Mobile Computing

Practical Assignment #1

Order and pay Android app for an electronics shop

The Acme Electronics Shop

FEUP, MEIC, 1st year, May 2022

Group #1

Diana Freitas	up201806230
José Rodrigues	up201708806
Juliane Marubayashi	up201800175

Index

1. Introduction

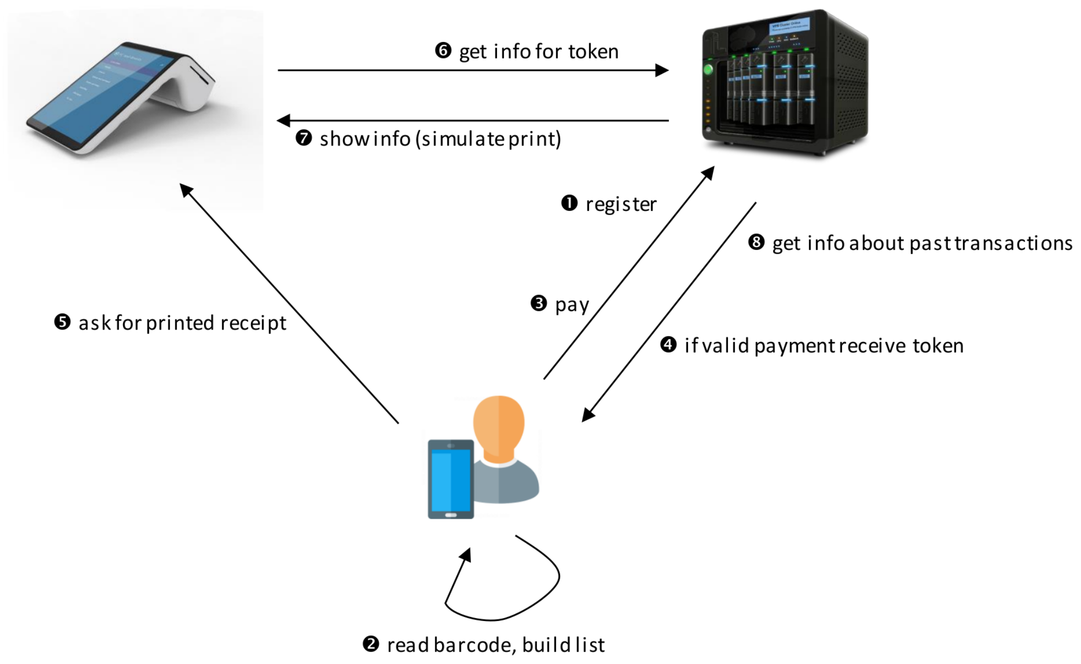
This project is, in a nutshell, a system to be implemented in a store that will allow a user to manage a virtual basket with his mobile device, by scanning products, paying for them, and accessing the purchase receipt.

The Acme Electronics Shop wants its customers to be able to access and purchase products in the store electronically, as well as to print the purchase receipts and check past purchases on the fly. The clients will be able to register in the application and then log into it, scan products from the store, gather them in a virtual basket, set their quantities, and checkout right on their mobile phones, completing the transaction digitally with the associated credit or debit card. When a user completes a transaction he will receive a QR code, which can then be used to print the receipt, in the Printer Terminal. Through their accounts, customers will also be able to analyze their past purchases.

In the following sections, this system will be explained in further detail. Details regarding how it was developed, the architecture, main features, security measures, and other relevant information will be discussed.

2. Architecture

2.1. Overview


❻ get info for token
❼ show info (simulate print)
❶ register
❽ get info about past transactions
❸ pay
❹ if valid payment receive token
❺ ask for printed receipt
❷ read barcode, build list

This system is composed of three different applications: the Customer Application, the remote REST service of the company (Server) and a Printer Terminal. It follows a Client-Server architecture where both the Customer Application and Printer Terminal must communicate with the server over Wi-Fi in order to authenticate and access resources or services.

All the communications with the server are done using the internet and the HTTP protocol, over Wi-Fi, through a REST API.

2.2. Customer Application

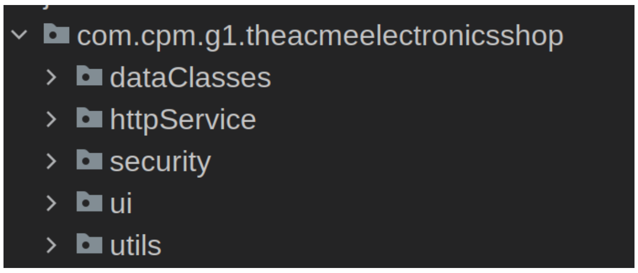
The Customer Application is an Android App that allows the customers of the Acme Electronics Shop to easily register themselves in the system, read barcodes of the shop's products, show details about the products and manage their shopping lists. It allows the users to pay for the items and print the receipt through a QR code, which is read by the Printer Terminal. It also allows them to consult past transactions.

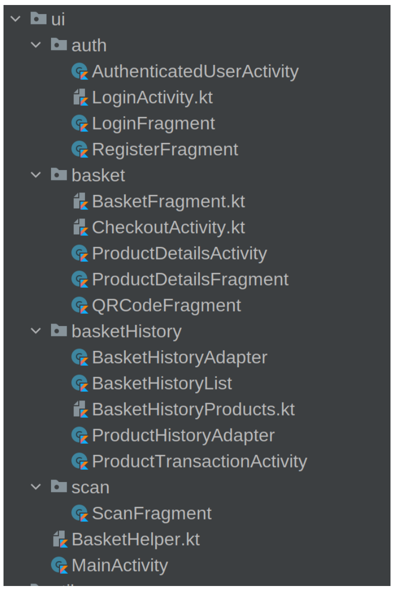
The application communicates with the supermarket's server through a REST API.

2.2.1. Code Structure

To better organize the code of the application, five packages were used:

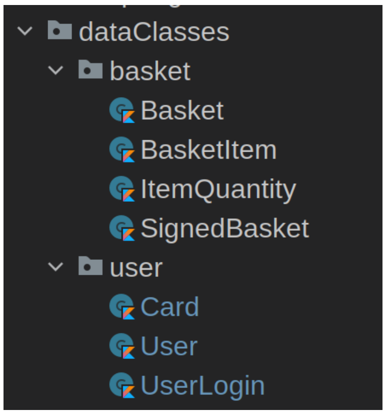
- ***dataClasses***: Includes all the data classes, each representing an entity;
- ***httpService***: Includes functions that can be used to send HTTP requests to the server and classes that implement *Runnable* and handle specific requests;
- ***security***: Includes the *Cryptography* class, which is used to generate the keys, sign and decrypt. It also includes the functions used to manage the *EncryptedSharedPreferences*, where the UUID of the user is stored;
- ***ui***: Contains all the activities and fragments that constitute the user interface:
    - ***MainActivity***: This represents the main activity of the application. It includes a bottom navigation bar, which can be used to move through the three main fragments of the application: *ScanFragment*, *BasketFragment* and *BasketHistoryList*.
    - ***scan***: A package containing the *ScanFragment*, responsible for reading and processing the barcodes and allowing the user to insert the scanned item in his basket.
    - ***basketHistory***: A package that contains the activities, fragments and adapter that were used to allow the user to see past transactions.
    - ***basket***: A package that contains the activities, fragments and adapter that were used to allow the user to see the items in his basket, to checkout and to generate the QR code, which will allow him to print the receipt.
    - ***auth***: A package that contains the activities and fragments used for the login, registration and logout.
    - ***BasketHelper***: A class that facilitates the interaction with the client-side database that stores the current basket of the user.
- ***utils***: includes a Singleton class with constants, mainly related to the keys.


```
com.cpm.g1.theacmeelectronicsshop
  dataClasses
  httpService
  security
  ui
  utils
```


```
ui
  auth
    AuthenticatedUserActivity
    LoginActivity.kt
    LoginFragment
    RegisterFragment
  basket
    BasketFragment.kt
    CheckoutActivity.kt
    ProductDetailsActivity
    ProductDetailsFragment
    QRCodeFragment
  basketHistory
    BasketHistoryAdapter
    BasketHistoryList
    BasketHistoryProducts.kt
    ProductHistoryAdapter
    ProductTransactionActivity
  scan
    ScanFragment
  BasketHelper.kt
  MainActivity
```

2.2.2. Data Representation

The package *dataClasses* includes all the data classes that were used to represent entities such as the Users, Products and Baskets. All of them implement the *Serializable* interface, which facilitates passing them between activities by using *Intents*.


```
dataClasses
  basket
    Basket
    BasketItem
    ItemQuantity
    SignedBasket
  user
    Card
    User
    UserLogin
```

Some of the data classes refer to the users of the application:

- **Card**: stores the details about a credit card;
- **User**: includes all the details about a user, such as name, address, among others;
- **UserLogin**: includes email and password and is used to facilitate the login operation;

Others are related to the products and baskets:

- **Basket**: describes the contents and characteristics of a basket. It includes the date and hour of purchase, the total, the uuid of the basket and a list of products, with identifiers and quantity of each product that was purchased;

- **BasketItem**: represents a product; It includes the id, name, brand, price, description, image URL and quantity of a product;
- **ItemQuantity**: includes the quantity and id of a product;
- **SignedBasket**: facilitates the checkout operation, by encapsulating the basket and the signature in a single data class.

In order to send the data stored in the instances of these data classes in the HTTP requests, it was necessary to transform them into Json Strings. For that, the Gson library was used.

2.2.3. Database

Given that the basket of the user is only sent to the server when the user actually proceeds with the purchase, an SQLite database was used to save the current basket of the user in the device. A single table, named *BasketItem*, stores the products that belong to the current basket of each user. Each object of this table associates a user with a given quantity of a product. This way, users that use the same device can save and retrieve their own current baskets.

```
db.execSQL("CREATE TABLE BasketItem(" +
       "_id INTEGER PRIMARY KEY AUTOINCREMENT," +
       "product_id LONG NOT NULL,"+
       "user_id TEXT NOT NULL,"+
       "quantity INTEGER DEFAULT 1, " +
       "date TEXT DEFAULT CURRENT_TIMESTAMP)")
```

The *BasketHelper* class, which extends the *SQLiteOpenHelper*, was used to facilitate the interaction with the client-side database. It provides methods to retrieve, insert, update and delete items of the basket of a specific user.

2.2.4. HTTP Requests

Considering that the communication between the Customer Application and the Server requires a large number of HTTP Requests, we decided to create a *httpService* package that encapsulates all the logic associated with the HTTP communication, avoiding code repetition and allowing us to use a similar logic every time a request needs to be sent to the Server. In this package, we can find multiple classes that implement Runnable interface, each of them representing a specific request that is sent to the Server in a separate thread. Each of these classes invoke the *sendRequest* function, providing the URI, the request type (POST or GET), the body, in the case of a POST request, and a callback function, which will be called when a response is received. The *sendRequest* function will interpret the response status code and will call the callback function, which receives as parameters a success flag and the response. The callback functions of each specific request are then responsible for showing a toast with an error message if the request failed or to proceed with his actions, normally by processing the response as a *Json* object.

2.2.5. Authentication and Input Validation

The *LoginActivity* is responsible for handling the registration and authentication of the user through the *RegisterFragment* and *LoginFragment*. When a user opens the application, the Login form (*Login Fragment*) is presented. He can authenticate by providing his email and password or he can move to the Register form (*Register Fragment*) where he is able to create a new account by providing his personal details, including email, credit card, among others.

When a user registers himself, the application generates a private and public key. It then sends a request to the Server's register endpoint (*/api/auth/signup*) with the data submitted by the user and with the public key.

Before sending the request to the Server, all the fields in the registration are validated. All of them are mandatory, some of them contain a minimum number of letters and all of them have a maximum number of letters. The limit of letters is implemented to avoid users filling the databases with huge amounts of information in an attempt to attack the server.

When the user inserts information that doesn't fit the format described in the field, a toast is shown and the field is highlighted in red.

The field expiration date of the credit card, as well as the email field, use regex to match the expected format (e.g. MM-yyyy). The Server also checks if the credit card has expired, and, if it has, it returns a 403 status code and the client will not be able to register. Otherwise, the user can register with success.

If the registration is completed with success, the user is redirected to the login screen, otherwise a toast is shown with an error message.

When the user logs in, the application sends an HTTP request to the respective endpoint (*/api/auth/signin*), which returns the UUID of the user if successful. This uuid is saved in the *EncryptedSharedPreferences*. If an error occurs (e.g. the password is invalid or the user does not exist), a toast is shown.

After logging in, the *MainActivity* starts and the current basket of the user is shown.

At any time, the user can decide to logout by clicking the button in the right corner of the action navigation bar. The user's UUID will be removed from the *EncryptedSharedPreferences* when the user logs out.

All the activities that allow the user to logout of the application extend the *AuthenticatedUserActivity*, which handles this operation.

2.2.6. Scan barcode

The *ScanFragment* allows the user to initiate a scan operation with his mobile device, launching an *Intents.Scan.ACTION* which will use the camera to start scanning in search for a 1-dimension *UPC_A* formatted barcode which corresponds to one of the existing products in the store.

Upon finding this barcode, it'll be scanned and the result will be processed into a string *prodID*, and this product ID will be then used in a get request sent to the server endpoint *api/products/id=<prodID>*.

This will be answered with a *JSONObject* that contains the information about the product that was just scanned (name, brand, price, description and image_url).

This information is then displayed on-screen using the *showProduct()* function, allowing the client to view the specifics about the product just scanned.

2.2.7. Basket

The *BasketFragment* displays the current basket of the user. For that, a request is sent to the endpoint *api/basket/products?ids=<ids>*, where *ids* is a comma separated list with the identifiers of the products of the user's current basket. The server sends back the details about each product of the basket, which are converted to *BasketItem* instances and stored in an array. An *ArrayAdapter* (*BasketAdapter*) is used to facilitate the conversion of the array objects into views.

For each item, the user can see the name, brand, price and image. He is also able to update the quantity or even to remove it from his basket. The total price is also shown in the bottom of the screen.

He can also find further details about each item by clicking on it. To implement this feature, the *BasketItem* associated with the clicked product is included in an *Intent* that is passed to the *ProductDetailsActivity*, which then displays the *ProductDetailsFragment*.

When the user presses the checkout button, a new activity - *CheckoutActivity* starts. A *SignedBasket* is built with the content of the current basket (just the identifiers, quantities and total) and with the UUID of the user, together with the signature of that same content. It is converted to a Json String and sent to the server (*api/basket/checkout*).

2.2.8. Checkout QR Code

At this endpoint, the server first validates the request parameters to see if they are correct and, if the basket isn't empty, it proceeds to verify the signature and validate the basket. If everything is correct, a unique *basketUUID* is assigned to the basket and saved in the database.

This *basketUUID* is then encrypted using the user's public key and sent back to the application as a response to the checkout request.

Back in the main application, having received the encrypted *basketUUID*, the app then decrypts it and sends it to the *QRCodeFragment*. This fragment will simply generate the QR Code image corresponding to this *basketUUID* using a *QRCodeWriter()*, encoding it into a pixel matrix (512x512), pixel by pixel, and then displaying this QR Code in an *ImageView*, which can be scanned by the Printer app.

2.2.9. History

In the app, users can access the history by selecting the appropriate option in the bottom menu bar. At first, it will display the list of all checkouts made by the user, each with the following information: the date, hour, number of distinct items purchased and its total value. In order to render this fragment, a history HTTP request is made to obtain the information about the checkouts performed by the user until that moment.

At first, it checks if the request body contains the user uuid and if it is signed. Once the presence of these fields is checked, the server verifies the signature. If the signature is checked with success and the content of the request body is trustable, the server sends all the previous transactions back to the Customer Application, with a status code of 200. Otherwise, a status code 400 will be sent, with a message indicating that the request could not be processed.

Each basket returned by the server contains a list of products, with their ids and quantities. If the user desires to check the content of a specific basket (e.g products name, price, quantity) a new request is performed: the List Products request.

2.2.10. List Products

Still in the app's history page, the user is allowed to click in a list item to check what products are involved in the purchase. At this moment, a list of items will be displayed, where each item describes a product with the following information: name, brand, quantity and price.

To obtain the information related to the product, a GET request is performed, where the parameters are the product's id. By receiving the request, the server sends the product's details. More information about the format of the HTTP requests can be checked at the postman documentation.

Although the HTTP request doesn't return how many products of each type were purchased, this information can be reused from history requests. In other words, the list of products is a subsequence of the history request and by calling an activity to display the list of products, the previous activity sends the result of the history HTTP request to the recently created activity. Thus, although the list products HTTP request just returns the characteristics of each product, the application already contains the quantity of each product.

2.2.11. Security

Asymmetric cryptography is used to ensure that the communication between the Server and the Customer Application is secure. The methods used to generate the keys, encrypt and sign are part of the *Cryptography* class, which belongs to the *security* package.

When the user registers himself, a pair of 512 bit RSA keys (public and private keys) are generated and securely stored in Android's *KeyStore*. The public key is sent together with the registration information to the Server, which stores it, and generates a 128-bit UUID that identifies the user. This value is kept securely in the *EncryptedSharedPreferences*.

This keys will are then used in three main scenarios:

1. Checkout:

   As briefly described in sections 2.2.7 and 2.2.8, when the customer is ready to pay, the device contacts the shop service, and sends the user *UUID* and the basket content, signed with the private key of the user, using *SHA256WithRSA* algorithm. The server must then verify the signature by using the customer's public key.

2. Receipt Token:

   If the checkout request is successful (the signature and basket are valid) the server will send a token that identifies the purchase. As this token gives access to the purchased goods, it is transmitted back encrypted with the user's public key and decrypted by the application with *RSA/ECB/PKCS1Padding* algorithm.

3. History:

   At any time, after registration, the customer can consult his past transactions. This requires the Server to be certain of the customer's identity. For that, it uses an approach similar to the one used for the checkout operation: the *userUUID* is signed with the private key of the customer and sent to Server, which will verify it before sending the transactions.

```
object KeyProperties {
   const val KEY_SIZE = 512
   const val KEY_ALGORITHM = "RSA"
   const val SIGN_ALGO = "SHA256WithRSA"
   const val ENC_ALGO = "RSA/ECB/PKCS1Padding"
   const val ANDROID_KEYSTORE = "AndroidKeyStore"
   const val KEY_ALIAS = "securityKey"
   const val serialNr = 123456789L
}
```

2.3. Server

The server was built using *MongoDB* database, *express* to support endpoints, and *docker* to support the use of these tools in any machine.

2.3.1 Code Structure

This project follows the recommended code structure for the *express* library, which contains the following folders:

- **models:** contains the format of each object that is stored in the non-relational database;

- **data:** contains a file *products.json*, that stores the products available in the shop and also a file *populate.js* that populates the *database* with the products listed in *products.json*;
- **routes:** contains the endpoints and routes of the server.
- **public:** contains a subfolder called *images* that store the images of the products.

2.3.2. Data representation

As described earlier, the data is stored in a non-relational database (*MongoDB*) and once the data is retrieved from the server it is processed as an object.

The server defines three types of data models: **user***,* **product** and **basket.**

The **user model** contains the information given by the user in the registration.

```
user: {
  _id: uuid // With 128 bits.
  pk: String,
  name: String,
  address: String,
  NIF: Number,
  email: String,
  password: String, // Encrypted
  card: {
    cardType: String,
    number: String,
    expirationDate: String,
  },
}
```

The **product model** contains the information about a specific product. The **populate** script, as mentioned in the previous section, uses the *products.json* to get the products information and stores it in the database using the format below:

```
product: {
   id: Number,
   name: String,
   brand: String,
   price: Number,
   description: String,
   image_url: String
}
```

The **basket model** contains the information about baskets that were purchased.

```
 basket: { userUuid: String,
    token: { type: String, default: ()=>{return uuid.v1()}},
    usedToken: { type: Boolean, default: false },
    products: [{
        id: Number,
        quantity: Number
    }],
    total: String,
    date: String,
    hour: String
}
```

As one might see, the basket stores its token and also a simple list of products that it contains. It also has a boolean, *usedToked*, which is used to guarantee that each purchase receipt is only printed once.

2.3.3. EndPoints

The endpoints are classified in three categories:

- Auth
- Basket
- Products

The full description of the API and endpoints can be checked on the postman documentation.

2.3.4. Security

As described in section 2.2.11, asymmetric cryptography is used to ensure that the communication between the Server and the Customer Application is secure.
When it comes to Cryptography, the Server uses the *crypto* module for two main operations: check signatures and encrypt sensitive content. The algorithms used for these operations match the ones used in the Customer Application.
In the Server, Cryptography is used in the following scenarios:

1. Checkout:
   When the client sends a checkout request to the endpoint */api/basket/checkout*, the Server must verify the UUID of the user and the basket content using the public key of the client which was sent upon his registration.

2. Receipt Token:
   After a valid checkout request, the server generates a unique token to give access to the purchased goods, and encrypts it with the user’s public key before sending it to the application.

3. History:
   When a user requests access to past transactions, the Server verifies the signature using the customer’s public key.

In addition to Cryptography, the Server also verifies the information of the User, particularly the validity of his credit card, in order to avoid fraudulent transactions.

The credentials of the customers are also stored securely by using *bcrypt*.

2.4. Printer Terminal

The printer terminal is simulated through another android application, representing the terminal, which will scan the QR code of the purchase and “print” the receipt for the client.

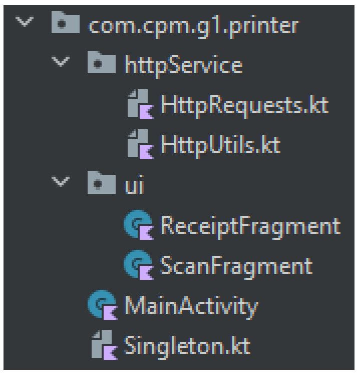
The process is simple: the application scans the QR code, sends a post request to the server with the basket UUID and either receives back a positive response, which holds the user’s name, NIF, the basket and a list of the products the basket contains with all the information about them, or receives a negative response, which could have many causes, such as this receipt having been printed already.

In case of a positive response, the printer app shows on screen a receipt with all the information about the purchase.

This process can only be accomplished once per basket, as requested.

2.4.1. Code Structure

Below, we can view the code structure, which is much simpler in comparison to the main application:



```
com.cpm.g1.printer
  httpService
    HttpRequests.kt
    HttpUtils.kt
  ui
    ReceiptFragment
    ScanFragment
  MainActivity
  Singleton.kt
```

- **httpService:** Just like the main app, these are the same functions, used to send and receive the requests to the server;
- **MainActivity**: This represents the main and only activity of the application. It holds and switches between the two fragments of the application, detailed below;
- **ui**: Contains the two fragments that constitute the user interface, the *ScanFragment,* responsible for the scanning of the QR code, and the *ReceiptFragment*, responsible for the receipt printing;

- **Singleton:** Just like in the main app, this class stores constants, but, in this case, it holds a single constant, with the *BASE_ADDRESS* of the HTTP requests.

2.4.2. Read QR code

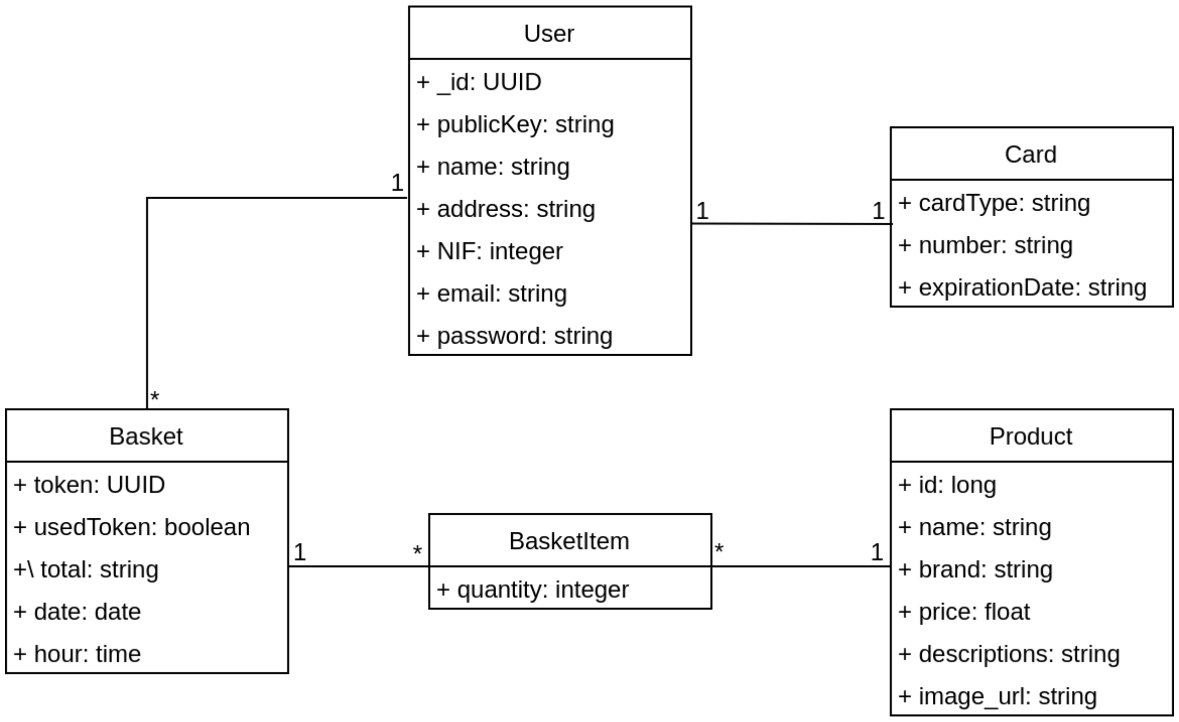
This happens in the *ScanFragment*, and similar to the one in the main application, it launches a scan action, this time scanning for a **QR_CODE** through the camera. Upon successful scan, it makes a post request to the server endpoint *api/basket/receipt* with the *basketUUID* it acquired from the QR code. This request will receive the information from the server and call the *changeToReceiptFragment()* function with the response as argument.

2.4.3. List basket details

After the *changeToReceiptFragment()* processes the information from the JSON response, it bundles it up, attaches it to the *ReceiptFragment*, and commits it onto the screen. In this fragment, the response will be handled, processing the JSON response accordingly, and the receipt will be built, line by line, and added to the receipt scroll view displayed on the screen to the customer.

3. Data Schema

Our overall system contains the following models: User, Card, Basket, BasketItem and Product.



Each **User** is characterized by his personal information: name, address and NIF; his credentials: email and password; public key, credit card information: type, which may be Credit or Debit, number and expiration date; and UUID, which identifies the customer in the Server.

A User can make many purchases (**Baskets**), which are characterized by an UUID, date, hour and by the total of the purchase. Each Basket has many **BasketItems**, which associate a Product with the quantity that was acquired. A Basket also has a flag that indicates if the token, which is used to print the receipt, was already used.

A **Product** has an id, a name, brand, price, description and image.

4. Main Features

4.1. Customer Application

- Login
- Register
- Logout
- Input Validation
- Key Generation and Usage (sign and decrypt with private key)
- Scan Barcode of a product
- Add scanned product to basket
- See current basket
- See details of basket product
- See history (previous transactions)
- See details about each old transaction
- Generate and view QRCode of the purchase
- HTTP Communication with the server
- Toasts when the server responds with error codes

4.2. Server

- Authentication (Register and Login users)
- Store and Retrieve Products
- Store and Retrieve Basket details
- Store and Retrieve past transactions
- Verify checkout request signature with client’s public key
- Credit card verification (always fails if expired or fails due to insufficient funds 5% of the time)
- Encrypt basket uuid with client’s public key

4.3. Printer Terminal

- Scan QR code of the client’s transaction
- Display the receipt with the details of the transaction

- The receipt can only be printed once

5. Applications usage and sequence of operations

The first interaction with this application starts with the authentication page (Fig.1). For an unregistered user, the first step is to create an account (Fig.2). After successful registration, the login screen will be presented (Fig.1).

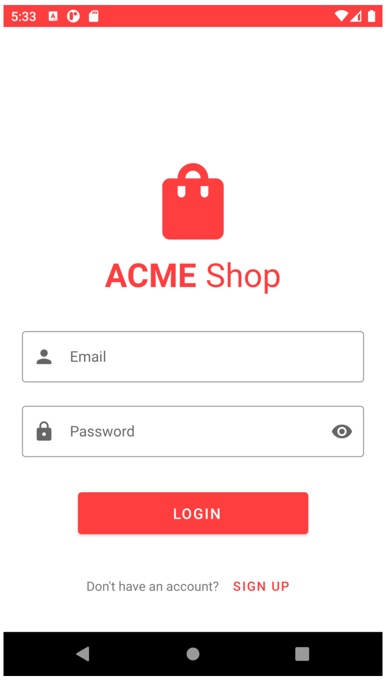


Fig.1: Login page

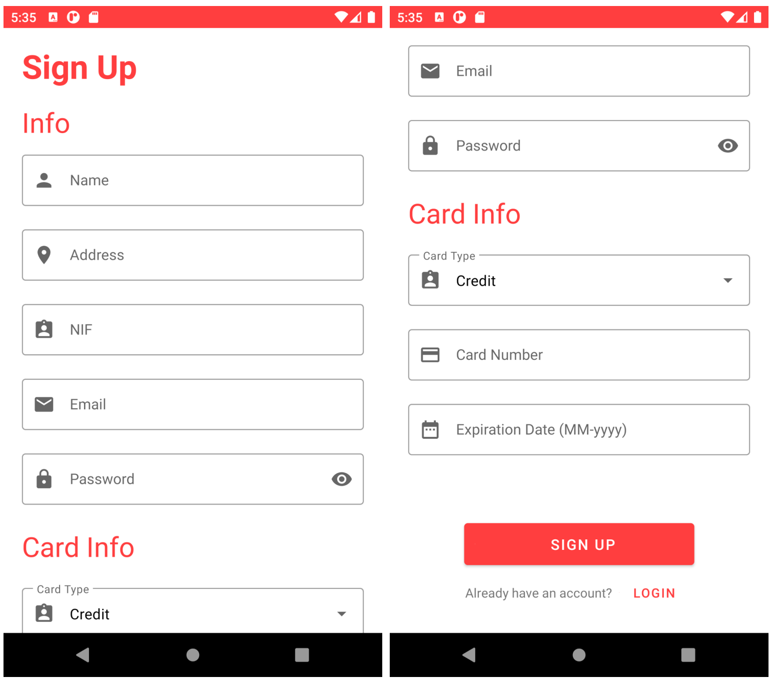


Fig.2: Registration Page

After the authentication, we move on to the main part of the application. We are taken to the main page, which has a top bar with the name of the current page on the left and a logout button on the right, and a bottom navigation bar that facilitates the navigation between the three main pages, the basket page (Fig.3), the scan page (Fig.4) and the history page (Fig.12).

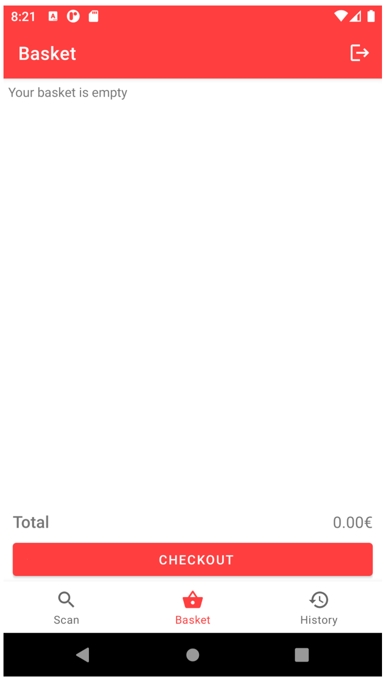


Fig.3: Basket page (empty)

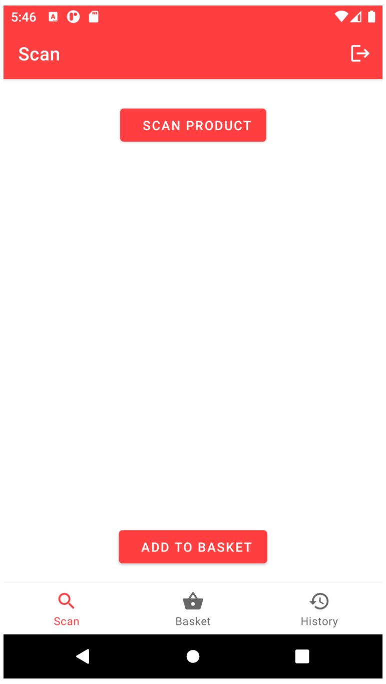


Fig.4: Scan page (empty)

The next logical step is to move to the scan page and hit the button **Scan Product**, which initiates the scan (Fig.5), and upon the successful scan of a barcode we are taken back to the scan page, now showing all the information about the product we scanned (Fig.6).

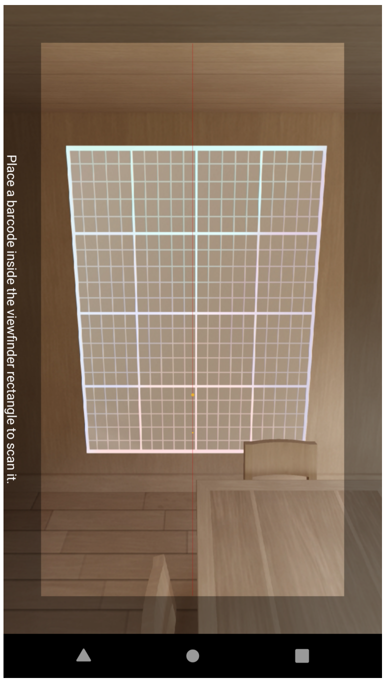
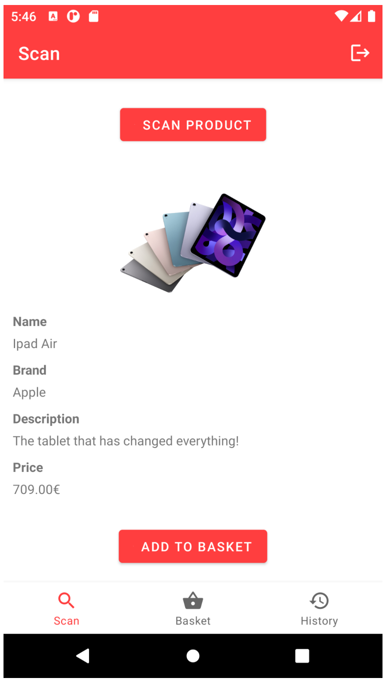


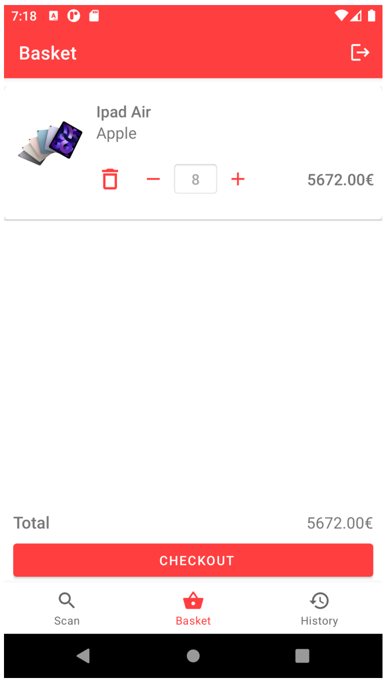
Fig.5: Scanning



Name
Ipad Air
Brand
Apple
Description
The tablet that has changed everything!
Price
709.00€

Fig.6: Scan page (filled)

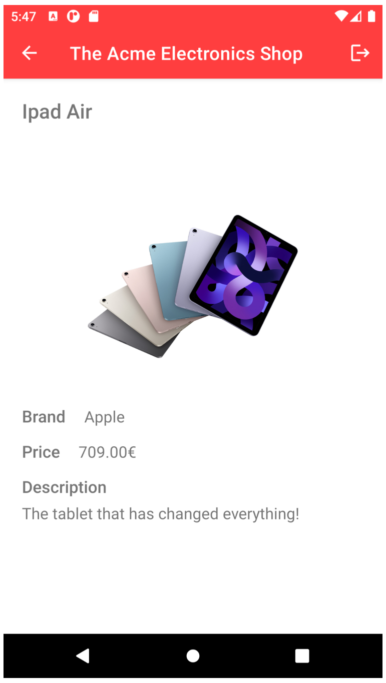
After scanning, we press the **Add to Basket** button to add the scanned product to the basket, and this takes us automatically to the basket page, where we can see the products we have selected and where we can change their quantity or remove them (Fig.7). We can also click on each of the items to see further details (Fig.8).



Ipad Air
Apple
8
5672.00€

Total 5672.00€

Fig.7: Basket page (filled)



Ipad Air

Brand Apple
Price 709.00€
Description
The tablet that has changed everything!

Fig.8: Product Info

At this point we're happy with our basket and want to pay for it. For that, we click the **Checkout** button, which will process our purchase and show us the QR code we can now use to print our receipt (Fig.9).

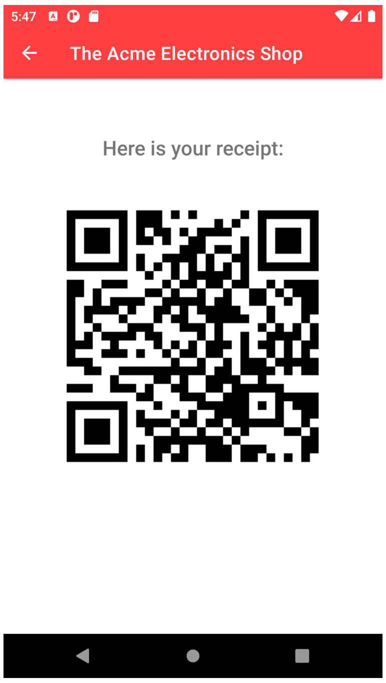


Fig.9: QR code receipt

Now, switching to the printer app, we have a simple display with the only button we need, the **Scan Receipt** button, which we will press to scan our receipt (Fig.10). Upon scanning the QR code, our receipt will be presented to us (Fig.11).

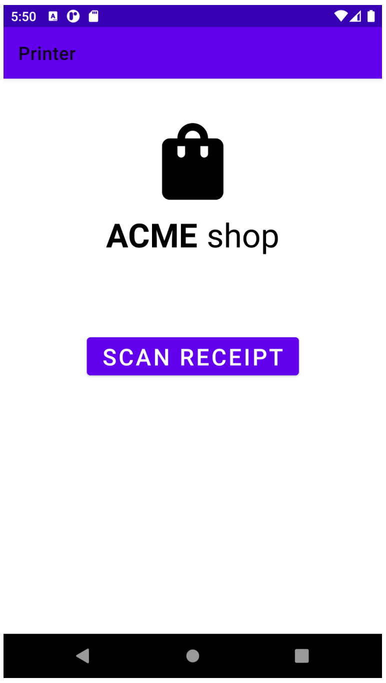
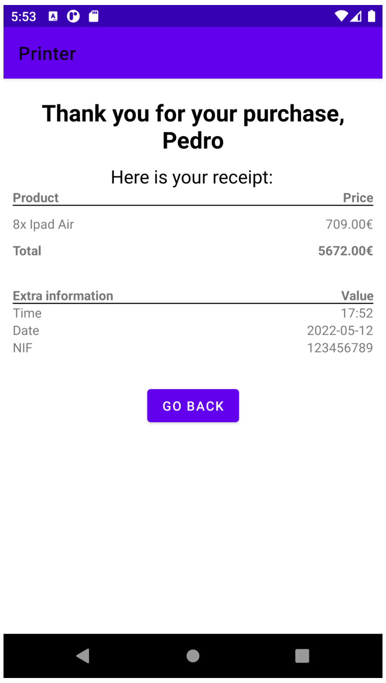


Fig.10: Printer app



Thank you for your purchase, Pedro

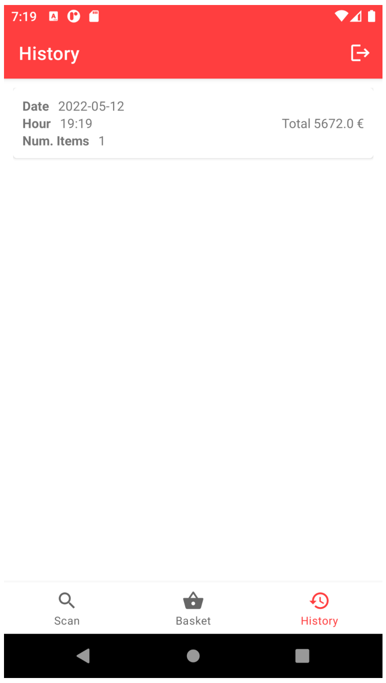
Here is your receipt:

Product	Price
8x Ipad Air	709.00€
Total	5672.00€

Extra information	Value
Time	17:52
Date	2022-05-12
NIF	123456789

Fig.11: Receipt

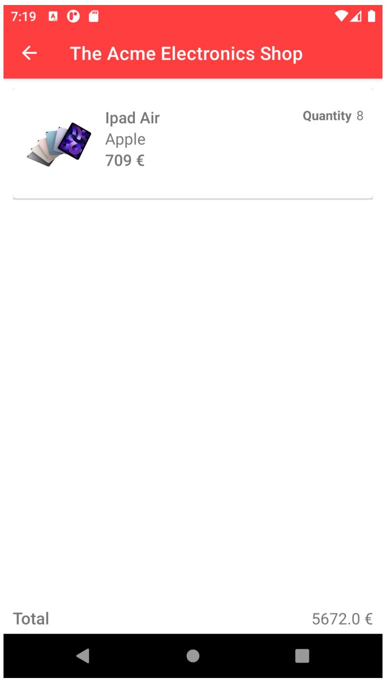
Finally, we can go back to the main app to check the history tab and see our past purchases where we can check the total, the date, the time and number of different items in the purchase (Fig.12). We can also click on each of these purchases to see in bigger detail what the purchase was composed of (Fig.13).


7:19

History

Date 2022-05-12
Hour 19:19
Num. Items 1
Total 5672.0 €

Fig.12: History page


7:19

The Acme Electronics Shop

Ipad Air
Apple
709 €
Quantity 8

Total 5672.0 €

Fig.13: Past purchase

6. Performed Tests

To guarantee that the system worked as expected, each feature was continuously tested, making sure that new increments did not break functionalities that were already implemented in previous iterations.

Furthermore, we tested the behavior of the Customer Application when providing invalid inputs, such as invalid credit card numbers, incorrect email format, and expired credit cards, among others.

The behavior of this application was also tested when the Server replied with error status codes, making sure that a *Toast* was presented to the User when errors occurred.

We tested printing the purchase receipt more than once, but as requested, it is not allowed, and after the first successful attempt it will not work and a *Toast* informing of this limitation is shown.

From these testing sessions, we were able to make some improvements to the developed features by, for example, setting that if the quantity of a product is below 1 it is removed from the basket, and imposing an upper limit of 99 units.

7. Project Setup

7.1. Required Software

- Android Studio
- Docker
- Docker Compose

7.2. Instructions

Assuming that all the required software is installed, the following steps must be followed in order to run the developed project:

1. To start the Server run the following command inside the **server directory**:

   > docker-compose up

2. Open the Customer App (**src directory**) and the Printer Terminal (**printer directory**), in different Android Studio windows;
3. Connect two Android devices to use as emulators;
4. List all the devices connected to your machine by using the following command:

   > adb devices

5. You must see your devices in the list. Now, as this system was developed for devices connected to the same network, we need to guarantee that the applications can access the Server. To achieve this, run the following command for each of the devices that were listed:

   > ***adb -s <device> reverse tcp:3000 tcp:3000***

6. Run both projects in Android Studio

8. Conclusion

In conclusion, we believe that we successfully achieved what was requested in this project, having developed a final product that fulfills every task it was meant to intuitively, while learning about all the involved mechanics and features we needed to complete the project.

We also want to emphasize that, in the printing process, at first, we intended to use both QR Code and NFC technologies, but we later realized that we only had one phone with the appropriate NFC and Android Beam capabilities, so we ended up only using the QR Code feature, due to that limitation.

Finally, some enhancements could be done to further improve our application, such as extra security measures on the server like upgrading to an HTTPS connection, a dark theme for the application as it is a common must nowadays, have more payment methods available, among others.

9. Bibliography and References

- **Project instructions**
  - https://moodle.up.pt/pluginfile.php/207717/mod_resource/content/3/cm_a1_2022.pdf

- **Google ZXing**
  - https://zxing.github.io/zxing/apidocs/

- **Kotlin Docs**
  - https://kotlinlang.org/docs/home.htm

- **EncryptedSharedPreferences**
  - https://developer.android.com/reference/androidx/security/crypto/EncryptedSharedPreferences

- **Android Documentation**
  - https://developer.android.com/

- **KeyStore**
  - https://developer.android.com/reference/java/security/KeyStore