ACME Supermarket

Integrated Masters in Informatic Engineering and Computation

Mobile Computation

Edgar Carneiro - up201503784
Sérgio Salgado - up201406136

Introduction

This report has as its main objective to explain in detail the implementation of the ACME Supermarket. The ACME Supermarket consists of two different mobile applications: one for the supermarket customer and the other for handling the validation and execution of the payments - the checkout terminal; and a server, capable of validating all customer actions and handling the ACME Supermarket's management logic.

The customer app is responsible for allowing the customer to create an account into the system, manage his shopping cart, consult past transactions and for payments. This payment is then sent to the checkout terminal, which confirms the user sent information with the server. The implementation of cryptography algorithms allows safe communication between the system's intervenients.

This project was created in the scope of the curricular unit Mobile Computation, which belongs to the Integrated Masters in Informatic Engineering and Computation.

Architecture

Description

This system architecture consists of the following:

- A customer mobile application, responsible for allowing the customer to do the following tasks: register a new account, login to an existent account that has been registered on the mobile being used, consult the account's past transactions, add and remove supermarket products from the current shopping order, confirm shopping orders, using loyalty vouchers and store and user credit discounts the customer may have;
- A checkout terminal, responsible for capturing the confirmed customer shopping order, send it to the server and receiving the server's response, thus allowing the opening of the supermarket's exit gate if the operation is successful;
- A remote server, responsible for the management of the ACME's Supermarket business logic, such as: registration and validation of customers, validation, analysis and persistence of customer payments, consultation of previously made transactions and access to available Vouchers and amount to discount. The server functions as ACME's back-office.

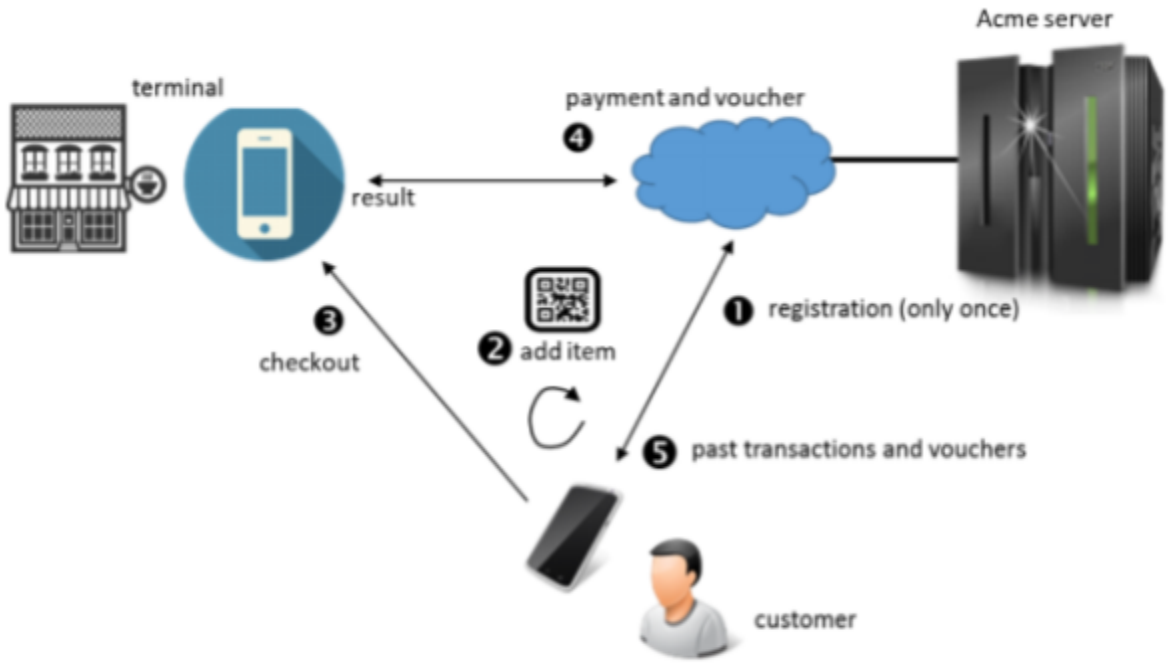


Figure 1: Diagram representing the used architecture in a simple way

Data Schema

For storing and handling the data related to the ACME's supermarket a database was built, managed by the remote server, using `sqlite3` and based on the following data schema:

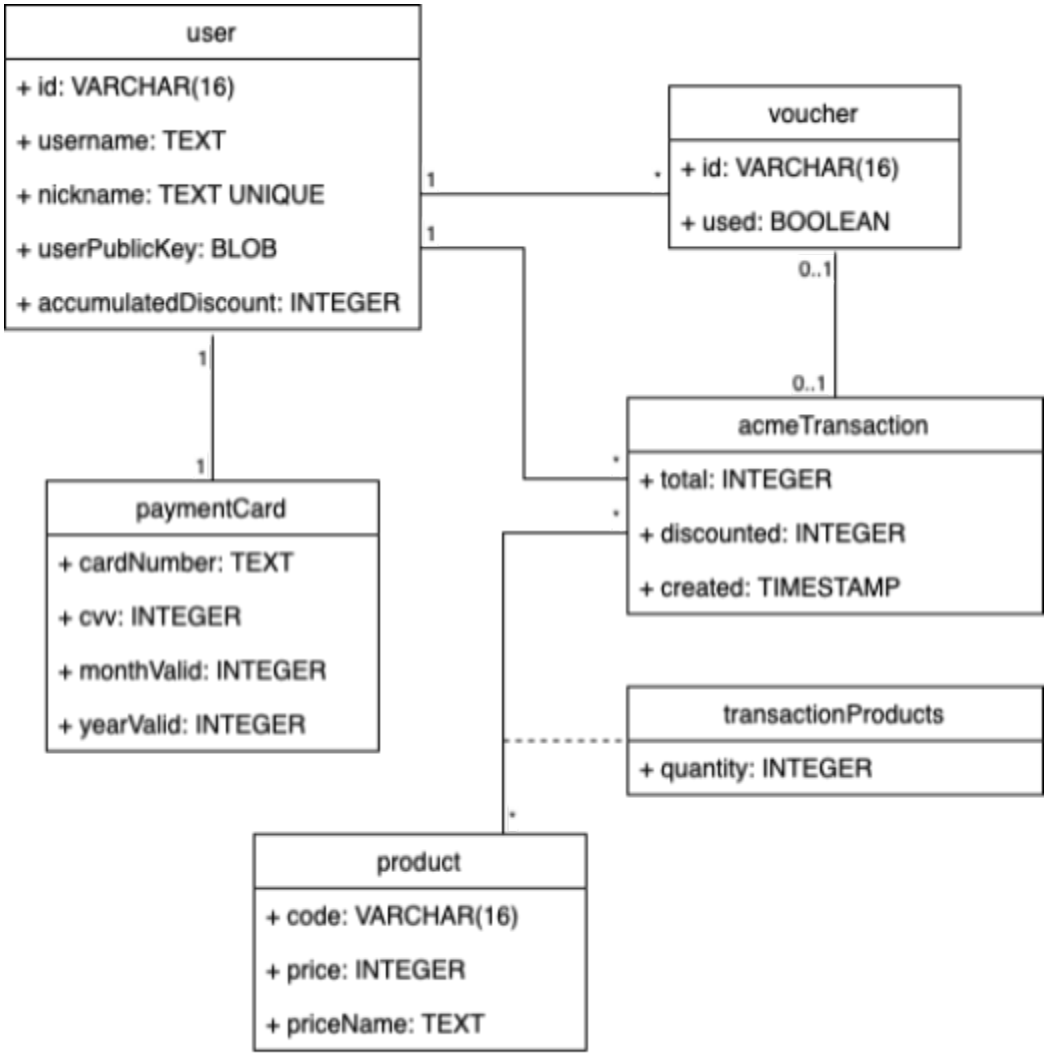


Figure 2: A conceptual diagram of the database schema

For accessing the `sqlite` code responsible for modelling the above schema, access the `/server/flaskr/db/schema.sql` file.

In a summarized manner, Figure 2 describes a User and his/her personal information, associated public key and current accumulated discount. Naturally, a User has an associated

payment card that he/she uses to perform payments. A User than has a set of associated vouchers that may be used or unused. Since the main focus of the application is to perform payments, the User will, as anticipated, also store the information relative to his/her past transactions. Each transaction has associated a set of products, in a certain quantity.

Client Application

The Customer Application is the application used to serve the end-user of the system: the Customer. It enables the User to register itself in the supermarket system, and from then on to make purchases in the ACME's supermarket, while applying vouchers and discounts, among other features. The application code organization is presented below, in Figure 3.

Figure 3: Customer App source code organization

A more in-depth analysis of each of the customer application's packages is presented below.

Activities

Package containing all the Activity classes used throughout the application. An activity represents a single screen with a user interface. An so, as expected, there will be an equal number of activities and of different screens the customer can visualize. In Figure 4 one can visualize the

Figure 4: Activity package's classes.

developed activities and their names, expected to explain the Activity purpose. Further ahead in this document, one can observe how the presented Activities interact with the remaining packages' contents.

Adapters

Package which stores auxiliary custom Adapter classes, whose purpose is to provide access to the data items contained within each list, storing information, binding logic to each item in the list and the view for each respective item. For example, a *ShoppingListAdapter* is responsible for binding the data contained inside the customer's shopping list and linking each item into a custom view.

As adapters allow for logic implementation inside each item, it is in the adapter classes that features such as allowing a swipe movement to delete a specific product entry from the shopping order are created.

Interfaces

Package containing custom interfaces created to standardize certain logic procedures between different classes, *e.g.* the template interface *ResponseCallable* is used inside each request containing a response, so that the arrival of the server's response can trigger a callable action, using the template type defined in the class implementing this interface.

Models

Package responsible for storing and handling the logic associated with the data used in the context of the application, using adequate representations. In Figure 5, one can observe the developed models and their names, expected to explain the model purpose.

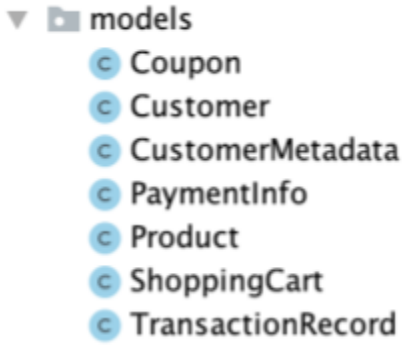


Figure 5: Model package's classes

Notice the natural mapping existent between the model package's class and the database schema's tables. The *Customer* and *CustomerMetadata* classes map to the *user* table, the *Coupon* class maps to the *voucher* table, the *Product* class maps to the *product* table and the *TransactionRecord* class maps to a set of entries of the *acmeTransaction* table. As it is obvious, the stored data fields will not be the exact same, since the customer app has only access to a representing fraction of the database's data.

Services

Package responsible for managing several classes that perform very specific actions. These actions vary from *HttpRequests* to the management of the *KeyStore*, among others. In Figure 6, one can visualize the developed services' package structure.

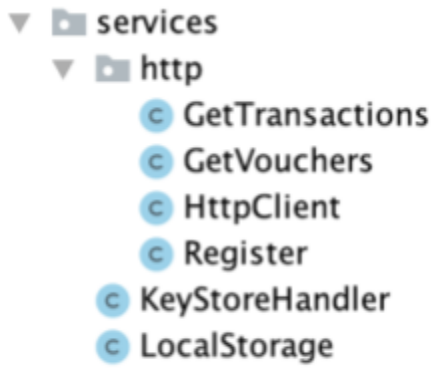


Figure 6: Services' package folder structure.

An in-depth analysis of these services is presented below:

- *http package*: module responsible for the definition and handling of all the requests made by the customer application and corresponding server responses. All classes, but the *HttpClient* which serves as the abstract base request class, represent requests to the remote server. Note that the appliance of the cryptographic responsibilities of the application are also ensured by this classes, even though the logic of such operations is not defined here. Hopefully, each class's name is explanation enough of the request the class is responsible for. It is also fulcral to mention the importance of the *ResponseCallable* interface (presented previously), that allows the return of customized responses, according to the necessities of each request.
- *KeyStoreHandler*: Class containing methods uniquely *static*, thus ensuring they can be used anywhere anytime in the code. It serves the purpose of handling the Java Keystore, thus complying capabilities such as the loading and storing of keys, as well as the serialization of keys from bytes.

- *LocalStorage*: Class containing methods uniquely *static*, thus ensuring they can be used anywhere anytime in the code. It has the responsibility of managing Android's SharedPreferences, thus allowing for the storage of information in-between sessions. It is using this mechanism that information such as the customer's password (among other relevant data) is stored for then to be tested in login, and loaded in case of matching of passwords.

Other files

There other files that are relevant in the context of the application, namely:

- *Constans*: File were a set of application context dependent variables are defined, *e.g.* the encryption and decryption algorithms used, the server endpoint, the key size, and others.
- *Utils*: Class that encapsulates methods uniquely *static*, thus ensuring they can be used anywhere anytime in the code, that are useful to all the remaining classes. This includes, for example, methods for encoding and decoding of Strings, concatenation of byte arrays, converting to and from *Base64, etc.*

Terminal Application

The terminal application follows an organization very similar to the one presented for the customer application. However, seeing that it has no necessity for all the packages, it only makes use of the activities, interfaces and services, more specifically the *http* package for performing the checkout request. In Figure 7, one can observe the developed folders and files structure on the terminal application.

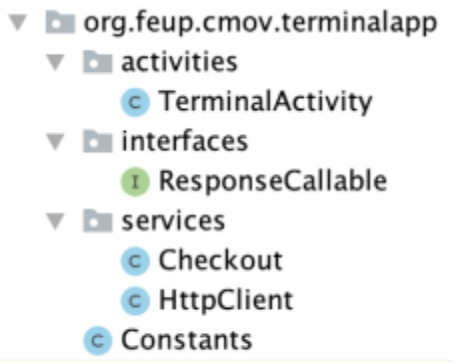


Figure 7: Terminal application's folders and files structure.

Server

For the development of the server, *Flask* was used. *Flask* is a lightweight Web Server Gateway Interface - WSGI - web application framework written in *python*. It is designed to make getting started quick and easy, with the ability to scale up to complex applications. The group chose to use it for the fast development associated with *python* and for its ease-of-use.

The ACME Supermarket remote server is responsible for serving the previously presented applications as a back-office, hence handling the supermarket side of business logic and assuring that transactions made are valid, secure and completed with integrality.

Packages Analysis

In Figure 8, one can observe the developed folders and files structure on the server *flask* application.

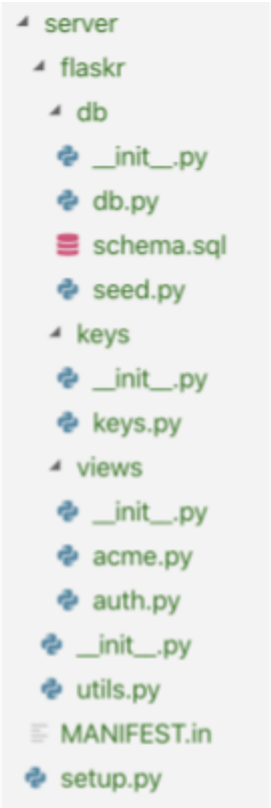


Figure 8: Flask server application folders and files structure

An in-depth analysis of the remote server's structure is presented in the following subsections.

Db

Package that contains all the files associated with database construction, management and interaction. The responsibilities are distributed as follows:

- *db.py*: File responsible for interacting with the database, thus retrieving the current database state, which allows other packages to get valuable information out of it. Additionally, it exposes two additional commands to the flask environment: `flask init-db`, for initializing the database; and `flask seed-db`, for seeding the database.
- *schema.sql*: File responsible for the definition of the database schema, previously presented in section *Architecture - Data Schema*.
- *seed.py*: File containing the code that will seed the database when the command `flask seed-db` is called.

Keys

Package responsible for the construction, management, serialization and deserialization of the keys used in asymmetric cryptography communication between the mobile applications and the remote server. Notice that for cryptography the external package *pyca/cryptography* is used.

In the *keys.py* file,one will find important methods for cryptography, such as: serialization and deserialization of a public key to/ from bytes containing the key in *DER* format, signing of a given message using a private key, verification of a given message using a public key, encryption of a message using a public key, generation of an asymmetric key pair, and serialization of an asymmetric key pair from a file containing the keys in *PEM* format. Additionally, a command is exposed to the flask environment: `flask gen-keys`, that allows to generate a new asymmetric key pair as the servers keys.

Views

Package responsible for defining the different blueprints, and respective views, that the server will contain. In a basica manner a view file represents handler of specific routes directed to the server. For more information, check the Blueprints and Views documentation.

The views that constitute the views package are:

- *auth*: View responsible for authentication. All authentication related routes - routes starting by `/auth/`, such as the register route, are handled by this view.
- *acme*: Main view of the server. Works as the default route, so any request not matching other views will be redirected to this view. Encompasses all business

logic related routes, namely the get transactions route, the checkout route, and others.

Other files

It is also important to highlight other files not inserted in any server package. Those of relevance are:

- *__init__.py*: The server folder's initialization file represents configuration file of the application. All server's blueprints are gathered and setted up here, as well as the configurations of both the database and the server asymmetric keys to the current application context.
- *utils.py*: Utility methods used by the entirety of the other packages. For instance methods for string to byte encoding and decoding, conversion from and to *Base64,* generic error handling and generation of UUID are defined in this file.

Routes

The routes provided by the remote server are present, in a summarized manner, in the following table:

URL	*Http* Method	Route details
`/auth/register`	POST	Route for registering a customer in the ACME Supermarket. Provided the necessary information, the supermarket replies with customer's UUID and the supermarket public key.
`/get-products`	GET	Route for getting the ACME available products. As a response to this route, an array of products each returned. Notice, however, that each one of them is signed with ACME's private key, thus ensuring their authenticity.
`/get-transactions`	POST	Route for getting a customer's past transactions. It receives a request with a body with the customer UUID and that same UUID signed by the customer's private key. It responds with an array of past transactions, where each transaction is encrypted with the customer public key and the totality of the message is signed with the server's private key.
`/get-vouchers`	POST	Route for getting a customer's available vouchers. It

		receives a request with a body with the customer UUID and that same UUID signed by the customer's private key. It responds with an array of available vouchers as well as available discount, where besides the available discount, each voucher is encrypted with the customer public key and the totality of the message is signed with the server's private key.
`/checkout`	POST	Route for sending the confirmation of a customer's purchase to the server. The customer sends its purchase information - products bought, voucher used, discount applied - to the server, with all the data signed by the user private key. The server responds with the purchase value in case of success and an error in case something went wrong.

Functionalities

In this section, an overview regarding the requested and extra implemented features is presented. Notice that the extra features are marked as such and that some extra steps were taken in order to improve the already requested features (*e.g.* quantity buttons in *Add Product* feature).

Register

A form is presented for the user to fill with his basic information, such as name and password, and the customer's debit/ credit card information for payment purposes. The information provided in this form is then sent to the server via a POST request and stored in Adnroid's local storage for login purposes. This form checks the length of fields such as the card number and CVV and informs the user with an error message in case any field does not meet the requirements.

If the request to the server is successful, the server sends the supermarket's public key for the app to store it locally and use it to communicate safely with the server. On the opposite side, if the request to the server fails, an error message is shown on the customer's screen detailing information about where in this process the error occurred.

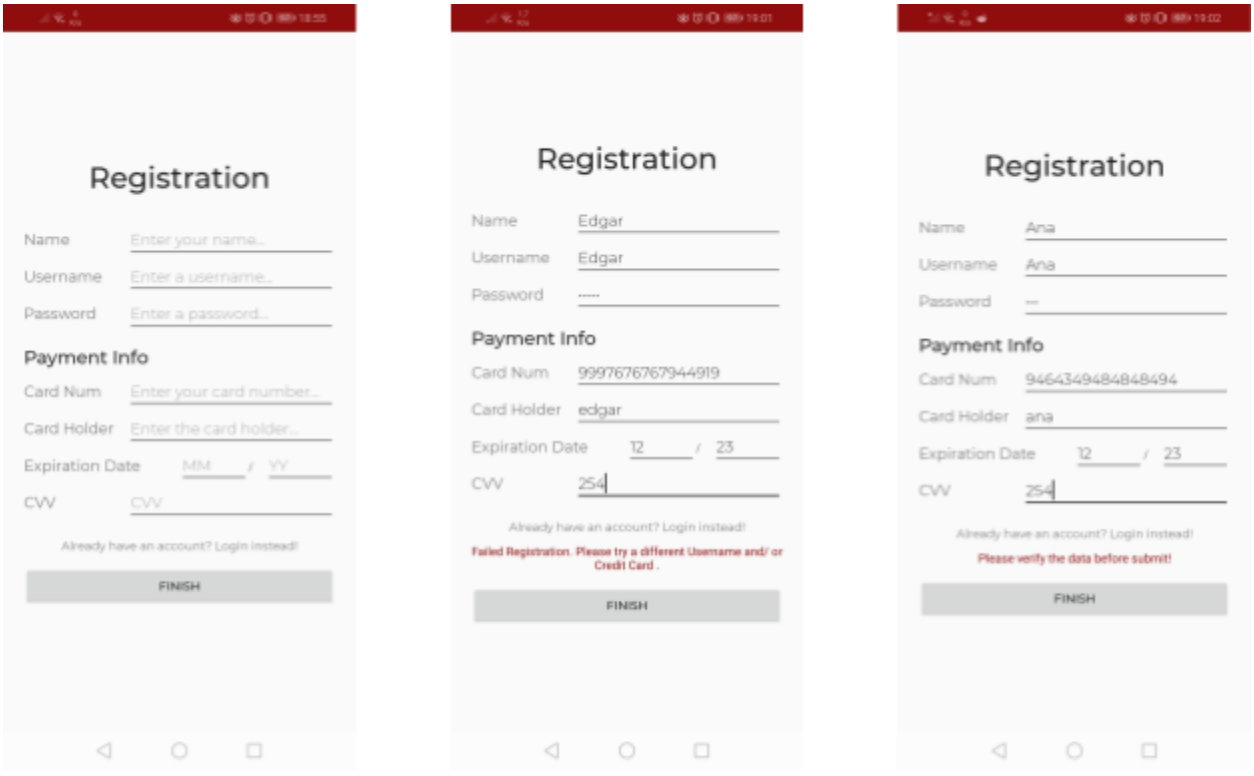


Figure 9: Registration process. From left to right: A clear registration form, a registration form using information of a previously registered user, a form with invalid fields.

Login

A form is presented to the user which asks user input for username and password, which is compared to stored credentials in local memory. If the credentials match with some entry stored in the device, a new Customer object is then created, granting the user permission to interact with the core of the app.

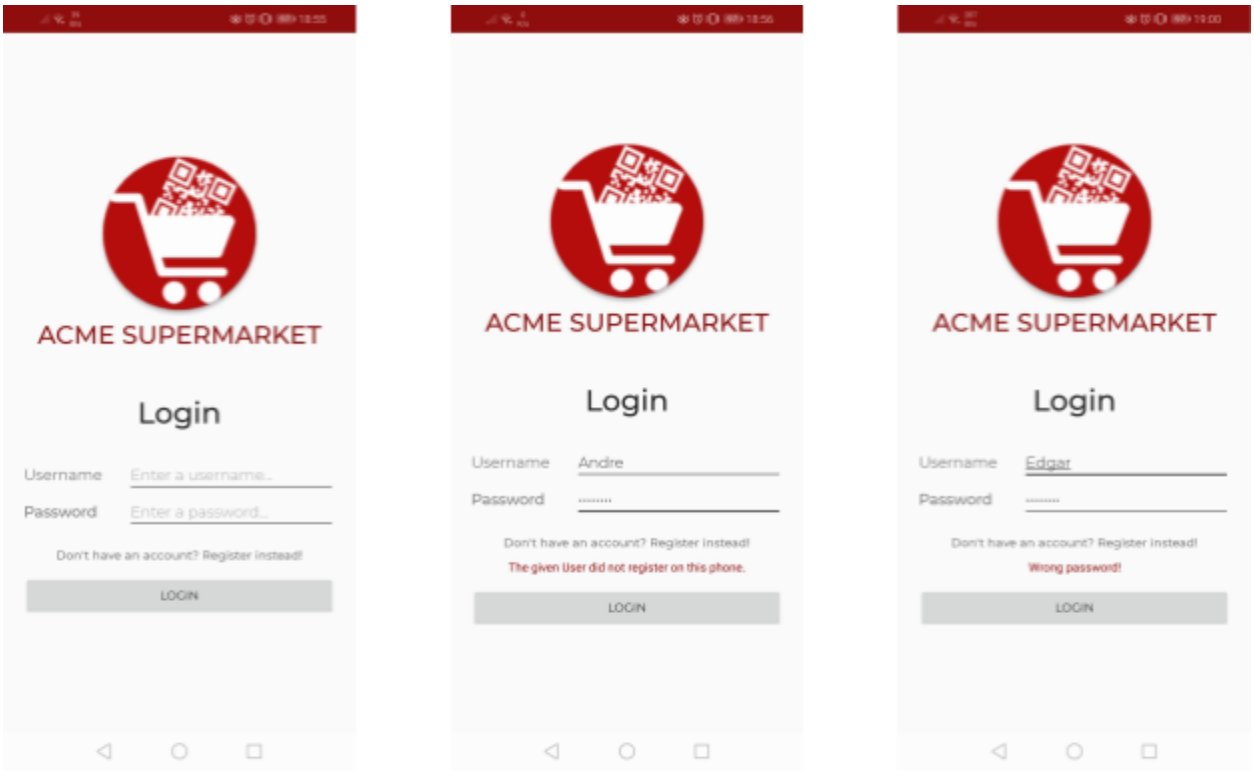


Figure 10: The login screen. From left to right: A clear login form, a login form where the user logging in did not register from the current device (failing the login process), a login form where the user inputs a wrong password.

Shopping Cart

The main functionality of the customer application, it allows the user to manage his current shopping order, either by adding new products or removing existent ones in the cart. It also allows the user to check the full value of the products inside the shopping cart and, in case this value surpasses 100€, inform the user that if checkout is made with the current cart, a loyalty voucher will be awarded to his account, which provides the option to discount a future purpose by 15% of the value and store that amount in store credit, which can further discount a future purchase.

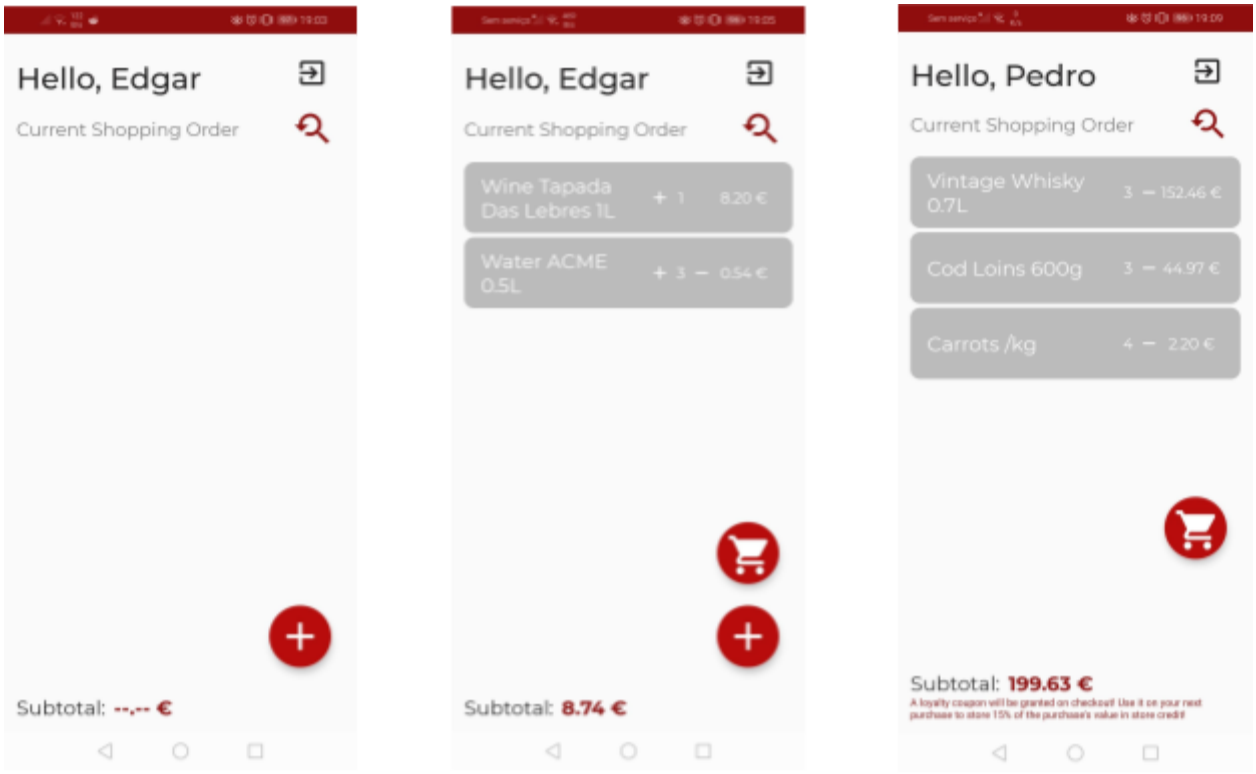


Figure 11: The shopping cart. From left to right: A clear shopping cart with no products, a shopping cart with two products inside, a full shopping cart with a subtotal high enough to make the customer earn a loyalty voucher.

Add Product

With the use of a floating action button with a '+' icon in it, the user can scan a QR code by using the device's camera. After the QR code signed by the supermarket is scanned, the user's device verifies the code authenticity, making sure the real supermarket created that code.

A new product entry is then added to the customer's shopping cart, updating the current cart value and displaying information to the user about the product.

Remove Product & Undo *(Extra Features)*

To remove an existent product from your shopping cart, a swipe movement on any item removes it from your cart, updating the current shopping order value.

After removing a product from your cart, a snackbar will pop up, allowing the user to undo the product removal.

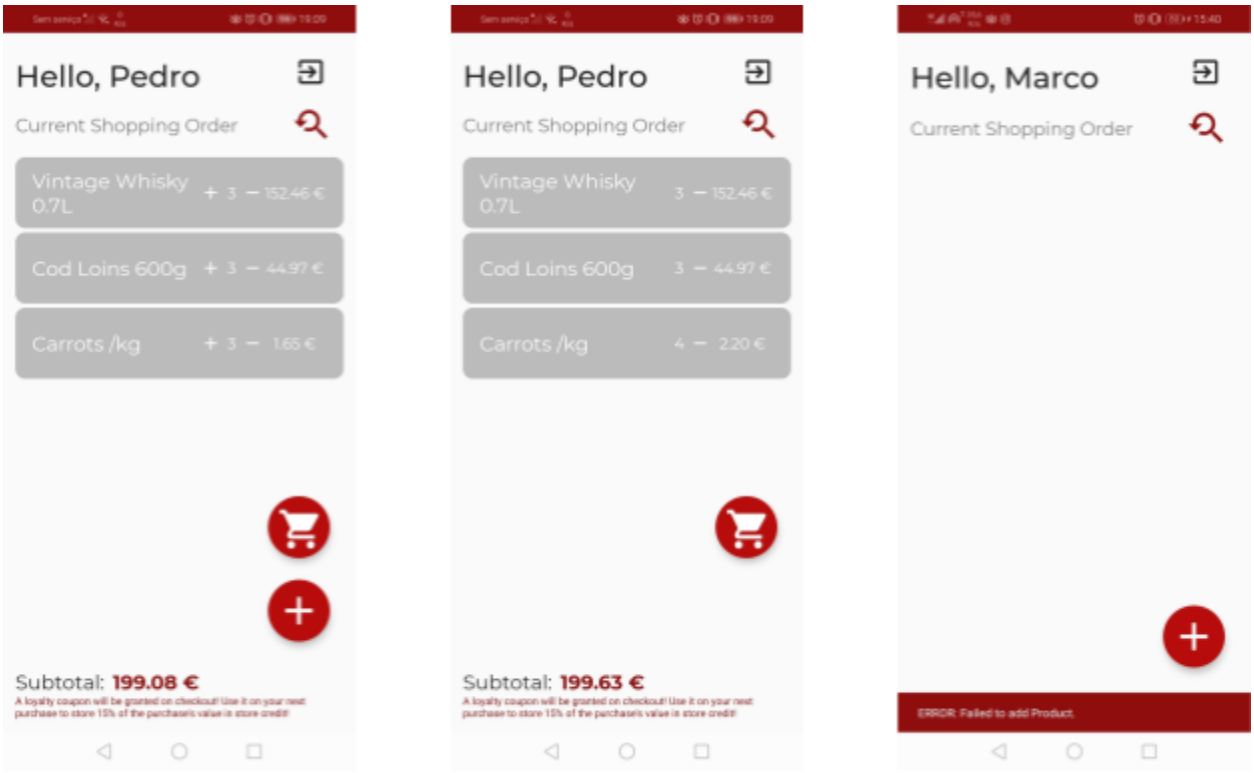


Figure 12: Adding a product. From left to right: A nearly full shopping cart, a full shopping cart, a screen which informs the user an error occurred when adding a new product to the cart.

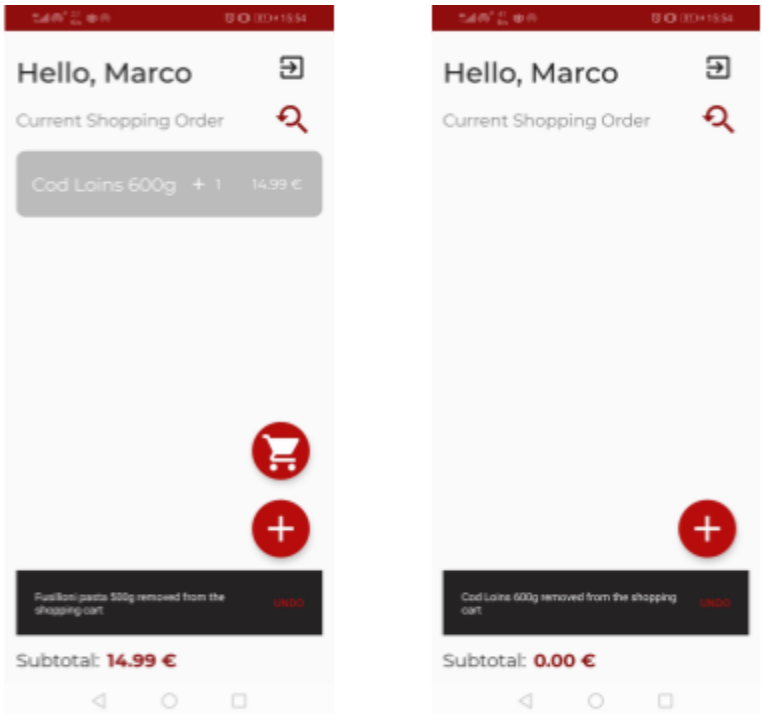


Figure 13: Removal of a product. On the left side, the 'Fusilloni Pasta' was removed from the shopping cart, triggering a pop-up for undoing the removal. On the right side, a clear cart after the removal of the last product.

Past Transactions Review

The user is able to consult the past transactions done by its account, with the pop up showing information about the payment information used, date of the purchase, the value of that purchase, if a voucher was used and if store credit was used to discount the purchase.

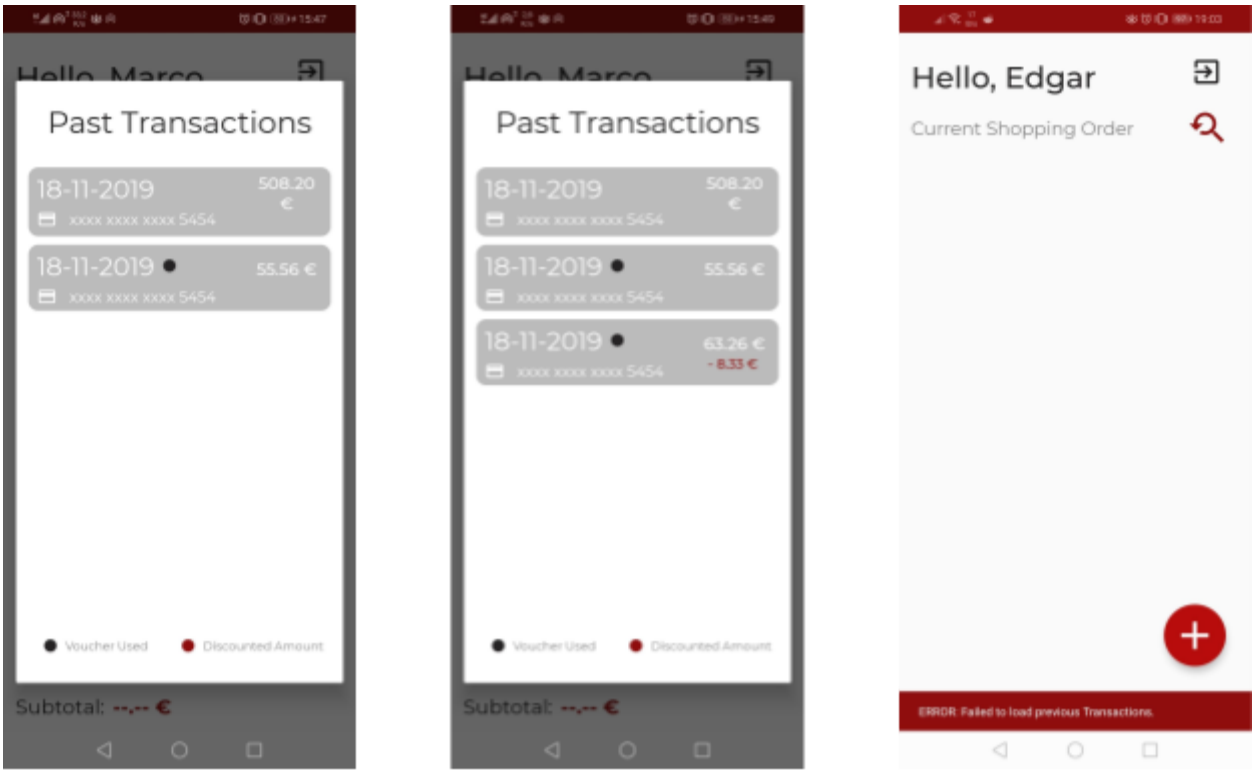


Figure 14: The past transactions feature. From left to right: A review of two past transactions (the first one with no voucher or discount applied and the second with a voucher used), a new past transaction with both voucher and store discount used, an error message when the past transaction request fails.

Purchase Confirmation

After the user submits the shopping order, a purchase confirmation activity is shown to the user, so the shopping order data can be confirmed and submitted to the supermarket terminal.

Usage of Vouchers

In this screen, the user can select one of his available vouchers earned by spending multiples of 100€ on the shop. This voucher provides the user with a 15% discount on the current purchase, storing the resultant value in store credit which can be used to discount a later purchase.

Store Credit Discount

The user is able to check a box whether he wants the stored store credit to apply a discount to the current purchase.

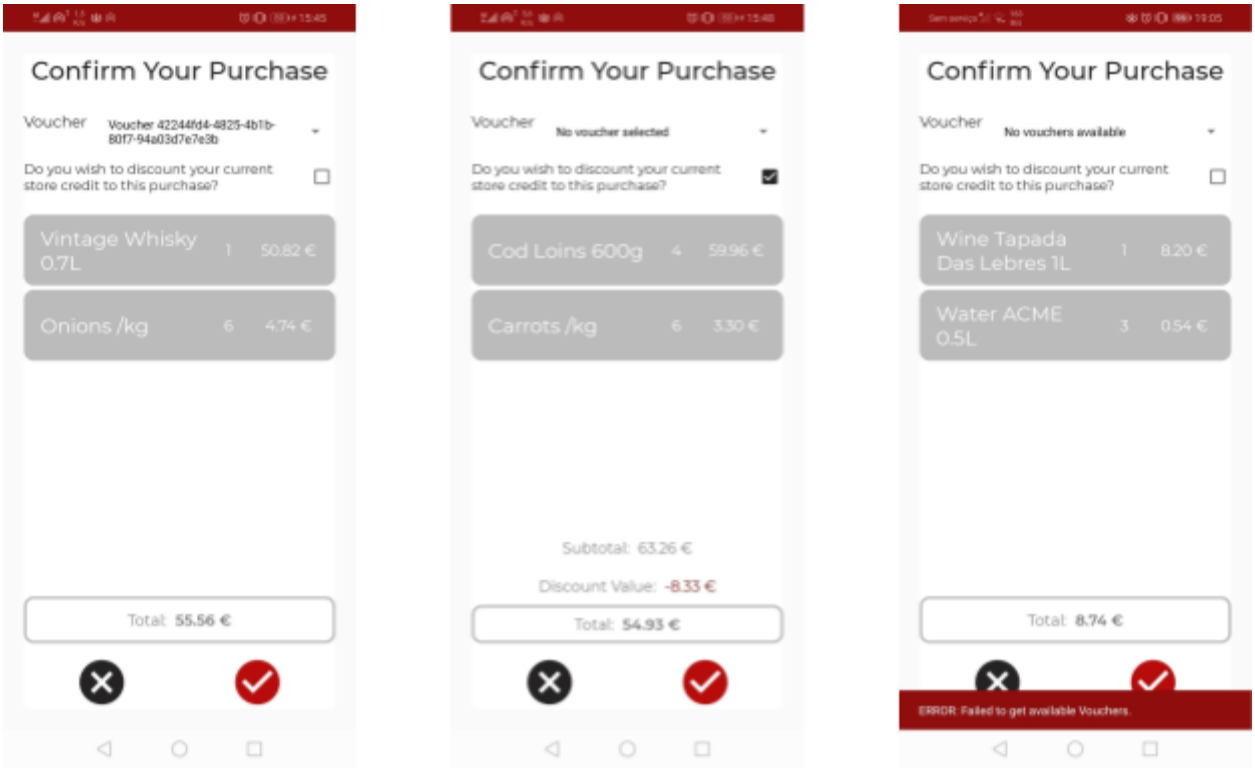


Figure 15: Purchase confirmation screen, where the user selects vouchers and toggles a discount option. From left to right: A purchase with a voucher selected, a purchase with the store credit discount toggled, a purchase confirmation screen where communication to the server fails when requesting the user's vouchers.

Checkout

After the user confirms the information about voucher, discount usage and the shopping order done, the user is taken to a new screen where a QR code referring to the recently done purchase, all relevant user information and options selected needed to confirm the purchase. This QR code must then be presented to the second application in this project, the terminal application. After the purchase is scanned by the terminal, communication is made between it and the central server, where the entire procedure is validated or not by the server.

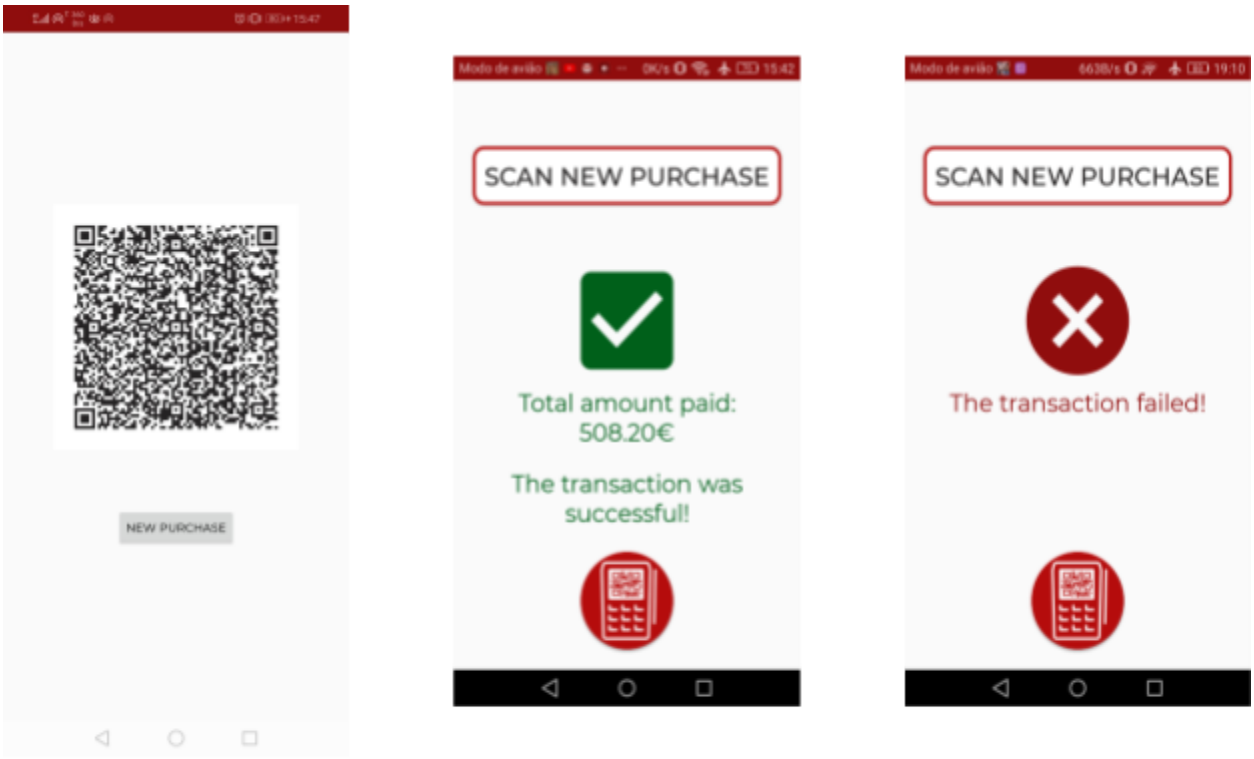


Figure 16: The checkout process. From left to right: Generated QR code on the customer's device which will be scanned by a terminal, terminal screen on a successful purchase, terminal screen when a failure occurs.

Implementation Details

In this section, the user can get knowledge regarding Implementation details, both in mobile applications and the remote server. Aspects such as the message trading protocol, cryptography practices, creation and interpretation of requests and respectives results, among others, are highlighted.

Message Modelling

For the communication between the remote server and the mobile applications a protocol had to be established.

To facilitate the process and seeing that in the remote server Flask was being used and that Flask has support for handling json, *Gson* was used in mobile *JAVA* applications. By using Gson, the mobile applications become able to interpret and handle *json* strings coming as a response to the made requests.

It is also important to preemptively state the need to convert from and to *Base64*, since the inner representation arrays made by *JAVA* and *Python* is different: *JAVA* uses signed chars while *Python* uses unsigned chars in the abstraction it presents the User to handle bytes.

Having these considerations in mind,the protocol used is defined below.

- Route: `/auth/register`
    - For sending the data to the server and making it easier to interpret on server side, the register route was developed using *json* for both the request and the response. An example of a possible request and respective response is presented below:
        - Request:


```
{"metadata":{"name":"Pedro","password":"edgar","publicKey":"MFwwDQYJKoZIhvcNAQEBBQADSwAwSAJBANZNb/F40h6Jp6sJ/WB92/oNN981d6oN9idQX1YB1hhf\nMV4z0GlJ9zCWSVEZ99S9mMmerxmWur7BI7G6r3cpYu0CAwEAAQ\u003d\u003d\n","username":"Pedro"},"paymentInfo":{"CVV":123,"cardNumber":"9238948293489293","cardValidity":{"month":1,"year":23}}}
```

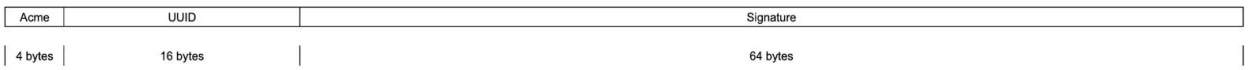

        - Response:


```
{"public_key":"MFwwDQYJKoZIhvcNAQEBBQADSwAwSAJBANj1kmumzLb3q5VzHVpfi2/d7MvZGVThpAult04LidIjLKQo/PQX3SiT7QXwH3fAKEqinOJdxSS0ZATe/FXygMsCAwEAAQ==",
"uuid":"aa64bd8c-4aee-4a89-a990-cd42a8da057b"}
```


        - Notice that the public key is in *Base64* and encoded in format ISO-8859-1.

- Route: `/get-transactions`

- For the application request, byte handling and direct byte writing to a buffer is used. For the server response, a mixture between byte writing and json loading is used.
  - Request format:



  - Request example:

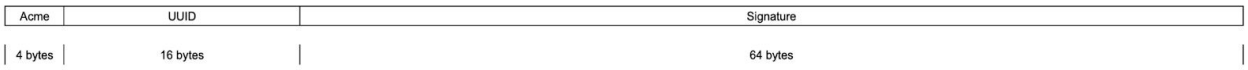
```
b'Acme\xaad\xbd\x8cJ\xeeJ\x89\xa9\x90\xcdB\xa8\xda\x05{\xa0\x9b?\x1a\xdd\xa
0\xb0\xc59\xff[\xdeF\x05\x946\xa2\x19S\xcb\x93\x9c}\xc5Lk\xa6\xac\xe8gC:\x0
6n7\xba\x1ej\x05\xd3\xb4W\\\xf8\xfb\xcdz\xdf\xcd\xff\xdb#M\x07P\x8d\x97\x91
;\xb9\x9cqAa'
```

  - Response format: An array of past transactions, where each transaction is encrypted with the customer public key and the totality of the message is signed with the server's private key.
  - Response example:

```
b'{"transactions":
["Hpv2m+gpUu9B7rfOMVEX0YocpwnXJRpAZY6BV4W+pHPTJ07BpaISnIacffRi1vjmxMlvyKeL3
J0U2I6NtsCPGg==",
"B0Xj/M8nk9cgVd6XNC3ysmyiI6BQA70WFktU1o+4MPiQDkeNhVS1G3pd3zpX1Len7KvTZ29Nqg
OOqpmjBUSbFA==",
"cg+Cc/YDLT/Q7/EI48fuMB3rOliyTczBFp/irjXHmhNemqphnyfCZ/h+RDiOz8gt3RR/rKKxEj
0wBPKwVA3WxA=="]}R\x86~\x13\x83yD\xde\xc9\xb3&,\xc51\xb5x\x8e\xdf\xae\'\xc5
;B&1:\xa7A-`\xde\x19\xa23\xee\x87\xb7\x05\\\xed\x8c\x17
@\xa4op\x17a\x9d\xd1\xa9Z{)k\xfe3\x84ZN\x9aA@'
eyJ0cmFuc2FjdGlvbnMiOiBbIkhwdjJtK2dwVXU5QjdyZk9NVkVYMFlvY3B3blhKUnBBWlk2QlY
0VytwSFBUSjA3QnBhSVNuSWFjZmZSaTF2am14TWx2eUtlTDNKMFUySTZOdHNDUEdnPT0iLCAiQj
BYai9NOG5rOWNnVmQ2WE5DM3lzbXlpSTZCUUE3MFdGa3RVMW8rNE1QaVFEa2VOaFZTMUczcGQze
nBYMUxlbjdLdlRaMjlOcWdPT3FwbWpCVVNiRkE9PSIsICJjZytDYy9ZRExUL1E3L0VJNDhmdU1C
M3JPbGl5VGN6QkZwL2lyalhIbWhOZW1xcGhueWZDWi9oK1JEaU96OGd0M1JSL3JLS3hFajB3QlB
Ld1ZBM1d4QT09Il19UoZ+E4N5RN7JsyYsxTG1eI7frifFO0ImMTqnQS1g3hmiM+6HtwVc7YwXIE
Ckb3AXYZ3RqVp7KWv+M4RaTppBQA==
```

  - Response example: Notice that in the given example the discount and everys single voucher is encoded and then passed to *Base64*. It is also possible to discern the *json* structure.

- Route: `/get-vouchers`
    - For the application request, byte handling and direct byte writing to a buffer is used. For the server response, a mixture between byte writing and json loading is used.
        - Request format:



        - Request example:

```
b'Acme\xaad\xbd\x8cJ\xeeJ\x89\xa9\x90\xcdB\xa8\xda\x05{\xa0\x9b?\x1a\xdd\xa
0\xb0\xc59\xff[\xdeF\x05\x946\xa2\x19S\xcb\x93\x9c}\xc5Lk\xa6\xac\xe8gC:\x0
6n7\xba\x1ej\x05\xd3\xb4W\\\xf8\xfb\xcdz\xdf\xcd\xff\xdb#M\x07P\x8d\x97\x91
;\xb9\x9cqAa'
```

        - Response format: An array of available vouchers as well as available discount, where besides the available discount, each voucher is encrypted with the customer public key and the totality of the message is signed with the server's private key.
        - Response example:

```
b'{"discount":
"lgsTCONFGPG1PHIc4Mo8wYkI+WXSE8bdUrvqxnZR5nILR4wcwA4wxZmo/WW7SF6yOUr1Wnw/H3
Bcv1dGYYUvHg==", "vouchers":
["KUetiXUI7u0sPDbCQx8D48iy0euVPiuiv5A5ywSCGrE8jDYn84+x612OLIX7x/uSe7Q4zdXNv
YCJq7IF1Lrz0g==",
"HB0zWO2fdC0hUnBbkEnpklob/2HkJyTK2GRPQnTtQWPEGQoSqXDego4IJqsg5yclzLc1ytmPu3
4YP2e9W1BkAw==",
"qunTwvwDluanATLG68TZ/cgun9RKHisWSjnWmnlCzlb43+yRQFSrTwWzcovUaq3Ro0X3db6Ita
WvNEsgVYbPdw==",
"kVYp0OrCWMAuIthLJxKJ+T0CrHRd/Cw5P+fhV3N1bUJAdOkQrJa14CzaDohDbJwYI5tOcTCZaW
Z2marU35w0rQ==",
"fTzWWxAqrdydolo+hpY/LTiivYvv5YTpm0s7LHmO1elNThUr7EMSSZFhZnatU4snjos00jxfFu
Vy9FFa1BABqg=="]}\xa4cAoN?\x07Q\xa8-\xbe\xc5\xe7p\xd1`\xecw*\x06\xfd\x19\x0
6w\xa8]\xf4t\x1a\x06\\\xe2,\xd3\xba&\xd7O~\x1d\x8bz\x8e\xcaA\xf5\x14:\xa9\x
e7\xd2\xef]\r\xd2QDE\xfe\xb7h\x0f\x05w'
```

        - Notice that in the given example the discount and everys single voucher is encoded and then passed to *Base64*. It is also possible to discern the *json* structure.

- Route: `/checkout`

- For the application request, byte handling and direct byte writing to a buffer is used. For the server response, a mixture between byte writing and json loading is used.
    - Request format:

    - Request example:

```
b'Acme\xaad\xbd\x8cJ\xeeJ\x89\xa9\x90\xcdB\xa8\xda\x05{\xa0\x9b?\x1a\xdd\xa
0\xb0\xc59\xff[\xdeF\x05\x946\xa2\x19S\xcb\x93\x9c}\xc5Lk\xa6\xac\xe8gC:\x0
6n7\xba\x1ej\x05\xd3\xb4W\\\xf8\xfb\xcdz\xdf\xcd\xff\xdb#M\x07P\x8d\x97\x91
;\xb9\x9cqAa'
```

    - Response format: A string containing the operation total price, in cents.
    - Response example:

```
50820
```

Cryptography

Cryptography proved to be one of the main focuses of the project, with the communication between all applications having to be secured. To do so, in the mobile applications, JAVA's security libraries were used with a main focus on Android's KeyStore. In the server side, the main security handler is *pyca/cryptography*, as previously stated.

Below, several import code sections from all applications are presented, in an attempt to elude regarding the security measures taken.

Mobile Applications

- Generation of the checkout message

```
        ArrayList<Product> products =
this.currentCustomer.getShoppingCart().toArrayList();
         ByteBuffer buffer = ByteBuffer.allocate(
```

```
                CHECKOUT_MSG_BASE_SIZE
                        + products.size() * Product.CHECKOUT_MSG_SIZE
                        + (voucherID != null? UUID_SIZE : 0)
        );

        // Loading Acme Tag and UUID
        UUID uuid =
UUID.fromString(LocalStorage.getCurrentUuid(this.getApplicationContext()));
        buffer.putInt(Constants.ACME_TAG_ID);
        buffer.putLong(uuid.getMostSignificantBits());
        buffer.putLong(uuid.getLeastSignificantBits());

        // Loading Products
        for (Product prod: products)
            buffer.put(prod.getProductAsBytes());

        // Loading Voucher choice and discount choice
        if (voucherID != null) {
            System.out.println(voucherID);
            UUID voucherUID = UUID.fromString(voucherID);
            buffer.putLong(voucherUID.getMostSignificantBits());
            buffer.putLong(voucherUID.getLeastSignificantBits());
            buffer.put((byte) 1);
        } else {
            buffer.put((byte) 0);
        }
        buffer.put((byte) (discount? 1: 0));

        // Signing everything
        byte[] msg = toBase64(buffer.array());
        byte[] content = concaByteArrays(msg,
toBase64(this.currentCustomer.signMsg(msg)));

        // As QRCode message
        String string = new String(content, StandardCharsets.ISO_8859_1);
```

- Validating and decrypting server response containing user vouchers and available discount

```
          // Get response
            int responseCode = urlConnection.getResponseCode();
            if (responseCode == 200) {
                byte[] encodedContent = customer.getSignedServerMsgContent(
```

```
readStream(urlConnection.getInputStream()).getBytes(StandardCharsets.ISO_8859
_1),
                        this.context
                );

                // Signature Verified
                if (encodedContent != null) {
                    EncodedResponse encodedResponse = new Gson().fromJson(
                            Utils.encode(Utils.fromBase64(encodedContent)),
                            EncodedResponse.class
                    );
                    // Handling encoded discount
                    byte[] decryptedDiscount = customer.decryptMsg(
                            Utils.fromBase64(Utils.decode(
                                    encodedResponse.getDiscount()))
                    );

                    if (decryptedDiscount != null) {
                                                                response   =   new
GetVouchersResponse(parseInt(Utils.encode(decryptedDiscount)));

                        // Handling encoded vouchers
                        for (String voucher: encodedResponse.getVouchers()) {
                            byte [] decryptedVoucher = customer.decryptMsg(
                                    Utils.fromBase64(Utils.decode(voucher))
                            );

                            if (decryptedVoucher != null)

response.addVouchers(Utils.encode(decryptedVoucher));
                        }
                    }

                }
            }
```

- Client signing, validating and decrypting

```
   public byte[] signMsg(byte[] msg) {
        PrivateKey pk =
KeyStoreHandler.getUserPrivateKey(this.metadata.getUsername());
        if (pk == null) {
```

```
            System.err.println("User does not have a private key!");
            return null;
        }

        try {
            Signature sig = Signature.getInstance("SHA256WithRSA");
            sig.initSign(pk);
            sig.update(msg);
            return sig.sign();

        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }

    public byte[] getSignedServerMsgContent(byte[] message, Context
context) {
        boolean verified = false;
        byte[] content = null;

        try {
            byte[] signature = Utils.fromBase64(Arrays.copyOfRange(
                    message, message.length - SIGNATURE_BASE64_SIZE,
message.length
            ));
            content = Arrays.copyOfRange(
                    message, 0, message.length - SIGNATURE_BASE64_SIZE
            );

            Signature sign = Signature.getInstance("SHA256withRSA");
            sign.initVerify(
                    KeyStoreHandler.getKeyFromBytes(
                            LocalStorage.getAcmePublicKey(context)
                    ));
            sign.update(content);
            verified = sign.verify(signature);
        }
        catch (Exception e) {
            e.printStackTrace();
        }
```

```
        return verified? content: null;
    }

    public byte[] decryptMsg(byte[] message) {
        try {
            Cipher cipher = Cipher.getInstance(Constants.DECRYPT_ALGO);
            cipher.init(Cipher.DECRYPT_MODE,
KeyStoreHandler.getUserPrivateKey(this.getUsername()));
            return cipher.doFinal(message);
        }
        catch (Exception e) {
            System.err.println("Failed to decrypt message");
            return null;
        }
    }
```

Server

- Encrypting and signing of transactions

```
   # Encrypting every single transaction
    content = b64_encode(encode(
        json.dumps({
            'transactions': [
                decode(b64_encode(encrypt(
                    encode(json.dumps({
                        'd': t['created'].strftime("%d-%m-%Y"),
                        't': t['total'],
                        'di': t['discounted'],
                        'v': t['voucherID'] is not None,
                    })),
                    user_key_from_bytes(user['userPublicKey'])
                )))
                for t in transactions
            ]
        })
    ))

    # Signing content
    final_content = decode(
        content + b64_encode(sign(
```

```
            current_app.config['PRIVATE_KEY'],
            content
        ))
    )
```

- Encrypting and signing of vouchers and discount

```
    # Encrypting every single voucher and the discount
    content = b64_encode(encode(
        json.dumps({
            'vouchers': [
                decode(b64_encode(encrypt(
                    encode(str(row['id'])),
                    user_key_from_bytes(user['userPublicKey'])
                )))
                for row in vouchers
            ],
            'discount': decode(b64_encode(encrypt(
                encode(str(user['accumulatedDiscount'])),
                user_key_from_bytes(user['userPublicKey'])
            )))
        })
    ))

    # Signing content
    final_content = decode(
        content + b64_encode(sign(
            current_app.config['PRIVATE_KEY'],
            content
        ))
    )
```

- Verifying user that checked out

```
    # Checking if User exists
    uuid = str(UUID_from_bytes(decoded_content[:UUID_SIZE]))
    decoded_content = decoded_content[UUID_SIZE:]
    user = db.execute(
        'SELECT userPublicKey, accumulatedDiscount FROM user WHERE id = ?',
        (uuid, )
    ).fetchone()
    if user is None:
```

```
        abort(401)

    # Verifying User through signature
    if not verify(user_key_from_bytes(user['userPublicKey']),
                  signature,
                  content):
        abort(401)
```

- Code for encryption, validation and signing

```
def sign(private_key, message):
    """Sign the given message using the given private key"""
    return private_key.sign(
        message,
        padding=padding.PKCS1v15(),
        algorithm=hashes.SHA256()
    )


def verify(public_key, signature, data):
    """Verify the given signature using the given public key.
    Returns true if verified, False otherwise"""
    is_signature_correct = True

    try:
        public_key.verify(
            signature=signature,
            data=data,
            padding=padding.PKCS1v15(),
            algorithm=hashes.SHA256()
        )
    except InvalidSignature:
        is_signature_correct = False

    return is_signature_correct


def encrypt(content, public_key):
    return public_key.encrypt(
        content,
        padding.PKCS1v15()
    )
```

Performed Tests

The following section encompasses a detailed description of all the tests performed, namely the scenario description of the test and the respective scenario result. Any User using the application shall be able to replicate any of the tests described below.

Register

Test Number	Scenario Description	Scenario Result
1	The form is submitted without any field being filled.	An error message appears reminding the user to check the data before submitting the form.
2	The form is submitted with a field containing invalid data, such as credit card number or password.	The same error message appears for the user to verify the data inserted before submission.
3	The form is submitted with all fields filled with valid information, but a user with that information already exists	An error message appears for the user to try again with a different username.
4	The form is submitted with all fields filled with unique valid information.	The freshly created user is taken to the application's main menu, where he/she can start shopping.

Login

Test Number	Scenario Description	Scenario Result
1	The form is submitted with information about a user which does not exist locally in the system.	An error message appears informing the user trying to login did not register using this device.
2	The form is submitted with information about a user which was registered in the system using this current device.	User logs into the service, taken directly to the main menu.

Shopping Cart

Test Number	Scenario Description	Scenario Result
1	The user presses the log out button situated in the top right corner of the screen.	User is taken back to the login screen to input account information.
2	The user presses the past transactions button situated slightly below the log out button.	A pop-up dialog appears on the screen, showing information about past transactions successfully done with that account.
3	The user presses the floating action button localized near the bottom right corner of the screen, marked with a plus sign.	A scanner application opens, allowing the user to scan a product QR code.
4	The user presses the plus button localized inside the scanned product card.	The quantity of the product in the card is increased by +1. If the sum of the products in the cart reaches 10, all plus buttons are hidden from the user. All snackbar pop-ups are dismissed.
5	The user presses the minus button localized inside the scanned product card.	The quantity of the product in the card is decreased by -1. If pressing the button leaves the remaining quantity of that product in 1, the minus button is hidden. All snackbar pop-ups are dismissed.
6	The user performs a swipe action horizontally in a scanned product card inside the shopping cart.	Despite of the quantity indicated on the product card, the product is removed from the shopping cart, freeing space for more items. A snackbar pop-up appears asking if the user wants to undo the deletion.
7	The user presses ‘Undo’ on the popped up SnackBar after swiping horizontally on a product card.	The recently deleted product reappears in the shopping cart, occupying the same position in the list as before.

8	The user scans a QR code which does not belong to a valid product.	An error pops up informing the user about scanning an invalid product.
9	The user scans a QR code which belongs to a product which already exists inside the shopping cart.	A quantity of +1 is added to the correspondent product card.
10	The user scans a QR code which belongs to a product which does not exist inside the shopping cart.	A new product card is inserted into the bottom of the shopping cart, with a quantity of 1.
11	The user adds a tenth product to the shopping cart, reaching its limit.	All buttons which would increase the quantity of any product or add a new product are hidden from the user.
12	The user's subtotal for the current shopping order reaches a multiple of a hundred euros.	A message shows up on the bottom of the screen, informing the user of earning a loyalty voucher if the current purchase is completed.
13	The application and the server cannot communicate between themselves, *e.g.* during a request which retrieves the user's past transactions.	An error snackbar pops up, informing that an error occurred while contacting the server.

Past Transaction Review

Test Number	Scenario Description	Scenario Result
1	The user consults a previously done purchase where neither a voucher or the store credit discount have been applied.	A simple transaction card shows up, showing basic information about the context of the purchase.
2	The user consults a previously done purchase where a voucher was applied, but the store credit discount was not.	A transaction card with a black circle near the transaction date appears.
3	The user consults a previously done purchase where a voucher was not applied, but the store credit discount was.	A transaction card with the discount amount shows up underneath the transaction original value, colored in red.

4	The user consults a previously done purchase where both a voucher and the store credit discount were applied.	A transaction card with both the black circle and the discount amount colored red show up.

Purchase Confirmation

Test Number	Scenario Description	Scenario Result
1	The user does not use a voucher nor applies the store credit discount.	The original value of the transaction shows up in the 'Total' field.
2	The user selects a voucher but does not apply the store credit discount.	The original value of the transaction shows up in the 'Total' field.
3	The user does not select a voucher but applies the store credit discount.	The stored value is discounted to the current transaction value and information about the discounted amount and the new total value show up.
4	The user selects both a voucher and the store credit discount.	The selected voucher shows up in the dropdown selection, with the original value of the transaction showing up in the 'Total' field.
5	The user is not yet ready to finalize the purchase, so the button marked with a cross is pressed.	The user is taken back to the shopping screen, with the shopping cart exactly how it looked.
6	The user has confirmed the information and is ready to finalize the purchase, pressing the button marked with a check.	The user is taken to the checkout screen, where a QR code is generated, which should be scanned by the terminal app.

Checkout

Test Number	Scenario Description	Scenario Result
1	The user gets the generated QR code scanned.	After the QR code is scanned, an entry is added on the past transactions activity.
2	The user presses the ‘New Purchase’ button on the bottom of the screen.	The user is taken to the main menu screen, where a new purchase can be made.

Setup & Way of Use

Setup

For initializing the entirety of the elements composing the project one has follow several steps, being those:

1. Server initialization
    a. Position yourself inside the `/server` folder.
    b. Activate a virtual environment. To do so, run the following commands:
        i. In Mac/ Linux:

```
python3 -m venv venv
. venv/bin/activate
pip3 install -e .
```

        ii. In Windows:

```
py -3 -m venv venv
venv\Scripts\activate
pip3 install -e .
```

    c. Then, run the following commands to launch the server:
        i. In Mac/ Linux:

```
export FLASK_APP=flaskr
flask init-db            # For initializing the database
flask seed-db            # For seeding the database
flask run                # For running the application
```

        ii. In Windows (using Powershell running as administrator):

```
$env:FLASK_APP=”flaskr”
flask init-db            # For initializing the database
flask seed-db            # For seeding the database
flask run                # For running the application
```

    d. The server shall now be running in port 5000.
    e. Seeing that the server will be accessed remotely using the internet, it must be visible to do so. Hence, with *ngrok* one can expose a local server as a public URL. To do so, download *ngrok* and then position yourself in the folder containing it and run:

```
./ngrok http 5000
```

f. After completing the previous step, the terminal will show something similar to the figure below:

```
ngrok by @inconshreveable                                       (Ctrl+C to quit)

Session Status                online
Session Expires               7 hours, 59 minutes
Version                       2.3.35
Region                        United States (us)
Web Interface                 http://127.0.0.1:4040
Forwarding                    http://cab3c370.ngrok.io -> http://localhost:5000
Forwarding                    https://cab3c370.ngrok.io -> http://localhost:5000

Connections                   ttl     opn     rt1     rt5     p50     p90
                              0       0       0.00    0.00    0.00    0.00
```

g. The *http URL* being forwarded is what is of interest and shall be copied. In this copy, the user should copy *cab3c370.ngrok.io*. Notice that the *http* is not copied since the mobile applications inject the http header by themselves.

2. Customer application initialization:
   a. Open Android studio and load the customer application.
   b. Open the `/app/src/main/java/org/feup/cmov/acmecustomer/Constants.java` file and change the *SERVER_ENDPOINT* value to the string you have previously copied.
   c. Open the `/app/res/xml/network_security_config.xml` file and add it as a new domain, *e.g.*:

```
<domain includeSubdomains="true">cab3c370.ngrok.io</domain>
```

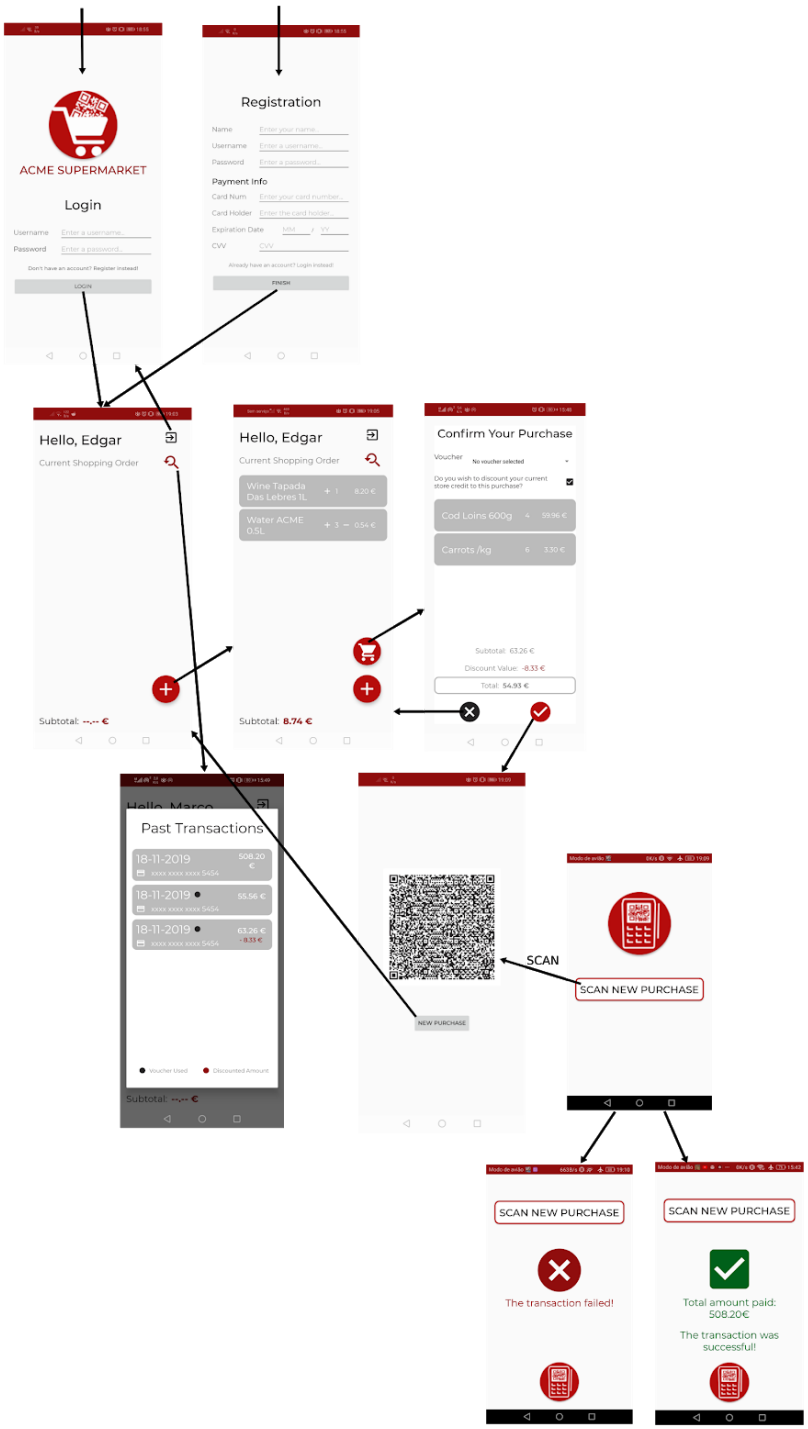
   d. Connect your mobile to your machine and run it there, as presented here.

3. Customer application initialization:
   a. Open Android studio and load the terminal application.
   b. Open the `/app/src/main/java/org/feup/cmov/terminalapp/Constants.java` file and change the *SERVER_ENDPOINT* value to the string you have previously copied.
   c. Open the `/app/res/xml/network_security_config.xml` file and add it as a new domain, exactly as it was done for the customer application.
   d. Connect your other mobile to your machine and run it there, as presented here.

Notice that for scanning the products present in the Annexes the server keys presented in the Annexes must also be used. For more information on how to do so, see the Annexes section.

Way of Use

To better understand how the application functions, a flow diagram is presented below.



Conclusion

The group believes that this project allowed us to put in practice a lot of skills acquired throughout multiple different curricular units in our master's course, more specifically, skills in programming languages such as *JAVA* and *Python* and how to use them to create a system which connects a backend server to multiple Android applications, while ensuring secure communications.

It is also relevant to state the difficulties the group encountered derived of the choice of tools used in the server: the way *Python* internally represents bytes proved to be a nuisance in the integration with cryptography algorithms and the way *JAVA* represents bytes. As a result, the group feels the time spent solving this issues was more than the one expected for this course, which prevents of developing even more additional features.

Nonetheless, the obstacles were surpassed and the developed implementation succeeds to correctly perform on the thoroughly developed battery of test scenarios. Additionally, several extra features were developed and the group believes that those greatly improve the user experience of the ACME supermarket applications.

To sum up, we also believe our implementation of the proposed communication protocol and implementation of multiple features, either core features for the applications to work accordingly or features we consider enhancements for the user experience, were thoroughly explored and brought more value to our learning experience, as we feel our implementation of this project was successful.

Bibliography

In the development of this project a vast set of references were used, namely:

- Mobile Computing Page and Resources, https://paginas.fe.up.pt/~apm/CM/
- Android Documentation for App developers, https://developer.android.com/docs
- Flask Documentation, https://flask.palletsprojects.com/en/1.1.x/
- Cryptography documentation, https://cryptography.io/en/latest/
- Python Documentation, https://docs.python.org/3/
- Stackoverflow, https://stackoverflow.com
- Nitranine, https://nitratine.net
- Crypto examples, https://www.cryptoexamples.com
- http://peetahzee.com/2015/02/securing-data-and-communication-with-rsa-across-android-app-and-python-server/

Annexes

Server Asymmetric Keys

For scanning the products presented next, one must also use the asymmetric keys presented below. To do so, copy them to the server, to the folder `/server/instance` with the respective names of *private_key.pem* and *public_key.pem*. The content of the *PEM* files is presented below:

- *private_key.pem*

```
-----BEGIN PRIVATE KEY-----
MIIBVAIBADANBgkqhkiG9w0BAQEFAASCAT4wggE6AgEAAkEA2PWSa6bMtverlXMd
Wl+Lb93sy9kZVOGkC6W3TguJ0iMspCj89BfdKJPtBfAfd8AoSqKc4l3FJLRkBN78
VfKAywIDAQABAkAW2VYATGLG6jKB3Mu8ls9iiqbDmSuXyuOx7lPtvj1XifcHYvQ3
DKIPlh8+VonyM9mP1Mzq2taOHROdpzkRq9KhAiEA/RaUt9b905WPo7VjEmB301ae
jm0nFFdklHyd+WTEDPkCIQDbdJSn2Ve3E5D6S13Je/RLBlNBIvSd7LRDJ2qGurMA
4wIhAMnE7tuBsuq0GcfSHCHSdrUuIUo7CyTUZ4NZtjfVdddBAiAPHLMT7+qTZ9yG
7+OweUG0XOYkySRxL5imOzOhvX+QIwIgTPQMDj2ZGEpx68CYTowEVJoYcgST5ZJa
KaO9QvZLTkk=
-----END PRIVATE KEY-----
```

- *public_key.pem*

```
-----BEGIN PUBLIC KEY-----
MFwwDQYJKoZIhvcNAQEBBQADSwAwSAJBANj1kmumzLb3q5VzHVpfi2/d7MvZGVTh
pAult04LidIjLKQo/PQX3SiT7QXwH3fAKEqinOJdxSS0ZATe/FXygMsCAwEAAQ==
-----END PUBLIC KEY-----
```

Products QR Codes

For testing the system as a whole and not having to generate new QR Codes from the products, the products QR Codes, using the afore-mentioned server keys and the products defined in the *seed.py* file, sorted by ascending price, are the following:

- *Water ACME 0.5L*, 0.18€

- *Carrots \kg*, 0.55€

- *Onions \kg*, 0.79€

- *Fusilloni pasta 500g*, 0.99€

- *Orange \kg*, 1.49€

- *Shrimp tempura 200g*, 3.79€

- *Wine Tapada das Lebres 1L*, 8.20€

- *Lamb leg \kg*, 8.99€

- *Cod loins 600g*, 14.99€

- *Vintage Whisky 0.7L*, 50.82€