that some architectural refactoring needs to be done.

Currently, TransferWise is moving towards the microservices architecture. Microservices is a software architecture paradigm that constitutes an application of small and independent services communicating with one another through standardized APIs. [2]

1.1 Scope and Goals

The main goals of this thesis are:

1. Analyze the pros and cons of the microservices architecture compared to the monolithic architecture.
2. Extract a single microservice from the monolithic application.
3. Provide a generalized approach to decomposing an application into microservices.

1.2 Methodology

This thesis is an empirical case study of TransferWise. The research design is experimental: the experiment is conducted by implementing a single proof-of-concept microservice and then generalizing the learnings into a replicable process.

1.3 Outline

The thesis consists of 5 chapters:

1. The first chapter defines the goals, scope and the methodology of the study.
2. The second chapter gives a technological background of TransferWise and discusses the main differences between the monolithic architecture and the microservices architecture.
3. The third chapter provides the implementation details and challenges of building the proof-of-concept microservice.
4. The fourth chapter declares a high-level architectural vision for TransferWise.
5. Finally, the fifth chapter outlines the main conclusions of this thesis.

2. Technological Background

Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations.
—M. Conway [3]

2.1 TransferWise Architecture

2.1.1 Technology Stack

The TransferWise codebase is mainly written in the Groovy programming language. Groovy is a dynamic object-oriented programming language that runs on the JVM. Groovy is considered to be an extension of the Java language as most Java code is also syntactically valid Groovy code. Groovy source code is compiled into Java bytecode which means that you can run Groovy code on any Java Virtual Machine, given that the Groovy JAR file is present in the classpath at runtime. [4]

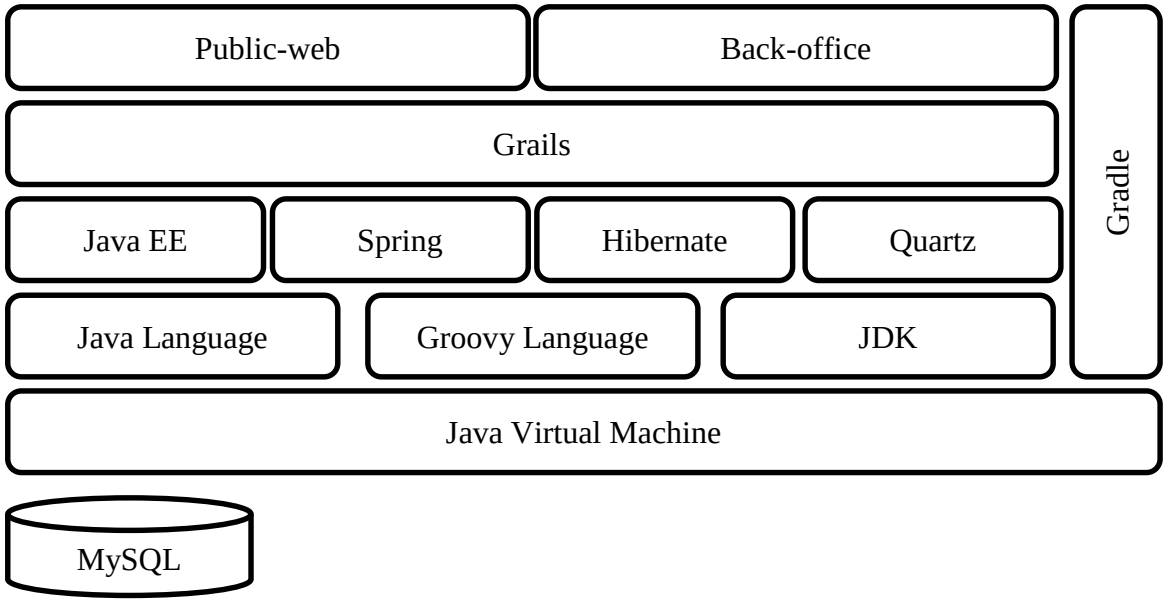

Figure 1 TransferWise Technology Stack

Most of the TransferWise codebase is based on the Grails framework. Grails is a Groovy-based web framework that leverages the best-of-breed Java frameworks like Spring MVC, Spring DI and Hibernate ORM. [5] The Grails framework advocates convention over configuration which in layman terms means that the application auto-wires itself based on naming conventions

instead of using an extensive set of XML configuration. It is based on the Java EE architecture, uses the Gradle build automation tool and is deployed to the Tomcat container. [6] Spock is the default testing and specification framework used in Grails.

2.1.2 Monolithic Architecture

The current software architecture style of TransferWise is largely monolithic. The main characteristics of a monolithic system are [7]:

- written in a single programming language
- single source code repository
- single IDE project
- compiled and packaged into a single runtime application
- high coupling
- low cohesion

Many teams structure their software code by layers. TransferWise is no different.

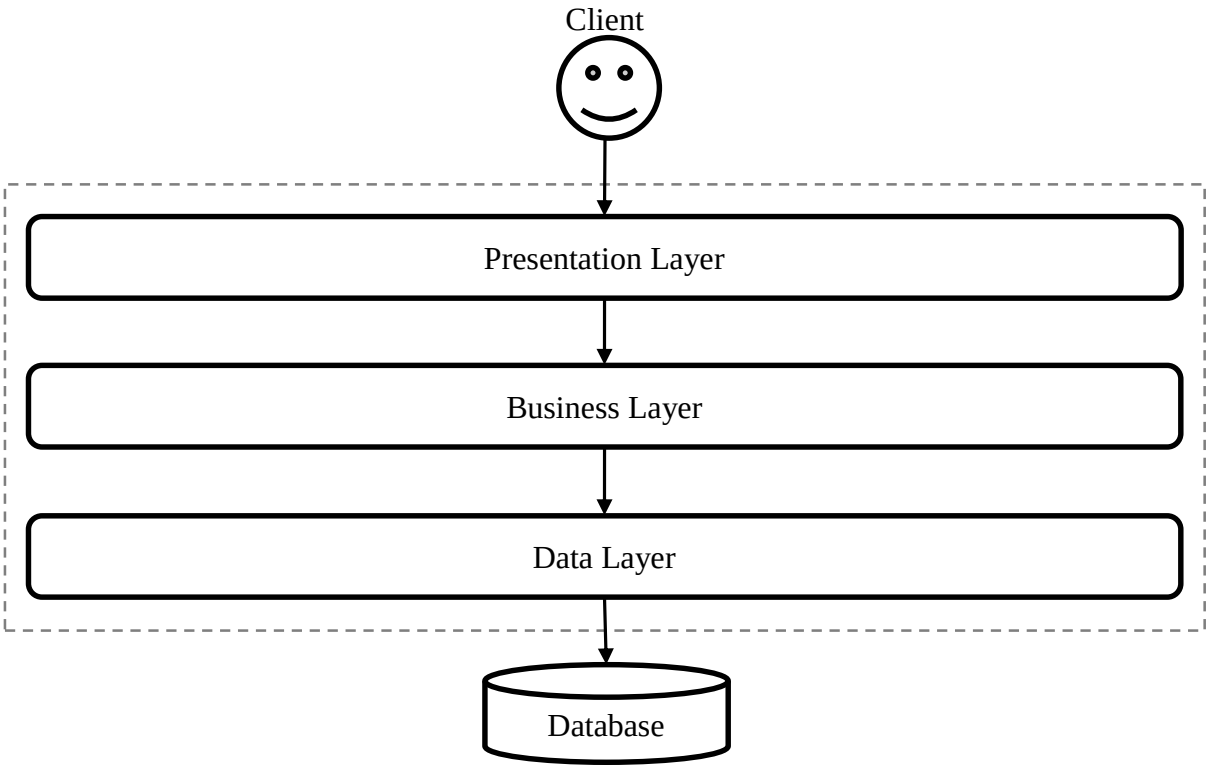

Figure 2 Layered Architecture

2.1.2.1 Presentation Layer

On the top, you have the client, which could either be a physical person using your software or another information system. The client communicates with the presentation layer over the HTTP protocol. The presentation layer contains all the user-oriented logic that manages the user interaction with the system. The presentation layer acts as a bridge between the user and the encapsulated business logic in the business layer. The presentation layer is implemented using the MVC design pattern. As the name suggests, it consists of 3 main parts: the Model, the View and the Controller. The Model represents the underlying data. The View provides a GUI to the user (or an API to an application). The Controller is responsible for interpreting the user input (HTTP requests in this case), validating it, calling the correct Service in the Business Layer, which returns the Model, and finally injecting the Model into the View. [8]

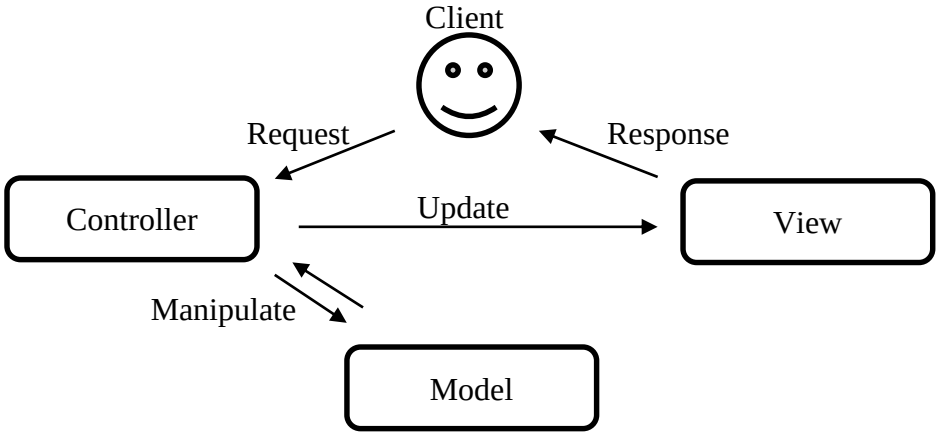

Figure 3 Model-View-Controller Pattern

2.1.2.2 Business Layer

The business layer contains the core functionality of the information system. It mostly consists of business entities and service classes which, in collaboration, define your domain model, your main business workflows and use cases. The service layer exposes a set of public interfaces that are used by the presentation layer and, in turn, communicates with the data layer to query the necessary persistence entities (classes that are mapped to database tables). [9]

2.1.2.3 Data Layer

The data layer consists of persistence entities that represent the underlying relational database tables and the main purpose of this layer is to centralize the data access functionality. This makes the application easier to maintain and configure. Many ORM frameworks like Hibernate

implementing the data access components automatically, so the developers have to write a smaller amount of (error-prone) boilerplate code. [10]

2.2 Monolithic Architecture

Allegedly, the very first version of TransferWise was written by one of the cofounders of the company over the weekend as a minimum viable product (MVP). When working in a fast-pace tech startup context, it is perfectly reasonable to build your application as a single monolith.

2.2.1 Advantages

An application built with a monolithic architecture has a number of advantages over other architectural styles (namely, the microservices architecture): [11]

- Simple to develop – you can import a single project into an IDE and easily run the whole application on your development machine.
- Simple to test – it is straight-forward to run automated functional tests and end-to-end tests on a single monolithic application.
- Simple to deploy – a monolithic application is packaged into a single deployable WAR file which can be easily deployed onto an application container (i.e. Tomcat).
- Simple to scale – you can easily run multiple instances of the same application behind a load balancer to scale the application horizontally.

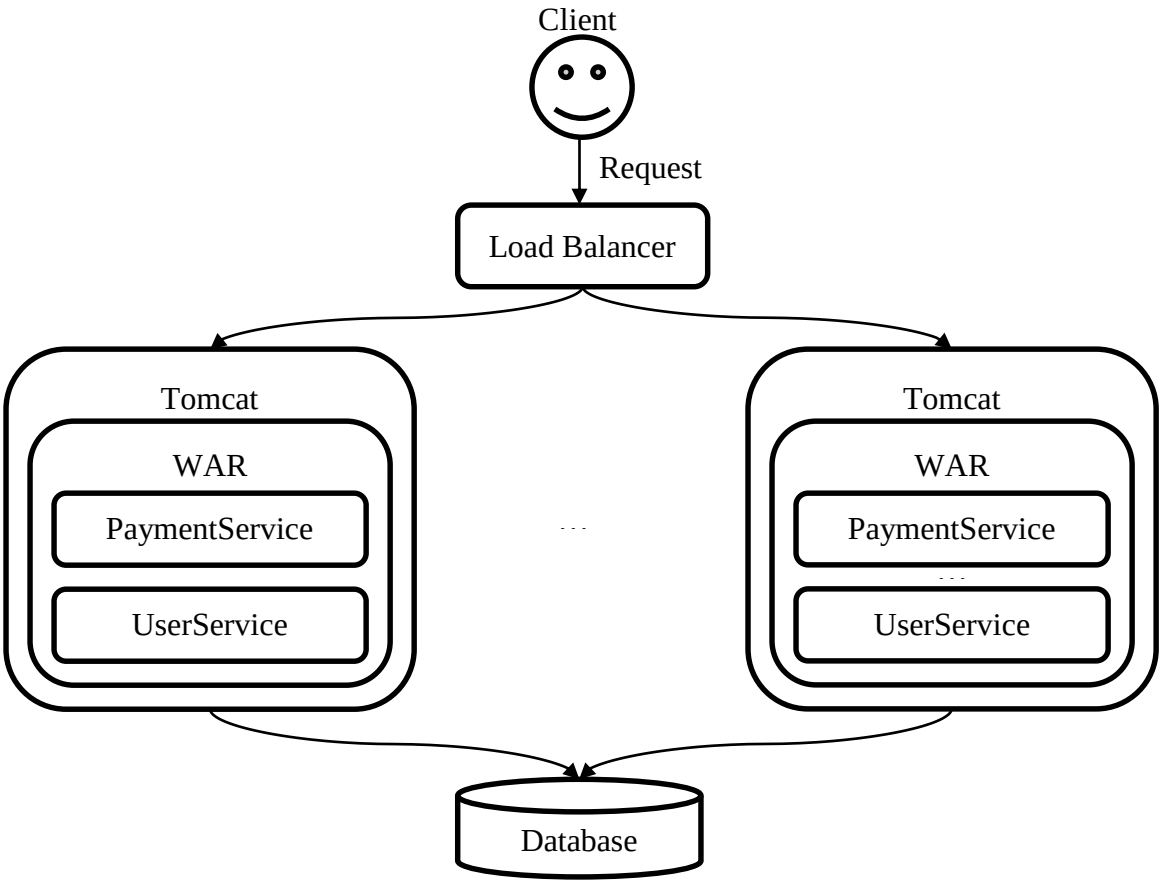

Figure 4 Monolithic Architecture

2.2.2 Downsides

However, as your application grows and your team size becomes larger, it becomes evident that the monolithic architecture has several downsides: [11], [12]

- Synchronous release cycles – a change made in a small subset of the system requires the entire monolith to be rebuilt, retested and redeployed. Continuous deployment is difficult and this discourages frequent and small updates.

- Modularity – over time, with many new developers joining the company, it becomes increasingly hard to keep your codebase modular. This, in turn, makes your code harder to understand, harder to develop and more error-prone, which slows down development.

- Slow IDE – loading the whole monolithic project into your development environment can make it painfully slow.

- Slow startup times – When running a single application in a single web container the startup time gradually slows down as the application grows. This impacts both development and deployment.

- Scaling – Each component of the monolithic system usually has different system requirements. Some may be CPU intensive, some memory intensive. It is impossible to scale individual components independently.

- Independent teams – once the organization grows and more developers join the engineering department, it makes sense to split them into independent teams. Each team can focus on a single functional area (for example, transfer creation or payment processing) and have a well-defined KPI (for example, the conversion rate or payment processing speed). Nevertheless, if all the teams are working on a single codebase, they need to coordinate and plan their development efforts, deployments and architectural decisions.

- Technological agnosticism – a monolithic application is based on a single technology stack. The whole application is written in a single programming language using a single framework and a common set of libraries. Different components of the system cannot be written in different programming languages. A monolithic system is, by definition, not technology agnostic.

2.3 Microservices Architecture

The microservices architecture pattern aims to address the limitations of the monolithic architecture pattern. The application is split into functional services: each service has high cohesion, and is, in most cases, relatively small (hence the name, *micro*service). For example, an application might be composed of services such as payment service, user service, etc. [13]

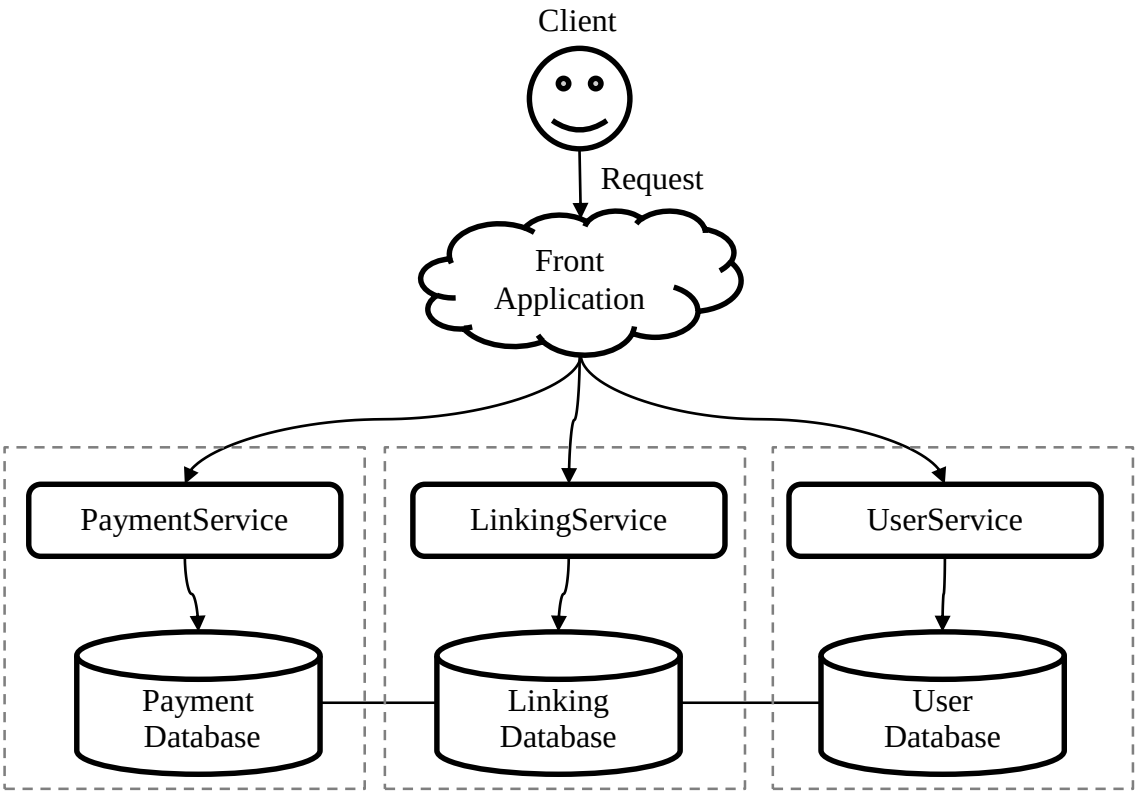

Figure 5 Microservices Architecture

2.3.1 Benefits

There are several benefits to using microservices architecture pattern: [14]

- Independent deployability – easy to frequently deploy new versions of a service. This, in turn, means that teams can develop, deploy and scale each service independently of all the other teams. This is a key factor for having truly autonomous teams at TransferWise.

- Lower complexity – each service is relatively small, has a single responsibility (the "S" in "SOLID" stands for the single responsibility principle), high cohesion and loose coupling. This makes understanding the subsystem easier for new team members and they can become productive much faster.

- Fast IDE – since every microservice is relatively small, it will not slow down the development environment and developers can be more productive.

- Technological freedom – each microservice could theoretically be coded in a different programming language and use a different technology stack. However, it still might make sense to keep the underlying platform of the microservices relatively

homogeneous. This allows to make the hiring process more standardized and makes it easier for developers to switch between teams. For example, TransferWise sticks with the JVM-based languages (Java, Groovy etc.).

- Easy to rewrite – microservices are, by definition, small. For that reason, it is comparatively easy to rewrite them. As long as the interface or the API contract does not change, the development team responsible for that specific microservice has the freedom to change any implementation details. This way, teams are able to independently take advantage of new emerging technologies (frameworks, libraries, etc.).

- Fault isolation – with microservices all boundaries are isolated, which means that the scope of potential problems is also isolated. This reduces the potential for damage and makes the systems easier to maintain.

2.3.2 Drawbacks

Regardless of the numerous benefits microservices have, it is important to understand the drawbacks of the microservices architecture pattern:

- Increased complexity – a distributed system is inherently more complex. Running the full application stack in your development machine is not as straight-forward as a monolithic application.

- Testing – integration testing different versions of microservices to make sure they are compatible with each other and running end-to-end tests is more complex in a distributed microservices-based application.

- Latency – it takes time to communicate through a network socket. It takes time to marshal and unmarshal objects to and from JSON (for example, in case of REST). [15]

- Overhead – each microservice runs in a separate JVM and uses its own application server, which requires extra computation time, memory and bandwidth.

3. Implementation

At TransferWise, there are 2 major monolithic applications. The first one is the public web that can be accessed from transferwise.com and is called *tw-web*. The second one is the back-office application that is used internally by the operations team and by the customer support team. It is called *tw-ninjas*, for historical reasons. As we can see from Figure 6, there are several interdependent submodules that are shared between these applications. Most of them are in the form of Grails plugins that are compiled and packaged into JAR files.

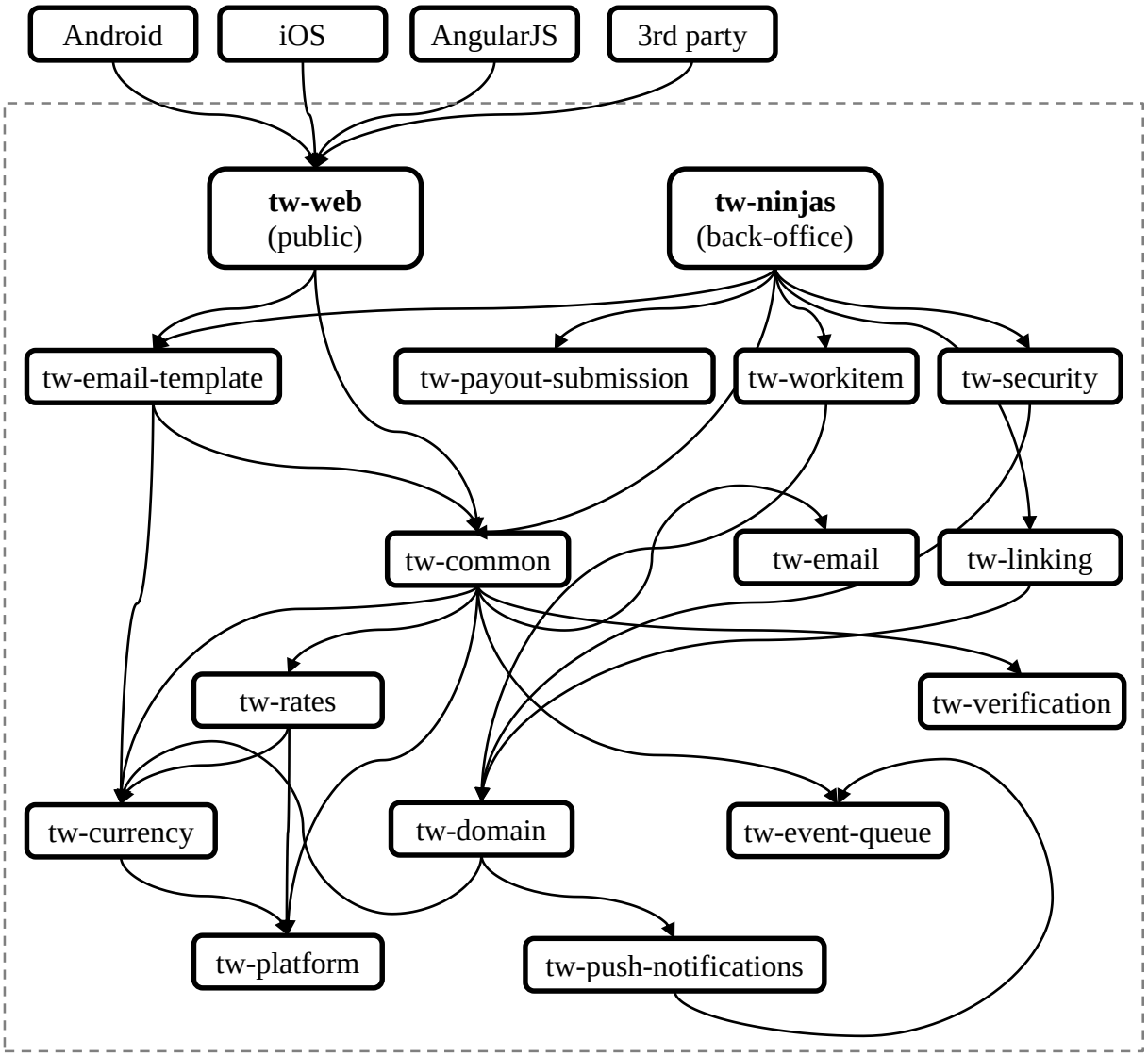

Figure 6 TransferWise Plugin Dependencies

Most of the Grails plugins were originally parts of the monolithic application. Over time, they were extracted as separate plugins and some of them have evolved into microservices. This is

a perfectly normal iterative approach to evolve the software architecture. In that sense, incrementally migrating the monolithic application to independent microservices is an organic part of the software evolution process.

3.1 Building the Microservice

The proof-of-concept microservice that will be built is called the RecipientService. Its main responsibility is to manage Recipients. Recipients represent bank accounts for receiving payments, and are country and currency specific. In other words, a BrazilianLocalRecipient has different fields with different validation rules than a SwissLocalRecipient.

3.1.1 Framework

There is a number of possible frameworks to choose from. Here, in TransferWise, the preferred choice is Spring Boot. Spring Boot is a convention-over-configuration framework that is designed to get you up and running as fast as possible. It takes an opinionated view on the Spring Framework and 3rd party libraries and allows you to create production-ready applications with minimum fuss. It features an embedded application server (Tomcat, by default), so there is no need to deploy WAR files. You can just create a simple Java class with the main method and run it from your favorite IDE. [16]

The simplest possible Spring Boot application looks like this (Java imports have been omitted for brevity):

```
@SpringBootApplication
@RestController
public class RecipientService {

        @RequestMapping("/")
        public String root() {
                return "The RecipientService works!";
        }

        public static void main(String[] args) {
                SpringApplication.run(RecipientService.class, args);
        }
}
```

Figure 7 Spring Boot Sample Application

```
dependencies {
    compile("org.springframework.boot:spring-boot-starter-web:1.2.3.RELEASE")
}
```

Figure 8 Spring Boot Sample Application Web Dependency

When you run the application, it will boot up Tomcat on port 8080 and run the Spring Application. If you visit the website through your browser, you see the message "The RecipientService works!"

When using Spring Boot, there is not a single line of XML configuration, not even web.xml (because of Servlet 3.0+). However, should you need control over what is happening with the application, you can easily override any of the default configuration values.

3.1.2 Libraries

In addition to the Spring MVC Framework, the technology stack includes:

- Spring Security for security
- Spring AOP for crosscutting concerns
- Spring Data JPA for the JPA-based data access layer
- Hibernate as the JPA specification implementation
- H2 in-memory database for integration testing
- MySQL database
- Slf4j and Logback for logging
- Jackson for JSON marshalling
- Tomcat as the application container
- JUnit for unit testing
- Mockito as the mocking framework
- Hamcrest for rule matching in tests

3.2 API Contract

Table 1 POST /api/v1/recipient/create parameters

Parameter Name	Data Type	Required	Description
name	String	yes	The name of the recipient. Either personal or business. Length between 3-255 characters. Only alphanumeric characters, cannot be blank.
type	String	yes	The recipient type. For example: iban, sortCode, aba, swiftCode. Must be a valid type.
currency	String	yes	The currency code of the recipient. For example: GBP, USD, EUR. Must be a supported currency.
email	String	yes	The email address of the recipient. Must be valid.
receiverType	String	no	The receiver type: either PRIVATE or BUSINESS.
addressFirstLine	String	no	The first line of the recipient address. Maximum length 255 characters.
addressPostCode	String	no	The post code of the recipient. Maximum length 32 characters.
addressCity	String	no	The city of the recipient. Maximum length 255 characters.
addressCountryCode	String	no	The ISO3 country-code of the recipient.
addressState	String	no	The state of the recipient. US-specific. Maximum length 2 characters.

Every recipient has also some currency/country specific fields.

Table 2 GET /api/v1/recipient/details parameters

Parameter Name	Data Type	Required	Description
recipientId	Long	yes	The unique identifier of the recipient.

Table 3 POST /api/v1/recipient/update parameters

Parameter Name	Data Type	Required	Description
recipientId	String	yes	The unique identifier of the recipient.
email	String	yes	The email address of the recipient. Must be valid.
bic	String	no	BIC or the Business Identifier Code of the bank. Maximum length 8 characters. Must be a valid BIC code.
addressFirstLine	String	no	The first line of the recipient address. Maximum length 255 characters.
addressPostCode	String	no	The post code of the recipient. Maximum length 32 characters.
addressCity	String	no	The city of the recipient. Maximum length 255 characters.
addressCountryCode	String	no	The ISO3 country-code of the recipient.
addressState	String	no	The state of the recipient. US-specific. Maximum length 2 characters.

POST /api/v1/recipient/validate

The same as the **/api/v1/recipient/create** request, but only goes through the validation phase.

Table 4 POST /api/v1/recipient/delete parameters

Parameter Name	Data Type	Required	Description
recipientId	Long	yes	The unique identifier of the recipient.

Table 5 GET /api/v1/recipient/list parameters

Parameter Name	Data Type	Required	Description
currency	String	no	The currency code of the recipient. For example: GBP, USD, EUR.
country	String	no	The ISO3 country-code of the recipient.

Table 6 GET /api/v1/recipient/listTypes parameters

Parameter Name	Data Type	Required	Description
sourceCurrency	String	no	The source currency code of the transfer. For example: GBP, USD, EUR.
targetCurrency	String	no	The target currency code of the transfer. For example: GBP, USD, EUR.
amount	String	no	The amount of the transfer. For example: 1000.54
amountType	String	no	The fix type of the transfer. Either “source” or “target”.

GET /api/v1/recipient/listRefundTypes

The same as the **/api/v1/recipient/listTypes** request, but only for refund recipients.

3.3 Implementation

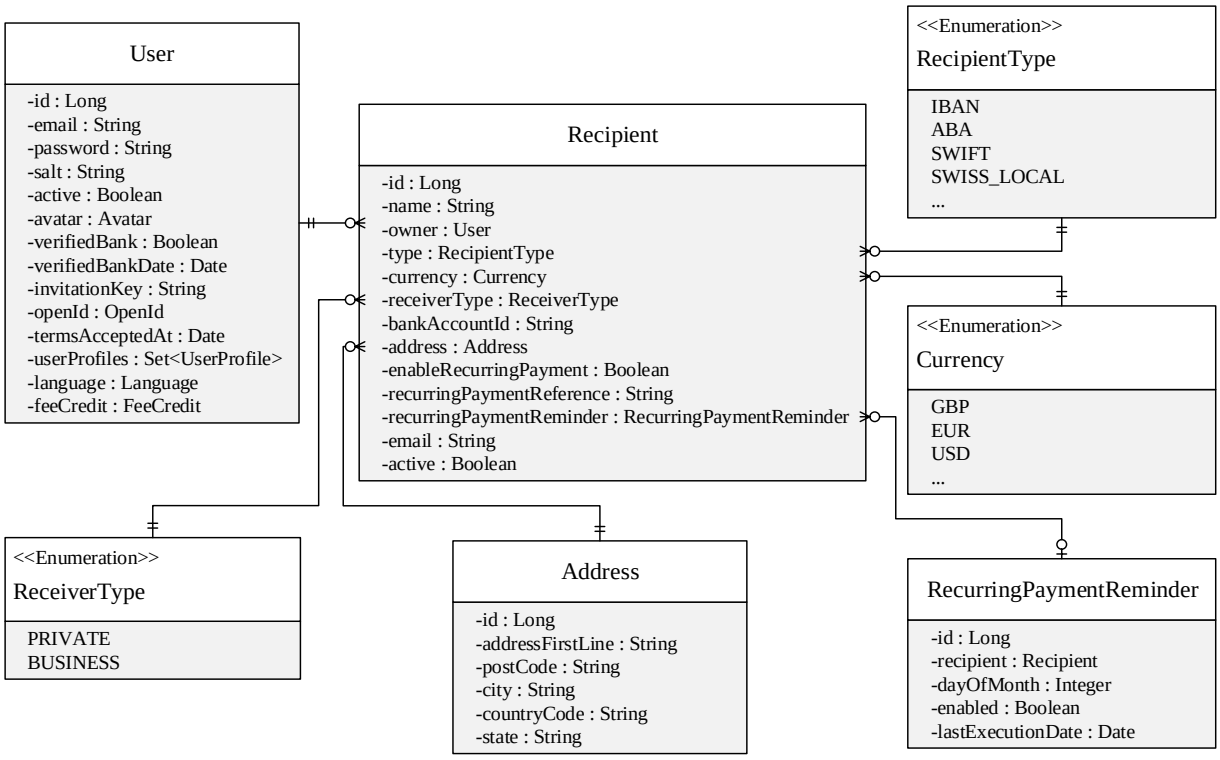

Figure 9 Entity Relationship Diagram

Table 7 Entity Field Descriptions

Entity Name	Field Name	Data Type	Description
Recipient	id	Long	Unique identifier of the recipient.
Recipient	name	String	Bank recipient owner's full name. Either personal or business name. Length between 3-255 characters. Only alphanumeric characters, cannot be blank.
Recipient	owner	User	Reference to the owner of the recipient.
Recipient	type	RecipientType	Reference to the recipient type. Not null.
Recipient	currency	Currency	Reference to the Currency. The recipient can receive funds in this currency only. Not null.
Recipient	receiverType	ReceiverType	A reference to the ReceiverType. Not null.
Recipient	bankAccountId	String	The bank account identifier in the CurrencyCloud system.
Recipient	address	Address	A reference to the Address. Not null.
Recipient	enable-Recurring-Payment	Boolean	Whether the user has set up a recurring payment to the recipient. Defaults to false.
Recipient	recurring-Payment-Reference	String	Recurring payment reference. Nullable.
Recipient	recurring-Payment-Reminder	Recurring-Payment-Reminder (RPR)	Reference to the RecurringPaymentReminder. Nullable.
Recipient	email	String	Email address of the recipient. Must be valid. Not null.
Recipient	active	Boolean	Whether the recipient is in active state or not. Defaults to true.

Entity Name	Field Name	Data Type	Description
User	id	Long	Unique identifier of the user.
User	email	String	Email address of the user. Not null.
User	password	String	Password of the user. Not Null.
User	salt	String	Password encryption salt. Not null.
User	active	Boolean	Whether the user is in active state or not. Defaults to false.
User	verifiedBank	Boolean	Whether the user has a verified bank account. Default to false.
User	verifiedBank-Date	Date	Date when the bank account was verified. Nullable.
User	invitationKey	String	If the user was invited by another user, the invitation key is stored in this field. Nullable.
User	openId	OpenId	Reference to the OpenId. Nullable.
User	termsAcceptedAt	Date	Date when the user accepted the terms. Nullable.
User	userProfiles	Set <UserProfile>	A collection of the UserProfiles (business or personal). Not empty.
User	language	Language	Reference to the language of the user. Defaults to English. Not null.
User	feeCredit	FeeCredit	Reference to the fee credit of the user. These can be manually assigned to users or given for referrals. Nullable.

Entity Name	Field Name	Data Type	Description
Address	id	Long	Unique identifier of the address.
Address	addressFirst-Line	String	The address first line. Maximum length 255 characters. Not null, not blank.
Address	postCode	String	Post code. Maximum length 32 characters. Nullable.
Address	city	String	City. Maximum length 255 characters. Not null, not blank.
Address	countryCode	String	Country code. Maximum length 16 characters. Not null, not blank.
Address	state	String	US state. Maximum length 2 characters.
RPR	id	Long	Unique identifier of the RecurringPaymentReminder.
RPR	recipient	Recipient	Reference to the Recipient.
RPR	dayOfMonth	Integer	Day of month. Minimum 1, maximum 31.
RPR	enabled	Boolean	Whether the reminder is enabled. Defaults to true.
RPR	lastExecution-Date	Date	The last execution date. Nullable.

Each TransferWise user has 0 or more Recipients. Each Recipient must have one and only one ReceiverType (either private or business), RecipientType (IBAN, SWIFT or some other type), Currency (the British Pound, the Euro or some other supported currency) and Address (user-defined). Optionally, a Recipient can have a RecurringPaymentReminder, in case they have set up a recurring payment.

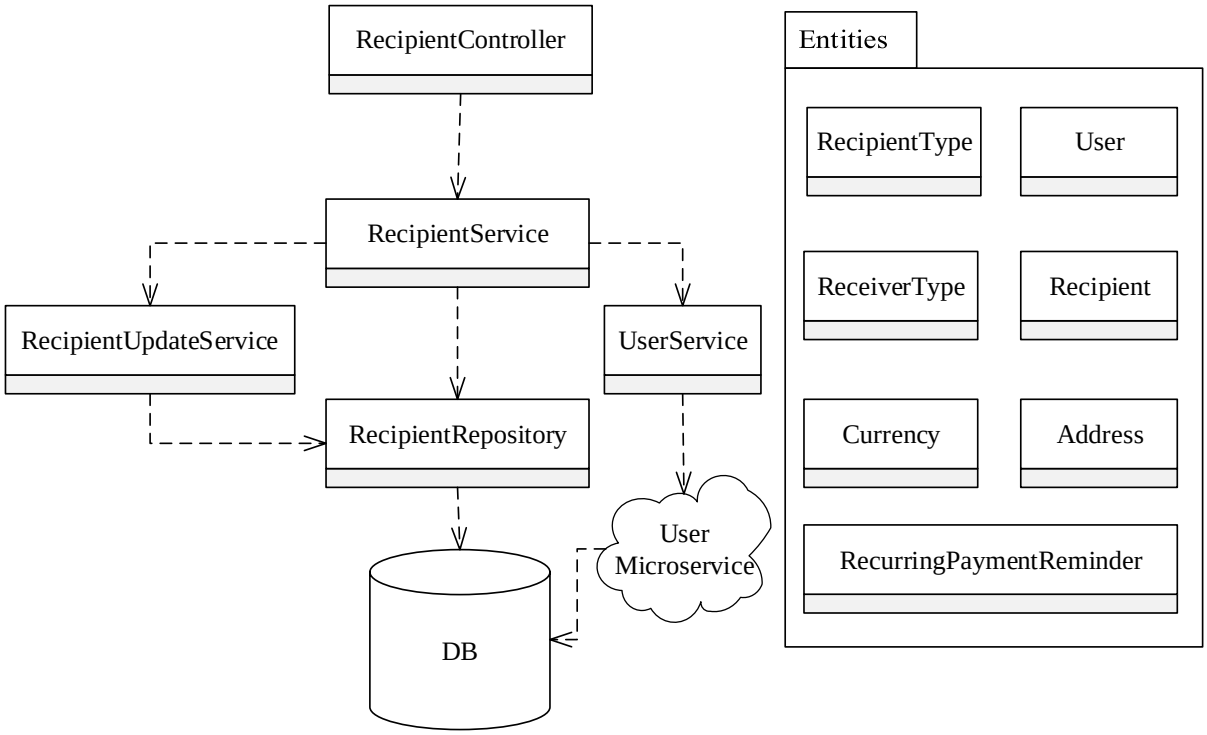

Figure 10 Class Diagram of the Microservice

The Recipient microservice has a single Controller, where all the API endpoints described in chapter 3.2 are defined. The controller is thin and hardly contains any logic. A Controller's sole responsibility is to take the incoming HTTP requests and delegate them to the Service. The services and domain entities is where the business logic is described.

The RecipientService class has a similar public interface to the RecipientController, except the method input parameters are validated DTOs. Since the Recipient update command has more complex business rules, then this logic is separated into its own class. The update call is delegated to the RecipientUpdateService.

The RecipientRepository is a JPA implementation of the Repository enterprise architecture pattern. *"The Repository mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects."* [17]

Figure 11 depicts the recipient creation request handling flow. The DispatcherServlet dispatches the request to the RecipientController. To do that, first it hands over the request to the HandlerMapping which delivers a handler that matches the incoming request. The HttpMessageConverter maps the incoming JSON into an object which, in turn, is passed to the RecipientController. All the necessary validations are done in the Controller layer and the

validated DTO is passed on to the RecipientService. This is where most of the business logic lies (verifications, blacklists etc.).

Once the service is done with all the necessary procedures, it inserts a new row into the database though the RecipientRepository. It then returns a special RecipientResponse DTO to the controller layer that only exposes the fields that are necessary for the view layer. The controller returns it to the DispatcherServlet which converts the DTO back into a JSON HTTP response.

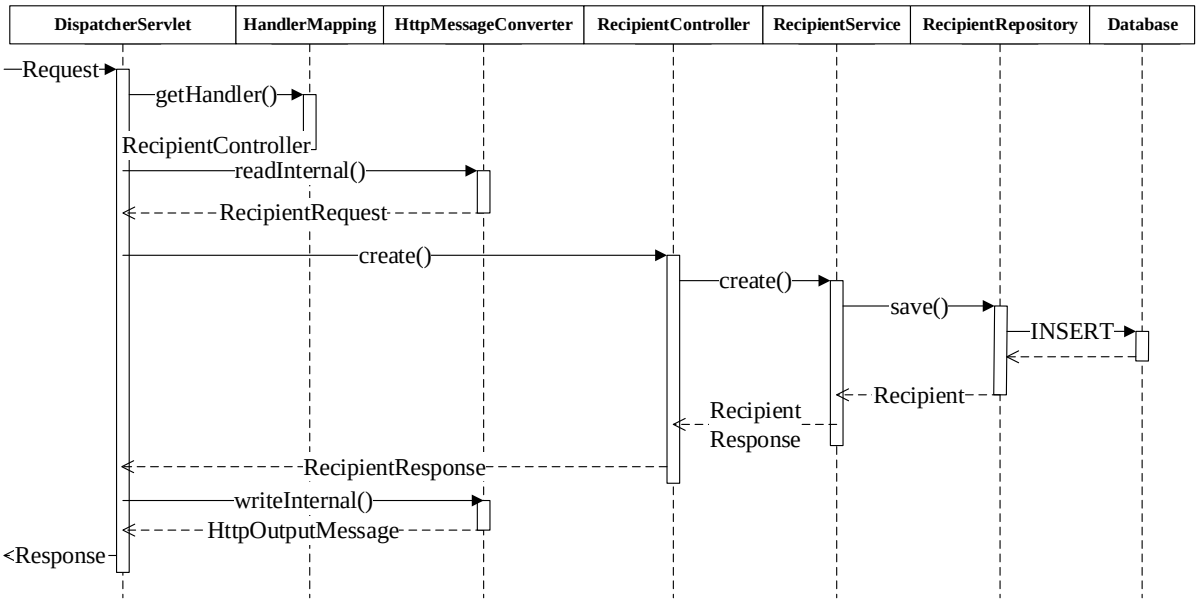

Figure 11 Sequence Diagram of Recipient Creation

3.4 Testing

TransferWise follows the test automation pyramid ideology. The test pyramid concept was first described by Mike Cohn in the book "Succeeding with Agile: Software Development Using Scrum" [18]. It is visualized in Figure 12:

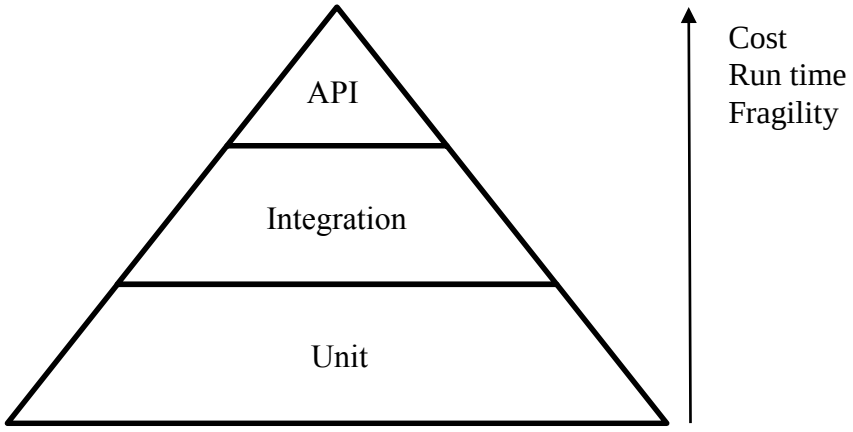

Figure 12 Test Automation Pyramid

3.4.1 Unit Testing

The test automation pyramid argues [19] that you should focus much more on writing unit tests than other types of tests (interface tests and integration tests). First, they are relatively easy to write. Using a unit testing framework like JUnit, you can just create an instance of a class, call some public methods with dummy input data and verify if they return the expected results. Second, they are extremely fast to run: you can run thousands of unit tests in a couple of seconds. Third, as they only test a single unit of code (usually a class), they can only break when the class under the test changes, which is a good thing. Behavior Driven Development (BDD) unit tests are very robust, since they only test the behavior of the unit, not the specific implementation details.

3.4.2 Integration Testing

Integration testing means that individual software components are combined together and tested as a group. This verifies that individual components (classes) interact correctly. Essentially it tests the interface between units. Since integration tests test a larger amount of units at once, they are slower to initialize and run than unit tests. Also, mocking out dependencies might be more complex than for unit tests (when using dependency injection, you can easily inject mock implementations of the services that your class uses). Moreover, integration tests are more fragile than unit tests, since they depend on the implementation of all individual units combined. A single line change in a single unit could break the whole integration test.

3.4.3 End-To-End Testing

End-to-end testing in the context of microservices means testing your service through the public (REST) API. To run end-to-end tests, we have to boot up the whole application, including the whole framework and libraries and start an in-memory database (H2, for example) with some sample data. This essentially means that running the tests is slow, writing the tests is often more complex, and they could easily break when a single implementation detail of the system changes. Therefore, it makes sense to minimize the amount of end-to-end tests you write and only focus on the happy path. Intrinsically, it should just be an automated smoke test.

3.5 Scalability

"Scalability is the ability of a system, network, or process to handle a growing amount of work in a capable manner or its ability to be enlarged to accommodate that growth." [20]

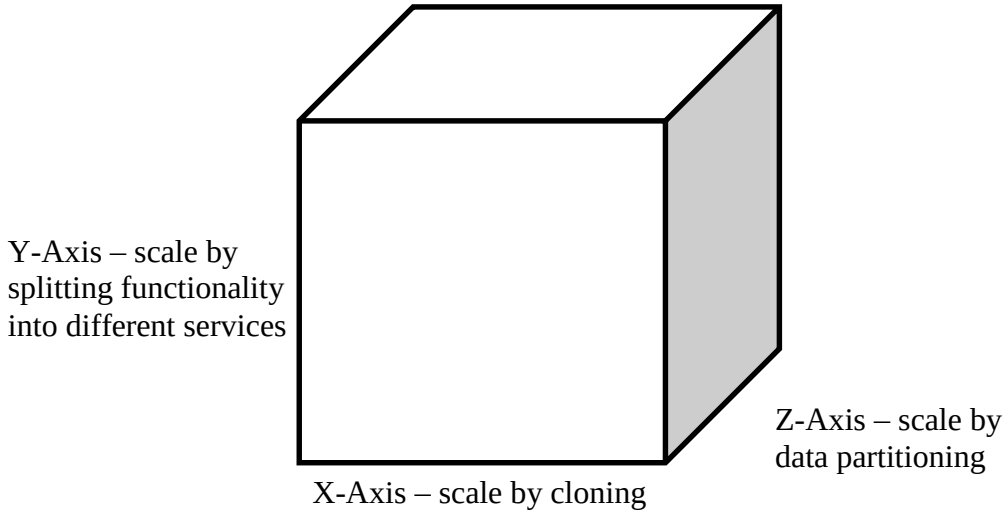

Figure 13 Scale Cube

The scale cube is an easy way to visualize the 3 main ways to scale an application. [21]

3.5.1 X-Axis Scaling

First, X-axis scaling is done by cloning the monolithic application across multiple nodes. This way, the load is spread across multiple instances and it is usually the first and the easiest step to scaling your application. Oftentimes, this strategy is sufficient to serve the needs of a medium-sized business. However, in case of a fast-growing global startup company like TransferWise, this solution is not adequate enough.

3.5.2 Y-Axis Scaling

Second, Y-axis scaling is done by decomposing the monolithic application into services. Usually, a single service represents a set of use cases. According to the Single Responsibility Principle, every service should be responsible over a single functionality provided by the application and that functionality should be encapsulated by the service. Robert C. Martin defines responsibility as a *reason to change*. Thus, a service should have one and only one reason to change. [22] For example, the RecipientService is *only* responsible for handling recipients (and not payments or users, etc.).

3.5.3 Z-Axis Scaling

Third, Z-axis scaling is done by splitting requests or transactions to a single service. For example, requests could be split based on some user characteristic. Essentially, Z-axis scaling is very similar to X-axis scaling: in both cases, server nodes run cloned copies of the application.

However, the main difference is that for Z-axis scaling each node handles only a subset of the incoming requests (and might only hold a subset of the data).

3.6 Service Discovery

The microservice architecture is not only about building individual microservices, but also about how to make the communication and discovery process between services as reliable and fault-tolerant as possible. To accomplish that, a service registry is needed. The service registry is the central repository that registers services and allows other services to look up and connect to services in that directory. One could think of it as a phone book for your microservices. [23]

There are three main requirements to the service registry:

1. high availability and consistency
2. service registration and monitoring mechanism
3. service lookup and connecting mechanism

Currently, TransferWise does not use a service discovery mechanism yet, but as the number of microservices grows, it will soon become a necessity.

3.7 Logging

TransferWise uses the sylog standard to aggregate logs from all the different services and applications. Syslog is essentially a central logging server, where applications send event messages. Most of the applications use the RFC 5424 syslog standard, which has a good support for multiline stacktraces that are essential for debugging errors in production.

On top of syslog, TransferWise uses the Elasticsearch ELK stack. That is: Elastisearch, Logstash and Kibana.

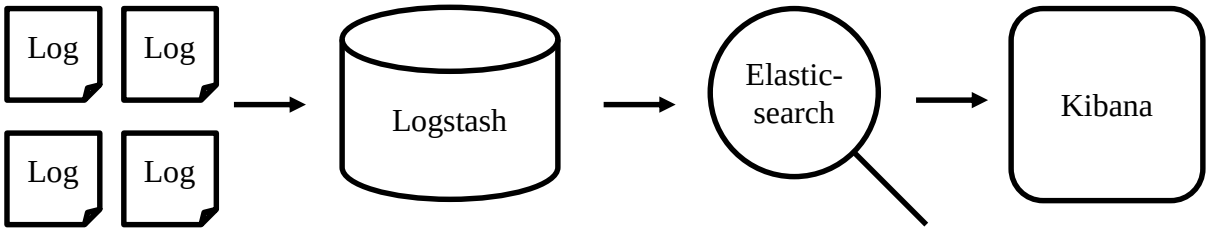

Figure 14 Centralized Log Management

As we can see from Figure 14:

- Logstash collects logs, indexes logs, processes logs and ships logs
- Elasticsearch stores the logs
- Kibana visualizes the logs

Distributed systems like the microservices architecture are great. However, they bring their own challenges. One of them being the traceability of requests across the network of microservices. Luckily, there is an easy solution to this problem. A common solution is to consistently carry a correlation identifier alongside every message that transits through the distributed system. Correlation IDs allow all downstream requests to be correlated with each other based on the unique ID. As TransferWise uses a central logging system, it is very easy to see the request trace throughout the entire distributed application stack (in Kibana). [24]

3.8 Containerization

Spring Boot builds a *fat JAR file* with all the required dependencies packaged including the Spring Framework, embedded Tomcat, etc. The size of this JAR file can be around 20MB. [25]

However, in the containerization world, libraries and the application should be separated. The application changes much more often than the underlying libraries, so it makes sense to separate them as they change for different reasons (SRP). This also speeds up the packaging process of the application, since only the application code is packaged without all the external dependencies.

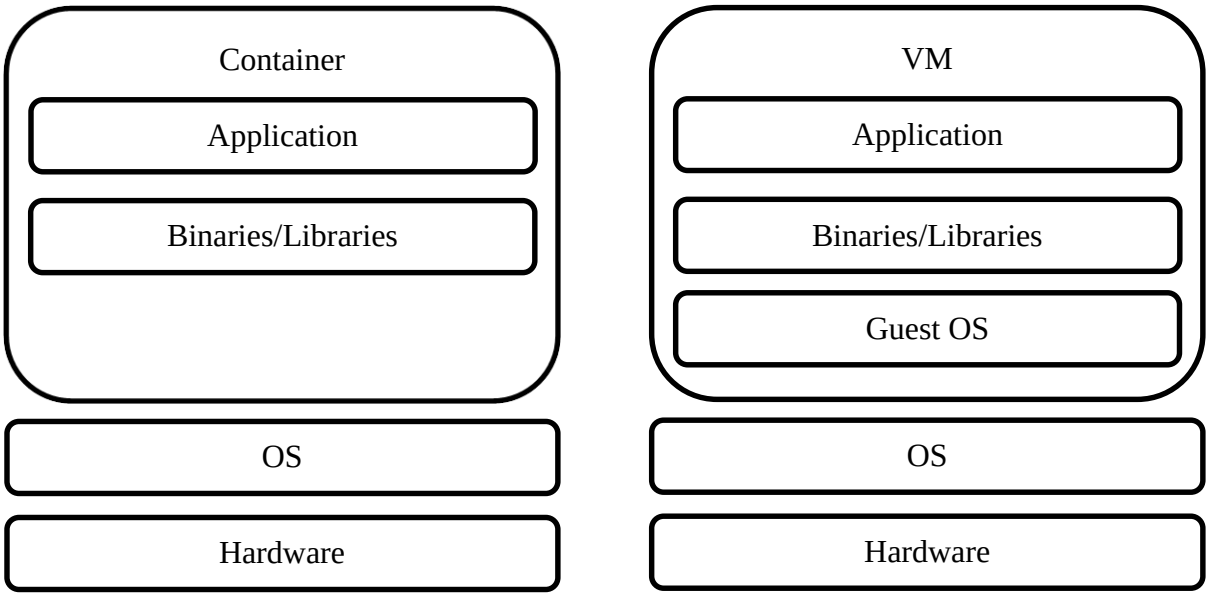

Figure 15 Containers vs. Virtual Machines

TransferWise uses Docker containers to build, ship and run some of its microservices. The main difference between containers and virtual machines is the (lack of a) virtualized OS. The virtual OS poses a significant overhead with its own kernel, memory management and device drivers. Therefore, to solve this problem, containers are executed with the Docker engine instead of the virtual machine hypervisor. Containers are smaller than VMs, enabling faster startups and shutdowns, native performance and smaller file size. However, they provide less isolation and greater compatibility requirements due to the shared drivers and kernel. [26]

3.9 Deployment

TransferWise is a Continuous Delivery (CD) company. It means that the engineering teams at TransferWise have short release cycles and the next version of the application can be reliably and automatically released at any time.

Figure 16 TransferWise Deployment Pipeline

Usually, a new version is released daily, sometimes multiple times per day. To be able to do this, it is essential to have a comprehensive test suite: unit tests, integration tests and functional tests (UI tests). The aim is to automate things as much as possible and for that reason TransferWise does not hire any manual testers or quality assurance engineers. In fact, there are none.

The same principles must hold true for microservices. Moreover, microservice allow TransferWise to extend its continuous delivery process even further. Ideally, every pushed commit should be automatically deployed to production, assuming that the build was successful and all the tests passed. Furthermore, there must be an automatic monitoring system in place that detects issues in production. If the error rates cross the predefined thresholds, the release is automatically rolled back.

4. Architectural Vision

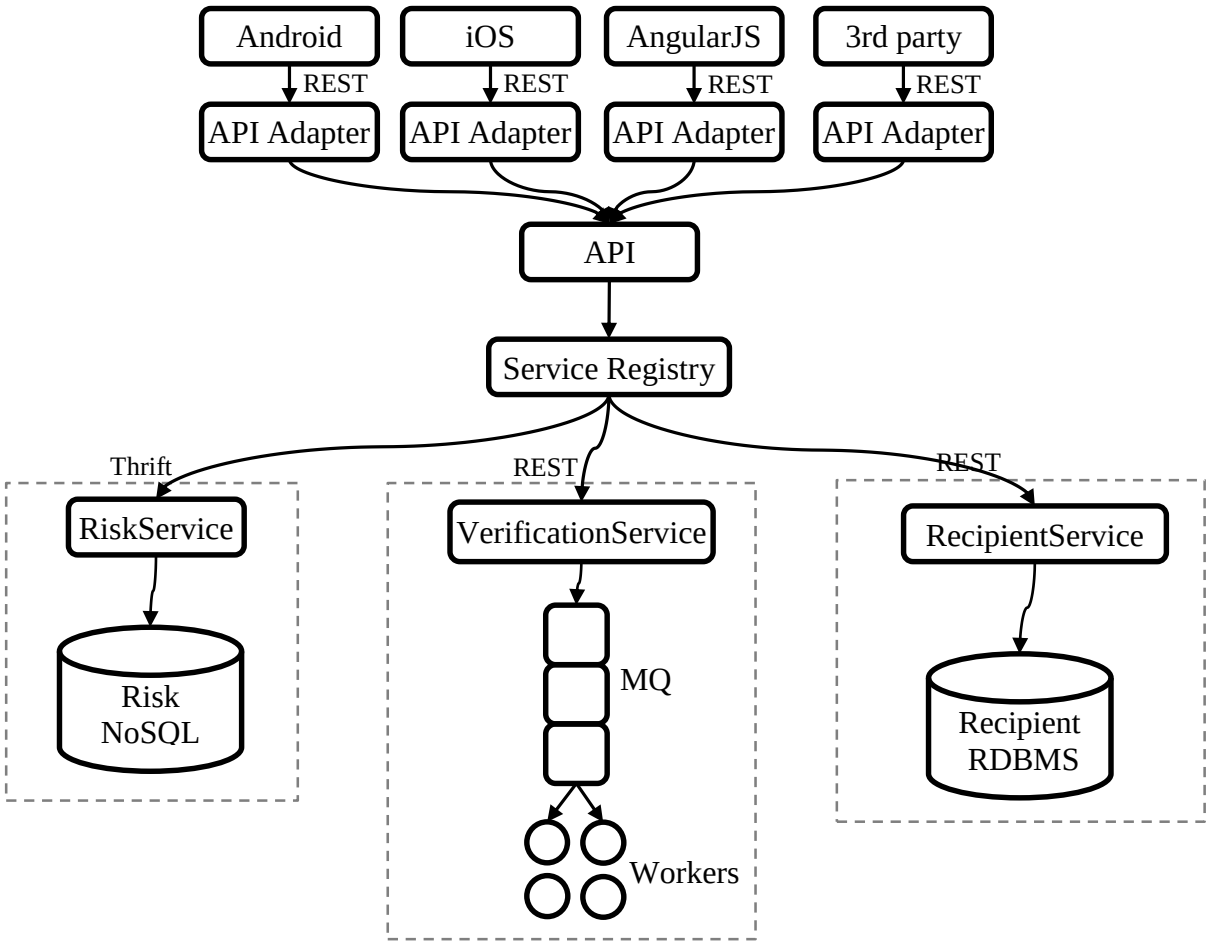

Figure 17 Architectural Vision of TransferWise

The architectural vision that is proposed for TransferWise is a fully distributed system. It consists of a pool of technology-agnostic microservices that all communicate with each other through a central high-availability service registry. Each microservice can use a different communication protocol, framework and database.

Each microservice exposes a public API that is aggregated into a single API endpoint. However, this API endpoint is not directly exposed to the public. Each public API consumer is provided with a client-specific adapter that aggregates individual API requests and provides a specifically tailored API response. [27]

4.1 Building Microservices

The core principles of building microservices can be summarized as follows:

- Modeled around business concepts – microservices should be structured around bounded contexts. Explicitly define the context of each boundary and be explicit about their interrelationships. [28]

- Hidden implementation details – microservices must be technology agnostic. All of the internal implementation details must be hidden, including the specific database technology.

- Independent deployability – each microservice must be independently deployable. This means that one should strive for a limited number of outgoing dependencies on other microservices. This allows the teams to work independently, too. Teams should not constantly orchestrate their deployments to production.

- Full automation – Automated testing is an essential part of the microservices architecture. Without full automation and continuous delivery it can become cumbersome to manually coordinate the infrastructure.

- Decentralization – Make sure the Conway's law is applicable to your organization. Each team should own their own service(s) and weak code ownership should be promoted.

- Failure isolation – Failures should be planned for. We should assume that service calls sometimes fail. Make sure that timeouts are reasonable, and understand what the impact on the customers is if a single module of the system is failing.

- Monitoring – Monitoring single service instances does not guarantee that the system as a whole works correctly. Set up a high level view of the system to make sure it functions correctly. Use correlation IDs for debugging.

5. Summary

The main goals of this thesis were as follows:

1. Analyze the pros and cons of the microservices architecture compared to the monolithic architecture.
2. Extract a single microservice from the monolithic application.
3. Provide a generalized approach to decomposing an application into microservices.

All of the abovementioned goals have been achieved. It is clear that the microservices architecture has an important role in the business growth of TransferWise. A horizontal business must also scale horizontally. The only way to do this sustainably is to decompose your application into independent autonomous services.

However, TransferWise is still far from ideal. There is a lot of work that needs to be done. The evolutionary architecture of TransferWise is firmly moving towards microservices. Change is inevitable – adapt to it.

Kokkuvõte

Käesoleva töö põhieesmärgid olid:

1. Analüüsida mikroteenuste arhitektuuri omadusi ning võrrelda neid monoliitse arhitektuuriga.
2. Eraldada üks näidismikroteenus monoliitsest rakendusest.
3. Kirjeldada üldistatud printsiipe järgmise mikroteenuse monoliidist eraldamiseks.

Antud töö raames said kõik eelnimetatud eesmärgid edukalt täidetud. Mikroteenuste arhitektuuril on TransferWise'i jaoks monoliitse arhitektuuri ees suur eelis. Mikroteenuste arhitektuur peegeldab ettevõtte horisontaalset struktuuri ja kommunikatsioonimudelit. Tarkvara arhitektuuri visandamine ettevõtte-näoliseks (ja vastupidi) aitab TransferWise'l jätkusuutlikult oma äri- ja arendustegevust skaleerida.

Palju on veel teha, praegune arhitektuur on veel kaugel ideaalsest visioonist. Ainuõige viis on inkrementaalselt liikuda samm-sammult uue arhitektuuri poole. Muutused on paratamatud, nendega tuleb õppida kohaneda.

References

[1] N. Peiris, "We Inspire Smart People and We Trust Them," TransferWise, 4 March 2015. [Online]. Available: http://tech.transferwise.com/we-inspire-smart-people-and-we-trust-them. [Accessed 10 May 2015].

[2] S. Newman, Building Microservices, O'Reilly Media, 2015.

[3] M. E. Conway, "How Do Committees Invent?," *Datamation*, pp. 28-31, April 1968.

[4] L. Vogel, "Groovy with Eclipse - Tutorial," Vogella, 29 January 2015. [Online]. Available: http://www.vogella.com/tutorials/Groovy/article.html. [Accessed 12 May 2015].

[5] The Grails Project, "The Grails Framework," 11 May 2015. [Online]. Available: http://grails.org. [Accessed 12 May 2015].

[6] L. Vogel, "Grails Development - Tutorial," Vogella, 8 December 2014. [Online]. Available: http://www.vogella.com/tutorials/Grails/article.html. [Accessed 12 May 2015].

[7] M. Zaleski, "CSC407 Software Architecture & Design," University of Toronto, Toronto, 2004.

[8] Microsoft, "Presentation Layer Guidelines," in *Microsoft Application Architecture Guide*, Seattle, Microsoft Press, 2009, pp. 67-82.

[9] Microsoft, "Business Layer Guidelines," in *Microsoft Application Architecture Guide*, Seattle, Microsoft Press, 2009, pp. 83-94.

[10] Microsoft, "Data Layer Guidelines," in *Microsoft Application Architecture Guide*, Seattle, Microsoft Press, 2009, pp. 95-114.

[11] C. Richardson, "Pattern: Monolithic Architecture," 2014. [Online]. Available: http://microservices.io/patterns/monolithic.html. [Accessed 12 May 2015].

[12] M. Fowler and J. Lewis, "Microservices," 25 March 2014. [Online]. Available: http://martinfowler.com/articles/microservices.html. [Accessed 12 May 2015].

[13] C. Richardson, "Building Microservices with Spring Boot – Part 1," 1 April 2014. [Online]. Available: http://plainoldobjects.com/2014/04/01/building-microservices-with-spring-boot-part1. [Accessed 11 May 2015].

[14] C. Richardson, "Pattern: Microservices Architecture," 2014. [Online]. Available: http://microservices.io/patterns/microservices.html. [Accessed 12 May 2015].

[15] R. C. Martin, "Microservices and Jars," 19 September 2014. [Online]. Available: http://blog.cleancoder.com/uncle-bob/2014/09/19/MicroServicesAndJars.html. [Accessed 13 May 2015].

[16] Pivotal Software, Inc., "Spring Boot," 2015. [Online]. Available: http://projects.spring.io/spring-boot. [Accessed 29 April 2015].

[17] M. Fowler, "Repository: Catalog of Patterns of Enterprise Application Architecture," 1 January 2003. [Online]. Available: http://martinfowler.com/eaaCatalog/repository.html. [Accessed 2015 May 25].

[18] M. Cohn, Succeeding with Agile: Software Development Using Scrum, Addison-Wesley Professional, 2009.

[19] M. Fowler, "Test Pyramid," 1 May 2012. [Online]. Available: http://martinfowler.com/bliki/TestPyramid.html. [Accessed 20 May 2015].

[20] A. B. Bondi, "Characteristics of Scalability and Their Impact on Performance," ACM, New York, 2000.

[21] M. L. Abbott and M. T. Fisher, The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise, Addison-Wesley, 2009.

[22] R. C. Martin, Agile Software Development, Principles, Patterns, and Practices, Prentice Hall, 2002.

[23] J. Long, "Microservice Registration and Discovery with Spring Cloud and Netflix's Eureka," 20 January 2015. [Online]. Available: https://spring.io/blog/2015/01/20/microservice-registration-and-discovery-with-spring-cloud-and-netflix-s-eureka. [Accessed 18 May 2015].

[24] D. Bryant, "Implementing Correlation Ids in Spring Boot (for Distributed Tracing in SOA/Microservices)," 29 May 2014. [Online]. Available: http://java.dzone.com/articles/implementing-correlation-ids. [Accessed 22 May 2015].

[25] S. Egorov, "Spring Boot's Fat Jars vs. Docker," 16 April 2015. [Online]. Available: http://bsideup.blogspot.com/2015/04/spring-boots-fat-jars-vs-docker.html. [Accessed 22 May 2015].

[26] S. Seshachala, "Docker vs VMs," 24 November 2014. [Online]. Available: http://devops.com/2014/11/24/docker-vs-vms. [Accessed 22 May 2015].

[27] D. Jacobson, "Embracing the Differences: Inside the Netflix API Redesign," Netflix, 9 July 2012. [Online]. Available: http://techblog.netflix.com/2012/07/embracing-differences-inside-netflix.html. [Accessed 1 May 2015].

[28] E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Prentice Hall, 2003.

Appendix 1. License

Mina, Erko Risthein (sünnikuupäev: 5. juuli 1991),

1. annan Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose **„Migrating the Monolith to a Microservices Architecture: the Case of TransferWise“**, mille juhendaja on Raul Liivrand,

1.1. reprodutseerimiseks säilitamise ja elektroonilise avaldamise eesmärgil, sealhulgas TTÜ raamatukogu digikogusse lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;

1.2. üldsusele kättesaadavaks tegemiseks Tallinna Tehnikaülikooli veebikeskkonna kaudu, sealhulgas TTÜ raamatukogu digikogu kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.

2. Olen teadlik, et punktis 1 nimetatud õigused jäävad alles ka autorile.

3. Kinnitan, et lihtlitsentsi andmisega ei rikuta kolmandate isikute intellektuaalomandi ega isikuandmete kaitse seadusest ja teistest õigusaktidest tulenevaid õigusi.

_____ (*allkiri*)

_____ (*kuupäev*)

Appendix 2. Metadata

Töö pealkiri (eesti keeles):
Monoliidi migreerimine mikroteenuste arhitektuurile: TransferWise'i näide

Töö pealkiri (inglise keeles):
Migrating the Monolith to a Microservices Architecture: the Case of TransferWise

Autor: Erko Risthein

Juhendaja: Raul Liivrand

Kaitsmise kuupäev:

Töö keel: eng

Asutus (eesti keeles): Tallinna Tehnikaülikool

Asutus (inglise keeles): Tallinn University of Technology

Teaduskond (eesti keeles): Infotehnoloogia teaduskond

Teaduskond (inglise keeles): Faculty of Information Technology

Instituut (eesti keeles): Informaatikainstituut

Instituut (inglise keeles): Department of Informatics

Õppetool (eesti keeles): Infosüsteemide õppetool

Õppetool (inglise keeles): Chair of Information Systems

Märksõnad (eesti keeles): mikroteenused, monoliit, tarkvara arhitektuur, TransferWise

Märksõnad (inglise keeles): microservices, monolith, software architecture, TransferWise