UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

Development of a cross-platform mobile application using cloud computing

Dissertation presented by
Jérôme NAVEZ , Xavier PÉRIGNON

for obtaining the Master's degree in
Computer Science

Supervisor(s)
Etienne RIVIÈRE

Reader(s)
Sana IMTIAZ , Suzanne KIEFFER , Marc LAINEZ

Academic year 2017-2018

Acknowledgement

We would like to thank our supervisor Professor Etienne Rivière for his availability and valuable advice, as well as the readers Sana Imtiaz, Suzanne Kieffer and Marc Lainez. We would also like to thank everyone who helped us in any way.

Thank you,

Jérôme Navez & Xavier Pérignon

Abstract

With the evolution of technologies, many solutions exist so that people can be continuously connected to their friends (e.g. Facebook), to their job (e.g. LinkedIn, Outlook), even with famous people (e.g. Twitter, Instagram). However, most of them are not adapted to stay connected with our families by not providing solutions to meet the specific needs of a family living under the same roof. Faced with this lack of solutions in this domain, we present our product: *Domus*. The aim of our thesis is to explain the different features of our solution as well as the software design work, from the requirements writing to the final production through all the technical aspects. *Domus* is a smartphone application for every member of a family. Divided into modules, it contains multiple features meeting the needs of people living in the same house (e.g. chat room, shopping lists, loyalty cards, budget, etc.). This application is connected with a backend in the cloud. In order to stand out from most existing solutions, some unique aspects of our application are implemented such as integration between modules so that the user can effortlessly achieve different features that are dependent on each other. Thanks to a good methodology based on the agile approach and a strong collaboration with our supervisor, we were able to achieve our objectives by implementing a functional application that perfectly meets most of the requirements defined beforehand.

Acronyms

AOT Ahead-Of-Time compilation

APK Android Package Kit

APNs Apple Push Notification service

AWS Amazon Web Services

CAP theorem Consistency, Availability and Partition-tolerance theorem

DLL Dynamic Link Library

FCM Firebase Cloud Messaging

GCM Google Cloud Messaging

GCP Google Cloud Platform

GDPR General Data Protection Regulation

GUI Graphical User Interface

HCI Human-Computer Interaction

HIG Human Interface Guidelines

HTTP Hypertext Transfer Protocol

IDE Integrated Development Environment

IaaS Infrastructure as a Service

JIT Just-In-Time compilation

JWT JSON Web Token

MAUs Monthly Active Users

MBaaS Mobile Backend as a Service

MVP Minimum Viable Product

PaaS Platform as a Service

RCU Read Capacity Unit

S3 Simple Storage Service

SNS Simple Notification Service

SSP Supply-Side Platform

SaaS Software as a Service

UI User Interface

UUID Universal Unique Identifier

UX User Experience

WCU Write Capacity Unit

Contents

Introduction

The goal of our thesis is to present the process of designing a cross-platform application using cloud computing techniques for the backend. Although we have not yet implemented all the desired modules, we are proud to reveal the current version of *Domus*. Our application is adapted to stay connected to our families, and offers solutions to meet the specific needs of a family or a group living under the same roof. *Domus* is a smartphone application that can be classified in the lifestyle category. Whether we have a device running Android or iOS, it will allow us to communicate and share resources (e.g. chat room, shopping lists, loyalty cards, budget, etc.) with our family.

Objectives

Our main objective was to deploy an application that works perfectly in real conditions by meeting previously defined requirements. We wanted to take care of the quality of our work with a suitable methodology. That is why we also had to be careful to apply good programming practices as well as reuse techniques in order to have a good structure and maintainable application.

In addition, we had to make backend design choices so that the application could be efficient regardless of the number of users, while taking into account the costs of the technologies used.

We also had to meet the needs of users. In order to reach a larger number of users, we decided to design our application running on Android and iOS. They will have the opportunity to report an error or an improvement during a testing phase in order for us to better understand their expectations and their vision of the final application.

Since the application is intended for both platforms (i.e. Android and iOS), one of the objectives of the thesis was to design a similar graphical interface between them.

A very important objective was also the security of the application. Indeed, databases contain some sensitive information about each family. We must therefore ensure that only one member of a family has the privileges of retrieving information and resources of this family.

Of course, our application had to follow the General Data Protection Regulation [40] concerning the protection of natural persons with regard to the processing of personal data and the free movement of such data. Indeed, our application had to process user data such as name and pictures.

From a more educational perspective, we also aimed to take advantage of this experience to understand all aspects of the application designer's work. Thus, we wanted to do a full-stack development to learn to master all the stages of application design and the different technologies at our disposal.

Our solution

Our solution is a social network that focuses on the needs of a home by improving communication between family members. We can differentiate ourselves from other famous social networks by using the notions of horizontal and vertical social network [8]. Horizontal social networks are public places with a very diverse user base and a wide topical range. For instance, people are connected to Facebook just because they are friends, there is no niche interest. On the contrary, vertical networks are closed and focalized on a topic. In the scientific domain, for example, ResearchGate is a social networking site for researchers and scientists from all disciplines. Our solution is a vertical social network. Indeed, this one has a highly segmented user base and specializes in a single subject: the family.

The smartphone application is intended for each member of the family and contains multiple features divided into modules. The current version of *Domus* already has the following modules:

Chat room A place where members can communicate in real time.

Shopping lists Shopping lists of products to be purchased by the family.

Loyalty cards Common family loyalty cards that can be shared.

Budget Information about refunds requested by the family's children.

Upcoming versions of *Domus* will contain additional modules such as a shared calendar of events for all family members, the distribution of household tasks, a shared storage to store pictures and documents, a contact directory of shared family contact numbers, geolocation to locate family members on a map, etc.

When we researched existing solutions on the market, we concluded that most of these generally offer only one feature (i.e. only one shared calendar, only one shopping list, etc.) or offer multiple features but did not have much integration between them. In order to stand out from most existing solutions, it is imperative not to neglect some unique aspects of our application such as integration between modules so that the user can effortlessly achieve different features that are dependent on each other.

We decided to connect the application with a backend in the cloud. Indeed, cloud computing has a number of benefits such as [24]:

- ease of use,
- increased storage capacity and automation,
- cost savings,
- agility, flexibility and scalability.

Motivation

Throughout our years of computer science studies, we have followed courses in order to have sufficient background to prepare us for professional life. This project allowed us to combine the different theoretical and practical skills acquired during our studies, as well as to deepen certain aspects of software design.

In proposing this thesis idea last year, our objective was to prepare ourselves for working life by carrying out a project in a real situation, from the requirements writing to the final production of the application. Throughout the implementation of our work, we were able to apply concepts such as:

- software design and development,
- cloud computing design,
- databases design,
- human-computer interaction,
- UI and UX testing,
- agile methodology.

Plan of the thesis

First, we will explain our solution by going through all its specifications and requirements. We will also discuss the situation of our application in relation to the GDPR given that it is topical. Once the application is well defined, we will discuss the architectural constraints of our solution and argue our technological choices at the backend and frontend level. An explanation of the development methodology used will also be presented. Based on these technological choices, we can then look at the architecture and implementation in order to understand the structure of our application. In order to avoid confusion, Table 1 defines different technical terms used at the frontend level that may be misinterpreted.

Besides the technical explanations of the application, a chapter will then be dedicated to the graphical user interface which will state the guidelines followed to build the interface. The use of these guidelines and the functionalities of the application went through a testing phase. Then, we will take a step back and analyse the use of the platform where costs, limits, and ways of profit will be discussed.

Before moving on to the conclusion, a chapter is dedicated to possible improvements for future versions.

Term	Description
Device	The user's smartphone or tablet
Client	An instance of the application running on a device
User	A user, not necessarily in relation to a family
Member	A user in relation with a family

Table 1: Frontend technical terms

Chapter 1

Requirements analysis

This chapter describes the features of our application. Starting from the use cases, the requirements of our solutions will be determined. The aim is to provide more formal and structured information on the scope of the application.

1.1 Use cases

In this section, we will present our solution by exposing its multiple features through use cases. To do this, we present the Littlecloud family as shown in Figure 1.1. The Littleclouds are a typical family composed of two parents, whose name are Alice and Bob, and their two children, named Carol and David. We will therefore describe the different features of our application by giving role-playing games to members of the Littlecloud family.

In order to better visualize the features presented, screenshots of the application can be found in appendices.

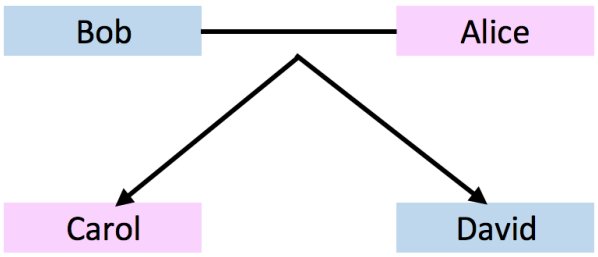


Figure 1.1: Littlecloud family tree

1.1.1 Identification

The application requires registration and authentication by the user in order to use it.

UC	Description
UC 1.1	Register
UC 1.2	Log in
UC 1.3	Automatic login
UC 1.4	Reset password

Table 1.1: Identification use cases

Bob, the father, decides to install the *Domus* application on his smartphone and opens it for the first time. Since he does not have a personal account yet, he creates one with his name, email and password (UC 1.1). Once his account is created, Bob activates it through an email confirmation in order to be able to connect to the application. Once the account is activated, he can log in with his email and password (UC 1.2).

If Bob exits the application and opens it again after logging in once, he is directly logged in without entering his e-mail and password (UC 1.3). He therefore does not waste time re-typing his personal information each time in order to enter the application.

Seeing that this application could be very useful, Bob decides to install it on his tablet. He has to sign in again on his new device in order to enter the main activity of the application. Unfortunately he just forgot his password. Luckily, he can reset it and generate a new one from his email address (UC 1.4).

1.1.2 Family management

The application allows the management of family accounts and its members.

UC	Description
UC 2.1	Add a family
UC 2.2	Invite by email
UC 2.3	Join a family
UC 2.4	Invite by Facebook
UC 2.5	Quit a family
UC 2.6	Assign roles to members

Table 1.2: Family management use cases

Bob does not belong to any family account. He decides to create the Littlecloud family and become administrator (UC 2.1). Bob prompts Alice, his wife, to install *Domus* and create a personal account. Then, he invites her to join the family account with her email (UC 2.2). She joins the family by accepting the invitation (UC 2.3). Alice can also invite people and their two children with Facebook (UC 2.4).

Users can be members of several families. Users can also leave a family account at any time (UC 2.5).

As an administrator, Bob can change the role (i.e. parent or child) of each member (UC 2.6). So Alice and Bob are listed as the parents. Carol and David are the children of the family account.

1.1.3 Chat room

A private chat room is available for each family.

UC	Description
UC 3.1	Send a message
UC 3.2	Send an image
UC 3.3	Save an image

Table 1.3: Chat room use cases

Bob, Alice and the children can communicate in real time with each other in a chat room. They can send messages (UC 3.1) and images (UC 3.2). Images can be saved locally on the smartphone (UC 3.3).

1.1.4 Shopping lists

The members of a same family can share shopping lists.

UC	Description
UC 4.1	Create a shopping list
UC 4.2	Add a product to a list
UC 4.3	Check/uncheck a product
UC 4.4	Select a loyalty card at the checkout
UC 4.5	Save a list as a template at the checkout
UC 4.6	Request a refund at the checkout
UC 4.7	Keep unchecked products in a remaining list
UC 4.8	Rename a list
UC 4.9	Remove a list
UC 4.10	Switch a list to active or template
UC 4.11	Edit a product
UC 4.12	Remove a product

Table 1.4: Shopping lists use cases

Alice, the mother, has added a "Food" shopping list earlier this week (UC 4.1). All week long, Littlecloud family members have added the missing products to the list. Each added product can contain a precision on the quantity and the estimated price (UC 4.2). On Saturday, David and Carol have some time and decide to go to the grocery store to buy the products on the food list. In the store, they go separate ways to find products more quickly. They each see in real time the products that have been checked (UC 4.3). Unfortunately, none of them could find the spinach.

At the checkout, David decides to pay. He selects a loyalty card, which he shows to the cashier (UC 4.4). He also decides to save the "Food" list in the templates so that he can reuse it for the next week (UC 4.5). After paying, David indicates on the application that he wants a refund (UC 4.6). Since the spinach was not found, a new list "Food (remaining)" was automatically added (UC 4.7).

Each list can be renamed (UC 4.8), deleted (UC 4.9) or switched to active or template (UC 4.10). Products can also be edited (UC 4.11) or removed (UC 4.12). When a product is added, it is absolutely required that its name is stated, other precisions are optional.

1.1.5 Loyalty cards

Loyalty cards can be shared between family members.

UC	Description
UC 5.1	Add a loyalty card
UC 5.2	Show a loyalty card
UC 5.3	Rename a loyalty card
UC 5.4	Change the logo of a loyalty card
UC 5.5	Remove a loyalty card

Table 1.5: Loyalty cards use cases

Bob has several loyalty cards, he would like to share them with his wife and children. He adds a loyalty card by specifying the shop name, optionally a logo and by scanning the barcode (or QR code) (UC 5.1). Once the loyalty card is added, any member of the Littlecloud family can select it to show the barcode (or QR code) when paying at the shop's cash desk (UC 5.2). The user can also rename the loyalty card (UC 5.3), change the logo (UC 5.4) or remove the card (UC 5.5).

1.1.6 Budget

The Budget module allows children to be refunded by their parents.

UC	Description
UC 6.1	Add a refund request
UC 6.2	Edit a refund request
UC 6.3	Refund a request
UC 6.4	Remove a refund request

Table 1.6: Budget use cases

Carol had a crush for a new pair of jeans and bought it. She indicates that she would like to be refunded by her parents. The application therefore adds a new refund request indicating the name, description and price (UC 6.1). While waiting for a refund, she can still change her request (UC 6.2).

Alice takes a look at the Budget module and sees that there are two refund requests. One from David for the weekly groceries, this request contains the list of the purchased goods. And another one from Carol. Since Alice is a parent, she decides to refund her two children (UC 6.3).

Once the request has been refunded, Carol can no longer edit it, she can only remove it (UC 6.4).

1.1.7 Settings

Each user can manage his account settings.

UC	Description
UC 7.1	Log out
UC 7.2	Change profile picture
UC 7.3	Change currency

Table 1.7: Setting use cases

1.2 Requirements

In order to meet all the needs of the use cases, several requirements are necessary.

The first requirement is to have a reliable **user management** that allows all classic actions to be performed (UC 1.x). A user also has access to several spaces but without having the same rights (UC 2.3). It is therefore necessary to define roles for these users according to the family in which they interact (UC 2.6).

A lot of family data needs to be stored. The modification and accessibility of this **structured data** must be reliable since users need this data to use the application and interact with the members of their family.

Of course, this data is private within a family. It is thus necessary that they are kept secret and accessible only by the members of this family and according to their role. This aspect concerns the **security** of the system.

In addition to structured data, users have the ability to add images within their families (UC 3.2). **Document storage** space is required. These images being also private, an access management identical to the structured data management is necessary.

Some actions need to be passed on directly to other family members. A service to **transfer information between two devices** is required.

Since several members can modify the same data at the same time (UC 4.3), it is better to have a **consensus system** in order to keep consistency in the data.

The application can also run **offline**. For some features, the user must be able to perform them offline. A data representation before disconnection must also be available. A **local storage** system must therefore be present in the application.

In addition to these requirements, it must be ensured that the application is in accordance with the legislation where it operates. It is in this perspective that the next section discusses GDPR.

1.3 General Data Protection Regulation

As *Domus* is used in the European Union and processes user data such as name and images, we have to comply with the GDPR [40]. The GDPR entered into application on 25 May 2018. In the context of our application, we must therefore apply these rules:

Consent Consent must be explicit for the data collected and the purposes for which the data are used. Children's consent must be given by the child's parent or tutor and be verifiable. The controllers (i.e. ourselves) must be able to prove consent. Such consent may be withdrawn at any time (Articles 4, 7, 8).

Right of access Users have the right to access their personal data and how this personal data is processed, the purposes of processing, with who the data is shared, and how this data is acquired (Article 15).

Right to erasure Users have the right to request the erasure of personal data (Article 17).

Right to the portability of personal data Users have the right to receive their personal data in a structured, commonly used and machine-readable format. Data that cannot be linked to the identity of the user are excluded (Article 20).

Right not to be the subject of profiling Users have the right not to be the subject of a decision exclusively based on automated processing, including profiling, which produces legal effects concerning it (Article 22).

Data protection principles Personal data must be stored in a secure computer system by pseudonymisation or tokenisation (Article 25).

Treatment activity records Treatment activities must be recorded along with the treatment objectives. The documents must be made available to the supervisory authority on request (Article 30).

Concerning *Domus*, the user must accept our privacy policy to be able to create an account and use our application. The user also has access to all data related to him. In addition to these two imperatives, we paid particular attention to the security of our system. However, in the current state of our application, some points are not yet respected. Indeed, the right to erasure and portability of personal data is not taken into account yet.

Currently, we do not profile the data we have. But in future versions of our application, if we incorporate the possibility of suggesting more relevant products to be purchased to a family, we will have to comply with the GDPR in terms of profiling.

Chapter 2

Technological choices

Before starting to implement a project, it is important to choose technologies that fit with the requirements. With this in mind, we analyzed and compared the different tools of developing a cross-platform application. From the backend point of view, several types of architecture are to be distinguished in order to choose the one which corresponds the best to the first version of the platform.

2.1 Backend

There are two classic ways to design a network architecture. The most common way for an application is to use a server side and a client side. There is, however, another way to achieve this using point-to-point communication.

In point-to-point communication, each node in the network acts as a client and a server. Data is distributed completely or partially between each node without centralization. In the case of the Domus application, each device in a family would therefore be a network node. The nodes would communicate using a push notification service or via a custom layer.

The problem with such an approach is that it is impossible to ensure that the data will remain available after uninstalling the application. The absence of centralization in a backend implies that if all family members uninstall the application, the data would be lost.

The network of our application thus works with the client-server principle. The application is connected to a backend in the cloud for several reasons [24]:

Ease of use Cloud computing is easy to get up and running. There is no need to download and install additional software. The cloud is also adaptable. Indeed, if we need more storage space, it is instantly available for a slightly higher price per month.

Increased storage capacity and automation The cloud offers virtually unlimited storage capacity and is flexible. File synchronization with all devices and file backups are also fully automated. Data is continuously kept consistent and up-to-date among all devices in use.

Cost savings We only pay for applications when needed and many applications are included free of charge. Usage can easily be scaled to fit our needs and adjusted according to peaks or troughs in demand.

Agility, flexibility and scalability Data can be viewed anywhere and edited simultaneously from multiple user locations. We must also be able to scale our cloud usage up or down on an as-needed basis and only pay for what we are actually using.

2.1.1 AWS as a PaaS

The solution we chose to use is Amazon Web Services as a platform as a service. This approach allows us to create a serverless architecture. In addition to inheriting the benefits of cloud computing, this type of PaaS has several other advantages:

Pricing AWS's pay-as-you-go approach allows us to pay only for what we use [13].

No maintenance Using a PaaS brings a higher level of abstraction than IaaS. This abstraction has the advantage of not having to manage the maintenance of the servers involved in our infrastructure.

Automatic scalability Most AWS components are capable of automatically scaling according to the load required. This is the case for Lambdas and DynamoDB for example.

Deployment speed Many varied services are available and allow applications to be developed faster. For instance, Cognito which manages user bases. The combination of these components makes it possible to obtain a result more quickly.

Despite these advantages, PaaSes also have some downsides:

Lack of standardization After choosing a PaaS cloud provider, it is difficult to migrate the entire infrastructure to another location. Even if some components between cloud providers are similar, their API is not the same.

Dependent of Amazon If Amazon decides to remove Cognito, we will have to redevelop the user management ourselves or go through another cloud provider. For example, although no official communication has been made, AWS appears to have abandoned SimpleDB [14].

2.1.2 Versus Monolith

An alternative to serverless architecture would be to make a monolith. Basically, a monolith places all the components of a system in the same application. For example, a Node.JS server with a MongoDB database, or an Apache server interpreting PHP and a SQL database.

A disadvantage of this approach is maintainability. If a new feature is added, the system gets bigger and bigger, then it is necessary to think about migrating the system to a more powerful server. It is also much more difficult to add new features because the system becomes too complex.

Whatever the load of the server, it will always be on and consume power, even if that load is zero. On the other hand, if the server is overloaded, the system will not hold, we will have the same problem as for maintainability.

2.1.3 Versus Microservices

An alternative to serverless architecture is to use microservices. Microservices separate the architecture into several components specialized in a task. These loosely coupled services interact and communicate with each other through the network. Each microservice has its own database.

A common way for micro services to communicate is to use RESTful APIs with HTTP. In this way communication is more standardized and it is easy to add other micro services. Another advantage is scalability and elasticity. Since each microservice is stateless, extending the number of instances of a service is not a problem.

Micro-service architecture involves the use of servers and containers. Whatever the platform load, there will be maintenance costs from the beginning of the installation. Compared to using a PaaS and a serverless architecture, the time to market increases as the effort to develop a microservice architecture is higher. To have a product quickly available, it is therefore better to use a PaaS.

2.1.4 Other PaaS providers

AWS is not the only solution when using a PaaS. Other platforms offer similar services, such as Google Cloud Platform, Microsoft Azure and Heroku. By following the requirements of our architecture, it is already possible to compare the different components offered by these different cloud providers (Table 2.1).

Service	AWS	GCP	Microsoft Azure	Heroku
NoSQL storage	DynamoDB	Cloud Datastore	Cosmos DB or Stockage Table	MongoHQ
Serverless computation	Lambda	Cloud Functions*	Functions	Dynos
File storage	S3	Cloud Storage	Stockage Blob	Bucketeer
Communication	API Gateway	Google Endpoints	API Apps	WSO2 API Cloud*
Users management	Cognito	Firebase	Mobile Apps	Auth0

Table 2.1: PaaS providers components

* Beta at the time of writing

2.2 Frontend

In order to reach a larger number of users, we decided to design our application running on Android and iOS. One solution would have been to follow the classic native approach by producing two different applications: one written in Java (Android) and the other in Swift or Objective-C (iOS). This approach has its benefits such as the use of official tools designed for this platform allowing them to be definitely supported as long as the operating system itself stays in development. In addition, this approach ensures compliance with human interface guidelines (HIG) and allows the user to get familiar experience interacting with the application (native UX/UI). However, this strategy is pretty expensive and time-consuming because the same code must be written twice implementing the same features.

2.2.1 Cross-platform development

Cross-platform development, in contrast, addresses this issue by providing powerful tools that use a single stack of technologies to develop both Android and iOS applications. Cross-platform development is an approach that allows a single codebase to be developed for several platforms. This approach has multiple advantages [27]:

Ease of learning Developers use a single technology stack and do not need to learn two different technologies. Although each platform has its own features and restrictions, the code can be written in a single language and GUI can be designed using a single tool.

Reusable code Part of codebase can be reused from one platform to the other. The code that is common for both applications such as business logic, data access and networking can be separated from the UI design.

Easy maintainability Maintenance is not performed on each platform separately. As the common code is written once, there is no chance to make some inconsistency between platforms.

A common potential issue of cross-platform development tools is that the developer cannot use the libraries that are familiar to him. Indeed, some libraries, e.g. for networking, become common for both Android and iOS. And a possible problem is that some features available only in the Android or iOS libraries may not have alternatives for cross-platform development tools. Then, there are several disadvantages which are explained below such as worse user experience, complicated integration of local preferences and notifications, UI disparities, lower flexibility, etc. These drawbacks differ according to the cross-platform development tools.

We therefore analyzed several cross-platform development software and frameworks in order to compare them and choose the one that best met our constraints and expectations. In the following sections, we will choose and explain the best cross-platform mobile development tools or frameworks, both because of the number of options they offer and their ease of use: Xamarin, Ionic and React Native [51, 52]. We will compare them according to multiple criteria (i.e. language stack, compilation method, performance, ease of deployment, library binding level, GUI, background code execution and pricing). We will then explain why we chose to use the development tool Xamarin.

2.2.2 Xamarin

Xamarin is a cross-platform development tool for building native Android and iOS applications, as well as Windows and Mac applications, using a single shared C# codebase and native libraries wrapped in the .NET layer. Xamarin has two approaches to build applications: Xamarin.Android/iOS and Xamarin.Forms. The first approach allows the developer to design layout files (AXML) for Android and storyboards for iOS using Visual Studio tools. All the native frameworks can be accessed easily. The graphical user interface is specific to each platform while code that rely on business logic can be shared between Android and iOS projects. The second approach allows the developer to design even the UI for two platforms at once [29].

Language stack As stated above, Xamarin uses C# and the .NET framework for all mobile platforms. The developer can translate in C# any code written in Java, Swift and Objective-C. The developer can also use .NET libraries in order to replace native open-source libraries available for Android and iOS.

Compilation C# is not compiled into the native code of the target CPU, but into some bytecode. C# can be either JIT (Just-in-time) compiled or AOT (Ahead-of-time) compiled. Both AOT and JIT can be used on Android. However, JIT compilation is not applied for iOS. AOT compiled code should run faster. In addition, with Xamarin, code can be compiled to a 64-bit target. Generally, 64-bit code runs faster and has a bigger memory footprint.

Performance Applications built with Xamarin.Android/iOS approach behave like native. Indeed, their cross-platform capabilities are focused mainly on sharing business logic rather than codebase. They use native user interface controls and leverage platform-specific hardware acceleration. Xamarin.Forms, in contrast, is focused on broad code sharing with less platform-specific behaviour. Almost the entire source code could be reused with this approach. This reduces code performance compared to the first approach.

Ease of deployment When the developer writes new code, the sooner he can observe the result, the more comfortable he is to develop. With Xamarin, the developer writes the code, builds it, deploys it on the simulator and then observes the result. A fast deployment

feature is also available on Android in debug mode that restarts the whole application, but without redeploying the whole APK.

Library binding Xamarin is very good at bindings and has a pretty full binding of the standard Android and iOS libraries. The developer can use libraries as if he was writing in native language (i.e. Java or Objective-C).

Graphical User Interface The Xamarin.Android/iOS approach is to use the platform-specific native user interface. Indeed, the developer can create a user interface with native UI layouts and controls. This approach ensures native look but consumes much time. The Xamarin.Forms, in contrast, automatically maps each page and its controls to platform-specific interface elements at runtime. This second approach makes the development process much faster at the cost of native look-and-feel.

Background execution Xamarin allows the developer to use the platform's familiar set of native features to run code in the background. The developer has to write different code due to all platform restrictions for both Android and iOS. However, he is not limited by using more restrictive platform features only.

Pricing Xamarin is an open source product. The IDE used to work with Xamarin is Visual Studio, which is proprietary and distributed on a subscription basis. It is available free for non-enterprise projects with up to five developers. It is also available behind Professional and Enterprise licenses containing more features.

2.2.3 Ionic

Ionic is a framework for building mobile, web, and desktop applications (i.e. hybrid applications) all with one shared code base and open web standards, using HTML5 and AngularJS. Contrary to Xamarin, Ionic does not use native widgets. Instead, it simply displays a web page written in HTML that mimics the design of native widgets [22].

Language stack Ionic uses web technologies such as HTML5, CSS, and JavaScript to write and run applications. It also uses Apache Cordova framework in order to access each device's capabilities like sensors, data, network status, etc. The Ionic main programming language is TypeScript (or simple JavaScript) that increases the quality of the code. Indeed, this language helps to spot and eliminate mistakes during code typing.

Compilation JavaScript is not compiled into the native code of the target CPU either, but into bytecode. Ionic uses JIT compilation for Android and WKWebView (platform browser) as the default for iOS. Although it is normally impossible to run JIT compilation on iOS, the WKWebView provides JIT conversion of JavaScript code down to machine code which improves rendering performance. In addition, with the current version of Apache Cordova, 64-bit mode on iOS is supported, but not on Android.

Performance Ionic does not use native components but web technologies to render an application and try to recreate native behaviour. This approach greatly reduces the speed and so Ionic cannot achieve performance results comparable to the other two cross-platforms on complex and rich applications.

Ease of deployment With Ionic, testing process is very fast. Indeed, using web technologies such as HTML and CSS allows the developer to run the mobile application directly in a browser. That means that if the developer changes some web files, only the necessary build steps are done, and the application is automatically updated as the code changes.

Library binding Unlike Xamarin, Ionic cannot just bind any random code the developer wants. Indeed, he needs to write some native adaptor code to create a plugin. Moreover, Ionic has its own abstraction layers between the standard library and the JavaScript application, which was not the case with Xamarin. This means that some OS features are missing.

Graphical User Interface Ionic does not use native widgets at all. The UI is created in HTML and CSS, and Ionic provides a set of Angular components to mimic the native widgets and make the application look native.

Background execution Ionic has a background mode Cordova plugin that does not provide means to actually run some background work, but only features such as avoiding the application interruption if it is backgrounded.

Pricing Ionic is also an open source product. A companion platform called Ionic Pro can be used and offers additional features such as more days of error history tracking and collaboration tools.

2.2.4 React Native

React Native is a framework developed by Facebook for building applications for different platforms such as Android and iOS, using JavaScript and React.JS. However, React Native works more like Xamarin. Indeed, a special templating language is used to make the GUI, and then native widgets are created [25].

Language stack React Native combines the advantages of JavaScript and React.JS (web framework). The developer can also write modules in Java, Objective-C or Swift languages when he needs them.

Compilation React Native uses JIT compilation for Android but interprets the JavaScript code for iOS. In addition, for the time being, 64-bit mode is not supported on Android, but it should be on iOS because Apple forces it.

Performance React Native performance is close to native. Indeed, code components are directly rendered to the native APIs. The shared codebase, i.e. approximately 80 percent of JavaScript code, can be used across the two platforms. The rest is native Android and iOS modules written in Java and Objective-C in order to optimize performance in complicated operations.

Ease of deployment React Native allows the developer to use either the classical approach, live reloading or hot reloading. The difference between live and hot reloading is that hot reloading does not involve a full application reload. Indeed, this approach replaces only changed parts of the application and preserves the state of the components.

Library binding As with Ionic, React Native cannot just bind any random code the developer wants and has also its own abstraction layers between the standard library and the JavaScript application, that leads to having only a partial support of OS features.

Graphical User Interface React Native modules interact with native Android and iOS UI controllers. However, not all native widgets are available. Even though the native widgets are used, the UI elements do not look native.

Background execution React Native only supports running background code on Android but not on iOS as we write this thesis.

Pricing React Native is a completely open source product and its libraries can be used for free.

	Xamarin		Ionic	React Native
Language stack	C#		JavaScript, TypeScript, HTML, CSS	JavaScript, Java, Objectice-C, Swift
Compilation	JIT or AOT AOT		JIT JIT (WKWebView)	JIT Interpreter
64-bit mode	Supported Supported		Not supported Supported	Not supported Supported
Deployment	Fast deployment (Android) and classical approach (iOS)		Deployment in browser	Live and hot reloading
UI widgets	Native		HTML, CSS	Native (not all available)
Library binding	Good at bindings and full binding of standard libraries		Moderate at bindings and partial binding of standard libraries	Moderate at bindings and partial binding of standard libraries
Background execution	Usual means		Plugin with few features	Supported only on Android
Pricing	Open source but eventual additional costs		Open source but eventual additional costs	Open source
	.Android/iOS	.Forms		
Performance	Close to native	Moderate	Moderate	Close to native
UI look-and-feel	Close to native	Moderate	Close to native	Low
Code reuse	Business logic, networking	Almost the entire code	Almost the entire code	Up to 80% of JavaScript code
Use cases	For all apps	For simple apps	For simple apps	For all apps

Table 2.2: Comparison of cross-platform mobile development tools

2.2.5 Our choice: why Xamarin?

Table 2.2 compares the different cross-platform mobile development software or frameworks according to the criteria defined above.

We chose to use the development tool Xamarin in order to carry out our project, and in particular Xamarin.Android/iOS, for multiple reasons. After comparison, we were able to conclude that Xamarin is the fullest and fastest tool compared to the others. Indeed, this one is a good approach to have a high performance, reliability and full hardware support. The performance metrics are comparable to those of Java for Android, and Swift or Objective-C for native iOS application development.

Moreover, Xamarin stays close to classical approach to the native applications development. As we already achieved some Android applications during our university studies, a lot of things are familiar. We can also use platform's features without restrictions and the development process is similar to the one with Android Studio and Xcode. We just still had to discover the world of iOS application development.

	Xamarin			
	.Android/iOS	.Forms	Ionic	React Native
Performance				
UI native look-and-feel				
Modern development features				
Code reuse				
OS features				

Table 2.3: Strengths and weaknesses of cross-platform mobile development tools

Then, as mentioned in the introduction to this thesis, we want to prepare for professional life. And Xamarin Company certainly emerged as one of the leading solutions for cross-platform development. We also aim to deepen our knowledge in C#/.NET, using the Microsoft and Visual Studio products.

Although React Native uses modern and fancy features such as hot reloading and instant updates, this approach has worse performance than Xamarin and does not fully offer native look-and-feel to applications. With Ionic, applications will look almost like native, but creating the UI in HTML and CSS may be painful and considerably reduces performance. Finally, we opted Xamarin.Android/iOS over Xamarin.Forms for essentially the same reasons (i.e. performance and UI look-and-feel). Indeed, Xamarin.Forms allows to develop simple applications where the UI is not as important as functionality.

Table 2.3 summarizes the main strengths and weaknesses of the different cross-platform mobile development tools we analysed. Although we chose to use the development tool Xamarin.Android/iOS in order to have a high performance, OS support and UI native look-and-feel, this approach also has a number of disadvantages [36]:

Code reuse Unlike the other tools mentioned above, Xamarin.Android/iOS is focused mainly on sharing business logic, networking and data access rather than codebase. This means that the GUI of each platform (i.e. Android and iOS) must be designed independently, which can take much time in the development process.

Limited access to open-source libraries Native development makes extensive use of open-source technologies. With Xamarin approach, some .NET open-source resources are provided. However, the choice is not quite as rich as it is for Android and iOS mobile application development.

Basic knowledge of native languages required As mentioned above, a platform-specific layer of code has to be written in order to build mobile applications with truly native look and feel. At least a basic knowledge of native technologies is therefore required (i.e. Java for Android and Swift/Objective-C for iOS). During our years of study, we only built Android applications. We therefore had to adapt quickly to iOS mobile application development.

Larger application size Xamarin applications are typically larger than native ones, sometimes up to twice the size of the latter. For instance, a simple "Hello World" application for Android might take up to 16 MB. Indeed, the content of a full package Android version looks like Figure 2.1. The package must include the application, the associated libraries, the content, the Mono runtime and the Base Class Library (BCL) assemblies.

Figure 2.1: Content of a full package Android version of "Hello World" application [32]

2.3 Development methodologies

The development of a mobile application until its release is a very long task. It is therefore important to evolve methodically. From the creation of the requirements to the production, including the testing phases, we needed an optimal organization to ensure professional quality and efficiency in our work.

In order to better understand how we proceeded throughout this project, we will first explain the approach on which we based our work: the Agile software development. Then, we will explain the first contacts we had with our supervisor. Since we wanted to build and deploy a mobile application as professionally as possible, our supervisor played the role of a potential customer.

2.3.1 Agile software development

We decided to use an agile approach for multiple reasons:

- we did not yet have a clear and precise idea of the application. The agile approach allowed us to adapt requirements as the project progressed,
- we knew that in the field of mobile applications, there is always a means to find new features, new modules to add during the project. The agile method was a good solution to cope with this constant evolution,
- once we met our supervisor, our relationship has been excellent from the start. This made communication easy and enjoyable. Mr Rivière was ready to get involved in our project in order to give constructive feedback,
- the methodology is well established and used in many companies nowadays.

The term "Agile" was popularized by the *Manifesto for Agile Software Development*, dating from 2001 [17]. Following a high failure rate of development projects during the 1990s, the developers behind this term concluded that it was necessary to find new methods to adapt to customer's needs. Indeed, it often happens that customers are unable to determine their needs precisely and definitively at the beginning of a project. This manifesto therefore contains a methodology for adapting to changing contexts and changing requirements throughout the development process. This manifesto advocates four values:

- **Individuals and Interactions** over processes and tools,
- **Working Software** over comprehensive documentation,
- **Customer Collaboration** over contract negotiation,
- **Responding to Change** over following a plan.

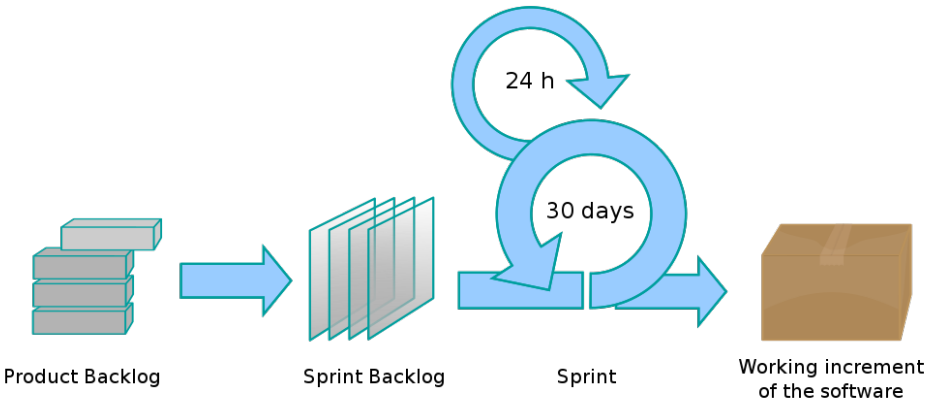


Figure 2.2: Scrum process

Our purpose was to set steps as soon as possible and then test them. After that, we adapted our future objectives according to the situation, and we repeated this plan until we obtained the final mobile application containing all the defined features. With the help of our supervisor, we therefore listed the features we wanted to obtain in order to select a portion to be completed in a given time. Once each portion of features was completed, we showed the partial but usable result to our supervisor so that he could give us feedback. This way of operating is based on one of the best known agile methods, the Scrum framework [38].

2.3.2 Scrum framework

This framework is based on the division of a project into sprints, which generally last between 2 to 4 weeks (Figure 2.2). This methodology proposes to start by listing and evaluating (on the basis of its complexity) all the features and requirements of the customer in the form of tasks to complete. It is then necessary to schedule the list according to the constraints of the developers as well as on the basis of dependencies between the elements. Once the blacklog is ready, we plan the first sprint by assigning a list of tasks to it, giving priority to the features that are most valuable to the customer. After the time allowed, the sprint ends with a demonstration of what was completed. In addition, a daily meeting is held between developers to keep track of the progress of the current sprint.

2.3.3 Crystal Clear

Another method we applied in our project was Crystal Clear [20]. It is not a precise method as such, as it does not directly propose a methodology like Scrum, but rather describes a work environment that encourages concentration and good communication between team members.

The principles of this method are largely found in the characteristics of the agile philosophy, such as iteration-based organization and the importance of frequently creating usable versions for testers. The Crystal Clear also places importance on the organization of the development team. This team can be described as follows:

– a small team gathered in the same workspace,

– osmotic communication, so that all members can share information and participate in

discussions. From this sharing debates can arise on the project that can lead to decision making,

- trust between the members,
- a reflexive improvement, i.e. a questioning, a constant search for what is not going well in order to improve the next iterations.

Chapter 3

Architecture & Implementation

The architecture of a system is probably the most important to explain. It defines how the system is organized and structured to form an ecosystem. This part was therefore very important and had to be well thought out in order to establish a robust, stable and easy to maintain architecture. In our application, the backend architecture is separated from the frontend architecture.

3.1 Backend

Our backend architecture is serverless, i.e. no server needs to be managed, maintained in the solution. As explained in the technological choices (section 2.1), this allows us to pay only when users are active. It is also easier to manage since we only worry about the functional part and not the maintenance of the servers (update, back-up, etc.)

3.1.1 Components

The backend is divided into several components, each with its own defined role:

Cognito User Pool Cognito manage user registration and connection as well as access control to applications. It works using User Pools where users register. The user is identified with a `sub` which is a UUID. Using a user name and password, they can then log in to retrieve an identity in the form of tokens [47]:

- **token id** defines the identity of the user,
- **token access** defines the accesses of the user,
- **token refresh** is used to refresh the two other tokens.

API Gateway API Gateway allows to define a RESTful API to let the client access the resources of the backend [46].

Lambda A Lambda function is a piece of code that runs in the cloud after an event triggers it. The particularity of Lambdas is that they do not run continuously: once the code has been executed, the function turns off. This is what gives the serverless characteristic of our Backend. Lambda are also stateless, i.e. they do not keep information between executions [50].

DynamoDB DynamoDB is a NoSQL database. It contains tables with a key that can be composed. A table contains items. An item is represented as a JSON document that can contain various values such as integers, lists or sets. DynamoDB follows the AP model of the CAP theorem. It means that the data obtained from the database are eventually consistent [48].

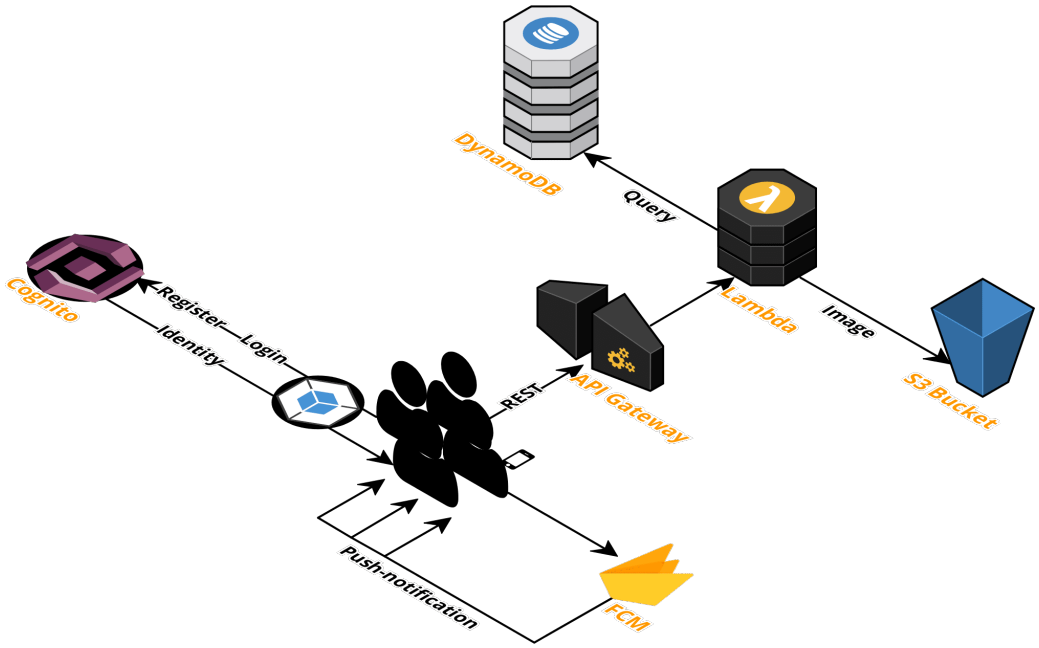


Figure 3.1: Backend architecture

S3 Simple Storage Service is a document storage service under the key-value principle. It is composed of buckets that contain keys to documents [49].

FCM Firebase Cloud Messaging is a push notification service that allows to transfer small data from client to client. A client registers to a topic to receive and send notifications. Firebase is a service provided by Google. However, the service can be used by Apple devices. When a notification is sent by a client, it is transmitted to FCM. Then FCM redistributes the notification to Android clients directly or to APNs for iOS clients. APNs then sends the notification to the iOS devices.

Following Figures 3.1 and 3.2, the first component with which the user interacts is Cognito User Pool, so that he can make an account and log in. When the user is a member of a family, the user's client can send various queries to the API managed by API Gateway. This same component redirects these queries to the Lambda service. Depending on the operation to be performed, an action will be performed either in the DynamoDB database or in an S3 bucket. If this action must be notified to members' other clients, a notification is sent to the family topic in FCM and redirected to the other clients.

The following sections explain how each component from the backend was set up and implemented.

3.1.2 Cognito User Pool

When a new user creates an account, he gives his given name, last name, email and password. These first 3 fields are the attributes of our User Pool. The password is of course not visible to us. The user will then be asked for the email and password the next time he logs in. With Cognito, it is also possible to reset the password if this one is forgotten.

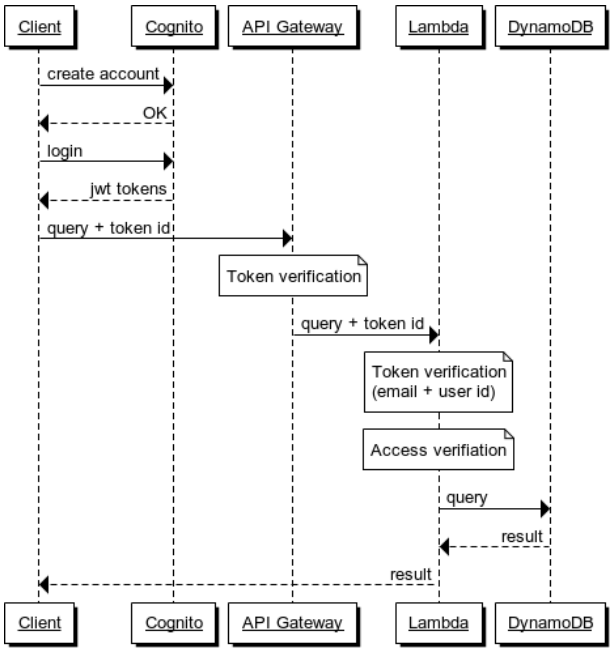


Figure 3.2: Backend query

3.1.3 API Gateway

The API Gateway allows customers to query the backend. It is the gateway between the frontend and the backend. Our API is relatively simple, it contains two routes: `/Family` and `/S3Image`. All queries pass through the first route, the second is used to retrieve images from S3. The `/Family` route returns a `Content-Type` set to `application/json`, while the second, `/S3Image`, returns a `Content-Type` set to `image/jpeg`. For each query that passes through API Gateway, the `token id` of the client is checked. The query is forwarded to the `Lambda` only if this token is valid.

3.1.4 Lambda

The lambda acts as a proxy between the Gateway API and DynamoDB. It is written in Python and is mainly useful for managing database access rights.

When a query is processed in the Lambda, it arrives in a JSON format separated into 3 attributes:

```
{
  'operation': ...,
  'parameters': ...,
  'cognito': ...
}
```

- **operation:** several types of operations are supported by Lambda, such as `update` or `put_item` which are taken over from the DynamoDB API [5]. Others such as `s3_put` and `safe_update_item` are operations complementary to the DynamoDB API. More detailed information on operations is available in the Operations section on page 24.
- **parameters:** this is a JSON object representing the operation parameters. For example, there is the `TableName` parameter that indicates in which table the operation must be

performed. The parameters are different for each operation and mostly follow the parameters defined by the DynamoDB API.

– **cognito:** encoded in base64, the value of this attribute is the `token id` of the user who sends the query. It is used for security reasons and be sure that the user has the right to make this query.

Processing

The first action performed is the verification of the provided `token id`. This token is sent to the `verify` function of the verification class. If the token is valid, the function returns a `User` object, otherwise it returns `False` and the query is dropped.

A `User` object contains 3 functions: `isAdmin` and `isParent` and `isChild`, which all take a string argument that is a family identifier and return `True` or `False`. This object is used to check whether a query can be performed or not by a user depending on his role.

This brings us to the second action that uses this `User` object. The query is analysed in details to define what type of operation will be done in the database. For this purpose, we defined a list of all the typical actions that our application is likely to send, as well as the necessary authorizations to send them (i.e. *admin*, *child*, *parent* or *none*). Typical actions in this list are actually functions that will review the parameters of the sent query. If one of the typical actions matches the sent query, the family identifier is extracted from the parameters and the `User` object is checked for authorization to perform it. Otherwise, the sent query is dropped because it is not listed in the valid actions.

After being sure that the user has the authorization to send the query, it is processed in DynamoDB with the right operation.

Operations

In most cases, an operation represents what type of call to the DynamoDB API will be sent. The API methods we use are:

– **`put_item`** creates an item in a table,

– **`get_item`** returns an item from a table based on its key,

– **`query`** returns items from a table based on an attribute of the key,

– **`update_item`** modifies an item (or adds an entry in an item list),

– **`delete_item`** deletes an item.

These methods can be related to the CRUD methods of a database. All these queries follow the syntax defined by the DynamoDB API.

In addition to these 5 operations directly using the DynamoDB API, we defined our own operations for a few specific cases. Below, we define the types of query each of these operations handle and what issues they solve or what features they add.

– **`safe_update_item`:** when a family is created, not every item related to a family is necessarily created in the other tables. For instance, if someone creates the *Littlecloud* family, an item is added in the `Family` table but not in the `LoyaltyCard` table. The item of the table `LoyaltyCard` linked to the family *Littlecloud* will only be created when a member

adds a first loyalty card. So, the `safe_update_item` operation creates the item relative to the table specified in the parameters if this item does not yet exist in the table.

- **`elastic_put_entry`:** this operation allows to add entries to the list of an item without worrying about reaching the maximum size of an item. This is especially useful for the `ChatMessages` table where items have a continuously increasing size.
- **`elastic_get_item`:** this operation goes along with the previous operation. It works like an abstraction of the `ChatMessages` table items. That is, if this operation is used to get an item, that item may be larger than the size limit of an item.

More information about these two previous operations are provided in the Resolving item size limitations section on page 26.

- **`s3_put`:** when an image is sent for storage in S3, it is sent in base64 to Lambda. This operation will add this new file in S3 and does not involve the DynamoDB database.
- **`s3_get`:** of course, it is also possible to retrieve an image from the S3. It is this operation that takes care of it. The supplied image key is used to retrieve the base64 image. Unlike other operations, this query is different in the sense that the returned data is not structured and encoded in JSON, only the base64 of the image is returned. For this type of query, the `Content-Type` is set to `image/jpeg` in the API Gateway. On the client side, this allows the image to be read more quickly.

3.1.5 DynamoDB

Family data is stored in DynamoDB as JSON documents. This component is only accessible via the Lambda which manages access. The database is composed of several tables:

Family This table contains general information about a family. For the moment, only the family name appears, but one could imagine storing the general parameters of a family. This item also contains the administrator identifiers.

Family-User This table is a many-to-many relationship representing the link between the Family table and Cognito users. It thus has a key composed of the identifier of a family and the `sub` of a user. A third attribute indicates the level of access the user has in the family (i.e. child or parent).

ChatMessages In addition to the family indentifier, there are 3 attributes in this table. First, a list containing all the messages. Each message in this list contains 4 other attributes: the author who is in fact the `sub` of the member who sent the message, the member's given name, the date and the text of the message. There can also be a fifth attribute if an image is attached to the message, in which case the image identifier in the S3 bucket is indicated. The second and third attributes of the `ChatMessages` table are `part` and `next`. These two attributes are used to extend the item if it exceeds the 400KB limit, more explanation on these two attributes are available in the section 3.1.5. The table key consists of the family identifier and the `part` attribute.

ShoppingList The key of this table is a key composed of two attributes, the family identifier and the list identifier. In this table, each item represents a list. In addition to these two attributes, there is the list name and a boolean that indicates if a list is active or if it is a template. The `products` attribute can be compared to a set except that each product can be reached via its identifier. Each product has 6 attributes: name, price, quantity, unit, an indication if the product is checked and a version number. The version number is used to keep a consistency between different clients. Further explanations are provided in the section 3.3.

LoyaltyCards Apart from its key, this table contains an attribute that contains all the cards of a family. In the same way as the products of the `ShoppingList` table, the cards can be reached via their identifier. A card is composed of 4 attributes: card name, logo url, barcode or QR code number and format. The format refers to the formats available in the ZXing library [53].

Budget This table is very similar to the previous one. It has an attribute `refunds` which all contain refund requests. A request is composed of 7 attributes: the member who posted the request, the amount, the name, a description and optionally an attribute specifying the parent who refunded the child.

The structures of the tables are provided in appendices.

UUIDs

Each table contains items which are specific to a family. DynamoDB does not provide a mechanism to auto-increment item IDs. We must therefore define by ourselves the UUID of the new items. In order to have a fairly high level of certainty on the unique side of a new identifier, it is calculated as such:

$$Current\ time\ in\ millisecond + random\ string$$

The random string follows the regex:

$$(a-z|A-Z|0-9)^8$$

An example of UUID is `1523871290047NfDL3lN8`. There is thus almost no chance that two items created in the same millisecond have the same ID (1 chance out of 218,340,105,584,896).

Concurrent operations

The concurrence of operations in DynamoDB is a major concern when several clients are allowed to modify the same item. This is especially the case for items in the `ShoppingList` table. This table represents a shopping list that contains several products. When modifying a product, it is important to be able to minimize the impact of this change to not rewrite the whole product. For this purpose, the modification is done only on the concerned attributes without rewriting the whole product.

In the case of the attribute `check` of a product, it is a little bit different. When several members of a family are at the store and complete the same shopping list, it is necessary to be able to refuse the check of a product by a member. Indeed, if this product has already been checked by someone, the member must avoid buying the product twice. To solve this problem, we use a Conditional Update. More explanation can be found in the section 3.3.2.

Resolving item size limitations

In the case of DynamoDB, one limitation caught our attention: the size of the items. Each item cannot exceed 400KB [16]. This raises a certain problem on items that extend continuously. It is the case of items that store chat room messages. To overcome this problem, we created a Python class called `ElasticDynamoDBItem` available on Github [26].

The principle of `ElasticDynamoDBItem` is simple, following Figure 3.3, on page 27: when inserting an entry in a list, if an exception is thrown because the item exceeds the size limit (step 1), a second item is created just like the first with a `messages` attribute that contains the new message (step 2). The `next` attribute of the first part is then set to `TRUE` since there is a

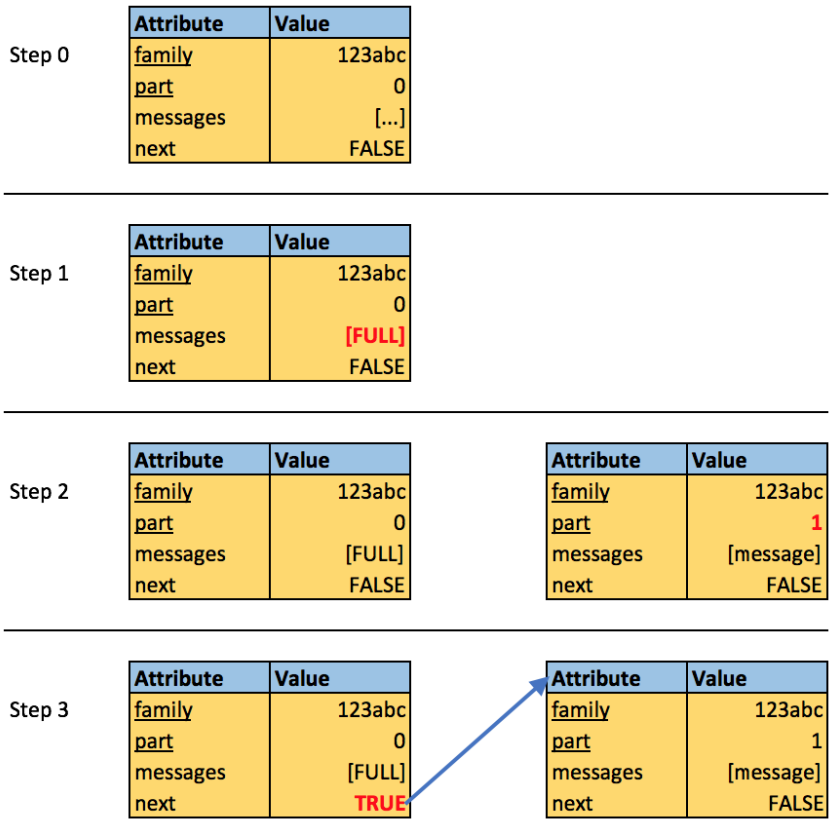


Figure 3.3: New part creation steps

new part (step 3). The process is repeated each time the list of messages in the last part is full. These new parts extending the first one are numbered to keep the right order of the lists. To get an item with the whole list, `ElasticDynamoDBItem` takes each part and merges each part of the lists. It is possible that these operations work in parallel execution. So, if two instances of the Lambda try to create a new part, one of them will only add the new entry in the new part.

Figure 3.4 shows a sequence diagram of how the messages are safely added. When a client sends a message, it uses the `elastic_put_entry` operation. The first thing the Lambda do is to search for the last part of the messages (i.e. where the `next` attribute is `False`). In the diagram, this is `part number 1`. Once this part is found, the Lambda will try to add the message to this part. From there, two events can occur:

[**success**] The message is successfully added, the Lambda sends back to the customer that everything went well.

[**new part**] `Part 1` is full, i.e. it is impossible to add the message because the item would exceed the 400KB limit. The Lambda reacts by creating a new item: `part 2`. By creating this item, it adds the message at the same time. From this request, two events can also occur:

[**success**] `Part 2` is successfully created, the Lambda then indicates that `part 1` contains a successor by setting its `next` attribute to `True`. The Lambda returns to the client that everything went well.

[**item already exist**] `Part 2` already exists. This event can occur if two clients ask at the same time to add a message to a part that is full. One of the two client creates

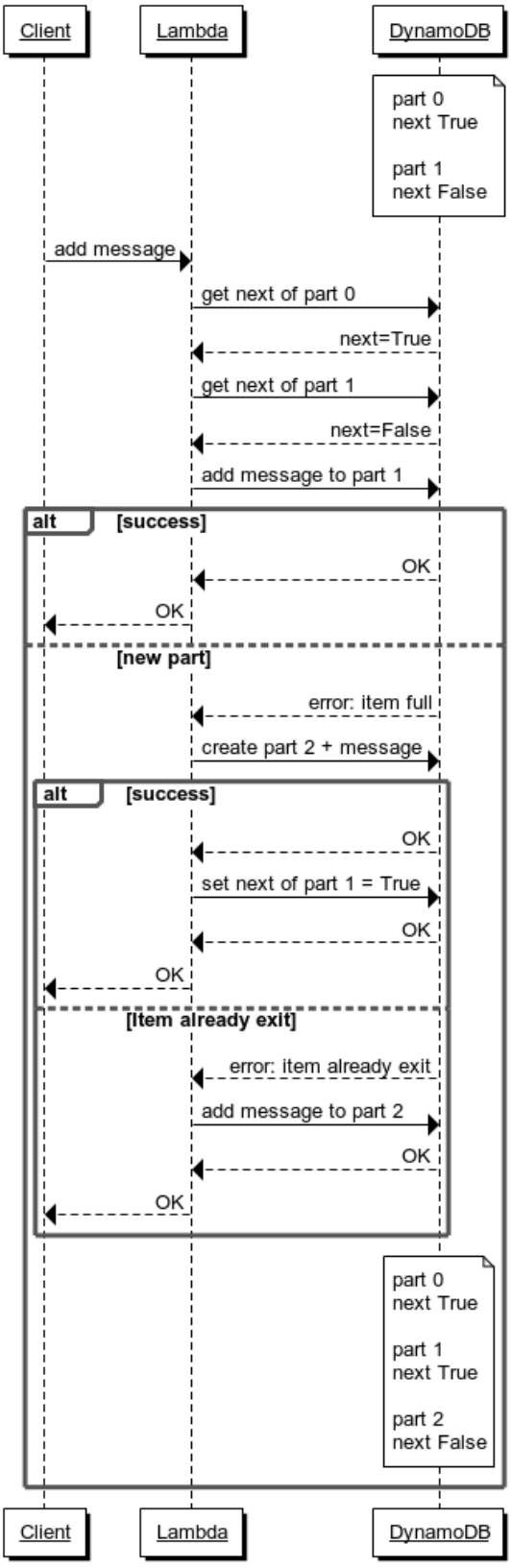


Figure 3.4: Safely adding a message

`part 2` before the other. In this case, the Lambda adds the message to the `part 2` already created. Then the Lambda returns to the client that everything went well.

3.1.6 S3

The Lambda also has access to S3 bucket to store images. There are images from the chat room and images that are profile pictures. This bucket is private and only the Lambda has a right of access. Since the Lambda defines whether a user has an access right or not, this access right alone guarantees the security of the images.

A user is able to edit his profile picture. This photo will be sent to the bucket with as key: `profile/sub` where `sub` is the user identifier provided by Cognito.

A member can share an image in the chat room of his family. In this case, the key to the image will be `family_id/images/image_id.jpeg` where `family_id` is the identifier of the family and `image_id` is a random UUID. This key is then referenced in the message sent to DynamoDB.

3.1.7 FCM

Firebase Cloud Messaging is the only backend component that is not an AWS service. It allows a client to send messages between users of our application without going through the backend. A client therefore receives information about changes made into the database, without having to request it. This also helps to counter the eventual consistency of DynamoDB that might not return the latest version.

The application uses the topic feature of FCM: a client can subscribe to a topic to receive notifications about that topic. He can also send notifications on this topic. For instance, the members of the family with the identifier `ABC123` will subscribe to the topic `ABC123`. In addition to registering to the family topic, the client also subscribes to the user's email. This topic will be used to send a notification concerning an invitation to join a family.

3.2 Frontend

As explained in the technological choices (section 2.2), we decided to use the development tool Xamarin as the frontend of our application. In this section, we will first explain how Xamarin.Android/iOS works, as well as its components. We will then make the link with the architecture of our project. Finally, we will explain how we manage the requirement of being able to use the application when offline.

3.2.1 How Xamarin works

Xamarin.Android/iOS is a cross-platform development tool that allows to create Android and iOS versions of the same application using a single language (C#). Certain aspects of the code base are platform specific (e.g. the UI layer), which we have developed repeatedly for each platform (Figure 3.5). We therefore started by creating a common code base. It contains business logic, database storage and network calls. Then, one project is created per target platform. It contains the graphical interface, navigation and components specific to each SDK. So, we can take advantage of the specificities of Android or iOS without reducing the user experience. Our final solution contains the following folders and projects:

– a shared project containing the code common to all projects,

– a Xamarin.Android application project,

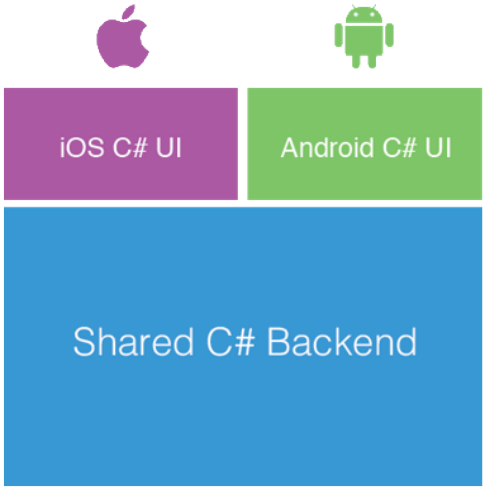


Figure 3.5: Xamarin.Android/iOS approach

– a Xamarin.iOS application project.

3.2.2 Xamarin architecture

Figure 3.6 represents the conceptual architecture, where each project includes all the shared source files. We can see six application layers [10, 11]:

Data Layer Non-volatile data persistence, such as SQLite database, XML files or any other suitable mechanism. For our project, we use a library that allows to store data locally with the key-value principle.

Data Access Layer Wrapper around the Data Layer that provides Create, Read, Update, Delete (CRUD) access to the data without exposing implementation details to the caller.

Business Layer Business entity definitions (the Model) and business logic. In our project, this layer contains shared objects, query bodies and tools used by both platforms.

Service Access Layer Network access services such as web services (e.g. REST) and simple retrieval of data from remote servers. This layer encapsulates the networking behaviour and provides a simple API to be consumed by the Application and UI layers. We use it to communicate with the AWS Gateway API, as well as for user authentication (Cognito).

Application Layer Code that is typically platform specific or code that is specific to the application (e.g. local preferences, permissions, etc.).

User Interface Layer User-facing layer that contains screens, widgets and the controllers that manage them. The vocabulary of the different components of the project differs according to the platform, but their functions remain essentially the same (Table 3.1).

Figure 3.7 represents the process of a query at the frontend level. When performing an action (e.g. get the messages), the controller asks the Query Builder to create a the body of the query by sending the action information. The Query Builder builds the body based on this information. Once the controller has received the body, he sends it to the RESTService. The RESTService then sends the query containing this body via an HTTP request. This same service will then serialize the response to create an object (e.g. a list of messages). This object will then be returned to the controller.

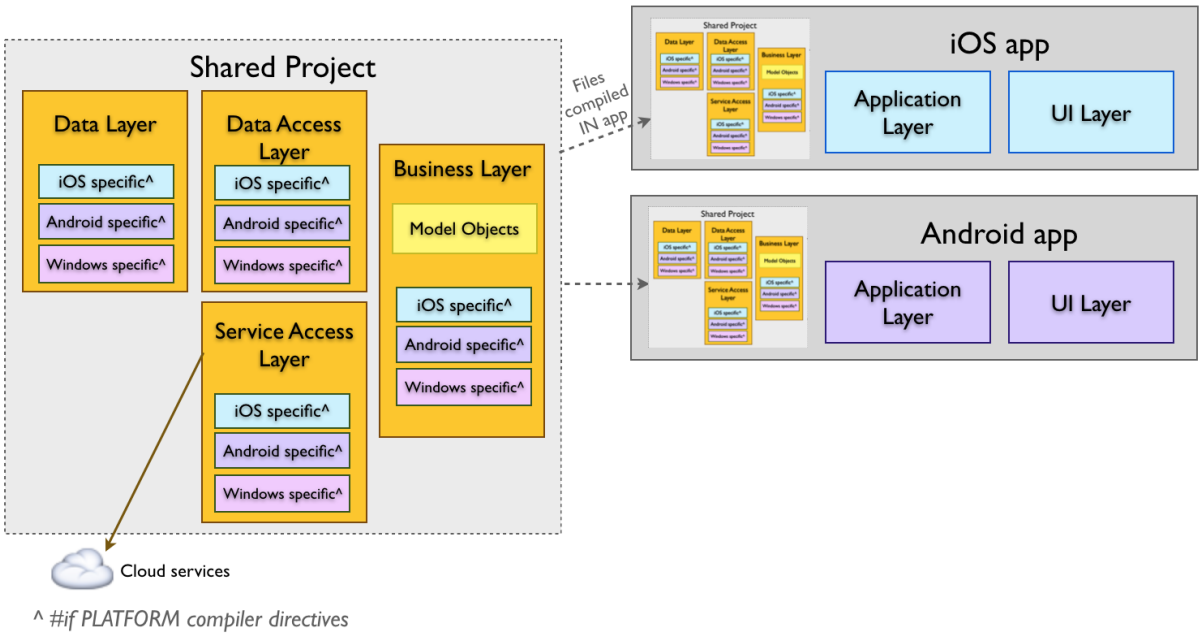


Figure 3.6: Xamarin typical application layers

Android	iOS
Layout (.axml)	UIView (.xib or .storyboard)
Activity	UIViewController
Fragment	UIView
Intent	UINavigationController

Table 3.1: Components vocabulary for Android and iOS

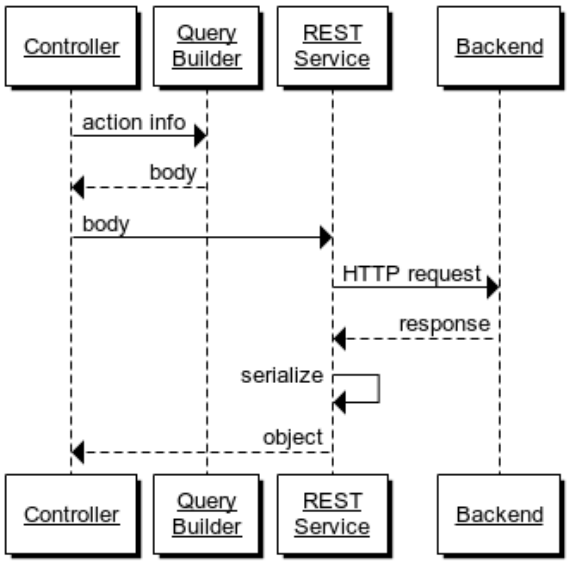


Figure 3.7: Frontend query

3.2.3 Push notification management

When a push notification reaches the user's device from the FCM service, the application must know how to send the data to the appropriate module. In our application, each platform manages this transmission in its own way.

Android

Figure 3.8 represents the process of push notifications on Android. In order to manage push notifications, we use the `LocalBroadcastManager` component [31]. This allows us to send broadcasts of Intents to local objects within our process. Each module likely to receive push notifications registers to a specific broadcast receiver. This component simply responds to broadcast messages from the FCM service. A broadcast receiver is implemented as a subclass of `BroadcastReceiver` class and overriding the `OnReceive()` method. In this method, each message is received as a Intent object parameter. Once the message is received, the appropriate operation is performed if the module is active.

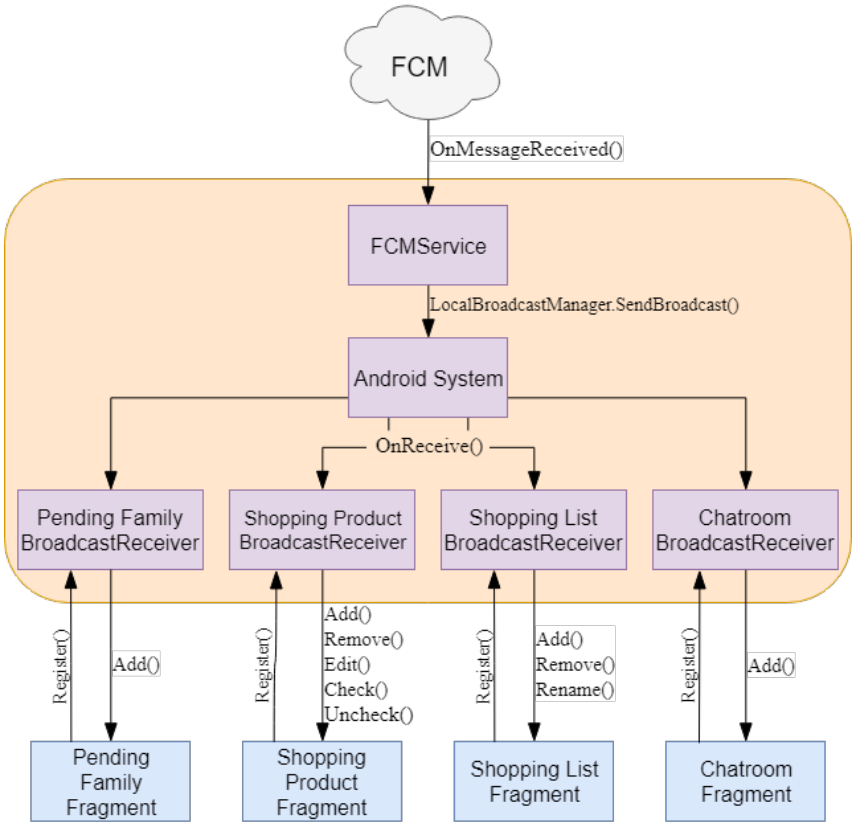


Figure 3.8: Push notifications process on Android

As the `OnReceive()` method is always called on the main thread (which is also referred to as UI thread), it is thread safe.

iOS

Figure 3.9 represents the processing of a received push notification. For example, during navigation, when the user arrives on the page containing the shopping lists, the application initiates a `ShoppingController` instance. This controller will then register to its specific Listener. When a push notifications is received, it arrive in the `DidReceiveRemoteNotification()` method

of the `AppDelegate` class. This information is then sent to the `FCMListener` class via the method `OnMessageReceived()`. This class detects which module the push notification is intended for. If it is intended for shopping lists, the `ShoppingListener` will decompose the information to detect which operation should be performed (`Add`/`Remove`/`Rename`). This operation will then be applied to the concerned controller previously subscribed.

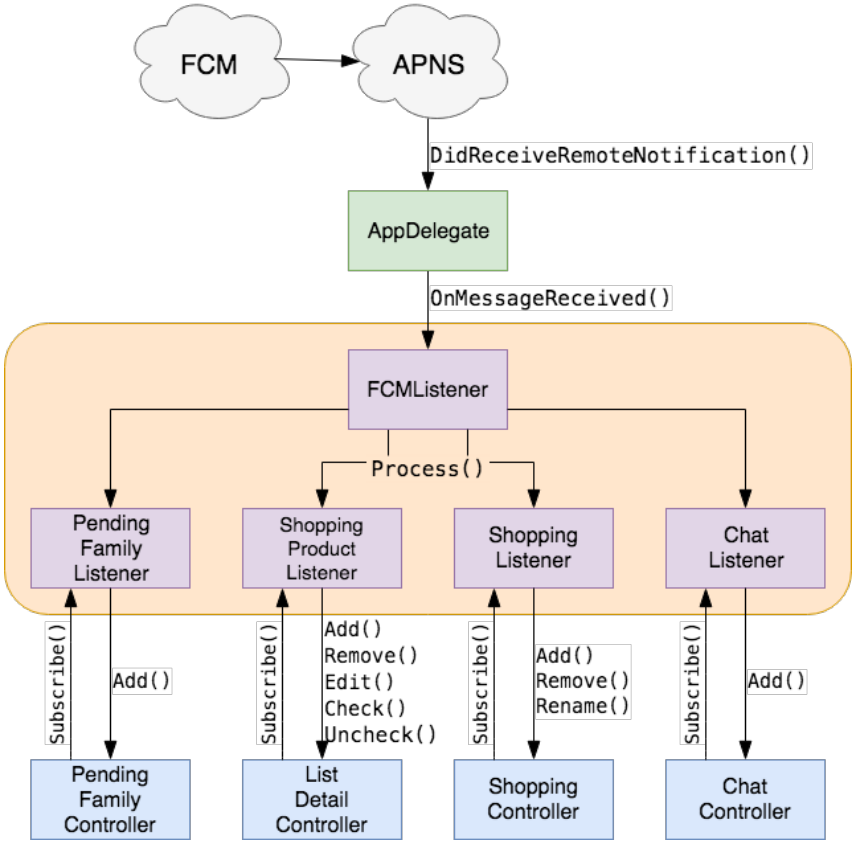


Figure 3.9: Push notifications process on iOS

Note that the process is thread safe. The `OnMessageReceived()` method of the `FCMListener` class is synchronized. This means that if several notifications arrive at the same time, they will not create conflict in the local data as they are applied one after the other.

3.2.4 Offline management

When a client loses its internet connection, queries sent to the backend fail. From this event, the application can react in 3 different ways depending on the type of request:

1. The first possible reaction from the application is to cancel the query and refuse it. The user will therefore receive an alert message indicating that his action will not be taken into account. Any modification of local data induced by this action are rolled back. Actions that are into this category are item changes that may conflict with the database and other system clients such as renaming, editing and deleting data. Note that due to the UUID, there is nearly no risk to have conflict when creating data.

2. The second category represents an exception to the first reaction concerning shopping list products. It is common to go shopping without internet connection. This is why it is possible to edit and delete a product to allow the user to shop with the application.

Action	Offline
Add message	✔
Add list	✔
Rename list	
Remove list	
Add product	✔
Edit product	✔
Check product	✔
Remove product	✔
Add refund	✔
Edit refund	
Apply refund	
Remove refund	
Add card	✔
Edit card	
Remove card	
Add family	
Edit role	
Invite	
Accept/Decline	
Quit family	

Table 3.2: Actions available offline

However, changes made will not be synchronized with other nodes (DynamoDB and other clients). They will be lost when the application restarts or if modifications of other nodes overrides them.

3. The last solution is to save the queries that could not be sent. In this case, changes are accepted locally. Queries will then be sent in order as soon as the internet connection is back. In this category are the actions that create data such as adding a new list and actions that rely on a consensus system. More explanation on these actions can be found in the following section (section 3.3).

Table 3.2 shows the actions that a user can perform offline (i.e. reactions 2 and 3).

Back online

There is a major difference between the clients of the application running on Android and those running on iOS.

When an Android client returns online, all push notifications that could not be received are resent to the instance. In this way, the instance automatically synchronizes itself [1]. In the case of an iOS client, push notifications that have not been received are forgotten. Sending a notification to an iOS instance works with the best effort principle: either the notification arrives directly or it is lost [30].

When a client returns online, it must synchronize to receive all the information it could not receive while offline. In the case of Android clients, FCM send the notifications that were

pending, there is no particular action to take. For iOS clients, synchronization must be forced by detecting when the device is online. The client will then get a copy of the data in the backend and add its locally added data to it. At the same time, the pending queries are sent to other clients via FCM and to the backend.

CAP theorem

Within a family, our system can be represented as nodes communicating with each other and sharing data. There is the backend side which combines Lambda and DynamoDB as the first node and the other nodes which are clients of the same family, for example Alice and Bob. Data is accessible via any node. It is therefore possible to define a model of our shared-data system from the CAP theorem [18]. The definition of the Bewer's **CAP** theorem is the following:

You can have at most two of these properties for any shared-data system:

- ***C**onsistency*
- ***A**vailability*
- *Tolerance to network **P**artitions*

In the case of our system, *consistency* would be the ability to have identical data between different nodes. *Availability* ensures that each node is accessible and contains data. *Tolerance to network Partitions* allows the user of a node to access data offline.

Each client contains a local version of the data. This version is updated by receiving push notifications from FCM or by pulling a more recent version of DynamoDB. This data is always available on each device. For example, Alice can read data from the application while being online or offline. The data present on the devices are therefore eventually consistent. They tend to be updated but it is possible to see outdated data. The system therefore follows an **AP** model of the CAP theorem.

The possibility to access the data offline and to have different existing versions of the data required us to apply consensus and synchronization mechanisms between clients of the same family.

3.3 Synchronization between clients

During an interaction between the frontend and the backend, not all queries are treated in the same way. Some do not need special measures and are directly accepted by the backend and sent to other clients. For instance, this is the case for queries that add objects like adding a message or a loyalty card. Others need to have an implemented consensus in order to avoid race conditions and be sure that each instance of the application is in a consistent state. This is the case for modification operations such as checking a product.

3.3.1 Without consensus

When the Lambda and DynamoDB accept the modification of a family data, this modification is also sent by the client to the other connected smartphones via a push notification. Thus, each smartphone receives this modification without having to query DynamoDB again. To send these notifications, we use the FCM service.

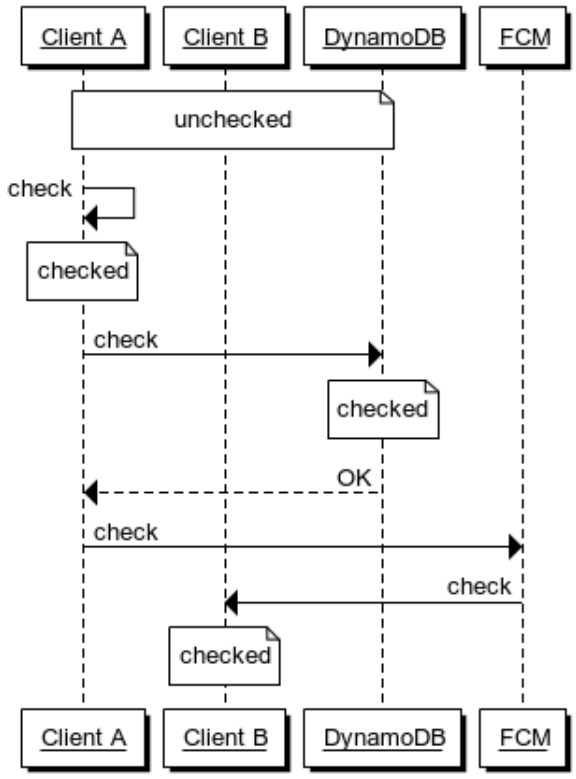


Figure 3.10: Simple check product

This kind of queries does not need consensus since a new item or a new entry added in a DynamoDB table will not override another. In these queries, we find for example the addition of a shopping list, a product or a message.

3.3.2 With consensus

When taking as an example the query which allows to check a product as illustrated in Figure 3.10, two problems can occur:

- First, imagine that Carol and David are shopping together and have to buy vegetable. Carol finds frozen vegetable while David finds them in the vegetable section. If they check the product on their shopping list at the same time, the problem is that they will possibly buy vegetable twice. One of them should receive information that the product is already checked to avoid this.
- Another problem is the arrival of FCM notifications to a smartphone. We cannot make assumptions about the order in which they arrive. Thus, in another situation where Carol checks a product and then unchecks it directly after, it is possible that David receives the check notification after the uncheck. So the state of the David's client will be inconsistent.

These two problems were solved using the `ConditionalUpdate` parameter available with the DynamoDB API and by indicating a version number on the check operations.

Conditionnal Update

To solve the problem occurring when two members check at the same time, we use DynamoDB conditional expressions. In the query, if the client sends a check for a certain product, this

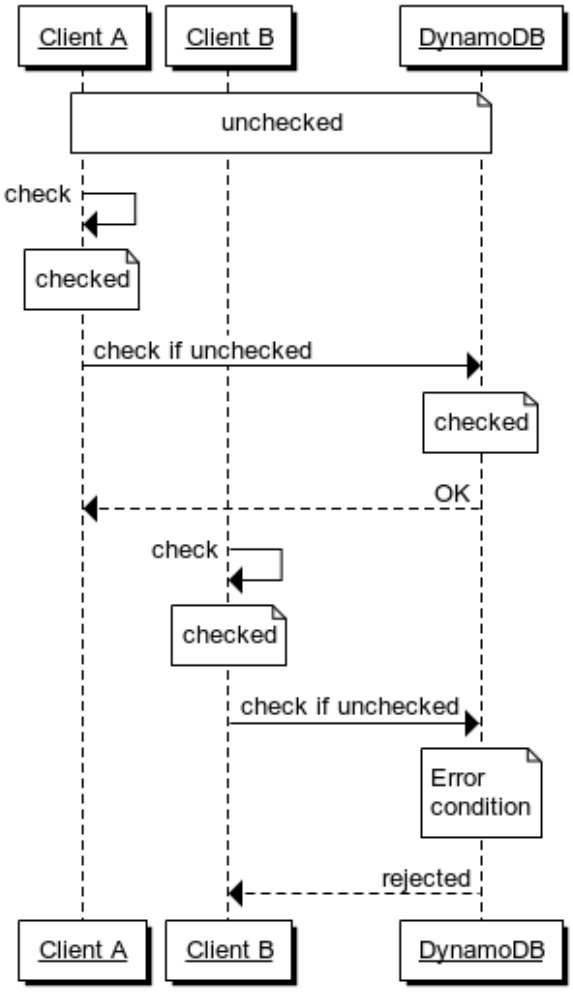


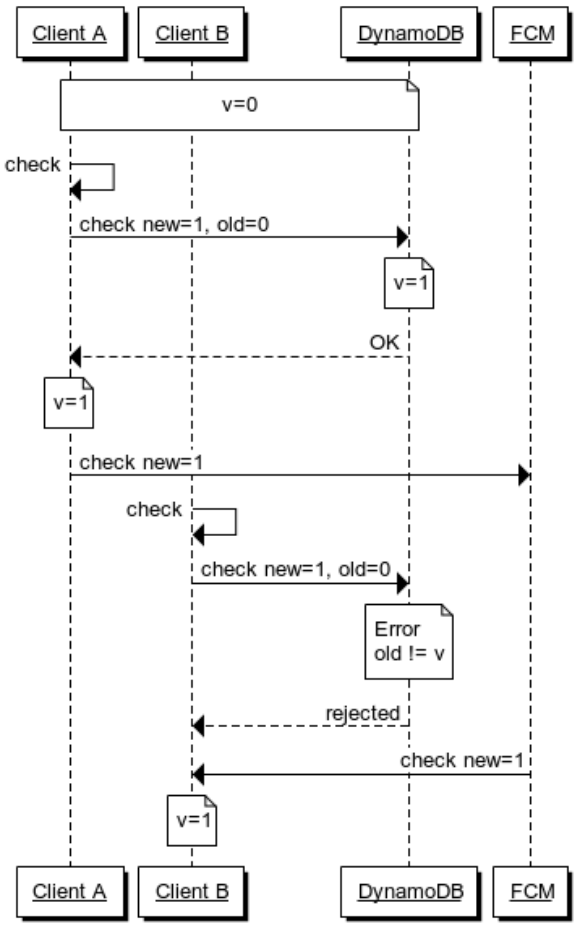
Figure 3.11: Conditional check product

product must be in the uncheck state in the database item. If this is not the case, the family member is warned that he should not buy the product because it was already checked. This condition is added as conditional expression to the DynamoDB query (Figure 3.11).

Versioning

Version numbers are used to define which version is older than another. When a push notification is received by FCM, the application will know whether or not to apply this message based on its version number.

Figure 3.12 represents the sequence diagram when several members check the same product for a short time. Following this diagram, each time a client (client A) checks a product, it indicates its local version number. DynamoDB will then accept the modification of the item only if this number is the number of the current version of the item in the database. When modifying the item, the number will then be updated by incrementing. If another client (client B) checks a product, its modification will be refused if its local version number does not match the one of the item in the database.



`v`	local version number
`new`	new version number
`old`	old version number

Figure 3.12: Conflictual check product

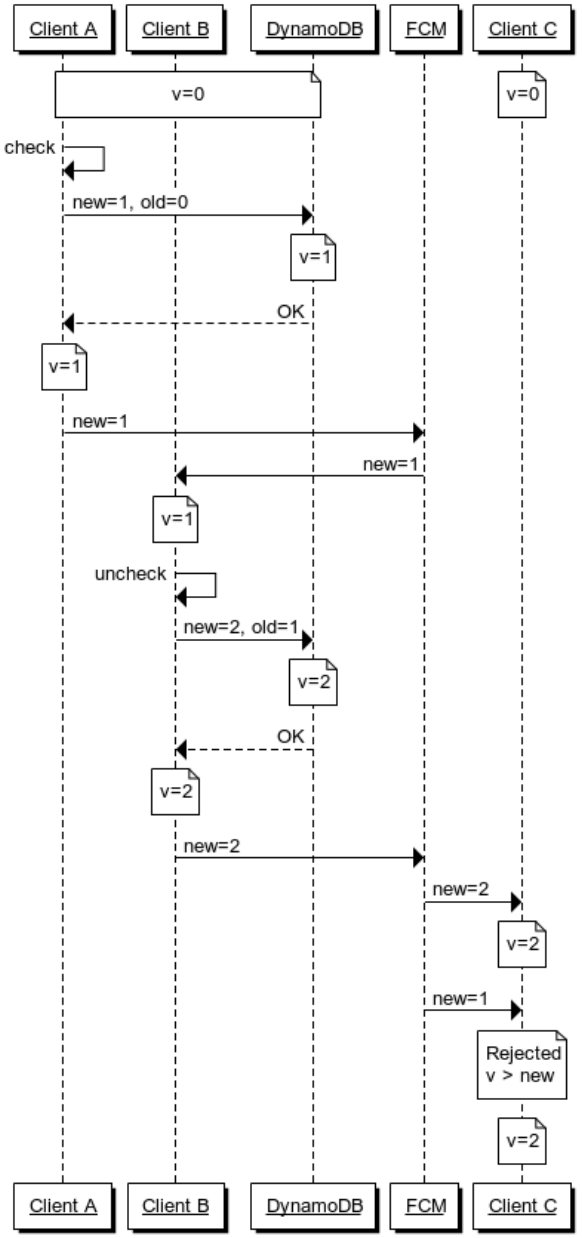


Figure 3.13: Sequential check product

In the case where several clients work on the same item, the version number increases. Following Figure 3.13, in the first two thirds of the diagram, client A and client B perform operations sequentially. The version of the product in DynamoDB arrives at 2. It is impossible to assert that a third client will receive these modifications in the right order. As shown at the end of the diagram, to overcome this uncertainty, client C only applies changes if this one is more recent (i.e. the version number received is greater than the local version number). The local version number is then updated. So if a modification arrives late, it will be rejected.

3.4 Security

Security is essential when developing applications. Users provide their data only if they trust the computer system. It is therefore necessary to process and store this data in a secure way to deserve this trust.

On the other hand, it is also our duty to limit users' actions to their permitted actions defined in the use cases. These limitations must take into account the family where the action is performed and the member's role in that family. Blocking a feature on the client side is not enough to ensure that the limits imposed on users are respected. These limits must also be verified at the backend level.

3.4.1 JSON Web Token

JSON Web Token (JWT) is an open standard (RFC 7519 [37]) that defines a way to secure a transmission of information between parties in a JSON object. This information can be verified because it is digitally signed by a third party authority [12].

In our application, when a client logs in, a request is sent to the Cognito User Pool service with its email and password. If the login information is correct, the Cognito service answers with 3 JWT [44]:

ID token This token contains all the user's information (i.e. `sub`, email, last name and given name). Its period of validity is one hour.

Access token This one is useful for access management, but we do not use it in our application.

Refresh token The last one is used to refresh all the tokens. We set its validity period at 30 days.

The client keeps these tokens in storage.

When a request is sent to the backend, it goes through a Gateway API that will redirect the request to the Lambda. The query contains the token identifying the user. A first verification is performed in the Gateway API: the request is accepted only if the token is present, valid and if it has been delivered by our Cognito User Pool. If the conditions are met, the request is sent to the Lambda. On the Lambda side, there are still accesses to verify. For example, a user can only change data in families where he is a member. We must therefore identify with certainty the user who sends the request. The information concerning his email and his identification number are encoded in the `token id` that was sent with the request. A second check of the token is therefore performed to extract these two pieces of information. In order to define if a user can perform a certain request, we specified 4 types of access in relation to a family. Administrator, member, parent and child access. For instance, an administrator can assign roles to members while a member cannot. A child can add a refund request while an adult can only refund them.

3.4.2 Images

Some modules of the application offer functionality involving retrieving images from the backend. Concerning private images of a family, it is a natural thing that these images are accessible only by family members.

In our backend, images are stored in an S3 bucket. The simplest way to allow users to retrieve images would have been to put this bucket in public and access the images via the image URI. A part of the image key is the family identifier and the other is a UUID. This makes the image URI difficult to predict. Nevertheless, since the bucket is publicly accessible, there is always a risk for an image to be referenced by a search engine.

To ensure the security of these images, we preferred to put the S3 bucket in private and to write functions allowing to get the images while passing by the Lambda. The Lambda systematically checks the identity of users who want to make a query. By reusing this security, the Lambda function can verify that the user is a member of the family whom the image belongs.

Chapter 4

Graphical User Interface

One of the objectives of the thesis was to achieve a similar graphical interface between the two platforms we are targeting. This objective was all the more challenging given that each interface had to be developed separately. Although the Xamarin.Forms framework allows to make a common interface between each OS, the fact to privilege the performances of the application directed us towards Xamarin Native.

The interface design is done in AXML for Android and with storyboards for iOS. Because of the guidelines and the different ways of conceiving the views of each OS, this first constraint quickly imposed two different ergonomics on us. The challenge was then to reconcile the guidelines of each platform with the objective of having similar views. Screenshots of the application can be found in appendices.

To help us in the design of these views, we followed the heuristics of Jacob Nielsen who offers several rules to increase the usability of an interface.

4.1 User interface guidelines

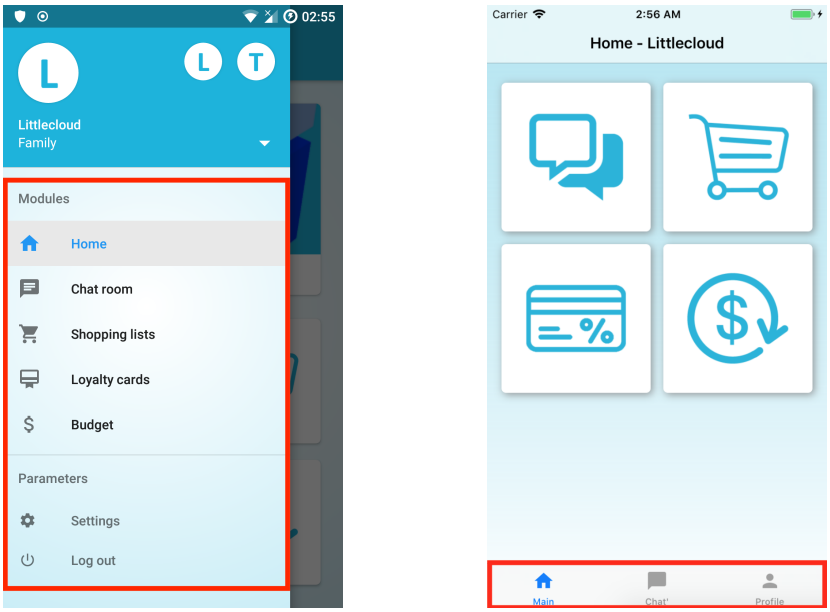
UI guidelines are a document describing all the rules and elements to design an interface. They are intended to help GUI designers to create efficient and effective interfaces in accordance with the general ergonomics of the platform. The goal of the companies that propose their guidelines (i.e. Google and Apple) is to encourage developers to create better quality applications.

For comparison, Figure 4.1 presents a guideline difference between the Android and iOS platforms. It concerns the navigation between the different views of the same level in the application. In the case of Android, the guidelines allow us to use a Navigation drawer [34]. However, in the case of iOS, the guidelines require us to use a Tab Bar [39].

4.1.1 Android

Android users expect our application to look and behave in a way that is consistent with the platform. We therefore followed material design guidelines for visual and navigation patterns. Material Design is a visual language that synthesizes the classic principles of good design with the innovation of technology and science. These guidelines are structured around five design basics [9]:

Layout Material Design layouts encourage consistency across platforms, environments, and screen sizes by using uniform elements and spacing. Our Android UIs use intuitive and



Android iOS

Figure 4.1: Menu specific views

predictable layouts, with consistent grids, keylines, and padding. Regarding responsiveness, we forced the application into portrait mode for ease of use.

Style The Material Design color system can be used to create a color theme that reflects our brand or style. Once we had chosen our brand color, we put it as the primary color theme. We reinforce our brand's style by showing brand colors at memorable moments (e.g. buttons, navigation drawer, floating buttons, etc.). We also paid attention to the legibility of the content on the screen.

Animation Motion helps make a UI expressive and easy to use. Whether it is for drawer navigation, floating buttons, swipe lists or recycler views, we were careful to make their motions show spatial and hierarchical relationships between elements, which actions are available, and what will happen if an action is taken.

Components In order to display an application's most important or timely information at a glance, we use app widgets throughout the UIs of our application. For each app widget used, we tried to follow its design guidelines.

Usability Accessible design allows users of all abilities to navigate, understand, and use our UI successfully. By designing clear layouts with distinct calls to action, we help users navigate our application. We include appropriate content labelling to give users confidence in knowing where they are in our application.

4.1.2 iOS

The iOS guidelines differ from those of Android on several points. They also influence the tools available to developers when they design views in a storyboard. For example, it is not possible to make a side menu by following the libraries and widgets natively proposed by iOS. To overcome these constraints imposed by the guidelines, it is imperative to use third-party libraries. However, this solution was quickly discarded since these libraries are sometimes difficult to use with the development of interfaces in a storyboard. The iOS guidelines are structured around six design principles [41]:

Aesthetic Integrity The appearance of an application must be integrated with the features it offers. In our case, despite the fact that many views mainly display data, we tried to create an immersive design that remains consistent with the sharing side of our application.

Consistency The ergonomics must implement standards familiar to the platform using the elements provided by the system. In this way, users must be able to recognize the application's operating principles by referring to their expectations. The iOS version of *Domus* includes many iOS principles, like long press gesture, tab bars, navigation controllers, etc.

Direct Manipulation Some actions involve an instant change in the display. In many views, the user has the ability to use gestures that involve a direct update of the screen. This is the case, for example, when displaying an image or when checking a product.

Feedback When a user performs an action, he must directly see the result of this action. Many visual feedback mechanisms are used: waiting screens for long loads, animations, etc. Overall, each action performed involves instant visual feedback on the interface. This principle implies that we had to implement a rollback system if an action was not accepted by the backend.

Metaphors This principle tends to make the actions of the digital world similar to the real world. This is the case, for example, with switches, sliders, scrolls, etc.

User Control The user is the master of the application, he must be able to perform the actions he wants. The application should not impose choices on the user and should allow the user to cancel choices if they were made by mistake. In our application, each action that risks modifying the data in an irremediable way is accompanied by a possibility to cancel the action.

4.2 Similar views

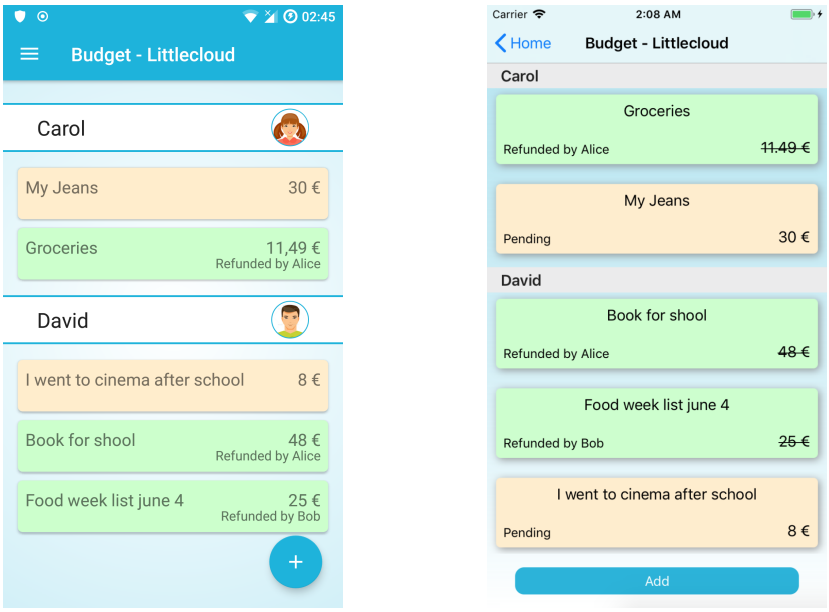
In accordance with the guidelines of both platforms, we tried to design similar views between the two platforms. The goal is to allow the user to stay familiar with the application if he moves from one platform to another. It is therefore necessary to use the same vocabulary between the different views. Despite some slight graphic differences, all views follow a similar structure.

As a tangible example, let us take the budget module of our application. This module is composed of two pages. Figure 4.2 represents the first page that displays all the refund requests. Figure 4.3 represents the page that shows the details of one of these refund requests.

4.3 Jakob Nielsen's heuristics

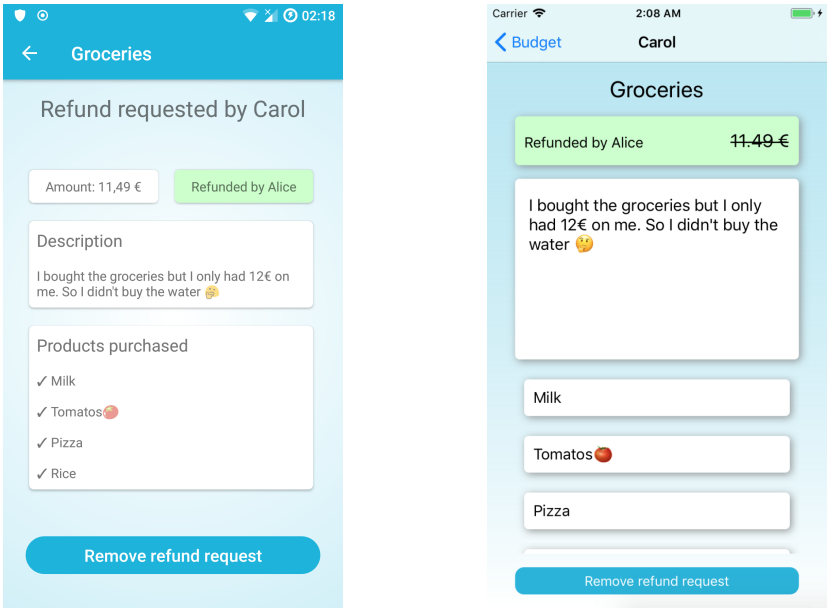
In addition to the guidlines of each platform, we decided to add *Nielsen's ten usability heuristics* to our guidelines to help us design the user interface. It is interesting to see that many of the guidelines defined by the platforms are taken from Nielsen's heuristics. These heuristics are mainly used to conduct an evaluation of the user interface.

A heuristic evaluation of a user interface helps to detect usability problems based on human factors principles. This type of evaluation involves testers reviewing the interface and its use in detail by following heuristics. These heuristics offer criteria that must be respected as well as possible to make the user interface the best it can be.



Android iOS

Figure 4.2: Budget views



Android iOS

Figure 4.3: Refund details views

As the developer of the user interface, we cannot evaluate ourselves. Our goal is therefore to make sure that this type of study gives good results. Throughout the development, we therefore tried to respect these guidelines as best we could.

According to Nielsen's book Usability Engineering, the heuristics are as follows [35]:

1. Visibility of system status
2. Match between system and the real world
3. User control and freedom
4. Consistency and standards
5. Error prevention
6. Recognition rather than recall
7. Flexibility and efficiency of use
8. Aesthetic and minimalist design
9. Help users recognize, diagnose, and recover from errors
10. Help and documentation

The following sections present the heuristics and developments that we carried out to respect them. It is of course possible that some parts of the application lack usability. Indeed, we believe that it is always possible to improve the ergonomics and ease of use of an interface. Nevertheless, we tried our best to make our application pleasant to use.

4.3.1 Visibility of system status

The user must get feedback about what is happening in the system. In our application, we paid attention to indicate the loading times when the user must be warned. The title of pages and pop-ups is also important, the user needs to know in which module and in which family he is in without having to guess it.

4.3.2 Match between system and the real world

System vocabulary should be as natural as possible and avoid system-specific technical words. Before starting the development, most of the vocabulary was already present in the use cases we had defined. The scope of our application's features relates to a use in everyday life, so there is no need to present technical words to users.

4.3.3 User control and freedom

This heuristic gives a user the possibility to cancel an action he committed by mistake. Each time content is added to a family, it is always possible to remove that content. In the case of shopping lists, it is possible to remove a product or list, this is also the case for refund requests and loyalty cards. In addition to being able to remove content, the user always has the option to go back in the application or cancel an action.

4.3.4 Consistency and standards

This heuristic imposes us two things in our application. The first is that the user interface must follow system conventions. One of the challenges of our application was to design similar interfaces for the two supported systems (i.e. Android and iOS). This difficulty was announced from the beginning when it came to creating the interface architecture. As shown in Figure 4.1, the natural way to travel between pages on the Android platform is to use a side menu. The equivalent of side menus for iOS applications are tab bars. For each platform, we chose to adopt the guidelines separately. In this way, a user of a specific OS becomes more easily familiar with the application.

This heuristic also implies that the user is not confused when moving from one platform to another. Concerning the more detailed pages of the application, we have much more freedom to design the interface. In this way, we developed similar interfaces that use the same vocabulary.

4.3.5 Error prevention

The system should anticipate errors that the user may make. It must provide good error messages and understandable instructions to correct them so that problems do not occur. In the case of our application, errors can mainly come from the data that users type. A typical example of one of our views is the minimum size a password can have: when the user types his password, a check is made on the field. If the password length is too small, the error is indicated to the user. This kind of verification has been done on all views that require the user to enter data. In this way, errors are returned at the client level and not at the backend level.

4.3.6 Recognition rather than recall

The system should make actions and options visible. Instructions for using the system must be visible when the user needs them. In the budget and shopping lists modules, we have implemented autocompletion for the fields that can benefit from it. Depending on these fields, suggestions are made based on store names, product names and units.

4.3.7 Flexibility and efficiency of use

The system should offer shortcuts for experienced users by putting accelerators that allow users to perform more frequently used actions. The current state of our UI does not contain such shortcuts. All users, whether new or experienced, use the same interface because there are no features that might seem like advanced options.

4.3.8 Aesthetic and minimalist design

The user must see the relevant information of what he is looking for. The most used information should be highlighted as opposed to the less used information which is placed in lower levels. In other words, the presentation of information should follow a hierarchy that emphasizes relevant data. The budget module represents this heuristic well. On the first page, the user will find information such as the name of the refund, the author and the amount. If the user selects a request, he will be directed to a page that contains the less important information: the description and the list of purchased products.

4.3.9 Help users recognize, diagnose, and recover from errors

Error messages are a common thing in a system. The user must be able to counter them and find a solution. Therefore, error messages must contain fairly precise information about the

nature of the error and how to resolve it.

There are two types of errors that can occur when a user performs an action: a connection error and the database that rejects an operation.

Connection error When a connection error occurs, the message *"Error, verify internet connection"* is presented to the user. His action is cancelled and he is invited to try again after making sure that he is connected to the Internet. This message may also be *"Connection error, it will be sync later"* for some actions that can be performed offline.

Database error This error occurs only when a user checks an item already checked by another member. In this case, the system informs the user that he should not buy the product twice.

4.3.10 Help and documentation

In complex systems to use, it is necessary to include documentation or a tutorial to help the user become familiar with the interface. Ideally, this documentation should be clear and not too large. Given that we tried to make an interface as instinctive as possible, our application has no tutorial or documentation. Nevertheless, it could be interesting to make a tutorial allowing the user to travel among a default family in order to let him discover the features.

Chapter 5

Testing phase

We started the testing phase once our application became a minimum viable product (MVP). An MVP is the most pared down version of a product with sufficient features to satisfy early adopters. An MVP has three key characteristics [45]:

- have enough value that users are willing to use it,
- demonstrate enough future benefit to retain early adopters,
- provide a feedback loop to guide future development.

The final set of features is only designed and developed after considering feedback from the product's initial users. In this section, we will discuss the different tools and techniques we used to gather feedback from initial users and testers. Our testing process comprises of validation, automated testing and user acceptance testing [42].

5.1 Software validation

Validation is the process of examining whether or not the software satisfies the customer requirements. Our supervisor had the role of validating the various implemented functionalities. Thus, during the various sprints, we could make sure each time that the functionalities matched the expectations. However, we will not go into details as we already pointed out this aspect in the development methodologies section (section 2.3).

5.2 Automated testing

Automated tools are used to detect failures and collect data analysis. We use the mobile application development platform Firebase [23]. Firebase is a Mobile Backend as a Service. In order to collect user feedback automatically, Firebase provides multiple products such as Google Analytics, Crash Reporting, Performance and Test Lab for Android.

5.2.1 Google Analytics

Google Analytics provides free and unlimited reporting on distinct events. The SDK automatically captures certain key events and user properties. Analytics reports help us to understand clearly how the users behave. So we can take informed decisions regarding application performance optimizations. For instance, Figure 5.1 represents the top screens when users interact with our Android version of the application. We can see on average how long the user stays on the splash screen, authentication screen or main activity screen. A possible improvement would then be to reduce the waiting time when a user wants to enter the application.

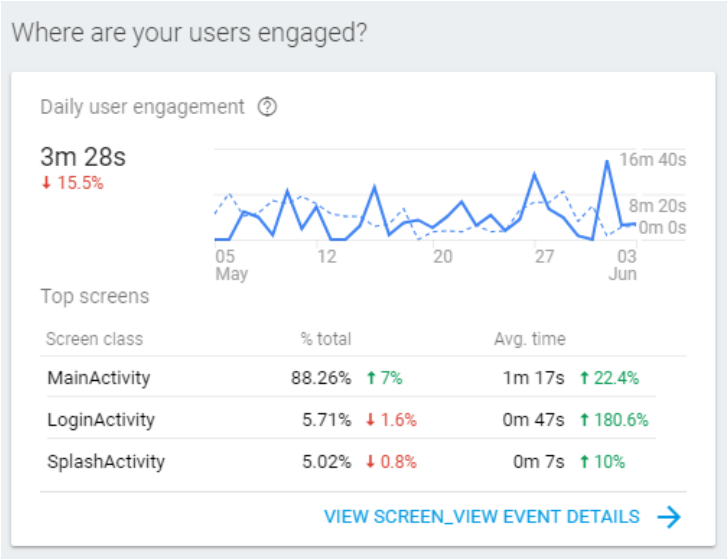


Figure 5.1: Top screens when users interact with the Android version of the application

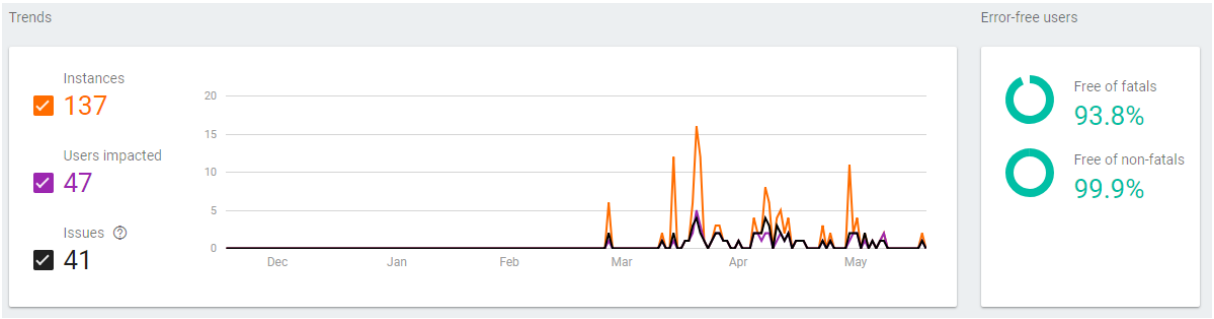


Figure 5.2: Overview of fatal issues occurred with the Android version of the application

5.2.2 Crash Reporting

Crash Reporting creates detailed reports on the errors in the application. Figure 5.2 shows an overview of the fatal issues with the Android version of the application during the last six months from June 4, 2018. Note that this overview is not representative as it shows all fatal errors, including those during debugging. Indeed, during the testing phase, we received only one error report. We were therefore able to correct this error for the following versions of the application. Crash Reporting becomes deprecated, and Crashlytics is now the primary crash reporter for Firebase. However, at the time of writing this thesis, it is not yet possible to integrate the Crashlytics SDK into a Xamarin project. That is why we still use Firebase's Crash Reporting product.

5.2.3 Performance

This service helps us to gain insight into the performance characteristics of our Android and iOS application. For instance, Figure 5.3 represents the average start up time of the Android version of our application. If this time exceeds a certain threshold, it means that there is an optimization problem to solve. Usually, thanks to this service, it is also possible to measure HTTP/S network requests. However, like Crashlytics, it is not yet possible to integrate the Firebase Performance SDK into a Xamarin project. It would have been useful for us to measure user HTTP/S network requests to the AWS Gateway API in order to be more accurate in cost

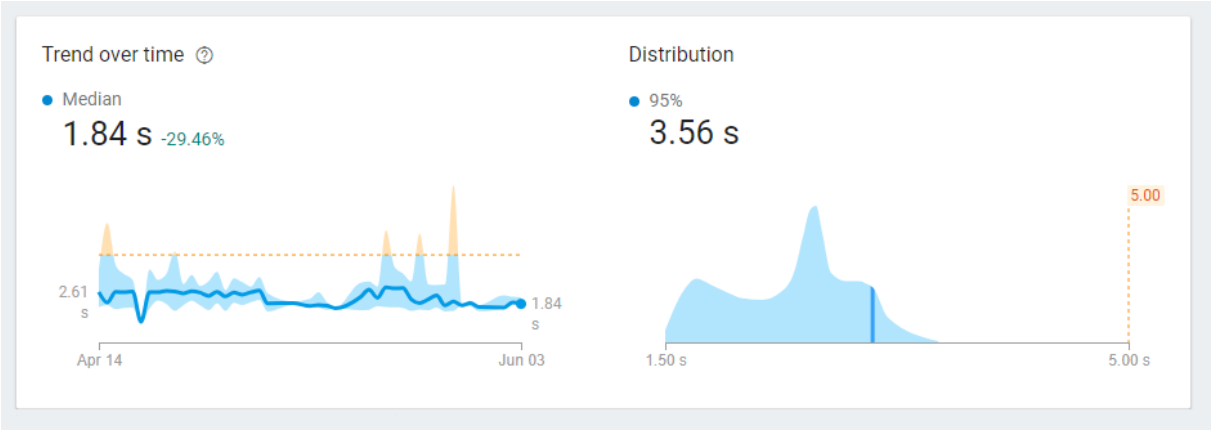


Figure 5.3: Average startup time of the Android version of the application

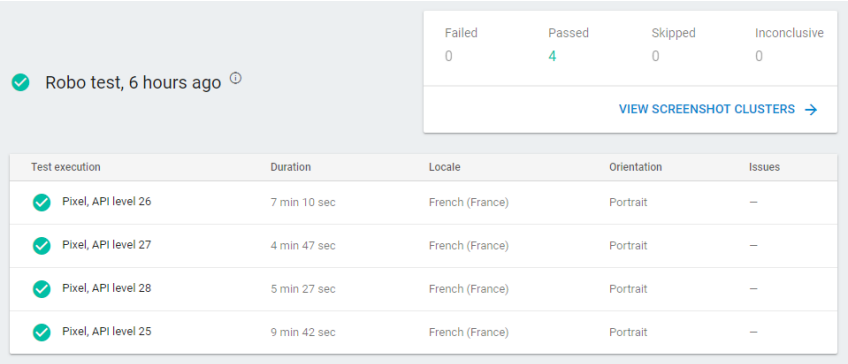


Figure 5.4: Robo tests performed on four different physical devices

calculation (section 6.2).

5.2.4 Test Lab for Android

Test Lab provides cloud-based infrastructure for testing Android applications. Physical and virtual devices are available to allow us to run tests that simulate actual usage environments. Test Lab can exercise the application automatically, looking for crashes (Figure 5.4). Test results, including logs, videos, and screenshots, are then made available in the project in the Firebase console.

5.3 User acceptance testing

User acceptance is a type of testing performed by users. Once the application become an MVP, it is tested for user-interaction and response. Acceptance testing is an important phase because we developed code according to our "own" understanding of the requirements and may not actually be what the users expect from such an application.

We proceeded directly to Beta testing without going through Pre-Alpha and Alpha testing [2]. Indeed, our product was tested by real users in a real environment. Beta version of our application was released to a limited number of users (i.e. our family members and fellow students) in order to obtain feedback on the product quality and make changes in the application accordingly. Beta testing helps reduce product failure risks and improve product quality.

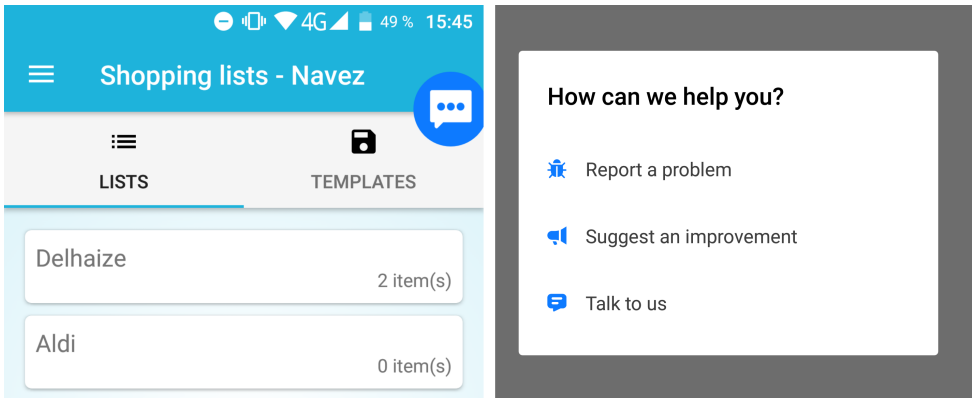


Figure 5.5: Instabug features

5.3.1 Instabug

In order to easily and automatically collect user feedback, we integrated Instabug software in our application [28]. This software provides bug reporting, in-app chats, and user surveys for mobile application. When users are testing the application, a floating button is displayed on the screen that gives access to multiple features (Figure 5.5):

Reporting a problem A screenshot is automatically captured and the users can highlight or blur parts of it. In addition, they can attach other screenshots, voice notes, or even attach a screen recording of the application.

Suggesting an improvement This functionality works in the same way as reporting a problem. A screenshot is automatically captured and the users can suggest improvements. They have the possibility to highlight a part of the screenshot in order to show us the precise place of the improvement suggestion.

Talking to us The users can directly communicate with us from our application with minimal effort providing a much better experience.

5.3.2 Test deployment

We then released the Beta Android version of the application on the Play Store to allow users easy access. In the case of the iOS version, Apple proposes a tool called TestFlight. It allows to test the use of an application among volunteers. The advantage is that it can be distributed to a limited number of users. Before making a version available to testers, the build of the version must go through the Beta App Review where a set of tests are performed by Apple. A versioning system is also available and allows to propose updates to the testers. The tester is then asked to try out the new indicated features. If a bug or a major error is detected by a tester, it is possible to invalidate a version.

5.3.3 Feedback received

We received several feedbacks from users during the testing phase. The advantage of Instabug is that we get notified instantly. We can therefore make changes in the application accordingly in the case of an error report. We collected five improvement suggestions and one error report for Android, and three improvement suggestions for iOS. Screenshots of these feedbacks can be found in appendices.

Chapter 6

Business Plan

Besides the technical aspects of an application, there is obviously the economic aspect. It is important to analyse the price that the infrastructure costs and try to detect weak points to decrease it. Moreover, any system has its limits in terms of use. Although the architecture allows us some flexibility in terms of load, AWS imposes limits that cannot be exceeded. It is necessary to plan how to deal with these limitations. These limits may also impose a maximum number of users. Making the application profitable to counter costs is also part of the business plan. The following sections review these aspects based on established hypotheses.

The calculation of costs and use of our platform is included in an excel file provided with the thesis. All the mentioned numbers are taken from this document. It is especially the case for the sections 6.3 and 6.4.

6.1 Hypotheses

In order to analyse the load of our backend, we must make several assumptions regarding the use of the application.

H1 The AWS region where the backend is running is us-west-2 (Oregon).

H2 The application is popular in Europe and the USA (From California to Croatia) and users often use the application between 10 a.m. and 6 p.m.

H3 Each user is a member of 1.5 families.

H4 There are 4 members in a family (So for 8 users, there are 3 families).

H5 75% of users have a profile picture.

H6 The size of a profile picture is 100KB.

H7 The number of times a user changes his profile picture is negligible.

H8 A user launches the application 3 times a day.

H9 Each time the application is used, the user uses each module once.

H10 Each user sends 5 messages per day in each of his families.

H11 One out of 20 messages is an image.

H12 A message has 30 characters.

H13 The size of a chat room images is 1MB.

H14 A user scrolls only up to 20 older messages per session in the chat room (this limits the load of the images).

H15 A family has 10 loyalty cards.

H16 A family constantly has 5 shopping lists.

H17 Twice a week, within a family, one list is added and another removed.

H18 Each shopping list contains 20 products.

H19 Each product is modified once.

H20 Each product is checked twice (whether the check is accepted or not).

H21 The minimum provisioned capacity of DynamoDB tables is manually set at one quarter of the necessary capacity during the period of use and each table has the same settings.

6.2 Costs calculation

Our backend works on the AWS principle of "pay only for what you use". It is therefore necessary to evaluate these costs and their evolution according to the growth in the number of users. Note that the FCM service is free, so there is no estimation for this component of our backend.

6.2.1 Cognito User Pool

For Cognito, AWS charges only the number of monthly active users (called MAUs). A user is considered active if he logged in at least once during the month. The number of MAUs will therefore be equal to the number of users. The price of the MAUs is decreasing but is free for the first 50,000 MAUs, then the price starts at $0.0055 per MAU per month [4].

Pricing Tier (MAUs)	Price per MAU
First 50,000	Free
Next 50,000	$0.00550
Next 900,000	$0.00460
Next 9,000,000	$0.00325
Greater than 10,000,000	$0.00250

Table 6.1: Cognito pricing

6.2.2 API Gateway

Concerning API Gateway, there are two prices to calculate: the number of calls to the API and the amount of outgoing data (the ingoing data is free). Since queries are all different from each other, we do not use the API Gateway cache and so do not have caching fees. The number of calls will be calculated taking into account the number of users and any assumptions involving requests to the API. The price is $3.5 per million calls [3].

Pricing Tier (TB)	Price per GB
First 10	$0.09
Next 40	$0.085
Next 100	$0.07
Next 350	$0.05

Table 6.2: API Gateway data transfer pricing

6.2.3 DynamoDB

The cost of using DynamoDB is calculated with 4 metrics: WCU, RCU, Indexed Data Storage and outgoing data transfer [6]:

WCU A writing capacity unit allows to write once per second for items up to 1KB in size.

$$WCU = \frac{w}{t} * \lceil sw \rceil$$

where:

$$\begin{array}{lcl} w & = & \textit{number of item writes,} \\ t & = & \textit{time in second,} \\ sw & = & \textit{size of an item in 1KB blocks.} \end{array}$$

The elasticity of the WCUs works so that only 70% of the capacity units provided are used. The price of one WCU is $0.00065.

RCU One unit of capacity corresponds to one strongly consistent read per second (or two eventually consistent reads per second) of items with a maximum size of 4KB. Since our queries finally use consistency, the formula can be written as follows:

$$RCU = \frac{\frac{r}{t} * \left\lceil \left(\frac{sr}{4}\right) \right\rceil}{2}$$

where:

$$\begin{array}{lcl} r & = & \textit{number of item reads,} \\ t & = & \textit{time in second,} \\ sr & = & \textit{size of an item in 1KB blocks.} \end{array}$$

As well as the WCU, the number of RCU provisioned will scale up if the number of RCU in use exceeds 70%. The price of one RCU is $0.00013.

Indexed Data Storage This metric represents the size that the data occupies in the database. To calculate this value, it will be necessary to estimate the average size of the items in each table. A particular attention must be given on the `ChatMessages` table where the items evolve by constantly growing. Indeed, the price calculated for the second month must take into account the size of the messages that were sent the first month. In addition to this, the average size added to an item per month is not the size on which the price will be calculated. This addition is cumulated throughout the month. The paid size for the first month is therefore this amount divided by two. The storage price is $0.25 per GB per month with the first 25 GB free.

Data transfer A data transfer cost must also be taken into account for data outgoing from DynamoDB.

Pricing Tier	Price per GB
Up to 1 GB	Free
Next 9.999 TB	$0.09
Next 40 TB	$0.085
Next 100 TB	$0.07
Over 150 TB	$0.05

Table 6.3: DynamoDB data transfer out pricing

Since our application is not used uniformly around the world, it is necessary to take into account the elasticity of the load to estimate the cost of WCUs and RCUs [7].

6.2.4 Lambda

In the case of Lambda, we benefit from 1 million free requests and 400,000 GB-seconds of compute time per month. This give us 3,200,000 seconds since our Lambda is 128MB.

As the data transfer price is the same as for EC2, the transfer between API Gateway and Lambda are free of charge.

The metrics to take into account are the number of RAM memory allocated, the execution time, and the number of requests. Without counting the free seconds, the price in dollar of an execution is calculated according to the following formula:

$$\lceil \frac{time\ in\ ms}{100} \rceil * \$2.08 \times 10^{-7}$$

There is also a cost of $\$2 \times 10^{-7}$ per request [15].

6.2.5 Simple Storage Service

The cost of using an S3 bucket is calculated according to the number of data present in the bucket and the quantity of data transferred during requests to the bucket. To calculate the storage price in S3, it is therefore necessary to take into account the insertion of images into the bucket. In the same way as the `ChatMessages` table, the amount of storage used by the images is constantly increasing. It is therefore also necessary to pay attention to the average space used and the cost of storage of previous months.

The storage price of a given GB is $0.023 per GB for the first 50 TB and decrease constantly. The two types of requests we use are PUT and GET which cost respectively $0.005 and $0.0004 per 1,000 requests [19].

Pricing Tier (TB)	Price per GB
First 50	$0.023
Next 450	$0.022
Over 500	$0.021

Table 6.4: S3 standard storage pricing

6.2.6 Data transfers

The paid data transfers to be taken into account are the outgoing data from API Gateway and the outgoing data from DynamoDB. The quantity of outgoing data from DynamoDB must take into account all queries that get DynamoDB table items. About the outgoing data from API Gateway, the size of queries that get images from the S3 bucket must be added to this amount. Although the calculation is different, the price of these two services are similar and start each at $0.09 per GB (Tables 6.2 and 6.3).

6.2.7 Cost distribution

The cost calculation of our infrastructure for a given month has the particularity that it must take into account the data created in previous months. These data have costs that may be related to residual costs. Some services will therefore become more and more expensive. For instance, the cost distribution of a set of 100,000 users for the first month is not the same as the costs for the 12th and 24th months.

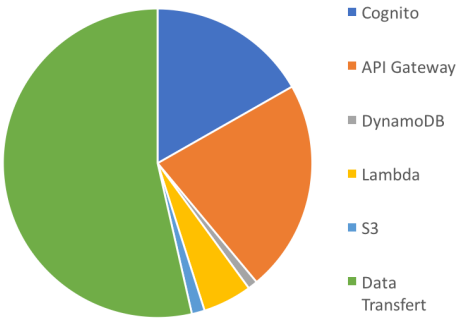


Figure 6.1: Cost distribution for the first month

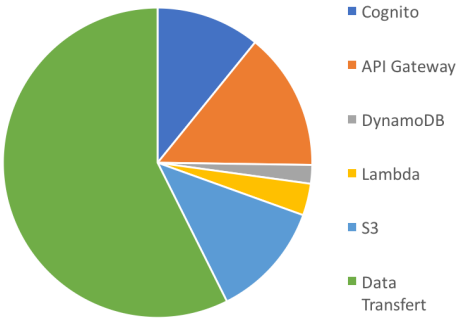


Figure 6.2: Cost distribution for the 12th month

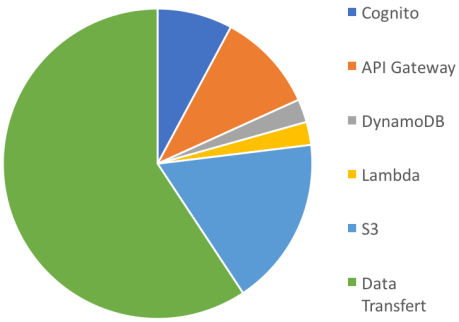


Figure 6.3: Cost distribution for the 24th month

In these three graphs (Figures 6.1, 6.2 and 6.3), we can observe the evolution of the cost distribution in relation to services over a two-year period. There are three services that seem to be strongly influenced by these residual costs: data transfers, data storage in S3 and the DynamoDB database. This problem is a serious disadvantage in our infrastructure and must be taken into account for future versions.

6.3 Limitations

When using an AWS component, it is necessary to pay attention to the limitations. This section presents the different components of our architecture, highlighting the limitations that

restrict the expansion of our application. The limitations shown are the default limitations for an AWS account. However, it is always possible to increase a limit by requesting it to AWS [16].

6.3.1 API Gateway

The only API Gateway limit that restricts our application traffic is the limit of 10,000 requests per second per region. Following the hypotheses in the section 6.1, this directly restricts the number of users to 25,000,000. A solution to overcome it would be to create other APIs in other regions. We would then be limited by the number of available regions.

6.3.2 Lambda

A region cannot run more than 1000 concurrent executions per region. To evaluate the impact of this limit, we need to know the average duration of an execution of our Lambda function. It is therefore necessary to calculate the weighted average of the execution times of each type of query according to the number of times they are called. From this, it is possible to calculate the number of sequential execution using the number of requests per second. To reach the limit, the number of users should be 7,750,000. Since the number of users is the most limited in the Lambda, it is the component that imposes the user limit of our application.

The solution could be to duplicate the lambda function in several regions. However, it is also possible to try to reduce the execution time of some time-consuming queries:

Get messages Every time a member retrieves the messages of the chat room of a family, all of them are sent back. To reduce the execution time of this request, the Lambda could only return one or two parts of these messages (section 3.1.5).

Upload an image To send an image, requests must go through the Lambda to verify the accesses. Currently, the images size are limited to a resolution of 1000×1000. Limiting the size of the images sent even more could be a solution, but it would decrease the quality of the application. By reconsidering the way accesses are managed, it would be possible to assign credentials to a client. This one will then be allowed to send an image directly into an S3 bucket. It will be necessary to use Cognito Identity Pools to distribute these accesses. This way, no more images will pass through the Lambda and the verification process will be made with a policy in S3.

6.3.3 DynamoDB

Items are limited to 400 KB. Although this limit is handled in the `ChatMessages` table, other tables still have their items limited by this size. This is the case for the `ShoppingList` and `LoyaltyCards` items, which are limited to 5700 products and 2200 cards respectively. As these limits are very high, we do not consider it necessary to surpass them. However, by applying the same principle as for the `ChatMessages` table, it is possible to surpass this limit.

Another more restrictive limit imposes a maximum number of capacity units per region (i.e. RCU and WCU). This limit is set at 20,000 RCU and 20,000 WCU. The RCU limit would be reached with 20 million users. Several solutions are possible to overcome this limit:

- migrate the tables to the Virginia region (us-east-1) which is the only region with a larger limit,
- decrease the number of items got in the tables (for example with the `ChatMessages` table items),
- distribute the tables in several regions with the DynamoDB Global Table functionality.

6.3.4 SNS

The reason we do not use SNS is because of its limitations. Like FCM, SNS also offers a functionality that allows a client to subscribe to topics (section 3.1.7). The problem is that the number of these topics is limited to 100,000 per account. If we used SNS, we would have been stuck at 70,000 users with no possibility of breaking this limit.

6.3.5 FCM

A strange limitation appeared during our tests. It seems that there is a flood limitation for iOS devices regarding push notifications. This limitation is not referenced in either the FCM or APNs documentation. When FCM sends a notification to an iOS device, it goes through APNs. APNs then sends the notification to the device. If notifications are sent at a frequency of at least 1 message per second for 10 to 20 seconds, the device will no longer receive notifications for one to two hours. Since this limitation only appears during intensive and abnormal activity, this problem is not considered critical.

6.4 Simulations

To make it easier to calculate the price of our backend, we included in the excel file a table that gives the cost of a month according to the evolution of the number of users. It is also possible to calculate the cost over several months to observe the impact of the continuously increasing size of the stored data.

Of course, the aim is also to observe the evolution of costs when launching the application up to a larger user base. For this purpose, we imagined a function defining an increasing number of users:

- the initial number of users is fixed at 100 at the beginning of the first month,
- during the first 12 months, the number of users increases at a rate of $x\%$ per month. This rate is defined according to the type of growth: pessimistic, neutral or optimistic,
- for the following months, the growth is constant and corresponds to the number of additional users during the 12th month.

In Table 6.5, we calculated the cost over a period of two years. The estimated prices are based on the above assumptions (section 6.1). The null, pessimistic, neutral and optimistic simulations have respectively a growth rate set at 0, 5, 10 and 20 % per month for the first 12 months. These rates are deliberately strongly separated in order to mark the difference between the simulations.

Simulation		1st month	12th month	24th month	Total for 2 years
Null	$	$1.91	$2.72	$3.68	$66.40
	👤	100	100	100	
Pessimistic	$	$1.97	$4.44	$9.53	$120.56
	👤	105	180	282	
Neutral	$	$2.03	$7.43	$21.35	$222.48
	👤	110	314	656	
Optimistic	$	$2.16	$20.09	$85.87	$728.98
	👤	120	892	2675	

Table 6.5: Cost and users per month

It is possible to calculate the average price that a user costs per month by observing Table 6.5. For a given month, simply divide the price by the number of users. Table 6.6 shows the cost progression of each user according to the month.

Simulation	$ / 👤	1st month	12th month	24th month	Difference
Null	$ / 👤	.0191	.0272	.0368	+93%
Pessimistic	$ / 👤	.0188	.0247	.0338	+80%
Neutral	$ / 👤	.0185	.0237	.0325	+76%
Optimistic	$ / 👤	.0180	.0225	.0321	+78%

Table 6.6: Cost per user per month

Unfortunately, the cost of a user is getting more and more expensive every month. This increasing price is explained by the fact that more user data is stored over the months. Of course, these data have a cost.

The simulations performed only take into account the cost of the platform. In an IT project, it is of course necessary to count how much the human resources mobilized to develop the project cost. Out of the total costs that composes a project, the price of the platform is probably one of the lowest.

For the application to be viable, revenues would have to grow at least as much as costs. The following section reviews the different alternatives for generating revenue with a mobile application.

6.5 Profit

Since the infrastructure of our application has a cost, it is important to define how to offset these costs by creating benefits. There are several ways to generate profit with an application.

6.5.1 Advertising

Adding ads in an application is a common solution to try to make an application profitable. The principle is to display ads to the user when using the application. Most of the time, they appear in banners and are quite distinct from the UI of the application but it is also possible to integrate them into the design of the application.

Developers must integrate a service called a supply-side platform that provides ads in the application. The revenue generated is calculated based on the number of display and the number of clicks users make on ads [33].

It is difficult to predict how much income advertising generates. Nevertheless, it is possible to estimate it from certain parameters: the number of daily active users (DAU), the click through rate (CTR) and the amount of money received per thousand of display (eCPM) [43].

6.5.2 Paid app

Paid applications are less and less popular [21]. They allow the user to directly enjoy all the features of the application by paying only once. This type of income is not so pertinent to our application model. Indeed, if a family wants to use the application, each member would have to buy it.

6.5.3 In-app purchases

In-app purchases are another way to make an application cost-effective. This model allows a user to unlock paid content from the application.

In our case, these contents can be modules. For example, if a family administrator pays for the shared calendar module, this module will be available for all family members. It is therefore a possible solution.

6.5.4 Premium

An application can also have different subscription types that can be renewed (freemium, premium, etc.). These different levels of subscriptions give access to features increasingly unlocked. It is an economic model often used because it helps to retain users.

In the case of our application, a family would have access to basic functionality for free. If this family wants to increase the number of simultaneous shopping lists, it will have to subscribe to a premium subscription to unlock this feature.

6.5.5 Targeted advertising

Users of the application provide a lot of information about their consumption habits and the stores they visit. This information can be used to offer personalized content to families. Several types of services could be created:

- product proposals in families shopping lists,
- indication of promotions available in stores,
- proposal of a store according to the products on the shopping list.

To implement these features, retailers must be allowed to provide information about products and promotions. A specific API or interface should therefore be set up for stores wishing to promote their products.

Since profiling would be done on user data, it is of course necessary to inform the user and to comply with GDPR in this domain.

In this model, the benefits do not come from the users but from the stores. Indeed, they would have to pay a certain amount to be able to enter their information into our application. This way, the application remains completely free with additional services.

6.5.6 Partnership

Another alternative for making a profit is to partner with a store brand. The user would be able to select items from the store without having to retype them. Like the previous solution, the financing of the application would then come from the brands associated with the application. However, the drawback is that the user would then be limited to only the stores present on the application. Here again, an API will have to be set up to allow products to be added.

Chapter 7

Future improvements

The development of an application or an IT project is never really finished. For an application to remain effective and follow market trends, it is essential to continuously maintain it. That is why, at the end of this thesis, we propose several improvement suggestions about functionalities and technical aspects.

7.1 Features

During the proposal of our thesis, we stated several modules that could be implemented in our application. These modules fit in well with our existing project and have the function of sharing elements that a family could have in common:

Shared calendar This module would merge the agendas between the different members of the family. For example, a goal would be to find a moment when all members are free in order to organize an event.

Shared storage For the moment, the only files shared between a family are the images exchanged via the chat room. It would be nice to allow members to share multiple file types and organize them into folders.

Directory contact Family members often share identical contacts like the hairdresser or the doctor. This module would share these contacts so that everyone could access them.

Distribution of household chores The distribution of household chores is often a source of conflict within a family. Why not propose defined schedules for each member and thus avoid a source of disputes.

Geolocation Location sharing could also have its place in the application. For example, a child could share his position to allow his parents to pick him up.

Of course, to stay in the perspective of having an application close to an ecosystem, special attention should be paid to the integration of the modules as a whole.

In addition to the integration of these new modules, other features could be added to increase the quality and scope of the application:

Identity providers An improvement would be the possibility to connect via an identity provider such as Google or Facebook.

Circles Another idea would be to increase the scope of the application by allowing roommates to use it. A user would no longer create a family account but a circle that would be either for a family or for roommates. A circle would then have a type defined as family or roommates. Features within circles would be adapted to the type of circle.

7.2 Technical aspects

During the development of the application, several technical problems were identified. For future versions of the application, some technical improvements will have to be made to make it more sustainable.

7.2.1 Reduce storage costs

The cost analysis showed us that the storage price in S3 was increasing more and more each month. This increase is due to the growing number of images. The images of an inactive family could be stored in a less expensive service. For example, Amazon Glacier allows to store files at a lower cost but with a longer retrieval time. Another solution would be to limit the number of images per family and automatically delete older images.

7.2.2 Reduce the load

The backend load influences the costs but also the maximum number of users. It is necessary to reduce it in order to obtain a durable application. We identified two operations that tend to increase this load: getting an image and getting all the messages of a family.

In the first case, the use of thumbnails would allow the backend to only return a lighter version of the image. It is only if the user wants to see the image in large that he will get the complete image. Another solution would be to store these images locally and get them from the device's local storage.

In the case of messages, one proposed solution was to return only the last messages. Older messages would only be sent if requested by the user.

7.2.3 Improve client-to-client communication

In our application, push notifications are the weak point of iOS devices. Indeed, if the device is disconnected for a certain period of time, it cannot be guaranteed that this device will receive all the notifications intended for it. To overcome this problem, certain approaches were mentioned as the creation of a "home-made" service to enable customers to communicate with each other.

Conclusion

We now arrive at the end of this thesis which focused on the full-stack development of an application. We can already confidently say that the objectives were achieved. One of the objectives was to build a scalable application that could maintain a low user load as well as a higher load. Although there are certain limitations, this objective is mainly achieved in the backend which is the critical point of the elasticity of the load. The second major goal was to create a cross-platform application available for Android and iOS from scratch. All features of our application are available for both platforms. The objective of making a cross-platform application is therefore achieved.

Throughout this software development project, we mobilized skills learned during our studies such as cloud computing, development methodology, human-computer interaction and many others. These skills allowed us to have a better quality application.

We were able to explore all aspects of creating such an application through the Agile methodology that accompanied us along the development of this project. Before embarking on the development of the application and its various functionalities, it was of course necessary to go through the classic stages of building an application which are the definition of use cases, the specification of requirements and the choice of technologies and architecture. These first steps are undoubtedly the most important as they will guide the entire further development process.

This thesis allowed us to learn new technologies, like the C# language with the .NET framework and the Xamarin tool. We already had the opportunity to develop Android applications during our studies, we were therefore able to put this knowledge to good use and reinforce it. However, iOS application development was brand new for us. Learning all these technologies takes a considerable amount of time, which must of course be taken into account when preparing the thesis. Nevertheless, we always learn better by doing.

Although defined at the beginning of the project, the construction of the backend architecture followed us with each iteration. For each implemented functionality, it had to be ensured that it did not jeopardize the elasticity of the backend. For example, we detected a weak point in the storage of messages that we solved by bypassing a DynamoDB limit. Moreover, being able to communicate devices of family members consistently with each other was also a challenge that we solved by applying consensus algorithms. Then, data security is a major concern when creating applications. It is even more true now that the GDPR has come into effect. Since the first sprint, we had to face it when authenticating users. Each request checks user accesses to ensure that no malicious actions are executed in the backend.

Our application contains four implemented modules. Each module is likely to be able to interact with other modules. In order to be able to integrate new functionalities, special attention had to be given to the maintenance of the application. We didn't want to arrive at a spaghetti code that would have cost us a lot of time when adding features. The application architecture that comes with the Xamarin framework helped us structure our code and meet this need.

One of the application's challenges was to make a similar interface between the two platforms, i.e. Android and iOS. Indeed, the use of Xamarin Native framework does not allow us to create the same interface for these two OS. Finally, for the general aspect, each application respects the guidelines imposed by its own platform. Nevertheless, more freedom is given for more detailed views. We therefore managed to define a fairly similar interface between both platforms.

Once our application had sufficient features to satisfy early adopters, we started the testing phase. The objective of this phase was to obtain as much feedback from initial users as possible. Thanks to relevant feedback, we were able to make changes in the application accordingly. This phase also allowed us to learn some testing processes and tools such as Firebase console and Instabug. These helped us reduce application failure risks and improve product quality.

At the end of the development we took a moment to step back and analyse the economic point of view of our application. The analysis of the backend load and costs allowed us to detect some critical points. In parallel with the limitations imposed by AWS, we had to find ways to improve future versions. Different approaches to make the application profitable were also suggested.

In conclusion, this thesis allowed us to get our hands on a variety of software tools and framework. We have also become familiar with mobile application development and cloud infrastructure. Moreover, this thesis was a good opportunity to use a variety of skills learned during our studies.

Bibliography

[1] *About FCM Messages | Firebase*. May 8, 2018. URL: https://firebase.google.com/docs/cloud-messaging/concept-options (visited on 05/24/2018).

[2] *Alpha Testing Vs Beta Testing*. June 8, 2018. URL: https://www.guru99.com/alpha-beta-testing-demystified.html (visited on 05/28/2018).

[3] *Amazon API Gateway – Pricing*. May 9, 2018. URL: https://aws.amazon.com/api-gateway/pricing/ (visited on 05/14/2018).

[4] *Amazon Cognito - Pricing*. May 9, 2018. URL: https://aws.amazon.com/cognito/pricing/ (visited on 05/14/2018).

[5] *Amazon DynamoDB - Amazon DynamoDB*. May 10, 2018. URL: https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_Operations_Amazon_DynamoDB.html (visited on 05/14/2018).

[6] *Amazon DynamoDB FAQs – Amazon Web Services (AWS)*. May 9, 2018. URL: https://aws.amazon.com/dynamodb/faqs/#What_is_a_readwrite_capacity_unit (visited on 05/14/2018).

[7] *Amazon DynamoDB Pricing – Amazon Web Services (AWS)*. May 9, 2018. URL: https://aws.amazon.com/dynamodb/pricing/ (visited on 05/14/2018).

[8] *An introduction to Vertical, Horizontal and Transversal social network*. May 14, 2018. URL: https://fr.slideshare.net/nicolagreco/an-introduction-to-vertical-horizontal-and-transversal-social-network (visited on 05/14/2018).

[9] *Android Developers*. Apr. 25, 2018. URL: https://developer.android.com/design/ (visited on 05/17/2018).

[10] asb3993. *Part 2 - Architecture - Xamarin | Microsoft Docs*. June 6, 2018. URL: https://docs.microsoft.com/en-us/xamarin/cross-platform/app-fundamentals/building-cross-platform-applications/architecture (visited on 05/26/2018).

[11] asb3993. *Sharing Code Overview - Xamarin | Microsoft Docs*. June 6, 2018. URL: https://docs.microsoft.com/en-us/xamarin/cross-platform/app-fundamentals/code-sharing (visited on 05/26/2018).

[12] auth0.com. *JSON Web Tokens - jwt.io*. May 22, 2018. URL: https://jwt.io/ (visited on 05/31/2018).

[13] *AWS Cloud Pricing Principles – Amazon Web Services (AWS)*. May 23, 2018. URL: https://aws.amazon.com/pricing/ (visited on 05/28/2018).

[14] *AWS Developer Forums: SimpleDB future? ...* May 28, 2018. URL: https://forums.aws.amazon.com/message.jspa?messageID=386897 (visited on 05/28/2018).

[15] *AWS Lambda – Pricing*. May 9, 2018. URL: https://aws.amazon.com/lambda/pricing/ (visited on 05/14/2018).

[16] *AWS Service Limits - Amazon Web Services*. May 12, 2018. URL: https://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html (visited on 05/14/2018).

[17] Kent Beck et al. “Manifesto for agile software development”. In: (2001).

[18] Eric A Brewer. “Towards robust distributed systems”. In: *PODC*. Vol. 7. 2000.

[19] *Cloud Storage Pricing | S3 Pricing by Region | Amazon Simple Storage Service*. May 9, 2018. URL: https://aws.amazon.com/s3/pricing/ (visited on 05/14/2018).

[20] Alistair Cockburn. *Crystal Clear: A Human-Powered Methodology for Small Teams: A Human-Powered Methodology for Small Teams*. Pearson Education, 2004.

[21] *Distribution of free and paid Android apps 2018 | Statistic*. May 30, 2018. URL: https://www.statista.com/statistics/266211/distribution-of-free-and-paid-android-apps/ (visited on 05/30/2018).

[22] Drifty. *Ionic Documentation*. June 6, 2018. URL: https://ionicframework.com/docs/ (visited on 05/25/2018).

[23] *Firebase*. May 8, 2018. URL: https://firebase.google.com/ (visited on 05/28/2018).

[24] *Five Reasons Cloud Computing Is Key To Business Success » Data Center Knowledge | Data Center Knowledge*. May 13, 2018. URL: http://www.datacenterknowledge.com/archives/2012/06/25/5-reasons-cloud-computing-is-key-to-businesss-success (visited on 05/14/2018).

[25] *Getting Started · React Native*. May 30, 2018. URL: https://facebook.github.io/react-native/docs/getting-started.html (visited on 05/25/2018).

[26] *gg0512/ElasticDynamoDBItem: A simple way to surpass the limitation of 400KB items in DynamoDB*. June 8, 2018. URL: https://github.com/gg0512/ElasticDynamoDBItem (visited on 06/08/2018).

[27] iAdmin. *Benefits of Cross-Platform Development - iTexico*. June 8, 2018. URL: https://itexico.com/blog/benefits-of-cross-platform-development (visited on 06/08/2018).

[28] *In-App Feedback & Bug reporting tool for iOS & Android mobile apps | Instabug*. June 9, 2018. URL: https://instabug.com/ (visited on 05/28/2018).

[29] Khairunj. *Xamarin Documentation - Xamarin | Microsoft Docs*. June 7, 2018. URL: https://docs.microsoft.com/en-us/xamarin/ (visited on 05/25/2018).

[30] *Local and Remote Notification Programming Guide: APNs Overview*. May 24, 2018. URL: https://developer.apple.com/library/content/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/APNSOverview.html#//apple_ref/doc/uid/TP40008194-CH8-SW1 (visited on 05/24/2018).

[31] *LocalBroadcastManager | Android Developers*. May 25, 2018. URL: https://developer.android.com/reference/kotlin/androidx/localbroadcastmanager/content/Local-%5C%5CBroadcastManager (visited on 06/01/2018).

[32] mgmclemore. *Application Package Size - Xamarin | Microsoft Docs*. June 6, 2018. URL: https://docs.microsoft.com/en-us/xamarin/android/deploy-test/app-package-size (visited on 05/25/2018).

[33] *Mobile App Advertising: Everything You Need to Know*. May 29, 2018. URL: https://splitmetrics.com/blog/ultimate-guide-to-app-advertising/ (visited on 05/29/2018).

[34] *Navigation drawer - Material Design*. May 8, 2018. URL: https://material.io/design/components/navigation-drawer.html (visited on 06/03/2018).

[35] Jakob Nielsen. “Usability Engineering”. In: Elsevier, 1994, pp. 115–148. ISBN: 0-12-518406-9.

[36] *Pros and Cons of Xamarin vs Native Mobile Development*. June 9, 2018. URL: https://www.altexsoft.com/blog/mobile/pros-and-cons-of-xamarin-vs-native/ (visited on 05/25/2018).

[37] *RFC 7519 - JSON Web Token (JWT)*. May 27, 2018. URL: https://tools.ietf.org/html/rfc7519 (visited on 05/31/2018).

[38] Ken Schwaber and Mike Beedle. *Agile software development with Scrum*. Vol. 1. Prentice Hall Upper Saddle River, 2002.

[39] *Tab Bars - Bars - iOS Human Interface Guidelines*. June 3, 2018. URL: https://developer.apple.com/ios/human-interface-guidelines/bars/tab-bars/ (visited on 06/03/2018).

[40] *The general data protection regulation - Consilium*. June 7, 2018. URL: http://www.consilium.europa.eu/fr/policies/data-protection-reform/data-protection-regulation/ (visited on 06/07/2018).

[41] *Themes - Overview - iOS Human Interface Guidelines*. June 3, 2018. URL: https://developer.apple.com/ios/human-interface-guidelines/overview/themes/ (visited on 06/03/2018).

[42] tutorialspoint.com. *Software Testing Overview*. June 8, 2018. URL: https://www.tutorialspoint.com/software_engineering/software_testing_overview.htm (visited on 05/28/2018).

[43] *Understanding App Monetization with Google AdMob - The Economic Times*. May 30, 2018. URL: https://economictimes.indiatimes.com/small-biz/marketing-branding/marketing/understanding-app-monetization-with-google-admob/articleshow/57217761.cms (visited on 05/30/2018).

[44] *Using Tokens with User Pools - Amazon Cognito*. May 9, 2018. URL: https://docs.aws.amazon.com/cognito/latest/developerguide/amazon-cognito-user-pools-using-tokens-with-identity-providers.html (visited on 05/14/2018).

[45] *What is a Minimum Viable Product (MVP)? - Definition from Techopedia*. June 9, 2018. URL: https://www.techopedia.com/definition/27809/minimum-viable-product-mvp (visited on 05/27/2018).

[46] *What Is Amazon API Gateway? - Amazon API Gateway*. May 11, 2018. URL: https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html (visited on 05/14/2018).

[47] *What Is Amazon Cognito? - Amazon Cognito*. May 9, 2018. URL: https://docs.aws.amazon.com/cognito/latest/developerguide/what-is-amazon-cognito.html (visited on 05/14/2018).

[48] *What Is Amazon DynamoDB? - Amazon DynamoDB*. May 11, 2018. URL: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html (visited on 05/14/2018).

[49] *What Is Amazon S3? - Amazon Simple Storage Service*. May 11, 2018. URL: https://docs.aws.amazon.com/AmazonS3/latest/dev/Welcome.html (visited on 05/14/2018).

[50] *What Is AWS Lambda? - AWS Lambda*. May 11, 2018. URL: https://docs.aws.amazon.com/lambda/latest/dg/welcome.html (visited on 05/14/2018).

[51] *Xamarin vs Ionic vs React Native: differences under the hood*. Apr. 25, 2018. URL: https://cruxlab.com/blog/reactnative-vs-xamarin/ (visited on 05/14/2018).

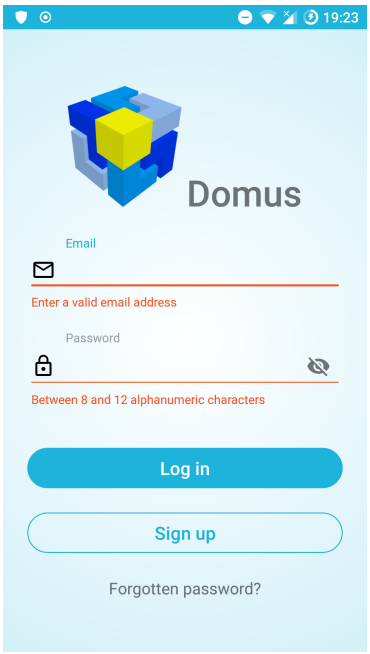
[52] *Xamarin vs React Native vs Ionic: Cross-platform Mobile Frameworks Comparison*. May 14, 2018. URL: https://www.altexsoft.com/blog/engineering/xamarin-vs-react-native-vs-ionic-cross-platform-mobile-frameworks-comparison/ (visited on 05/14/2018).

[53] *zxing/zxing: ZXing ("Zebra Crossing") barcode scanning library for Java, Android*. May 25, 2018. URL: https://github.com/zxing/zxing (visited on 05/25/2018).

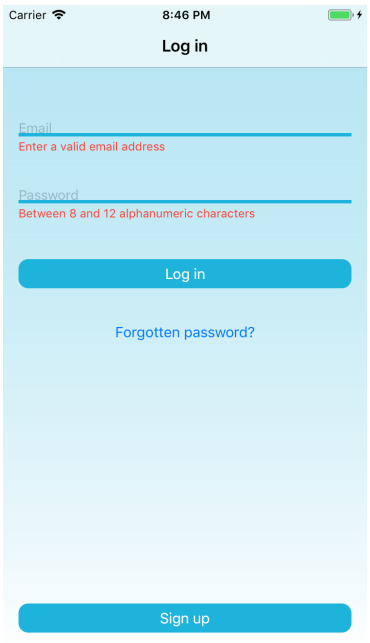
Appendices

Android & iOS views

Authentication



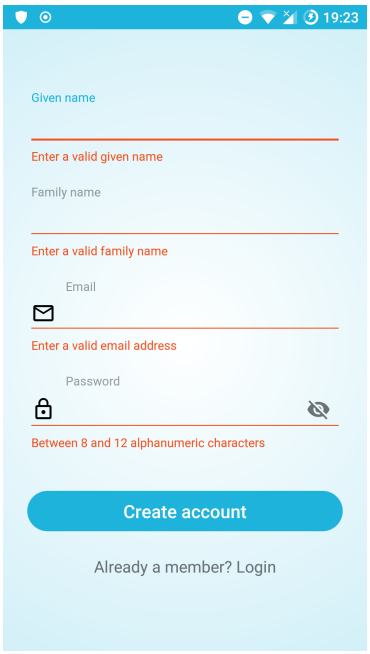
Android



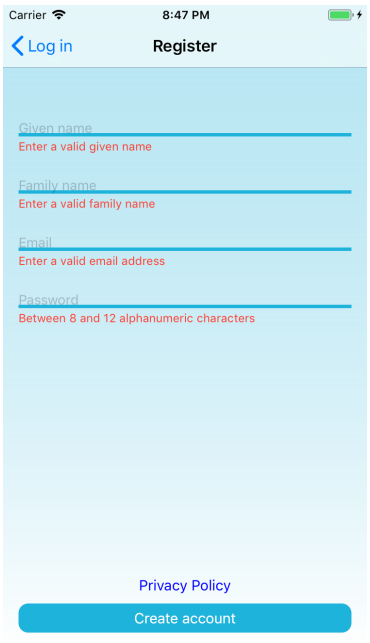
iOS

Authentication

Registration

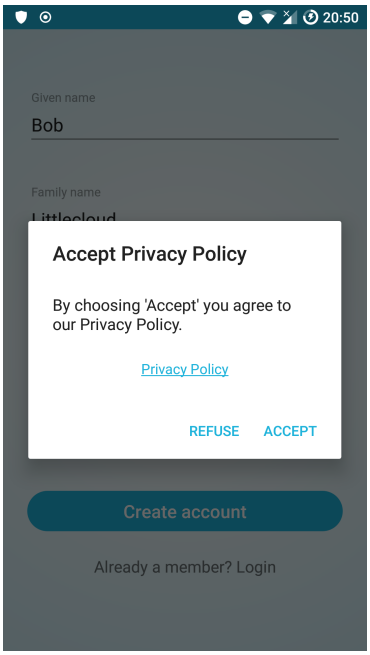


Android

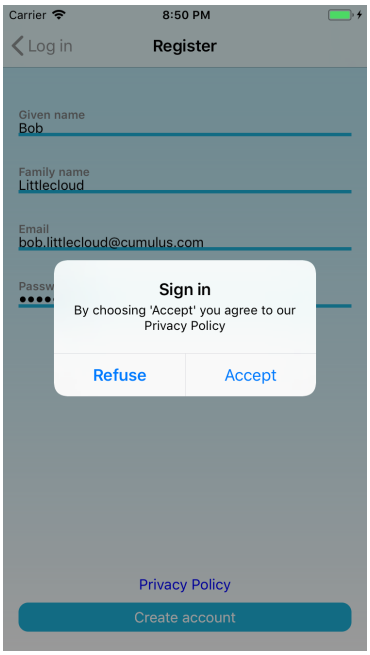


iOS

Registration



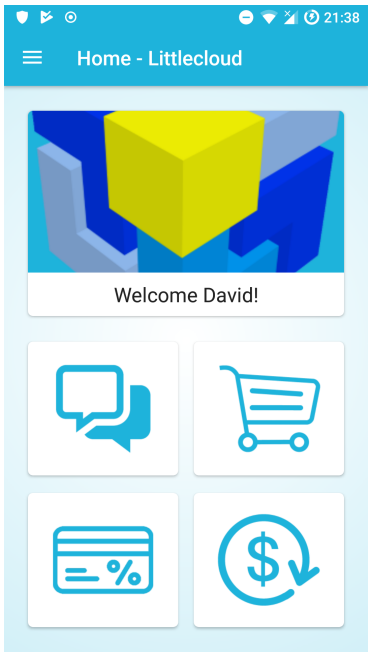
Android



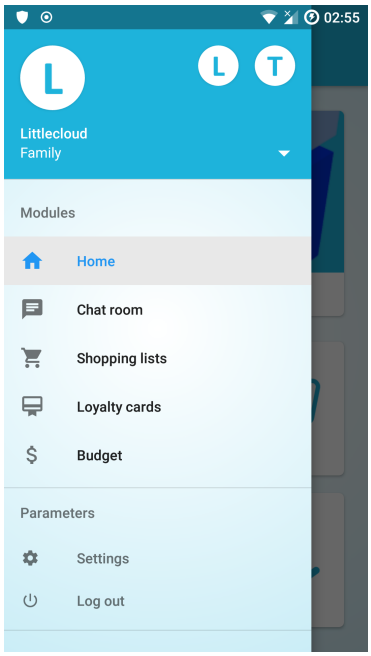
iOS

Privacy Policy

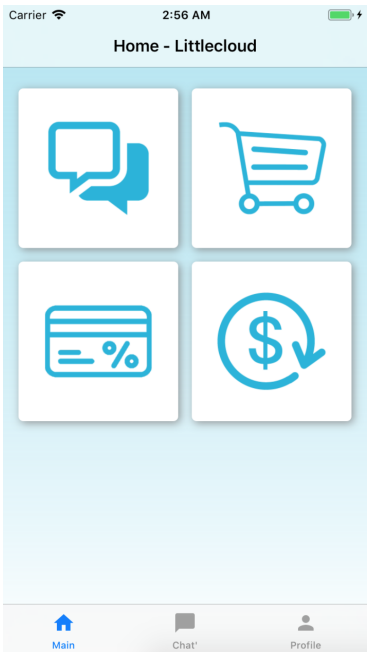
Home



Android



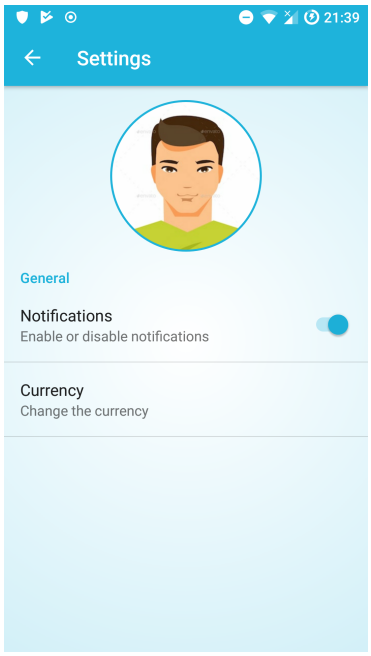
Android



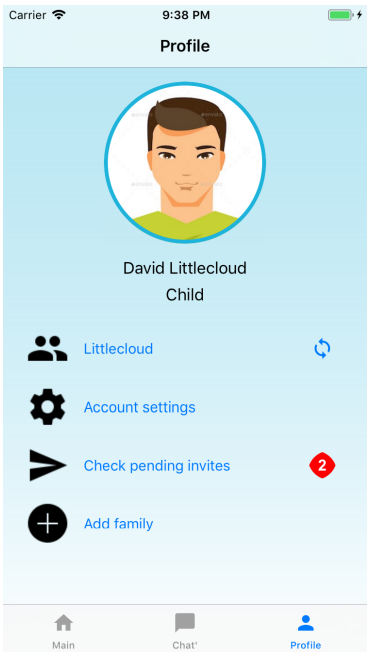
iOS

Navigation

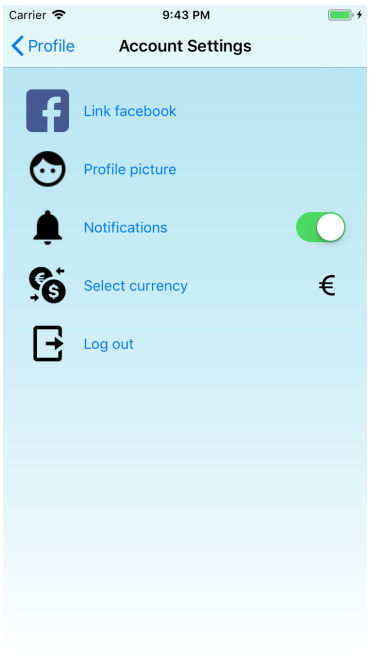
Profile/Settings



Android



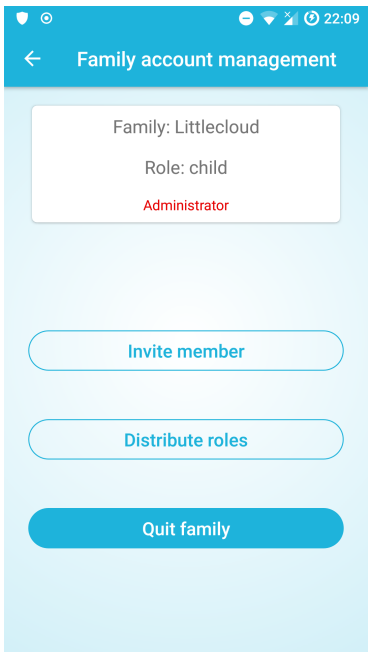
iOS



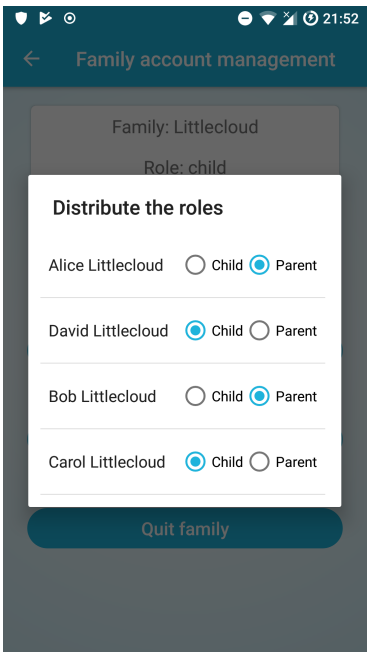
iOS

Profile & settings

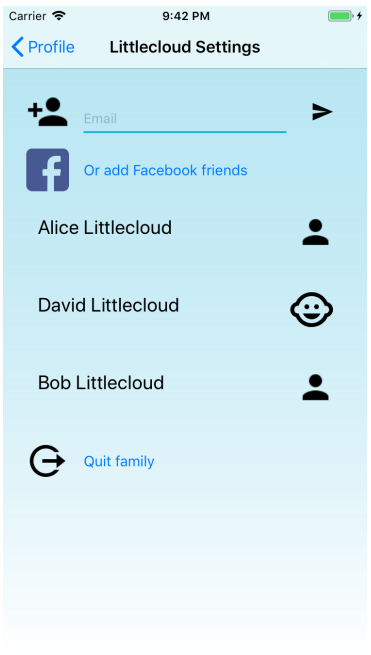
Family management



Android



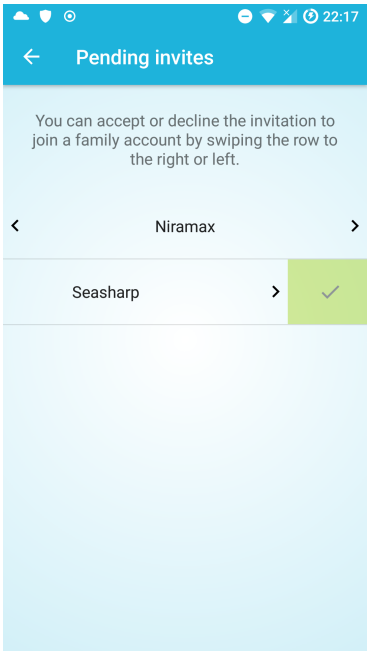
Android

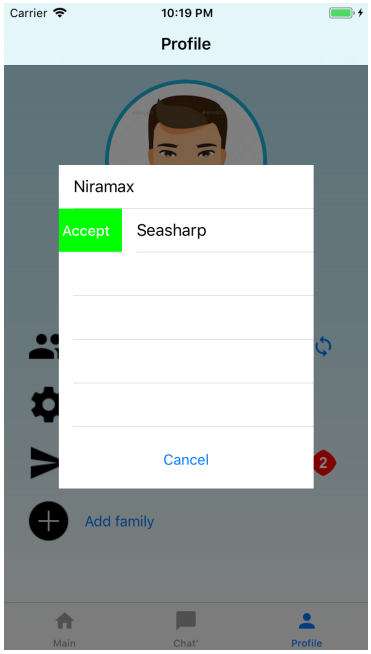


iOS

Management & roles distribution

Pending invites

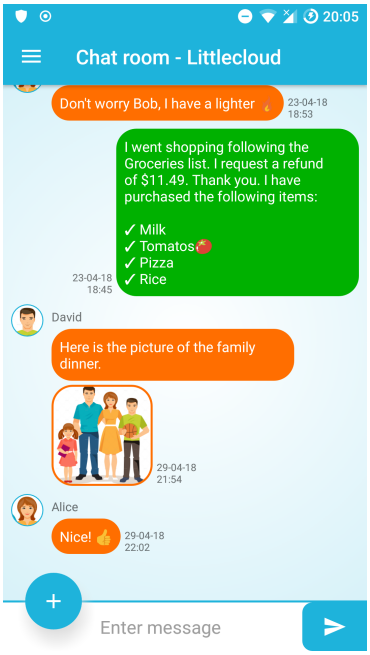




Android iOS

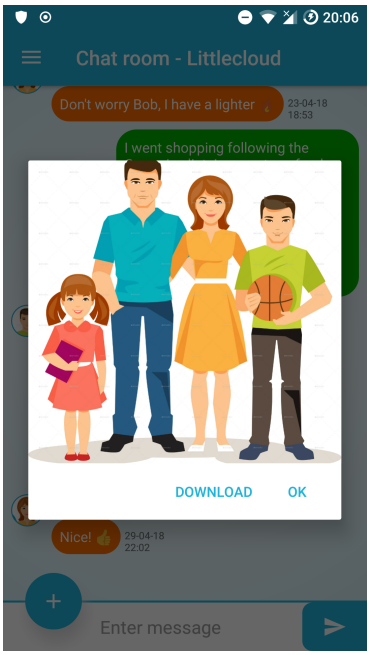
Pending invites

Chat room

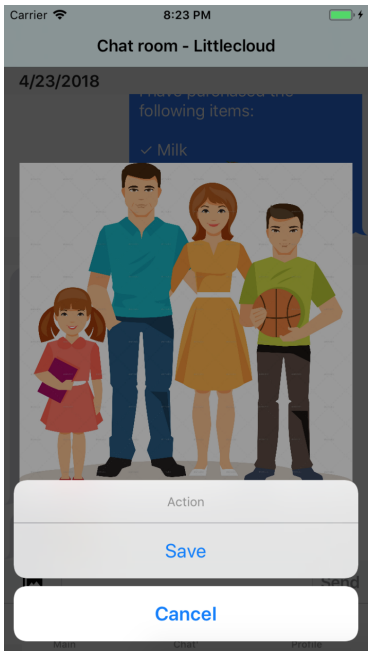


Android iOS

Chat room



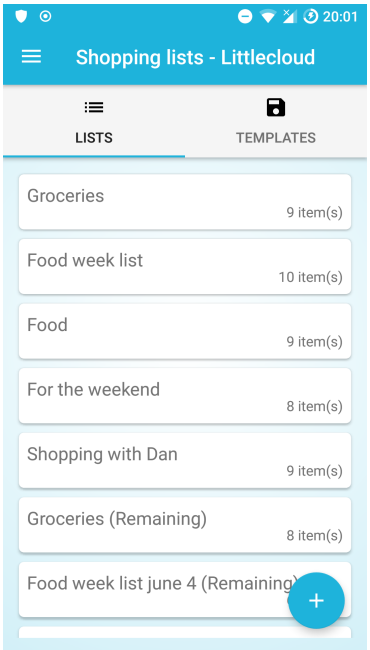
Android



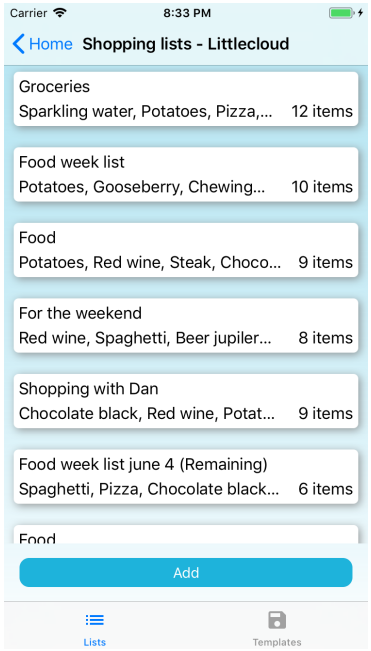
iOS

Display of an image

Shopping lists

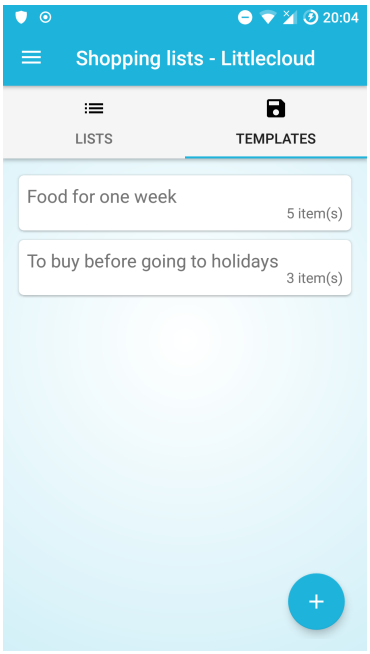


Android

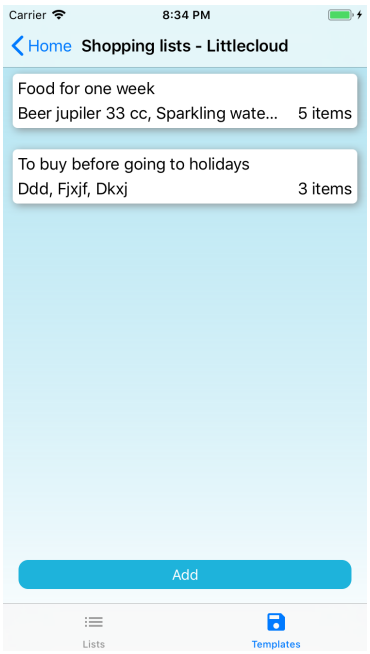


iOS

Lists

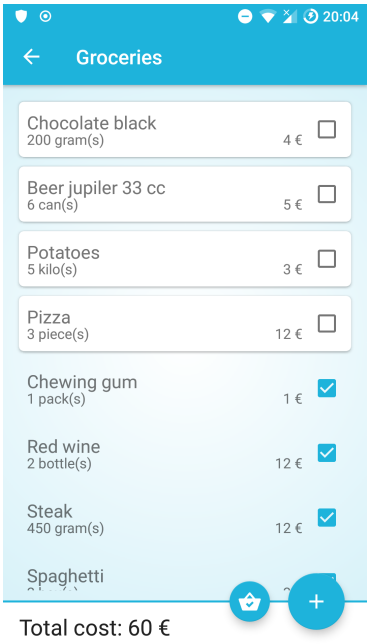


Android



iOS

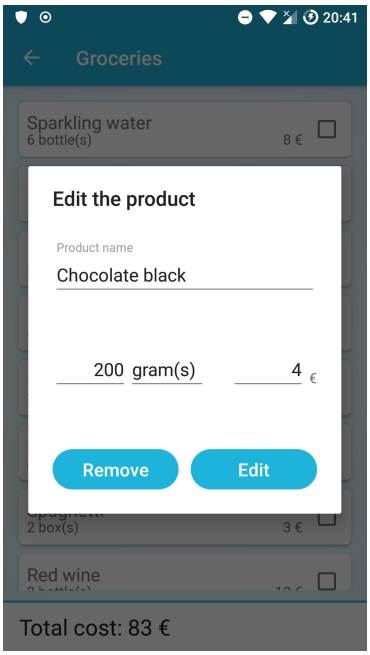
Templates



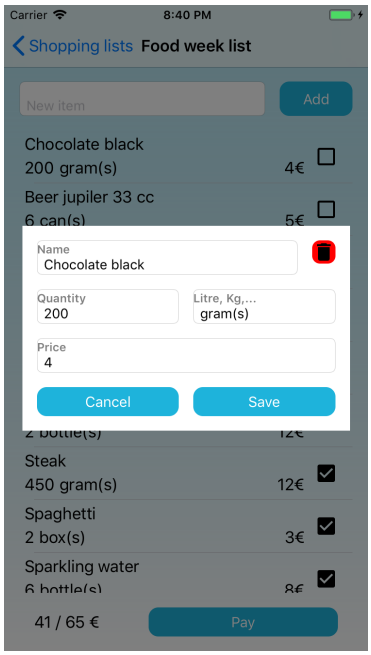
Android

iOS

Products list

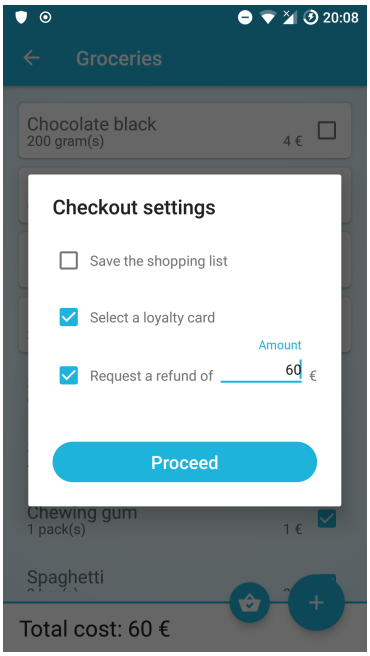


Android

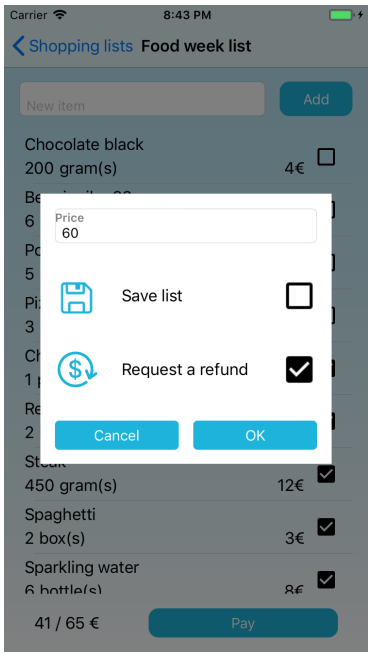


iOS

Product modification



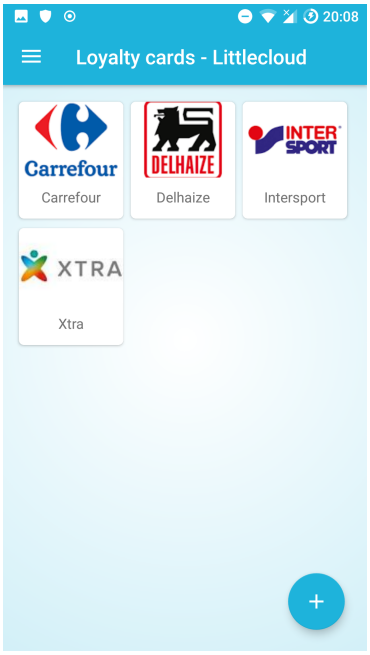
Android



iOS

Checkout

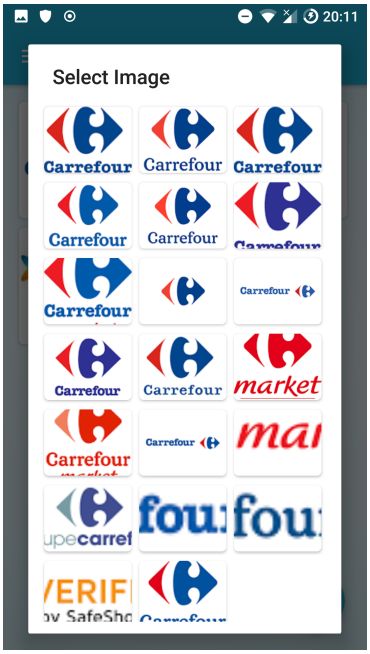
Loyalty cards



Android

iOS

Loyalty cards



Android

iOS

Image modification

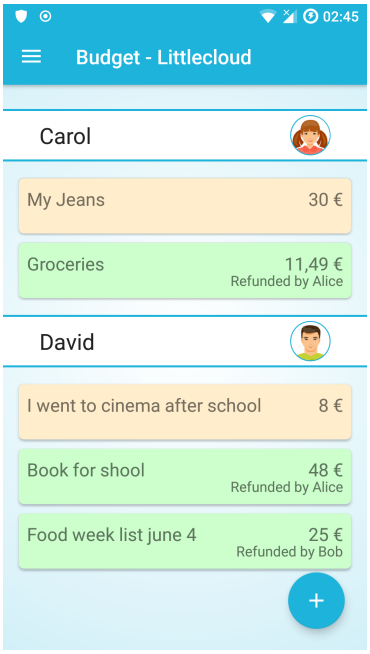
Android iOS

Barecode

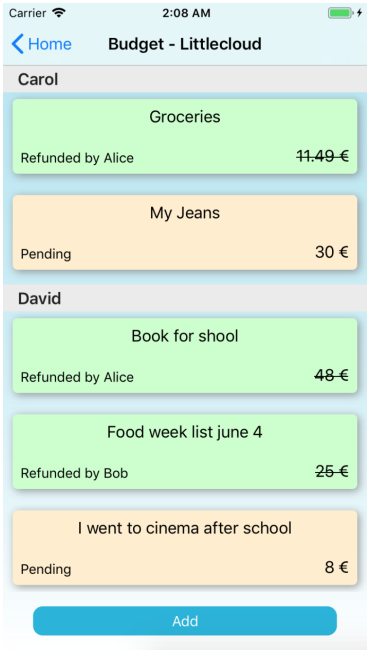
Android iOS

QR code

Budget

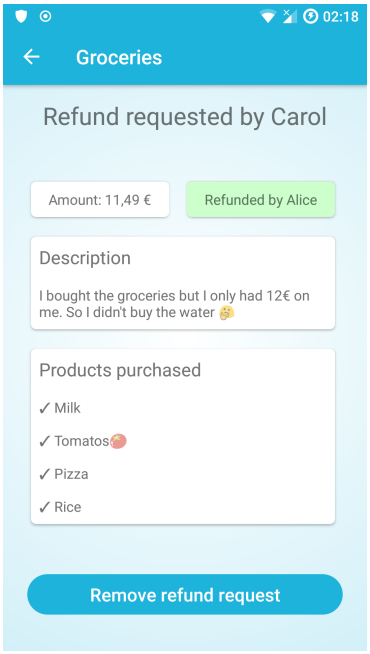


Android

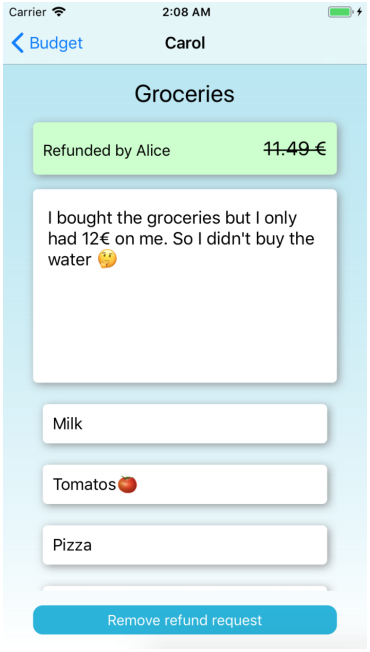


iOS

Budget



Android



iOS

Refund request

DynamoDB tables

Family

```
{
  "admins": [ "string" ],
  "id": "string",
  "name": "string"
}
```

Family-User

```
{
  "family": "string",
  "name": "string",
  "role": "string",
  "user": "string"
}
```

ChatMessages

```
{
  "family": "string",
  "messages": [
    {
      "author": "string",
      "image": "string",
      "name": "string",
      "text": "string",
      "time": "string"
    }
  ],
  "next": boolean,
  "part": integer
}
```

ShoppingList

```
{
  "active": boolean,
  "family": "string",
  "products": {
    "item_id": {
      "check": boolean,
      "check_version": integer,
      "name": "string",
      "price": number,
      "quantity": number,
      "type": "string"
    },
    ...
```

```
  },
  "list": "string",
  "name": "string"
}
```

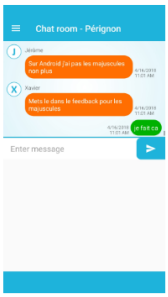
LoyaltyCards

```
{
  "cards": {
    "card_id": {
      "code": "string",
      "format": integer,
      "logo": "string",
      "name": "string"
    },
    ...
  },
  "family": "string"
}
```

Budget

```
{
  "refunds": {
    "refund_id": {
      "amount": number,
      "author": "string",
      "description": "string",
      "name": "string",
      "productlist": [ "string" ],
      "sub": "string",
      "refunder": "string"
    },
    ...
  },
  "family": "string"
}
```

Bug & improvement reporting from Instabug



Première lettre d un mot par défaut en majuscule

Email: [redacted]
Device And OS: motorola Moto G (5), OS Level 24
Locale: en_BE
Screen Size: 1080x1920
Density: xxhdpi
Location: Overijse, Belgium
App Version: 1.0.4 (4)
Duration: 00:02:52
Tags:

Android report #1

Comme sur facebook, ne montrez que la date du message si on clique sur le message, ca fera moins encombre

Email: [redacted]
Device And OS: motorola Moto G (5), OS Level 24
Locale: en_BE
Screen Size: 1080x1920
Density: xxhdpi
Location: Overijse, Belgium
App Version: 1.0.4 (4)
Duration: 00:03:35
Tags:

Android report #2

Si je relance l app je devrais être sur la dernière famille avec qui j ai discuter/fait une liste par sur la premiere de la liste

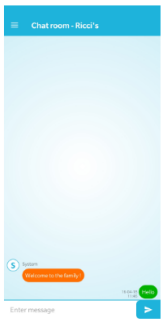
Email: [redacted]
Device And OS: motorola Moto G (5), OS Level 24
Locale: en_BE
Screen Size: 1080x1920
Density: xxhdpi
Location: Overijse, Belgium
App Version: 1.0.4 (4)
Duration: 00:00:12
Tags:

Android report #3

Changer le message de l'attente de confirmation du mail "cancel" vers "ok"

Email: [redacted]
Device And OS: OnePlus ONEPLUS A5010, OS Level 27
Locale: fr_BE
Screen Size: 1080x2160
Density: xhdpi
Location: Houtain-le-Val, Belgium
App Version: 1.0.4 (4)
Duration: 00:00:17
Tags:

Android report #4



Pour les chats, proposer la création de salon pour éviter que tout le monde parle dans la même discussion, tout en gardant le salon général

Email: [redacted]
Device And OS: OnePlus ONEPLUS A5010, OS Level 27
Locale: fr_BE
Screen Size: 1080x2160
Density: xhdpi
Location: ,
App Version: 1.0.4 (4)
Duration: 00:04:44
Tags:

Android report #5

10x5+10*2=70€ but the total is 7€. If it is an error, ok. If this normal, i think this is not intuitive and/or practical.

Nice application :)
Olivier

Email: [redacted]
Device And OS: motorola Nexus 6, OS Level 25
Locale: fr_FR
Screen Size: 1440x2560
Density: xxhdpi
Location: Ottignies, Belgium
App Version: 1.0.4 (4)
Duration: 00:06:14

Android report #6

TH [redacted] - iOS

April 16, 2018 11:31:53 AM

Report a problem

Le menu n'est pas super beau. Pourquoi les Loyalty cards sont en bas en pas en grand par exemple ?

Show Translation

Bug Details

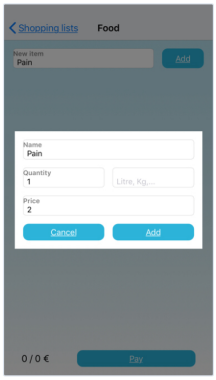
APP VERSION 1.0.18 (1.0.18)	DEVICE iPhone SE
OS iOS 11.3	CURRENT VIEW MainController
LOCATION Ixelles-Elsene, Belgium	DURATION 00:20:49

More Details

iOS report #1

JR [redacted] - iOS

April 18, 2018 7:36:39 PM



Suggest an Improvement

Ça serait bien une sorte d'auto complétion lorsqu'on ajoute un produit

Show Translation

Bug Details

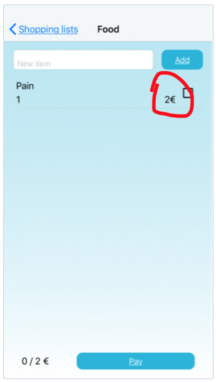
APP VERSION 1.0.18 (1.0.18)	DEVICE iPhone 6
OS iOS 11.2.6	CURRENT VIEW ItemPopUpController
LOCATION Leuven, Belgium	DURATION 00:01:26

More Details

iOS report #2

[redacted] - iOS

April 18, 2018 7:39:33 PM



Report a problem

Pas moyen de changer en dollar par exemple ?

Show Translation

Bug Details

APP VERSION 1.0.18 (1.0.18)	DEVICE iPhone 6
OS iOS 11.2.6	CURRENT VIEW ListDetailController
LOCATION Leuven, Belgium	DURATION 00:04:10

More Details

iOS report #3

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve **www.uclouvain.be/epl**