U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Mobile Computing
5th year of the Integrated Masters in Informatics and Computing Engineering (MIEIC)

Shop and Pay Android app for a supermarket
The Acme Electronic Supermarket

Group elements:

Diogo Luis Rey Torres - 201506428 - up201506428@fe.up.pt
Filipe Oliveira e Sousa Ferreira de Lemos - 201200689 - ee12140@fe.up.pt

November 14, 2019

Index

1.Introduction

This project consists of a shopping and payment system for an electronic supermarket. It was developed using a client-server architecture and composes essentially three main applications: the customer app, the terminal app and the supermarket server.

A supermarket needs to process shopping and payments in an efficient way for the customer. When a customer enters the supermarket, he can add the products he wants to buy into his application (reading the products QR Code and adding them into the shop basket). Therefore, when he finishes selecting the products, he can move to the checkout by showing the QR Code of his transaction to the supermarket terminal. There, the payment is made using the customer's associated credit card and the supermarket gate is opened. With this in mind, the system implements all the necessary requirements for its operation presenting them in an intuitive and pleasant way for the final user.

In the next sections, the application will be described in finer detail, namely its architecture, implementation details and relevant issues. This report concludes with a small reflection about the work done and possible future enhancements.

2. Architecture

2.1 Description

This application is based on a Client-Server architecture that allows clients and servers to communicate over a computer network, where the server shares their resources with clients. The sharing information between the server and clients is made through a REST protocol, where clients make requests they are interested in, and the server responds accordingly.

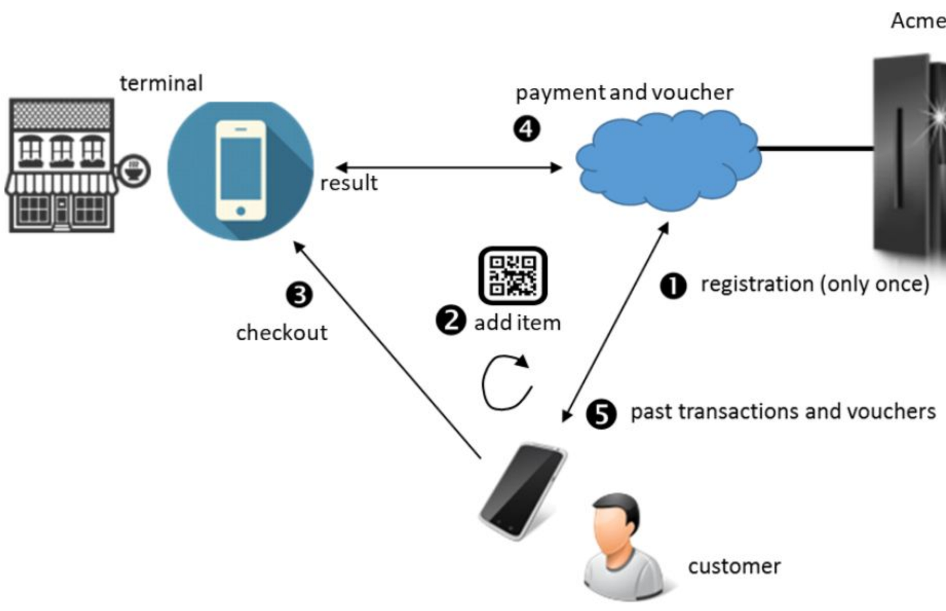


Figure 1: System Architecture

The used architecture consists of four applications: the customer application (AcmeSupermarket), the product QR code generator (GenProductTag), the terminal application (TerminalApp) and the supermarket server (Server) with its own database.

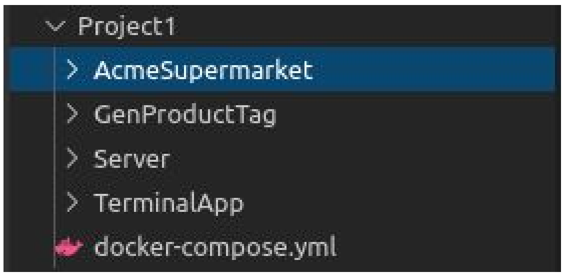


Figure 2: Solution Overview

The three applications run independently in an Android environment. The server is containerized, allowing the application to run quickly and reliably from one computing environment to another, as well as allowing a quickly deployment if necessary. Docker-compose is used to setup the server and database container.

Most of the communication with the server is made by using cryptography, more specifically, asymmetric keys that provides authenticity in the network (See subsection 2.4).

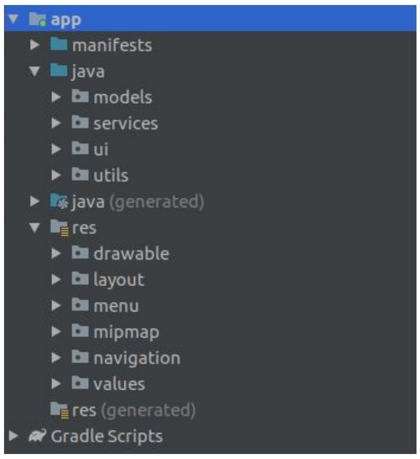
2.2 Applications

2.2.1 Customer Application

This application is based on *Android Framework* that presents to the customer a way to efficiently shop and checkout his products in the supermarket. It communicates with the supermarket server through REST calls to synchronize data about transactions, vouchers and to register himself into the supermarket. Also, if the customer tries to do the login (locally) and it doesn't exist, a call to the server is made to check if the user exists in the database.

2.2.1.1 Code Structure

The customer application has the manifest, the java code and the resources.



```
▼ app
  ▶ manifests
  ▼ java
    ▶ models
    ▶ services
    ▶ ui
    ▶ utils
  ▶ java (generated)
  ▼ res
    ▶ drawable
    ▶ layout
    ▶ menu
    ▶ mipmap
    ▶ navigation
    ▶ values
    res (generated)
▶ Gradle Scripts
```

Figure 3: Customer App Overview

The java code is divided into 4 packages: *models*, *services*, *ui* and *utils*.

2.2.1.1.1 Models Package

The **models** package is responsible for creating, managing and saving the entities used during the app. The entities are: client, creditCard, transaction, transactionItem and voucher.

2.2.1.1.2 Services Package

The **services** package was created to host easily new services to the app. The current services are: **cryptography** service and **repository** service.

2.2.1.1.2.1 Cryptography Package

The **cryptography** service consists of 3 classes: *Constants, CryptoBuilder and CryptoKeysManagement*.

The *Constants* class contains some configurations used like the encryption algorithm, the key size and others utilitary strings.

The *CryptoBuilder* class contains functions to sign and validate the messages using a given private/public key.

The *CryptoKeysManagement* class contains functions to generate and save the keys into the KeyStore, to get public/private keys from the KeyStore and also convert a key in PEM format into the classes supported by Java.

2.2.1.1.2.2 Repository Package

The **repository** service also consists of 3 classes: *AbstractRestCall, AcmeRepository and Constants*.

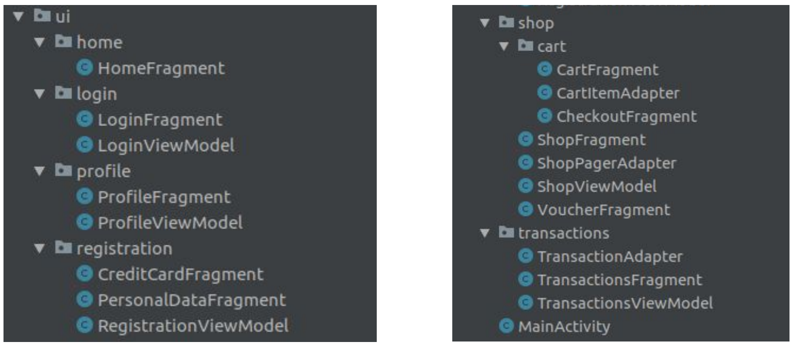
The *AbstractRestCall* is an abstract class created to avoid code repetition. This class is responsible for setup a connection to the REST call, send the payload and get the response.

The *AcmeRepository* class contains methods to generate the user signature and to validate a signature provenient from supermarket server or the products read. Also contains all the classes necessary to interact with the supermarket server, each class is a REST call that extends *AbstractRestCall*.

The *Constants* class contains some configurations used like the server URL and the respective endpoints.

2.2.1.1.3 UI Package

The **UI** package is divided into features, which was decided to be the best approach compared to splitting the code in activities, adapters, and so on, because all the related code remains in the same package facilitating the development.



```
▼ ui
  ▼ home
      HomeFragment
  ▼ login
      LoginFragment
      LoginViewModel
  ▼ profile
      ProfileFragment
      ProfileViewModel
  ▼ registration
      CreditCardFragment
      PersonalDataFragment
      RegistrationViewModel
```

```
▼ shop
  ▼ cart
      CartFragment
      CartItemAdapter
      CheckoutFragment
    ShopFragment
    ShopPagerAdapter
    ShopViewModel
    VoucherFragment
▼ transactions
    TransactionAdapter
    TransactionsFragment
    TransactionsViewModel
  MainActivity
```

Figure 4: UI Package Overview

This application only contains a *MainActivity* and its composed by fragments. This allows a better performance in the application, a better management of the lifecycle and it will be used with a navigation graph eliminating the need of the *fragmentTransaction* to switch between views.

This package is divided into 6 packages: home, login, profile, registration, shop and transactions. All of them use fragments to show their functionality, they are integrated into the MainActivity and most of them have a correspondent view model. This view model allows a better management of the data in each view and an easy way to share information among the different fragments (See subsection 2.2.1.3). Some views have an adapter associated to inject the data provided by the server. Finally, most of the input has some validation like checking if a certain input is empty.

2.2.1.1.4 Utils Package

The **utils** package was created to host some utility functions and constants used in the application (not specific to a given service as mentioned before).

2.2.1.2 App Navigation

This application adopts a new way to implement the navigation across the different pieces of content within the app. As mentioned before, this application has only one main activity and various fragments.

The navigation component has three key parts that are described below:

- **Navigation graph**: An XML resource called *nav_graph* that contains all navigation-related information in one centralized location.

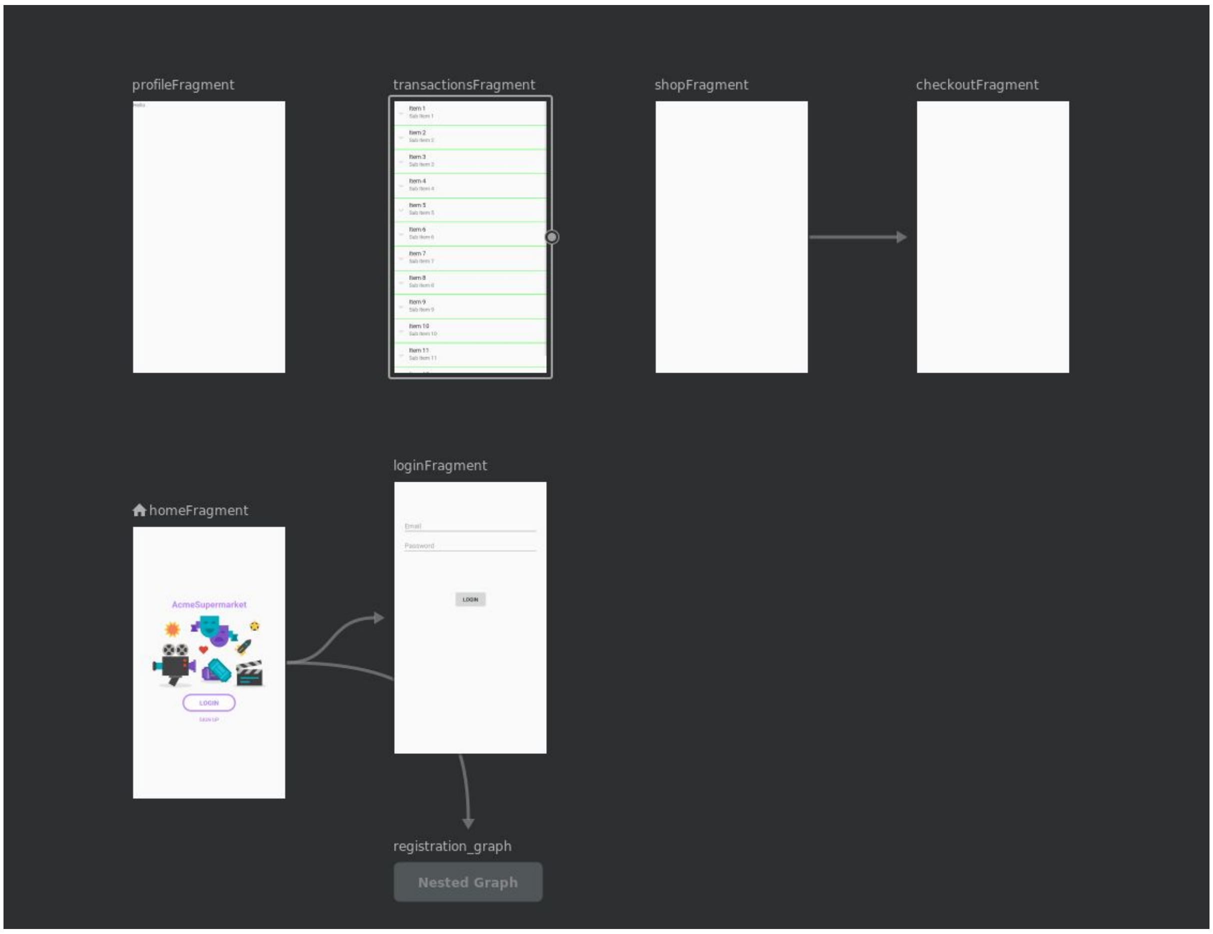


Figure 5: Navigation Graph Overview

- **NavHost**: Hosted on the *content_main.xml* included in the main activity is an empty container that displays destinations from the navigation graph. This container wraps a default NavHost implementation that displays fragment destinations.
- **NavController**: An object that manages app navigation within a NavHost. The NavController orchestrates the swapping of destination content in the NavHost as users move throughout the app.

The NavController allows the application to navigate directly to a specific destination by replacing the fragments but also allows the creation of a menu in an automated way. This menu, called the navigation drawer, is a UI panel that shows the app main navigation menu (only visible after the authentication process).

2.2.1.3 ViewModels

The *ViewModel* class is used to store and manage the UI-related data in a way that survives configuration changes such as screen rotations.

When the screen rotates the system destroys ou recreates a UI Controller and consequently the UI-related data is lost, one way to prevent this behaviour is the use of *onSaveInstanceState()* method and restore its data from the bundle in *onCreate()*, but this approach is not suitable for large amounts of data.

Another problem is that when the UI controller needs to update the view accordingly with a REST Call made to the server, it takes some time to return. The *viewModel* with *LiveData* objects allows setting up an observer to these fields and change them when the value is updated by an asynchronous request.

Finally, this approach makes the sharing of data between components easier because the fragment can request the *ViewModelProviders* for a specific *ViewModel* associated to the current activity, avoiding the need to make each fragment know each other, being only required to know how to take advantage of the shared *ViewModel*. If one of the fragments disappears, the other one keeps working as usual.

2.2.1.4 Authentication

To handle the authentication process an enum class was created in *LoginViewModel* that handles the user's authentication state and exposes it via *LiveData* object, so the app can decide where to navigate by setting an observer in this field.

If the user is not authenticated, then he can only navigate between the home page, the login page and the registration section (this one is composed by two different pages to separately handle the personal info from the user's credit card info). If the user is authenticated, then he can access all the other pages (transactions page, profile page and shop page).

The authentication is made in the login page or the signup process, where after the user triggers the login/signup button if the information is valid the state changes to AUTHENTICATED, otherwise the state changes to INVALID_AUTHENTICATION and an error message is shown to the user. Also, when the user opens the app the first state is UNAUTHENTICATED.

2.2.1.5 Local Storage

Some data is persisted locally in the application internal storage. Once registered, the user details are saved to a file "{username}_clientData.txt", which is accessed every time a login operation is attempt on that user. This avoids requesting the server for credentials confirmation, since the existence of such a file and successively password match, grants entrance to the shop screen.

Information regarding user transactions and vouchers are also persisted in files that follow the same format - "{username}_transactionsData" and "{username_vouchersData"} - which are updated every time a sync operation is requested by the user.

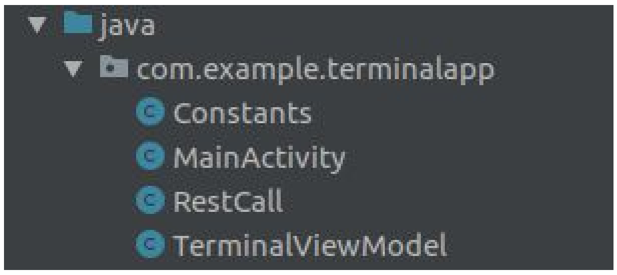
Note that, this data storage format allows having multiple clients using the same device.

2.2.2 Terminal Application

This application is based on *Android Framework* that represents the terminal where the customer shows the transaction QR code to finish his purchase. It communicates with the supermarket server through REST calls to make the transaction and presents back the total value paid and the background color changes to green or red according to the success of the operation. Also, in case of error shows a message retrieved by the database like the voucher is already used.

2.2.2.1 Code Structure

The code structure is much simpler comparatively with the customer application because this application has less functionalities.



```
▼ java
  ▼ com.example.terminalapp
      Constants
      MainActivity
      RestCall
      TerminalViewModel
```

Figure 6: Terminal App Overview

Basically, this application has a *MainActivity*, a *RestCall* class that contains the rest call to the server to perform the checkout, a *TerminalViewModel* to handle the information retrieved by the server, and a *Constants* file to hold some information like the server URL and respective endpoints but also some variables used in the QR Reader.

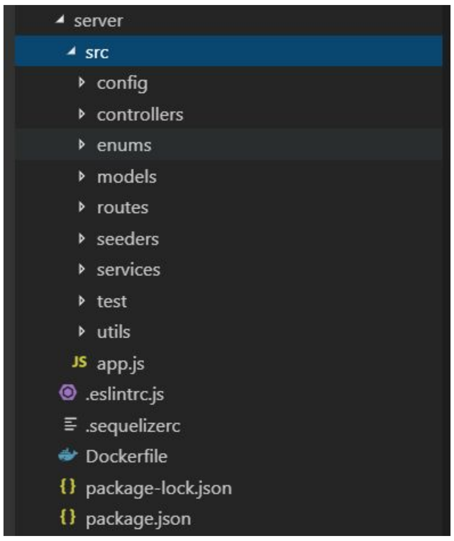
2.2.3 Product QR code generator Application

This application is based on a provided demo application provided and adapted to handle a key pair generated by the supermarket server. This application allows the creation of the QR Code to the products with some information like name, price and an identifier. This information is signed by the supermarket private key to avoid fake labels.

Note: Throughout the applications, calls to the "Barcode Scanner" application by *ZXing Team* are regular and necessary because it is used to parse the QR code data.

2.2.4 Server

The server is based on *ExpressJs* framework with the following structure:



```
◢ server
  ◢ src
    ▸ config
    ▸ controllers
    ▸ enums
    ▸ models
    ▸ routes
    ▸ seeders
    ▸ services
    ▸ test
    ▸ utils
    app.js
  .eslintrc.js
  .sequelizerc
  Dockerfile
  package-lock.json
  package.json
```

Figure 7: Server's folders

This structure allows a good division between models, logic, services and routes that follows good programming practices. The server has a database in *postgres* that saves all the information necessary to manage the store.

2.2.4.1 Endpoints

The table below shows all the endpoints that the server provides.

Type	URL	Description
POST	/signup	Registers the user with his personal info and credit card information.
POST	/login	Logins the user if exists in the database and updates the user's public Key.
POST	/transactions	Generates a transaction associated to a user
POST	/transactions/user	Retrieves all the transactions by a given user
POST	/users/user	Retrieves all the user's information by a given user
POST	/vouchers/unused/user	Retrieves all the vouchers unused by a given user

2.3 Data Schema

2.3.1. Description

This database contains the following models: User, CreditCard, Transaction, TransactionItem and Voucher.

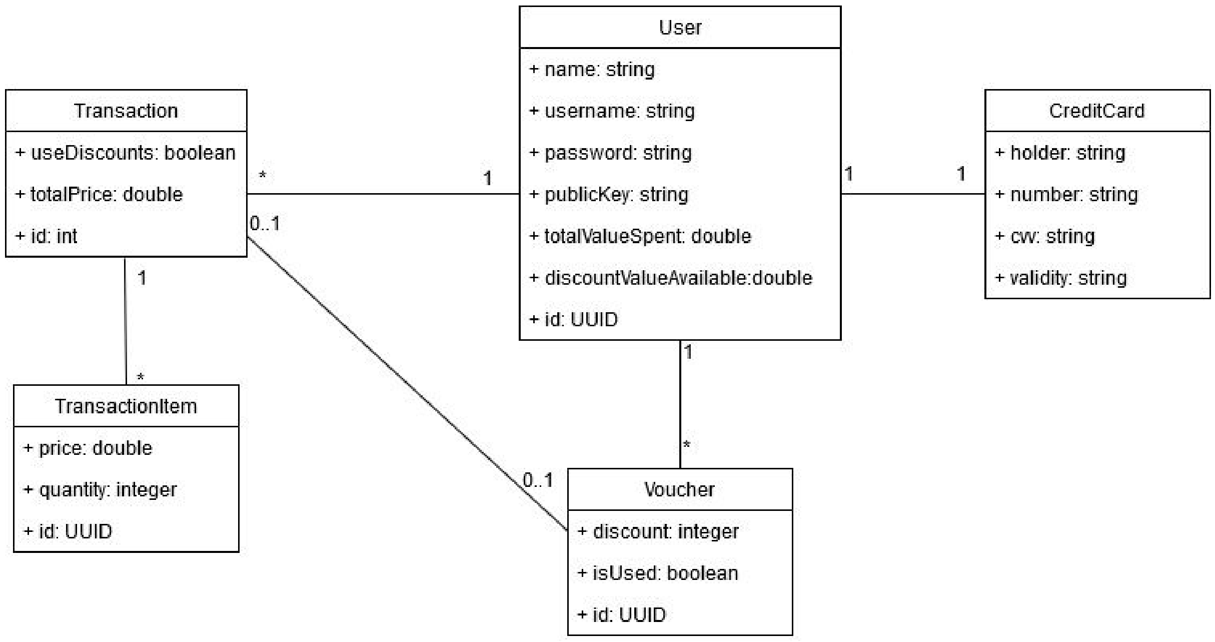


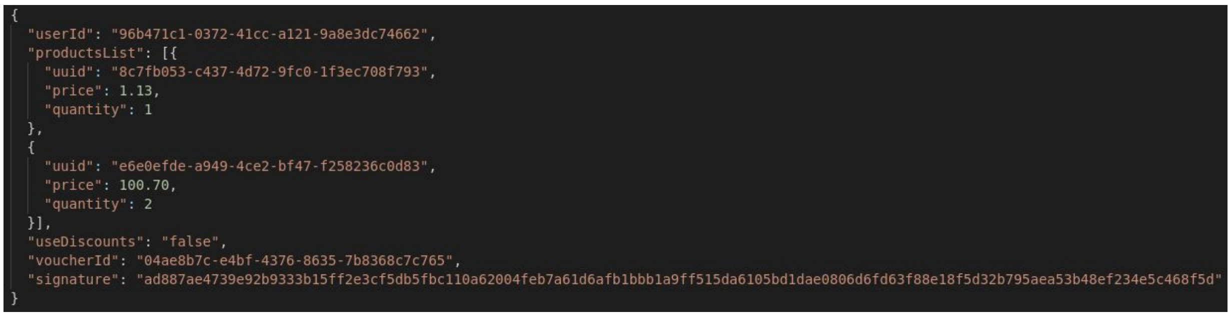
Figure 8: Database Schema

This schema essentially consists of a *User* that contains some personal information, his public key (for security purposes), data about the value spent in the supermarket and the discount accumulated so far. Also, each *user* has a *credit card* associated.

Each user has his own transactions with some information like if it was used the discount accumulated and the total price of this transaction. Each *transaction* has products (*TransactionItem*) associated to it, which have information like the price of the product, its id and the quantity purchased in that transaction. Also, when the user's total value spent surpasses a new multiple of €100.00, a new *voucher* associated to him is generated. A *voucher* has a discount value associated and a boolean to refer if it's used or not. Additionally, a *voucher* can be used in a *transaction* and a certain percentage of the total paid is calculated and added to the accumulated discount so far.

2.3.1. Checkout Information

The checkout information contains the list of products, the voucher's id (if used), the user's id and a boolean that represents if the user wants to use the discount accumulated so far. Appended to this information, a signature is added that can be seen in the image below.



```
{
  "userId": "96b471c1-0372-41cc-a121-9a8e3dc74662",
  "productsList": [{
    "uuid": "8c7fb053-c437-4d72-9fc0-1f3ec708f793",
    "price": 1.13,
    "quantity": 1
  },
  {
    "uuid": "e6e0efde-a949-4ce2-bf47-f258236c0d83",
    "price": 100.70,
    "quantity": 2
  }],
  "useDiscounts": "false",
  "voucherId": "04ae8b7c-e4bf-4376-8635-7b8368c7c765",
  "signature": "ad887ae4739e92b9333b15ff2e3cf5db5fbc110a62004feb7a61d6afb1bbb1a9ff515da6105bd1dae0806d6fd63f88e18f5d32b795aea53b48ef234e5c468f5d"
}
```

Figure 9: Checkout's JSON object

2.4 Security

Cryptography is achieved by using asymmetric keys. These keys are generated with the algorithm "SHA256WithRSA" and have 64 bytes length. There are two pair of asymmetric keys: one to the server and another one to each client and the system uses these keys in three main cases:

- The first one is used to encrypt the supermarket product QR-code labels with the supermarket's private key, preventing anyone of producing fake ones. The customer application possesses the corresponding public key to read them. This key is transmitted in the registration process.
- The second one takes place during the checkout, when the transaction's information is signed with the user's private key (generated in the registration process), appending this signature to the transaction' details, and transmitted together. To verify the signature, the server use the corresponding user's public key which was transmitted on the registration process.
- The last one is when the user synchronizes his data with the server (asks for the transactions, unused vouchers and personal info). He sends his identifier signed with his private key. After the server verifies the signature, it sends back all the information requested signed by the supermarket private key.

Additionally, an attempt to establish an HTTPs connection was done, in order to improve security in the communications but, as the SSL keys were self-signed (SSL keys were generated to the localhost and not certified by a Certificate Authority), it would be necessary to create in the Android application a *TrustManager* to accept this certificate, manually. To sum up, the changes in the application would open some vulnerabilities and we concluded that it was an unnecessary effort.

3.Functionalities included

3.1. Customer App

- Login a client - locally (registered on that mobile device);
- Login a client via server (registered on a different device);
- View the client’s current cart list;
- View the client’s vouchers list;
- View the client’s past transactions;
- View the client’s profile - account basic details (username, full name and creation date), total amount spent, available discount and credit card information;
- View the purchase QR code;
- Register a new client;
- Add items to the cart via QR code scanner;
- Increase/Decrease the quantity of a specific item;
- Remove an item from the cart list;
- Clear the entire cart list;
- Apply discount available to the total price;
- Generate the purchase QR code;
- Sync transactions, vouchers, total amount spent and available discount with the database;

3.2. Terminal App

- Scan client’s transaction QR code;
- View the payment confirmation and total price.

All requested features have been implemented as well as some extra features. Also, all the features were tested manually.

4.Screen Captures illustrating the main sequences of use

The App flow starts by opening the *Homepage* (Figure 10). Here the user can decide between creating an account (Figure 12 and 13) or logging in an existing account (Figure 11).

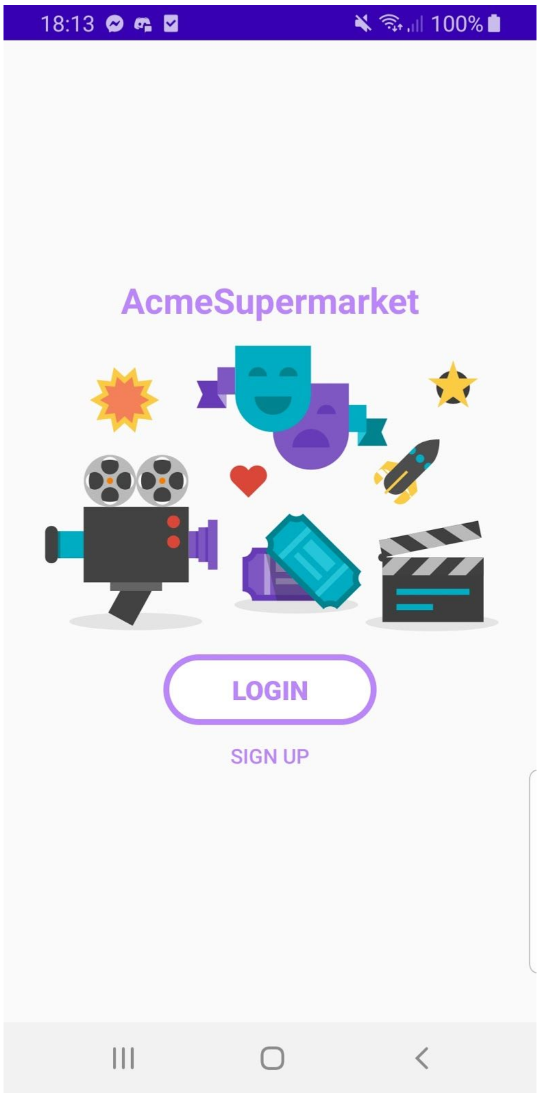


Fig. 10 - Homepage Screen

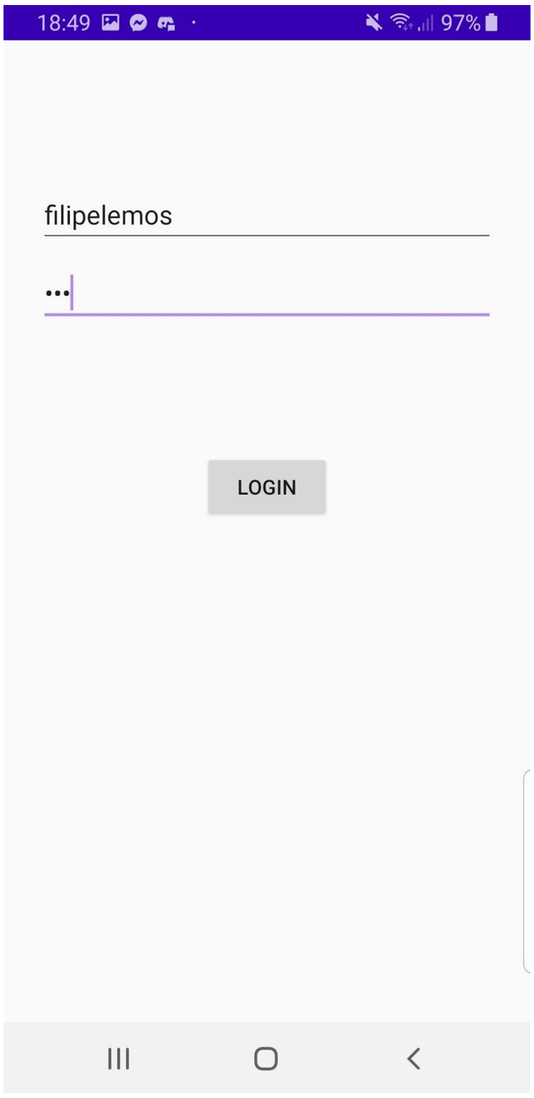
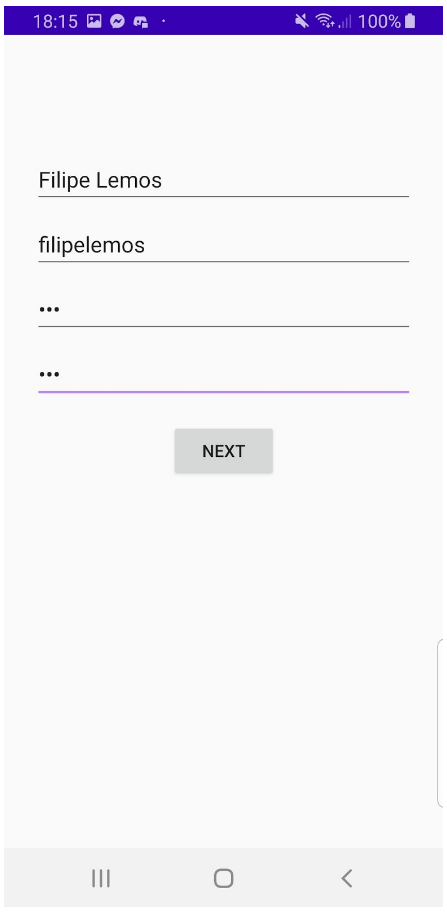


Fig. 11 - Login Screen


18:15 100%

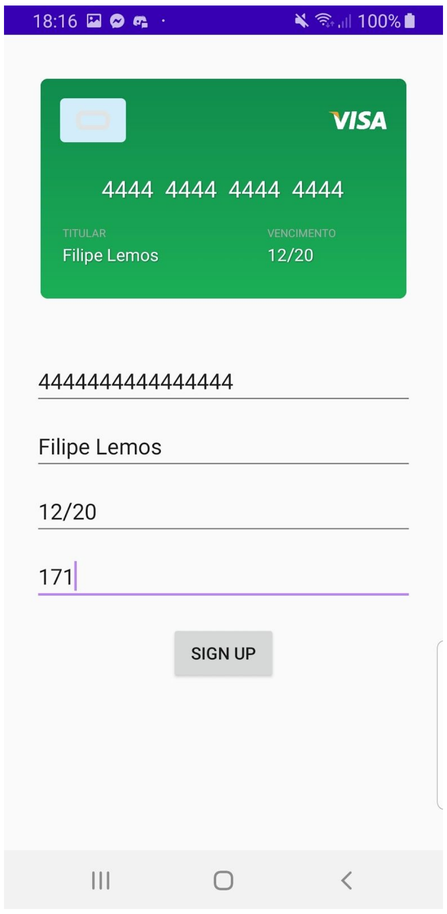
Filipe Lemos

filipelemos

•••

•••

Fig. 12 - Register Screen 1


18:16 100%

4444444444444444

Filipe Lemos

12/20

171

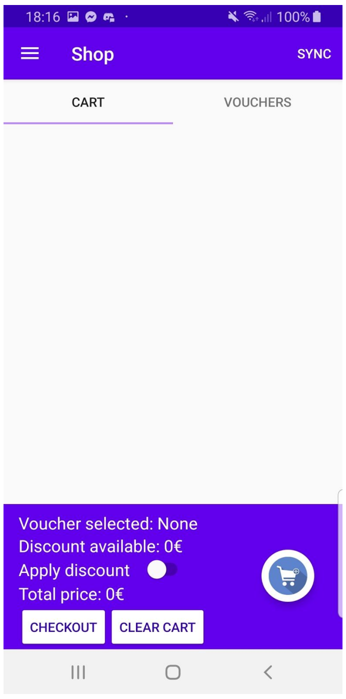
Fig. 13 - Register Screen 2

After successfully logging in or register an account, the user will be presented with the Shop Screen (Fig. 14). Clicking on the cart item, opens a bar code scanner (or asks the user permission for downloading it), allowing the user to scan the product QR codes (Fig. 15). Once this app finds a QR code, it adds the corresponding item to the cart list (Fig. 16).

Each item row identifies the item name, its unit price, the current quantity and has three buttons: one for increasing the item quantity (green plus), one for decreasing the item quantity (red minus) and one for deleting that item entry (trash bin).

Apart from the cart list, the Shop screen also has a blue section, on the bottom of the screen, that informs the user which voucher was activated (from the voucher list on Fig.17), the available user's discount and the current purchase total price. It also has a button to apply the discount on the current purchase, a button to reset the cart list and a "Checkout" button. This last button generates a QR code for the current purchase (Fig. 18), allowing the user to show it to the terminal and completing the transaction.

The application has a menu (Fig. 19) that can be found by pressing the settings icon on the toolbar. This menu allows the user to navigate between the shop screen, the transactions screen (Fig. 21) or the profile screen (Fig. 20).



Voucher selected: None
Discount available: 0€
Apply discount
Total price: 0€

Fig. 14 - Shop Screen

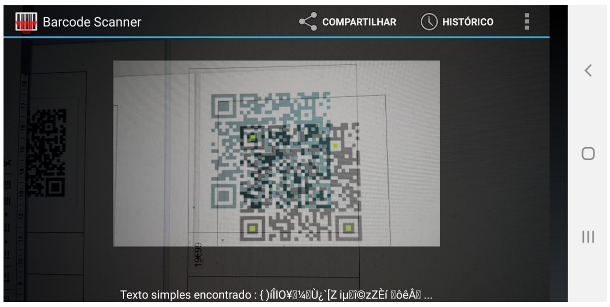
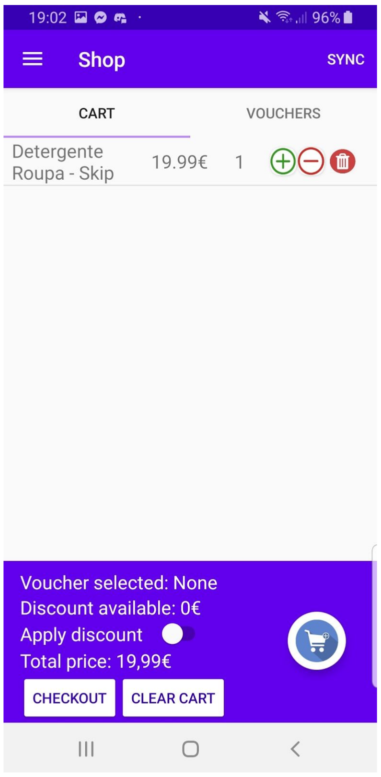


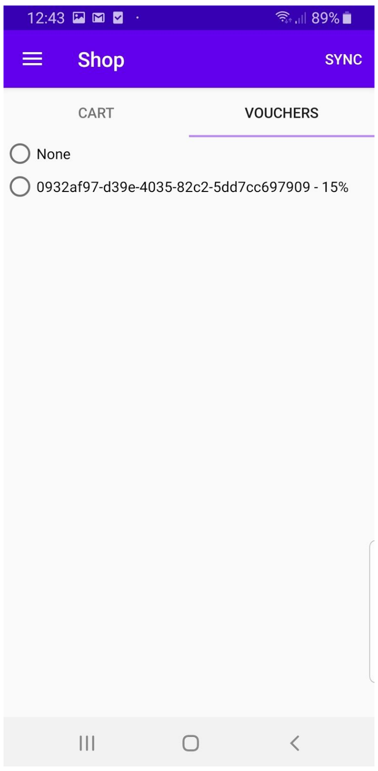
Fig.15 - Barcode Scanner



Detergente Roupa - Skip	19.99€	1

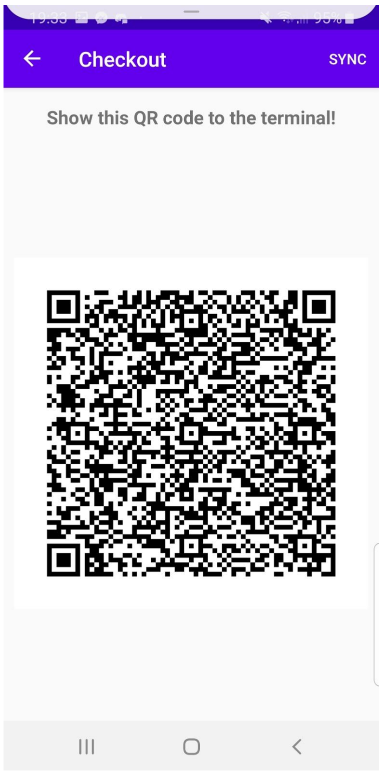
Voucher selected: None
Discount available: 0€
Apply discount
Total price: 19,99€

Fig.16 - Cart after adding an item



None
0932af97-d39e-4035-82c2-5dd7cc697909 - 15%

Fig.17 - Vouchers List



Show this QR code to the terminal!

Fig.18 - Transaction QR code

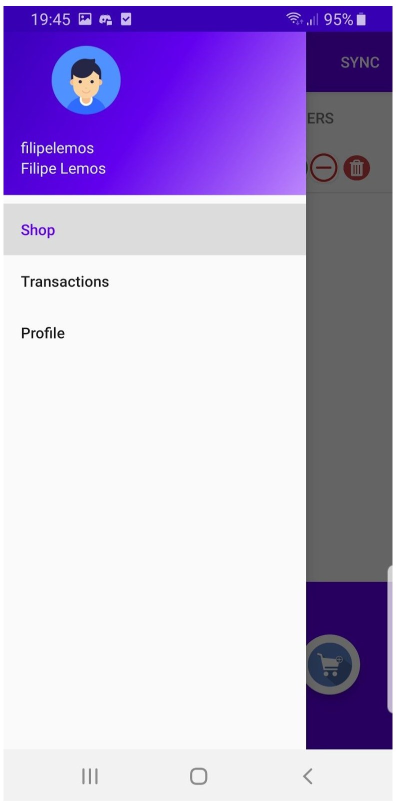
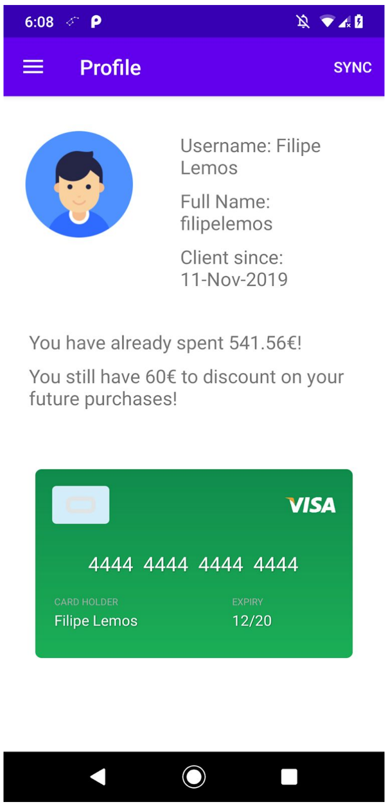


Fig. 19 - Side Menu



Username: Filipe Lemos

Full Name: filipelemos

Client since: 11-Nov-2019

You have already spent 541.56€!

You still have 60€ to discount on your future purchases!

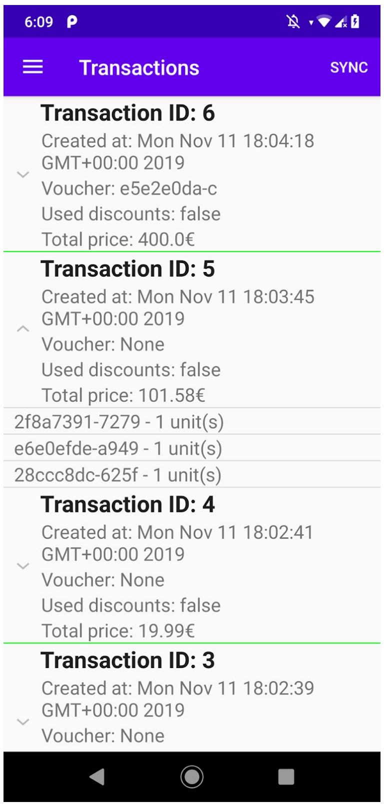
VISA

4444 4444 4444 4444

CARD HOLDER Filipe Lemos

EXPIRY 12/20

Fig.20 - User Profile



Transaction ID: 6
Created at: Mon Nov 11 18:04:18 GMT+00:00 2019
Voucher: e5e2e0da-c
Used discounts: false
Total price: 400.0€

Transaction ID: 5
Created at: Mon Nov 11 18:03:45 GMT+00:00 2019
Voucher: None
Used discounts: false
Total price: 101.58€
2f8a7391-7279 - 1 unit(s)
e6e0efde-a949 - 1 unit(s)
28ccc8dc-625f - 1 unit(s)

Transaction ID: 4
Created at: Mon Nov 11 18:02:41 GMT+00:00 2019
Voucher: None
Used discounts: false
Total price: 19.99€

Transaction ID: 3
Created at: Mon Nov 11 18:02:39 GMT+00:00 2019
Voucher: None

Fig.21 - Transactions Screen

Once the user wants to finish the shopping, he may show the generated QR code (Fig.9) to the terminal App (Fig.22) and wait for the green screen (Fig.23) to complete the transaction. If the user tries to use the same voucher (before synchronizing the application data), a red screen (Fig.24) will be displayed on the terminal App, and the purchase won't be validated.

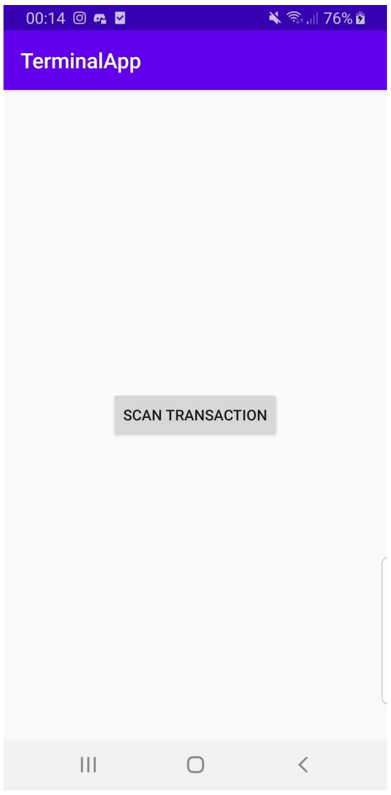
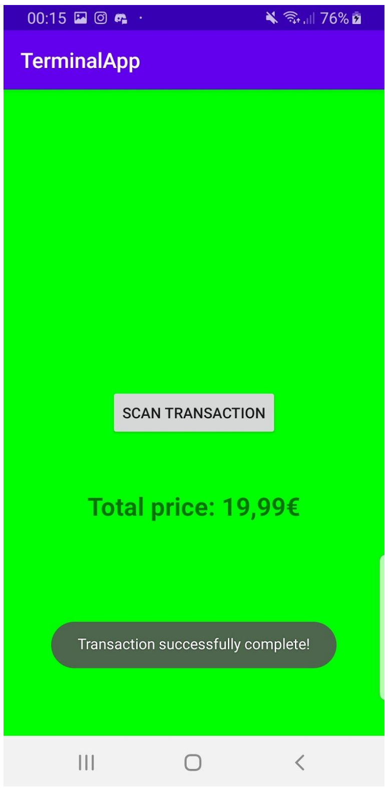


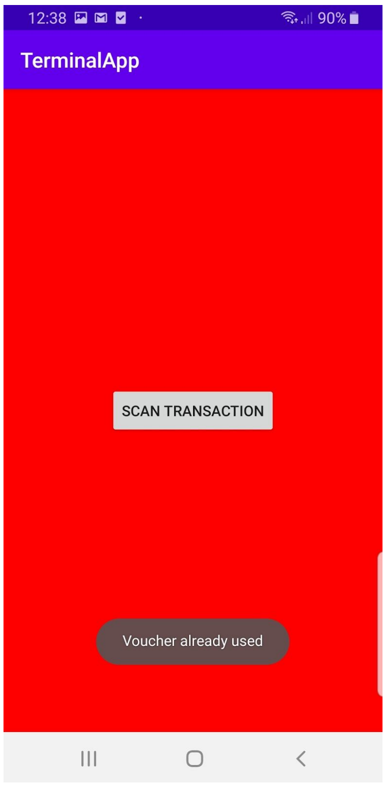
Fig.22 - Terminal App



Total price: 19,99€

Transaction successfully complete!

Fig.23 - Successful Transaction



Voucher already used

Fig.24 - Invalid Transaction

5.Conclusion

With this work, the group realized the importance of a well structured architecture in the development process in order to achieve a good final result. We were able to learn the basic operation of applications developed with *Android Framework* as well as the main problems associated with them (compatibility between versions, data persistence on screen rotations, communication between views and navigation across the application).

The final project provides a set of applications that can interact with each other in order to create a shopping and payment system for an electronic supermarket.

Despite this, as always, there can be enhancements done to further elevate its quality and performance. In this case, we can develop automated tests to verify that all functionalities are operational, as well as implementing a HTTPS connection in the server.

6.Bibliography

- Android Navigation
  - https://developer.android.com/guide/navigation
- Android Cryptography
  - https://developer.android.com/guide/topics/security/cryptography
- Android Keystore
  - https://developer.android.com/training/articles/keystore
- Android LiveData
  - https://developer.android.com/topic/libraries/architecture/livedata
- Android ViewModel
  - https://developer.android.com/topic/libraries/architecture/viewmodel
- Android Fragments
  - https://developer.android.com/guide/components/fragments
- Nodejs Crypto
  - https://nodejs.org/api/crypto.html
- Client-Server Architecture
  - https://en.wikipedia.org/wiki/Client%E2%80%93server_model
- Page of the curricular unit - CMOV
  - https://paginas.fe.up.pt/~apm/CM/
- Docker
  - https://www.docker.com/
- Docker-compose
  - https://docs.docker.com/compose/overview/
- Expressjs - API
  - https://expressjs.com/en/4x/api.html
- Postgres Documentation
  - https://www.postgresql.org/docs/11/index.html

7. Resources

7.1. Software Used

- *Android Studio provided by Google*
- *Visual Studio Code* provided by *Microsoft*
- *Docker and docker-compose*

7.2. Setup Project

It's assumed that the *Android Studio* is already installed. The apks of the android application wasn't generated because the server ip address changes in the local computer(because it's tested locally and the server wasn't deployed).

7.2.1. Installing Docker and Docker Compose

Before starting you'll need to have **Docker** and **Docker Compose** installed on your PC. The official instructions are in Install Docker and in Install Docker Compose.

Note: If you are getting permission error on the docker run hello-world or if you get a warning ".docker/config.json: permission denied run..." follow these instructions.

7.2.2. Configured containers

To start the environment :

```
$ docker-compose up
```

Note: To interact with the containers see *docker-compose.yml* and what ports are exposed.

7.2.3. Use Android Applications

Before running the projects it's necessary to change the server IP in the Customer and Terminal Application, to do that follow the next steps:

1. Go to *AcmeSupermarket/app/src/main/java/services/repository/Constants.java*
2. And changes the *ACME_REPOSITORY_URL* to your local IP.
3. Go to *TerminalApp/app/src/main/java/com/example/terminalapp/Constants.java*
4. And changes the *ACME_REPOSITORY_URL* to your local IP.

Note: the android devices need to be connected to the same network as the server.

Run the projects in the android studio.