ITI Bachelor Project

August 2018

PocketHepia
A student card platform based on NFC

Maxime Alexandre Lovino

Supervised by
Prof. Mickaël Hoerdt

h e p i a
Haute école du paysage, d'ingénierie
et d'architecture de Genève

Hes·so Genève
Haute Ecole Spécialisée
de Suisse occidentale

Contents

List of Figures

Acknowledgements

Terms and Definitions

IDE est un acronyme pour Integrated Development Environment. Il s'agit d'un programme intégrant un éditeur de texte avec coloration syntaxique et autocomplétion du code pour un ou plusieurs langages de programmation ainsi que des fonctions de compilation, débogage et tout autre fonctionnalité permettant de faciliter et fluidifier le travail du développeur.

JVM est un acronyme pour Java Virtual Machine. Il s'agit d'une machine virtuelle permettant d'executer un programme compilé en bytecode Java sur un ordinateur ou un terminal mobile. On parle de langage fonctionnant sur la JVM lorsque la compilation du langage produit du bytecode Java..

NFC est un acronyme pour Near Field Communication qui représente un ensemble de protocoles de communication permettant de communiquer à des distances de quelques centimètres en utilisant l'induction électromagnétique.

Chapter 1

Introduction

Since starting my studies at HEPIA almost three years ago after two years at EPFL, I've been shocked by the lack of commodities and study spaces for students. Some of this lack is due to the difference in scale between the two schools, one of them being composed of mainly three building in a constrained city environment and the other an almost-autonomous campus outside the city. But, actually, there isn't really a lack of space at HEPIA, but a lack of space that students can use to study. Most of the classrooms are locked when not in use by a teacher. When asking about why that is the case, I've been told that it was mainly for security concerns because of the equipment present inside the rooms. If the rooms stayed unlocked, there was no way of knowing who stole or broke something. I suggested the idea of giving access to select students to these classrooms to study and work on projects but it wasn't practical because copies of keys had to be made, the students had to make a money deposit to make sure that they didn't run away with the keys etc.

One solution would have been to use our student cards as an electronic door key to access the rooms. We could also enable other uses for the cards, such as using them as electronic wallets to simplify the payments at the canteen during lunch break. When I suggested the idea, people mostly laughed at me and said "[...] they're never going to do it, unless someone actually does it, presents it as a finished product and then they decide to use it." So, here I am, after 3 years studying at HEPIA and for my Bachelor Project I decided to work on this exact idea. A multi-function student card platform built on NFC technology with accompanying mobile and web application for administrators to manage the platform and for students to monitor their usage statistics.

1.1 NFC and NDEF

1.1.1 Introduction to NFC

NFC (Near Field Communication) technology is a low range wireless communication technology based on electromagnetic induction. Its usages are widespread, ranging from access control to contactless credit card payments and public transportation tickets.

The range of a NFC communication is more or less 4cm. NFC can be implemented in a physical chip on a card, in a discrete reader or in a smartphone. Most Android smartphones and all iPhones after 2016 have a NFC chip.

There are three operating modes for NFC devices. A device is considered a *Full-NFC* device if it can operate in the three modes, but most of the NFC devices we interact with only support some of the modes:

- The ”card emulation” model, also called passive mode enables a device to be used as NFC tag containing data similar to the ones present in physical NFC cards. This mode is the only one available on devices with no power, for example credit cards. This mode is called passive because it uses power from the active NFC reader to power it and enable access to its information. It can also be used on a smartphone to emulate a physical card. It is for example the mode used by mobile payments application such as Apple Pay and Google Pay. The NFC reader simply detects a standard NFC card, without knowing that is in fact emulated by a smartphone.
- The reader mode, also called active mode. This mode allows reading NFC tags data with a smartphone or a specific reader, for example in an electronic doorknob or a metro station gate.
- The peer-to-peer mode is used by two smartphones to exchange information between them. This mode is used on Android by the Android Beam service which allows to send pictures or other data between two Android devices by placing them back to back. In this case, the NFC peer-to-peer mode is used to initiate the connection and pair the device and then the data transfer is carried over Bluetooth or Wi-Fi Direct.

Figure 1: A NFC card reader on the London Underground (active mode) and a Oyster Transport Card (passive mode) [1]

1.1.2 NDEF Format

The NDEF (NFC Data Exchange Format) format is a standardised format used to encode data on NFC tags. By using a standard format, it allows for easier exchange of information between NFC devices. A NDEF message is composed of NDEF records. Each of these records contains metadata that provides information to help decode the data payload contained in the message.

The format of a NDEF record is the following[2]:

```
Bit 7     6       5       4       3       2       1       0
------  ------  ------  ------  ------  ------  ------  ------
[ MB ]  [ ME ]  [ CF ]  [ SR ]  [ IL ]  [        TNF          ]

[                          TYPE LENGTH                        ]

[                        PAYLOAD LENGTH                       ]

[                           ID LENGTH                         ]

[                          RECORD TYPE                        ]

[                               ID                            ]

[                            PAYLOAD                          ]
```

The first byte form the header of the NDEF record, the fields composed the header are:

- The `TNF` (Type Name Format) is a 3 bits value used to identify the record type. The most common are "Empty record" and "Well-Known-Record" to define a record for which the data type is defined by a RTD (Record Type Definition) in the NDEF format, for example plain text or URI.
- The `IL` (ID LENGTH field) is a boolean 1 bit value to tell if the `ID LENGTH` field is present or not in the record.
- The `SR` (Short Record Bit) is a boolean 1 bit value to tell if the `PAYLOAD LENGTH` has a size of 1 byte or less.
- The `CF` (Chunk flag) is a boolean 1 bit value to tell if the record is the first part or a chunk of a bigger record. The value will be 0 if it is the only part or the last chunk of a record.
- The `ME` (Message End) is a boolean 1 bit value to indicate that the record is the last of the NDEF message.
- The `MB` (Message Begin) is a boolean 1 bit value to indicate that the record is the first of the NDEF message.

1.1.2.1 Text payload

To give an example of a data payload contained in a NDEF record, we are going to take a closer look at a plain text payload.
The text payload format is defined like this[3] :

```
Bit 7     6       5       4       3       2       1       0
------  ------  ------  ------  ------  ------  ------  ------
[ ENC]  [RFU ]  [                IANA LENGTH                 ]

[                        IANA LANGUAGE CODE                  ]

[                            TEXT                            ]
```

The first byte is called `Status Byte` and contains `ENC` which stores the value of the encoding used in the payload (0 for UTF-8, 1 for UTF-16) and `IANA LENGTH` which stores the length of the `IANA LANGUAGE CODE` field found afterwards. The `RFU` bit is always set to 0. In the `IANA LANGUAGE CODE` is stored the ISO/IANA language code specifying the language of the text stored in the payload as defined in RFC 3066[4], for example "en-US" for American English. The encoding of the language code is always ASCII. Finally we have the text whose length is the length of the payload minus 1 byte and the length of the language code field. The text can be decoded using the encoding specified in `ENC`

1.2 Project inspiration - Camipro EPFL

1.2.1 The card and the official core platform

The inspiration for this project mainly comes from my experience studying at EPFL. The EPFL student card, named Camipro[5], contains an RFID chip that allows to perform various tasks that simplify student life on campus. The use cases for the Camipro card are the following:

- Electronic wallet to pay at every canteen on campus as well as select third-party retailers (for example the Migros shop present on campus)
- Access key to unlock doors and buildings
- Card to collect documents sent to the centralised printing pool system at any printer on campus
- Rent public bicycles[1] on campus and in the Lausanne area
- Rent cars by linking a Mobility[2] subscription to the card

[1] All students have access to a free Publibike account on their card `https://www.publibike.ch/en/publibike/`

[2] `https://www.mobility.ch/en/`

- Borrow books at the library
- Use electric car chargers on campus
- Turn on booked electrical barbecues present on campus[3]

An accompanying web platform called MyCamipro was built as part of the Camipro launch to manage and activate the different services on the card as well as see the recent transactions and the rooms we were given access to.

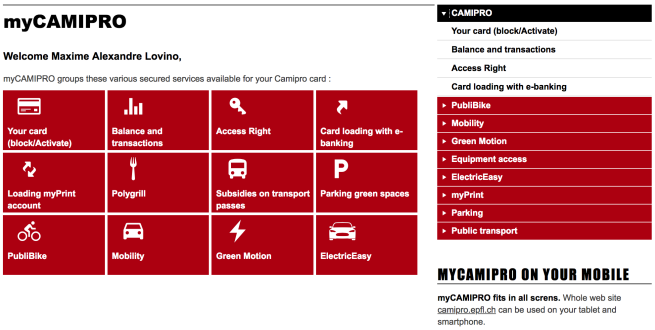

Figure 2: Screenshot of the MyCamipro website

1.2.2 PocketCampus

With the increased usage of smartphones by students, in 2010 a team of 20 computer science students decided to build the PocketCampus application as part of a software engineering class. They continued working on the project after the academic project was finished and it became the official EPFL application in 2013. [6].

Initially you could mainly see the balance of your Camipro card on the app, but version after version, the development team added new integration in the app by collaborating with different services at EPFL. These functions include:

- Accessing the menus of all canteens on campus
- Searching through the whole EPFL directory
- Having access to IS-Academia data to see course schedule and grades
- Printing from your smartphone on the EPFL print system
- Accessing Lausanne public transportation itineraries
- Accessing Moodle documents

[3]The service PolyGrill allows students to book free barbecues on campus and access them with their camipro `https://camipro.epfl.ch/cms/site/camipro/lang/en/polygrill_electric_barbecues`

After having integrated every requested features, the team launched a beta web version of PocketCampus for EPFL in June 2018. They also started diversifying their business by working on PocketCampus as a platform that can be integrated in other companies and stopped working exclusively with EPFL. They announced plans on partnering with Lausanne University (UNIL) to integrate their platform there.

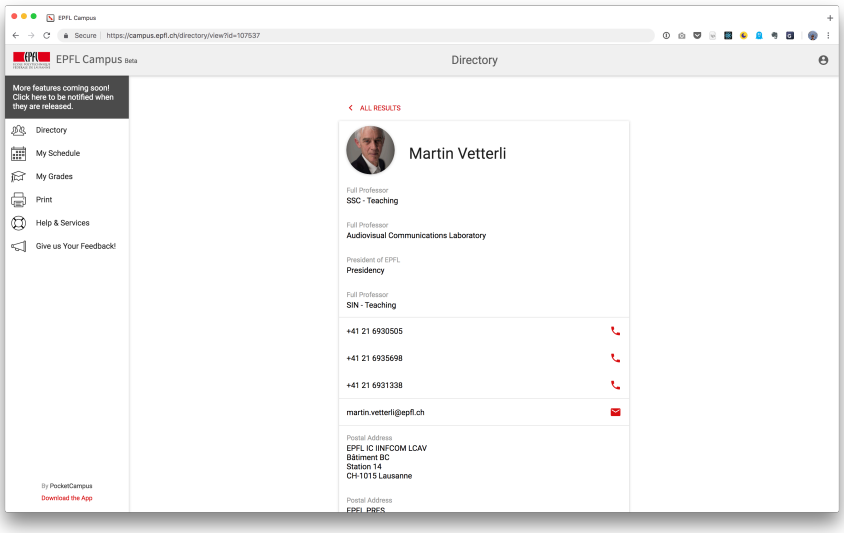

Figure 3: Screenshot of the web version of PocketCampus

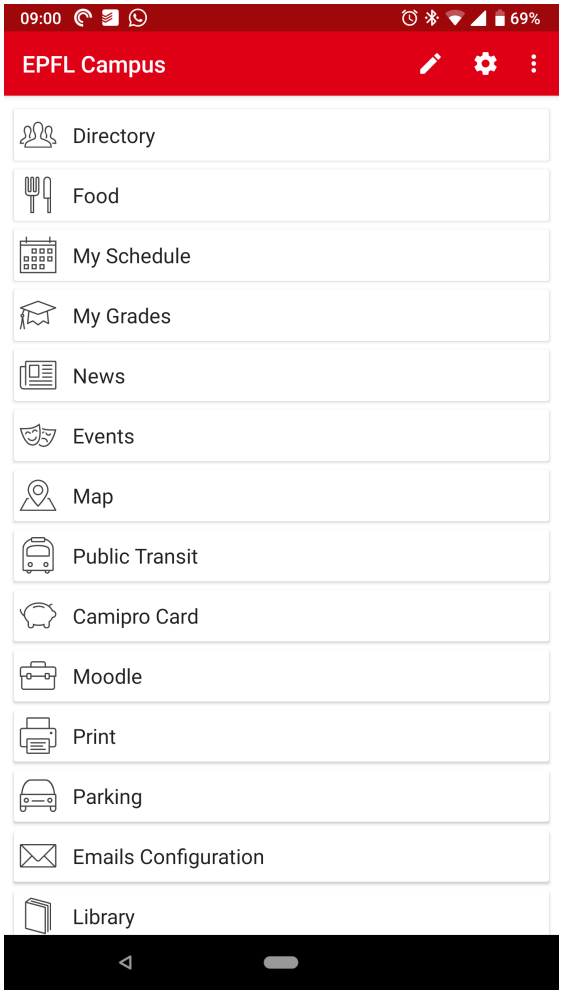

Figure 4: PocketCampus Android application

Chapter 2

The project

The project is called PocketHepia, reminiscent of the name of the project at EPFL and will consist of the two parts of the EPFL project presented earlier: the student card platform, as well as the applicative platform to access information.

The idea of this project is not to build as many features as the EPFL platform due to the time constraints inherent to the Bachelor Project, but rather to focus on some key aspects of the card at first, namely payments, access control and library. We are also building features on the payment side that are not available on the EPFL platform, for example the ability to send money between users. The project of course will be extensible with other features outside the scope of the Bachelor Project.

In this chapter, we will present the main features, roles and user stories for the project as specified at the beginning of this one. All of these are part of the project, but not necessarily part of the Bachelor Project. The idea was to specify the entire project and to start implementing a subset of the functionality for the academic project and continue working on the rest as well as new features outside this scope. Mainly we wanted to remain independent of other school services and infrastructure because of the associated time it would take to get permissions and setup integrations, as well as restrictions to development environments to secure access to these services. For each element of this chapter, we will specify whether it is going to be implemented as part of the Bachelor Project.

The project is composed of three main components:

- The physical student card
- An administration component
- An user facing component

A mobile application and a web application will be developed and both will offer the same user facing features as well as specific administrative features relevant to each application.

2.1 Features of the app

As stated before, we decided to limit the project to the three most important features in our opinion and, out of these, two are going to be implemented as part of the bachelor project.

2.1.1 Payments

The first feature is the electronic wallet functionality. We decided to start with this one because it would be bring the most benefits to the students and would not need specific hardware or collaboration with any school service. Every user has an electronic wallet linked to its account on the platform and can then send money to any other user by choosing the amount to send and tapping the user's card on his phone. Users can be assigned a specific role (see section 2.2) to allow them to receive money directly from a card.

There will be two main ways to add money to an user wallet. Either the user can give cash money to a person with administrative role that will then add money to the user's balance. Or the user will be able to add money to its account by using its credit card on the web application.[1]

2.1.2 Accesses

To continue, the second element of the project is the handling of access control through the platform. This was the original pain point we noticed at HEPIA with the lack of an easy way to handle accesses to the available classrooms. We will be able to create different areas to group rooms inside them. For example, an area could be an entire floor or labs for a specific section of HEPIA.

As far as giving access to users, we can give access to a room from a given date and specify if needed an end date for the access. For example, we can setup access for a user until the end of the current semester. We also decided to introduce an optional time range during the day during which the given access is active, so we can for example allow a student to enter a room only between 8am and 7pm.

We also specified the ability to delegate the administration of an area to a user, so that access to rooms can be handled at a department or section level in the school for example (see section 2.2). This feature is planned but will not be implemented as part of the Bachelor Project.

[1] At the moment, the project is still in the development phase so no bank accounts have been linked to this project and you will be able to recharge your account only using a specific test credit card generated for development purposes

2.1.3 Library books

Finally, the last feature of the project would be an integration with the library to be able to use the card as a library card. Our approach at first was to think about creating the book loans on our platform with the ISBN to identify the book and integrate with the REST API built for BibApp but it wouldn't have been very effective because the library already handles the loans through the NEBIS platform.

The correct way to handle this would be to link the library NEBIS identifier for every user with their account on our platform and then ask NEBIS for API Access to their platform to retrieve book loans for all users and display them on the mobile and web application. This would also benefit people working at the library as they would not need to change their current workflow to accommodate.

We don't have the time during the bachelor project to start a discussion with NEBIS to get access to the required information so this feature will have to be implemented later on.

2.2 Roles

We defined a set of roles for the users on the platform. First, every member of the platform is a simple user. This means that the person has an account on the platform, can login to the web and android application and has read access to its payments, its accesses and its other information and can send money to other users.

Then, there is the admin role that can be added to an user. This role enables the creation of users, attribution of roles, the consultation of administrative logs as well as the creation of access components (rooms, readers) and the attribution of accesses to users.

While these two roles would have been enough to handle all our features. We decided to specify roles specific to different components of the platform.

One of these roles is the "Accept payments" role. This is specific to a canteen or a shop that wants to handle payments using the student card. While every user can send money to another user from the mobile app by tapping the other user's card, this isn't very practical for a canteen or a shop where the transactions should go the other way around. So the shop creates the payment and taps the user's card to take money from it. At first, we wanted this feature to be available to everyone but we thought about security concerns with this solution because a student could create a payment from the app and start tapping his phone on lost cards or even on people's pocket and if the card was detected it would "steal" their money. So by enabling this feature as a role, we could only allow trusted people, such as canteen's owners to receive payments in that way.

Furthermore, due to the introduction of GDPR recently, we created a specific role called "Auditor" to access sensitive logs concerning all users, mainly access logs and transaction history. Only a user with this role can view all transactions between users and access logs for every room.

Then, there are two roles that will not be implemented as part of the Bachelor project, the "Can invite" and "Area admin" role. The first is to allow specific users to create temporary accounts without needing to contact an administrator. A use case for this would be a teacher creating a temporary account for a visiting colleague from another school. The "Area admin" role consists of delegating the administration of the accesses for an area to an user. Similar to the concept of DNS zones delegation, an administrator could for example give this role to the ITI section dean to allow him to give access to the rooms present on his floor.

Finally, the last role is the "Librarian" role that as its name implies is attributed to users working for the library. This role enables the creation of books loans at the library for users. As stated in section 2.1.3, this role will certainly be removed completely because there is already a system in place to handle book loans at the library.

2.3 The models

2.3.1 User Accounts

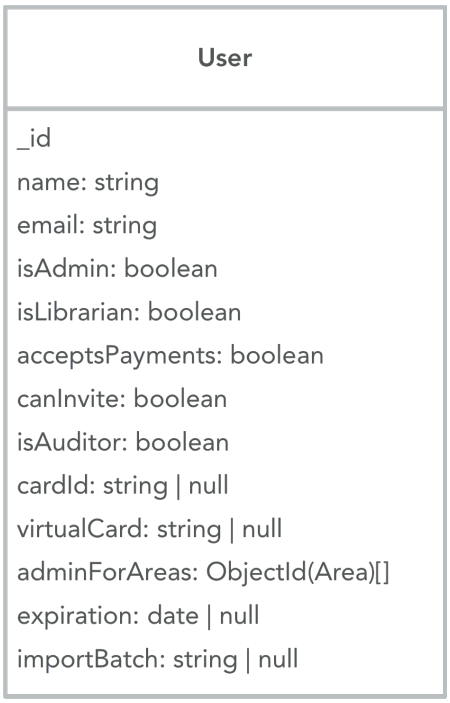

Figure 5: User Model

The first of our models is the User. A user collection is created to store all users on the platform. Users are identified by their email address and will login with their email and password, their full name is also stored. The authentication specific fields will be added to the model by Passport.JS (see section 2.5). A boolean field is also stored for each permission.

The expiration date, the virtual card and the delegation of admin zones are not implemented as part of this bachelor project.

Finally, the `importBatch` is set if the user is imported with a CSV file so that we can group all users imported together and undo the import if necessary.

2.3.1.1 Integration with existing AAI user accounts

If this project is put in production at HEPIA, we will have to only change this model to integrate it with an existing authentication service, such as AAI or with a directory

LDAP service.

In that case, we would remove information already present in the authentication service such as name and email and just store the unique identifier from the authentication service in our model. This model would act as an augmented database on top of the existing service. We would then forward all login/password authentication request to the authentication service and would create the entry for the user in our database after the first login.

This is inspired by the way Nextcloud[2] for example stores user accounts that are linked to a directory service, for example an Active Directory. It stores the full Distinguished Name from the directory service as the unique identifier for the account to link the local account to the directory account.

While we could have asked Switch-AAI to integrate AAI authentication in our application, the time constraints of the project were too tight to launch a discussion and we wanted to stay independent of other services in this phase to avoid being slowed down by for example needing to run our servers on the HEPIA network exclusively to access the directory.

2.3.2 Payments

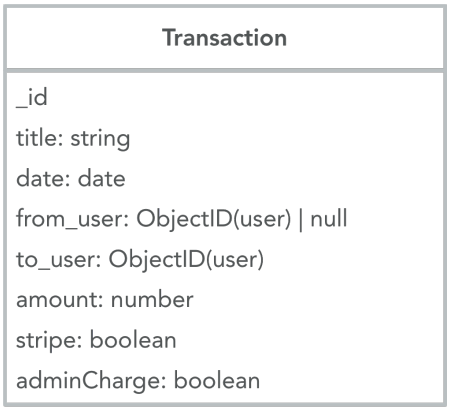

Figure 6: Transaction Model

To handle payments, we defined a Transaction model that represents a money transaction. We store the id of the `to` and `from` users instead of embedding the users document inside the transaction in order to avoid using too much space in the database. The number of transactions can grow very high and it is a waste of space to store two full

[2] `https://nextcloud.com/`

user documents in each transaction[3]. Finally, we handle the two type of recharges for the user account by setting the `from` user to `null` because the money does not come from any user account and set the corresponding `stripe` or `adminCharge` field to true.

The balance of a user is calculated from the list of transactions in which the user takes part. It is calculated as the sum of the transactions in which the user is the beneficiary minus the sum of transactions in which he makes the payment.

2.3.3 Accesses

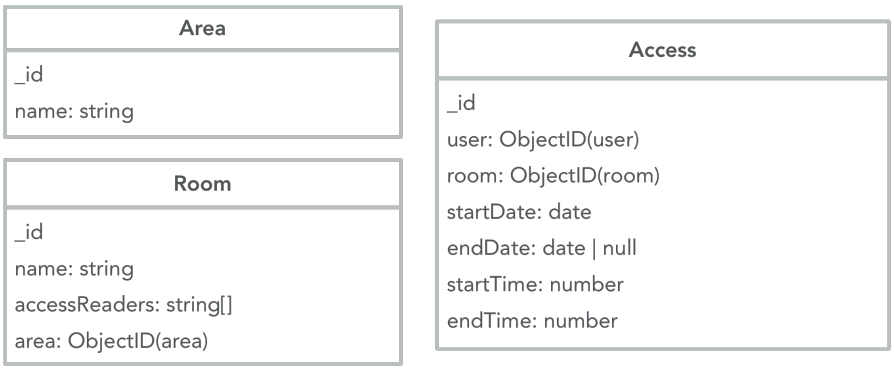

Figure 7: The three models handling accesses

The access part of PocketHepia is composed of three models. We defined Areas, Rooms and Accesses. An area contains multiple rooms and each room has a specific set of accesses. Similarly to the payments model, we reference from the child to the parent with the ID of the parent. The room has the id of the area it is part of, and the access stores the id of the user and the room.

When deleting an area, we delete all rooms contained in that area. When deleting a room, we delete all accesses associated with the room. This is done using Mongoose hooks (see section 3.2.3.4) to ensure that on every removal, we cascade it through the children.

[3]For more on that, see section 3.2.2 on Mongo References

2.3.3.1 Access requests

2.3.3.2 Access logs

2.3.4 Logs

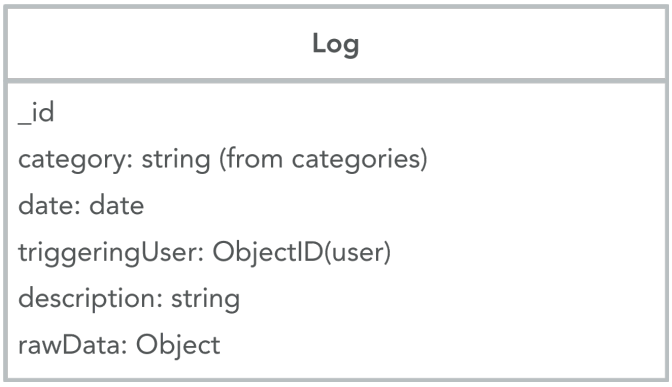

Figure 8: Log Model

Finally, we decided to log all administrative actions and make the logs visible to all admins on the website. Giving access to a room or creating a new account can be sensitive, so we have to keep an historic of those actions.

We defined a Log model to store the logs in the DB and several log categories to distinguish the different logs and filter them.
The log categories are:

- User creation
- User changes its password
- User deleted
- Admin imports users
- Admin cancels an import
- Admin assigns a NFC Tag
- Admin removes a NFC Tag
- Admin adds money to a user balance
- Admin resets a user password
- Admin changes a user permissions

- Admin creates an area
- Admin deletes an area
- Admin creates a room
- Admin deletes a room
- Admin gives an access
- Admin removes an access

As well as categories for features we will not implement yet:

- Admin delegates an area
- Admin removes an area delegation
- Admin completes an access request
- Admin deletes an access request

The category for a log entry is stored as a `string` and we setup validation on the model to ensure the `string` used corresponds to a category.

We also store an embedded object we call `rawData`, this allows us to store additional information specific to each category. For example for a permission change, we store the user before and after the changes.

Finally, the field `triggeringUser` stores the id of the user that did the action, we can then filter the logs to find out everything a specific person did.

2.4 User stories

From the beginning, we wrote simple user stories as a roadmap for the project features we wanted to implement. All of these features are not part of this academic project but are part of the broader student card project.

User story name	**Priority**	**Status as of July 11th 2018**
User must be able to login using username and password (then using JWT to communicate with backend)	1 - Essential	Done
Admin can create users from the website	1 - Essential	Done

User story name	**Priority**	**Status as of July 11th 2018**
User can view its total balance and a summary of its information on the homepage (web and application)	2 - Important	Done
User can view all its transactions on the transactions page (web and application)	2 - Important	Done
User can view its total balance on the transactions page (web and application)	2 - Important	Done
User can view all its accesses to rooms on the access page (web and application)	2 - Important	Done
Admin can delete users from the website	2 - Important	Done
Admin can reset password for all users from the website	2 - Important	Done
Admin can create users in batch by importing a CSV file	2 - Important	Done
Admin can create areas from the website	2 - Important	Done
Admin can create rooms from the website	2 - Important	Done
Admin can give access to a room for a user from the website (start/end date and timerange)	2 - Important	Done
Admin can remove an access from the website	2 - Important	Done
Admin can view all accesses for a user	2 - Important	Done
Admin can view all accesses for a room	2 - Important	Done

User story name	**Priority**	**Status as of July 11th 2018**
Admin can assign a physical card to a user from the Android app	2 - Important	Done
Admin can remove a physical card for a user from the Android app	2 - Important	Done
Admin can view all administrative logs from the website and filter them	2 - Important	Done
All administrative actions should be logged	2 - Important	In progress (all implemented actions are logged)
User having the "Accept Payment" permission can create a payment from the Android app and scan a user card to validate the payment	2 - Important	Done
User can change its password	3 - Normal	Done
User can send money to another user from the Android app	3 - Normal	Done
Admin can undo a user batch import	3 - Normal	Done
Admin can delete areas from the website	3 - Normal	Done
Admin can delete rooms from the website	3 - Normal	Done
Admin can change permissions of users from the website	3 - Normal	Done
Admin can add money to a user account	3 - Normal	Done
User can view all its borrowed books on the books page	4- Nice to have	Not started

User story name	**Priority**	**Status as of July 11th 2018**
All accesses should be logged	4- Nice to have	Not started
User can add money to its account using its credit card (using Stripe)	4- Nice to have	Frontend in progress
User can create a virtual card from the application and use it	4- Nice to have	Placeholder and navigation on Android only
User can request access to a room from the website	4- Nice to have	Not started
Admin can delegate admin rights for an area to a user	4- Nice to have	Not started
Admin can view and mark as done room access requests	4- Nice to have	Not started
Auditor should be able to view access logs	4- Nice to have	Not started
Auditor should be able to view all payments logs	4- Nice to have	Not started
User having the "can invite" permission can create a temporary user from the Android app and the website	4- Nice to have	Not started
User having admin rights for an area can give access to an user to a room in that area (start/end date and start/end hour)	4- Nice to have	Not started
User having admin rights for an area can view and mark as done room access requests for that area	4- Nice to have	Not started
Onboarding setup process in web frontend to create first user when no users exists in the DB	4- Nice to have	Not started

User story name	Priority	Status as of July 11th 2018
Ability to connect to LDAP (Active Directory for example) for users and authentication	4- Nice to have	Not started
Librarian can create a loan for a book for a user from the Android app	5 - Not relevant	Removed
Librarian can view current bookings for all users from the website and Android app and mark the borrowing as Complete (all history on website)	5 - Not relevant	Removed

Table 2.1: User stories

2.5 Technological choices

2.5.1 The data and backend

For the data and backend part of the project, we decided to build a unique backend for both the web and mobile clients. We would need a database to store the data of our models and a service to access to access that data. The easier solution would be to build a REST API in front of a database. To do this, we saw two main possibilities: a PHP REST API coupled with a SQL Database or a Node.JS Express REST API coupled with a Mongo DB. While we could mix these and use a SQL Database with Node.JS, we wanted to use the better affinity possible between the two. With the increase in popularity in Javascript based backends, we decided to rule out the PHP option and keep the Express/Mongo option. This is also due to the fact that PHP feels dated and does not provide a clear structure unless you use a PHP Framework to complement it.

The second solution that was suggested was to use an object syncing platform to directly synchronise data with all of our clients. One of these platforms is *Realm Platform*[4] and we decided to study the advantages of it for our project and decide if we should use it over the Express/Mongo option.

[4] `https://realm.io/products/realm-platform`

2.5.1.1 Realm vs. Express-MongoDB

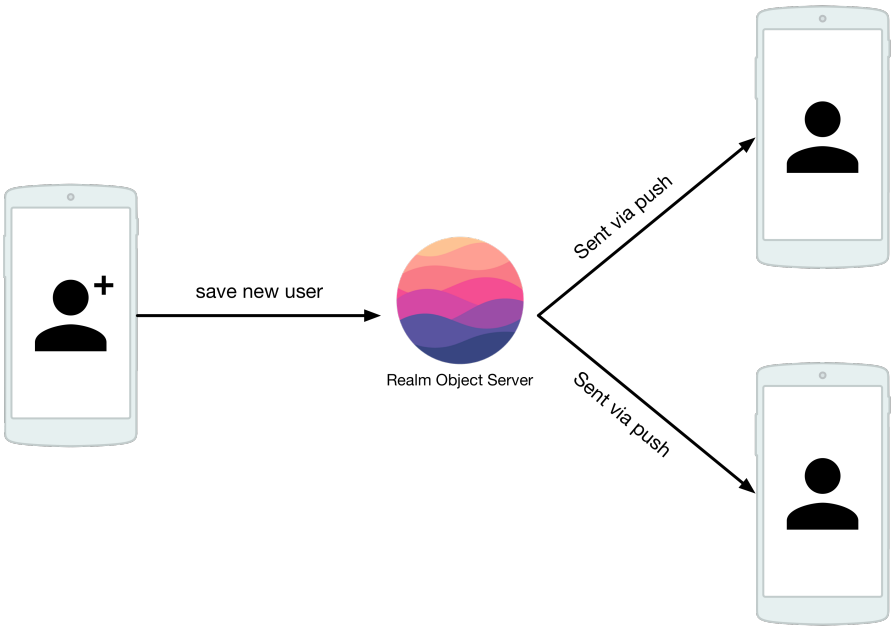

Figure 9: Saving an object in Realm

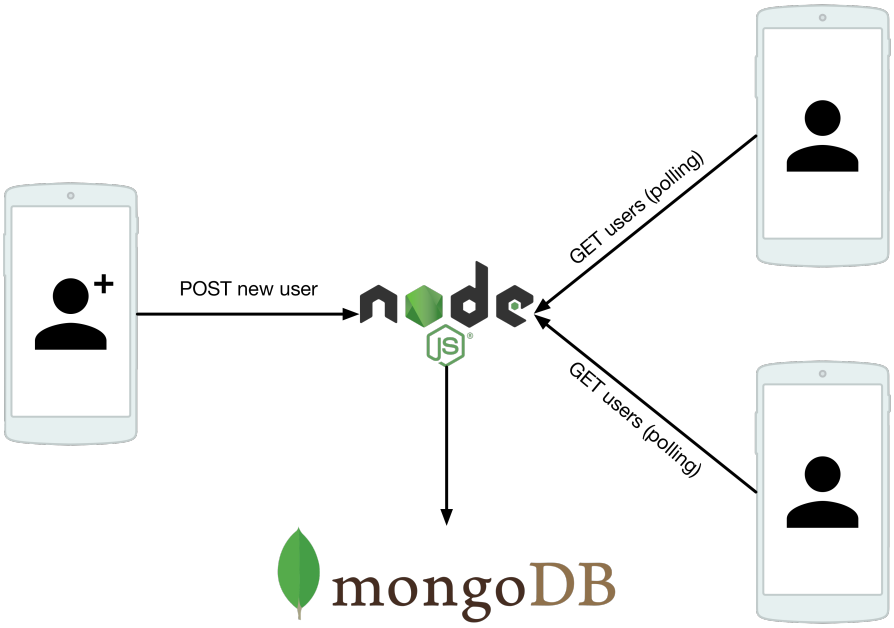

Figure 10: Saving an object in Express/Mongo

The Realm platform integrates the database directly within an object server that takes care of providing access and sending data to all the clients connected to it. This is possible thanks to the Realm runtime that you need to integrate in your client-side application and will replicate the database server on your local client.

On the contrary, using the Express/Mongo approach, we have the Mongo database and we built a REST API in front of it to provide access to the data. We do not need any particular runtime on the client-side and we can just handle data how we want.

As we can see in figures 9 and 10, when we create a new object on a client in Realm and save it locally, it is directly sent to the server and then pushed to other devices without needing any action. On the Express/Mongo side, you save the new object to the server and then other clients can retrieve it.

2.5.1.1.1 Advantages of Realm The advantages of Realm apply mainly on the mobile front. Realm is built to the ground up for the constrained resources and connectivity of mobile devices. It is offline first and will cache changes to be sent to the Realm server when the connection is back. Moreover, the push synchronisation of data back to the clients is the most efficient way to retrieve new data as opposed to polling data from a server at a regular interval.

2.5.1.1.2 Only for specific use cases The problem with Realm is that, beyond mobile, it is not very appropriate for our project. There are many pain points and small decisions that were taken by the Realm team that impose a way of doing things that may not be compatible with every project.

For example, one of these problems is the user authentication part and the user collection. In Realm, a user is not an object like others. It is a special object that is also used to connect to the Realm engine to handle sync. So you are limited to using Realm authentication to use the platform. Moreover, there is no way for a user with admin rights to retrieve a list of users on the client-side. This data is not accessible, unless you use the Realm Studio GUI client.

Another point is that, since you directly interact with the database/object server on the client-side, the code can easily be changed to access other people's information unlike a REST API where access to the database is regulated by the API. To avoid this, Realm has a system of permissions that are assigned to saved objects, but you need to assign the permissions every time you create a new object. Admins do not have direct access to everything, so you also have to add a permission for every admin every time. We wouldn't even imagine trying to add multiple set of roles here, it would be too cumbersome to handle.

Finally, the final point that made us abandon Realm for this project is the lack of clear solution to use it on the web. You have two ways to access the data from a web client. Either activate the GraphQL integration and write GraphQL queries to interact with the data from the web frontend. Or use the Node.JS Realm runtime to connect to the object server and serve as an API to access the data. So, even if we used Realm, we would still need to write an API. We're better off just going with the Express/Mongo option in that case.

It is a shame because the product provide real advantages on mobile that would take time to implement with a custom data solution, but it just isn't flexible enough and is really mobile-first while the web seems a bit like an after-thought for them. Even in their marketing material, they only present mobile-only applications such as chat or photo-sharing applications.

2.5.2 Mobile application

We needed to create a mobile client to access the information and status of a users' data from a smartphone as well as complete some administrative tasks.

We decided to build an Android application as well as coding the website to be responsive so it can be used on any smartphone. For the standard user, the mobile application is the one to use as it will provide access to all information, sync data offline and enable payments to other users. For the administrators, the mobile application will allow them to manage physical cards for users and for the rest of the administration, they can use the website on their smartphone.

2.5.2.1 Why the need for an application?

While we could have achieved most of the functionality with a web application, there are still some elements that require building a full mobile application. Mainly, access to device features such as NFC. With a mobile web app, we cannot have access to the NFC chip on the smartphone and thus we cannot manage physical cards or make payments through the app. This is also the reason why we only built an Android app and not an iOS one. The API for NFC on iOS[7] is basic and was released in 2017, it does not allow writing on NFC tags[5] and only supported a subset of the formats of the Android API.

Apart from this limitation, all the other functionality could have been incorporated in a web application, for example by making a Progressive Web App (PWA). PWA are an hybrid between a mobile application and a web application and they run on desktop and mobile operating systems. Their main advantages compared to traditional web app is that they can be "installed" on the device and act like a normal application and that they generally store data offline and can sync data in the background through the user of Service Workers.

[5] Unless you are a big company partnering directly with Apple

2.5.2.2 Why go native?

We could have also found a middle ground between the mobile web application and the native web application by using a mobile application framework such as Ionic or React Native to build our mobile application using web technologies. This would have allowed us to reuse some components built for the web in our mobile application.

We decided to opt out of that route and build our application using native Android technologies to better take advantages of the cutting-edge features announced during Google I/O 2018 [6] aiming to modernise and improve the Android development workflow (see section 3.1).

2.5.3 Web Frontend

Finally, we opted to use a frontend framework to provide a good structure to our client-side web application instead of relying on static custom HTML/CSS/JS.

The three main frameworks available at the moment are Angular[7], React and Vue.

Vue is the youngest of the three and, while it is already growing a big community, its documentation is not as complete as the two others and its also growing rapidly which means that some of our code might become obsolete rapidly.

React and Angular are more mature, with Angular being the oldest of the three and are both baked by big companies: Facebook for React and Google for Angular. The biggest difference between the two is that React is Javascript-based and Angular is Typescript-based. We decided then to use Angular to take advantage of the typed nature of Typescript as well as to use Typescript for the first time in a project.

[6] Google I/O is the annual Google developer conference held in June for developers using Google technologies (Chrome, web, Android, iOS,...)

[7] There is a distinction to be made between AngularJS (Javascript) and Angular.io (Typescript). While both coexist, you should use Angular.io as it is the version that is developed going forward. In this document, we will be talking about Angular.io when mentioning Angular

Chapter 3

Technologies

3.1 Developing for Android

Android is a mobile operating system developed by Google and launched in September 2008. The core of Android is open source and is known as the *Android Open Source Program (AOSP)*. But nobody really knows or uses the AOSP version of Android, every phone manufacturer, even Google, forks the AOSP version to build its own for their devices. What is often referred as "pure Android" is the Google implementation of Android on their Nexus and Pixel lines of smartphones. Android runs on all kinds of platforms, from mobile phone to watches, as well as tablets and laptops.

A SDK and an IDE are provided for developing Android applications. Those applications are written mainly in Java, with the ability to write native code in C++ using the Android NDK[1] and interact with the Java code using JNI. The support for a second language named Kotlin was announced at Google I/O 2017 (see section 3.1.2). Applications run in a virtual environment on Android called ART. You can think of ART as the equivalent of the JVM for desktop Java development.

3.1.1 Android Fragmentation

One of the biggest problems for developers is the fragmentation of OS versions running on Android devices. As of May 2018, only 62.3% of devices were running Android Marshmallow (version 6) or later. Android Marshmallow was released in 2015, meaning that 37.7% of devices were running software that was more than 3 years old without security updates or new features (see table 3.1.1).

[1] This is used for example to use some C++ libraries, like OpenCV.

Year	Version Name	Usage of this version	Usage of this version or later
2010	Gingerbread	0.3%	100.0%
2011	Ice Cream Sandwich	0.4%	99.7%
2012	Jelly Bean	4.3%	99.3%
2013	Kit Kat	10.3%	95.0%
2014	Lollipop	22.4%	84.7%
2015	Marshmallow	25.5%	62.3%
2016	Nougat	31.1%	36.8%
2017	Oreo	5.7%	5.7%

Table 3.1: Android OS Fragmentation (May 2018)[8]

3.1.1.1 The problem with updates

This problem is inherent to the very nature of the Android operating system. Each phone manufacturer, or OEM, can fork its own version of Android and integrate its own skin and set of apps on top of it. When a new version of the operating system is released by Google, they can't just start using it. They have to first adapt and test all their apps and customisation with the new version before it can ship to the devices. This is a time (and by extension money) consuming process and many OEMs just don't care about maintaining their devices for more than one or two years. To make matters worse, in certain countries, such as the United States, mobile carriers have to validate and apply their own custom apps and settings on top of the OS, adding to the time needed to validate an update and causing a supplementary potential roadblock to the release of an update.

3.1.1.2 What Google is doing about this?

Google has in the recent years taken different actions to ensure that most of the devices run safe and up-to-date software on them without depending on the OEMs willingness to maintain their products.

3.1.1.2.1 Updating core apps through the Play Store One of the most successful changes to Android in recent years has been to slowly move all Android core applications that are present on every Android device[2] to the Google Play Store. These apps include for example Gmail, Google Calendar, the browser (AOSP browser and

[2]Actually, not every Android device. Some Asian markets, mainly China don't have access to these apps because they don't have access to any Google services. This is an edge case that won't be discussed in this document.

Google Chrome) and even apps like the Phone Dialer and Contacts apps. This move to the Play Store allows Google to update these applications more frequently without needing a full operating system update. While it may be seen as a necessary evil at first, because it is the only way for them to update these apps if the OEM don't apply operating system updates, it is actually a very useful move because it allows for faster iteration on these applications and quicker response for bug fixes.

If we compare this to the other major mobile operating system, iOS, all main applications on the platform are bundled with the OS. So, if Apple needs to update the Safari browser to support a new web API they have to release a full OS version and push it to all their devices instead of just updating the application that needs an update as Google would do on Android.

3.1.1.2.2 Android Support Library

3.1.1.2.3 Project Treble

3.1.1.2.4 Security updates

3.1.2 Kotlin

As discussed in section 3.1, Kotlin has become a primary Android language in 2017. After having used this language for a first Android project last year, I decided to build all my subsequent Android application in Kotlin. The simplifications and reductions in code length provided by this language compared to Java make developing Android applications more enjoyable. It also helps avoid many runtime errors by catching many error-prone scenarios at compile time. In this section, we will go in further details in some of the advantages provided by Kotlin and how Google is encouraging developers to use Kotlin by introducing new Kotlin-specific features in the Android SDK.

3.1.2.1 Full interoperability with Java

First of all, the fact that Kotlin is fully compatible with Java and runs on the Java Virtual Machine (JVM) allows it to be easily integrated in an existing project or interact with existing Java libraries. You can just start writing classes in Kotlin and call existing Java classes from it and vice-versa. Then, when you want to convert Java code to Kotlin, Android Studio integrates a tool to convert a file to Kotlin almost perfectly. There are just some small changes to write afterwards, for example concerning the conversions of variables to constants. The IDE will even suggest converting Java code pasted inside a Kotlin file to the Kotlin version of the code.

3.1.2.2 Data classes

Java is often defined by its detractors as a very verbose language, requiring to write a lot of repetitive boilerplate code[3] for simple tasks. One of these tasks is the creation of classes with accessors and mutators for some of the class fields and overriding `equals` and `toString` methods. In Object Oriented Programming, we often have to write a lot of small classes just to match the Models in our applications. These are often referred as Beans or POJOs[4]. Kotlin introduces the concept of data classes[10] to simplify the implementation of these type of classes. By using a data class, you get "for free" an accessor (and mutator) for each field of the class, a correct overriding of the `equals` method, an overriding of the `toString` method listing the values of all fields in the instance and a `copy` method corresponding to a copy constructor in Java. For example, if we take a simple person class in Java:

```
class Person {
  private String firstName;
  private String lastName;
  private int age;

  public Person (String firstName, String lastName, int age) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
  }

  public Person (Person other) {
    this.firstName = other.firstName;
    this.lastName = other.lastName;
    this.age = other.age;
  }


  public String getFirstName () {
    return this.firstName;
  }

  public String getLastName () {
    return this.lastName;
  }

  public int getAge () {
```

[3]Boilerplate code or boilerplate refers to sections of code that have to be included in many places with little or no alteration[9]

[4]Plain Old Java Object

```
28      return this.age;
29    }
30
31    @Override
32    public boolean equals (Object o) {
33      if (this == o) return true;
34      if (o == null || getClass() != o.getClass()) return false;
35      Person person = (Person) o;
36      return age == person.age && firstName.equals(person.firstName) &&
        ↪  lastName.equals(person.lastName);
37    }
38
39    @Override
40    public String toString () {
41      return "Person{" +
42          "firstName='" + firstName + '\'' +
43          ", lastName='" + lastName + '\'' +
44          ", age=" + age +
45          '}';
46    }
47  }
```

And then the same class written using data classes in Kotlin:

```
1  data class Person(val firstName: String, val lastName: String, val
   ↪  age: Int)
```

3.1.2.3 Constants first

As in Java, you can define variables and constants in Kotlin but, in the later you will be more inclined to use constants because you can use them in more cases and the compiler suggests using constants when it detects that a variable is never assigned a new value. To use a constant, you use the keyword `val` instead of `var`.

One example of a case where you can use a constant in Kotlin but not in Java is when using blocks such as try/catch or control flow blocks. In Kotlin, these blocks return a value so you can assign the value returned by the block to a constant. The value returned by a block is the last line of the block or the last line of the taken branch of a control flow block.

For example, let's say that we are parsing a `String` as an `Integer` and we want to check if the value is between a given range. When parsing, an Exception can be returned if

the value is not a number, so we have to be aware of that.

In Kotlin, we can directly assign the value of the whole try/catch block to the constant:

```kotlin
fun main(args: Array<String>) {
    val ageInput = "Not a number string"

    // isAdult can be a constant,
    // without needing a initial temporary assignment
    val isAdult = try {
        val age = ageInput.toInt()
        when(age){
            in 18..99 -> true
            else -> false
        }
    }catch (exception: NumberFormatException){
        false
    }

    println(isAdult)
}
```

However, in Java, we have to define a variable first and then we have to reassign it according to the branch we are in:

```java
public class ConstantsDemo {
  public static void main (String[] args) {
    String ageInput = "Not a number string";

    //isAdult can't be final
    boolean isAdult = false;

    try{
      final int age = Integer.valueOf(ageInput);
      if (age > 18 && age < 99){
        isAdult = true;
      }
    }catch (NumberFormatException e){
      isAdult = false;
    }

    System.out.println(isAdult);
  }
```

```
}
```

3.1.2.4 Class extensions

Another advantage of Kotlin is the ability to write class extensions. This is useful to add methods to classes present in libraries for example and is used extensively as part of the Android KTX project (see section 3.1.4.1).

For example, if we are using the `ByteArray` class and we want to convert the value to a hexadecimal `String`, we can write a class extension method or property for the `ByteArray` class that we can then call on any instance of the class.
This is done in this way:

```
// This creates an extension method
fun ByteArray.toHex(): String {
    val sb = StringBuilder()
    this.forEach {
        sb.append(String.format("%02x", it))
    }
    return sb.toString()
}

// This creates an extension method that we can read
val ByteArray.hexString: String
    get() {
        val sb = StringBuilder()
        this.forEach {
            sb.append(String.format("%02x", it))
        }
        return sb.toString()
    }
```

And can then be called like any methods or properties:

```
fun main(args: Array<String>) {
    val bytes = byteArrayOf(202.toByte(), 254.toByte()) //This is
    ↪  0xCAFE

    println(bytes.hexString) //Displays "cafe"
    println(bytes.toHex()) //Displays "cafe"
}
```

3.1.2.5 Null safety

In terms of safety, Kotlin introduces concepts already seen in other languages such as Scala or Swift but not present in Java, mainly around the handling of `null` values[11].

In Java, if a function returns a `String` or another `Object`, it can also return a `null` value and if we forget to check if the value is not `null` and try to access it, the program will throw a `NullPointerException` at runtime and it will certainly crash. Runtime errors are unexpected and can happen at anytime so they should be avoided at any cost.

In Kotlin, they introduce the concept of optional types, denoted by a `?`. This allows for compile time check of `null` values risks. First of all, a function that returns a `String` can't return a `null` value. To return a `null` value you specifically have to return an optional type, in this case a `String?`. This optional type can only be accessed after checking if it contains a non `null` value. After the check, it will be *smart-casted* as `String` in the corresponding code block. The program will not compile if an optional type is used without checking if it contains a value first which avoids the possibility of having `NullPointerException` at runtime[5]

For example, an usage of the optional types is a function that finds an element in a list and returns the index of the element in the list. If the element is not contained in the list, it will return a `null` value. The function should then return a `Int?` and this value can only be used after specifically checking that the value is not `null`:

```
fun findInList(list: List<String>, element: String): Int? {
    for ((index, value) in list.withIndex()) {
        if (value == element) {
            return index
        }
    }
    return null
}


fun main(args: Array<String>) {
    val list = listOf("never", "gonna", "give", "you", "up")

    val index = findInList(list, "never") //index is an Int?
```

[5]This is valid for Kotlin code. When interacting with existing Java code, the Java code can use annotations `@Nullable` and `@NotNull` to mimic optional types but they are not required annotations so you can still run into problems depending on the Java library. An error could also happen if we bypass the `null` safety check by using the *not-null assertion operator*, see paragraph 3.1.2.5.1

```
15
16      //println(list[index]) //This wouldn't compile because no check
17
18      if (index != null){
19          println(list[index]) // We have checked so here index is an
            ↪ Int
20      }else{
21          println("Not found")
22      }
23  }
```

3.1.2.5.1 Optionals-related operators Kotlin also introduces a number of specific operators to handle optional types:

- The *not-null assertion operator*: This operator is `!!` and is used to tell the compiler that we unwrap the value from an optional without checking if it contains a value first. This is the *living dangerously* operator.
- The *safe call operator*: This operator is `?` and can be used when trying to access members and methods for optional types instances. It will return an optional type as well, for example if we have a `user: User?` and we want to get its name, we can use `val name = user?.name` and this will return the name of the user if the user is not `null` and otherwise will return `null`.
- The *Elvis operator*: This operator is `?:` and is used to indicate a value to use if the value is `null`. For example, with the user's name before, `val name = user?.name ?: "Paul"`. The "Paul" name will be used if the expression returns a `null` value

3.1.3 NFC on Android

3.1.4 Android Jetpack

Android Jetpack is a set of tools and libraries designed to improve the quality of Android applications by providing guidelines and code to implement common behaviours. Initially launched as *Android Architecture Components* at Google I/O 2017, it has been renamed in 2018 and gained new features. The main goals of Android Jetpack are to accelerate development by providing adaptable common features, to help with tedious activities such as background work and lifecycle management that require a lot of boilerplate code[9] and to improve the quality of applications by providing modern and robust core libraries.[12]. You do not have to use Android Jetpack libraries but they can be seen as a set of good practices and tools to help development.

3.1.4.1 Android KTX

Android KTX is a Kotlin-specific part of Android Jetpack. It provides Kotlin extensions for Android and Jetpack libraries to take advantage of Kotlin features and make code more concise[13]. It is composed of different modules, each related to a library or use case on Android. KTX is also open source and actively developed on GitHub[14]. You can contribute to it by creating new extensions or just suggesting them to the development team.

An example of Kotlin features used in KTX is the improvement to the usage of Lambdas and anonymous functions as well as the usage of these functions to avoid chaining calls.

For example, when replacing a fragment in a view, you can use a block with KTX in which you write the replacement operation and the block will begin and commit the transaction automatically:

```
1  //Without Android KTX
2  supportFragmentManager
3      .beginTransaction()
4      .replace(R.id.my_fragment_container, myFragment, FRAGMENT_TAG)
5      .commitAllowingStateLoss()
6
7  //With Android KTX
8  supportFragmentManager.transaction(allowStateLoss = true) {
9      replace(R.id.my_fragment_container, myFragment, FRAGMENT_TAG)
10 }
```

Similarly when working with a SQLite Database and writing a transaction, the KTX version will take care of catching the exception and cancelling the commit if an exception is raised:

```
1  //Without Android KTX
2  db.beginTransaction()
3  try {
4      // insert data
5      db.setTransactionSuccessful()
6  } finally {
7      db.endTransaction()
8  }
9
10 //With Android KTX
11 db.transaction {
12     // insert data
```

```
13 }
```

3.1.4.2 Room - Data persistence

There are three main ways to store persistent data in an Android application. First, you can use files, either user accessible files or files only available from the app code. Secondly, you can use `SharedPreferences` to store key/value pairs. And finally, you can use a database. Databases on Android are built on SQLite and have always been the best way to store data on Android. The problem though was that the code to write to use databases was too long and complex and you could easily run into errors due to the usage of bad practices or wrong SQL syntax for example. Moreover, you always had to parse the data returned from the database into models and vice-versa so it always felt like you had to do the work twice.

With Room[15], part of the Architecture section of Android Jetpack, the goal is to simplify the usage of SQLite databases by providing an abstraction layer on top of it. Used in conjunction with ViewModel and LiveData, it simplifies the handling and presentation of data on Android.

As seen in figure 11, the application interacts with the Room Database to get an instance of Data Access Objects (DAO) and retrieve Entities. All of these are generated by the library, we just have to provide simple ”configuration” classes and interfaces.

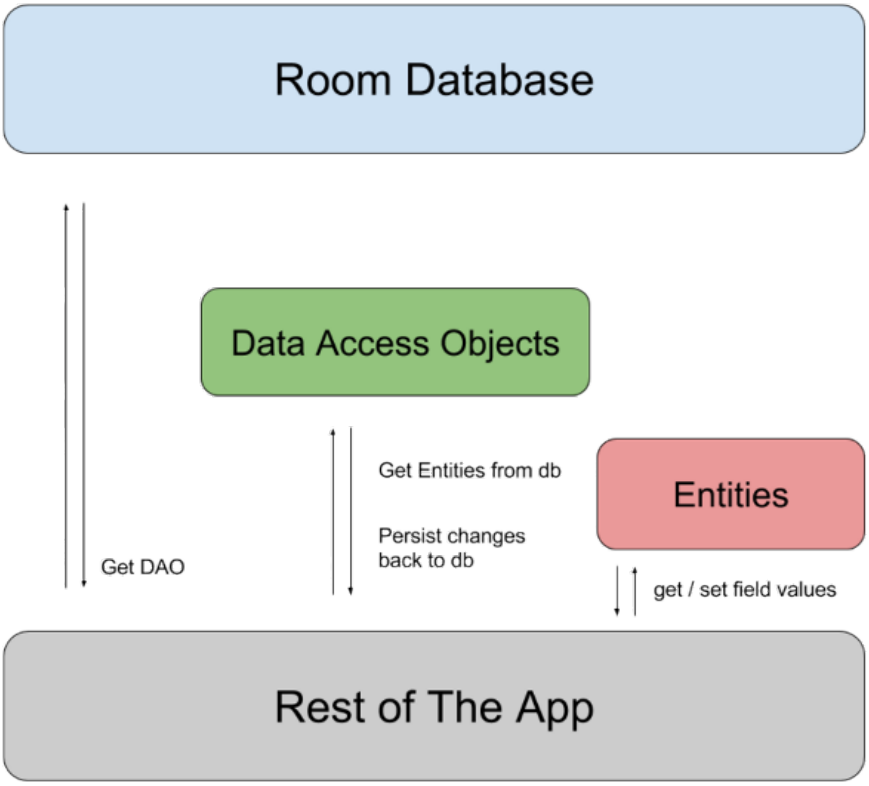

Figure 11: Room architecture[16]

The Entity[17] is the model that we are saving in the database. The advantage here is that we do not need to build a model and a separate table, we can just annotate our model with `@Entity` and other annotations to define the database table automatically. For example, a `User` entity is just a data class in Kotlin:

```
import android.arch.persistence.room.Entity
import android.arch.persistence.room.PrimaryKey

@Entity
data class User(
        @PrimaryKey val id: String,
        val name: String,
        val email: String,
        val isAdmin: Boolean,
        val cardId: String?,
        val virtualCard: String?,
        var balance: Double?,
        val isLibrarian: Boolean,
```

```
        val acceptsPayments: Boolean,
        val canInvite: Boolean,
        val isAuditor: Boolean
)
```

Then, we have to write a DAO[18] interface for the entity to create methods to interact with the entity in the database. There are methods automatically created for us, such as insertion and deletion of single entries. Queries returning elements from the database have to be written in SQL, but the IDE provides completion and there is compile time check of SQL syntax with table and column names. Finally, we can easily ask for returning LiveData for a query simply by specifying in the signature function.
For example, for the `User` DAO, we can just wrap all the return values from `SELECT` queries in `LiveData` and we will get them in that format:

```
import android.arch.lifecycle.LiveData
import android.arch.persistence.room.*
import ch.maximelovino.pockethepia.data.models.User


@Dao
interface UserDao {
    @Query("SELECT * FROM user")
    fun getAll(): LiveData<List<User>>

    @Query("SELECT * FROM user WHERE id LIKE :id")
    fun findById(id: String): LiveData<User>

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    fun insert(vararg users: User)

    @Delete
    fun delete(user: User)

    @Query("DELETE FROM user")
    fun deleteAll()
}
```

Finally, you can then write the Database[19] class by providing an array of the entities and creating an abstract fun to retrieve each DAO. It is also suggested to make the Database class a Singleton[16]:

```kotlin
import android.arch.persistence.room.Database
import android.arch.persistence.room.Room
import android.arch.persistence.room.RoomDatabase
import android.content.Context
import ch.maximelovino.pockethepia.data.dao.UserDao
import ch.maximelovino.pockethepia.data.models.User


@Database(entities = arrayOf(User::class), version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao

    companion object {
        private var db: AppDatabase? = null

        fun getInstance(context: Context): AppDatabase {
            return if (db != null) {
                db!!
            } else {
                db = Room.databaseBuilder(context,
                ↪  AppDatabase::class.java, "db").build()
                db!!
            }
        }
    }
}
```

3.1.4.2.1 ViewModel The ViewModel[20] bridges the gap between the data and the UI data in a lifecycle aware way. As opposed to storing data in variables in an activity, data in a ViewModel will remain persistent when configuration changes, for example when rotating the screen.
It is suggested to also implement a Repository behind the ViewModel to abstract the source of the data, database or network for example. In the Repository, we can also write wrapper around insertion tasks to run them on separate threads because you can't run database operations on the main thread[6].
The user Repository and ViewModel look like this:

```kotlin
import android.arch.lifecycle.AndroidViewModel
import android.annotation.SuppressLint
```

[6]Queries returning LiveData already run on a background thread, due to the LiveData returning value asynchronously

```
import android.app.Application
import android.os.AsyncTask
import ch.maximelovino.pockethepia.data.AppDatabase
import ch.maximelovino.pockethepia.data.dao.UserDao
import ch.maximelovino.pockethepia.data.models.User

class UserRepository(application: Application) {
    private val db: AppDatabase =
    ↪ AppDatabase.getInstance(application.applicationContext)
    private val userDao = db.userDao()
    val allUsers = userDao.getAll()

    fun insert(user: User) {
        InsertAsyncTask(userDao).execute(user)
    }

    inner class InsertAsyncTask(val dao: UserDao) : AsyncTask<User,
    ↪ Void, Unit>() {
        override fun doInBackground(vararg users: User?) {
            val user = users[0] ?: return
            dao.insert(user)
        }
    }
}

class UserViewModel(application: Application) :
↪ AndroidViewModel(application) {
    private val repository: UserRepository =
    ↪ UserRepository(application)
    val users = this.repository.allUsers
}
```

3.1.4.2.2 LiveData LiveData[21] is an observable holder class that allows to react to data changes. The advantage of LiveData is that it is lifecycle aware, so you can observe the data by passing it an Activity or Fragment and will pause and resume accordingly when the Activity is paused or resumed. Otherwise, we could try refreshing the UI when the data changes and if the Activity is not in the foreground, the application would crash. LiveData allows us to always update our UI with the latest data, for example data from a Room Database.

You can just call the `observe` method on a LiveData instance and pass a lifecycle

component, in this case a Fragment and an Observer callback to handle the data.

```
val user = userDao.findById(id)

//Here "this" is a fragment, activity or other lifecycle component
user.observe(this, Observer {
  //"it" is the data
  if (it != null) {
    //The user data changed
    //update view for example
  }
})
```

3.1.4.2.3 Putting everything together When putting everything together, we can see the interactions in figure 12 with the ViewModel providing LiveData to the Fragment or Activity that can update the UI by observing it. The `ViewModelProviders.of()` is used to retrieve the requested ViewModel from the system.

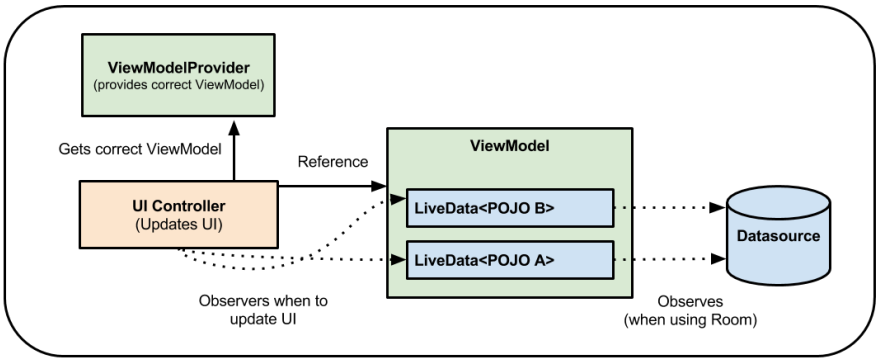

Figure 12: View model interactions[20]

If for example, we couple this with a RecyclerView and decide to get a list of all users from the database to display it, it gives us the following code:

```
override fun onCreateView(inflater: LayoutInflater, container:
 ↪  ViewGroup?, savedInstanceState: Bundle?): View? {
  // Inflate the layout for this fragment
  val view = inflater.inflate(R.layout.fragment_admin, container,
   ↪  false)

  viewAdapter = UserListAdapter(this.context!!)
```

```
6     val viewManager = LinearLayoutManager(this.context!!)
7     view.findViewById<RecyclerView>(R.id.users_recycler_view).apply {
8       layoutManager = viewManager
9       adapter = viewAdapter
10    }
11
12    usersViewModel =
      ↪  ViewModelProviders.of(this).get(UserViewModel::class.java)
13
14    usersViewModel.users.observe(this, Observer {
15      if (it != null)
16        viewAdapter.setData(it)
17    })
18
19    handleFabDisplay()
20    return view
21  }
```

3.1.4.3 Work Manager - Background jobs

3.1.4.4 Navigation

3.2 NoSQL Database - MongoDB

3.2.1 Storage as JSON/BSON documents

3.2.2 How to handle references

3.2.3 Mongoose

3.2.3.1 Schemas

3.2.3.2 Promises with Mongoose

3.2.3.3 Populating references

3.2.3.4 Hooks

3.3 NodeJS et Express

3.3.1 JavaScript ES6

3.3.2 NPM Modules

3.3.3 Promises

3.3.4 Middlewares

3.3.5 PassportJS

3.4 Angular 6

3.4.1 Single-page webapp

3.4.2 Dependencies injection

3.4.3 Architecture of an Angular Application

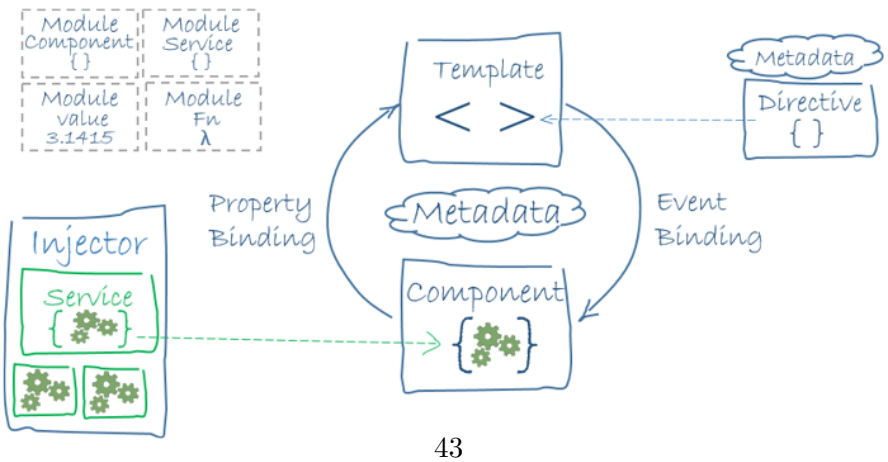

Figure 13: Architecture of an Angular application[22]

3.4.3.1 Components

3.4.3.1.1 Templates

3.4.3.1.2 Communication between components - Input/Output

3.4.3.2 Services

3.4.3.3 Routing and Guards

3.4.4 RxJS - Observables

3.4.5 Reactive Forms

3.4.6 TypeScript

3.4.7 Angular Material

3.5 Containers - Infrastructure as code

3.5.1 Docker

3.5.2 Orchestration avec Docker Compose

3.5.3 Google Cloud

3.5.3.1 Kubernetes

3.5.3.2 Google Container Registry

3.5.4 Azure

Chapter 4

Implémentation

4.1 Modules du projet

4.2 Déploiement

4.2.1 Architecture du projet

4.2.2 Proxy Nginx

4.3 Le backend commun

4.3.1 Authentification

4.3.2 Endpoints

4.3.3 Interaction avec la base de données

4.4 Le frontend Angular

4.4.1 Routing

4.4.2 Components

Chapter 5

Résultats

Chapter 6

Conclusion

6.1 Future works

Bibliography

[1] Oyster FAQs: how to use your card - visitlondon.com. https://www.visitlondon.com/traveller-information/getting-around-london/oyster-faqs/how-to-use-your-card. (Accessed on 23.02.2018).

[2] lady ada. About the NDEF Format | Adafruit PN532 RFID/NFC Breakout and Shield | Adafruit Learning System. https://learn.adafruit.com/adafruit-pn532-rfid-nfc/ndef, May 2015. (Accessed on 01.07.2018).

[3] NFC Forum. *NFC Text Record Type Definition Technical Specification*, June 2017. (Accessed on 07.01.2018).

[4] H. Alvestrand. Tags for the Identification of Languages. RFC 3066, RFC Editor, January 2001.

[5] Camipro | EPFL. https://camipro.epfl.ch/page-6801-en.html. (Accessed on 21.06.2018).

[6] EPFL — PocketCampus. https://pocketcampus.org/epfl-fr/#epfl-support-fr. (Accessed on 22.06.2018).

[7] Core NFC | Apple Developer Documentation. https://developer.apple.com/documentation/corenfc. (Accessed on 30.06.2018).

[8] Distribution dashboard | Android Developers. https://developer.android.com/about/dashboards/. (Accessed on 07.06.2018).

[9] Boilerplate code - Wikipedia. https://en.wikipedia.org/wiki/Boilerplate_code. (Accessed on 18.06.2018).

[10] Data Classes - Kotlin Programming Language. https://kotlinlang.org/docs/reference/data-classes.html. (Accessed on 18.06.2018).

[11] Null Safety - Kotlin Programming Language. https://kotlinlang.org/docs/reference/null-safety.html. (Accessed on 01.07.2018).

[12] Android Jetpack | Android Developers. https://developer.android.com/jetpack/. (Accessed on 02.07.2018).

[13] Android KTX | Android Developers. https://developer.android.com/kotlin/ktx. (Accessed on 01.07.2018).

[14] android/android-ktx: A set of Kotlin extensions for Android app development. https://github.com/android/android-ktx. (Accessed on 01.07.2018).

[15] Room Persistence Library | Android Developers. https://developer.android.com/topic/libraries/architecture/room. (Accessed on 02.07.2018).

[16] Save data in a local database using Room | Android Developers. https://developer.android.com/training/data-storage/room/. (Accessed on 02.07.2018).

[17] Defining data using Room entities | Android Developers. https://developer.android.com/training/data-storage/room/defining-data. (Accessed on 02.07.2018).

[18] Accessing data using Room DAOs | Android Developers. https://developer.android.com/training/data-storage/room/accessing-data. (Accessed on 02.07.2018).

[19] Database | Android Developers. https://developer.android.com/reference/android/arch/persistence/room/Database. (Accessed on 02.07.2018).

[20] ViewModel Overview | Android Developers. https://developer.android.com/topic/libraries/architecture/viewmodel. (Accessed on 24.06.2018).

[21] LiveData overview | Android Developers. https://developer.android.com/topic/libraries/architecture/livedata. (Accessed on 02.07.2018).

[22] Angular - Architecture overview. https://angular.io/guide/architecture. (Accessed on 23.06.2018).