November 17, 2020

Faculdade de Engenharia da Universidade do Porto

Mobile Computing (CMOV)

Assignment #1

The ACME Café Order System

Professor:

Miguel Monteiro - apm@fe.up.pt

Group:

César Pinho – up201604039@fe.up.pt

Rui Guedes – up201603854@fe.up.pt

Introduction

Within the mobile computing course scope, we were assigned the task of creating an ordering system for a cafeteria, with this system being constituted by three actors. These actors are a mobile phone application for the customer, a terminal at the cafeteria counter for receiving the customer's order, and a server acting as a rest service API.

In the client application, it is possible to access and browse all the available products in the cafeteria menu, create your custom order, and check your personal information, previous purchases, and available vouchers.

The counter terminal is only used to receive the customers' orders and send them to the server for validation. Validation can either be positive or negative. If positive, it shows the order number, the voucher used, and the customer's total amount. Otherwise, it only shows an error message, and the order is denied.

The current report describes the architecture and database schema adopted, the implementation details of each application developed, and a diagram representing the mobile applications' ways of use.

Architecture

The architecture chosen for this system was considered regarding Android programming's **best practices**: Such an approach allowed us to combine all the needed functionalities for data storage, both in the mobile application and on the server, and the connection and synchronization between these two databases to maintain the data integrity. Since this system's main application is the client's mobile application, the choice of architecture was mainly based on such an application.

The client's mobile application architecture is broken into different layers to follow the **separation of concerns principle**. As such, the architecture is composed of the UI (User Interface) module (**Activity / Fragment**), followed by the UI data source, supporting **Live Data** structures (**View Model**), and the module responsible for handling data operations (**Repository**). This last module, in which the View Model one delegates the data-fetching process, is responsible for acting as a mediator between different data sources, such as, in our case, a persistent model (**Room**) and a web service (**Retrofit**).

The architecture selected was a lot more straightforward on the server-side since it consists of an **Express** server that exposes a **REST API** for the client's mobile application. Such service acts as a point of access to allow the complete system to operate in a distributed way. Also, the REST API supports different endpoints according to the system requirements. The server possesses its own database (**Postgres**), which is considered to be the system's central database because the client's database functions only as a replica for a particular user.

We also decided to deploy the server by using **Heroku** so that more complex features could be added to the system, such as the capability of logging in into the system on different devices.

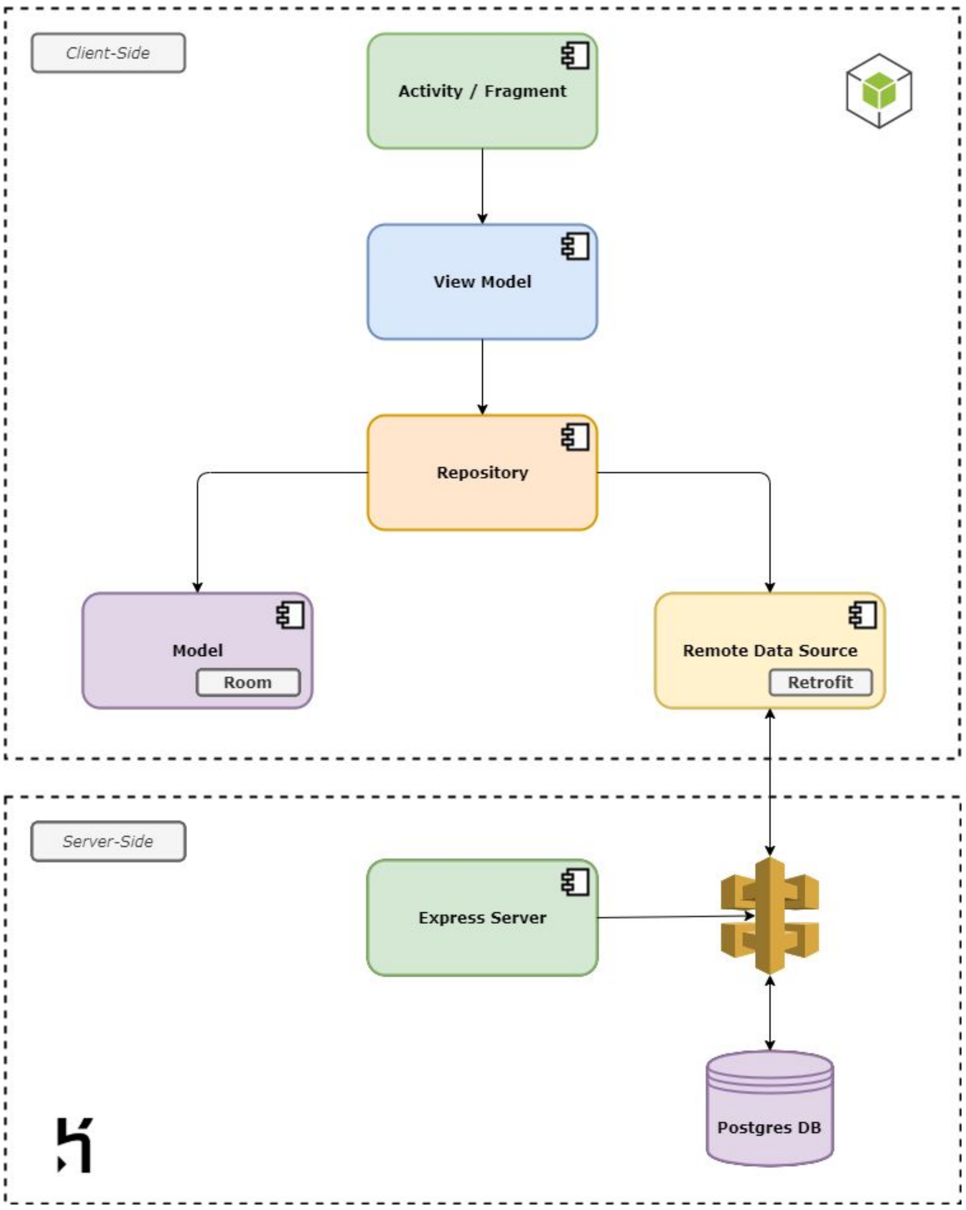


Fig. 1: Architecture Schema

Database schema

According to the system's requirements and due to some additional implemented features, to be referred later, the **server's database** is specified according to the following **UML diagram**.

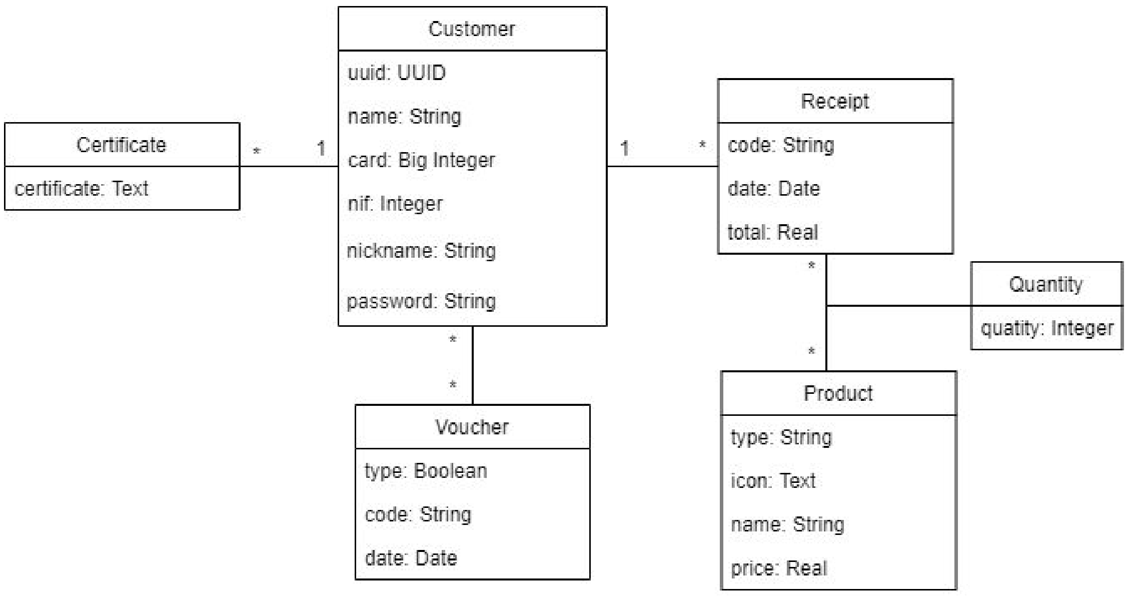


Fig. 2: Server Database Schema

The **application database** schema is not here presented since it only represents a replica of the server's database for a specific user. Consequently, this database is slightly different from the server's one. For instance, it does not need to have a particular table for the certificate objects storage since they are stored in the client's application **Android Key Store**. However, the remaining tables are equal to the ones presented in the diagram above.

Implementation

For this system´s implementation, two mobile applications were created together with a remote server. The following sections below explain in detail both the **technological stack** of the system and the main **implementation details**.

Server

The remote server was implemented using **Express**, a minimal and flexible ***Node.js*** web application framework that provides a robust set of features for web and mobile applications. The critical aspect of this framework selection is that it is excellent at optimizing **API's** construction.

The team also decided to deploy the server under **Heroku**, a platform as a service (**PaaS**) that enabled us to build, run, and operate our server entirely in the cloud. This decision resulted from additional requirements set for our application, such as logging in into the system on different devices and global access of information, which only made sense if our server could run in a distributed way.

Differently from the client's application database, the server's one is built under **PostgreSQL**.

On the server-side, there were also used several other libraries to fulfill the system requirements. Here is a list of all the additions libraries used:

- **bcrypt** - used for safely passwords storage and validation
- **node-forge** - used for cryptography operations such as the signature validation and public key extraction from the certificate body.
- **sequelize** - a promise-based Node.js ORM for Postgres and other SQL languages. Used for managing the database.
- **uuid** - used for the creation of the user's unique identifier

Client

For the mobile applications' development, we utilized the **Kotlin** programming language, native to android, to expand our knowledge area in terms of programming languages.

As previously mentioned, the **Retrofit** library was used to establish communication with the server with the help of the **Gson** library that allows the encoding and decoding of **JSON** objects on android.

The communication between the two mobile applications was implemented through the generation and scanning of **QR codes**. The client application generates the QR code containing all the information of the customer order needed for validation. The terminal application scans the QR code, decodes the information, and sends it to the server to carry out the validation and posteriorly processing. Then the server returns a response that can either be negative or positive. This last response is accompanied by the order number, the applied voucher, and the customer's total amount, which information is displayed on the terminal screen. For all manipulation of QR codes, the **Zxing** library was used.

In addition to this and to facilitate the inclusion of product images in the application, we used the **Picasso** library to download images without blocking the view and adjusting the image size according to particular needs. It also contains a mechanism for allocating cached images to avoid constant downloading, and the cache used is automatically managed by the library.

REST Service Description

Before proceeding to the REST Service description, it is worth mentioning that all the user input is being validated on the client-side and on the server-side. This way, we disable the possibility of forged requests being successfully performed. The server-side validation is a little bit more complicated than the one on the client's side, mainly due to integrity verifications. In contrast, there is only the need to check if the input is valid and according to the application expectations on the client's side.

[POST] /register

The /register endpoint is the one responsible for registering the customers into the application. It expects to receive a set of parameters (name, card, nif, nickname, password, and certificate) and validates them through a set of regexes. Also, to be valid, the user's nickname must be unique within all the records on the server's database. If everything succeeds, a UUID is generated, the password is hashed using the bcrypt library, and the information is stored correctly on the server's database. On a successful response, it is expected to be included information about the user, mainly its uuid. Otherwise, an error message with the 400 status code is generated and sent back to the user.

It is worth mentioning that on the client-side, the application is responsible for generating the Private and Public Key and storing them on the Android Key Store, allowing further and secure communications with the server. The client's side transmits its public key in the form of a certificate during the registration process.

[POST] /login

Similar to the registration endpoint, the /login one is responsible for authenticating a user. This endpoint expects three fields: the user's nickname, password, and Public Key in the form of a certificate. This last parameter is optional and only to be used by users whose current device does not contain information about their Private Key on the Android Key Store. This way, it is possible for the same user to use two different devices to register and authenticate into the application. That's why there is on the server-side a table dedicated to storing the user's Public Key. This endpoint's remaining operations are pretty straightforward as it verifies the user's credentials and, if valid, authenticates the user into the application.

[GET] /products

The /products endpoint is responsible for returning the available products regarding the last updated menu on the server's database. In terms of validation, it only checks for the presence of the "version" field. This field corresponds to the last update of the client's product database table. The endpoint compares this field with its products' internal data. Suppose needed, either new products were added or older ones updated. In that case, the endpoint returns a list of products so that the client's database can be updated, and the customer can see the most recent and updated information. Otherwise, the server informs the client that there is no need for updating its product information.

[POST] /receipts

Through the /receipts endpoint, we can retrieve all the receipts associated with a particular user. The endpoint expects three fields: an uuid, a signature, and a timestamp. This last field is used to avoid capture and replay attacks by determining the difference between this value and the current time. If higher than the specified fault tolerance, the endpoint will reject the request and respond with the 400 status code, marking it as a BAD_REQUEST due to an invalid timestamp. If the timestamp is valid, the endpoint checks both the client's existence in the database and the validity of the signature received by using the user's Public Key previously stored in the server's database for decryption.

After all the validations being made with success, the server then responds with all receipts' information, such as the id, date, total, and products' bought information, associated with a specific user.

Also, due to the implementation of logging into different devices, the receipts' information is never deleted from the server's database. This way, it is possible for any customer to access its receipts information on any device and at any point in time.

[POST] /vouchers

The /vouchers endpoint is responsible for returning all the vouchers associated with a particular user. This endpoint is identical to the /receipts endpoint as it expects the same fields as this last one and performs precisely the same validations. The main difference between these two endpoints is that the /vouchers one returns all the vouchers associated with a particular user regardless of their type.

[POST] /purchase

The /purchase endpoint can be considered the most complex endpoint of the entire server's service. This endpoint is responsible for handling customers' orders. It expects several different fields that compose a valid order: a list of products bought, a signature, an uuid, total amount, and optionally a voucher.

This endpoint starts by validating the request and ensuring that it is valid in terms of structure and expected fields. If everything matches the validity constraints, the endpoint checks for the user's existence in its database through the supplied uuid. Assuming that the user is valid, the endpoint computes the hash of the received request and checks for data integrity by validating the signature received using the customer's Public Key. After such validations, the server analyzes the voucher applicability, if part of the request. If valid, it applies it to the customer order and consequently deletes it from the server's database.

After the validation process is performed, the server inserts the customer order information into the database. Before responding to the received request, the server also checks for the need to generate new vouchers for the customer. If needed, generates and stores them into its database.

This endpoint has two different responses. Either it responds with a 400 status code informing that the order is invalid or responds with success sending back to the terminal application the order id, applied voucher, and total amount paid by the customer.

Application ways of use

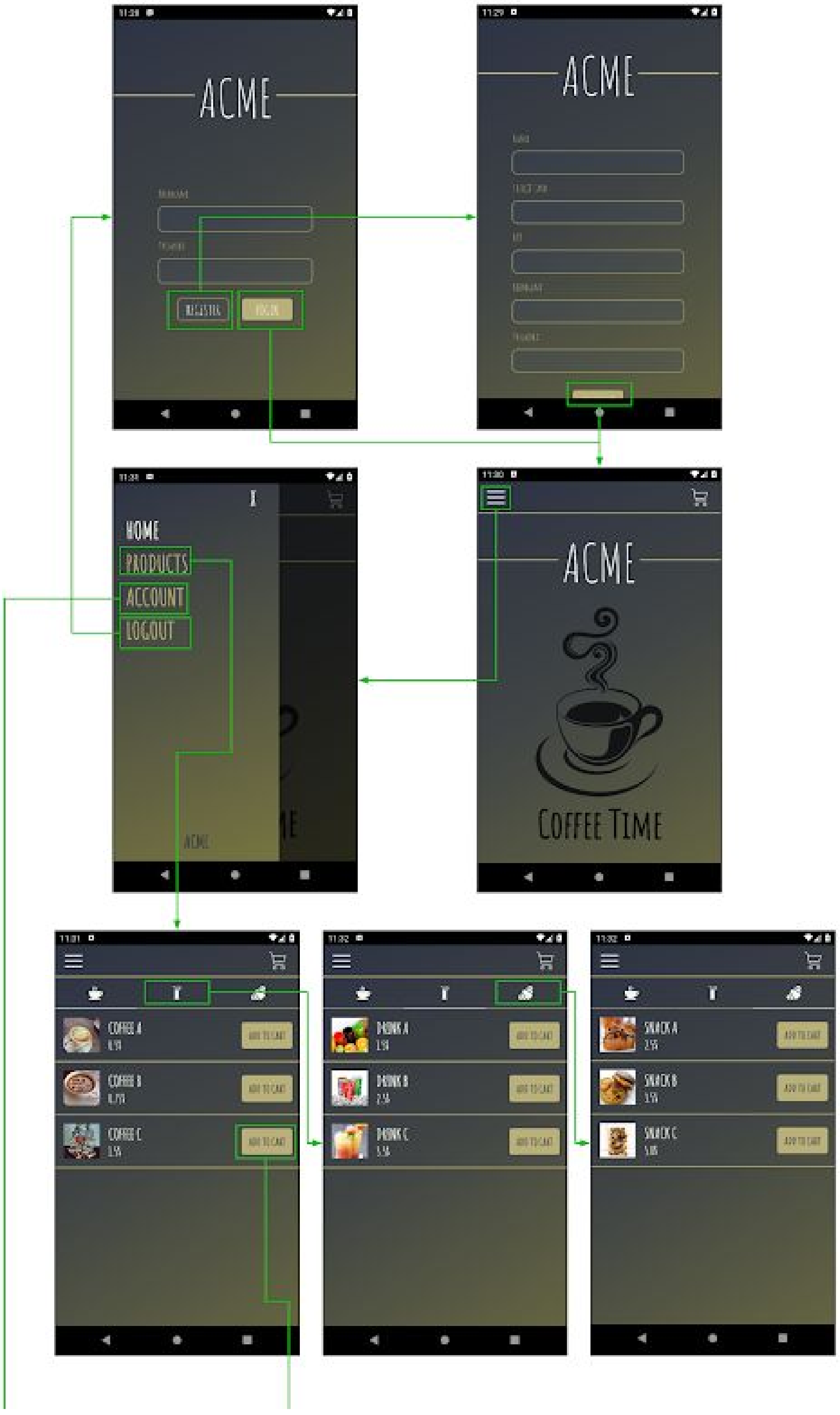
This section explains the interfaces' modus operandi for both applications. A diagram is presented for each application with links representing the redirections made and the touches/actions needed to achieve the corresponding screens.

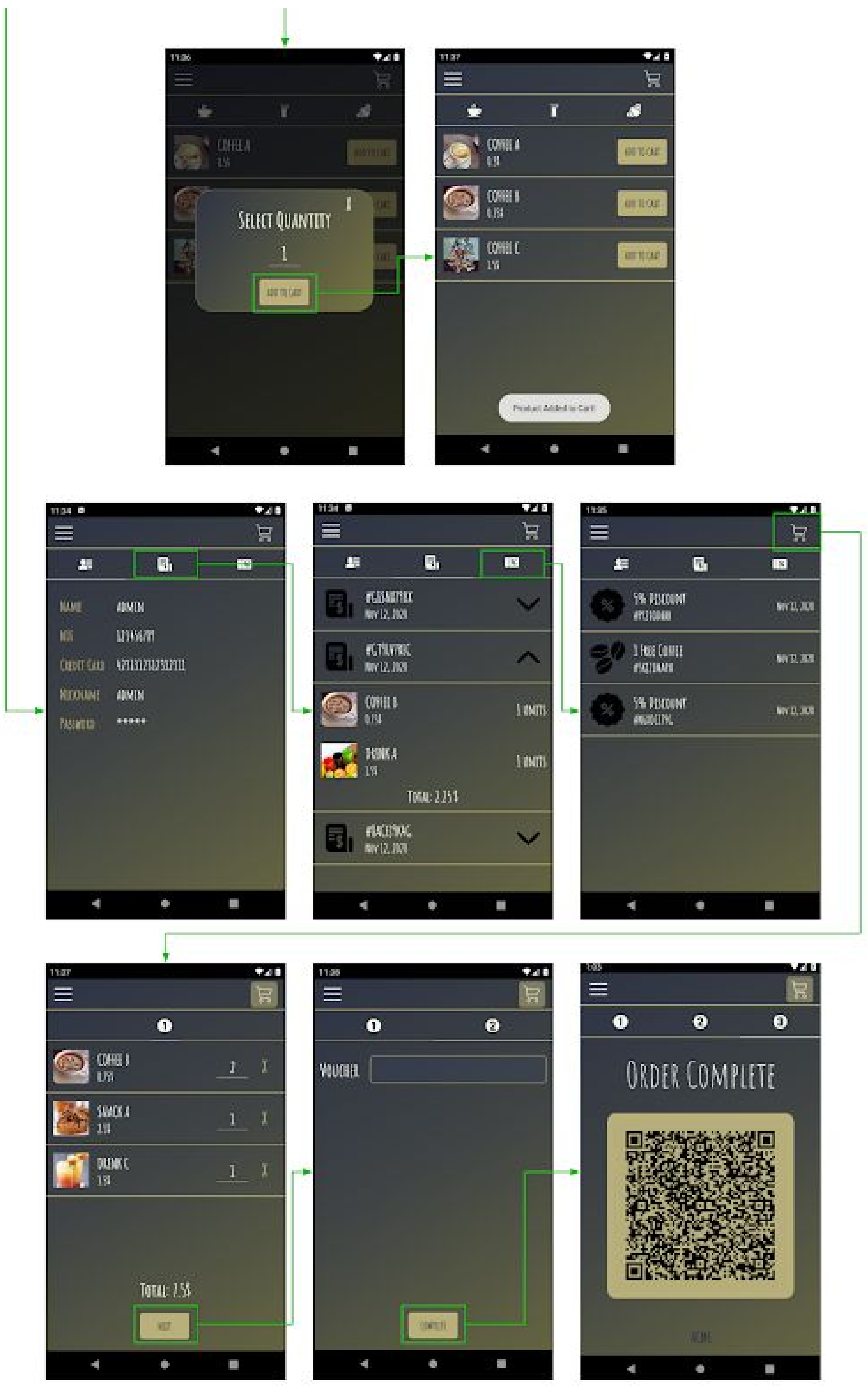
ACME App

The following diagram shows the flow of possible actions in the customer's application. Initially, we have the **Login** and **Register** screens accessible by anyone. After the login, we have an initial screen (**Home**) consisting of a menu (a drawer that allows navigation throughout the application and the logout action) accessible by clicking on the "burger menu" icon. We also have the cart button that provides access to the **Cart** screen. Access to the menu and the cart screens is possible from any application screen as long as the user authenticated.

Both the **Products** and the **Account** screens are composed of three tabs that can be accessed by clicking on the tab bar or swiping horizontally. The products are divided into categories (**Coffees**, **Drinks,** and **Snacks**), one per tab. On the **Account** screen, it is possible to access personal information, previous purchases (including the list of products purchased), and available vouchers.

On the Cart screen, it is possible to visualize the products added with the corresponding quantities. The cart is editable, being possible to edit product quantities or even remove them. The total amount to be paid is the reflex of the cart composition. An optional input field is presented to the user for voucher insertion. Whenever satisfied, the user can complete his order and hit the complete button that generates the corresponding QR code.


ACME
REGISTER
LOGIN
HOME
PRODUCTS
ACCOUNT
LOGOUT
COFFEE TIME
COFFEE A
COFFEE B
COFFEE C
DRINK A
DRINK B
DRINK C
SNACK A
SNACK B
SNACK C
ADD TO CART

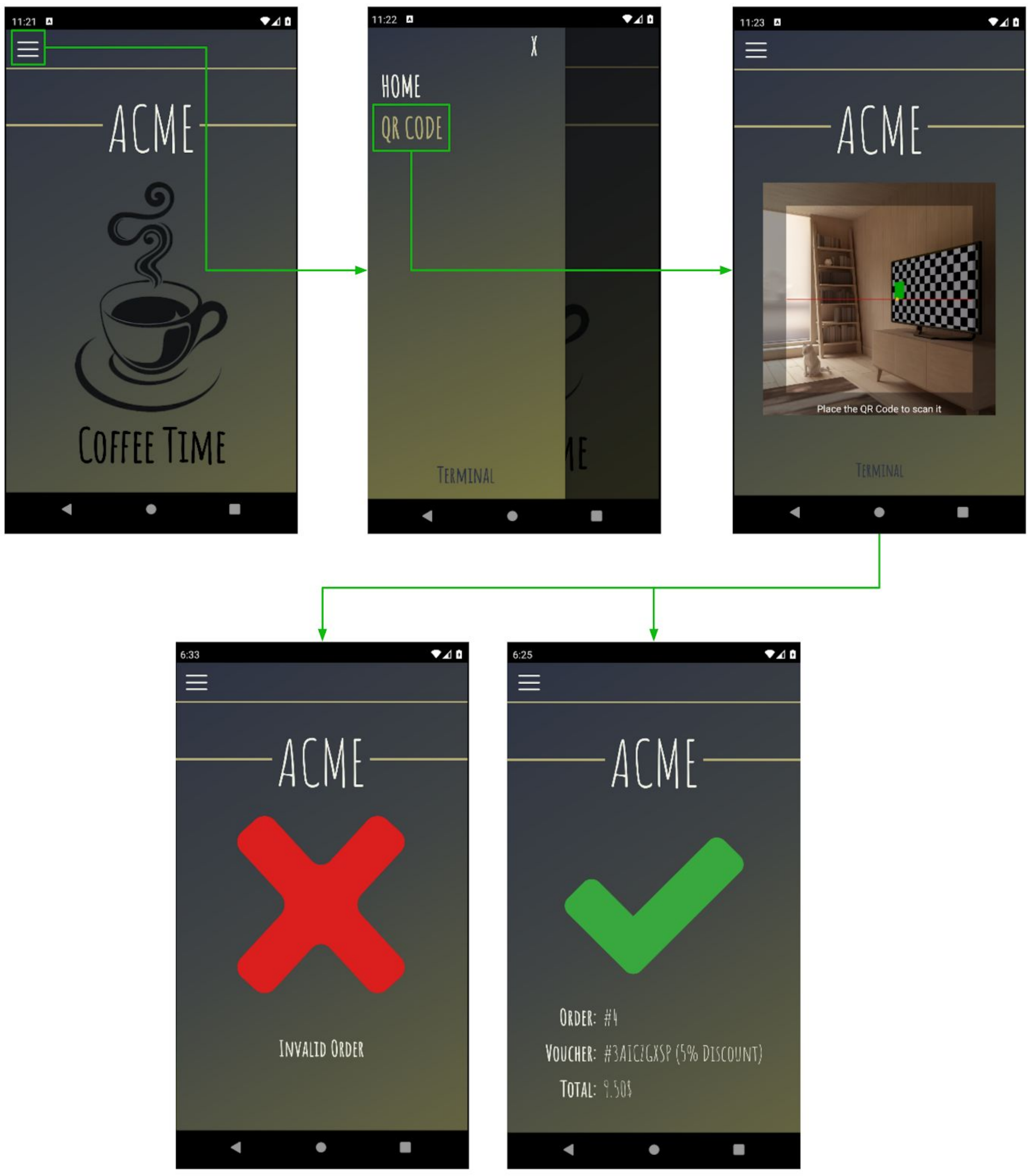


Select Quantity
1
Add to Cart
Coffee A
Coffee B
Coffee C
Add to Cart
Product Added to Cart
Name Admin
NIF
Credit Card
Nickname Admin
Password *****
Coffee B
Drink A
Total: 2.25$
5% Discount
1 Free Coffee
5% Discount
Coffee B
Snack A
Drink C
Total: 2.5$
Next
Voucher
Complete
Order Complete
Home

ACME Terminal

As its main features, the terminal application has the reading of the QR code and the visualization of the server's response. As such, its usage diagram is much simpler. Initial access is done directly to the **Home** screen, which contains a menu for navigation similar to that on the client's application but only with two options: the **Home** and **QR Code** screens.

The QR code scanner is automatically initialized on the QR Code screen. After scanning a QRcode, the information read is sent to the server for validation. On a successful server response, the order's information, including the order number, the voucher applied, and the total amount paid by the customer, is displayed. Otherwise, it is shown an error message.



Order: #4
Voucher: #3AICZGXSP (5% Discount)
Total: 9.50$

References

- Android Guide - https://developer.android.com/guide
- Kotlin - https://play.kotlinlang.org/byExample/overview
- Android Key Store - https://developer.android.com/training/articles/keystore
- Architecture Best Practices - https://developer.android.com/jetpack/guide
- Express - https://www.express.com/
- *Node*Js - https://nodejs.org/
- Heroku - https://www.heroku.com/
- PostgeSQL - https://www.postgresql.org/
- bcrypt - https://www.npmjs.com/package/bcrypt
- node-forge - https://www.npmjs.com/package/node-forge
- uuid - https://www.npmjs.com/package/uuid
- Sequelize - https://sequelize.org/
- Retrofit - https://square.github.io/retrofit/
- Gson - https://github.com/google/gson
- ZXing - https://github.com/zxing/zxing
- Picasso - https://square.github.io/picasso/