POLYTECH GRENOBLE

Smart Self Service

Project Report

RICM4

Étudiants
Alicia ABONNENC
Gilles BONHOURE

Encadrants :
Didier DONSEZ
Jérôme MAISONNASSE

Avril 2017

Sommaire

1. Introduction

Smart Self Service aims to be a platform for lending objects by controlling connected lockers, making it possible to lend objects without having to meet up. Eventually, this platform could be used to lend objects only to friends and/or to earn prizes/points/money if you lend a lot of objects.
On this web application users can have 2 different roles : Regular user or Admin (or both).

A regular user can :

- **Drop off** an object of his own for other users to borrow.
- **Pick up** one of his own objects that he left and that is not reserved by another user.
- **Take off** or borrow an object that has been left by another user.
- **Bring back** an object he borrowed before.

All actions above require to unlock a locker. That is done by going to an **unlocking page** and entering a 4 digits code that is communicated to the user under the **actions** section. The access to this unlocked page is given by a QRcode, on the physical locker, that has to be scanned (using a smartphone for example).
Users don't have to be logged in to access to the unlock page but have to be to access any other page.

Admins can :

- **Add a locker** to the database
- **See all lockers** and informations linked to them such as pending actions and code to open them.
- **Manage users rights** to add admins or remove admin rights.

2. Technologies used

A. Web application

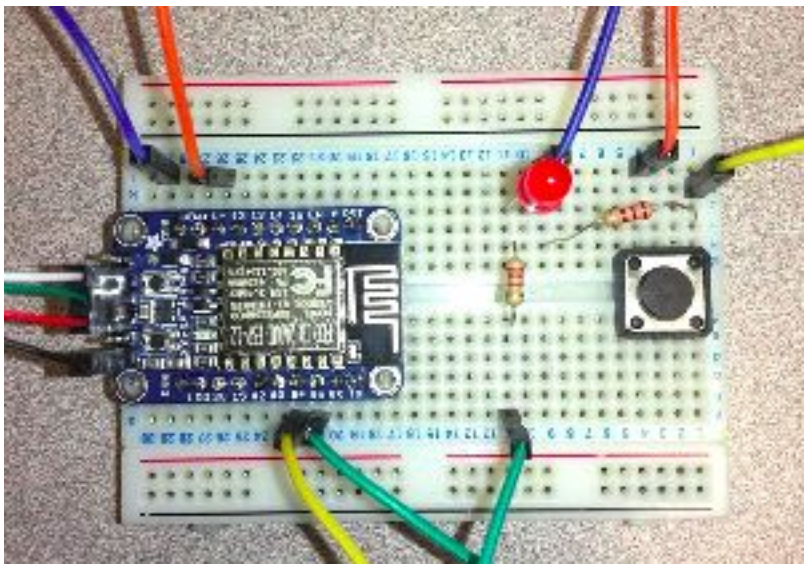
The web application was coded using **Meteor** (a MVVM framework) with Blaze. Meteor allowed us to have a real time, responsive web application. It uses **Mongo Database**, a non relational database, stocked in json collections.

We used the following packages to make the web app :

- ***iron-router***
  The package used to display different templates depending on the route (url).
- ***Accounts-password***
  Used to be able to sign up and login as a user. We did not use the Meteor accounts-base package because we added information to the user's database
- ***jayuda:flx-qrcode***, a simple package used to display an QR code from an URL
- ***reactive-var*** & ***session***, used to pass variables in between templates.
- ***twbs:bootstrap*** & ***tsega:bootstrap3-datetimepicker***
- ***http***, used to send messages to the Lockers' chips.

B. Lockers

To control the locker, we use a ESP8266 Wi-Fi chip. This chip can allow us to control the lockers (open, close the door) by sending order by aREST via HTTP.
Here is a photo of the circuit we made. The LED represents the door lock when it is lit up the lock is unlocked, otherwise it is locked. The button here represents the contact sensor when pressed, it means that the door is closed. (Allowing us to know when to activate the locker's lock).



The LED is connected to the pin 5 of the chip and the sensor is connected to the pin 4 of the chip.

However, during this project we could not sucessfully light up the LED because the aREST cloud service was down, and we were not able to deploy our own server on a local network.
(cf. Annexe 1)

3. Organization of the application

A. Overview

Meteor runs into a specific architecture. Here are some notions about it, prior to the explanation of our project organization :

- A **Route** is the simplest way to define a page/view in the app.
- A **Template** is a section of the webpage in HTML, can be included in any page by calling Route.render("template_name") in a javascript route definition. It is also possible to define variables and functions that can be called inside the HTML by implementing them inside the JS template functions like **helpers** or **rendered**.
  See more informations here : https://www.meteor.com/tutorials/blaze/templates

As any Meteor application, ours is mainly composed of two classic parts : the **Client** and the **Server**. The second one here only defines the access rights among the database. The Client includes a **main.js** file, linked to an html document which displays the generic application UI. The **helpers.js** file contains all the globals helpers accessible from any other classes. Finally, every file from the **router folder** is imported in the main. These files also import and create routes to go to all the UI **templates** defined in the **imports** folder.

B. Routing system and UI manager

(see annexe 2)

As previously explained, the UI package is called by a routing process. We chose to split the different templates into folders in order to organize them by theme. Each template defines a view/page in the web application.

The first part contains all the **general** pages of the web-app, those which only display informations (home, search,...). The second part is the **administration** one, where only admin can go and manage the various aspects of the website, like user rights and lockers. You can notice that the "locker" folder itself contains an admin part because it was more logical from our point of view to have every templates linked to lockers in this folder. Thereby, we have a **Locker** folder where the unlocking page (the one which is accesisble even when not logged-in) is defined and where other various display and/or creation views are defined.
The **Object** folder contains all the template which allow the **user** to drop off, pick up, bring back, take off, and manage the items he owns and/or borrowed. Finally, the **user** package implements a register and login form, and gives access to the pending actions of an user.
Many of these files import some "api" files to have access to the database collections.

C. Database

Concerning the database, we used **MongoDB** which is already implemented in Meteor. This technology is a **document-oriented database**. This means that we couldn't use the traditional class-architecture for relational database to design the data management. Instead we made collections which contained links between other collections. (see Annexe 3)

To compare it with a relational database, each "class" is represented with a **JSon** file, also know as a **collection**.

In our model, the three collections are the **Users**, the **Lockers** and the **Objects**. Ass you can see in the *Annexe 3's* Database UML we put an **history in**to every object in order to keep track of the journey of this object. On the same idea, the user collection possesses a list of **pending actions** to be able to know what the user has to do, and eventually display it to him as a reminder.

The **history** is particularly interesting for possible improvements, for as the app is now, it is not displayed to the user.

4. Difficulties encountered

Several elements lead us to not having the connection between the ESP8266 and the web application, also meaning not having a locker to open during this project.

First, we did not know what technologies to study in details before the 9th week of the project, for as in the beginning we were told, we were going to continue a project done last year by Phelma students. So we studied the technology they used (MQTT, Mosquito…), but we also self-learned how to code using Meteor, which was useful as our final product is a Meteor application.

Secondly, we were not able to have a functional ESP8266 chip before the last week of the project. We successfully communicated with the chip on the first day but after that the cloud of aREST was down. We think that we mostly lacked time, because by studying this technology more, we could have deployed our own server on a local network and would not have to use the cloud.

Lastly, we were not fixed on the scenarios until quite late in the project. At the beginning we thought about having a camera on the lockers that would scan QR codes and recognize them as orders and open lockers. Now, what our application does, is allowing users to unlock the lockers via a web-page, accessible by scanning a QR code printed on the lockers, by entering a 4 digits code.

5. Possible improvements

A. Connecting with lockers

On of the main thing that should be done if this project is continued is to communicate with the ESP8266 chip (via the cloud or not) to have interaction between our app and the lockers. Then to go further it would be good to have our own server, sending the orders to the lockers so that the down time is controlled by us.

B. Security

At the moment, every operation made on the database is done on the client's side. For security concerns, it would be way better to remove the **insecure** package in the meteor app and have every operations done on the meteor server.

Also, in order to have the information contained in the **Accounts.users** collection, we used a **Publish-Subscribe** system but did not remove the **auto-publish** package for we were not sure whether the other collections (Lockers and Objects) would still be accessible. So an improvement would be to remove this autopublish package and maintain the system working as now.

Moreover, there is no guarantee that, on creation of the account, the user really is who he says he is. One thing that could be done is verifying the email address that is given (there already is a "verified" field in the Database. Cf Annexe 3). And there is no restriction on the password that is given, adding rule to make it at least N characters and/or have at least digits and characters in it could help make the application better.

C. Expanding scenarios

First of all, an easy view that could be added to the object management functionality is to display the **history** of the object so that the user knows who borrowed his object. That would be helpful, if the object returns broken, to know who to blame.

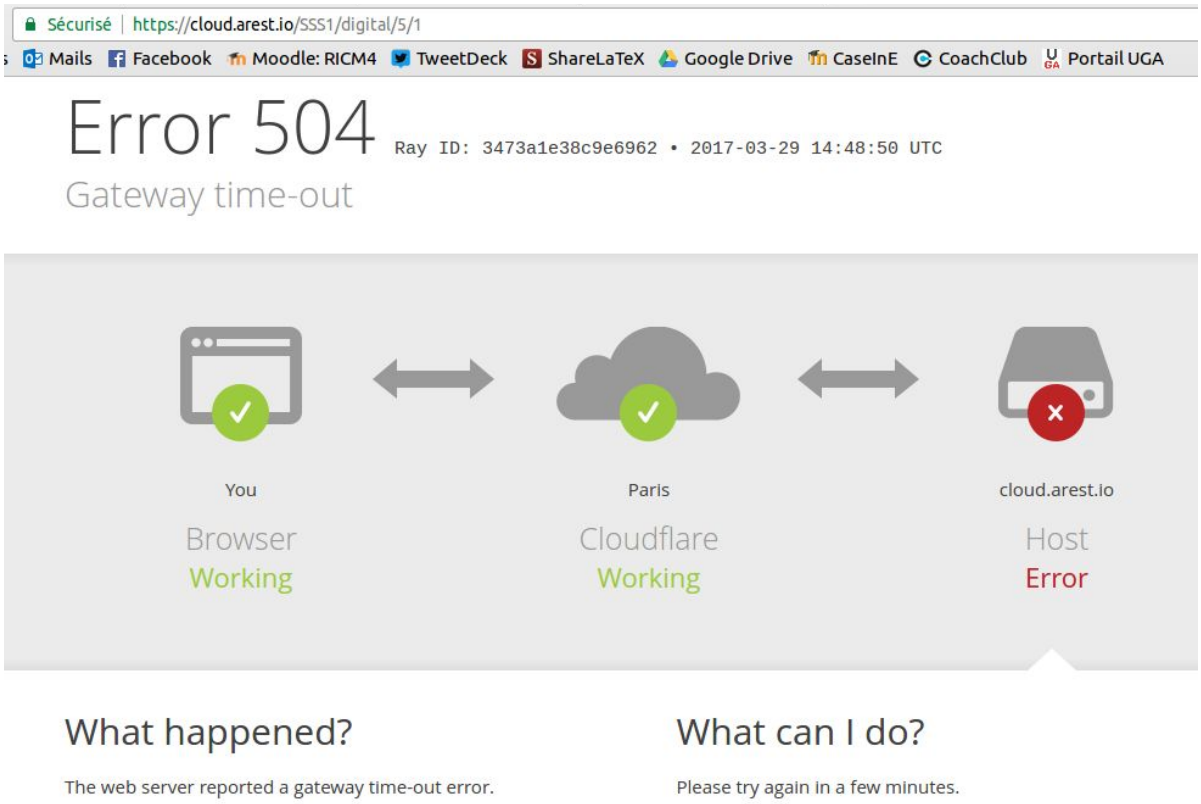
Then, at the moment, all objects are available for everyone to borrow as long as they are connected. One scenario that could be implemented is to have first a list of friends (and so a search function and why not more informations about users like pictures, bio...) then to be able to **restrict** the list of users who are able to interact with certain objects.

Another scenario would be to have review systems (to rate users who borrowed my items) and a mean to contact admins if there is a problem (eg. locker closed before I could put my object in it and I don't have the password anymore).

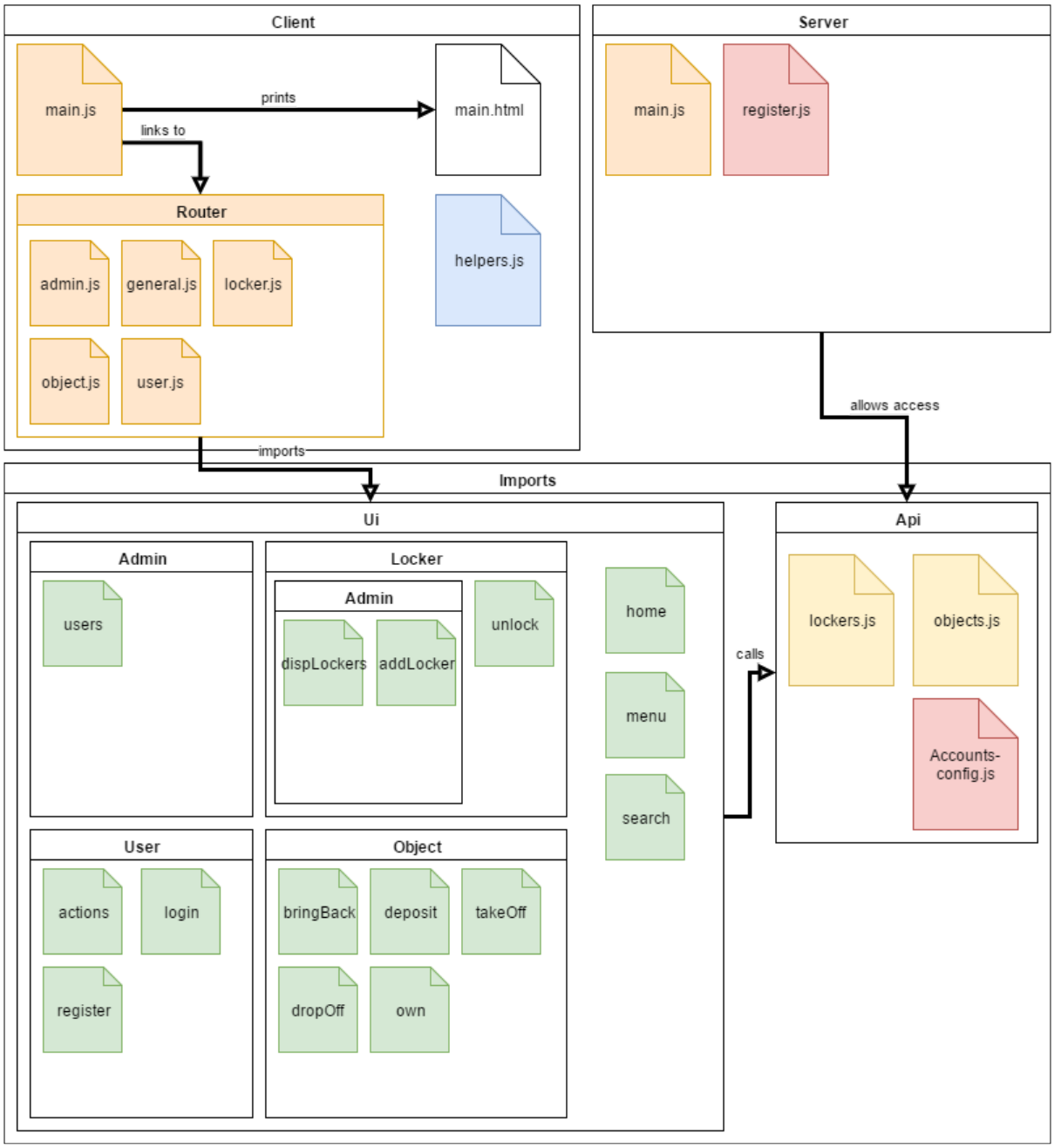
Also, this application could be extended with a selling scenario (meaning adding money into the system) and/or swapping scenario (meaning attributing value to objects). In both cases, it would be good to have another security concerning items because for now there is nothing else then trust ensuring a user its object will come back.

Annexes

Annexe 1 : Host is down, aREST



Annexe 2 : Application Structure



Annexe 3 : Database Structure