Project Report

Mobile and Ubiquitous Computing - 2020/21

Course: MEIC

Campus: Alameda

Group: 02

Name: Hugo Filipe Dias Marques Number: 89460 E-mail: hugofmarques@tecnico.ulisboa.pt

Name: João Carlos Morgado David Number: 89471 E-mail: joaodavid@tecnico.ulisboa.pt

Name: Daniel Abrantes Gonçalves Number: 91004 E-mail: daniel.a.goncalves@tecnico.ulisboa.pt

1. Features

Component	Feature	Fully / Partially / Not implemented?
Mandatory Features	Support Multiple Pantry Lists and Shopping Lists	Fully
	Associate Lists with a Location	Fully
	Automatic List Opening When Near Location	Fully
	Show/Edit List Information	Fully
	Show/Edit List Items	Fully
	Associate/Filter Items with Shopping Lists	Fully
	Select Items with Barcode	Fully
	Add Pictures to Items	Fully
	List Synchronization	Fully
	Store Checkout Queue Estimation	Fully
	Crowd-source Pictures	Fully
	Crowd-source Prices	Fully
	Caching	Fully
	Cache Preloading	Fully
Securing Communications	Encrypt Data in Transit	Fully
	Check Trust in Server	Fully
Meta Moderation	Data Flagging	Not implemented
	Data Sorting and Filtering	Not implemented
	User Trustworthiness	Not implemented
User Ratings	Item Rating Submission	Fully
	Show Average Ratings	Fully
	Rating Breakdown (Histogram)	Fully
User Accounts	Account Creation	Not implemented
	Login / Logout	Not implemented
	Account Data Synchronization	Not implemented
	Guest Access	Not implemented
Social Sharing	Sharing Items	Fully
	Sharing Item Data in Context	Partially
Optimize the Shopping Process	Optimize for Time	Not implemented
	Optimize for Cost	Not implemented
User Prompts	Prompt Photo Submission	Fully
	Prompt Price Submission	Fully
Localization	Translate Static Content	Fully
	Translate User Submitted Content	Fully
Suggestions	Compute Most Likely Item Pairings	Not implemented
	Prompt the User with Suggestions	Not implemented
Smart Sorting	Compute Item Ordering in Store	Fully
	Sort Shopping List for Store	Fully

Optimizations

We try to avoid sending too many requests to the webserver. For example, when updating the queue estimate times, we request for all stores at once, instead of sending a separate request for each one.
For smart sorting, since the item database can be somewhat big, we decided to have the server do the sorting for the client. The client sends the barcodes it wants sorted, to which the server sorts according to its matrix, replying with the barcodes in the correct order. The client then applies that order to its shopping list. Items that do not have a barcode appear in the end of the list and, since we use a stable sorting algorithm, they will be kept in alphabetical ordering in case of ties or products without barcodes.
We decided to share pantry lists using URLs instead of QR codes or other type of written codes. QR codes usually require both users to be in the same location for it to be a practical solution, as most applications do not support reading QR codes from files. Written codes are cumbersome to copy and write, and, while easier to share through messages or voice, we do not think it is a

great solution either. We found this solution, of sharing through URLs registered in the manifest, to offer the best usability, allowing the users to share the pantry lists directly through their application of choice.

For usability, we make use of simple touches for the most common options, long touches for opening more detailed menus or changing quantities more rapidly. We also use a side menu which supports a swipe right for opening, with the two main "directories", pantry lists and shopping lists, and finally we provide the usage of the commonly known "swipe down to update" for most menus that can be updated through the server, such as shared lists.

For dynamic translations, we keep a cache on the server with the previous translations. This is an optimization because translation APIs are usually expensive, and so must be used with caution. And instead of for example, all English users translating every single item to English every time, we only need to do the translation one time. This technique is not applicable to the driving time API, because, first, it is more uncommon to request pairs of locations than repeated translations, but mostly because the driving times depend on traffic conditions and so it would be wrong to assume they are static.

We allow users to select a "default store". A default store is a store where the user might do most of his shopping from, and so, conveniently, when adding a new product on the application, that store is automatically selected as having the product, while other stores are not selected by default.

2. Mobile Interface Design

An activity wireframe is in Appendix A, in the diagram we use black arrows to represent single touches while the red arrows represent long touches, which trigger different transitions. Transitions that pop the back stack were hidden for readability reasons. Dialogs and the initial splash screen are also not represented.

We have also hidden some transitions between some screens and the google maps activity and the barcode activity, which are in a blue rectangle. These transitions are ones that are available in the options menu, any time we have a barcode or a directions icon. Also hidden, is the interaction for the google maps application, which is invoked when a user requests the directions for a store, which opens the google maps application with the wanted path in navigation mode.

https://drive.google.com/file/d/1AXyptpqBZdKf8ceerUR21LBUXnh7KOpj/view?usp=sharing

3. Server Architecture

3.1 Data Structures Maintained by Server and Client

The client maintains all the user's pantry lists and the products inside them, which we call Items in the context of a pantry list. It also maintains an object for each store, which is then used to generate a shopping lists based on where the products are available. Most of our objects are identifiable by a randomly generated UUID, which applies to pantries, stores, products (we allow the same product to appear in multiple pantry lists) and product images. All these structures have the necessary data to be identifiable and hold enough information for our application requirements, such as names, barcodes, quantities, etc. For the shopping lists, we also use additional structures to aid with changing quantities and roll them back if the user wishes to.

The server, similarly, keeps track of most properties of the objects so that they can be shared between users, but also keep some additional structures such as Beacons, which are used for queue estimate times, Ratings, which are kept for crowd-sourced ratings as well as StoreSortings, which keep a matrix of what products are bought before others, to be able to sort them given that criteria. Most features are separated in what could be almost classified as micro-services, but all running on a single process. This means that most features are implemented in a single service, independent of the others. This implies that each service keeps track of the structures that are needed in order for it to function.

The shared pantries, pantries which have been shared with other users, are maintained by the server and require internet connection to be altered, while others can be changed offline.

While the server keeps the identifiers of the shared objects, do keep in mind that those are used when requested explicitly, and for most crowd-sourced features stores are usually identified by their location and products by their barcode. Product prices, ratings and "smart sorting" are examples of such cases.

3.2 Description of Client-Server Protocols

Our clients only communicate with the server through HTTPS, using a REST API and JSON objects for communication. In this sense, all communications are based on a pull approach from the client. The requests are serialized/deserialized in the client side using a JSON parsing library.

The clients and server keep similar enough structures to simplify their communication, and so most of the communication can be done by serializing the clients' structure into json, store them on the server, and when sent back simply parse them into Kotlin objects.

3.3 Other Relevant Design Features

To reduce the network usage and storage occupied by our application, the application compresses the size of each image that is uploaded, as it does not require a high-definition resolution and this way it improves the performance of each operation.

4. Implementation

The application was developed in Android Studio, using Kotlin, which we fully recommend as an alternative to Java. We decided to use a Side Menu navigation based on one of the templates that come with Android Studio, consequently, most of what would usually be activities are instead implemented as fragments, however, we still make use of the following activities:

- SplashScreenActivity: this is the entry point of our application, responsible for showing a pretty loading screen while it fetches the location and new updates, to avoid confusion with the user. Once ready, it jumps the SideMenuNavigation activity. We do not fetch the current location as to avoid a long loading time on the app start-up, instead we get the last known location and request a better update on the background after going into the main activity.
- SideMenuNavigation: this is the central activity for our application, having every submenu of the application. The side-menu itself allows changing between Pantry Lists and Shopping Lists. After SplashScreenActivity finishes, it jumps to either PantryUI, ShoppingListUI, ProductUI, or to the more general PantriesListUI. The first three screens can be shown automatically at beginning if it is close to the location of a Pantry List or of a Shopping List, or after clicking in a shared URL to open the corresponding list or product.
- LocationPickerActivity: used when a user wants to input a location for a pantry list or a store, returns the resulting location, making use of Google's Maps SDK.
- BarcodeScannerActivity: used to read barcodes for either looking up products or saving barcodes in existing products, this makes use of Google's ML Kit for bar code scanning with CameraX.

Besides those main components, we also make use of Bing Maps Routes API for calculating the driving times, and MPAndroidChart for creating the histograms for product ratings. In the usage of all the components that require communicating with a third-party API, we use our server as proxy in order to not expose our API keys and make the client only need to communicate with a single agent. This also allows the server to do any additional behaviors such as, in the case of the translations, acting as a cache for previous translations.

As mentioned, the communication between the client and server is done through HTTPS. We make use of Volley, which works in an asynchronous fashion, making use of callbacks. Due to this, we did not have to create many threads manually because the libraries handled most of our necessities. The SplashScreenActivity, however, must launch another thread while loading so it does not block the main thread. In mobile applications it is sometimes recommended to do disk reads/writes in a different thread, in our experience, the reads and writes performed did not affect performance at all, and so we did not find that necessary, which might be due to the small size of the stored images. In a different setting, it could be worth taking that into account.

For keeping the data updated, most data are requested only when necessary (instead of polling through a service), however, we implemented a "swipe down to update" on most menus. Most of the data updates resulting from callbacks, such as the one mentioned, are then propagated to the current active fragment through a callback that is registered on "ShopIST", the component that we use for sharing global state. Besides storing current helpful callbacks, "ShopIST" also keeps the whole application state, such as pantry lists, current location, and some other useful properties and functions used throughout our application. The data structures are kept on the "domain" package, with domain objects and DTOs for communication.

On the “utils” package, we have common utilities such as communicating with the server, and, most importantly, the cache. The cache was implemented using Android’s LruCache, making use of a “CacheItem” for indirection, which points to the files stored on the filesystem and returns their size according to the corresponding file size, allowing us to easily implement a LRU Cache with a file size limit. On the same package we also have a LocaleHelper, for dealing with localization, and a LocationUtils with some wrapper methods around the Android methods for getting a new location and the last known location. This package also contains the QueueBroadcastReceiver, which is the receiver for WiFi-Direct events, in this case, simulated through Termite.

The UI components reside in the “ui” package, containing the application’s fragments and dialogs. The information is passed between fragments using Bundles, which are passed when a given Navigation transition is triggered. Since all fragments have a top bar, which corresponds to a different fragment in the SideMenuActivity, it is controlled through the TopBarController, which has function to activate and deactivate top bar buttons as needed.

For persistent storage, we resorted to simple files, serializing either through JSON (and Gson), or using Java’s serialization for storing some metadata about the cache. Java’s serialization is known to be slow, but it did not seem to be a problem in our application, but there are also alternatives known to work well as an alternative to the traditional serialization.

The backend was implemented in Node.js, with TypeScript, mostly due to previous experience and simplicity of use. Even though Node.js might have some problems, relating to its type safety and single threaded nature, since the project’s scope is small, we’ve decided the simplicity outweighed the problems it could have brought. Were we to increase the project’s scope we might have needed to choose another language.

The server is separated in:

- model: contains the domain classes that model the domain and implement their corresponding logic.
- services: the services act mostly as a bridge between the model and the routes, keeping track and organizing the necessary objects for the current execution.
- routes: expose the API endpoints and call their associated services.

As recommended in the project statement, all data is non-persistent and stored in simple runtime TypeScript objects. The fact that our server runs on a single thread also simplifies our worries related to concurrent requests, as the execution is based on an “event queue”.

6. Simplifications

The application assumes that a store only has a beacon associated, removing the possibility of a client connecting with multiple beacons in a single store.

Related with the synchronization of the Pantry Lists (with everything associated with it, as Products, Stores, etc.), the application assumes that everyone has the last update, as imposed by the own application – as it does not allow any update if not connected to the server –, so when synchronizing it just substitutes the value updated and does not care about conflicts. This might result in synchronization problems, if the connection is unstable or slower, as, for example, two users at the same time change different fields, only one would be stored.

The application also assumes that the users will accept all the permissions when requested, as it would not function properly otherwise. This way it does not verify if all the permissions are always granted, which in a real-world deployment is incorrect, as the users do not comply with all the permissions requested.

For the queue time estimates we make some assumptions. We do not take false positives into account, for example, if a user enters the beacon and immediately leaves it, because he might have forgot an item, it will take it as if he had completed the checkout. We experimented with setting a threshold, but it made testing more cumbersome as we had to wait over that threshold and it is not easy to define a good threshold, so we decided to remove it mostly for testing purposes, as it also does not affect our prototype. We also assume that each store only contains one beacon and so there are never, at any moment, two beacons overlapping one another, which simplifies detecting when we should send request for registering having left the queue. We also assume that every beacon starts with the name “ShopIST” for detecting, but this, in fact, could be a realistic scenario for a deployment of our application.

As mentioned, we do the sharing of pantry lists through an URL, because a URI was not clickable in the message application of the emulator. This leads to a problem where, when opening a shared link, Android prompts to either opening with the web browser or with our application. We assume that the user picks the application in that case as we do not have a server ready. We found this

to be worth mentioning, even though it is not a serious problem, as a solution could be to simply host a webpage that redirects to the Google Play Store.

7. Bugs

In the additional component, Social Sharing, Sharing Data in Context was not fully implemented, as we did not have time to implement sharing a message with the product image as thumbnail, when sharing a product.

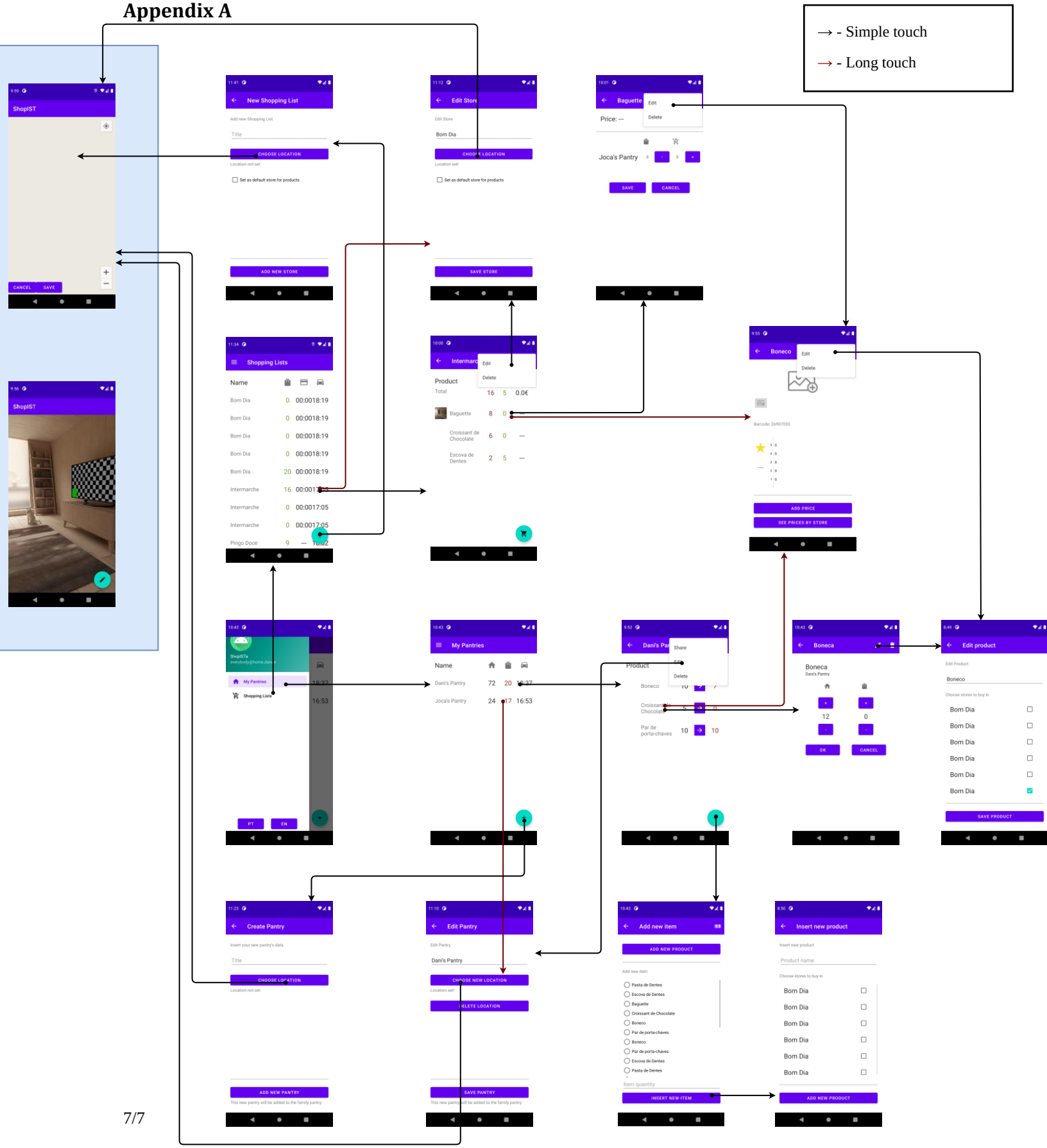
Sometimes, after adding an image, the image will not show on the product view immediately, requiring a swipe down to refresh. This issue is probably due to some race condition within the callbacks and happened so rarely that we could not properly debug it.

8. Conclusions

This project allowed us to learn plenty about mobile applications, Android development, and Kotlin, being very complete and thorough, however, we do sincerely believe that the project was way too much work. In comparison to the older projects, we found that the difference in the domain complexity was enough to set back the development of the rest of the features for a long time. And mostly, even though it certainly is interesting to develop those problems, a lot of features did not help our learning of mobile applications. When compared to other courses, this project was more than double the work of others, and, from our experience, most students tend to agree with this sentiment.

However, this project still gave us a nice overview of what a real product is, how we should engineer our mobile applications. As the domain of this project was based on real needs, there was a sense of usefulness and not just like one more project to do.

Appendix A


→ - Simple touch
→ - Long touch
ShopIST
New Shopping List
Edit Store
Baguette
Price: ---
Joca's Pantry
SAVE
CANCEL
CHOOSE LOCATION
Set as default store for products
ADD NEW STORE
SAVE STORE
Shopping Lists
Name
Bom Dia
Intermarche
Pingo Doce
Product
Total
Croissant de Chocolate
Escova de Dentes
Boneco
ADD PRICE
SEE PRICES BY STORE
My Pantries
Dani's Pantry
Joca's Pantry
Share
Delete
OK
Edit product
Boneca
Choose stores to buy in
SAVE PRODUCT
PT
EN
Create Pantry
Edit Pantry
CHOOSE NEW LOCATION
DELETE LOCATION
ADD NEW PANTRY
SAVE PANTRY
Add new item
ADD NEW PRODUCT
Pasta de Dentes
Baguette
Par de porta-chaves
INSERT NEW ITEM
Insert new product
Product name