ACME Supermarket

Project Report

Integrated Masters in Informatics and Computing Engineering

Mobile Computing
Academic Year 2019/20, 1st Semester

Daniel Ribeiro de Pinho - 201505302
Rúben José da Silva Torres - 201405612

Faculty of Engineering of the University of Porto
Rua Roberto Frias, s/n, 4200-465 Porto, Portugal

November 20, 2019

Contents

1 Introduction

This software was developed throughout the duration of the first project in the Mobile Computing curricular unit. This consisted in the development of a system for a fictitious supermarket (the ACME Supermarket), where users can pick up items from the shelf, scan them using QR codes, and then proceed to leave the store after a simple checkout method.

1.1 Overview of system modules

This system is composed of three main programs:

The customer app: This is the app that is installed on any supermarket customer's phone. It allows the customer to scan the QR codes present on items, review their current cart, consult their past transaction history, and checkout of the store.

The supermarket terminal app: This app is installed on the supermarket terminals. They read the customer's checkout QR code and communicate with the ACME server to conclude the transaction and open the supermarket exit doors.

The ACME server: This program runs on the ACME servers and handles all communications between the two Android apps, as well as dealing with the supermarket database maintenance.

These three programs are further explained in more detail in their respective sections.

1.2 Code structure

1.2.1 Android App structure

The codebase of the two Android applications was structured using the **Model-View-Presenter** (MVP) architectural pattern. This model, similarly to the Model-View-Controller (MVC) pattern, has three main components:

Model: The Model is the component that handles the data. The classes that represent this component usually have either some sort of local data or have connections to the server where the data is stored.

View: The View is related to the user interface. It receives the data that is to be presented by the Presenter component, and does not have any processing operations.

Presenter: The presenter presents the View with the data to be shown to the user, acting as the middleman between the Model and the View. The Presenter also performs any processing operations that may be required when going from the Model to the View, and vice-versa.

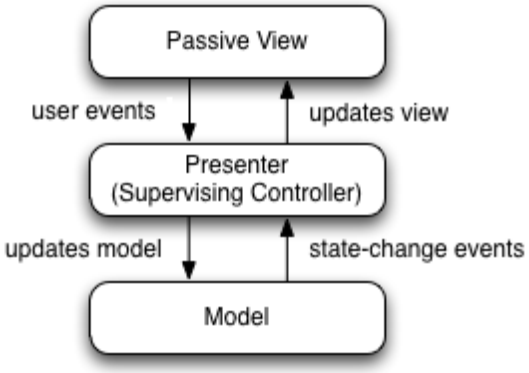


Figure 1: A diagram of the interactions of MVP components.

In this, each package in the applications has a class for the Model, a class for the View and a class for the Presenter.

The classes for the View are represented by the `Activity` classes themselves, and `View` interfaces were created instead, which are then implemented by the Activity classes.

The Model is represented by the `Interactor` classes (which *interact* with the server), with support from the classes in the `model` package in the supermarket customer app (which include auxiliary functions and classes to connect with the server and classes that represent data structures).

1.2.2 Server structure

In contrast with the structure of the Android apps, the server uses the **Model-View-Controller** (MVC) architectural pattern. This architecture has three types of components:

Model: The central component of the pattern. It is the application's dynamic data structure, independent of the user interface. It directly manages the data, logic and rules of the application.

View: Representation of the information. In the server's case, the View is represented by the routes.

Controller: Accepts input and converts it to commands for the model or view.

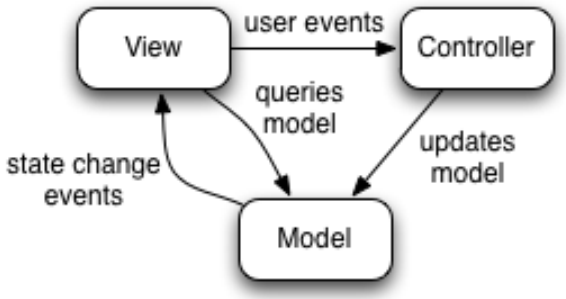


Figure 2: A diagram of the interactions of MVC components.

2 Customer app

The customer app codebase is divided into eight packages: `mainmenu`, `cart`, `checkout`, `history`, `login`, `register`, `model`, and `utils`. The first six of these are directly related with an app Activity, while the former two provide support functions.

2.1 The main menu

The main menu can be found in the `mainmenu` package. The activity uses the `activity_main` layout and has a very simple user interface, with a large button to access the QR code scanning function and two smaller buttons that lead to the cart and history activities. This menu only appears after some form of authentication has been done.

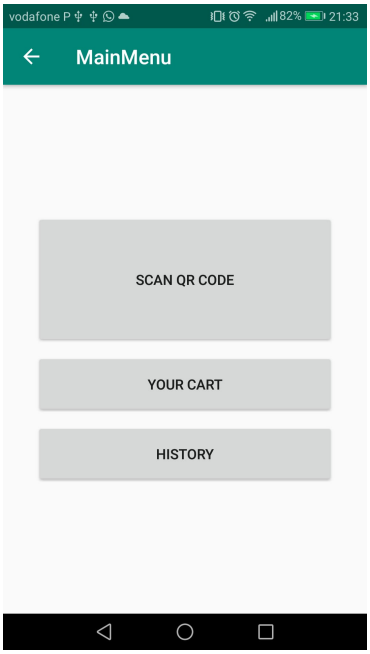


Figure 3: A screenshot of the main menu.

2.1.1 Scanning items

The system does not have an in-app scanner implemented. However, when the user taps on the QR scan button, the application calls forth an intent looking for a `zxing`-compatible activity to scan QR codes. If the user has an application installed on their device that is compatible, that activity is loaded; otherwise, the user is invited to download an app from the Google Play store.

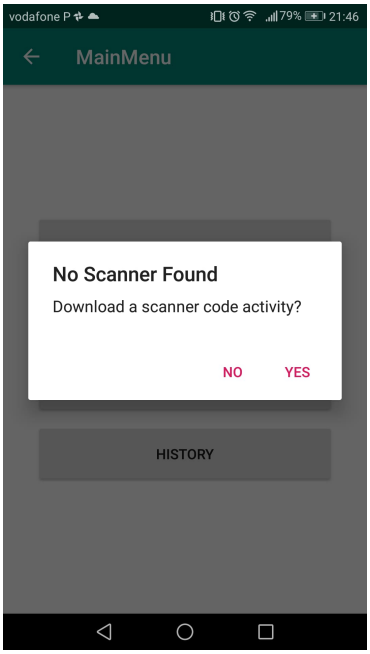


Figure 4: An error message informing the user they don't have a scanner installed on their device.

After scanning a valid QR code, the activity receives the contents from the external app and processes the data contained within, making use of the cryptography functions present in the support packages, updating the in-app cart.

2.2 The shopping cart and the checkout process

2.2.1 Cart management

The shopping cart can be found in the `cart` package. The activity uses the `activity_cart` layout and contains options for the user to decide whether they want to use a voucher and/or the balance available on their account (implemented using the `Switch` widget), as well as a list of the items already present in the cart.

This list was implemented using a `RecyclerView`; in order to show the items linearly, a `LinearLayoutManager` was used. The `RecyclerView` uses the `CartItemAdapter` and `CartItemViewHolder` classes in order to fetch and display information to the `recycle_cart_item` layout.

In order to provide a better user experience, the voucher and balance switch labels show how many unused vouchers the user has, as well as dis-

playing the remaining balance in the user's account. If the user does not have any available vouchers or balance, the respective switch is disabled and greyed out.

Finally, the checkout button displays the total price of the items in the cart, and it can only be used if the cart has at least one item.

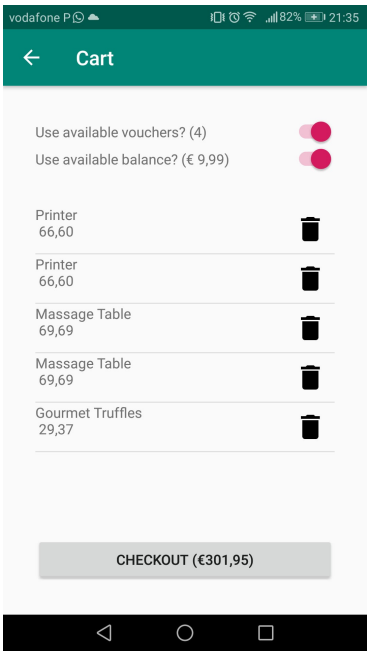
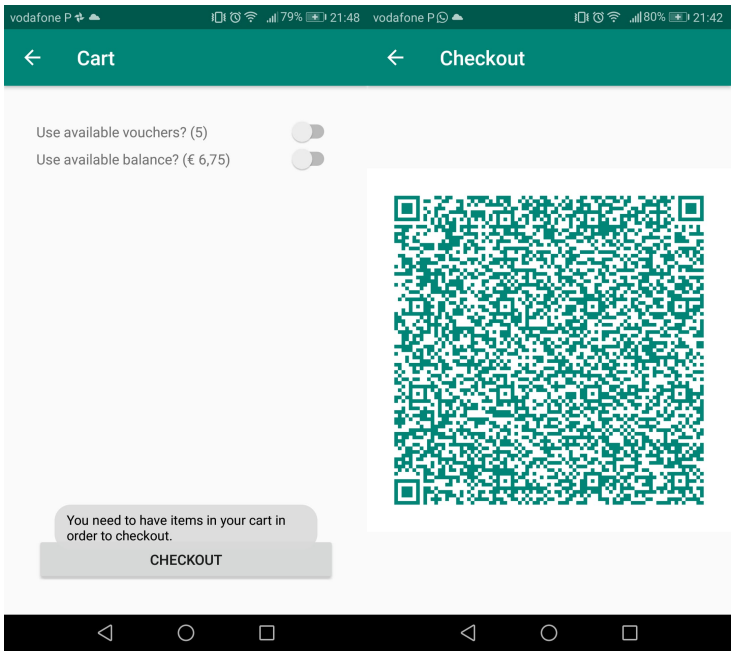


Figure 5: Example of the Cart Activity

2.2.2 Checking out



(a) The app doesn't allow checking out with an empty cart.

(b) A QR code ready to be scanned with the Terminal app.

Figure 6: The checkout process.

After the user scans one or more items, they are able to checkout. The checkout button present in the Cart activity leads to the `CheckoutActivity`, which can be found in the `checkout` package.

Com recurso à classe que representa o checkut, assina com as chave privada SHA256withRSA, criada na altura de registro ou login. Esta assinatura e o texto que utilizado para fazer assinatura são adicionados a esta classe. Posteriormente esta class em Json, com recurso Gson

2.3 The client history

The client history can be found in the `history` package. In the same vein to what happens in the Cart activity, the `HistoryActivity` class uses a RecyclerView object in the `activity_history` layout in order to show information in a modular way.

This RecyclerView is able to display objects using a `LinearLayoutManager` and the `HistoryItemAdapter` and `HistoryItemViewHolder` classes, which receive the information that is fetched from the server and display it. This process is aided by the `HistoryInteractor` and `HistoryPresenter` classes.

Each past transaction is shown using the item UUIDs and their price, followed by the total cost of the transaction and the account balance that was used for the discount. Each transaction is shown in chronological order, from the oldest to the most recent.

Figure 7: Example of the History Activity

2.4 Authentication

The user is authenticated using a JWT token. This token is generated by the server using HMAC with SHA-256 as the algorithm, which is then saved on the device (using the app `SharedPreferences`) when the user peforms their registration on the system or logs in.

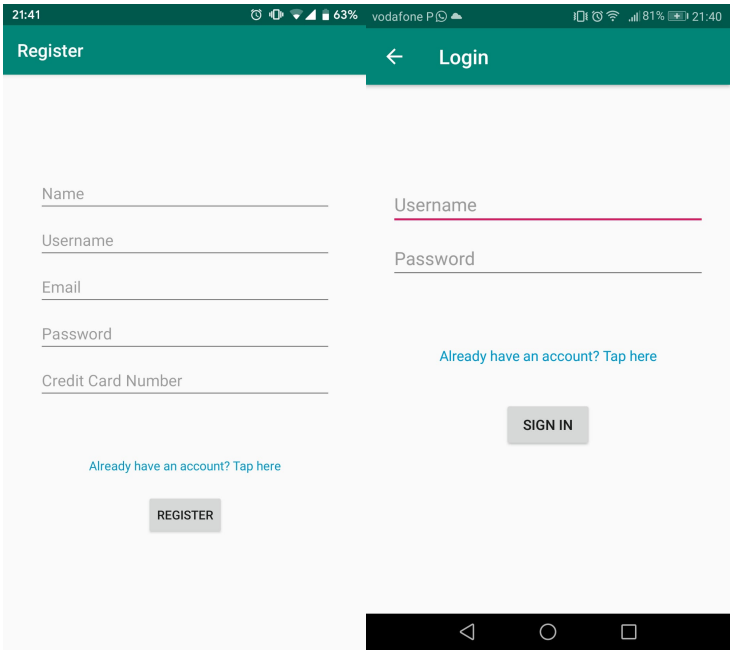
In the event that the application does not have the JTW token or if the token expires, the application requires the user to register or login again.

2.4.1 Registering and logging in

The register process is done in case the user does not have an account. If that is not the case, the user can log in instead, but the process is quite similar for either one of them.

The user finds a simple form asking for some basic information (such as a username, e-mail address, password, among other fields). After filling the form that is shown, the application generates the user's public and private keys and sends the former to the server, along with the field data. This is done so that the server can ensure that the checkout information corresponds to the user and does not debit the wrong user account.

After performing the registration/login successfully, the app receives the supermarket public key and allows the user to access the rest of the application. This key is used to read products' QR codes, since they are encrypted.



(a) Register Activity. (b) Login Activity

2.4.2 The local login process

Every time the app is launched, the user is required to login. However, in the event that the device the app is running on has biometric sensors (e.g. fingerprint sensor, facial recognition scanner), the user can access the app functions using the biometric data that is saved on the device if the

authentication token is present.

If the device does not have biometric sensors, the normal server login is performed instead.

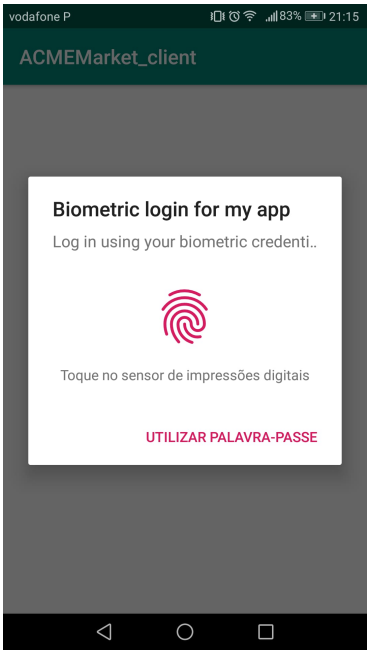


Figure 9: Fingerprint local authentication

2.5 The support packages

2.5.1 The `model` package

The model package implements and is responsible for all data representation in the app, including the data that are to be sent and received by the ACME server.

When the app needs to communicate with the server, the classes are converted to JSON (and vice-versa) using Google's GSON library. This communication is performed using the retrofit2 library. In order to perform these communications, there are the `SupermarketAPI` interface and the `Interactor` singleton, which implement the basis necessary to the server communication.

Each activity's `Interactor` identifies the server route that is needed and then calls it. These routes are declared in the `SupermarketAPI`, along with additional required information for their usage.

The `Interactor` present in the `NetworkLayer` package is a singleton with common functions called by the Activities' Interactors.

2.5.2 The `utils` package

The `DBinSharedPreferences`, `RSAKeys`, `Constants`, and `QRCodeSupport` classes compose the `utils` package, whose main function is to provide auxiliary functions to the app's behavior.

`Constants` has several objects that represent constants, which are referenced all over the rest of the application.

The `RSAKeys` class has functions that create keys, load keys and functions that aid in the conversion between Strings, keys, and vice-versa.

The `DBinSharedPreferences` class allows for the app to store objects as strings in the app's Shared Preferences, since otherwise important information could be lost.

Finally, `QRCodeSupport` has functions that help with dealing with the information in the ACME QR codes, like, for example, converting between floats and pairs of integers for item prices.

3 Supermarket terminal app

The terminal app is a simple app that has as its main function scanning the QR code generated by the client app in the Checkout activity and communicating with the server in order to effectively perform the purchase.

4 ACME server

4.1 The supermarket database

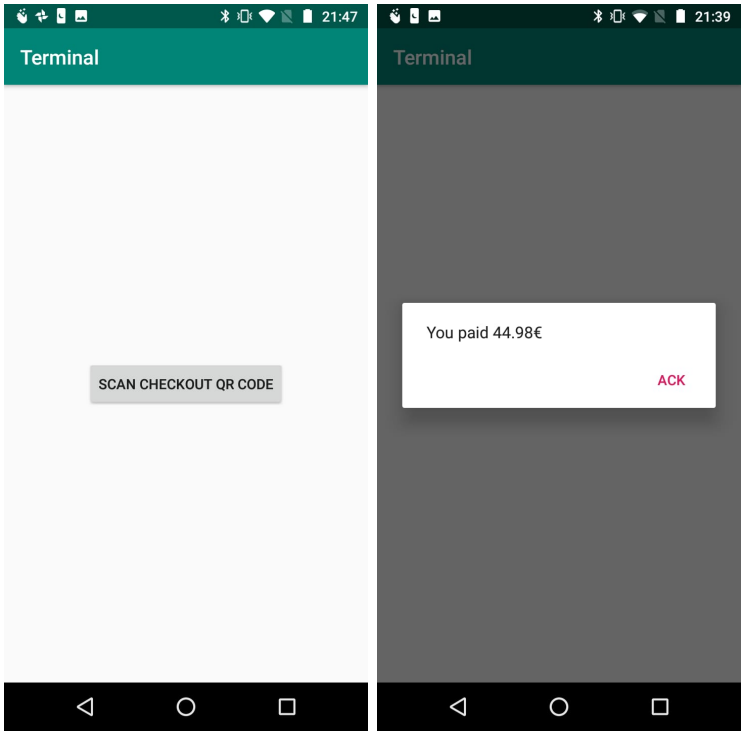
The database has several classers that essentially represent the normal functions of the app and a regular supermarket. Among these we find `User`, `ShoppingList` (representing a purchase), `Vouchers`, and `Products`.

A User has several ShoppingLists, which each have several Products. A User also has vouchers tied to their account.

4.2 Application-Server communications

4.2.1 Main server routes

`post /signup`: using when registering.



(a) The main terminal UI, with the checkout button. (b) A dialogue appears when the checkout is successful.

Figure 10: The Terminal app.

`post /login`: using when logging in.

`post /api/shoppingList`: used in the checkout process. It identifies the user using their UUID and verifies the request's signature. In case the verification succeeds, a ShoppingList is created in the database with the respective products and additional information. It is also responsible for creating new vouchers every 100€, and updating the user's balance if a voucher is used.

`get /api/shoppingList`: requires a JWT token. Using this token, we can know who made the request and we return the user's Shopping Lists.

`get /api/shoppingList/{id}`: requires a JWT token. Using it, we can know who the user is and return the ShoppingList identified by the ID in case it's owned by the token's user.

`get /api/voucher`: requires a JWT token. Using the token, we can identify the user and then return the unused vouchers.

4.2.2 Criptography

Criptography is used in this context in order to ensure security. In order to read the products' QR codes, we use the supermarket public key since they are encrypted with ACME's private key; this makes it impossible to create fake products.

The public key decrypts the information contained in the QR code and adds the product to the cart. This key is transmitted during the registry/login process, and the key generation algorithm used is RSA with 512 bits.

When checking out, the user information is signed with the user's private key. Along with that information, the string used in the signature and the signature itself are sent to the server.

The server will only perform the purchase in case the signature is verified with the user's public key. A key length of 512 bits was used, along with the SHA256withRSA algorithm.

The user's signature is proof that they have the private key that matches their public key. This means that the message is not secret and anyone can decrypt it. However, when they do so, they have proved that the user is the creator of the ciphertext, preventing fraudulent purchases.

5 Conclusions and future work

This project allowed us to learn a lot about Android development and put the knowledge we acquire during classes into practice. Mobile devices are becoming more prevalent every day and knowing how to work for those devices can make us become more ready for the labour world.

We realized that dealing with the information related to the Android documentation can be complicated, as there are several versions and a lot of components get deprecated over time and there's a constant need of renewal.

As for future improvements, we would like to improve the quality of the apps' look and feel for a better user experience. We would also like to employ Firebase on the system, as it would allow us to send voucher-related notifications, among other types. Firebase would also let us introduce the knowledge of services we learned in class.

All in all, this was a productive learning experience and we grew as people during this project.

References

[1] Google. *Gson: A Java serialization/deserialization library to convert Java Objects into JSON and back*. `https://github.com/google/gson`. 2019.

[2] Mike Potel. "MVP: Model-View-Presenter the Taligent programming model for C++ and Java". In: *Taligent Inc* (1996), p. 20.

[3] GWT Project. 2019. URL: `http://www.gwtproject.org/articles/testing_methodologies_using_gwt.html`.

[4] Rakshit Soral. *Architectural Guidelines to follow for MVP pattern in Android*. Dec. 2018. URL: `https://android.jlelse.eu/architectural-guidelines-to-follow-for-mvp-pattern-in-android-2374848a0157`.