Software Engineering 2
CLup project by
Robert Medvedec
Toma Sikora

POLITECNICO
MILANO 1863

Implementation and Test deliverable

Deliverable:	ITD
Title:	Implementation and Test deliverable
Authors:	Robet Medvedec, Toma Sikora
Version:	1.0
Date:	7-February-2021
Download page:	https://github.com/robertodavinci/Software_Engineering_2_Project_Medvedec_Sikora/Implementation/Clup
Copyright:	Copyright © 2021, R. Medvedec, T. Sikora – All rights reserved

Contents

List of Figures

List of Tables

1 Introduction

1.1 Purpose

This document provides a detailed view of the architecture and the implementation of the CLup system. Based on both RASD and DD documents provided in this project, that can both be found on the above provided GitHub page, this document specifies the development process and used frameworks and philosophies, as well as explains the code structure and design rationale behind the whole system.

The system implementation is done as an Android app that is available for every Android device that supports Android 8.0 and above.

Detailed code can be found on provided GitHub page and it can also be imported as an Android Studio project in order to make adjustments to the app.

1.2   Scope

CLup is a simple application that helps store managers with handling large crowds inside their store and store customers with planning more efficient and safe grocery shops. The target audience for this application includes every person that shops for groceries in a store, so all demographics fall into this category.

Faced with a worldwide pandemic of the COVID-19 virus countries across the world imposed strict health measures in line with the recommendations of the WHO. To combat the spread of the virus, governments introduced decrees that limited the movement of the population to a certain degree. Only essential movement, such as: going to work, grocery shopping or outdoor exercise, was deemed acceptable. Although successful in the mitigation of the disease, the act put a serious strain on society on many levels. To help reduce the stress and anxiety, many aspects of everyday life involving close contact can be considered and improved upon.

This project aims to help with, and resolve the issues surrounding grocery shopping. As we all know, grocery shopping is an essential activity which involves close contact inside the store. Since the COVID-19 virus spreads mainly through airborne particles, this activity plays a key role in its mitigation. To reduce crowding inside the stores, supermarkets need to restrict access to their store and keep the number of people inside below the optimal maximum capacity.

The main idea is to enable store customers to enter a queue from home (or wherever they find themselves) through simple interaction with the application.

1.3   Definitions, Acronyms, Abbreviations

1.3.1   Definitions

- **Application**: a computer (mobile) program that is designed for a particular purpose.

- **QR code**: a machine-readable code consisting of an array of black and white squares, typically used for storing URLs or other information for reading by the camera or a scanner.

- **Smartphone**: a mobile phone that performs many of the functions of a computer, typically having a touchscreen interface, internet access, and an operating system capable of running downloaded apps.

- **Google Maps**: a web mapping service developed by Google, used both as a standalone app and as an integrated mapping solution in most of the apps.

- **Android**: most popular operating system for smartphones and tablets, developed by Google and partners.

- **Firebase**: platform created by Google for creating mobile and web applications, can be used as the database.

1.3.2   Acronyms

- **RASD**: Requirement Analysis and Specification Document

- **COVID-19**: Virus responsible for the spread of the coronavirus disease 2019

- **CLup**: Customer Line-up

- **API**: Application programming interface, computing interface which defines interactions between multiple software intermediaries

- **WHO**: World Health Organization

- **GUI**: Graphical user interface

- **DB**: Database

- **AES**: Advanced Encryption Standard

1.3.3   Abbreviations

- **Gn**: nth goal.

- **Rn**: nth functional requirement.

- **App**: Application.

1.4   Revision History

- **Version 1.0**: First .tex document created and added all together; 7th February 2021

1.5   Reference Documents

- Specification document "R&DD Assignment A.Y. 2020-2021.pdf"

- Specification document "Implementation Assignment A.Y. 2020-2021.pdf"

- Presentations Software Engineering 2, Politecnico di Milano

- Star UML - Program used for creating diagrams

- Fundementals of Software Engineering - C. Ghezzi, M. Jazayeri, D. Mandrioli

1.6   Document Structure

The implementation and test deliverable document is divided into six main chapters. The chapters are: Introduction, Requirements, Adopted Development Frameworks, Code Structure, Testing, and Installation Instructions. Each one of them encapsulates a specific aspect of the development of an application based on the RASD and DD documents released earlier. In a few paragraphs the main focuses of each of the chapters is presented.

The first chapter provides an introduction to the matter at hand. Firstly, it recapitulates the purpose of the document and scope of the project. Secondly, it explains the most important definitions, acronyms, and abbreviations. And lastly, it provides an overview of the document revision history, reference documents, and this document as a whole.

The second chapter revolves around the requirements of the project. The chapter introduces the main requirements in mind when the implementation took place and lists them by importance. It explains why each one is important, what are the consequences of its implementation, and argues on the implementation decisions on each and every one of them. A strong focus is also on the relationship between the requirements of the earlier documents vs the requirements of the implementation.

The third chapter gives a thorough overview of the adopted development frameworks. It is further divided based on the programming languages and frameworks, additional algorithms and middleware, and the database model. Each decision is presented with both its advantages and flaws, and the choice is argumented.

The fourth chapter gives a precise account of the code structure of the specific implementation. All used classes, interfaces, and enumerations used in the implementation are listed and explained. Furthermore, some of the more important code samples are also provided.

The fifth chapter revolves around the testing done on the implementation. The testing wad done in three phases: unit testing, UI testing, and security testing. Some inconveniences in testing connected to the former choices are also explained.

The sixth chapter contains the installation instructions of the apk provided with this document.

2 Requirements

2.1 Idea of requirements implementation

This version of the application is not meant for the mass market. It is rather a rough concept and a representation of the design ideas presented in the previous documents. Whilst most of the main functionalities are implemented and are pretty much ready to go (at least on a smaller scale), some of the more detailed functionalities are not completely implemented, if they are implemented at all.

Some of the design ideas of the app are also based on the fact that this will not be used on a mass scale. For a version that would go on the market, some things would be adapted and changed and some other technologies might be considered. More on these topics in the following chapters.

2.2 Original project requirements

The original idea of CLup system required the following functionalities:

- Simple to use

- Support majority of devices

- Used by both customers and store employees

- **User features**

  – Selecting a desired store from the list of supported stores

  – Requesting a virtual ticket

  – Booking a visit

  – Getting an estimation of travel time to the store and waiting time

  – Notifying customers about their ticket state

- **Store features**

  – Opening and closing a virtual store

  – Controlling the influx of customers to the store

  – Managing queue of customers

  – Ticket scanning mechanism

2.3   Requirements implementation

The following functionalities are implemented into this version of CLup system:

- **Simple to use**
  Selected app design consists of only a few colors that are providing maximal simplicity and clarity when using the app. Every time an application takes more than a second to load a certain data, a loading screen is shown to ensure that the user is aware of the loading action. Every action in the app is logical and prevents the user from doing some irreversible actions or something they would not like to do in that situation. Finally, app is considerably fast and provides clear instructions on main actions at all times. No additional menus or hidden buttons are implemented that could possibly confuse the user.

- **Support majority of devices**
  App is built for all Android devices (tablets and phones) that support Android version 8.0 (Oreo) or above. Since application only has sense to be used on a mobile or tablet device, no desktop application has been developed. Android mobile phones take up around 72% of all mobile devices worldwide, with the Android 8.0 version or above being around 91% of that number, which means that this app covers around 65% of all mobile phones. This number could easily be brought to well over 90% with an iOS version of the app which would be relatively easy to port and adapt to iPhones and iPads. The idea of making only an Android version was due to the fact that the main functionalities would work exactly the same on both operating systems and only one version is sufficient to showcase the idea and design rationale of the project.

- **Used by both customers and store employees**
  The same app is used for both requesting a ticket and controlling the store customer influx. Since only store managers that are in control of the store have user accounts, it was easy to separate those two functionalities of the app. Customers don't require to make accounts and they can use the app simply by downloading and installing it on their phones.

- **Selecting a desired store from the list of supported stores**
  A simple drop down menu of the stores is located in the main part of the app. To simplify the certain store search progress, stores have been arranged by both the city and the store name. Customer firstly selects the city of the desired store and then the store chain name, after which a list of all stores with the same name in that city are displayed along with their addresses, making searching for the store easy and fast.

- **Requesting a virtual ticket**
  After selecting a store, customer can request a virtual ticket with a simple click. Before generating a ticket there will be a number shown indicating how many customers are currently in queue for that exact store, allowing customer to plan his arrival. After the customer gets his ticket, he can check the ticket state by simple press of a button and see whether they are in queue or whether they can enter the store. After the ticket has been activated, customers have 5 minutes to

enter the store before the ticket expires, which would force customers to request another ticket.

- **Opening and closing a virtual store**
Store managers have certain stores connected to them and they can control both the influx of customers and the store opening/closing. A simple push of a button can used to open the store and allow ticket requests, while the same can be done with the closing of the store, which deletes all the tickets that are currently in queue or active.

- **Controlling the influx of customers to the store**
Influx control is done with two main functions - ticket scanning and store exit registration. Every time a valid ticket has been scanned, the ticket is immediately invalidated and a customer is allowed to go into store. Store attributes regarding current occupants and maximum number of occupants are updated, allowing the store manager to see the store status and availability at all times. A store exit is registered with a simple button press, without scanning in order to speed up the process and reduce the number of close encounters between the store manager and the customer, allowing another customer that is currently in queue to enter the store.

- **Managing queue of customers** Queue managing has been explained in the requirement above.

- **Ticket scanning mechanism** Ticket scanning is done using a simple QR code scanner that is encoded. After its decoding, the ticket is validated and the customer is either allowed or denied entrance to the store. QR code scanning is done with the phone camera so no additional hardware is required.

The following functionalities are NOT implemented into this version of CLup system:

- **Booking a visit** This feature is not required to be implemented in the groups of two. However, it can easily be added as most of the functionalities are written, like timeslots, and this implementation would only require minor adjustments.

- **Getting an estimation of travel time to the store and waiting time** In our specification documents, this features was solely meant to be used with "Book a visit" feature, which is the reason it is not implemented in this version. The main reason for excluding waiting time estimation was the need for the tickets to be scanned twice in order to get the average shopping time data, thus increasing the possibility of spreading the disease. This can also easily be implemented if a greater need for this feature is shown. We have replaced it with the number of customers in front that can also help the customer to assume the average wait time.

- **Notifying customers about their ticket state** Same as the example above, we decided to exclude the notifications to keep simplicity of the app since requesting a ticket without reservation is mainly done when being close to the store, due to ticket's short lifespan in order to keep the queue going fast. This requirement can easily be added together with "Book a visit" feature since adding push notifications is not a large task.

3 Adopted Development Frameworks

3.1 Adopted programming languages and frameworks

The CLup app as already mentioned has been developed in Android Studio, an IDE for building Android apps based on IntelliJ IDEA software from JetBrains. Android Studio can be used with three programming languages - Kotlin, Java, and C++ , all used for object oriented programming of the app.
Our choice was to go with Java since that's the language we're most familiar with and a language that exists a lot longer than Kotlin, which is Google's preferred language for Android app dev, so more examples can be found online for easier referencing.
Java also offers vast and detailed documentation as well as similar syntax to some other more popular languages, contrary to Kotlin.
Android Studio also uses an internal UI design tool, which is based on XML language that uses layouts, items, and resources for designing and creating different app screens. It features a real-time design screen making an UI design a lot smoother and faster experience.
During the development we have used a Google Pixel 3 virtual machine, that is also integrated in Android Studio, for testing and other app-related purposes.
Several phones of different sizes, vendors, and Android versions have been used for real-life testing, and those are: Samsung Galaxy Note 9, Xiaomi Redmi Note 9 Pro, and Huaweii P Smart.
Even though our original plan was to use MySQL database for the backend of this project, we have decided to use Google's Firebase that was mentioned as an alternative in the Design document. More details on this design choice in the sections below.

3.2 Adopted additional algorithms and middleware

One of the APIs that was planned to be used, Google Maps API, did not make the cut in this version. The reasoning behind that is already explained in the previous chapter and it touches on the lack of "Book a visit" feature that was not required to be made for smaller groups. Using this API would further complicate things and we did not find it necessary just for the ticket queueing function of the app, since this function would mostly be used on the fly and without too much planning, which would mean that the customers would already know to which store they are going and how to get there.

One of the main challenges of the system was securing a proper encryption for the QR code which is to be scanned by the store manager. If we put no encryption on it, the customers can easily scan the code on their own, see how it looks, and the create their own QR codes to skip lines and get faster access to the store.
For that reason we have decided to use AES or Advanced Encryption Standard which is a specification for electronic data encryption. It is one of the most widely used standards in the world and regarded as one of the most secure.

This standard uses either 128, 192, or 256 bit keys to encode and decode data. It is also symmetrical, making it easier to do encryption/decryption without exchanging

keys.

The way AES works is that it takes plaintext and a key, encodes it using a special algorithm, and then sends the ciphertext to the destination. The receiver receiver the ciphertext and decodes it using the same key, getting the original message.
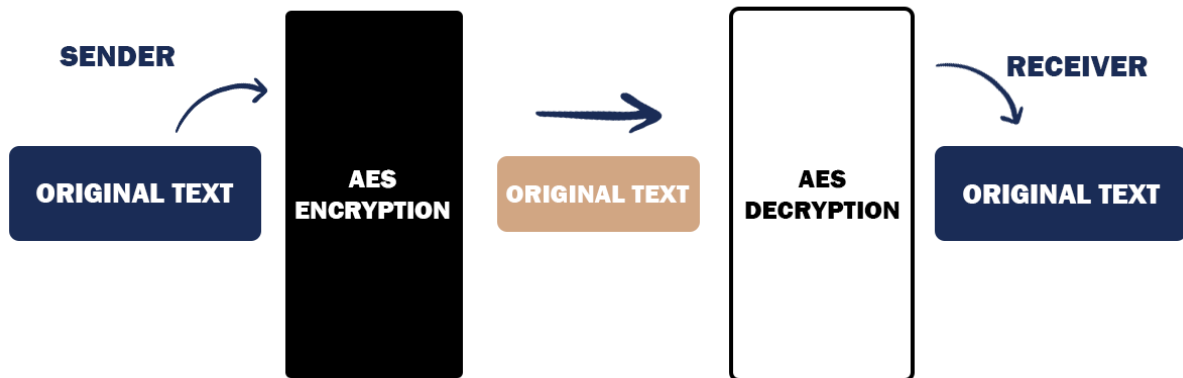
Figure 1: **AES sketch**

Each store in our database has a unique 128 bit key with mixed characters. This key is used for encryption when requesting a ticket and then for decryption when scanning a ticket. The key is not sent at any moment over the network and it's not shown to the customer or the store manager at any time, making the app very secure in that regard. If user tries to scan the QR code, they will get something like this:

"-112w-22w34w-41w34w-120w2w-22w87w-93w-41w65w11".

Good luck with trying to get any data from that! Even NSA uses it. Our encryption code generates random signed 8 bit integers which are then interleaved with a letter, "w" in this scenario, for easier decryption later. The code for the algorithm is provided here.

Entities - StrongAES

```java
// StrongAES.java
package com.example.clup;

import java.security.Key;
import javax.crypto.Cipher;
import javax.crypto.spec.SecretKeySpec;
public class StrongAES
{
    // function that Encrypts and Decrypts data
    public byte[] AESEncrypt(String plaintext, String key){
        try {
```

```java
            Key aesKey = new SecretKeySpec(key.getBytes(), "AES");
            Cipher cipher = Cipher.getInstance("AES");
            // encrypt the text
            cipher.init(Cipher.ENCRYPT_MODE, aesKey); // encrypted with a key
            byte[] cy = cipher.doFinal(plaintext.getBytes()); // encryption is
                returned in bytes[] array
            //System.out.println(cy);
            return cy;
        }
        catch(Exception e){
            //return 'D';
            return "false".getBytes();
        }
    }

    public String AESDecrypt(byte[] cyphertext, String key){
        try {
            Key aesKey = new SecretKeySpec(key.getBytes(), "AES");
            Cipher cipher = Cipher.getInstance("AES");
            cipher.init(Cipher.DECRYPT_MODE, aesKey); // decrypted with a key
            String cy2 = new String(cipher.doFinal(cyphertext)); // decryption
                is returned in String
            //System.out.println(cy2);
            return cy2;
        }
        catch(Exception e){
            System.err.println(e);
            return "Error";
        }
    }
}
```

3.3   Database model

Google's Firebase is a very powerful databse tool for two reason - it is very simple and it's easily integrated with Android apps. It is not a standard database tool that uses SQL language (it is NoSQL), but rather separates its data by using JSON structures and a classic key-value methodology. Every item can have a key and a value, as well as be a parent and have multiple children. However, any item that has a child or children cannot have any values, but is rather just represented with a key.

This all makes extracting data very simple, but it generates a problem with data multiplication since there is no way of connecting some items with a pointer or by id. What must be done is a manual data copying, which increases size and complexity of the structure.

Android Studio and the whole Android environment has already built in functions that make working with Firebase much easier, which was the main reason for this

selection. Functions for user registration and login are already set up and are super secure, not even requiring storing user password in the database.

The whole Firebase interface is also very well done and allows maximum customization as well as adaptability to the certain project. It is located on a Google server, so there is no need to worry about any other sever that runs a normal database, since Google servers are always up and have very small limitations, at least for a smaller scale project.

You can see the way we have structured our database in the following examples.

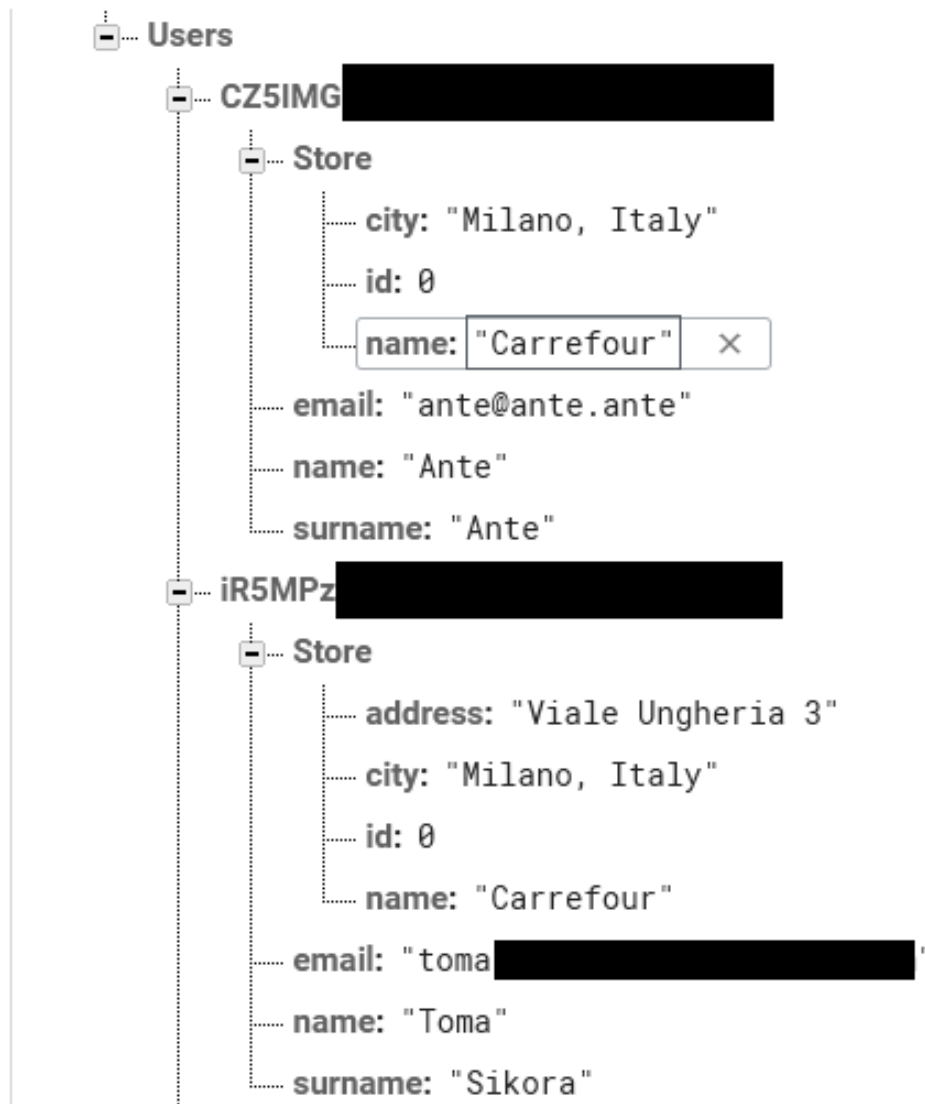Figure 2: **Our Firebase database 1**

Figure 3: **Our Firebase database 2**

The two main items are "Stores" and "Users". Stores are ordered by cities, then by store names, and finally by id. Each store has an address, a key, a current ticket id, a maximum number of customers, an occupancy number, and an open indicator. Also, each store has another set of children "Tickets", which is created dynamically.

Each user is defined by "UserID" which is a Firebase identification. Connected to the user are a specific Store, which here only consists of a city, an ID, and a name, so we can reference it in the other "Store" structure, as well as an email, a name, and a surname, with the latter two being used only for registration, which has been omitted from the final app version. Therefore store managers and stores are added manually for additional security and data consistency.

The main Firebase issue is the data retrieval - sometimes the servers take up to 5

seconds to return simple data! In this case it is not a major issue, since the speed is not the most important function of the application, but the server dependability of getting the data back on time can cause some problems.

All in all, in case of a a real-world app for a much larger scale, we would probably transfer to MySQL mainly because of speed and stability which are not strong sides of Firebase.

4   Code Structure

4.1   Classes, Interfaces, and Enumerations

The structure of the code is similar to that explained in both RASD and DD documents. The app communicates with the database through controllers and services, and the database returns the data back.

Since there is no backend on Firebase database, we've changed our design rationale from DD from thin-client to fat-client. This way pretty much all of the work is done by the application, with the database being only used for storing and loading data. Since the operations are very simple, this provides no issues to the functionality of the app. After adding "Book a visit" feature, app would get significantly more demanding and a switch to another database with backend implementation would be welcome. However, this version of the app only takes a few megabytes of space and is very fast even on older phones, with the only decrease of speed being because of the already mentioned Firebase data delay.

One of the main challenges was to sync the synchronous application with the asynchronous database model. This meant making some design decisions that will both look good and feel right when using the app. The app is expected to be fast and consistent, which is something our database can't guarantee, so getting those two in line was a bit of an issue.

Most of the work here is done by using listeners - functions that work on another thread and wait for the data from the database. This allows the app to keep working properly on one thread without waiting for the data. When data arrives, app updates the data that is on the screen. When the wait is too long, that is more than a second, loading screens are introduced to keep the dynamic feel of the app.

The source code division is following:

- **Entities**

    - ApplicationState
    - Store
    - StoreManager
    - Ticket
    - TicketState (ennumeration)
    - Timeslot
    - User
    - UserType (ennumeration)

- **Services**

    - DatabaseManagerService (interface)

- DirectorService (interface)
- EnterService (interface)
- ExitService (interface)
- LoginManagerService (interface)
- QueueService (interface)
- StoreSelectionManagerService (interface)
- TicketService (interface)
- **Implementation**
  - * DatabaseManager
  - * Director
  - * LoginManager
  - * RequestManager
  - * StoreManager
  - * StoreSelectionManager

- **Controllers**

  - CustomerController
  - EncryptionService (not used in this version, encryption has been directly implemented in other classes)
  - ForgotPasswordController
  - HomeController
  - LoginController
  - PreLoginController
  - QrController
  - RegisterController (not used in this version)
  - ScannerController
  - StoreController
  - StoreManagerController
  - StrongAES
  - TicketController
  - UserProfileController (not used in this version)

- **Listeners**

  - OnCheckTicketListener
  - OnCredentialCheckListener
  - OnGetDataListener
  - OnGetTicketListener
  - OnGetTimeslotListener

> – OnTaskcompleteListener

All of the files are classes besides the ones that are described differently. Every controller has an additional *activity_controllername.xml* file that defines the design of an app page on the phone.

4.2 Code examples

Here are provided some code examples for the components. At least one component from each section is provided to give an example how the rest of the components from that section look.

Entities - Store

```java
// Store.java
package com.example.clup.Entities;

public class Store {
    public String name, address, city; // variables that define each Store
    public int maxNoCustomers, id; // variables that define each Store

    // Store constructors
    public Store(){}
    public Store(int id, String name, String city){
        this.id = id;
        this.name = name;
        this.city = city;
    }
    public Store(int id, String name, String address, String city){
        this.id = id;
        this.name = name;
        this.address = address;
        this.city = city;
    }
    public Store(int id, String name, String address, String city, int
        maxNoCustomers){
        this.id = id;
        this.name = name;
        this.address = address;
        this.city = city;
        this.maxNoCustomers = maxNoCustomers;
    }
    // Store getters
    public String getAddress() {
        return address;
    }
    public String getCity() {
        return city;
    }
    public String getName() {
```

```java
        return name;
    }
    public int getId() {
        return id;
    }
    public int getMaxNoCustomers() {
        return maxNoCustomers;
    }
}
```

Entities - TicketService

```java
// TicketService.java
package com.example.clup.Services;

import com.example.clup.Entities.Store;
import com.example.clup.Entities.Ticket;
import com.example.clup.OnCheckTicketListener;
import com.example.clup.OnGetDataListener;
import com.example.clup.OnGetTicketListener;
import com.example.clup.OnTaskCompleteListener;

public interface TicketService {
    public void getTicket(Store store, OnGetTicketListener
        onGetTicketListener);
    public void checkTicket(Ticket ticket, OnCheckTicketListener
        onCheckTicketListener);
    public void checkQueue(Store store, OnCheckTicketListener
        onCheckTicketListener);
    public void cancelTicket(Store store, Ticket ticket,
        OnTaskCompleteListener onTaskCompleteListener);
}
```

Entities - RequestManager

```java
// RequestManager.java
package com.example.clup.Services.Implementation;

import com.example.clup.Entities.Store;
import com.example.clup.Entities.Ticket;
import com.example.clup.Entities.TicketState;
import com.example.clup.Entities.Timeslot;
import com.example.clup.OnCheckTicketListener;
import com.example.clup.OnGetDataListener;
import com.example.clup.OnGetTicketListener;
import com.example.clup.OnGetTimeslotListener;
import com.example.clup.OnTaskCompleteListener;
import com.example.clup.Services.DatabaseManagerService;
import com.example.clup.Services.QueueService;
import com.example.clup.Services.TicketService;
import com.google.firebase.database.DataSnapshot;
```

```java
import com.google.firebase.database.DatabaseError;

import java.sql.Time;
import java.sql.Timestamp;
import java.util.ArrayList;
import java.util.List;

public class RequestManager implements QueueService, TicketService {
    private StoreSelectionManager storeSelectionManager;
    private DatabaseManager databaseManager = DatabaseManager.getInstance();

    // Average waiting time and tickets lists have not been implemented in
        this version, as well as timeslots
    // This can be used as an template for implementing "Book a visit" feature
    List<Ticket> tickets;
    //TODO
    private int averageMinutesInStore = 15, maxId = -1;

    // retrieves Ticket and sets its state based on other Store attributes
    @Override
    public void getTicket(Store store, OnGetTicketListener
        onGetTicketListener) {
        //System.out.println("Get ticket rm");
        maxId = -1;
        databaseManager.getStore(store, new OnGetDataListener() {
            @Override
            public void onSuccess(DataSnapshot dataSnapshot) {
                if
                    (Integer.parseInt(dataSnapshot.child("open").getValue().toString())
                    == 0) {
                    onGetTicketListener.onFailure();
                    return;
                }
                maxId =
                    Integer.parseInt(dataSnapshot.child("maxId").getValue().toString());
                Ticket ticket = new Ticket(maxId + 1, store);
                int occupancy =
                    Integer.parseInt(dataSnapshot.child("occupancy").getValue().toString());
                int maxNoCustomers =
                    Integer.parseInt(dataSnapshot.child("maxNoCustomers")
                .getValue().toString());
                int activeTickets = 0;
                for (DataSnapshot i :
                    dataSnapshot.child("Tickets").getChildren()) {
                    if
                        (i.child("ticketState").getValue().toString().equals("ACTIVE"))
                        activeTickets++;
                }
                if (occupancy + activeTickets < maxNoCustomers) {
                    ticket.setTicketState(TicketState.ACTIVE);
```

```java
                    ticket.setTimeslot(new Timeslot(new
                        Timestamp(System.currentTimeMillis() + 1000 * 60 * 5)));
                        // wait for customer 5 mins
                } else {
                    ticket.setTicketState(TicketState.WAITING);
                    ticket.setTimeslot(new Timeslot(new Timestamp(0)));
                }
                databaseManager.persistTicket(ticket);
                onGetTicketListener.onSuccess(ticket);
            }
            @Override
            public void onFailure(DatabaseError databaseError){
            }
        });
    }

    // checks the Ticket state, whether it's ACTIVE or WAITING
    @Override
    public void checkTicket(Ticket ticket, OnCheckTicketListener
        onCheckTicketListener) {
        maxId = -1;
        databaseManager.getStore(ticket.getStore(), new OnGetDataListener() {
            @Override
            public void onSuccess(DataSnapshot dataSnapshot) {
                if(dataSnapshot.child("Tickets").
                hasChild(String.valueOf(ticket.getId())) == true) {
                    if(dataSnapshot.child("Tickets")
                    .child(String.valueOf(ticket.getId())).
                    child("ticketState").getValue().toString().
                    equals("ACTIVE")) {
                        //how much does he have left
  onCheckTicketListener.onActive(Timestamp.
  valueOf(dataSnapshot.child("Tickets").child(String.
  valueOf(ticket.getId())).child("expires").getValue().toString())));
                    } else {
                        //calculate people in front
                        int peopleAhead = 1;
                        for (DataSnapshot i :
                            dataSnapshot.child("Tickets").getChildren()) {
                            if
(i.child("ticketState").getValue().toString().equals("WAITING") &&
   Integer.parseInt(i.getKey()) < ticket.getId())
                                peopleAhead++;
                        }
                        onCheckTicketListener.onWaiting(peopleAhead);
                    }
                }
                else {
                    onCheckTicketListener.onBadStore("Ticket has already been
                        used");
```

```java
                }
            return;
        }

        @Override
        public void onFailure(DatabaseError databaseError){
            onCheckTicketListener.onBadStore("Bad store information -
                reload app");
        }
    });
}
// checks the current store queue to see how many customers are in line
@Override
public void checkQueue(Store store, OnCheckTicketListener
    onCheckTicketListener) {
    maxId = -1;
    databaseManager.getStore(store, new OnGetDataListener() {
        @Override
        public void onSuccess(DataSnapshot dataSnapshot) {
            //calculate people in front
            if
                (Integer.parseInt(dataSnapshot.child("open").getValue().toString())
                == 0) {
                onCheckTicketListener.onBadStore("The store is not open");
                return;
            }
            int peopleAhead = 0;
            for (DataSnapshot i :
                dataSnapshot.child("Tickets").getChildren()) {
                if
                    (i.child("ticketState").getValue().toString().equals("WAITING"))
                    peopleAhead++;
            }
            onCheckTicketListener.onWaiting(peopleAhead);
            //System.out.println("AAAAA" + peopleAhead);
            return;
        }
        @Override
        public void onFailure(DatabaseError databaseError){
        }
    });
}
// cancels and deletes a Ticket
@Override
public void cancelTicket(Store store, Ticket ticket,
    OnTaskCompleteListener onTaskCompleteListener) {
    databaseManager.getTicket(store, String.valueOf(ticket.getId()), new
        OnGetDataListener() {
        @Override
        public void onSuccess(DataSnapshot dataSnapshot) {
```

```java
            if (dataSnapshot.getValue() == null) {
                onTaskCompleteListener.onFailure(0);
                return;
            }
            dataSnapshot.getRef().setValue(null);
            onTaskCompleteListener.onSuccess();
            return;
        }
        @Override
        public void onFailure(DatabaseError databaseError){
        }
    });
    }
}
```

Entities - OnTicketCheckListener

```java
// OnTicketCheckListener.java
package com.example.clup;

import java.sql.Timestamp;

public interface OnCheckTicketListener {
    public void onWaiting(int peopleAhead);
    public void onActive(Timestamp expireTime);
    public void onBadStore(String error);
}
```

Entities - HomeController

```java
// HomeController.java
package com.example.clup;

import androidx.appcompat.app.AppCompatActivity;
import androidx.core.app.ActivityCompat;
import androidx.core.content.ContextCompat;

import android.Manifest;
import android.content.Context;
import android.content.Intent;
import android.content.SharedPreferences;
import android.content.pm.PackageManager;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;

import com.example.clup.Entities.ApplicationState;
import com.google.firebase.FirebaseApp;
import com.google.firebase.auth.FirebaseAuth;

public class HomeController extends AppCompatActivity implements
```

```java
View.OnClickListener{

private Button storeButton, loginButton;
public static final String MyPREFERENCES = "MyPrefs" ;
private static final int MY_CAMERA_REQUEST_CODE = 100;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_home_controller);

    storeButton = (Button) findViewById(R.id.storeButton);
    loginButton = (Button) findViewById(R.id.loginButton);
    // Update user
   if (FirebaseAuth.getInstance().getCurrentUser() == null)
      System.out.println("NOPE");

    // checks for camera permission and asks for it if it's not permitted
       - scanner will crash the app if
    // the camera is not enabled
    if (checkSelfPermission(Manifest.permission.CAMERA)
          != PackageManager.PERMISSION_GRANTED) requestPermissions(new
             String[]{Manifest.permission.CAMERA},
             MY_CAMERA_REQUEST_CODE);

    storeButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            startActivity((new Intent(v.getContext(),
                StoreController.class)));
        }
    });
    loginButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v2) {
            if (FirebaseAuth.getInstance().getCurrentUser() == null)
                startActivity((new Intent(v2.getContext(),
                    LoginController.class)));
            else
                startActivity((new Intent(v2.getContext(),
                    PreLoginController.class)));
        }
    });
}

@Override
public void onClick(View v) {
}
// Sets the action of a back button pressed from Android
@Override
```

```java
public void onBackPressed () {
    ((ApplicationState) getApplication()).clearAppState();
    // Clears stack of activities
    finishAffinity();

}
}
```

5 Testing

Two main types of testing have been done for this version of the app - Unit testing and Firebase testing. Unit testing could not have been done through the usual means as JUnit tests, but rather as ordinary tests due asynchronous nature of Firebase, which makes it impossible to test with JUnit tests.

The only real JUnit test is done for AES encryption/decryption. More on that can be found in the latter subsections.

5.1 Unit testing

Unit testing has been done for the following actions of fetching Firebase data:

- **Fetching store cities**

```java
Director director = new Director();

director.getStoreSelectionManager().getStoreCities(new OnGetDataListener() {
    @Override
    public void onSuccess(DataSnapshot dataSnapshot) {
        for (DataSnapshot i : dataSnapshot.getChildren()) {
            System.out.println("DB city: \"" +i.getKey());
        }
    }
});
```
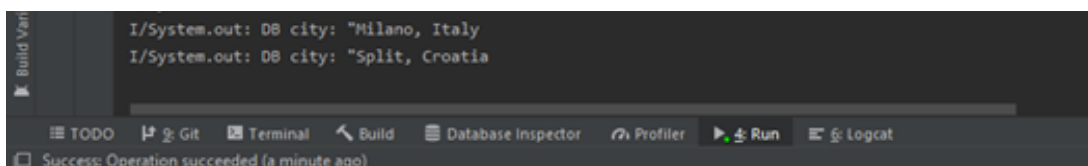
Figure 4: **Fetching store cities test 1**

```
I/System.out: DB city: "Milano, Italy
I/System.out: DB city: "Split, Croatia
```

≡ TODO   ⌥ 9: Git   ⊠ Terminal   ⚒ Build   ⊟ Database Inspector   ⌁ Profiler   ▶ ⛟ Run   ⊑ ⛟ Logcat
⊡ Success: Operation succeeded (a minute ago)

Figure 5: **Fetching store cities test 2**

- **Fetching store chains in a city**

```
Director director = new Director();

director.getStoreSelectionManager().getStores( city: "Milano, Italy", new OnGetDataListener(){
    @Override
    public void onSuccess(DataSnapshot dataSnapshot) {
        for(DataSnapshot i : dataSnapshot.getChildren()) {
            System.out.println("Store chains in Milano, Italy: "+i.getKey());
        }
    }
});
```

Figure 6: **Fetching store chains in a city test 1**

```
I/System.out: Store chains in Milano, Italy: Carrefour
    Store chains in Milano, Italy: Esselunga
```

Figure 7: **Fetching store chains in a city test 2**

• **Fetching store chain addresses**

```
Director director = new Director();

director.getStoreSelectionManager().getStoreAddresses( city: "Milano, Italy", name: "Carrefour", new OnGetDataListener(){
    @Override
    public void onSuccess(DataSnapshot dataSnapshot) {
        for(DataSnapshot i : dataSnapshot.getChildren()) {
            System.out.println("Carrefour store with index: "+i.getKey()+" and location in Milano, Italy: "+i.child("address").getValue());
        }
    }
});
```

Figure 8: **Fetching store chain addresses test 1**

```
I/System.out: Carrefour store with index: 0 and location in Milano, Italy: Viale Ungheria 3
I/System.out: Carrefour store with index: 1 and location in Milano, Italy: Viale Ungheria 4
```

Figure 9: **Fetching store chain addresses test 2**

• **Checking credentials for a store manager login**

```
Director director = new Director();

director.getLoginManager().manageLogin( email: "toma.petar.sikora@gmail.com", password: "Toma123", UserType.STORE_MANAGER, new OnCredentialCheckListener() {
    @Override
    public void onSuccess(String storeName, String storeCity, int storeId) {
        System.out.println("Credentials successfully checked! Store manager logged in!");
    }

    @Override
    public void onFailure() {
        System.out.println("Failed to check credentials and log in!");
    }
});
```

Figure 10: **Successful login test**

```
I/System.out: Credentials successfully checked! Store manager logged in!
```

Figure 11: **Successful login test results**


```
Director director = new Director();

director.getLoginManager().manageLogin( email: "toma.petar.sikora@gmail.com", password: "wrong password", UserType.STORE_MANAGER, new OnCredentialCheckListener() {
    @Override
    public void onSuccess(String storeName, String storeCity, int storeId) {
        System.out.println("Credentials successfully checked! Store manager logged in!");
    }

    @Override
    public void onFailure() {
        System.out.println("Failed to check credentials and log in!");
    }
});
```

Figure 12: **Failed login test**


```
I/System.out: Failed to check credentials and log in!
```

Figure 13: **Failed login test results**

- **Acquiring ticket to get in the virtual line**


```
Director director = new Director();
Store store = new Store( id: 0, name: "Carrefour", address: "Viale Ungheria 3", city: "Milano, Italy");

director.getDatabaseManager().getStore(store, new OnGetDataListener() {
    @Override
    public void onSuccess(DataSnapshot dataSnapshot) {
        System.out.println("Ticket with max number is: "+dataSnapshot.child("maxId").getValue());
    }
});

director.getRequestManager().getTicket(store, new OnGetTicketListener() {
    @Override
    public void onSuccess(Ticket ticket) {
        System.out.println("Successfully acquired ticket with number: "+ticket.getId());
    }

    @Override
    public void onFailure() {

    }
});
```

Figure 14: **Acquiring ticket test**


```
Ticket with max number is: 6
I/System.out: Successfully acquired ticket with number: 7
```

Figure 15: **Acquiring ticket test results**

- **Checking the state of the ticket**

```java
Director director = new Director();
Store store = new Store( id: 0,  name: "Carrefour",  address: "Viale Ungheria 3",  city: "Milano, Italy");

director.getRequestManager().getTicket(store, new OnGetTicketListener() {
    @Override
    public void onSuccess(Ticket ticket) {
        System.out.println("Successfully acquired ticket with number: "+ticket.getId());
        director.getRequestManager().checkTicket(ticket, new OnCheckTicketListener() {
            @Override
            public void onWaiting(int peopleAhead) {
                System.out.println("The person with the ticket is waiting for his turn.");
            }

            @Override
            public void onActive(Timestamp expireTime) {
                System.out.println("The person with the ticket should go to the store and enter.");
            }
        });
    }

    @Override
    public void onFailure() {

    }
});
```

Figure 16: **Checking the state of the ticket test**

```
I/System.out: Successfully acquired ticket with number: 9
I/System.out: The person with the ticket is waiting for his turn.
```

Figure 17: **Checking the state of the ticket test results**

- **Cancelling the ticket**

```java
Director director = new Director();
Store store = new Store( id: 0,  name: "Carrefour",  address: "Viale Ungheria 3",  city: "Milano, Italy");

director.getRequestManager().getTicket(store, new OnGetTicketListener() {
    @Override
    public void onSuccess(Ticket ticket) {
        System.out.println("Successfully acquired ticket with number: "+ticket.getId());
        director.getRequestManager().cancelTicket(store, ticket, new OnTaskCompleteListener() {
            @Override
            public void onSuccess() {
                System.out.println("Ticket cancelled!");
            }

            @Override
            public void onFailure() {
                System.out.println("Failed to cancel ticket!");
            }
        });
    }

    @Override
    public void onFailure() {

    }
});
```

Figure 18: **Cancelling the ticket test**

```
I/System.out: Successfully acquired ticket with number: 12
I/System.out: Ticket cancelled!
```

Figure 19: **Cancelling the ticket test results**

- **Checking ticket logic 1**

  Beginning state:
  Maximum store occupancy is 2, current occupancy is 0. There are 2 active tickets with numbers 11 and 12, with expire time 11:01 (5 minutes after activation), 3 waiting tickets with numbers 13, 14, and 15.

```java
Director director = new Director();
Store store = new Store( id: 0,  name: "Carrefour",  address: "Viale Ungheria 3",  city: "Milano, Italy");

director.getStoreManager().updateQueue(store);

director.getDatabaseManager().getTickets(store, new OnGetDataListener() {
    @Override
    public void onSuccess(DataSnapshot dataSnapshot) {
        for(DataSnapshot i : dataSnapshot.getChildren())
            System.out.println("Ticket no. "+i.getKey()+", with ticket state: "+i.child("ticketState").getValue());
    }
});
```

Figure 20: **Checking ticket logic test 1.1**

```
I/System.out: Ticket no. 11, with ticket state: ACTIVE
        Ticket no. 12, with ticket state: ACTIVE
        Ticket no. 13, with ticket state: WAITING
I/System.out: Ticket no. 14, with ticket state: WAITING
        Ticket no. 15, with ticket state: WAITING
```

Figure 21: **Checking ticket logic test 1.1 results**

Customer with ticket number 11 scans his ticket and the store manager lets him in.

```java
String qrCodeText = "Milano, Italy; Carrefour; Viale Ungheria 3; 0; 11";
director.getStoreManager().manageEntrance(qrCodeText, new OnTaskCompleteListener() {
    @Override
    public void onSuccess() {
        System.out.println("Admitted a customer in!");
    }

    @Override
    public void onFailure() {
        System.out.println("Failed to scan ticket!");
    }
});
```

Figure 22: **Checking ticket logic test 1.2**

```
I/System.out: Admitted a customer in!
```

Figure 23: **Checking ticket logic test 1.2 results**

After a couple of minutes, store manager automatically updates the queue:
In the meantime, ticket with number 12 has expired, and the ticket with number 13 is activated.

```java
Director director = new Director();
Store store = new Store( id: 0,  name: "Carrefour",  address: "Viale Ungheria 3",  city: "Milano, Italy");

director.getStoreManager().updateQueue(store);

director.getDatabaseManager().getTickets(store, new OnGetDataListener() {
    @Override
    public void onSuccess(DataSnapshot dataSnapshot) {
        for(DataSnapshot i : dataSnapshot.getChildren())
            System.out.println("Ticket no. "+i.getKey()+", with ticket state: "+i.child("ticketState").getValue());
    }
});
```

Figure 24: **Checking ticket logic test 1.3**

```
I/System.out: Ticket no. 13, with ticket state: ACTIVE
I/System.out: Ticket no. 14, with ticket state: WAITING
      Ticket no. 15, with ticket state: WAITING
```

Figure 25: **Checking ticket logic test 1.3 results**

When a customer exits, the occupancy changes and another customers ticket is activated.

```
Director director = new Director();
Store store = new Store( id: 0,  name: "Carrefour",  address: "Viale Ungheria 3",  city: "Milano, Italy");

director.getStoreManager().manageExit(store);
```

Figure 26: **Checking ticket logic test 1.4**

```
Director director = new Director();
Store store = new Store( id: 0,  name: "Carrefour",  address: "Viale Ungheria 3",  city: "Milano, Italy");

director.getStoreManager().updateQueue(store);

director.getDatabaseManager().getTickets(store, new OnGetDataListener() {
    @Override
    public void onSuccess(DataSnapshot dataSnapshot) {
        for(DataSnapshot i : dataSnapshot.getChildren())
            System.out.println("Ticket no. "+i.getKey()+", with ticket state: "+i.child("ticketState").getValue());
    }
});
```

Figure 27: **Checking ticket logic test 1.5**

```
I/System.out: Ticket no. 14, with ticket state: ACTIVE
I/System.out: Ticket no. 15, with ticket state: ACTIVE
```

Figure 28: **Checking ticket logic test 1.4 and 1.5 results**

- **Checking ticket logic 2**

  Beginning state:
  Maximum store occupancy is 2, tickets with number 1 and 2 are activated, ticket with number 3 is waiting. Person with ticket number 3 tries to enter the store before his ticket is activated.

```
Director director = new Director();
Store store = new Store( id: 0,  name: "Carrefour",  address: "Viale Ungheria 3",  city: "Milano, Italy");

String qrCodeText = "Milano, Italy; Carrefour; Viale Ungheria 3; 0; 3";

director.getStoreManager().manageEntrance(qrCodeText, new OnTaskCompleteListener() {
    @Override
    public void onSuccess() {
        System.out.println("Admitted the customer in!");
    }

    @Override
    public void onFailure() {
        System.out.println("Customer denied entrance!");
    }
});
```

Figure 29: **Checking ticket logic test 2**

```
I/System.out: Customer denied entrance!
```

Figure 30: **Checking ticket logic test 2 results**

5.2   Firebase testing

Firebase has its own implemented testing that tests the stress the app puts on the system. It crawls through the entire app and returns results based on memory, network, and CPU usage. It also takes into account UI responsiveness and framerate.

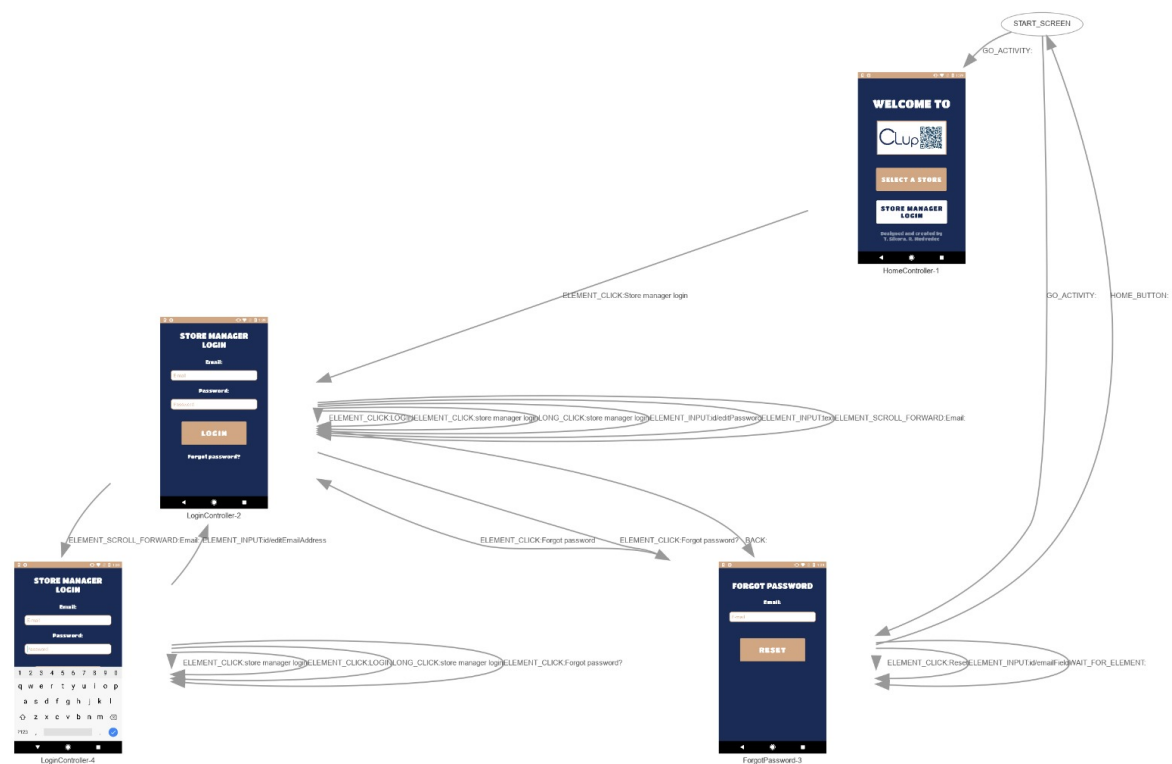Here are the results of Firebase testing:

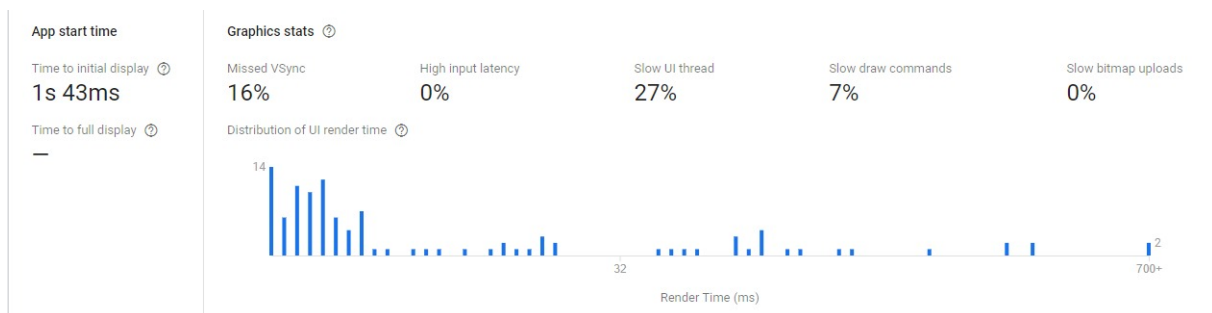Figure 31: **Firebase test crawling diagram**

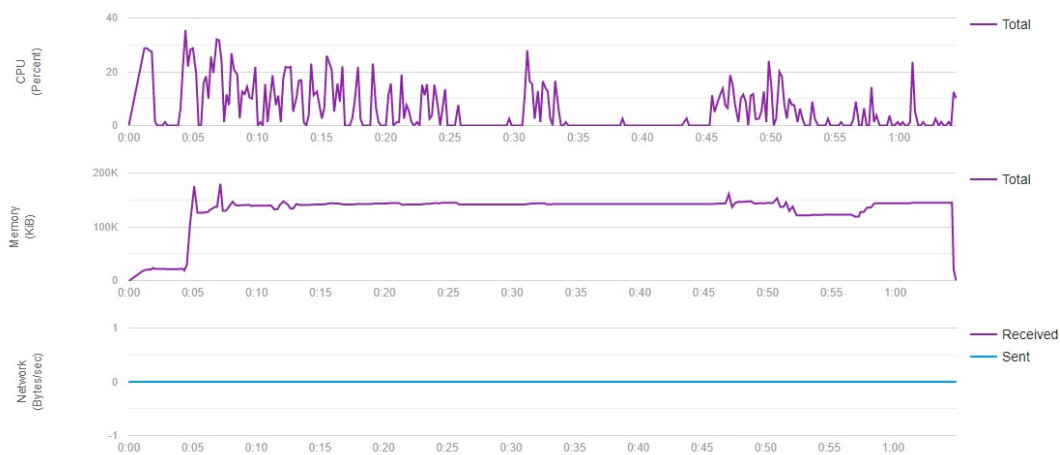Figure 32: **Firebase test UI responsiveness**

Figure 33: **Firebase test performance and resource usage**

5.3 Security testing

The only real JUnit test from Android Studio was ran on this component and the code looks like this:

```java
// Hello.java
package com.example.clup;

import org.junit.Test;

import static org.junit.Assert.assertEquals;



/**
 * Local unit test to test whether AES encryption of tickets works as intended.
 * The test will execute on the development machine (host).
 */
public class EncryptionUnitTest {
    @Test
    public void encryption_is_correct() {
        StrongAES strongAES = new StrongAES();
        String text = "This text should remain the same after encryption and
            decryption.";
        String key = "YaBcmo5Tz3hb8piW";
        assertEquals(text, strongAES.AESDecrypt(strongAES.AESEncrypt(text,
            key), key));
    }
}
```

The code returns the correct value which means that the test is passed.

6   Installation Instructions

Installing and testing the app is easy. All that needs to be done is to download .apk file from "DeliveryFolder" of our GitHub page (`https://github.com/robertodavinci/Software_Engineering_2_Project_Medvedec_Sikora/tree/main/DeliveryFolder`) and install it on the phone.

Make sure to allow apps from untrusted source in the settings so that the app can be properly installed since this version of the app is not signed for Google Play Store.

When asked for permission to use camera, "Allow" must be pressed in order for the QR code scanner to work properly. If the camera is not allowed, the app will return to the Home screen every time a scanner is opened.

There is also a possibility of importing an entire project in Android Studio and running the app on an emulator. Simply download the whole "Implementation" folder from our repository and import the project to Android Studio. The setup should be done automatically.

40