CShare development report

Clara Gros
Babacar Toure

June 2020

Abstract

As part of our second-year project at CentraleSupélec in the science media pole, we choose to focus on the development of a food redistribution application to fight against food waste, intended for the scale of a university residence. This application would allow students to donate their surplus and collect others' food for free. That aims at limiting the ecological impact of students on a campus.

Contents

List of Figures

List of Tables

1 Introduction

As part of our second-year project at CentraleSupélec in the science media pole, we chose to focus on the development of a food redistribution application to fight against food waste, intended for the scale of a university residence. This application would allow students to donate their surplus food and collect others' food for free. That aims at limiting the ecological impact of students on a campus.

This report compiles all the work we have done so far (from December 2019 to June 2020). It gives an account of the evolution of our project, both on an entrepreneurial and technological aspect. On the one hand, it contains first the documents that allowed us to refine the need and define the specifications for the product. On the other hand, it includes diagrams and documents necessary to better understand the architecture of the solution and the technologies in used. Since the application is still under development, the report also offers hints and suggestions to encourage and continue the development of CShare. Up to now, we provide a robust API as well as an Android application developed according to the best practices recommended by Android itself. Nevertheless, as beginners, we encourage the review of this report and of our source code by experts to ensure the reliability and robustness of the server and of the application.

2 Needs analysis

2.1 Identification of needs

2.1.1 General concept

Our goal is to develop an application that allows users who are residents on a university campus to be able to give away their excess food and collect others' for free. This project was inspired by the other similar already existing applications such as Too Good To Go. Since we faced the problem of food waste ourselves as students, we decide to develop an application that would allow for redistribution of unused food on the scale of a campus.

2.1.2 Market study

Through a survey using Google Forms and distributed to all CentraleSupélec students via social networks, we estimate how receptive university students would be. We successfully reached out to over a hundred students from several campuses: the goal of this survey was to evaluate and refine the need for such an application, as well as quantify the potential user base we could expect to have, see appendix A.1.

Need One of the objectives of this survey is to evaluate how much food is wasted and the reasons for that wastage: what we found out reported in the charts below is that over two thirds of interrogated students said to waste food products, the main ones being fruits, vegetables and dairy products. The reason mentioned by three quarters of respondents is products were closing in on or had already passed their expiration date, with another major cause being leaving campus on holidays. Figures 1, 2 and 3 summarize students' needs and expectations regarding such an application.

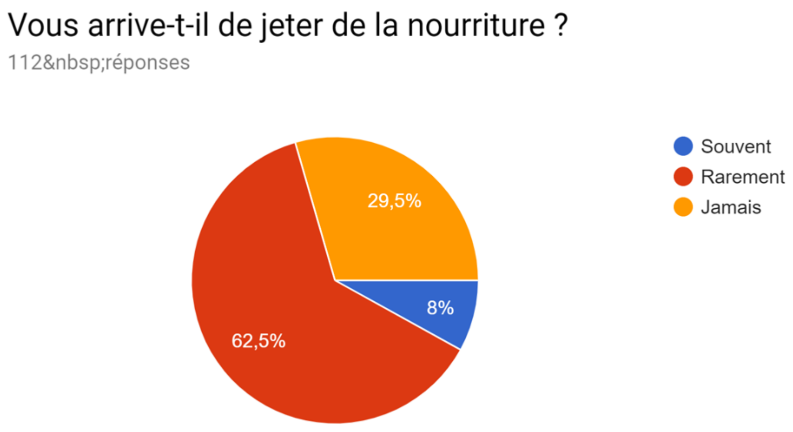


Figure 1: Frequency of wastage among students

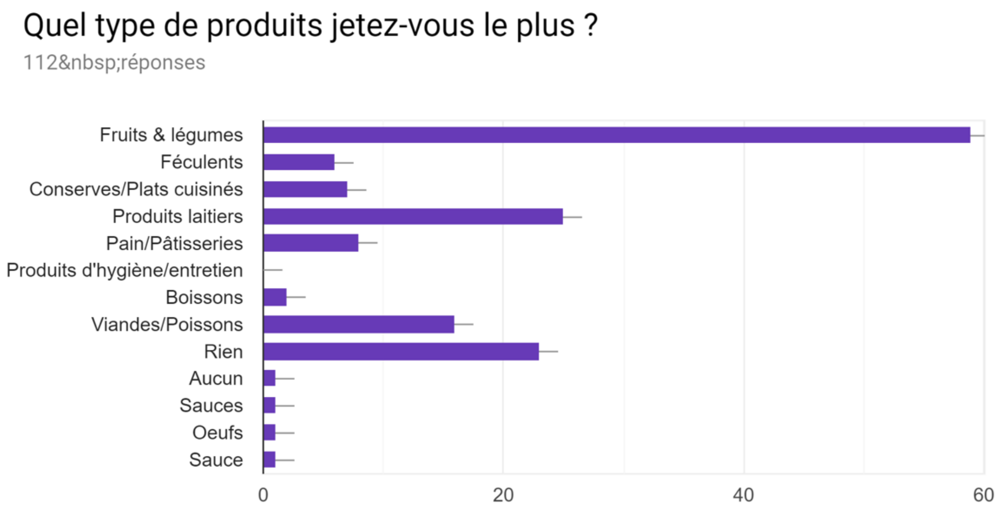


Figure 2: Type of products often wasted

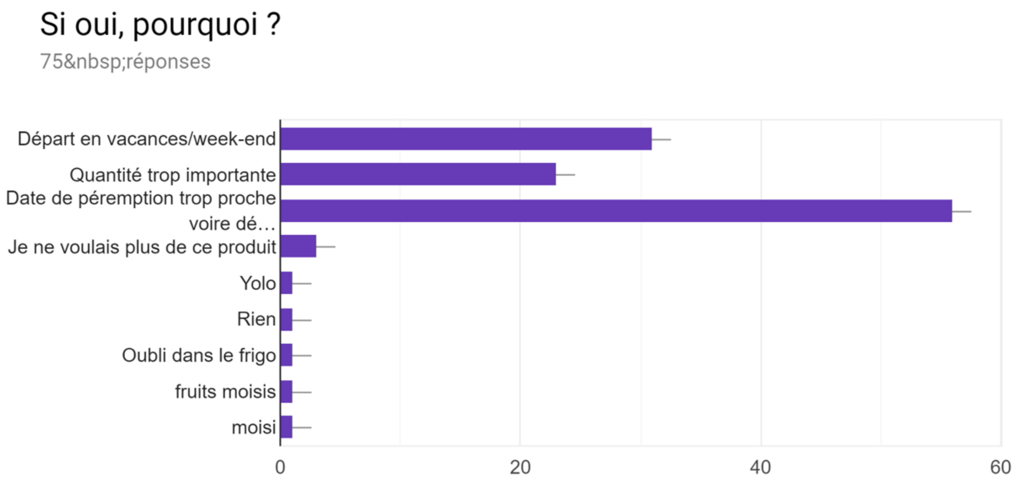


Figure 3: Reasons for waste

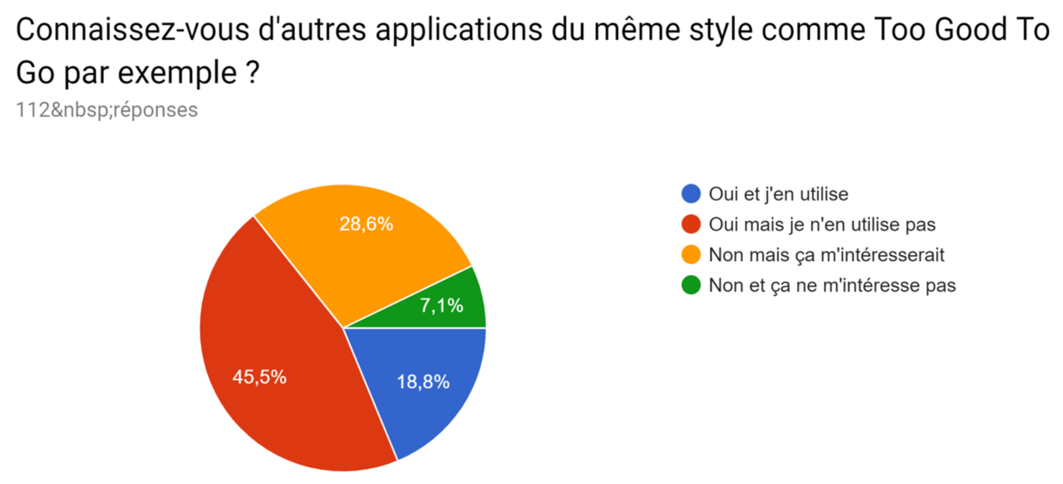
Already existing solutions There already exist anti food waste application that fulfill a similar role, the most widely used one being Too Good To Go (4). However, this application functions differently, as it targets grocery stores willing

Too Good To Go

HOP HOP food

Figure 4: Similar anti food waste applications

to sell their excess food at a lower price to individuals. We ask students if they knew of one and in that case if they actively use it, or if they did not know of any and in that case were interested in using one. We were pleasantly surprised to discover the number of respondents (over a quarter) who did not know of any such application but were willing to start using one as graphically shown in figure 5.



Connaissez-vous d'autres applications du même style comme Too Good To Go par exemple ?

112 réponses

Figure 5: Knowledge of similar food applications

From this chart we can see that less than a third of people who knew of such applications use one, but close to 80% of respondents who did not would be interested in using one, from which we can infer currently available applications do not fulfill the demand criteria on a university campus well enough and/or do not appeal enough. We hope to provide a more relevant solution to attract a large user base.

Digging a little deeper, we ask students from Gif who know about the Too Good To Go application but do not use it what is keeping them from downloading it. We got several interesting answers. First, the reason that stands out the is the fact of being isolated on the campus of Gif. Indeed, students do not live close to supermarkets and therefore cannot afford to order a basket because it is too complicated in terms of transportation. On the other hand, with Too Good To Go, although cheaper, the batches of products are paid for. Moreover, very often, the customer does not know in advance the contents of the basket. All these reasons hinder students in the use of such applications which nevertheless contribute to reduce the environmental impact of waste.

We found another interesting application named Hop Hop Food (see figure 4) that offers a product like ours. Yet, they only have 10k users throughout France (by way of comparison, Too Good To Go has more than 10 million users). The reason for this lack of success is that they are deployed on a country scale (versus campus scale for CShare) which makes it impractical because the products available on the application are often far away from our location.

It is through all these flaws that CShare's added value is illustrated. On the one hand, unlike Too Good To Go, not only the application but also the ordering of products is completely free for the user. Furthermore, reducing product exchanges to a single campus allows users to access food quickly and easily without having to take any transportation.

Limitations We also question the possible limitations and drawbacks this project would have, and according to respondents one limitation is users' willingness to accept food products from other people, which as seen in the chart below 6 is less on average than their willingness to give away unwanted products, although not by as much as we would have thought.

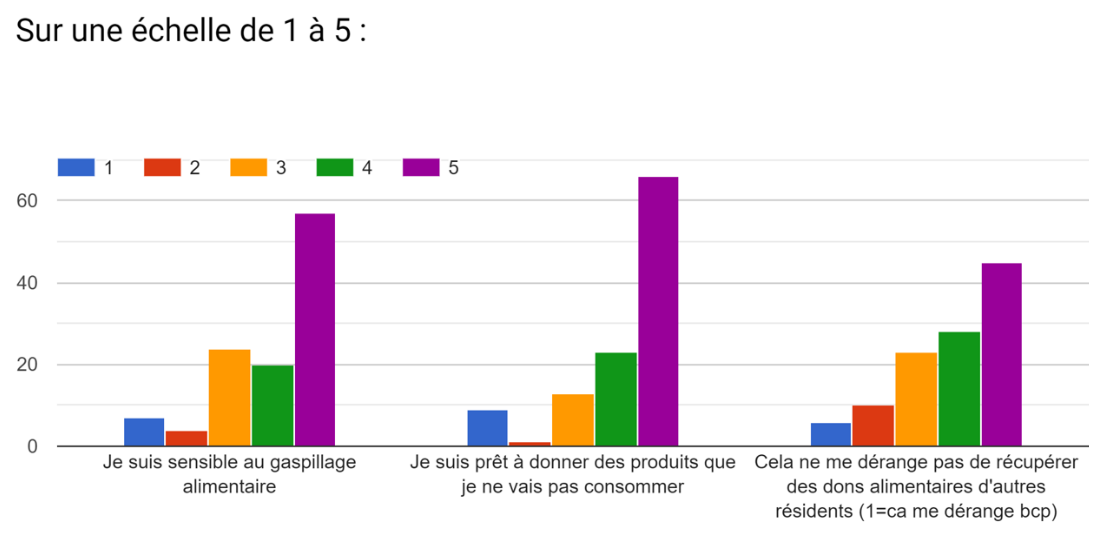


Figure 6: CShare potential limitations

Some also explicitly mentioned the safety risk of food coming from an unverified source: “Ce qui me dérangerait, c’est de ne pas savoir comment a été stockée la nourriture et de ne pas savoir par exemple si la chaîne du froid a bien été respectée.”

That is why we wish to later add 5 stars grading system to ensure the supplier good faith’ to give good products or at least allow users of the application to be able to report any malicious user 5.1.2.

In addition, regarding the health crisis of COVID-19, this confidence score assigned to each user to judge its reliability becomes more than essential because it is necessary that the sanitary conditions of food storage are irreproachable.

Support We have received very positive reviews on our project, and many verbally expressed their support: “Une excellente initiative !”, “Très bonne idée d’appli !”

2.2 Specifications

2.2.1 Objectives

Our main objective is to code a mobile application that communicates data to an API based on a database as well as a web administration dashboard application. The mobile application is expected to allow users to create an account, specify their personal information and be able to log in and out (for more information on user experience, see the navigation wire frames B.2). Once logged in, each user will have a double role. On the one hand, he can act as a supplier and share products he wants to give on the platform. On the other hand, he can also be a customer and order himself products posted by other users. There must also be a way to establish communication between two users (a supplier and a customer) during a transaction. Finally, each user will have a trust score to allow the administrator to moderate malicious users through the administrator dashboard.

2.2.2 Resources and constraints

Time constraint: We are limited on time as the project is expected to only last until the end of the school year. We are aware that this delay will not allow us to put our application into production, which will only be developed in Android for the moment. Yet, we would like to propose a first functional, cleanly coded and well-documented MVP so that we can then propose it to partners of the school who will be able to continue the project with a view to deploying the solution on the three CentraleSupélec campuses on both iOS and Android platforms.

Financial constraint: We have no budget to fund this project. Therefore, it seems wise to contact associations and partners of the school, especially the association Impact CentraleSupélec which promotes ecology through various projects to raise awareness and improve practices (see the Development prospects section 5.4). By being affiliated with this association, we would ensure that we have a minimal budget for the design, the review of the code by an expert, the hosting as well as the communication campaign once the application is in production.

Human constraint: Since we have little experience and no means other than online tutorials to train us, we need to consider using experts to ensure the reliability and robustness of both our API and our mobile application. To do this, we could call upon the Junior Entreprise of CentraleSupélec (JCS) who could submit the project to experts in this field.

Safety constraint: As we plan on maintaining databases that can contain personal information regarding users, it is necessary to tackle information security. We need to be GDPR compliant. As our application will store and share private information among users it is necessary to impose adequate Terms of Use as well as allow active moderation of publications (see the Development prospects section 5.1.2).

2.2.3 Use cases

To conclude this needs analysis, the figure 7 below is the use case diagram that summarizes the main specifications and functionalities of the project in the form of a black box view of how both users and administrators can interact with our system. For the sake of readability, extra use cases like log out and change password or reset password are not represented on this diagram.

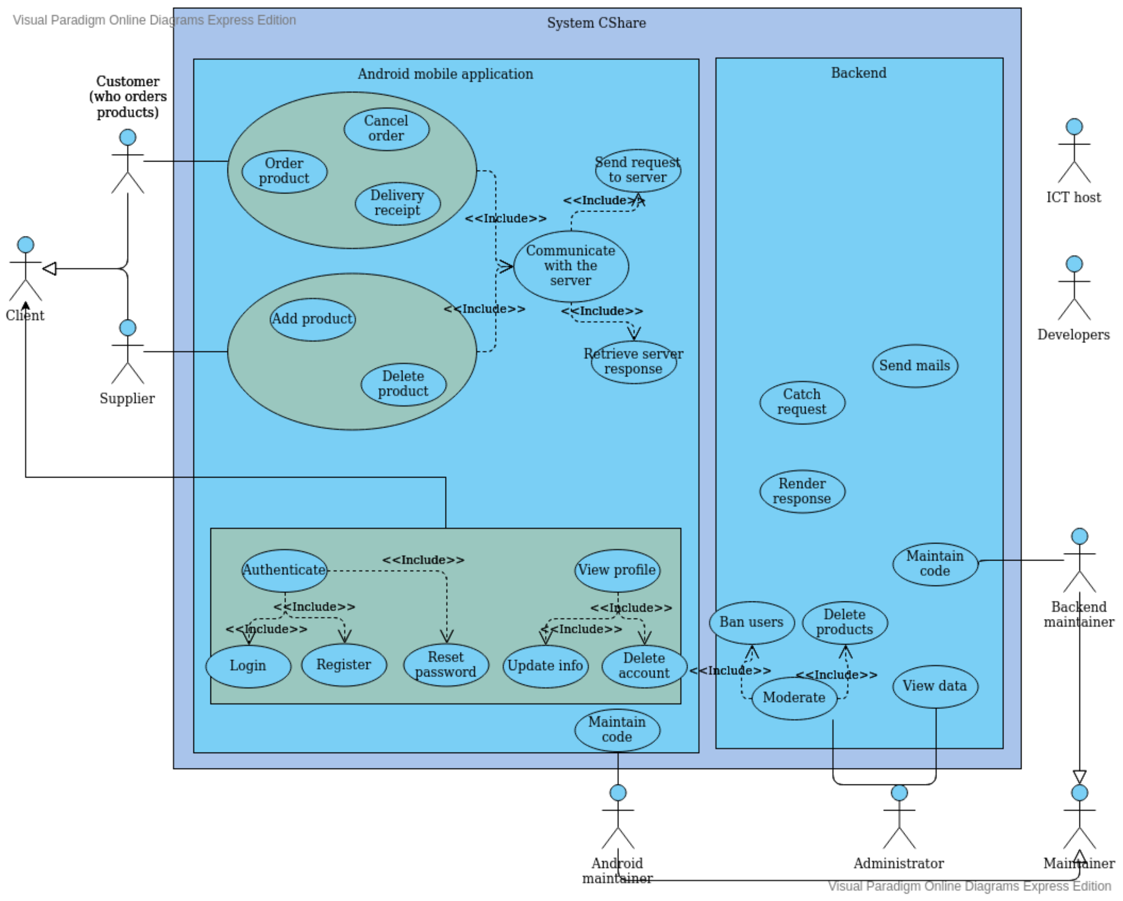
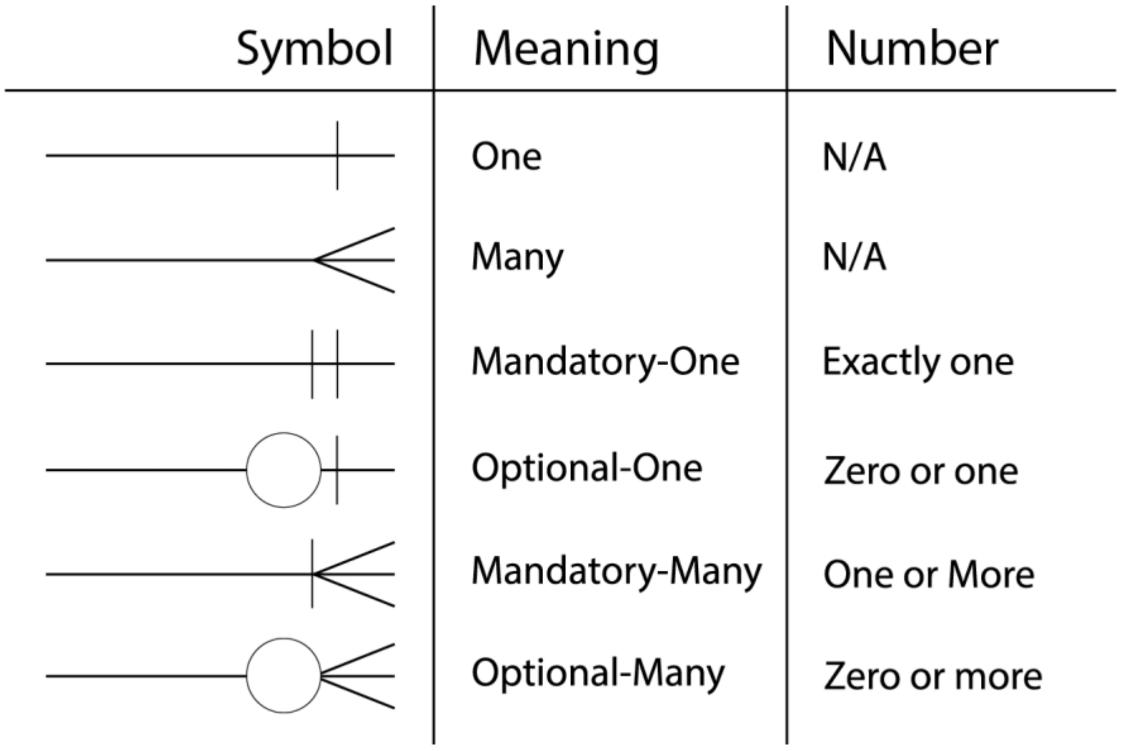


Figure 7: CShare use cases diagram

3 Solution architecture

3.1 Database modeling

The diagram 9 below shows the entity-relationship model (ER model) with the cardinalities using Crow's Foot notation. For further information on Crow's Foot notation, please refer to the picture 8 that sums up notations in used.



Symbol	Meaning	Number
	One	N/A
	Many	N/A
	Mandatory-One	Exactly one
	Optional-One	Zero or one
	Mandatory-Many	One or More
	Optional-Many	Zero or more

Figure 8: Crow's Foot notation

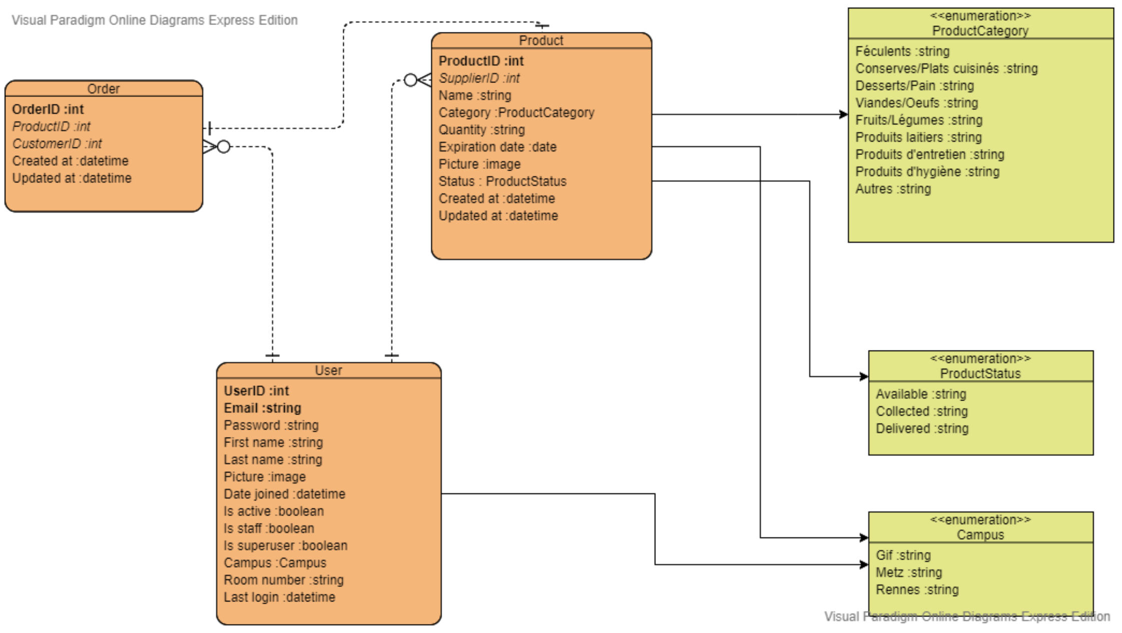
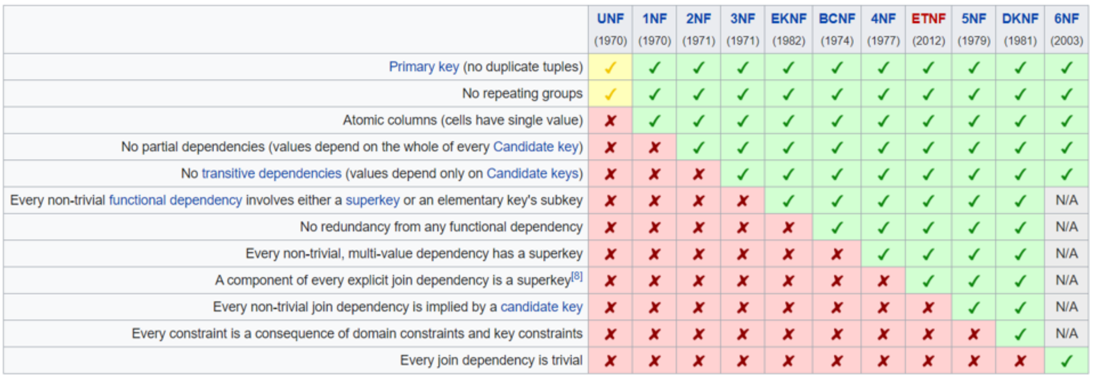


Figure 9: Entity-relationship diagram

The structure of this database is relatively simple. It is based on the presence of users who represent both/either a supplier and/or a customer. The table User has two primary keys : the user ID and the email. The product table primary key is the product ID. This table contains the information intrinsic to the product as well as the user who posted the product within a foreign key. To schematize the order of a product, we created the order table whose primary keys are the order ID and the couple customer + product. Indeed, the order table contains a foreign key symbolizing the customer and a foreign key representing the ordered product knowing that the information of the supplier is contained in the product. Thus we avoid redundancy.

Database normalization Database normalization is the process of structuring a relational database in accordance with a series of so-called **normal forms** in order to reduce data redundancy and improve data integrity. It was first proposed by Edgar F. Codd as part of his relational model. Normalization entails organizing the columns (attributes) and tables (relations) of a database to ensure that their dependencies are properly enforced by database integrity constraints. In the figure 10 from this website Wikipedia below are the normal forms (from least normalized to most normalized).



	UNF (1970)	1NF (1970)	2NF (1971)	3NF (1971)	EKNF (1982)	BCNF (1974)	4NF (1977)	ETNF (2012)	5NF (1979)	DKNF (1981)	6NF (2003)
Primary key (no duplicate tuples)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
No repeating groups	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Atomic columns (cells have single value)	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
No partial dependencies (values depend on the whole of every Candidate key)	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓
No transitive dependencies (values depend only on Candidate keys)	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓
Every non-trivial functional dependency involves either a superkey or an elementary key's subkey	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	N/A
No redundancy from any functional dependency	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	N/A
Every non-trivial, multi-value dependency has a superkey	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	N/A
A component of every explicit join dependency is a superkey[8]	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	N/A
Every non-trivial join dependency is implied by a candidate key	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	N/A
Every constraint is a consequence of domain constraints and key constraints	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	N/A
Every join dependency is trivial	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓

Figure 10: Normal forms features

Informally, a relational database is often described as "normalized" if it meets third normal form. Most 3NF relations are free of insertion, update, and deletion anomalies. The objective is to determine the degree of our database to see if it is well built.

- Our database is at least degree **1NF** since all the columns of the table contain values of the same type and all the data is atomic, i.e. each piece of information is assigned to a separate data field.

- Our database is also at least degree **2NF** since, in addition to meeting the criteria for degree 1NF, each non-key attribute is fully functional, depending on the primary key. Indeed, the ids of each table are super keys that allow us to uniquely name the individual lines of each table (no **partial dependency**).
- Our database is also at least degree **3NF** since, in addition to meeting the criteria for degree 2NF, there is no **transitive dependency** i.e. there is no non-prime attribute that depends on other non-prime attributes rather than depending upon the prime attributes or primary key.
- However, our database does not reach the normal form of Boyce Codd, **BCNF**, since in the user table, we have dependencies between key parts.
  user ID $\rightarrow$ email
  email $\rightarrow$ user ID

Thus, the degree of normality of our database is 3NF which is a good compromise between the number of tables and the ease of updates. In practice, most databases stop at this level.

3.2 API architecture

3.2.1 Definition

An API is a set of definitions and protocols that facilitates the creation and integration of application software. API is an acronym that stands for **Application Programming Interface**. APIs allow a product or service to communicate with other products and services without knowing the details of their implementation (see figure 11 extracted from here). APIs are sometimes considered as contracts, with documentation that constitutes an agreement between the parties: if part 1 sends a remote request according to a particular structure, the software in part 2 will have to respond according to the defined conditions.

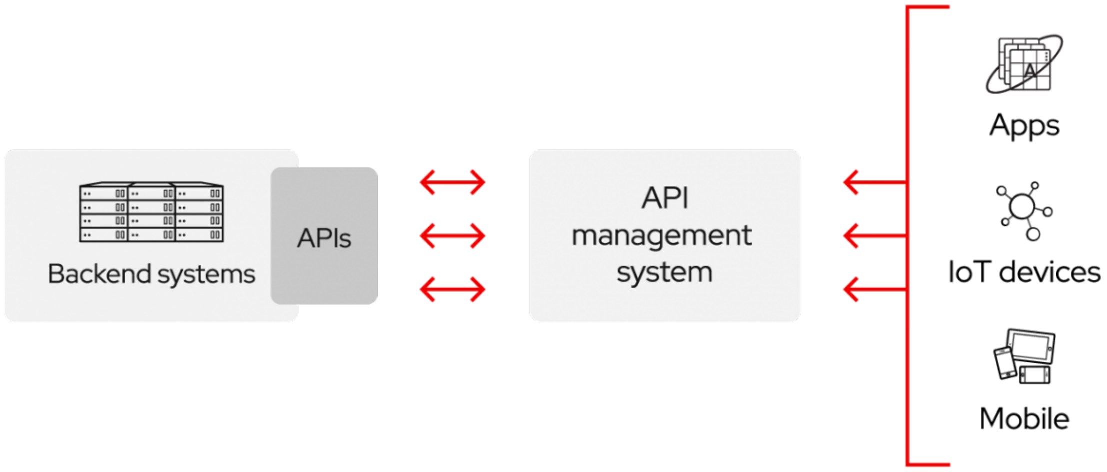


Figure 11: General principle of an API

For the client to communicate with the server that stores the data, we therefore use a remote API, designed to interact via a communication network. Here, "remote" means that the resources handled by the API are not on the device making the request. No database exists in the user's phone. We use Internet as a communication network and therefore our API is designed based on web standards. Our Web API uses the HTTP protocol for its request messages. The response messages are in the form of a JSON file. This format is one of the most common formats because the data it contains is easy to manipulate for other applications.

3.2.2 CShare API endpoints documentation

Since the major functionalities of the application are based on requests to the server, we used the **Postman Advanced API Testing Tool** that provides a nice visual interface to test our API endpoints. This way, regardless of the client, we were able to ensure that the requests were valid on the server side. Moreover, thanks to this tool, we are able to generate a complete documentation of the endpoints provided by the API that you can find in appendix B.1.

Figure 12: Postman logo

3.2.3 RESTful API

Historically, in order to standardize the exchange of information between the ever-increasing number of APIs, a protocol had to be developed: the **Simple Object Access Protocol**, or SOAP. APIs designed according to the SOAP protocol use XML format for their messages and receive requests via HTTP or SMTP. SOAP aims to simplify the exchange of information between applications running in different environments or written in different languages. **REpresentational State Transfer**, or REST, is another attempt to standardize. Web APIs that meet the constraints of the REST architecture are called RESTful APIs. The two differ in one fundamental way: SOAP is a protocol, while REST is a style of architecture. This means that there is no official standard that governs RESTful Web APIs. According to the definition proposed by Roy Fielding in his thesis "Architectural Styles and the Design of Network-based Software Architectures", APIs are RESTful if they meet the six design constraints of a RESTful system.
The CShare API tends to be RESTful because of its following features:

- **A client-server architecture:** A REST architecture is composed of clients, servers and resources and processes requests via the HTTP protocol.
- **A stateless server:** Client content is never stored on the server between requests. Session status information is stored on the client through a token.
- **Caching:** Caching allows you to dispense with certain interactions between the client and the server.
- **Resource identification in requests:** Resources are identified in requests and are separated from the representations returned to the client.
- **Manipulation of resources by representations:** Clients receive files that represent the resources. These representations must contain enough information to be modified or deleted.
- **Self-describing messages:** All messages returned to the client contain enough information to describe how the client should handle the information.

We choose to use the REST architecture for our API because of the numerous CRUD requests we have to make via the mobile application. Indeed, the biggest advantage of REST is the understanding that the Web is not a set of HTML pages but a set of resources, one of the representations of which is HTML. Others are possible such as JSON, XML, PDF, etc... On these resources, it is possible to perform 4 main operations which are GET, POST, PUT and DELETE, from the CRUD in summary. This is important when thinking about the choice of software architecture. If some web services are action-oriented (more complex than the simple CRUD operations) and require high security or a rigid exchange contract, SOAP will often be the most suitable protocol. Yet, as our web service is resources oriented with simple operations on them, REST will be the best choice.

3.3 Mobile application

3.3.1 Navigation in application

The wire frames that you can find in the appendix B.2 provide a representation of the structure of the application and allow you to navigate through the application within a simple hyperlinked PDF file.

3.3.2 Package diagram

Below is the package diagram of our Android mobile application (see figure 13). This kind of structural diagram shows the arrangement and organization of model elements. It is useful to have an overview of the structure between different modules. During the development phase, drawing up this diagram helps us a lot to apply as much as possible the concept of separation of concerns described in more detail in the section 4.3.1 dealing with the application architecture.

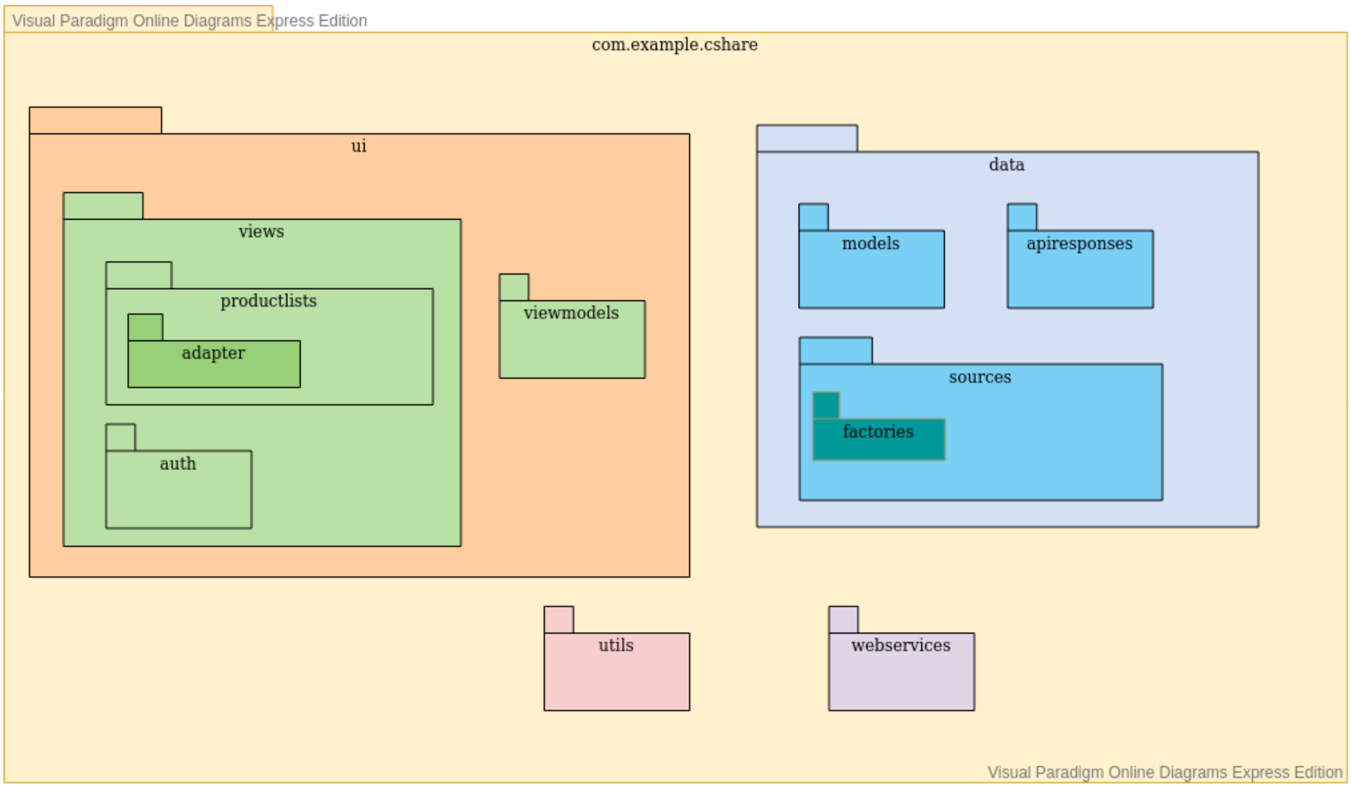


Figure 13: Android application package diagram

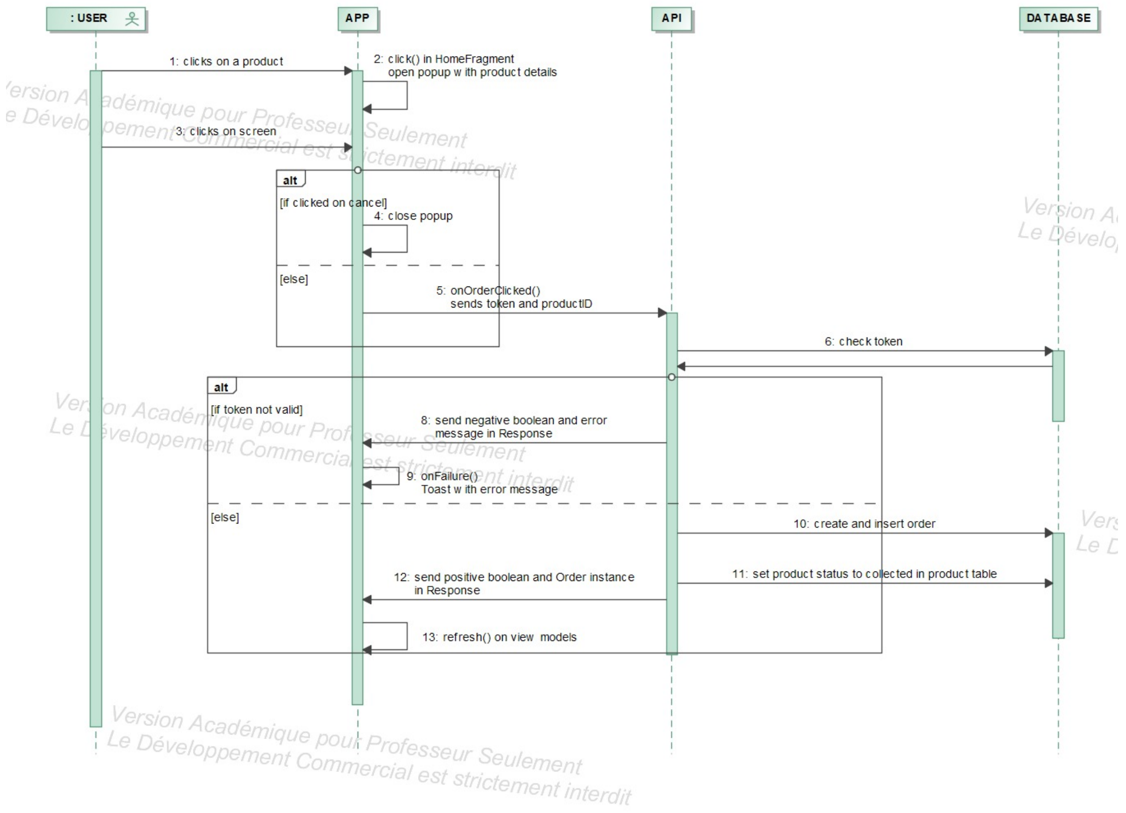
3.3.3 Class diagrams

You will find in the appendix B.3 all the class diagrams written in java. In order not to get lost in them, please refer to the package diagram displayed above 13.

3.4 Sequence diagrams

To model the behaviour of our mobile application, we draw up a sequence diagram that shows object interactions arranged in time sequence. It depicts the objects and the events scenario by showing actions and messages with horizontal arrows. In the appendix B.4, there are all the sequence diagrams we have made which are zooms on the main features of the application.

Below is the collect sequence diagram (see 14 that represents the action of collecting an available product via the application. The action here starts when the user decides to click on a product in the list of products of the home page. Here chronologically the user clicks on a product and the application opens a pop-up containing the product details. If the user clicks on "cancel", the pop-up closes. If the user clicks on the order button, the onOrderClicked() method that posts the token and the product to the API is called. The API checks the token. A token is required for any request. If it is not good it returns a Response object containing a negative boolean and an error which is displayed in a Toast of the application. If it is good, the API inserts the order in the order table and changes the status of the product to "collected". In return, the API sends a Response object consisting of a positive boolean and the command.



: USER
APP
API
DATABASE
1: clicks on a product
2: click() in HomeFragment open popup w ith product details
3: clicks on screen
alt
[if clicked on cancel]
4: close popup
[else]
5: onOrderClicked() sends token and productID
6: check token
alt
[if token not valid]
8: send negative boolean and error message in Response
9: onFailure() Toast w ith error message
[else]
10: create and insert order
11: set product status to collected in product table
12: send positive boolean and Order instance in Response
13: refresh() on view models
Version Académique pour Professeur Seulement
Le Développement Commercial est strictement interdit

Figure 14: Collect sequence diagram

3.5 Deployment

Deployment is the final phase of the project. We have decided to do it when the application is completely ready, i.e. when it has all the necessary features to provide a great user experience. The deployment will be done in a progressive way. Indeed, it is necessary to deploy a Beta version of the project to test it. We will deploy first the back-end or server thanks to a host. At that moment we will be able to provide an IP address to reach the API. Then we will test the mobile application to finally deploy it on the PlayStore.

3.5.1 Server deployment

The server consists of an API and a database. Its deployment must be done through a reliable and inexpensive hosting provider.

ViaRézo is the computer and network association of the CentraleSupélec campus in Gif. They provide multiple IT services to their members, including high-speed internet access. They have willingly accepted to host our server as they do so for school's associations. https://viarezo.fr/ allocates a Virtual Machine (an emulation of a computer system) for every server. The deployment of the server is to be done in several steps:

1. Creation of a Virtual Machine to host the server
2. Association of the IP address with a DNS entry (our domain name will be cshare-cs.fr/api/v1/)
3. Put the code of the back on the VM (including the database)
4. Deployment of the server

3.5.2 Application distribution

There are several application distribution and testing tools available.

- Google Play: With Google Play, there is an open or closed deployment. An open deployment is such that anyone with the PlayStore can download the application. On the other hand, a closed deployment is one in which the developer specifies who can download the application by creating an mailing list. There can be up to 2000 testers. Google Play is also a distribution tool. Yet, Google Play does not include test management tools. It just proposes to create a Google community along with testers.
- Crashlytics (fabric): Free tool that allows to perform tests with no limit on the number of users on an application. This tool offers the possibility to test both Android and iOS applications. It reports on all crashes and also provides metrics on acquisition, engagement, retention among testers.
- **HockeyAPP / App Center:** This tool can be used on Android, iOS or Windows applications. It allows to send a download link to users, to make

crash reports in addition to the distribution. Compared to Crashlytics, the user can send feedback directly. It is also possible to test several versions of the application at the same time.

We will distribute our Android application through Google Play for the obvious reason that it is better known and offers the minimal features we need to test and distribute.

To deploy an application on Google Play, we need to:

1. Create of a Google Publisher account that costs 25€
2. Test the application
3. Publish the application by uploading the APK file to the Google Account

3.5.3 Deployment diagram

The following deployment diagram (see diagram 15 shows the physical entities in communication once the application is deployed.

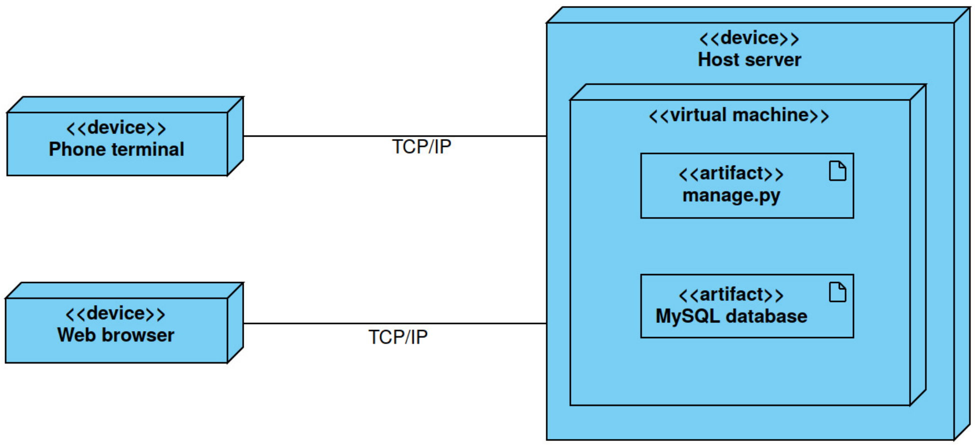


Figure 15: Deployment diagram

In this deployment diagram, there are three physical objects (nodes): the mobile phone, the computer and the host server. The smartphone is the terminal on which the application is running. A web browser will be used on the computer to access the administration dashboard. Both will communicate with the server through the TCP/IP protocol. TCP/IP (Transmission Control Protocol/Internet Protocol) or the internet suite is a model and a set of protocols used to transfer data through the internet. Here it means that every node has an IP address and is reachable thanks to it. The host node consists of a virtual machine that runs a MySQL database and the Django REST API.

3.5.4 Maintenance

Maintenance is very important because after deployment, we must ensure that users get the most out of the application. It is therefore essential that they can navigate easily through the application and for this, the content must be compliant. Thanks to Django, we have developed an administration interface on which an administrator will be able to regulate the content posted through the application (see the product reporting 5.1.2 section). The administrator will be able to block users, delete products and orders but also to sort and query the data in the database.

We have written a documentation of the administration dashboard so the administrator can get quickly familiar with this tool. (see appendix C.3).

4 Implementation

Before starting to talk about the implementation of the project, we want to redirect each reader who wants to resume the development of CShare on his own machine to the appendices C.4 and C.2 . Indeed, the appendix C.4 presents the instructions to correctly set up the project locally and the appendix C.2 is nothing but the documentation of the Java code of the Android application.

4.1 Tools and development environment

4.1.1 State of the art

Before developing the application, we have scanned all existing technologies to make an informed choice regarding our selection of languages used but also software, frameworks and databases (see appendix C.1).

4.1.2 Technical environment

In this section, based on the perimeter we have defined (see section 2.2) and on the state of the art mentioned above (see section C.1), we choose the best suited

Figure 16: Mainstream mobile operating systems

technologies to build our app. Indeed, we have used two different technologies on the server side and on the client side as well as one remote database.

Client-side The customer will have a mobile application and must be connected to the Internet to be able to make donations or collect food products. Given our time constraints, we build be a native mobile application because we do not have time neither to learn a new language, nor to develop an hybrid application. The choice therefore was to be made between the two mainstream operating systems: Android and iOS (16).

With a combined market share of Android and iOS of about 99%, these systems have become unstoppable. However, according to this Journal Du Net article, in the first quarter of 2020, Android had 78.8% of the OS market share in France, followed by iOS with 21.1%, according to data collected by Kantar World Panel (see chart 17).

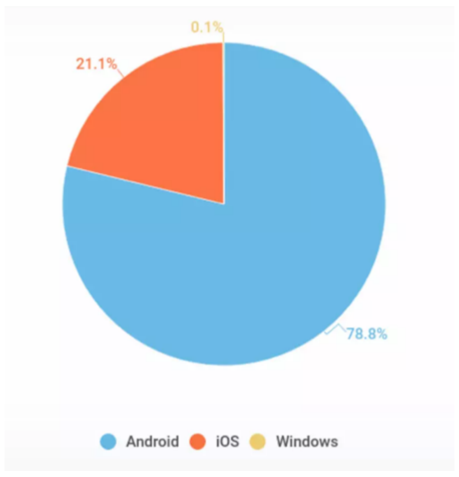


Figure 17: Distribution of OS on mobile phones in France in 2020

Android can count on a market share about four times higher than that of Apple. So we choose Android not only because the underlying programming language is Java (which we already know a little bit about) but also because on the scale of a university residence, we would like to reach as many people as possible and consequently, develop on the most widespread OS. Still, we keep in mind that it remains possible to extend the project or to entrust it to future students in order to be able to develop the application under iOS (see 5.3 iOS extension section).

Figure 18: Django python web framework

Server-side The role of the administrator would be more of a moderator role with regard to both users and food products circulating on the application. That is why we immediately thought of Django (see figure 18), a framework delivered by Python that almost instantly creates an administrator interface to keep control over the database with the basic CRUD operations. Therefore, we would provide the administrator with a very simplistic Web application to moderate the mobile application. In addition, it will provide an API to communicate with the database without writing SQL queries but with a model system (handled through the Django ORM). In the development phase we will use our computer as a local server.

Choice of the database For the choice of the remote database, we choose a relational database. These are most popularly used and useful for handling structured data that organizes elements of data and standardizes how they relate to one another and to different properties. This is exactly what we need because we will have several tables (see appendix **??**) and we will need to establish links between them all all the time. We choose to use the MySQL database manager (see figure 19). First of all, for economic reasons: we conduct a project using OpenSource products or products under free license from manufacturers. Certainly, if we were in a large project, we would have chosen a proprietary database option like Oracle (which is for many the best choice for study, practice or consensus). In addition, MySQL is quite intuitive and easy to use, even if these features are limited enough, it will be more than enough to meet the needs of our application.

Figure 19: MySQL DBMS

4.2 Back end libraries

4.2.1 Django

The communication between the server and the client is done by means of an API (Application Programming Interface) that tends to be RESTful to provide a great deal of flexibility. For this purpose, we chose to use the Django REST Framework python library which is a documented, accessible and powerful tool to build Web REST APIs. The REST framework will allow us to build an API describing the resources used in our Android application. To better understand its integration in the server, see the figure 20 below.

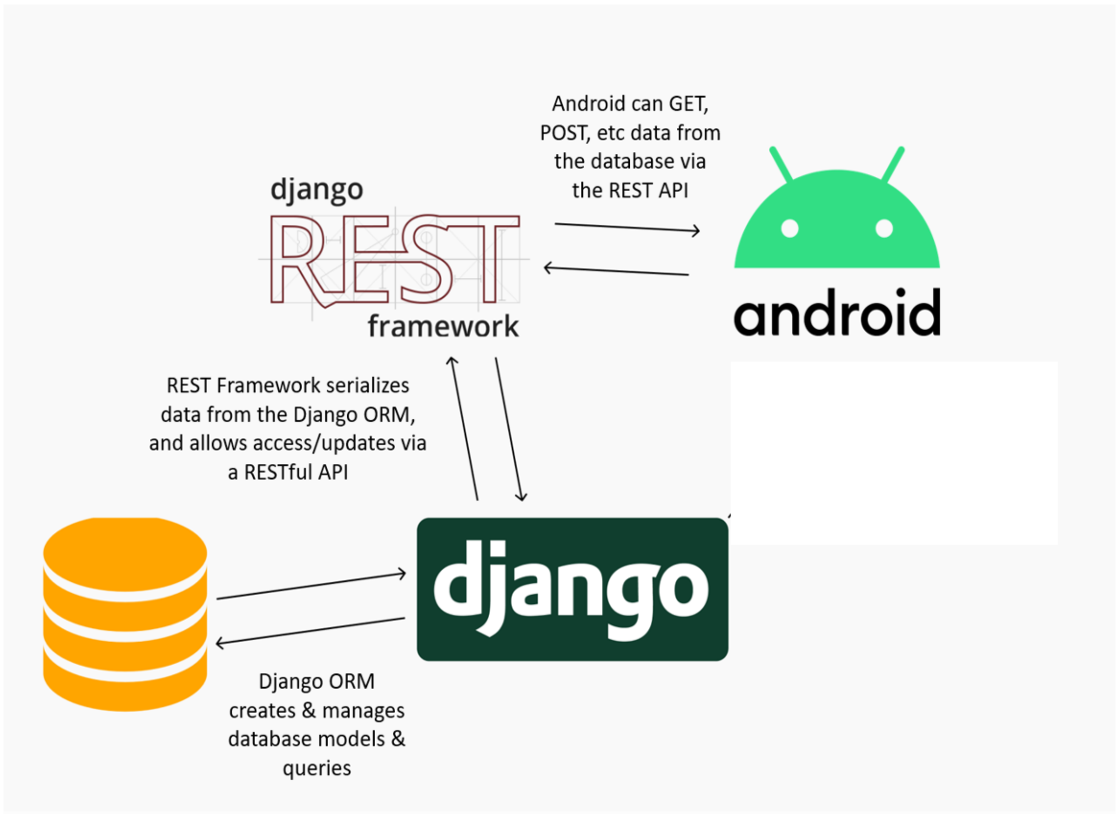

Android can GET, POST, etc data from the database via the REST API
django REST framework
android
REST Framework serializes data from the Django ORM, and allows access/updates via a RESTful API
django
Django ORM creates & manages database models & queries

Figure 20: Server architecture

4.2.2 Authentication

We have developed an authentication method based on Django token authentication. The token-based authentication works by exchanging email and password for a token that will be used in all subsequent requests so to identify the user on the server side. Token authentication is suitable for client-server applications, where the token is safely stored. It is important to note that the default token implementation has some limitations such as only one token per user, no built-in way to set an expiry date to the token but this is still quite acceptable in the case of CShare. We decided to use the library Django REST auth library that says: "Since the introduction of django-rest-framework, Django apps have been able to serve up app-level REST API endpoints. As a result, we saw a lot of instances where developers implemented their own REST registration API endpoints here and there, snippets, and so on. We aim to solve this demand by providing django-rest-auth, a set of REST API endpoints to handle User Registration and Authentication tasks.". Therefore the library provides automatically and according to the REST protocol, endpoints to create an account, log in and out, see its information, etc. It also allows the management of passwords (reset or change in case of loss). This is very useful to speed up the development process of our server as we were pressed for time.

4.3 Client libraries

4.3.1 Android architecture patterns

After having produced a first functional MVP rather quickly to get user feedback on our application, we decide to follow best practices and Android recommended architecture for building a more robust and production-quality application.

Problems we faced As beginners, we first put almost all the UI and data logic at the same place in our views (activities/fragments). For smaller projects, that is not really a problem but when an application gets bigger, this no **separation of concerns** can turn into a problem. Indeed, without any well-defined architectural pattern, it gets harder to make slight changes to a part of the code without always having a huge scanning effect through the rest of the code. Moreover, managing the activities and fragments life cycles become harder the most stuff we put in there. Also the spaghetti code is hard to test because many different parts depend on each other. Architectural patterns are there to help us create more separated modular components where each part of the program has a well-defined responsibility and can ideally be modified or replaced without touching any other component.

MVVM architecture Android Jetpack provides a bunch of different architecture components that are supposed to help us build more robust and maintainable apps. Giving the little time and experience we have; we do not use all the architecture components at our disposal but only the ones that make sense to us. The approach Android endorses consists in an **MVVM architecture** (that stands for Model View Model Model architecture). As you can see in the schema 21 below, if implemented correctly, we would have a nice and clean architecture where all layers are modular and decoupled from each other and every part has well-defined responsibilities: every layer only knows about the components directly below it. In short, the ViewModel class gets the data from the repository but it does not have to know how the data is retrieved from the different data sources (whether from a remote API or from a local database) and the UI controller trust the data from the ViewModel onto the screen but it does not have to store the data itself or initiate any database operations directly.

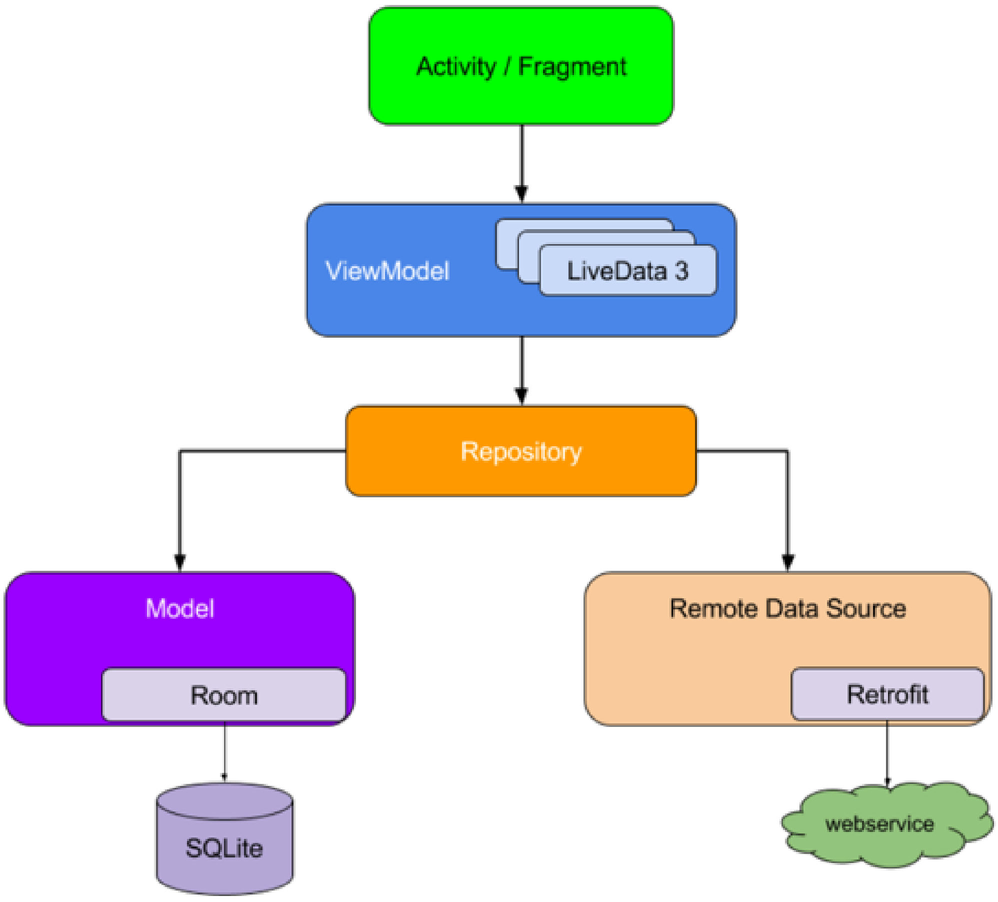


Figure 21: Model View ViewModel architecture principle

Let's briefly describe these architecture components:

- **UI controller** The controller (activity or fragment) is dedicated to the manipulation of the GUI.
- **ViewModel** The ViewModel component basically has the job of holding and preparing all the data for the UI so we do not have to put any of it directly into our views. Instead, the view connects to this ViewModel, gets all the necessary data from there and then only has the job of drawing it onto the screen and reporting user interactions to the ViewModel. The ViewModel then forwards these user interactions to the underlying layers of the application either to load new data or to make changes to the initial data set. So, the ViewModel basically works as the gateway for the UI controller to the rest of the application. This way we keep our activity and

fragment classes clean. The best specificity about the ViewModel class is it survives configuration changes (see the schema 22 below). For instance, when rotating an Android device or make any other runtime configuration change, the activity on the screen gets destroyed and recreated for the purpose of providing an alternative layout file or other resources. That means we lose the state of this activity and its member variables and we basically start with a completely new one. The ViewModel, on the other hand, is automatically restored when the configuration changes, so as soon as the new activity or fragment is created, it receives the same instance of ViewModel that still holds all the data, and the callback is invoked immediately using the current data.

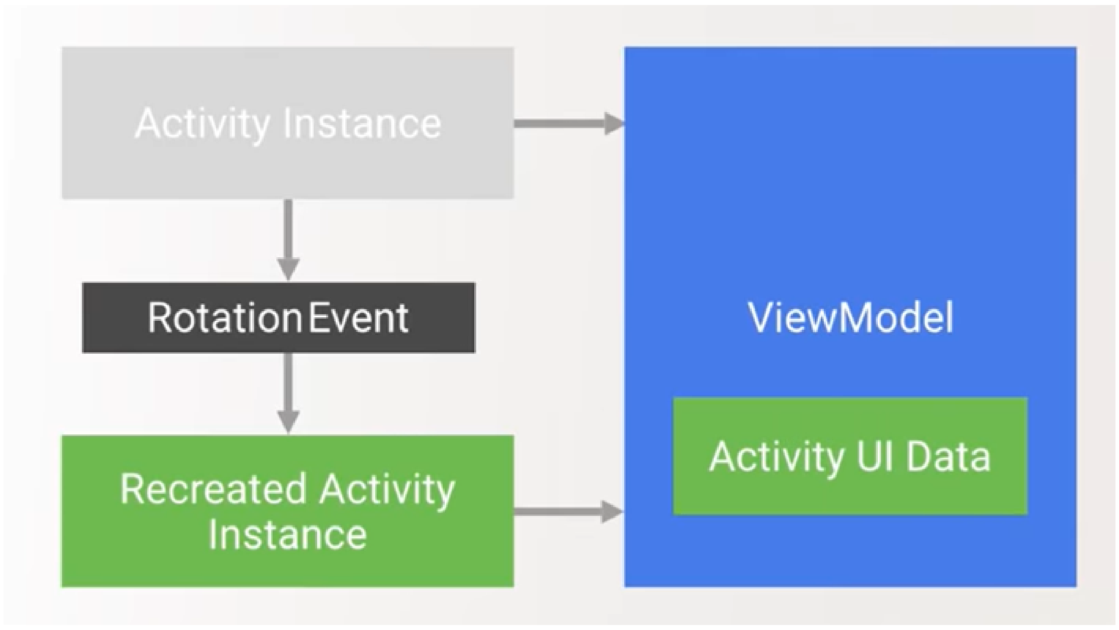


Figure 22: ViewModel survives configuration changes

- **Live Data** The data present within the ViewModel will be mainly of the LiveData type. LiveData is a wrapper that can hold any type of data and it can be observed by the UI controller which means whenever the data in this LiveData object changes, the observer automatically gets notified with the new data and can refresh the UI in reaction to it. With LiveData, we do not need to manage the unsubscription process ourselves and that LiveData has been thought for Android and its life cycle which means that it knows when the controller that observes it, is in the background and automatically stops updating it until it come back into the foreground so we do not have to manually stop and resume observations in our controllers lifecycle methods and it also cleans up any references when the associated controller is destroyed. To summarize, the ViewModel survives configuration changes and in it we have LiveData observables that automatically does the right thing at the right time in the lifecycle of our controllers and together they save us from a lot of lifecycle related issues and potential bugs and memory leaks.

- **Repository** A first idea for implementing the ViewModel might involve directly calling the data source to fetch the data and assign it to our LiveData object. This design works, but by using it, our application becomes more and more difficult to maintain as it grows. It gives too much responsibility to the ViewModel class, which violates the separation of concerns principle. Additionally, the scope of a ViewModel is tied to an Activity or Fragment lifecycle, which means that the data from the data source is lost when the associated UI object's lifecycle ends. This behavior creates an undesirable user experience. Instead, our ViewModel delegates the data-fetching process to a new module, a repository. This is an ordinary Java class and not an architecture component, but it is a recommended approach because in the repository we can mediate between different data sources (webservers and local SQLlite databases, such as persistent models and caches). This way, the ViewModel does not have to interact with the different data sources directly: it does not have to care where the data comes from or how it is fetched and instead repository modules handle data operations. They provide a clean API so that the rest of the application can retrieve this data easily. They know where to get the data from and what API calls to make whenever data is updated.

- **Data sources** Inside each repository, we will have the different classes allowing us to access our data, which are more commonly called Data Sources. For example, these classes can take the form of DAOs, or communication interfaces with APIs.

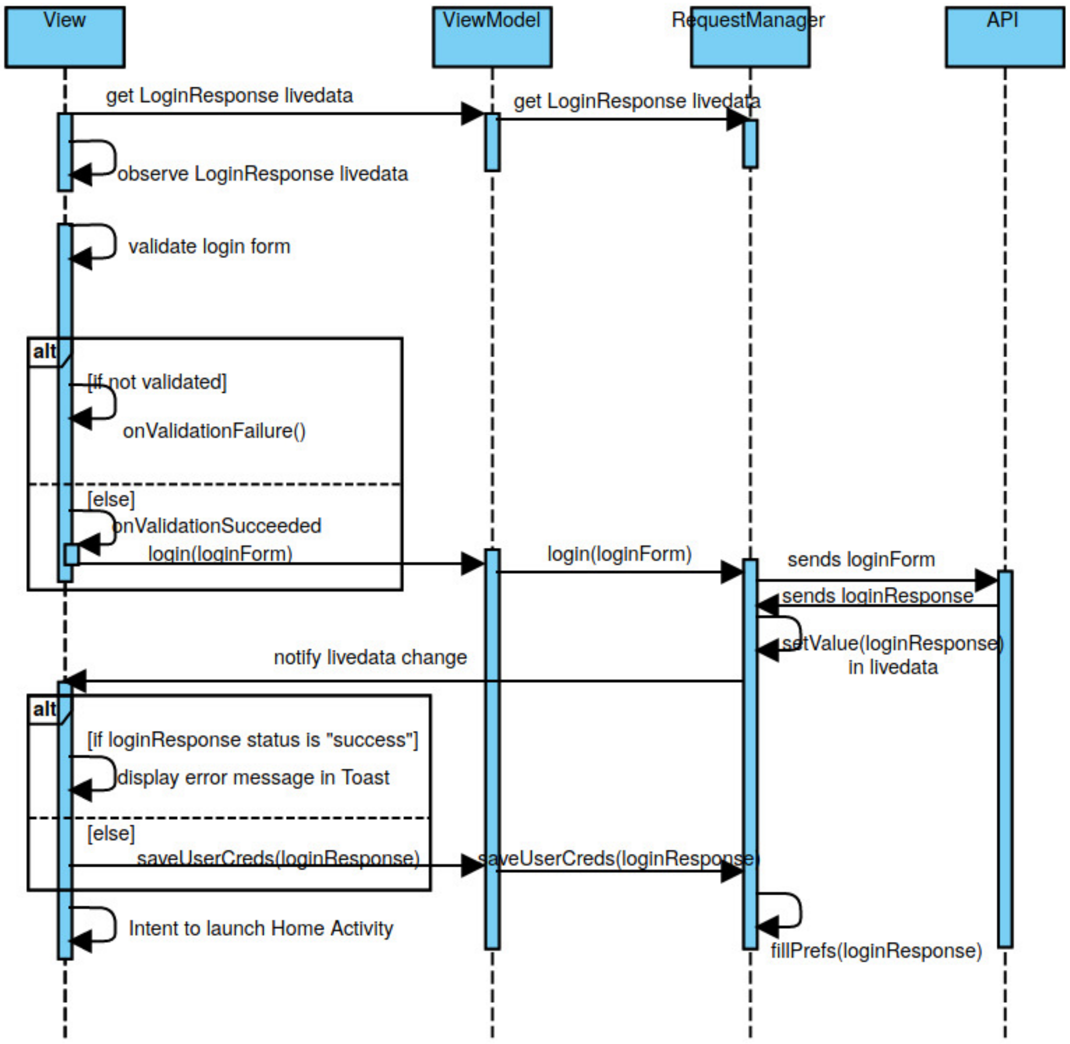


Figure 23: Login sequence diagram according to the MVVM architecture

The sequence diagram above (see 23) shows an extract of our java code to be able to log in to the CShare application. We have chosen to integrate it in this report to illustrate concretely how the MVVM architecture is implemented in our mobile application. In a few words, the LoginActivity first retrieves the LiveData object wrapping the response to the login request (initially, the status attribute of this object is set to neutral). Then, throughout its lifecycle, it subscribes to this object to observe its changes. Once the user submits the form to login, thanks to a callback function that is called once the form is validated, a loginForm object is created containing the information to send to the server and the viewmodel's login method is called which calls the repository's login method to be able to send the request to the server. Once the response from the server is received, the repository updates its live data object containing the login response (the status attribute of this object is now set to success), which notifies the server of the login activity of the change. Following this change, if the response is positive (i.e. the user has successfully connected and then the status attribute of the response object equals success), then the activity done records the connection data in the shared preferences via a call to a method of the viewmodel. Eventually, an intent redirects the user to the main activity.

4.3.2 RecyclerView

Interacting with a set of data is something that is recurrent in mobile applications in general. On Android, this is presented to the user with a list, a grid or any structured representation. The user can then view this data and scroll through this list. He can interact via single click or long click events.

In older versions, such a display was mainly managed by two essential classes: ListView and GridView. In version 5.0 Lollipop, Google introduced the famous RecyclerView: A modernized version of these two classes, which not only offers freedom to customize the content of the list in question, but also considerably increases the performance of the application in terms of time and memory consumption. Indeed, building a view is a heavy operation of the Android system. Therefore, in a list that can contain hundreds of items, the system will only create the number of views visible on the screen, and no more. When you scroll through your views and one of them disappears, it is "recycled" for reuse (as shown in figure 24).

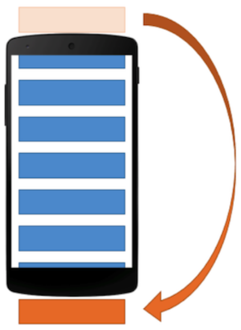


Figure 24: Recycler view

In the RecyclerView model, several different components work together to display your data. The overall container for your user interface is a RecyclerView object that you add to your layout. The RecyclerView fills itself with views provided by a layout manager that you provide. You can use one of our standard layout managers (such as LinearLayoutManager or GridLayoutManager), or implement your own.The views in the list are represented by view holder objects. These objects are instances of a class you define by extending RecyclerView.ViewHolder. Each view holder is in charge of displaying a single item with a view. For example, if your list shows music collection, each view holder might represent a single album. The RecyclerView creates only as many view holders as are needed to display the on-screen portion of the dynamic content, plus a few extra. As the user scrolls through the list, the RecyclerView takes the off-screen views and rebinds them to the data which is scrolling onto the screen. The view holder objects are managed by an adapter, which you create by extending RecyclerView.Adapter. The adapter creates view holders as needed. The adapter also binds the view holders to their data. It does this by assigning the view holder to a position, and calling the adapter's onBindViewHolder() method. That method uses the view holder's position to determine what the contents should be, based on its list position.

This RecyclerView model does a lot of optimization work so you don't have to:

- When the list is first populated, it creates and binds some view holders on either side of the list. For example, if the view is displaying list positions 0 through 9, the RecyclerView creates and binds those view holders, and might also create and bind the view holder for position 10. That way, if the user scrolls the list, the next element is ready to display.

- As the user scrolls the list, the RecyclerView creates new view holders as necessary. It also saves the view holders which have scrolled off-screen, so they can be reused. If the user switches the direction they were scrolling, the view holders which were scrolled off the screen can be brought right back. On the other hand, if the user keeps scrolling in the same direction, the view holders which have been off-screen the longest can be re-bound to new data. The view holder does not need to be created or have its view inflated; instead, the app just updates the view's contents to match the new item it was bound to.

- When the displayed items change, you can notify the adapter by calling an appropriate RecyclerView.Adapter.notify() method. The adapter's built-in code then rebinds just the affected items.

4.3.3 Android paging library

One problem we faced with the Android application we build is we request long product list data which we do not require a single time as a user only sees a small chunk of data at a time. One way to solve this problem is to listen to the scroll of RecyclerView and load more data when a user reaches the end of RecylerView. In this approach it was difficult to maintain efficiency, also UI logic data logic get complicated which makes debugging and testing of the project difficult.

For the rescue here comes a library named Android Paging Library. Paging Library is part of Android Architecture Components. It adds abstract layers between UI logic and data logic which makes code more clean and testable. Also, Paging Library is highly configurable with its PagingList Configuration (page size, initial page size...). Also, Paging Library efficiently handles fetching data from server updating on UI with minimum changes.

The next figure 25, extracted from this Google Codlab, illustrates the general principal of pagination in Android and below is a short description of the Paging library and its main components:

- **PagedList** - a collection that loads data in pages, asynchronously. A PagedList can be used to load data from sources you define, and present it easily in your UI with a RecyclerView.
- **DataSource and DataSource.Factory** - a DataSource is the base class for loading snapshots of data into a PagedList. A DataSource.Factory is responsible for creating a DataSource.
- **LivePagedListBuilder** - builds a LiveData<PagedList>, based on DataSource.Factory and a PagedList.Config.
- **BoundaryCallback** - signals when a PagedList has reached the end of available data.
- **PagedListAdapter** - a RecyclerView.Adapter that presents paged data from PagedLists in a RecyclerView. PagedListAdapter listens to PagedList loading callbacks as pages are loaded, and uses DiffUtil to compute fine-grained updates as new PagedLists are received.

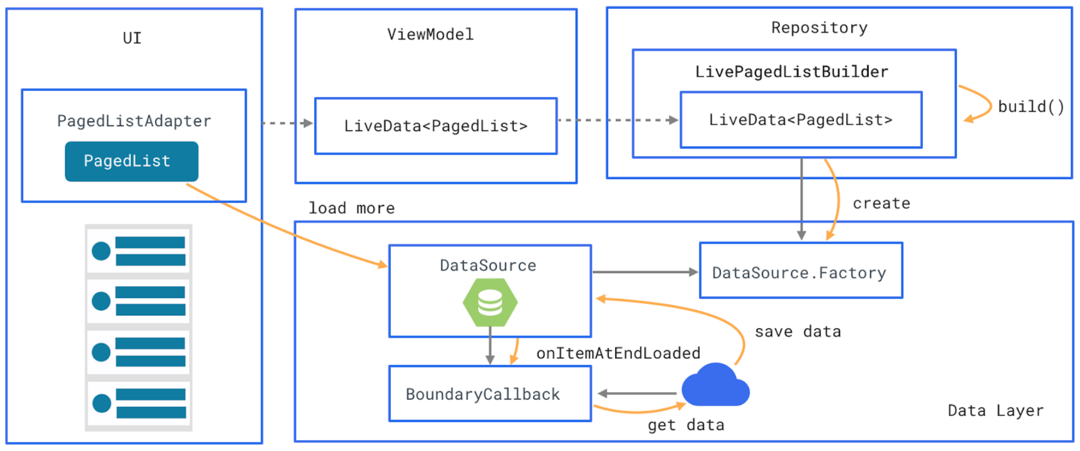


Figure 25: Paging library principle

4.3.4 Network client

Regarding Android side, we quickly realized developing our own type-safe HTTP library to interface with a REST API would have been a real pain: we would have had to handle many aspects, such as making connections, caching, retrying failed requests, threading, response parsing, error handling, and more. Retrofit, on the other hand, is a well-planned, documented and tested library that save us a lot of time. It is dead-simple to use. It essentially lets you treat API calls as simple Java method calls, so you only define which URLs to hit and the types of the request/response parameters as Java classes.

The entire network call as well as JSON parsing is completely handled by it (with help from Gson), along with support for arbitrary formats with pluggable serialization/deserialization. Documentation is great and the community is huge.

Retrofit is also compatible with the HttpLoggingInterceptor class of the OkHttp library which logs request and response information whenever the client issues a request.

4.3.5 Security

Our mobile application is very safe because it has very few security holes as it consists almost entirely of requests to the server. The only data locally stored in the Shared Preferences are the token, the user id and the is_logged_in boolean (that checks whether a user has logged in or not while launching the application). To prevent the user from accessing those, we did encrypt these key-value pieces of data using the Security Library part of Android Jetpack, that provides an implementation of the security best practices related to reading and writing data at rest, as well as key creation and verification. The Security library contains the following class to provide more secure data at rest: EncryptedSharedPreferences that wraps the SharedPreferences class and automatically encrypts keys and values using a two-scheme method. Keys are encrypted using a deterministic encryption algorithm such that the key can be encrypted and properly looked up. Values are encrypted using AES-256 GCM and are non-deterministic.

4.3.6 GUI

Image handling Retrofit does not come with support for loading images from the network by itself. So we used another small library called Picasso. Not only Picasso is just as easy to use as Retrofit, but it also avoids over-burdening the core library.
Moreover, Retrofit has full support for POST requests and multipart file uploads. Indeed, with *application/x-www-form-urlencoded*, the body of the HTTP message sent to the server is essentially one giant query string — name/value pairs are separated by the ampersand (&), and names are separated from values by the equals symbol (=). But for each non-alphanumeric byte that exists in one of our values, it's going to take three bytes to represent it. This is very inefficient for sending large files. Hence this method can not be used for sending image files. That's where *multipart/form-data* comes in. Multipart sends a single object (a file) in various parts, each part is separated by a boundary and has some portion of object's data. Each part also has its own headers like Content-Type for instance.

Forms and validation Regarding form validators, Django Rest Framework already takes care of setting up some validators on fields but we made the choice to set up validators directly from the client side to send the HTTP request only if the form is valid: this avoids useless and time-consuming requests. For this, we use the Android Saripaar which is a simple, feature-rich and powerful rule-based UI form validation library for Android. It is the simplest UI validation library available for Android which works with annotations placed above the views to impose rules on it. So for example, above an EditText intended to collect an email, the annotation *@Email* is placed on it to check that the right pattern is entered in the EditText. Then, once the form is filled in, the library sets up listeners to apply this or that method according to the validity or not of the form.

Design Out of curiosity to learn and in order to attract partners, we had to establish the first rules concerning the design, in terms of colours and shapes to make the CShare project attractive. So we created a provisional logo (see image

?? using the open source tool http://icon-library.com/icon/ enhanced by Google, as well as a first draft of the color chart. Finally, to implement this, we relied on the Material Design which is a design system library - backed by open-source code - that helps teams build high-quality digital experiences.

Figure 26: CShare temporary logo

5 Development prospects

At this stage of our project, at least we know how to implement the main functionalities. So new challenges are upcoming such as enhancing the user experience. Below are some essential features we want to add to our MVP. In addition to the first studies we carried out at the beginning of the project, halfway through, after having developed a first functional MVP, we decided to interview a panel of potential users by having them test our application. All the relevant remarks that we have retained are noted in the feedbacks appendix A.2. This could also be a potentially interesting source of development prospects.

5.1 Functionalities

5.1.1 User experience

It is important to make an application that is responsive, ergonomic and with a design that fits its context. For the moment, although operational, our application still lacks elements to provide a pleasant user experience. This is why we have chosen to delay the deployment phase. Indeed, if the application is not intuitive at first glance or if it lacks some of the usual functionalities that exist in all social network applications, the user may uninstall the application immediately and not using it again. In order to make the navigation in the application more fluid and intuitive in the manner of mainstream applications, we plan to develop the functionalities below in the possible future of the project.

Search bar An essential feature we should implement is the search among the available products. A search bar would be necessary in the Home fragment to allow the user to search for specific products by entering text.

Filtering In addition to this search feature, we also wish to make it possible for the user to filter available products according to their category.

Scanning bar codes As far as adding products is concerned, we have noticed that filling out a form can be tedious and painful for the user. So we thought of using an API that would automatically fill in the form from the product barcode. A simple scan of the barcode would then allow all the required data to be pre-filled, which would be a considerable time saver for the user. In practice, the API provided by Open Food Facts (see figure 27) seems to meet all our criteria. Open Food Facts is a free database of food products with ingredients, allergens, nutrition facts and all the tidbits of information we can find on product labels. To give some figures that testify to the popularity and therefore the richness of the data in this database, 15000+ contributors like you have added 660 000+ products from 150 countries using our Android, iPhone or Windows Phone app or their camera to scan bar codes and upload pictures of products and their labels.

OPEN FOOD FACTS
l'information alimentaire ouverte

Figure 27: Open Food Facts free database and API

To give you an overview of how to use the API, here is an example URL to read data for a product: `https://world.openfoodfacts.org/api/v0/product/737628064502.json`. The returned JSON file contains a lot of product information including the main information we need to fill out the form to add a product. Further information is available on the wiki.

5.1.2 Product reporting

For the comfort of users, any non-compliant or irrelevant product should be deleted by the administrator and the malicious supplier should be banned from the platform. To do so, each user will be affected with a trust score. The more reports a product will get, the lesser score the user will have. Regarding to a predefined threshold, the administrator will then have the possibility to remove the user and all his products from the application.

However, this system of moderation has flaws. Let's say there is a user who receives a bad trust score, he'll be banned from the platform. So he deletes his account and according to Article 17 of the GDPR, which highlights the **Right to Erasure**, he asks that all his data be deleted from the system. Indeed, the right to be forgotten means that when a user asks you to remove their data acquired through your website or mobile app, you are obligated to remove every personal detail you hold about them in all systems. Once his data is removed, nothing prevents him from recreating an account 100% identical to the previous one. To overcome this problem and therefore facilitate the administrator's work, one could consider applying a hash function on the user's data list and keep the result in an additional table. Each time an account is created, it would be sufficient to compare the new hashed key generated from the fresh data with the existing keys to see if the created account is 100% identical to an old one.

5.1.3 Instant messaging

One of the most important weaknesses of our application at the moment is the way users communicate with each other. Indeed, after testing on a panel, it appears that emails are not at all adapted to very short and informal exchanges such as those generated by CShare. Initially, one could imagine connecting users by SMS or via a third party instant messaging application such as WhatsApp (which is very popular among CentraleSupélec students). However, the best thing to do would be to build an instant messaging module completely integrated to CShare like Instagram for example. This would fluidify exchanges and encourage the number of transactions between students. At the moment, we haven't had time to investigate the question so we don't know what tools to use for such an implementation.However, in the medium term, we think that a simple redirection to an application like WhatsApp would be quite viable and appreciated by users.

5.1.4 Design enhancement

As mentioned before, we had to work on the design of our application, although this is not our main objective, to be able to present and sell it to potential partners.

However, the design is not complete because imprecise and not adaptable to any screen size for the moment. Furthermore, we still need to establish a proper graphic charter that should group and codify:

- The logo that must be able to be adapted and declined on all supports. It will be found on flyers, websites and on Google Play store, for example.
- The fonts and typographical attributes to be used, generally a title font and a content font.
- The choice of colours through colour schemes adapted to the requirements of the different communication media and the different backgrounds available (coloured or white background). It must include either the Pantone, CMYK or RGB and hexadecimal values of each colour used.
- The rules for inserting each element (logo, title, baseline,...) and for each medium: margins and positioning in the document. Editorial rules (tone and style) to help any writer to make his documents consistent.

This could be done in collaboration with CentraleSupelec's ssociation (whose logo is shown below 28) because once the charter is well defined, Android Studio allow easily to implement the front.

Figure 28: CSDesign CentraleSupélec association

5.1.5 Food education

One of our recent ideas regarding the guideline of our application, which aims to educate to a new mode of responsible consumption by limiting waste, would be to develop this educational aspect by introducing different concepts to the users of the application.

Nutrition facts The first idea that came to us naturally with regard to popular nutrition applications such as Yuka, would be to be able to indicate in more detail the nutritional facts of each product added to the platform. For example, we could think about the number of kilo calories or more meaningful indicators such as the nutri-score. Besides, the nutri-score is a reliable and popular indicator at the moment: in 2019, consumer associations are supporting a European citizens' initiative to propose to the European Commission that the nutri-score should become mandatory. It has already been adopted by Spain, France, Switzerland and Belgium.

Returning to the Open Food Fact (5.1.1) application mentioned earlier to make it easier to add products to the platform, we noticed that the API was already providing very interesting nutrition facts for existing products such as calcium, cholesterol, energy, iron, fiber, fat, salt, proteins, sugars, trans-fat, vitamins values.

In view of what has just been said, the platform could also integrate small explanatory documents to explain what this nutrient is used for, what daily intake we need, etc. We thus could develop a whole educational approach to teach the user to eat healthier.

5.2 HTTPS protocol

On the server side, Django provides a full description of possible security holes. However, considering our due date and the fact that our application does not contain sensitive data, we have chosen to limit ourselves to deploying the site in **HTTP**. If we keep using HTTP protocol, malicious users may intercept authentication data or any other information transferred between the client and the server and, in some cases, for active network attackers, that data may be modified in either direction. Another fundamental step for security is to switch requests from HTTP to HTTPS. This is currently not feasible because the Django runserver command does not support local HTTPS requests. Yet, before any deployment, we will have to setup the HTTPS protocol.

5.3 iOS extension

During the design phase of the project, made the choice to develop a native Android application. Indeed, we were taking a course on development under Android Studio so we had basic knowledge in it and this has influenced our choice for faster development. However, with hindsight and considering not only Metz but also the Rennes and Gif campuses, where the proportion of people using iOS is not negligible, we might have had to go for a hybrid app to deploy it on both iOS and Android.

5.4 Building partnerships

In addition to the network association of the Gif campus, ViaRézo (see logo 29), which agreed to host our server, in order to consider a continuation of this beautiful project which is not yet completed, since we both have a gap year on next year, we need to entrust the project to be sure that it will be taken over by students in the following years.

Figure 29: ViaRézo CentraleSupélec association

ImpactCS First, we contact the sustainable development association of the school, called Impact (see logo 30), which agreed to help us. Our CShare project is therefore officially attached to Impact CentraleSupelec. Thanks to this collaboration, we will be able to obtain a logo and a graphic charter provided by the design association CSDesign for free (without this affiliation, it would have cost 150€). In addition, Impact offers us to continue the project by entrusting it to students next year. Finally, when the time comes to deploy the application on the stores, Impact undertakes to communicate so that it can have a large number of users.

Figure 30: Impact CentraleSupélec association

6 Conclusion

After becoming familiar with all the development tools and after learning and implementing features such as queries, we realized mobile application development requires a lot of organization. It is necessary to have a clear model of the application and to do many iterations to be sure to meet the client needs. On the other hand, the developer has to be meticulous about good practices so that he does not end up with an application that needs to be rebuilt from scratch or that is poorly optimized and then provides a bad UI experience. We also note that we should not hesitate to go in depth through docs during the learning phase because many technologies lighten the development process if they are known.

So far we are proud of our achievements in terms of security of user actions on the client side and the way we manage queries. We also appreciated the fact that we were able to build up skills in Android development based on best practices. Nevertheless, hybrid development and deployment of a server and of the mobile application on the Play Store remain challenges to be overcome.

In addition to the technical aspect of the project, CShare allowed us to build a real team spirit. We were also able, for the first time, to work continuously on a long project by organizing ourselves using project management and versioning tools (all the more necessary in this lock-down period). We also appreciated the fact that we were able to mobilize external contributors, be they associations, teachers or even young developers to help us build this project that we hold dear to our hearts.

Appendix A Market study

A.1 Identification of needs

The **res/market_study/survey.pdf** presents the quantitative and qualitative questions posed to the student panel surveyed. The document **survey_results.pdf** in the same folder shows the results obtained with approximately a hundred responses. Please note that the comments of the study are not always very relevant given the freedom of the question but we present them to you anyway to see the constructive opinions.

A.2 Feedback on our first MVP

You will find this file entitled **feedbacks.pdf** in the res/market$_{s}tudy/directory$.

Appendix B Application architecture

B.1 API

Please refer to the **cshare_api.pdf** in the doc/ folder to get a sneak peek of all the endpoints provided by the API.
If you ever want to be added to the Postman Workspace to have an access to our Postman environment, please contact us by email at clara.gros@student-cs.fr or at babacar.toure@student-cs.fr.

B.2 Wire frames

Please visit the **report/wireframes** directory to access to hyperlinked pdf that allow to navigate between screens representing our mobile application.
If you want to get a more precise idea of our app and of our design, we invite you to watch the video demo in the **report**/ directory.

B.3 Class diagrams

To see the class diagrams, please open the folder res/architecture_diagrams/mobile_app/class_diag
which contains all the relevant class diagrams, each in sub directories corresponding to the package name in which the classes have been developed (see the package diagram 13.

B.4 Sequence diagrams

To see the sequence diagrams, please open the folder res/architecture_diagrams/mobile_app/sequence_diagrams which contains every sequence diagrams we have drawn.

Appendix C Implementation

C.1 State of the art

Please refer to the **state_of_the_art.pdf** file in the report/ folder to access our state of the art concerning technologies we have been through.

C.2 Android documentation

Please refer to the **index.html** file in the doc/javadoc/ folder to get a precise documentation of our Android code.

C.3 Administration

Please refer to the **admin_doc.pdf** file in the doc/ folder to learn how to use the administration dashboard to moderate the CShare application content.

C.4 Setup the project

Please refer to the **README.md** file in the ../ folder to quickly have access to the instructions to start developing the application on your own laptop.