HAW
HAMBURG

Bachelor Thesis

Oleksii Skorykh

Migration of NoSQL (Cassandra) to relational database (Postgres) on high demanded distributed system

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Oleksii Skorykh

Migration of NoSQL (Cassandra) to relational database (Postgres) on high demanded distributed system

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Bachelor of Information Engineering
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr.-Ing. Martin Lapke
Zweitgutachter: Dr. Ivan Morshel

Eingereicht am: 28. November 2019

ABSTRACT

Oleksii Skorykh

Thema der Arbeit
Migration einer NoSQL (Cassandra) Datenbank zu einer relationalen Postgres Datenbank in einem verteilten Hochverfügbarkeitssystem

Stichworte
Datenbanken, Postgres, Cassandra, SQL, NoSQL

Kurzzusammenfassung
Die in dieser Bachelor Thesis diskutierte Migration von einer NoSQL zu einer SQL Datenbank, ins Besondere von Cassandra nach Postgres, umfasst das Design des relationalen Modells, die technischen sowie die kundenspezifischen Anforderungen und den Migrationprozess.

Oleksii Skorykh

Title of Thesis
Migration of NoSQL (Cassandra) to relational database (Postgres) on high demanded distributed system

Keywords
Databases, Postgres, Cassandra, SQL, NoSQL

Abstract
In this bachelor thesis discussed migration of the critical system from NoSQL to SQL database, in particular from Cassandra to Postgres, In the course of the work covered all important steps, like the design of the relational model, defining technical and customer requirements and migration process

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

DRY	Don't Repeat Yourself
DB	Database
CRUD	Create, Read, Update, Delete
API	Application Programming Interface
UML	Unified Modeling Language
TBD	To be discussed
BI	Business Intelligence
MVC	Model-View-Controller

Part I

INTRODUCTION AND THEORY

1 INTRODUCTION

1.1 MOTIVATION

Nowadays, it is impossible to imagine a system without a database. Even when information is stored in files, it can already be considered as some Database (DB). High demand and usage bring many challenges in this area, such as: implementing new types of DB, optimising existing DB solutions for specific purposes, migrating from one type to another, and others.
In software it is usually possible to find multiple solutions for a problem, the same applies to DB. However, they will differ by performance, scalability, efficiency, and what is also crucial for critical systems — maintainability.

1.2 GOAL OF THE THESIS

The goal of the thesis is to change the existent architecture of the system and start to use a relational database instead of non-relational. This work will try to give an answer for the question: "When it makes sense to change the concept of the database in the working system, and if it worth it?", and will be supported with example from the industry, as this work is written together with FREENOW.

1.3 STRUCTURE OF THE THESIS

Here is a brief description of the thesis, providing a short overview of what each chapter contains.
Chapter 2: "Theory:" this chapter provides a theoretical background about relational and non-relational databases, and shows a comparison between them with examples.
Chapter 3: "Requirements:" this chapter gives an overview of functional and nonfunctional requirements for the system.
Chapter 4: "Design:" this chapter describes the design of the relational database model, system overview and architecture of the system, as well as software design.
Chapter 5: "Implementation:" in this chapter, implementation decisions and applied technical solutions are explained.

Chapter 6: "Test:" this chapter provides an overview of the tests that were made to validate the correct behaviour of the system and an explanation of the test results.
Chapter 7: "Summary:" this chapter describes archived results and future improvements.
Appendix: this chapter contains code listings and some technical explanations.

2 THEORY

2.1 RELATIONAL AND NON-RELATIONAL DATABASES

A database can be of two major types: production-oriented or exploratory. An exploratory database is designed to explore possibilities (usually in the future) and to plan possible future activities. Thus, a production-oriented database is intended to reflect reality, while an exploratory database is intended to represent what might be or what might happen. In both cases, the accuracy, consistency, and integrity of the data are essential. Database management involves the sharing of large quantities of data by many users - who, for the most part, conceive their actions on the data independently from one another. [2]

2.1.1 *Relational Databases concept*

The relational model represents the database as a collection of relations. A relation is nothing but a table of values. Every row in the table represents a collection of related data values. These rows in the table denote a real-world entity or relationship.

The table name and column names are helpful to interpret the meaning of values in each row. The data are represented as a set of relations. In the relational model, data are stored as tables.

Some terms related to relational model:

1. Attribute: Each column in a Table. Attributes are the properties which define a relation. e.g., Id, Address, and others.
2. Tables: In the Relational model, the relations are saved in the table format. It is stored along with its entities. A table has two properties rows and columns. Rows represent records and columns represent attributes.
3. Tuple: It is a single row of a table, which contains a single record.
4. Relation Schema: A relation schema represents the name of the relation with its attributes.

5. Degree: The total number of attributes which in the relation is called the degree of the relation.

6. Cardinality: Total number of rows present in the Table.

7. Column: The column represents the set of values for a specific attribute.

8. Relation instance: Relation instance is a finite set of tuples in the RDBMS system. Relation instances never have duplicate tuples.

9. Relation key: Every row has one, two or multiple attributes, which is called the relation key.

10. Attribute domain: Every attribute has some pre-defined value and scope which is known as an attribute domain.

2.1.2 *Non-Relational Databases concept*

A non-relational database is a database that does not use the tabular schema of rows and columns found in most traditional database systems. Instead, non-relational databases use a storage model that is optimised for the specific requirements of the type of data being stored. For example, data may be stored as simple key/value pairs, as JSON documents, or as a graph consisting of edges and vertices.

What all of these data stores have in common is that they do not use a relational model. Also, they tend to be more specific in the type of data they support and how data can be queried. For example, time-series data stores are optimised for queries over time-based sequences of data, while graph data stores are optimised for exploring weighted relationships between entities.

The term NoSQL refers to data stores that do not use SQL for queries, and instead use other programming languages and constructs to query the data. In practice, "NoSQL" means "non-relational database," even though many of these databases do support SQL-compatible queries. However, the underlying query execution strategy is usually very different from the way a traditional Relational Database Management System (RDBMS) would execute the same SQL query. [1]

2.2 COMPARISON OF POSTGRES AND CASSANDRA

In this section, PostgreSQL and Cassandra will be discussed more closely and compared with each other. However, it is difficult to compare these databases because they belong to different groups, this chapter will try to cover the pros and cons of both databases.

2.2.1 *PostgreSQL*

PostgreSQL is an open source relational database management system that began as a University of California, Berkley project. It has enterprise class features such as SQL windowing functions, the ability to create aggregated functions and also utilise them in window constructs, common table expressions, and streaming replications. What sets it apart from other databases is the ease with which user can extend it without changing the underlying base - and in many cases, without any code compilation. [6]

PostgreSQL has earned a strong reputation for its proven architecture, reliability, data integrity, robust feature set, extensibility, and the dedication of the open source community behind the software to consistently deliver performant and innovative solutions. PostgreSQL runs on all major operating systems and has powerful add-ons such as the popular PostGIS geospatial database extender. [4]

2.2.1.1 *Pros of PostgreSQL*

1. Scalable. Vertical scalability is a hallmark of PostgreSQL. Considering that almost any custom software solution tends to grow, resulting in database extension, this particular option certainly supports business growth and development.

2. Support for custom data types. PostgreSQL natively supports a large number of data types by default, such as JSON, XML, H-Store, and others. PostgreSQL takes advantage of it, being one of the few relational databases with strong support for NoSQL features. Additionally, it allows users to define their data types. As software business model may need different types of databases throughout its existence for better performance or application comprehensiveness, this option brings improved flexibility to the table.

3. Easily-integrated third-party tools. PostgreSQL database management system has the strong support of additional tools, both free and commercial. The scope of these includes extensions to improve many aspects. For example, ClusterControl provides impressive assistance at managing, monitoring, and scaling SQL and NoSQL open source databases. In case of

heavy load, there are possibilities of using built-in backup and restore utilities.

4. Open-source and community-driven. Postgres is entirely open-source and supported by its community, which strengthens it as a complete ecosystem. Additionally, developers can always expect free and prompt community assistance.

2.2.1.2 *Cons of PostgreSQL*

1. Changes made for speed improvement requires more work than in other databases as PostgreSQL focuses on compatibility.

2. On performance metrics, it is slower than for example MySQL.

2.2.2 *Cassandra*

Apache Cassandra is a free, open source, distributed data storage system that differs sharply from relational database management systems. Cassandra first started as an incubation project at Apache in January of 2009. Cassandra is being used in production by some of the biggest properties on the Web, including Facebook, Twitter, Cisco, Rackspace, Digg, Cloudkick, Reddit, and more. Cassandra has become so popular because of its outstanding technical features. It is durable, seamlessly scalable, and tuneable consistent. It performs blazingly fast writes, can store hundreds of terabytes of data, and is decentralised and symmetrical so there’s no single point of failure. It is highly available and offers a schema-free data model. [5]

2.2.2.1 *Pros of Cassandra*

1. Write Speed. It is able to handle such a large volume of writes by first writing to an in-memory data structure, then to an append-only log. These data-structures are then "flushed" to a more permanent and read-optimised file at a later time. The logs are simply used for recovery of the in-memory data when an outage occurs.

2. Multi-DC Replication. Out of the box, Cassandra comes with multi data center replication. This replication will copy the information to any number of instances of the Cassandra process. These can be used for geographical performance or for

disaster recovery or both. The multi-datacenter setup is as simple as changing a single line in a configuration file and updating your schema.

3. Tunable Consistency. When it comes to replicated data, there should be a way to decide what happens when an outage occurs in one, or more of the nodes. Cassandra allows, on a query-by-query basis, to decide how to handle potential issues.

4. JVM Based. Apache Cassandra is written in Java. This means that it can integrate tightly with other JVM based applications. In addition to this, the JVM has massive amounts of support and tools to troubleshoot different problems that may arise.

5. CQL. CQL (Cassandra Query Language) is a familiar way of querying Cassandra. It is a subset of SQL and has many of the same features, making the transition from an SQL based RDBMS to Cassandra less complicated.

2.2.2.2 *Cons of Cassandra*

1. No Aggregations. Because Cassandra is a key-value store, doing things like SUM, MIN, MAX, AVG and other aggregations are incredibly resource intensive if even possible to accomplish.

2. No Ad-Hoc Queries. Beneath the covers, the Cassandra data storage layer is basically a key-value storage system. This means that designing how data model should look, depends on the queries that is going to be executed, rather than the structure of the data itself. This can lead to storing the data multiple times in different ways to be able to satisfy the requirements of the system.

3. Unpredictable Performance. Because Cassandra has many different asynchronous jobs and background tasks that are not scheduled by the user, the performance can be unpredictable. This means that it is possible to see performance impacts that may not be related to a query, or volume of queries. This can make troubleshooting performance issues rather difficult.

2.2.3 *Functional comparison*

In this subsection, there is a sum up of the information discussed in previous chapters. In the table presented functional comparison of PostgreSQL and Cassandra.

Features	**PostgreSQL**	**Cassandra**
Tables or Collections	✓	✓
Primary Key	✓	✓
Partition Key	✓	✓
Foreign Key	✓	✗
Global Secondary Indexes	✓	✗
Integrity Constraints	✓	✗
Single-Key & Multi-Key	✓	✗
JOINs	✓	✗
Data Auto-Expiry	✗	✓
Data Volume Stored	Medium	Large
Aggregations	Built-in	External Frameworks
Data Types Stored	Structured Data	Unstructured Data
Data Organisation	Normalised	Denormalised
Fault Tolerance	Manual Failover	Automatic
Use Case	Complex Relational	Simpler Non-Relational

Table 1: How NoSQL data modelling differs from SQL

Part II

THE SHOWCASE

Actual design, implementation and test

3 REQUIREMENTS

3.1 CUSTOMER REQUIREMENTS

Migration from Cassandra to Postgres should be done on an existent critical system, and this implies certain restrictions. The current system supports the creation, storage and managing vouchers that are used by different parties, and a new solution should be able to do the same operations. These vouchers can be used during payment for taxi trips, so having hundreds of thousands trips every day and thousands users every minute that can manage vouchers in their profile, create a significant load for a system and requires it to run without interruptions. So one of the main technical requirement is smooth migration without direct influence on the running system.
All in all, customer requirements can be presented as a list:

1. Support of all operations with vouchers from the old DB solution.
2. Migrate all old information from existing data storage to a new one.
3. Switch to a new solution should be done without any impact on the running system.
4. The migration process should be reliable and safe.

3.2 TECHNICAL REQUIREMENTS

With defined customer requirements, technical one can be derived.

1. Support of Create, Read, Update, Delete (CRUD) operations in new system.
2. New relational schema should be created.
3. All data from Cassandra should be transferred to new Postgres DB.
4. There should be no interruption in work, during migration.

5. Every step during migration should be revertible.
6. In every moment, switch to old system should be possible.

3.2.1 *Software requirements*

To be aligned with current technical stack, developed solution should use Java 8/11, desired DB is Postgres with version 11.

3.2.2 *Data team requirements (External requirement, not in the scope of this thesis)*

Data team would like to consume information from data storage, to process information about vouchers and share it with the Business Intelligence (BI) department. During a discussion with stakeholders, it was decided that the newly developed system will have a function to emit events, that will be later on consumed by Data team. This change will ensure that the system has no direct dependencies, and future changes to architecture can be done with no harm to consumers of data.

4 DESIGN

4.1 DATABASE MODEL

In this section, old and new database models will be presented, with description and explanation of the tables. Diagrams are in simplified form, with only primary and foreign keys, because they contain much information and will make the reading process more difficult. Full diagrams can be found in the appendix section A.1 in the Figure 25 and Figure 26.

Diagrams contain some terms that are related to vouchers, that better to explain now.

1. Deposit voucher - is a process when a passenger adds the voucher to his account to use in future taxi trip.
2. Voucher redemption - is a process when voucher used during payment for the trip and is deducted from passenger account.
3. Referral vouchers are vouchers that every passenger can give to his friend that does not have an account yet, to have a discount on their first trip.

4.1.1 *Cassandra*

In this subsection, the non-relational model can be found, and it is presented in Figure 1. It has a set of tables that stores different data. The main advantage of this model is quick to search, the ability to insert a significant amount of data simultaneously in an asynchronous way. The disadvantage of this model is a big amount of repetitive information in the tables, that leads to a problem that if an error occurred during saving to one of the tables, there would be different data for the same object, that makes information unreliable.

1. *Voucher_by_code*, *Voucher_by_id*, *Voucher_by_media_code* - these tables store voucher entity, and contains the same information, but have a different primary key. This can be explained as a drawback of the non-relational model that information have to be duplicated in order the system to have a quick search for different fields.

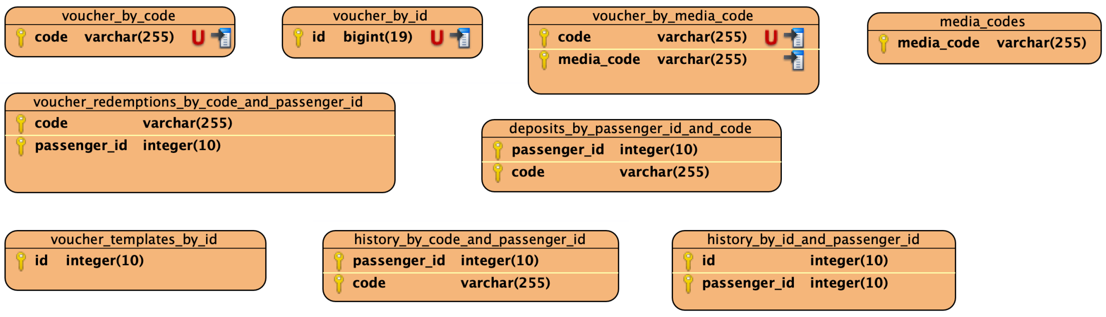


Figure 1: Cassandra database model

2. *media_codes* - this is a special table that stores vouchers, that are created for marketing purposes via an appropriate web system.

3. *voucher_templates_by_id* - this table used as a storage for voucher templates, this templates used for creation referral vouchers.

4. *deposits_by_passenger_id_and_code* - this table keeps track of all deposits of the vouchers done by passengers.

5. *voucher_redemptions_by_code_and_passenger_id* - this table stores information about redemptions of the vouchers.

6. *history_by_code_and_passenger_id*, *history_by_id_and_passenger_id* - these two tables store the same information about historical data, when voucher was deposited, when redeemed, what passenger used it, and so on.

4.1.2 *Postgres*

This subsection presents a relational model in Figure 2, and it is developed as a new solution for the system.

1. *voucher* - this is a main table in the relational model, it stores voucher information and identified by media_code.

2. *voucher_codes* - this table stores voucher codes, that are associated with voucher table. It can be multiple voucher codes that are related to the same media_code.

3. *voucher_templates* - this table used as a storage for voucher templates, this templates used for creation referral vouchers.

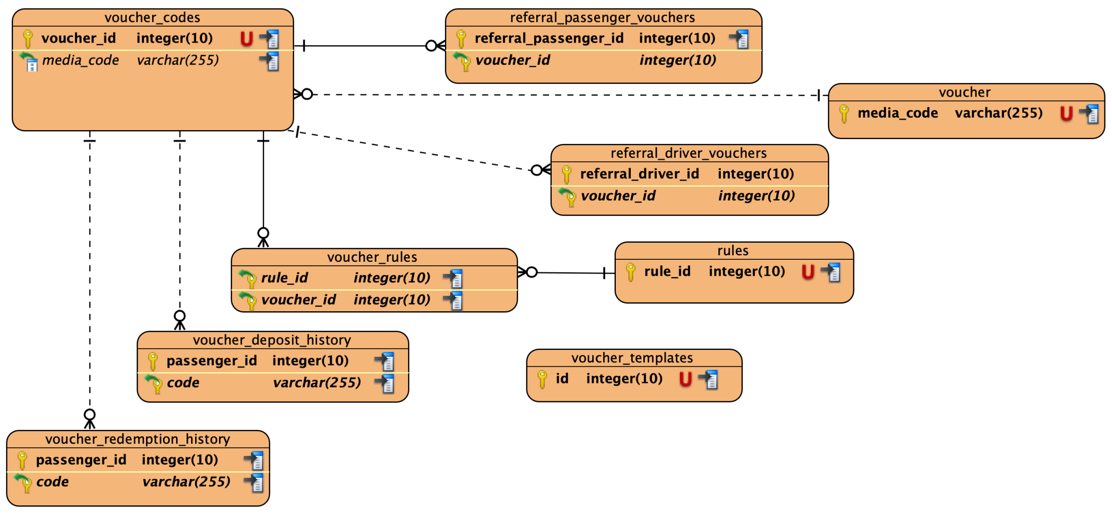


Figure 2: Postgres database model

4. *voucher_deposit_history* - this table keeps track of all deposits of the vouchers done by passengers.
5. *voucher_redemption_history* - this table stores information about redemptions of the vouchers.
6. *referral_passenger_vouchers* - this table stores information related to referral vouchers that are given from one passenger to another, such as name of the passenger and ID of the voucher.
7. *referral_driver_vouchers* - this table stores information related to referral vouchers that are given from driver to passenger.
8. *rules* - this table stores information about validation rules, that are used to validate a voucher. For example, COUNTRY_RULE - checks that country code in the voucher is the same as country code where passenger currently tries to use it.
9. *voucher_rules* - this table stores connection between voucher and rules associated with it.

4.2 HIGH LEVEL DESIGN

This section describes the design and overview of the system, from different prospectives as well as a migration process diagram where is explaination in details what is the plan for migration.

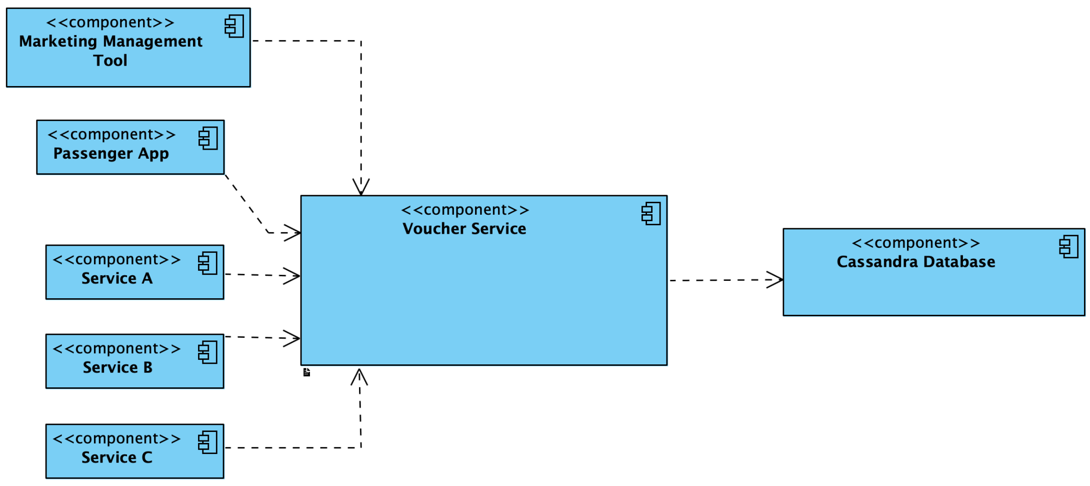
4.2.1 *System overview*

In this subsection is presented a system overview diagram. In the Figure 3 is a current system overview. It contains several elements:

1. Voucher Service: this is the main block of the diagram in the middle, it represents microservice that handles all functions related to the vouchers, such as creation, deletion, edition, depositing to passenger account, using during the payment, and others.

2. Cassandra Database: the component on the right side of the voucher service is data storage that keeps information about the vouchers.

3. Others: on the left side of the vouchers service, there are examples of other components that connect to the voucher service to get information from it. For example, Marketing Web Tool is web application to create vouchers, Passenger App is the primary client of the system and sends a request to deposit voucher, or to redeem it during the payment for the tour.



<<component>> Marketing Management Tool
<<component>> Passenger App
<<component>> Service A
<<component>> Service B
<<component>> Service C
<<component>> Voucher Service
<<component>> Cassandra Database

Figure 3: System overview

4.2.2 *Migration process concept*

This subsection covers the topic of migration from the old database to a new one. Migration plan contains four steps, and all of them presented in the diagrams with explanation. There is a short legend for the diagrams presented in this subsection.

1. An arrow from Voucher Service to Database - represents write operation, it is also marked with blue colour.

2. An arrow from Database to Voucher Service - represents read operation, it is also marked with red colour.

The Figure 4 shows an initial state of the system before the migration. Voucher Service is connected only to Cassandra database, it writes and reads all information from it.

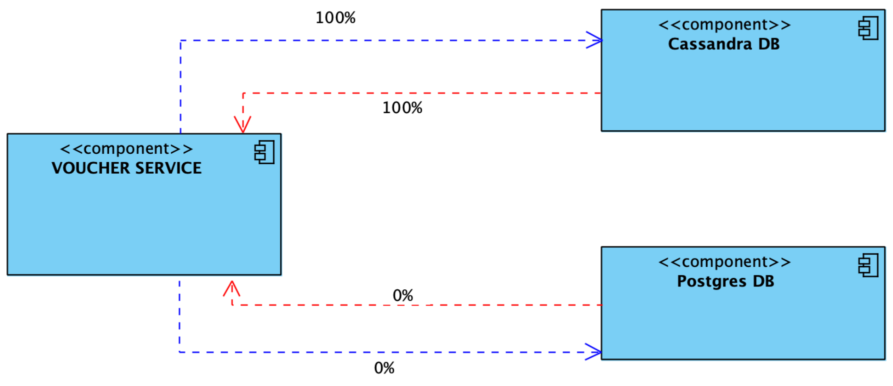


Figure 4: Initial setup of system before migration

The Figure 5 shows the first step of the system migration. The Voucher Service is connected to both Cassandra and Postgres databases and writes information to both of them. At this step, information is only read from Cassandra, and data in Postgres should be closely monitored to see if there is any inconsistency between Cassandra and Postgres.

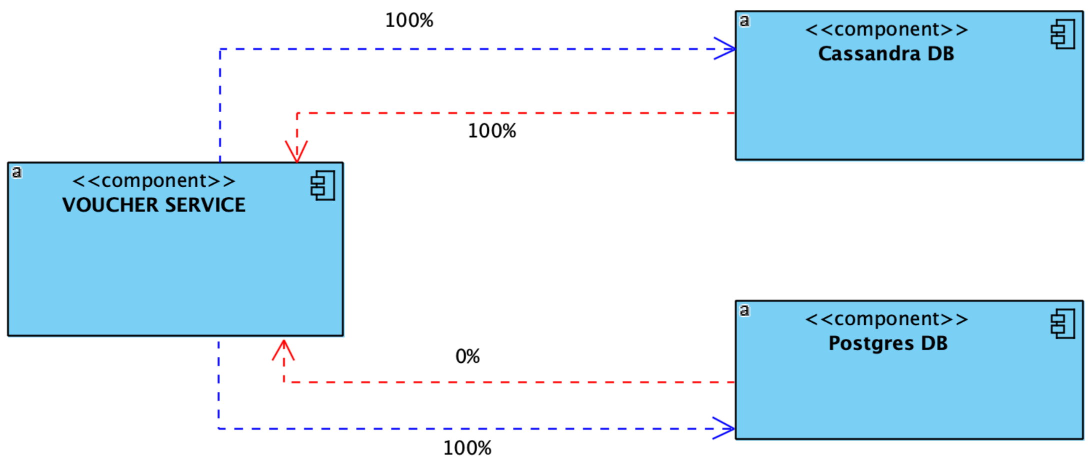


Figure 5: First migration step

The Figure 6 shows a second step of the system migration. Voucher service connected to both Cassandra and Postgres databases, and writes information to both of them. In this step, information is read from both databases, and this behaviour is adjustable, the plan is to read 80% from Cassandra and 20% from Postgres.

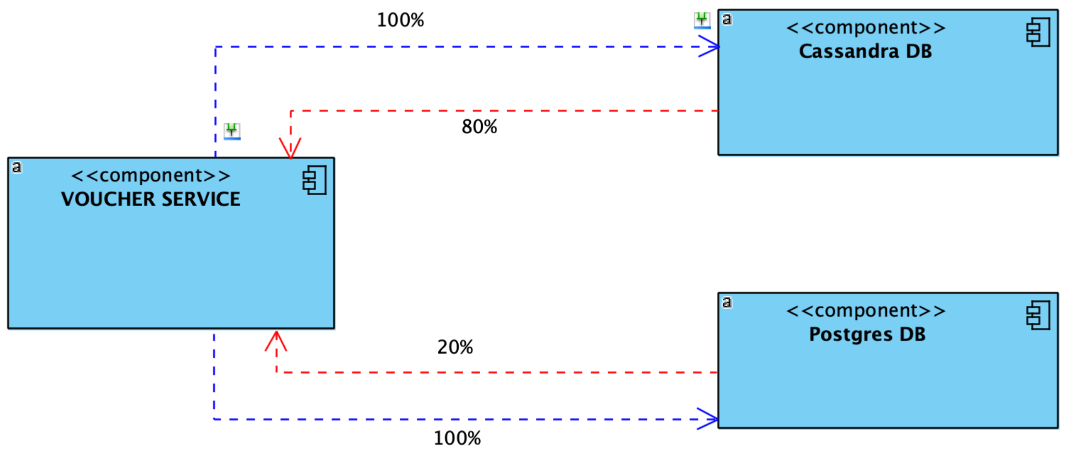


Figure 6: Second migration step

The Figure 7 shows a third step of the system migration. Voucher Service is connected to both Cassandra and Postgres databases and writes information to both of them. At this step, Voucher Service start to read 100% from Postgres, and we keep writing information to Cassandra to have ability immediately switch databases in case of errors.

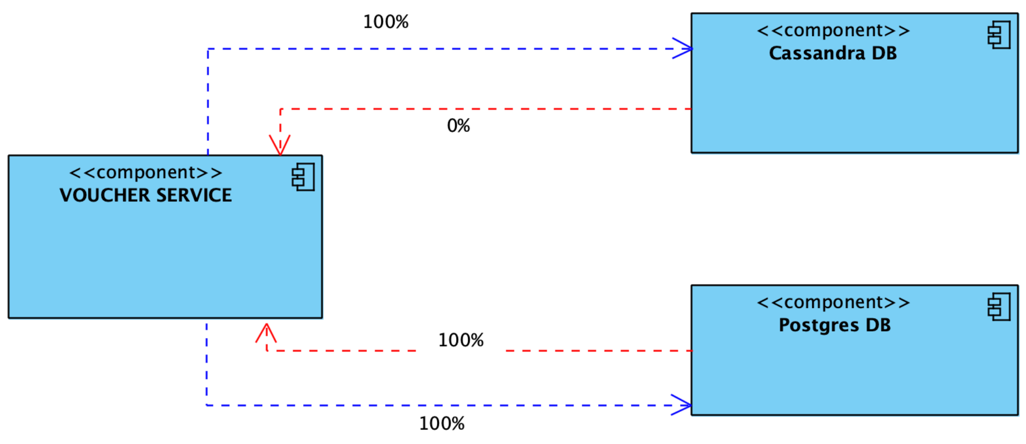


Figure 7: Third migration step

The Figure 8 shows a last step of the system migration. Voucher Service is connected only to the Postgres database. At this step, the Voucher Service communicates only to Postgres. All write and read operations performed using a new setup. Furthermore, old Cassandra will be shutdown.

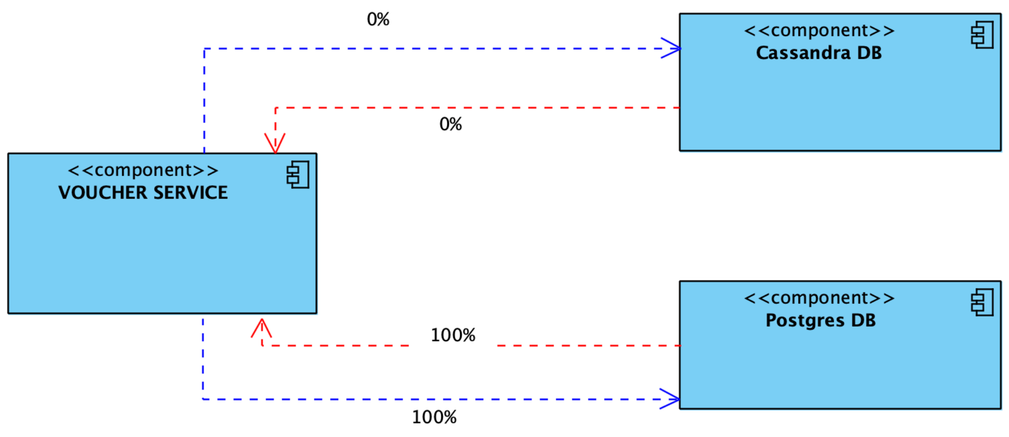


Figure 8: Last migration step

4.3 SOFTWARE DESIGN

In this section, the superior software design is presented. Several diagrams show software design overview in different stages: before migration, during and after migration. An explanation supports them, more implementation details are in Chapter 5. The general architecture of the system follows a Model-View-Controller (MVC) pattern. MVC offers architectural benefits — it helps to write better organised, and therefore more maintainable code. [3]

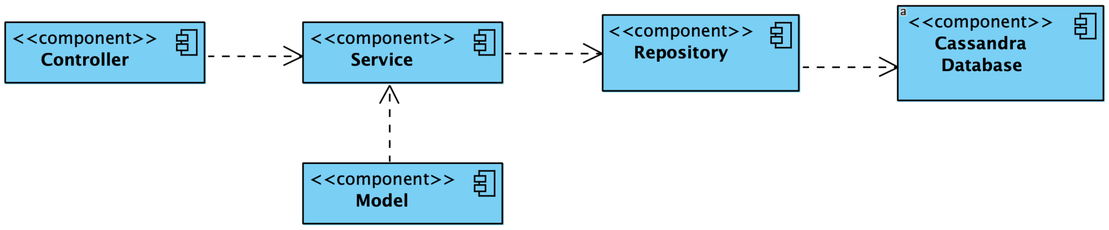


Figure 9: Software top-level overview before migration

In Figure 9 is presented software architecture design before the migration. It contains several components:

1. Controller: In this component, Application Programming Interface (API) for the entire system is defined. It is the entry point, and all requests for the system are handled in the controller layer.

2. Service: this component layer contains all business logic, does manipulations with objects.

3. Model: Model layer contains all objects that are used in the system.

4. Repository: This component is the interface for database communication.

5. Cassandra Database: Database where objects are stored.

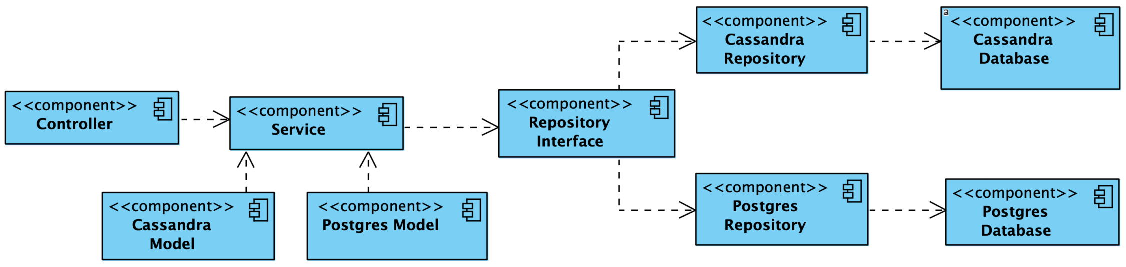

<<component>> Controller
<<component>> Service
<<component>> Repository Interface
<<component>> Cassandra Repository
<<component>> Cassandra Database
<<component>> Postgres Repository
<<component>> Postgres Database
<<component>> Cassandra Model
<<component>> Postgres Model

Figure 10: Software top-level overview during migration

In Figure 10 is presented software architecture design during the migration. It has more components than before, but the controller and service stay almost untouched. The main benefit for all external system, they API is not changed. An architecture change is using Repository Interface, and this is a component that handles which data source should be used for storing and reading data. This architecture gives the flexibility to use not only different data sources but also different frameworks for communication to DB. Added components:

1. Cassandra Model: Contains all objects related to Cassandra Database.
2. Postgres Model: Contains all objects related to Postgres Database.
3. Repository Interface: As described above this components do routing between different data sources.
4. Cassandra Repository: This component is the interface for Cassandra database communication.
5. Postgres Repository: This component is the interface for Postgres database communication.
6. Postgres Database: Database where objects are stored.

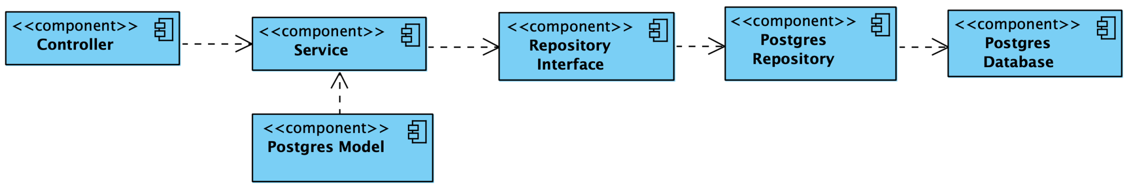

<<component>> Controller
<<component>> Service
<<component>> Repository Interface
<<component>> Postgres Repository
<<component>> Postgres Database
<<component>> Postgres Model

Figure 11: Software top-level overview after migration

In Figure 11 is presented software architecture design after the migration. All parts related to Cassandra are removed. Moreover, the repository interface is staying, in case other data sources will be connected in the future.

5 IMPLEMENTATION

5.1 COMMUNICATION BETWEEN LAYERS

This section covers all implementation details and changes during migration. There are no changes between controller and service layer communication, and major changes are from service to repository interface, which makes decisions what data source Postgres or Cassandra to use. In the subsection, 5.1.2, is described the implementation of communication to the database. Migration brings space to small improvements, so it was decided to use a modern framework(JPA) for all database communications to Postgres, to make maintaining and future development more straightforward.

5.1.1 *Service to Repository layer communication*

In this subsection, communication between service and repository layer is described. There are a lot of different classes because system accesses different tables in the database. In this part will be presented examples of implementation of read and write operations for one table, but it looks similar for other tables as well. Read example in listing 1 shows function to find a voucher or list of vouchers by voucher code. In the beginning, the value from the configuration is stored in the variable, what percentage of requests should get data from Cassandra, the value can be from 0 to 100. And then helper function is used, see Listing 2, to determine should request use Cassandra or Postgres. Implementation of a helper function is straightforward; it generates a random value between 0 and 100 and compares to the value that was taken from configuration.

```
public List<VoucherByCode> findByCode(final String code) {
       Long readRatio = config.getCassandraPostgresReadRatio();
       if (isCassandra(readRatio)) {
           return voucherByCodeRepositoryCassandra.findByCode(
               code);
       }
       else {


           return voucherByCodeRepositoryPostgres.findAllByCode
               (code)
               .stream()
               .map(voucherCodesMapper::mapVoucherCodes)
               .collect(Collectors.toList()); } }
```

Listing 1: Find voucher by code

```
private boolean isCassandra(Long ratio) {
        return (getRandomNumberInRange() <= ratio);
    }

    private static int getRandomNumberInRange() {
        return (int) (Math.random() * ((100) + 1));
    }
```

Listing 2: Helper function, to determine to read from Cassandra or not

Listing 3 shows an example of saving voucher to the database. At the beginning of the method, in two variables, configuration values are stored , isCassandraEnabled and isPostgresEnabled. And then there is a check, should data be saved to one database or both. There is no need to have percentage values like in reading example because data should be consistent in both databases. Otherwise, the split-brain situation can happen, when part of the data is stored in one database, and part in the other one. And then attempt of the system to find particular information will be unsuccessful.

```
public void save(VoucherByCode voucherByCode)  {
        boolean isCassandraEnabled = config.isCassandraEnabled()
            ;
        boolean isPostgresEnabled = config.isPostgresEnabled();
        if (isCassandraEnabled) {
            voucherByCodeRepositoryCassandra.save(voucherByCode)
                ;
        }
        else if (isPostgresEnabled) {
            voucherByCodeRepositoryPostgres.save(
                voucherCodesMapper.mapVoucherByCode(
                voucherByCode));
        }
    }
```

Listing 3: Save voucher

5.1.2 *Repository to Database layer communication*

This chapter provides comparison of the same functionality when system communicates to Cassandra or Postgres, to show improvements in readability and maintainability of the code.

```java
public class MediaCodeVoucherCountRepositoryCassandra {
    private final CachingPreparedStatementCreator stmtGetCount =
         new CachingPreparedStatementCreator(
        "select voucher_count from media_code_voucher_count
            where media_code = ?;");
    private final Session session;

    public MediaCodeVoucherCountRepositoryCassandra(Session
        session) { this.session = session;}

     public Long getCount(String mediaCode) {
        try {
            BoundStatement statement = stmtGetCount.
                createPreparedStatement(session).bind(mediaCode)
                ;
            ListenableFuture<ResultSet> resultSet = session.
                executeAsync(statement);
            ListenableFuture<Long> voucherCount = Futures.
                transform(resultSet, (Function<ResultSet, Long>)
                 rs -> {
                assert rs != null;
                Row row = rs.one();
                return row == null ? 0 : row.getLong("
                    voucher_count");
            });
            return voucherCount.get();
            } catch (InterruptedException | ExecutionException e
                ) {
            throw new RuntimeException("", e); } }
```

Listing 4: Get number of voucher codes for voucher in Cassandra

Listing 4 and 5 shows an implementation of function to get the number of the voucher codes for a particular voucher. From complex function using old technologies, it becomes an understandable and small piece of code.

```java
public interface VoucherRepositoryPostgres extends JpaRepository
    <Voucher, Integer> {
```

```
@Query(value = "select count from voucher where media_code = ?"
    , nativeQuery = true)
   Long getCount(String mediaCode); }
```

Listing 5: Get number of voucher codes for voucher in Postgres

6 TEST

6.1 TEST OF REQUIREMENTS

This section is about checking technical requirements, and how they are fulfilled in the new system.

1. Support of CRUD operations in new system.
2. New relational schema should be created.
3. All data from Cassandra should be transferred to new Postgres DB.
4. There should be no interruption in work, during migration.
5. Every step during migration should be revertible.
6. In every moment, switch to old system should be possible.

Req. #	Result
1	Common repository interface has all functions as old system
2	A new database model is created and described in section 4.1.2
3	This step is not finished completely due to difficulties
4, 5, 6	Enabling and disabling of the database is controlled by configuration

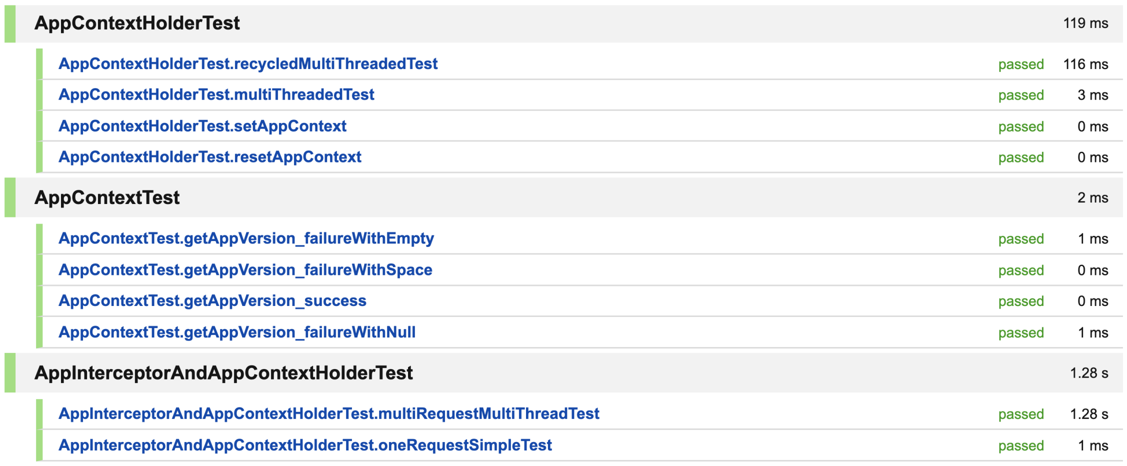
Table 2: Comparison of the system to technical requirements

6.2 UNIT TESTS

This section shows test results of unit tests. System contains 365, that covers all layers. UNIT TESTING is a level of software testing where individual units/ components of a software are tested. The purpose is to validate that each unit of the software performs as designed. A unit is the smallest testable part of any software. It usually has one or a few inputs and usually a single output.

6.2.1 *System context test*

General tests that checks if system can startup, handle single or multiple requests, Figure 12

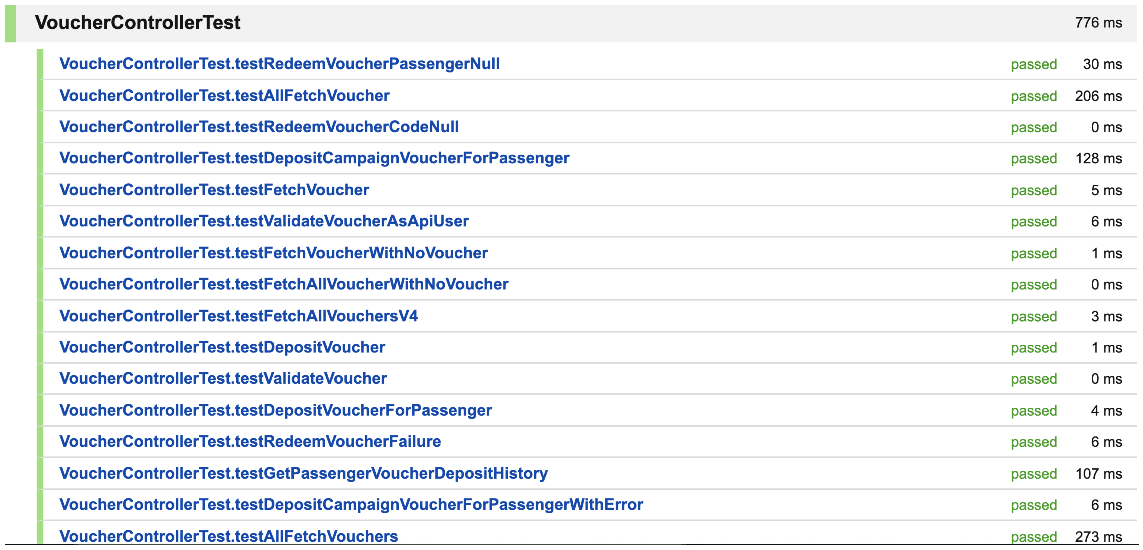


AppContextHolderTest		119 ms
AppContextHolderTest.recycledMultiThreadedTest	passed	116 ms
AppContextHolderTest.multiThreadedTest	passed	3 ms
AppContextHolderTest.setAppContext	passed	0 ms
AppContextHolderTest.resetAppContext	passed	0 ms
AppContextTest		2 ms
AppContextTest.getAppVersion_failureWithEmpty	passed	1 ms
AppContextTest.getAppVersion_failureWithSpace	passed	0 ms
AppContextTest.getAppVersion_success	passed	0 ms
AppContextTest.getAppVersion_failureWithNull	passed	1 ms
AppInterceptorAndAppContextHolderTest		1.28 s
AppInterceptorAndAppContextHolderTest.multiRequestMultiThreadTest	passed	1.28 s
AppInterceptorAndAppContextHolderTest.oneRequestSimpleTest	passed	1 ms

Figure 12: System context test

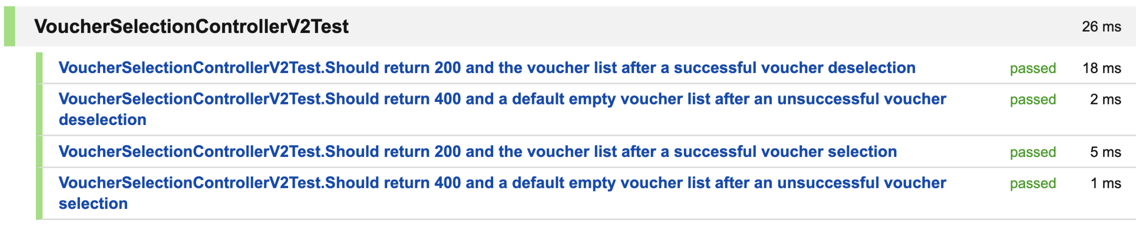
6.2.2 *Controller test*

Controller layer tests to check main functionality to get, deposit, redeem vouchers, Figure 13, and to select and deselect them, Figure 14



VoucherControllerTest		776 ms
VoucherControllerTest.testRedeemVoucherPassengerNull	passed	30 ms
VoucherControllerTest.testAllFetchVoucher	passed	206 ms
VoucherControllerTest.testRedeemVoucherCodeNull	passed	0 ms
VoucherControllerTest.testDepositCampaignVoucherForPassenger	passed	128 ms
VoucherControllerTest.testFetchVoucher	passed	5 ms
VoucherControllerTest.testValidateVoucherAsApiUser	passed	6 ms
VoucherControllerTest.testFetchVoucherWithNoVoucher	passed	1 ms
VoucherControllerTest.testFetchAllVoucherWithNoVoucher	passed	0 ms
VoucherControllerTest.testFetchAllVouchersV4	passed	3 ms
VoucherControllerTest.testDepositVoucher	passed	1 ms
VoucherControllerTest.testValidateVoucher	passed	0 ms
VoucherControllerTest.testDepositVoucherForPassenger	passed	4 ms
VoucherControllerTest.testRedeemVoucherFailure	passed	6 ms
VoucherControllerTest.testGetPassengerVoucherDepositHistory	passed	107 ms
VoucherControllerTest.testDepositCampaignVoucherForPassengerWithError	passed	6 ms
VoucherControllerTest.testAllFetchVouchers	passed	273 ms

Figure 13: Voucher Controller test

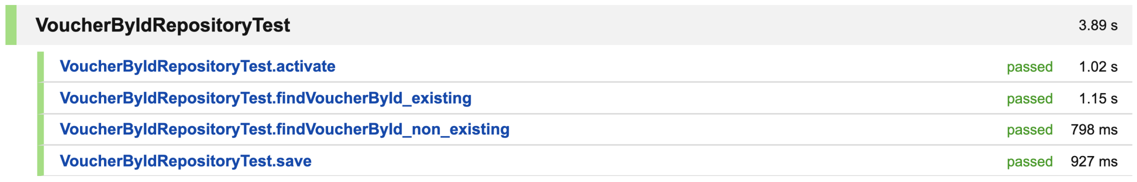


VoucherSelectionControllerV2Test		26 ms
VoucherSelectionControllerV2Test.Should return 200 and the voucher list after a successful voucher deselection	passed	18 ms
VoucherSelectionControllerV2Test.Should return 400 and a default empty voucher list after an unsuccessful voucher deselection	passed	2 ms
VoucherSelectionControllerV2Test.Should return 200 and the voucher list after a successful voucher selection	passed	5 ms
VoucherSelectionControllerV2Test.Should return 400 and a default empty voucher list after an unsuccessful voucher selection	passed	1 ms

Figure 14: Voucher Selection Controller test

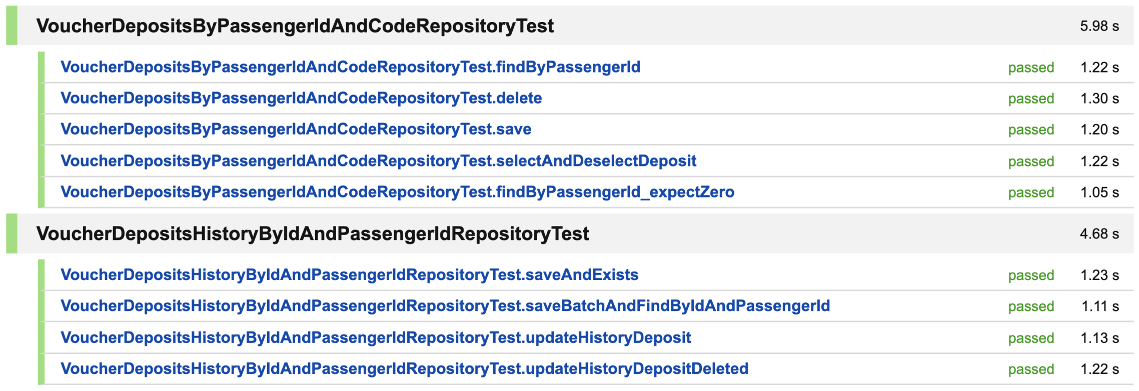
6.2.3 *Data access layer test*

Tests to check operations with database. For example history tables Figure 16, or table to get voucher information Figure 15.



VoucherByIdRepositoryTest		3.89 s
VoucherByIdRepositoryTest.activate	passed	1.02 s
VoucherByIdRepositoryTest.findVoucherById_existing	passed	1.15 s
VoucherByIdRepositoryTest.findVoucherById_non_existing	passed	798 ms
VoucherByIdRepositoryTest.save	passed	927 ms

Figure 15: Voucher Repository test



VoucherDepositsByPassengerIdAndCodeRepositoryTest		5.98 s
VoucherDepositsByPassengerIdAndCodeRepositoryTest.findByPassengerId	passed	1.22 s
VoucherDepositsByPassengerIdAndCodeRepositoryTest.delete	passed	1.30 s
VoucherDepositsByPassengerIdAndCodeRepositoryTest.save	passed	1.20 s
VoucherDepositsByPassengerIdAndCodeRepositoryTest.selectAndDeselectDeposit	passed	1.22 s
VoucherDepositsByPassengerIdAndCodeRepositoryTest.findByPassengerId_expectZero	passed	1.05 s
VoucherDepositsHistoryByIdAndPassengerIdRepositoryTest		4.68 s
VoucherDepositsHistoryByIdAndPassengerIdRepositoryTest.saveAndExists	passed	1.23 s
VoucherDepositsHistoryByIdAndPassengerIdRepositoryTest.saveBatchAndFindByIdAndPassengerId	passed	1.11 s
VoucherDepositsHistoryByIdAndPassengerIdRepositoryTest.updateHistoryDeposit	passed	1.13 s
VoucherDepositsHistoryByIdAndPassengerIdRepositoryTest.updateHistoryDepositDeleted	passed	1.22 s

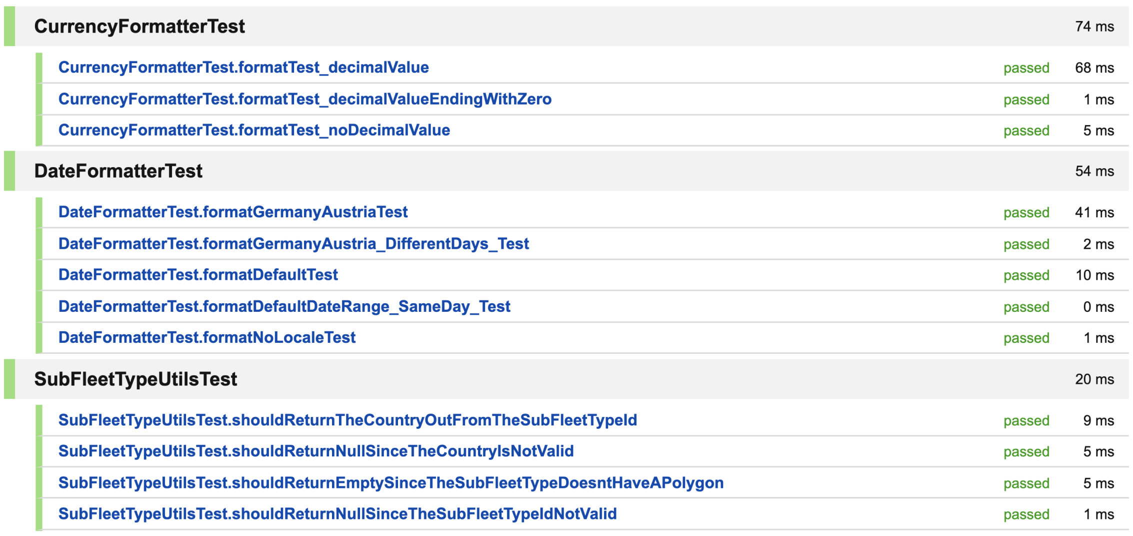
Figure 16: Voucher History Repository test

6.2.4 *Service layer test*

This layer responsible for all business logic in the system, so it has 195 tests. To make it more readable, service layer test results are in the Appendix A in Figure 18, 19, 20, 21, 22, 23, 24

6.2.5 *Utility layer test*

Tests that check helper functions, Figure 17.



CurrencyFormatterTest		74 ms
CurrencyFormatterTest.formatTest_decimalValue	passed	68 ms
CurrencyFormatterTest.formatTest_decimalValueEndingWithZero	passed	1 ms
CurrencyFormatterTest.formatTest_noDecimalValue	passed	5 ms
DateFormatterTest		54 ms
DateFormatterTest.formatGermanyAustriaTest	passed	41 ms
DateFormatterTest.formatGermanyAustria_DifferentDays_Test	passed	2 ms
DateFormatterTest.formatDefaultTest	passed	10 ms
DateFormatterTest.formatDefaultDateRange_SameDay_Test	passed	0 ms
DateFormatterTest.formatNoLocaleTest	passed	1 ms
SubFleetTypeUtilsTest		20 ms
SubFleetTypeUtilsTest.shouldReturnTheCountryOutFromTheSubFleetTypeId	passed	9 ms
SubFleetTypeUtilsTest.shouldReturnNullSinceTheCountryIsNotValid	passed	5 ms
SubFleetTypeUtilsTest.shouldReturnEmptySinceTheSubFleetTypeDoesntHaveAPolygon	passed	5 ms
SubFleetTypeUtilsTest.shouldReturnNullSinceTheSubFleetTypeIdNotValid	passed	1 ms

Figure 17: Utility layer test

Part III

SUMMARY AND OUTLOOK

7 SUMMARY

7.1 RESULTS

The topic of this thesis is the migration of the non-relational database to relational one in a critical system. In the fast-growing system, it is crucial to have the ability to debug, change and add new functionality without problems and significant developer effort. Migration to Postgres, is an excellent example of it, at the start of the migration, old architecture had many dependencies between components that provide more technical depth. After several iterations of refactoring that have been made in a scope of the bachelor thesis, the system now has flexible and scalable architecture, increased performance and less points of failure. Furthermore, this improvements and extended documentation made system easier to understand to other developers.
Answer to question raised at the beginning of the thesis: "When it makes sense to change the concept of the database in the working system, and if it worth it?" is :

1. All stakeholders must have a clear picture of what result they want to achieve.
2. All technical steps and architecture decisions should be sorely discussed together with alternatives.
3. There must be full control during migration to have the ability to revert the process in case of any errors. Error tracing brings an important point, that system should provide a sufficient amount of logs and metrics to see the state of it, and identify problems as soon as possible.

7.1.1 *After migration comparison*

The main difference is the use of a relational database model that gives the flexibility of creating and editing entities(vouchers). There are more points to mention:

1. One constraint that was in an old system is the possibility to create a voucher with a maximum of 50 thousand voucher codes, due to faulty architecture of the database model and

process around it. Using Postgres, this limitation is removed, and it gives flexibility for marketing managers.

2. As discussed in previous chapters in Cassandra information is stored in multiple tables, that creates difficulties when there is a need to delete or change many rows. For vouchers, information must be changed in 3 tables where the main voucher object is stored, and then in all history tables and respectively in tables that store redemption and deposits. In Postgres, to do the same operations, just one table should be changed - voucher.

3. Usage of modern frameworks reduces code complexity, and new system architecture provides flexibility and scalability for future growth.

7.2 OUTLOOK

This section covers future steps and what is need to be done to finish the project.

1. Migrate data from the old database to a new one. During implementation of this task, some problems were faced.
   a) Cassandra does not support most of the SQL queries such as aggregations and search using a not primary key, that makes challenging to migrate many data.
   b) Implementation of the custom solution takes much time and goes beyond the scope of the thesis.

   The solution is to use the help of the data team because they have experience to solve similar problems.

2. Create dashboard and track metrics to identify issues with database and system.

3. Add fallback to critical parts of the system.

4. During the migration to a new system, more features have been added, that should be adopted to a new database model.

REFERENCES

[1] Microsoft Azure. *Non-relational data and NoSQL*. https://docs.microsoft.com/en-us/azure/architecture/data-guide/big-data/non-relational-data. Accessed on 2019-09-25. 2018.

[2] E. F. Codd. *The Relational model for Database Management*. 2nd. Boston, MA, USA: Addison–Wesley, 1990.

[3] Google. *MVC Architecture*. https://developer.chrome.com/apps/app_frameworks/. Accessed on 2019-10-30.

[4] The PostgreSQL Global Development Group. *What is PostgreSQL?* https://www.postgresql.org/about/. Accessed on 2019-10-27.

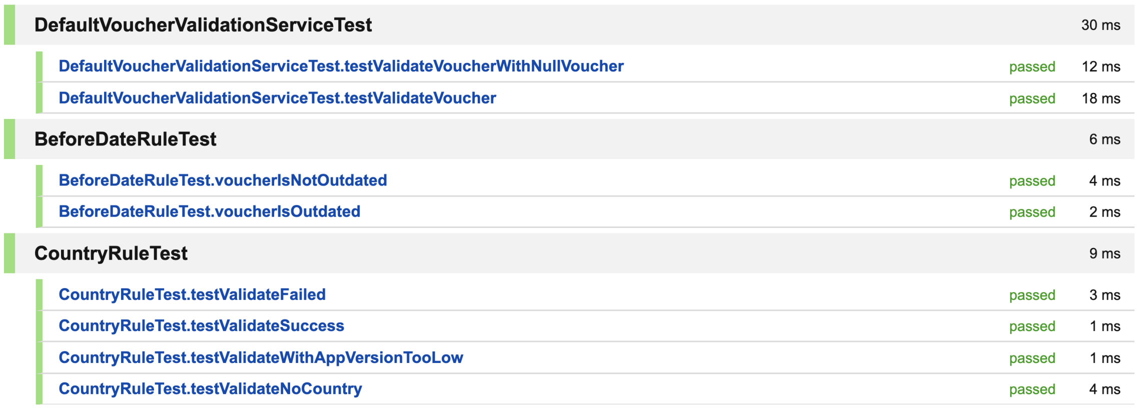
[5] Eben Hewitt. *Cassandra: The Definitive Guide*. 1st. O'Reilly Media, 2011.

[6] Regina Obe and Leo Hsu. *PostgreSql: Up and Running*. 1st. O'Reilly Media, 2012.

Part IV

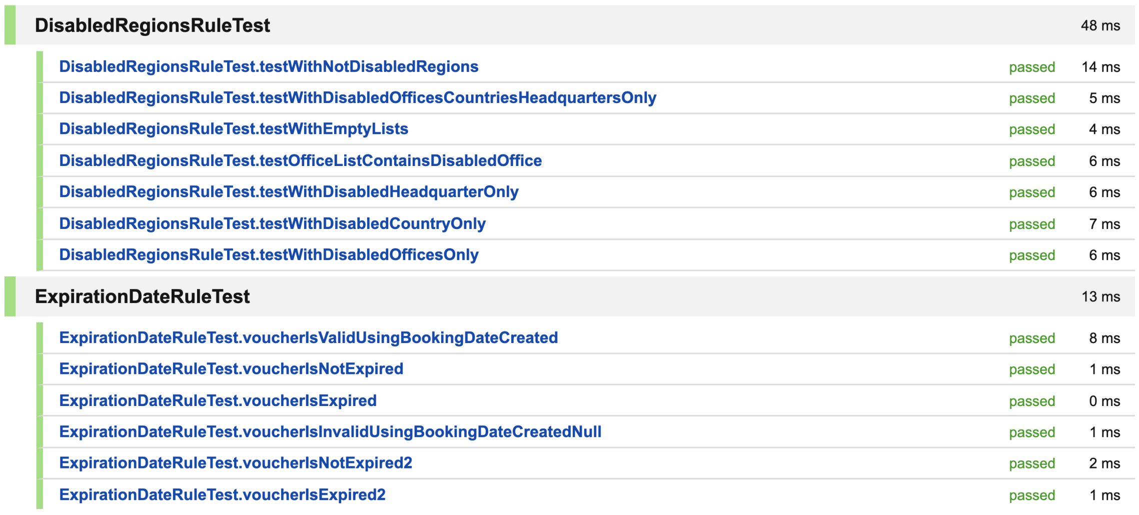
APPENDIX

A APPENDIX



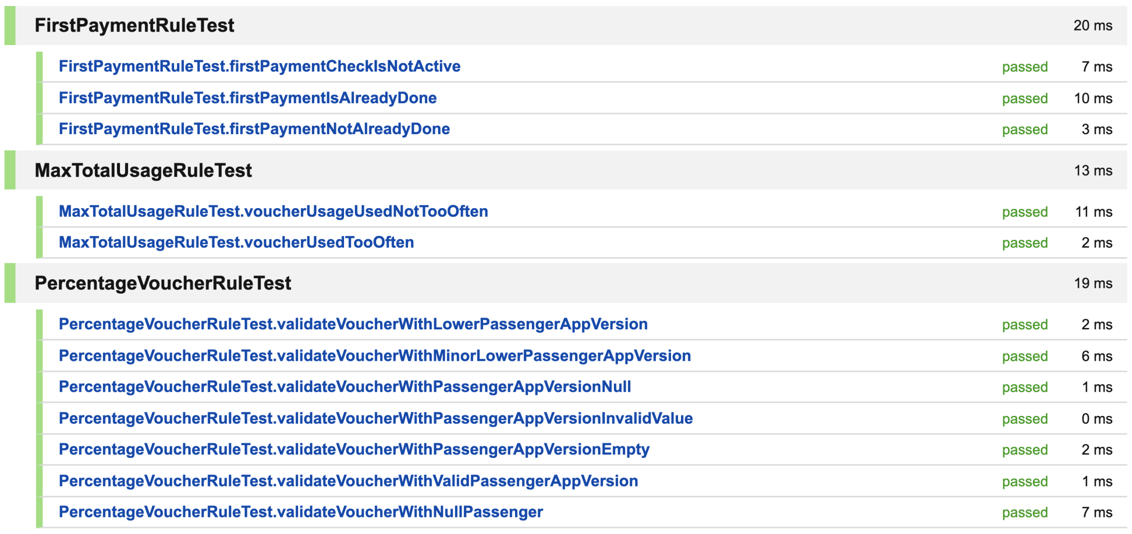
DefaultVoucherValidationServiceTest		30 ms
DefaultVoucherValidationServiceTest.testValidateVoucherWithNullVoucher	passed	12 ms
DefaultVoucherValidationServiceTest.testValidateVoucher	passed	18 ms
BeforeDateRuleTest		6 ms
BeforeDateRuleTest.voucherIsNotOutdated	passed	4 ms
BeforeDateRuleTest.voucherIsOutdated	passed	2 ms
CountryRuleTest		9 ms
CountryRuleTest.testValidateFailed	passed	3 ms
CountryRuleTest.testValidateSuccess	passed	1 ms
CountryRuleTest.testValidateWithAppVersionTooLow	passed	1 ms
CountryRuleTest.testValidateNoCountry	passed	4 ms

Figure 18: Service layer tests Part 1



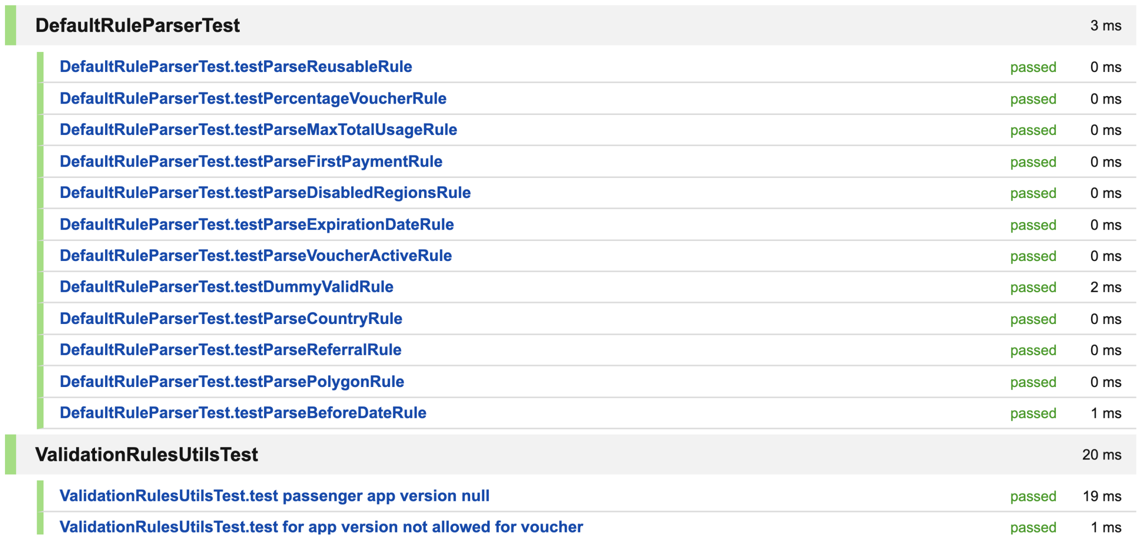
DisabledRegionsRuleTest		48 ms
DisabledRegionsRuleTest.testWithNotDisabledRegions	passed	14 ms
DisabledRegionsRuleTest.testWithDisabledOfficesCountriesHeadquartersOnly	passed	5 ms
DisabledRegionsRuleTest.testWithEmptyLists	passed	4 ms
DisabledRegionsRuleTest.testOfficeListContainsDisabledOffice	passed	6 ms
DisabledRegionsRuleTest.testWithDisabledHeadquarterOnly	passed	6 ms
DisabledRegionsRuleTest.testWithDisabledCountryOnly	passed	7 ms
DisabledRegionsRuleTest.testWithDisabledOfficesOnly	passed	6 ms
ExpirationDateRuleTest		13 ms
ExpirationDateRuleTest.voucherIsValidUsingBookingDateCreated	passed	8 ms
ExpirationDateRuleTest.voucherIsNotExpired	passed	1 ms
ExpirationDateRuleTest.voucherIsExpired	passed	0 ms
ExpirationDateRuleTest.voucherIsInvalidUsingBookingDateCreatedNull	passed	1 ms
ExpirationDateRuleTest.voucherIsNotExpired2	passed	2 ms
ExpirationDateRuleTest.voucherIsExpired2	passed	1 ms

Figure 19: Service layer tests Part 2



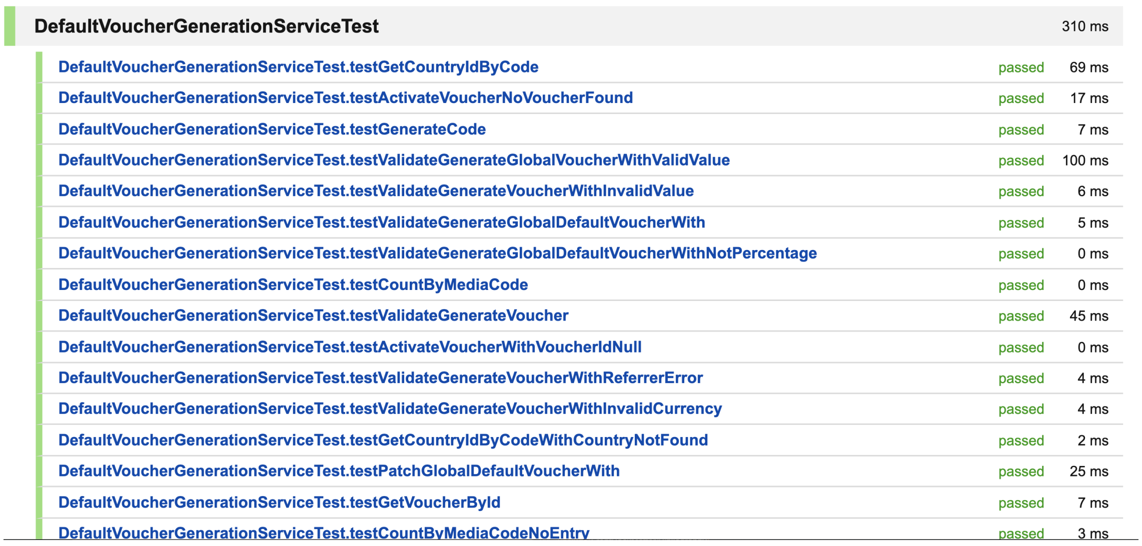
FirstPaymentRuleTest		20 ms
FirstPaymentRuleTest.firstPaymentCheckIsNotActive	passed	7 ms
FirstPaymentRuleTest.firstPaymentIsAlreadyDone	passed	10 ms
FirstPaymentRuleTest.firstPaymentNotAlreadyDone	passed	3 ms
MaxTotalUsageRuleTest		13 ms
MaxTotalUsageRuleTest.voucherUsageUsedNotTooOften	passed	11 ms
MaxTotalUsageRuleTest.voucherUsedTooOften	passed	2 ms
PercentageVoucherRuleTest		19 ms
PercentageVoucherRuleTest.validateVoucherWithLowerPassengerAppVersion	passed	2 ms
PercentageVoucherRuleTest.validateVoucherWithMinorLowerPassengerAppVersion	passed	6 ms
PercentageVoucherRuleTest.validateVoucherWithPassengerAppVersionNull	passed	1 ms
PercentageVoucherRuleTest.validateVoucherWithPassengerAppVersionInvalidValue	passed	0 ms
PercentageVoucherRuleTest.validateVoucherWithPassengerAppVersionEmpty	passed	2 ms
PercentageVoucherRuleTest.validateVoucherWithValidPassengerAppVersion	passed	1 ms
PercentageVoucherRuleTest.validateVoucherWithNullPassenger	passed	7 ms

Figure 20: Service layer tests Part 3



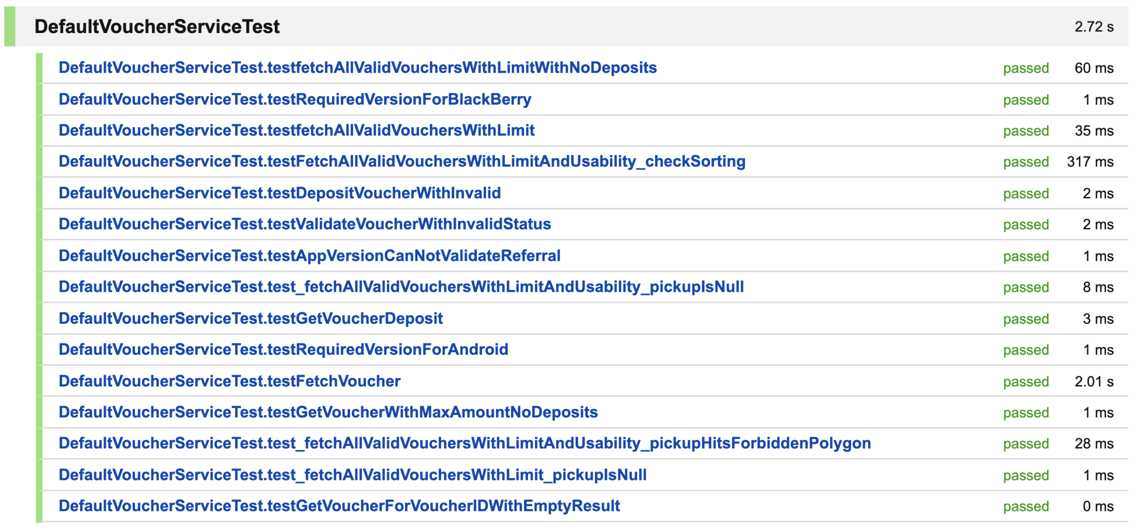
DefaultRuleParserTest		3 ms
DefaultRuleParserTest.testParseReusableRule	passed	0 ms
DefaultRuleParserTest.testPercentageVoucherRule	passed	0 ms
DefaultRuleParserTest.testParseMaxTotalUsageRule	passed	0 ms
DefaultRuleParserTest.testParseFirstPaymentRule	passed	0 ms
DefaultRuleParserTest.testParseDisabledRegionsRule	passed	0 ms
DefaultRuleParserTest.testParseExpirationDateRule	passed	0 ms
DefaultRuleParserTest.testParseVoucherActiveRule	passed	0 ms
DefaultRuleParserTest.testDummyValidRule	passed	2 ms
DefaultRuleParserTest.testParseCountryRule	passed	0 ms
DefaultRuleParserTest.testParseReferralRule	passed	0 ms
DefaultRuleParserTest.testParsePolygonRule	passed	0 ms
DefaultRuleParserTest.testParseBeforeDateRule	passed	1 ms
ValidationRulesUtilsTest		20 ms
ValidationRulesUtilsTest.test passenger app version null	passed	19 ms
ValidationRulesUtilsTest.test for app version not allowed for voucher	passed	1 ms

Figure 21: Service layer tests Part 4



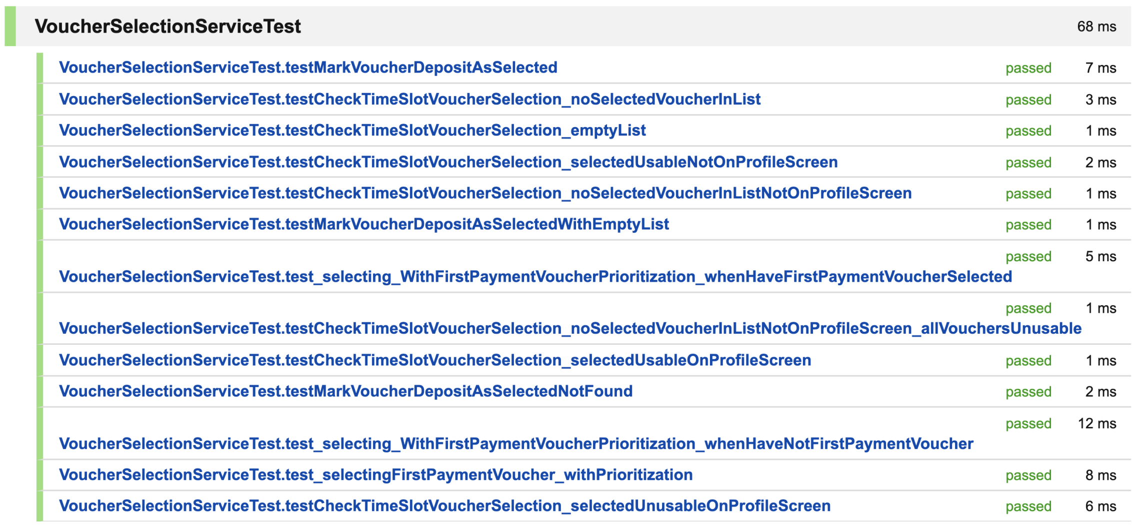
DefaultVoucherGenerationServiceTest		310 ms
DefaultVoucherGenerationServiceTest.testGetCountryIdByCode	passed	69 ms
DefaultVoucherGenerationServiceTest.testActivateVoucherNoVoucherFound	passed	17 ms
DefaultVoucherGenerationServiceTest.testGenerateCode	passed	7 ms
DefaultVoucherGenerationServiceTest.testValidateGenerateGlobalVoucherWithValidValue	passed	100 ms
DefaultVoucherGenerationServiceTest.testValidateGenerateVoucherWithInvalidValue	passed	6 ms
DefaultVoucherGenerationServiceTest.testValidateGenerateGlobalDefaultVoucherWith	passed	5 ms
DefaultVoucherGenerationServiceTest.testValidateGenerateGlobalDefaultVoucherWithNotPercentage	passed	0 ms
DefaultVoucherGenerationServiceTest.testCountByMediaCode	passed	0 ms
DefaultVoucherGenerationServiceTest.testValidateGenerateVoucher	passed	45 ms
DefaultVoucherGenerationServiceTest.testActivateVoucherWithVoucherIdNull	passed	0 ms
DefaultVoucherGenerationServiceTest.testValidateGenerateVoucherWithReferrerError	passed	4 ms
DefaultVoucherGenerationServiceTest.testValidateGenerateVoucherWithInvalidCurrency	passed	4 ms
DefaultVoucherGenerationServiceTest.testGetCountryIdByCodeWithCountryNotFound	passed	2 ms
DefaultVoucherGenerationServiceTest.testPatchGlobalDefaultVoucherWith	passed	25 ms
DefaultVoucherGenerationServiceTest.testGetVoucherById	passed	7 ms
DefaultVoucherGenerationServiceTest.testCountByMediaCodeNoEntry	passed	3 ms

Figure 22: Service layer tests Part 5



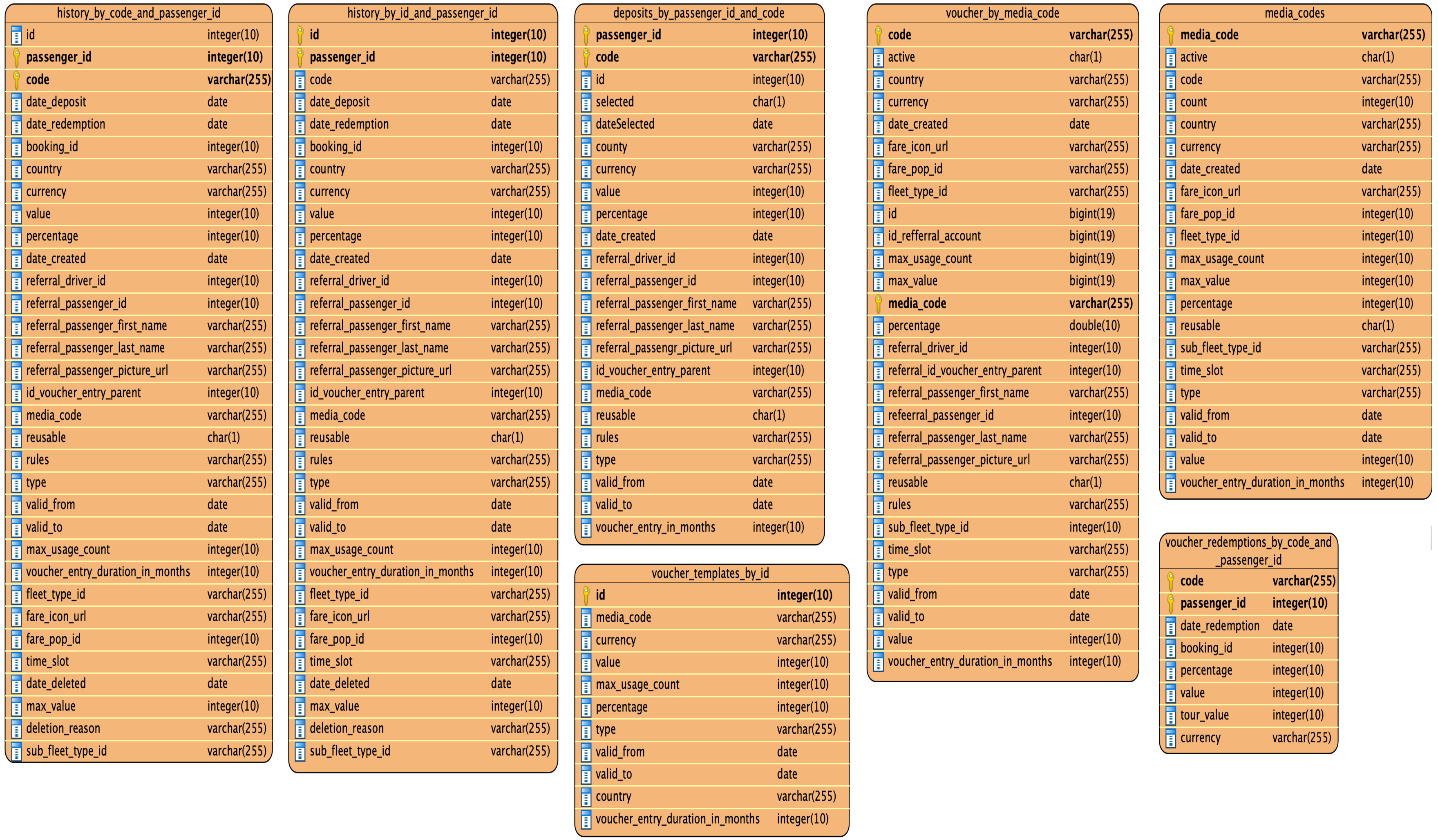
DefaultVoucherServiceTest		2.72 s
DefaultVoucherServiceTest.testfetchAllValidVouchersWithLimitWithNoDeposits	passed	60 ms
DefaultVoucherServiceTest.testRequiredVersionForBlackBerry	passed	1 ms
DefaultVoucherServiceTest.testfetchAllValidVouchersWithLimit	passed	35 ms
DefaultVoucherServiceTest.testFetchAllValidVouchersWithLimitAndUsability_checkSorting	passed	317 ms
DefaultVoucherServiceTest.testDepositVoucherWithInvalid	passed	2 ms
DefaultVoucherServiceTest.testValidateVoucherWithInvalidStatus	passed	2 ms
DefaultVoucherServiceTest.testAppVersionCanNotValidateReferral	passed	1 ms
DefaultVoucherServiceTest.test_fetchAllValidVouchersWithLimitAndUsability_pickupIsNull	passed	8 ms
DefaultVoucherServiceTest.testGetVoucherDeposit	passed	3 ms
DefaultVoucherServiceTest.testRequiredVersionForAndroid	passed	1 ms
DefaultVoucherServiceTest.testFetchVoucher	passed	2.01 s
DefaultVoucherServiceTest.testGetVoucherWithMaxAmountNoDeposits	passed	1 ms
DefaultVoucherServiceTest.test_fetchAllValidVouchersWithLimitAndUsability_pickupHitsForbiddenPolygon	passed	28 ms
DefaultVoucherServiceTest.test_fetchAllValidVouchersWithLimit_pickupIsNull	passed	1 ms
DefaultVoucherServiceTest.testGetVoucherForVoucherIDWithEmptyResult	passed	0 ms

Figure 23: Service layer tests Part 6



VoucherSelectionServiceTest		68 ms
VoucherSelectionServiceTest.testMarkVoucherDepositAsSelected	passed	7 ms
VoucherSelectionServiceTest.testCheckTimeSlotVoucherSelection_noSelectedVoucherInList	passed	3 ms
VoucherSelectionServiceTest.testCheckTimeSlotVoucherSelection_emptyList	passed	1 ms
VoucherSelectionServiceTest.testCheckTimeSlotVoucherSelection_selectedUsableNotOnProfileScreen	passed	2 ms
VoucherSelectionServiceTest.testCheckTimeSlotVoucherSelection_noSelectedVoucherInListNotOnProfileScreen	passed	1 ms
VoucherSelectionServiceTest.testMarkVoucherDepositAsSelectedWithEmptyList	passed	1 ms
VoucherSelectionServiceTest.test_selecting_WithFirstPaymentVoucherPrioritization_whenHaveFirstPaymentVoucherSelected	passed	5 ms
VoucherSelectionServiceTest.testCheckTimeSlotVoucherSelection_noSelectedVoucherInListNotOnProfileScreen_allVouchersUnusable	passed	1 ms
VoucherSelectionServiceTest.testCheckTimeSlotVoucherSelection_selectedUsableOnProfileScreen	passed	1 ms
VoucherSelectionServiceTest.testMarkVoucherDepositAsSelectedNotFound	passed	2 ms
VoucherSelectionServiceTest.test_selecting_WithFirstPaymentVoucherPrioritization_whenHaveNotFirstPaymentVoucher	passed	12 ms
VoucherSelectionServiceTest.test_selectingFirstPaymentVoucher_withPrioritization	passed	8 ms
VoucherSelectionServiceTest.testCheckTimeSlotVoucherSelection_selectedUnusableOnProfileScreen	passed	6 ms

Figure 24: Service layer tests Part 7



history_by_code_and_passenger_id	
id	integer(10)
passenger_id	**integer(10)**
code	**varchar(255)**
date_deposit	date
date_redemption	date
booking_id	integer(10)
country	varchar(255)
currency	varchar(255)
value	integer(10)
percentage	integer(10)
date_created	date
referral_driver_id	integer(10)
referral_passenger_id	integer(10)
referral_passenger_first_name	varchar(255)
referral_passenger_last_name	varchar(255)
referral_passenger_picture_url	varchar(255)
id_voucher_entry_parent	integer(10)
media_code	varchar(255)
reusable	char(1)
rules	varchar(255)
type	varchar(255)
valid_from	date
valid_to	date
max_usage_count	integer(10)
voucher_entry_duration_in_months	integer(10)
fleet_type_id	varchar(255)
fare_icon_url	varchar(255)
fare_pop_id	integer(10)
time_slot	varchar(255)
date_deleted	date
max_value	integer(10)
deletion_reason	varchar(255)
sub_fleet_type_id	varchar(255)

history_by_id_and_passenger_id	
id	**integer(10)**
passenger_id	**integer(10)**
code	varchar(255)
date_deposit	date
date_redemption	date
booking_id	integer(10)
country	varchar(255)
currency	varchar(255)
value	integer(10)
percentage	integer(10)
date_created	date
referral_driver_id	integer(10)
referral_passenger_id	integer(10)
referral_passenger_first_name	varchar(255)
referral_passenger_last_name	varchar(255)
referral_passenger_picture_url	varchar(255)
id_voucher_entry_parent	integer(10)
media_code	varchar(255)
reusable	char(1)
rules	varchar(255)
type	varchar(255)
valid_from	date
valid_to	date
max_usage_count	integer(10)
voucher_entry_duration_in_months	integer(10)
fleet_type_id	varchar(255)
fare_icon_url	varchar(255)
fare_pop_id	integer(10)
time_slot	varchar(255)
date_deleted	date
max_value	integer(10)
deletion_reason	varchar(255)
sub_fleet_type_id	varchar(255)

deposits_by_passenger_id_and_code	
passenger_id	**integer(10)**
code	**varchar(255)**
id	integer(10)
selected	char(1)
dateSelected	date
county	varchar(255)
currency	varchar(255)
value	integer(10)
percentage	integer(10)
date_created	date
referral_driver_id	integer(10)
referral_passenger_id	integer(10)
referral_passenger_first_name	varchar(255)
referral_passenger_last_name	varchar(255)
referral_passengr_picture_url	varchar(255)
id_voucher_entry_parent	integer(10)
media_code	varchar(255)
reusable	char(1)
rules	varchar(255)
type	varchar(255)
valid_from	date
valid_to	date
voucher_entry_in_months	integer(10)

voucher_templates_by_id	
id	**integer(10)**
media_code	varchar(255)
currency	varchar(255)
value	integer(10)
max_usage_count	integer(10)
percentage	integer(10)
type	varchar(255)
valid_from	date
valid_to	date
country	varchar(255)
voucher_entry_duration_in_months	integer(10)

voucher_by_media_code	
code	**varchar(255)**
active	char(1)
country	varchar(255)
currency	varchar(255)
date_created	date
fare_icon_url	varchar(255)
fare_pop_id	varchar(255)
fleet_type_id	varchar(255)
id	bigint(19)
id_refferral_account	bigint(19)
max_usage_count	bigint(19)
max_value	bigint(19)
media_code	**varchar(255)**
percentage	double(10)
referral_driver_id	integer(10)
referral_id_voucher_entry_parent	integer(10)
referral_passenger_first_name	varchar(255)
refeerral_passenger_id	integer(10)
referral_passenger_last_name	varchar(255)
referral_passenger_picture_url	varchar(255)
reusable	char(1)
rules	varchar(255)
sub_fleet_type_id	integer(10)
time_slot	varchar(255)
type	varchar(255)
valid_from	date
valid_to	date
value	integer(10)
voucher_entry_duration_in_months	integer(10)

media_codes	
media_code	**varchar(255)**
active	char(1)
code	varchar(255)
count	integer(10)
country	varchar(255)
currency	varchar(255)
date_created	date
fare_icon_url	varchar(255)
fare_pop_id	integer(10)
fleet_type_id	integer(10)
max_usage_count	integer(10)
max_value	integer(10)
percentage	integer(10)
reusable	char(1)
sub_fleet_type_id	varchar(255)
time_slot	varchar(255)
type	varchar(255)
valid_from	date
valid_to	date
value	integer(10)
voucher_entry_duration_in_months	integer(10)

voucher_redemptions_by_code_and _passenger_id	
code	**varchar(255)**
passenger_id	**integer(10)**
date_redemption	date
booking_id	integer(10)
percentage	integer(10)
value	integer(10)
tour_value	integer(10)
currency	varchar(255)

Figure 25: Cassandra database with all attributes

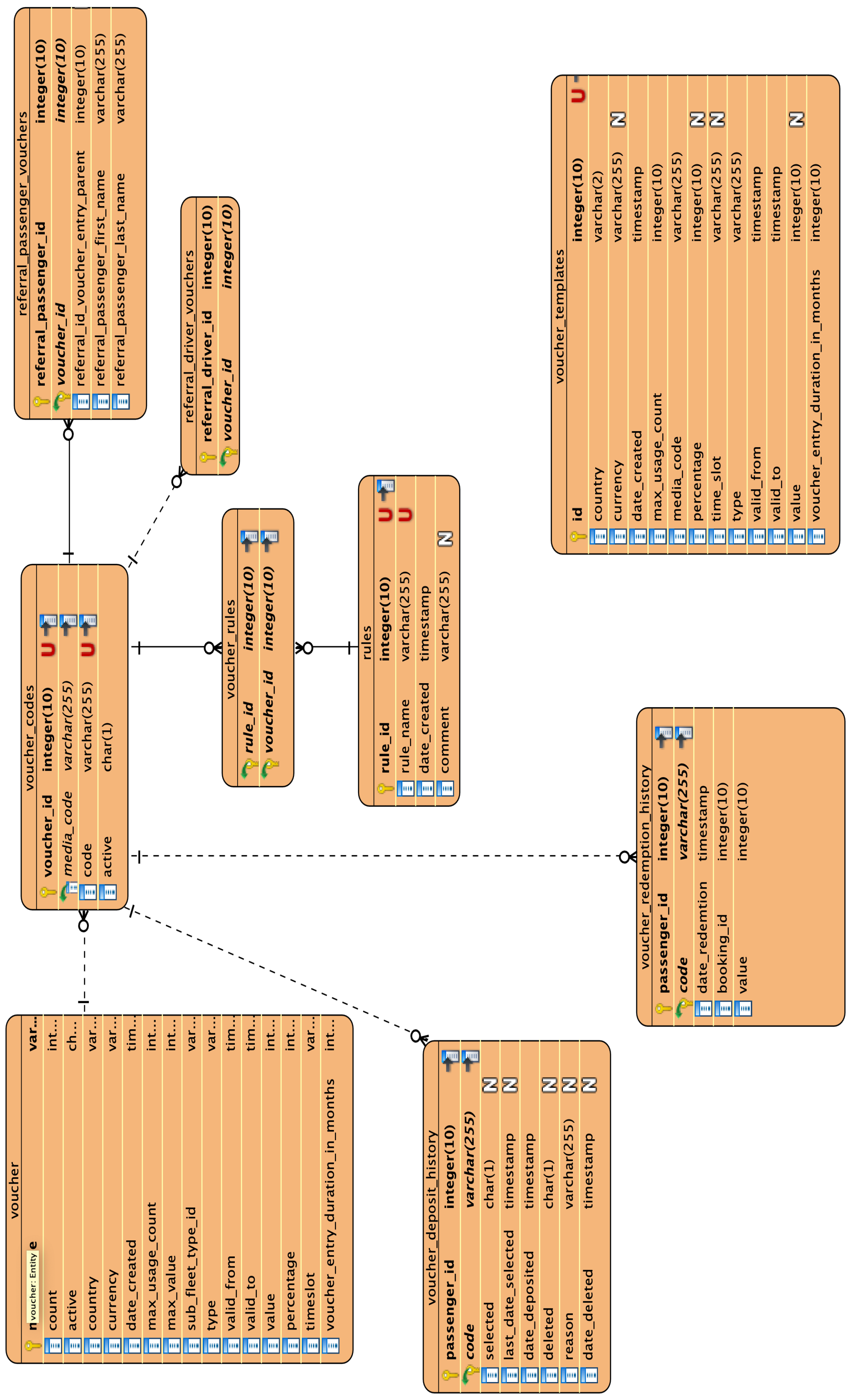


Figure 26: Postgres database with all attributes

```java
public class VoucherCodesRepository {
    private VoucherByCodeRepositoryPostgres
        voucherByCodeRepositoryPostgres;
    private VoucherByCodeRepositoryCassandra
        voucherByCodeRepositoryCassandra;
    private VoucherCodesMapper voucherCodesMapper;
    private VoucherGenerationConfig config;

    public List<VoucherByCode> findByCode(final String code) {
        Long readRatio = config.getCassandraPostgresReadRatio();

        if (isCassandra(readRatio)) {  return
            voucherByCodeRepositoryCassandra.findByCode(code); }
        else {
            return voucherByCodeRepositoryPostgres.findAllByCode
                (code)
                .stream()
                .map(voucherCodesMapper::mapVoucherCodes)
                .collect(Collectors.toList()); } }

    public void save(VoucherByCode voucherByCode)  {
        boolean isCassandraEnabled = config.isCassandraEnabled()
            ;
        boolean isPostgresEnabled = config.isPostgresEnabled();
        if (isCassandraEnabled) {
            voucherByCodeRepositoryCassandra.save(voucherByCode)
                ;
        }
        else if (isPostgresEnabled) {
            voucherByCodeRepositoryPostgres.save(
                voucherCodesMapper.mapVoucherByCode(
                voucherByCode));
        }
    }

    private boolean isCassandra(Long ratio) {
        return (getRandomNumberInRange() <= ratio); }

    private static int getRandomNumberInRange() {
        return (int) (Math.random() * ((100) + 1)); }}
```

Listing 6: Full example with voucher codes tables

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name:	______________________________
Vorname:	______________________________

dass ich die vorliegende Bachelorarbeit mit dem Thema:

Migration of NoSQL (Cassandra) to relational database (Postgres) on high demanded distributed system

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

____________	____________	______________________
Ort	Datum	Unterschrift im Original