UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

Open Source Geospatial Health Symptoms Tracker on Android Using Extreme Programming Techniques

Dissertation presented by
Alexandre HAUET , Tanguy VAESSEN

for obtaining the Master's degree in
Computer Science

Supervisor(s)
Marc LAINEZ, Pierre SCHAUS

Reader(s)
Kim MENS, Nicolas LAURENT

Academic year 2016-2017

Abstract

During this thesis, we explored the benefits of *Extreme Programming*, *Continuous Delivery* and other linked methodologies through the development of a solution that aims to help patients track their symptoms when they happen and share them with their doctors to help refine their diagnosis. We learned about methodologies and practices that allow the production of good quality code. In our opinion this is something that is not enough taught at university where the final result of a project is often evaluated regardless of the quality of the code. The solution is composed of four components. First, a mobile application used by patients to record their symptoms and share them with their doctors. Then, a website used by doctors to consult reports of their patients. Next, a backend to collect symptoms, analyze data, send emails and more. Finally, a utility chatbot used to simulate patients and automatically deploy the backend whenever a new version was available. This master thesis made us experience the whole life cycle of an application, from setting up the deployment infrastructure to regularly pushing code to production.

Acknowledgments

We would like to thank our supervisor, Marc Lainez, for all his great pieces of advice, his support and his time during the past year.

Our thanks also go to Pierre Schaus, our co-supervisor, who helped us when we needed it.

We also thank Dr Coralie Bonnier for her collaboration, her time and her feedback during the different steps of our project, her advice was really useful and appreciated.

Then, we would like to thank Marina Evaristo for her help and the motivation she gave us.

Finally, we thank the people who helped us during the writing of this thesis: Apolline Deglasse and Corentin Mulders for their relevant corrections.

Contents

Chapter 1

Introduction

Motivation

Being students at a University, we realized that in most of the projects we carried out, only the output, the result of the program was evaluated. We received very little feedback on the quality of the code we wrote. How can we be sure that we are writing "good code"?

In response to this observation, we wanted to force ourselves to produce software of great quality (in term of code).

Nowadays software development methodologies are becoming more and more important. Words such as *Agile*, *Waterfall*, *Scrum*, ... are common in computer projects. There are a lot of different methodologies. That is why instead of trying them all we decided to focus on studying the advantages of some of them.

Context

Diagnosing a disease is something very complicated. The doctor does not always have all the information to perform the right diagnosis.

For the patient it is sometimes difficult to remember all the information that the doctor needs. Moreover, during a consultation, a patient is not always unbiased and might present a worse version of his symptoms because he feels depressed or a better version because humans tend to be optimistic and forget bad things.

For certain diseases taking notes of the appearance of symptoms is mandatory. Being precise and assiduous in note-taking can be really hard.

Therefore we developed a solution that makes note-taking easier, provides the ability to share symptoms with doctors and analyzes information coming from multiple users.

Roadmap

The solution that we propose is composed of different parts. A mobile application for the patient in which he can log his symptoms. With the authorization of the patient, the application records information about the different symptoms such as the geolocation, the date, etc. Thanks to these pieces of information, we can collect some additional data such as the weather conditions during the appearance of the symptom.

A second part is the backend to which all the occurrences of symptoms are sent. It exposes a REST API[1] and performs some asynchronous tasks to analyze symptoms.

[1] A RESTful API is an Application Programming Interface that uses HTTP requests to access and modify data. It follows the REST architecture style that defines some constraints (like *Stateless* and *Client-Server*) and properties (like *Scalability* and *Simplicity*): `https://en.wikipedia.org/wiki/Representational_state_transfer`

A web application is available for the doctors. On this website, they can view all the information about the appearances of the symptoms of their patients.

And finally, we use some data mining tools to analyze the data collected.

Approach

During the realization of a project from scratch, we applied the *Extreme Programming* methodology and other linked technologies. The goal was to see how it influences the project: does it improve the quality of our code, our productivity, etc?

We did this master thesis into two phases. During the first phase, we learned about methodologies and technologies: Extreme Programming [2], Test Driven Development [1], Continuous Delivery[8], ... During the second phase, we used these techniques and the tools associated with them to implement the project.

Deliverables

From the beginning, we decided to open source all of our code because we were going to use lots of open source technologies, our project might interest people working in the health sector and we were going to face issues other would too. Also, an open source approach forces us to get serious about *Extreme Programming*'s best practices.

All the work done during this master thesis is visible on the following public repositories: `https://github.com/GeoHealth`

The applications implemented during this thesis are available on the following links:

- **Mobile application**
  `https://play.google.com/store/apps/details?id=com.geohealth.happi`
- **REST API**
  Base URL: `https://api.happi-doctor.be`
  Definition: `https://geohealth.github.io/HAppi_backend`
- **Frontend doctors**
  Home page: `https://happi-doctor.be`
  Report example: `https://happi-doctor.be/#/report?token=3QknCrY7QxGLPwCyUuBizZtd&email=doctor%40mail.com`
- **LCM web interface**
  `https://api.happi-doctor.be/data_analysis/users_having_same_symptoms`
- **Global statistics on Kibana**
  `http://happi-static-kibana.getforge.io`

Document organization

In this document, we first present some theory about programming methodologies and explain how we have adapted them to fit our needs.

Then we present the solutions we have built by describing each project and how they work together. We also explain the difficulties we encountered and how we solved them.

After that we discuss testing and, how important it is to write good tests, how to know they are good and cover enough parts of a project and what are the different types of tests that exist.

We present briefly the tools we used. There are a lot of them because we covered the entire development process, from scratch to production.

Then we explain what improvements can be made: what could be added to make the solution better and what would we do differently after having done this thesis.

In the two last sections, we present our feedback for each sprint we have done and we explain how, with all the things we have learned and the experience we have acquired, we would manage a new project.

Finally, we conclude by explaining why we think that we are better developers now than we were before this thesis and how our work can be used to improve life and the health sector.

Chapter 2

Methodologies

In the software development world, there is a very diverse set of methodologies. There are methodologies for security, project management, development, ... The usage of these methodologies in a project is meant to improve the quality of the application, the quality of the code, the realization of the application, ... Previously a lot of projects did not succeed due to a lack of management, rigor... Thus specialists have defined techniques to avoid encountering these types of problems over and over again.

In this chapter, we present the methodologies that we used, with a brief introduction and how we adapted them to our project.

2.1 Agile

Agile was created as an alternative to the Waterfall methodology which was considered unfit in a context where software requirements tend to change very often. The term "Agile" regroups different methodologies based on some principles that are defined in an *Agile Manifesto*[3]. The four fundamental values of Agile are:

- Individuals and Interactions Over Processes and Tools
- Working Software Over Comprehensive Documentation
- Customer Collaboration Over Contract Negotiation
- Responding to Change Over Following a Plan

Agile methods are based on an iterative development cycle, incremental and adaptive.

2.1.1 Extreme programming

There are many methodologies following the four Agile core values described above. Extreme Programming (XP) is one of them. XP is more focused on the project implementation aspect than on the management part. The concept of XP is very simple: to push simple principles to the extreme. XP is based on the five following values: communication, simplicity, feedback, courage and respect. XP is committed to making a project more flexible and open to changes by following some principles and practices.

The values of XP are important but they are vague, thus they are redefined in "Principles" to become: rapid feedback, assume simplicity, incremental change, embracing change and quality work.

The goal of the "Practices" is to make programming more effective. But it should be kept in mind that some practices may or may not be useful depending on the context. Let us look at some practices we used in our master thesis from a theoretical point of view. These practices

come from the book *Extreme Programming Explained: Embrace Change*[2]. Then, we describe how we adapted them to match our needs.

Pair programming

Two programmers work together on the same workstation. One of the developers, called "the driver", holds the keyboard. He is responsible for writing the code. The second, called "the co-pilot", ensures that the other does not make mistakes and suggests changes when he considers them useful. The roles are often reversed, the driver becomes the co-pilot and vice versa. This technique is very useful because it allows the developers to be more attentive, to produce a code of better quality and especially to have an exchange between developers so they can learn new things.

Energized work

We must be efficient during the day, this is why we must avoid tiring us. After working fourteen hours in a row, during the next days our work is of lower quality due to fatigue. This is why it is preferable to work eight hours per day and take breaks. If we want to be more efficient, there are surely other solutions than increasing our hours of work. For example, programming during two hours when our mobile phones and email notifications are turned off.

Stories

We used stories to represent clients' needs. Efforts to achieve stories must be estimated (hours, points...).

Weekly Cycle

Plan the work one week at a time so that developers can focus on Friday as deadline. At the beginning of the week, review the past week and plan the stories that should be done this week. Split them in tasks and let the team members assign them to these tasks.

Ten-Minute Build

Automatically build the project and run the tests should take no more than ten minutes. If it takes longer, developers do it less frequently, leading to later feedback. And we know that having a feedback on something we worked on an hour ago is frustrating because we already have moved on to the next task.

Continuous Integration

Continuous Integration (CI) is the practice of frequently integrating the changes of all developers. The goal is to provide quick feedback to developers so they can fix errors as soon as they appear, i.e. when they are cheap to fix. Integrating the changes means combining them, building the project and running the tests. This integration should be done as frequently as possible to discover errors as soon as possible.

Two types of CI exist: asynchronous and synchronous. In asynchronous CI, developers move on to the next task and receive a feedback from the automated integration process later. The disadvantage is the same as the one mentioned in *Ten-Minute Build*: when receiving this late feedback, developers might need to dive into the previous task. In synchronous CI, developers wait for the automated integration process to finish. During this time, they take a coffee and discuss of what they have just done.

Adaptations

In this section, we describe how we have adapted the theoretical practices that have been presented in the previous sections. Indeed, when we do Extreme Programming, it is not possible (and not desirable) to stick to the letter of all the concepts.

During the realization of our thesis, we did not do pair programming all the time. In fact, we realized that it is not always effective to do pair programming. When meeting a bottleneck such as the discovery of a new language or the realization of a complicate algorithm, pair programming helps. But we noticed that when the task to be performed is simple, pair programming can be a waste of time. For this kind of task we prefer to perform code review: when a team member adds new code or changes the current code, another member of the team must validate his changes.

As discussed in the next section, we did not do weekly cycle but sprints of two weeks. This choice is motivated by the fact that we could not have a meeting with our supervisor every week and were more comfortable with the fact of having a weekend in the middle of our sprint either to rest or to work hard, depending on the amount of work left.

We tried to stick to the *Ten-Minute Build* practice and succeeded quite well. It is important for us to have a quick feedback from the automated integration process.

We have made Continuous Integration on our four main projects (the application for the patients, for the doctors, the backend and Hubot). We used a very popular CI tool: *TravisCI* (see section 5.7.1). It was responsible for building and testing a project every time a new commit was published. Continuous Integration is really interesting because it provides quick feedback. Also we are sure that our application works fine because after each commits all the tests of the project are ran.

2.1.2 Scrum

Scrum is an Agile project management methodology. The main aspect of this approach is to make a team responsive to changes. In this methodology, a sprint is the basic unit of development. A sprint has a duration of one month or less during which a "terminated", usable and potentially deliverable version of the software is created (in our case, sprint duration is two weeks).

Poker planning

Poker planning is a method to estimate the complexity of features to be developed. The team estimates user stories thanks to cards representing the fibonacci sequence. Here is the procedure of a planning poker party:

1. A member of the team reads the user story
2. Each member of the team estimates the complexity of the story by giving it some points
3. If everybody has assigned the same points, the team estimates another story; else each member explains his choice and the team votes again

The advantage of poker planning is that each participant can express himself. Also by estimating the stories, we can estimate how many points a team can handle during a sprint. This is called the *velocity* of a team. This value is useful to plan future sprints.

Review and retrospective

A review is done at the end of a sprint to look at the work that has been done and the work that was planned but could not be finished. At the end of this review, a team can compare its expected velocity to the real one and adapt the next sprint accordingly.

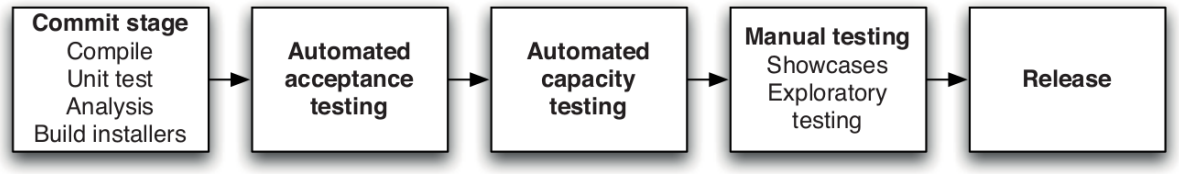


Figure 2.1: Continuous Delivery - Deployment Pipeline

At the end of a sprint, the team members also do a retrospective, asking themselves what went right and what went wrong during the sprint. They try to identify improvements for the next sprint.

Adaptation

This master thesis focuses on Extreme Programming methodology but we also used some principles of Scrum. This is not incompatible because Extreme Programming is more focused on code quality and Scrum is more focused on project management. From Scrum, we used the concept of "sprints". Our sprints had a duration of two weeks beginning by a session of poker planning. At the end of each sprint, we performed a sprint review and a sprint retrospective.

Sprints

The detail of all the sprints we have made is visible in this section 7

2.2 Continuous Delivery

The goal of Continuous Delivery (CD) is to release and deploy software in short cycles. It is the natural extension of Continuous Integration and push it further by deploying automatically the product to production (or other environment). Because ultimately the product must be published to users, Continuous Delivery makes this "release step" a trivial one so that it can be performed often and quickly.

To bring CD into a project, we implement incrementally a *Deployment Pipeline*. "Deployment Pipeline is an automated manifestation of your process for getting software from version control into the hands of your users" [8]. The deployment pipeline is integrated into the Continuous Integration tools and divides the whole process in stages represented on figure 2.1 and explained below:

- **Commit stage**: Compile the sources and store the binaries in an **Artifacts store**[1]. Run the unit test suites and code analysis.
- **Automated acceptance test**: Assert that the system works at functional and non-functional level. Assert that it meets the needs of the customers.
- **Manual test stage**: Assert that the system is usable and fulfills its requirements. Detect any defects not caught by automated tests. Verify that it provides value to users.
- **Release stage**: Deliver the system to the users (either by deploying the product or shipping new binaries).

[1]The artifacts store is independent from the version control since the binaries are derivated from the sources. It does not require a backup because the binaries can easily be re-build from sources. It is used to store and serve a specific version of the binaries to the different stages of the Deployment Pipeline.

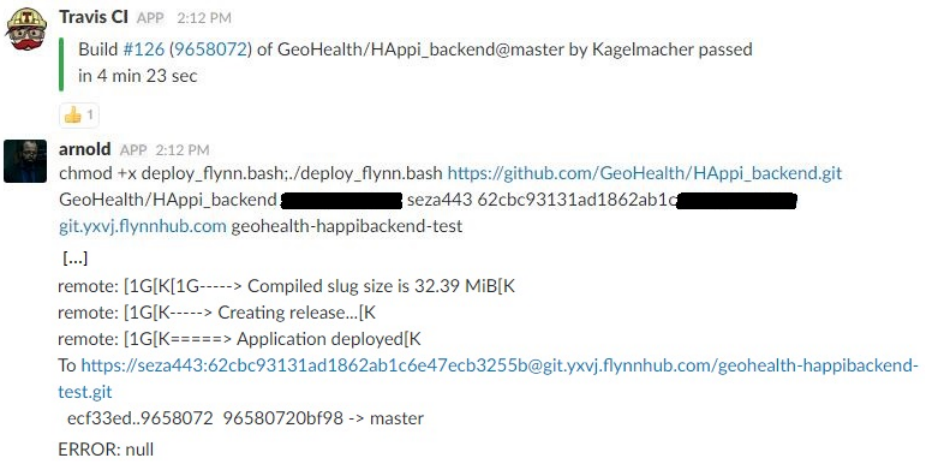


Figure 2.2: Continuous Delivery - Automatic deployment of the backend when tests succeed

2.2.1 Adaptation

During the development of our solution, we have successfully implemented a deployment pipeline for our backend. *TravisCI* was in charge of the commit stage: it fetched and built the dependencies; it ran the unit tests and integration tests; it posted a message to a dedicated channel in Slack. From there, *Hubot* (see section 5.7.4 for a presentation of *Hubot*) took over and pushed the new version of the backend to the "*dev*" environment of *Flynn* (see section 5.7.3 for a description of *Flynn*) as illustrated on figure 2.2. Once the backend is deployed in this "*dev*" environment, we are in the "manual test stage" of the deployment pipeline. To move to the release stage, we simply have to press a button on *Flynn*.

For our other projects (mobile application, frontend for doctors and *Hubot* itself), we did not implemented such a pipeline or at least not all the steps. The reason was that the schedule of the installation in production was different for each project: the mobile application was not published on a store yet so we had nowhere to push it; the frontend for doctors was developed later and was not published a lot of times so it would have been a waste of time to implement such a pipeline; *Hubot* itself, again was not published often enough to build this pipeline.

2.3 Test-driven development

Test-driven Development (TDD) is a software development process composed of cycles during which the writing of tests and code alternate. There are different practices to achieve TDD.

2.3.1 Benefits

TDD has an impact on the development of a project. On the long term, TDD improves developer productivity. R. Jeffries and G. Melnik prove, in *TDD - The Art of Fearless Programming* [11], that writing tests first is more effective than adding tests later and reduces production bug density by 40–80%.

2.3.2 Process

In this project, here is the process we followed: first we write a single test that fails (test is "red"), then we write the minimum amount of code to make the test pass (test is "green"), finally we refactor the code if needed, knowing that our tests will tell us if we broke something ("Red-Green-Refactor"). This is an iterative process, writing a "red" test, making it "green", refactoring code, starting over. The benefit is the fact that we write the least code possible and

we write code that works (almost) directly. This ensures a better code because we are sure there is no unnecessary code. In addition, a code with fewer lines includes fewer errors.

Chapter 3

Solutions

This chapter explains the different projects on which we worked to build a symptoms-tracker solution that can be used by patients to record symptoms and by doctors to aid diagnosis. It starts with an overview of the different projects, showing how they work together. Then we detail each project individually.

We called our solution *HAppi*, short form for Health Application. The goal of the mobile application is to give an easy way for patients (especially with chronic diseases) to track their symptoms and share them with their doctors. Specifically for this kind of patients, it is very common for the doctors to ask them to write down when some symptoms appear. It is usually done using a simple notebook. But keeping a notebook up to date can be difficult for some people: you forget your notebook, you are too lazy to take a note. The notebook also can quickly become messy and when patients are in an appointment with their doctors, their feelings and memories are distorted because they are tired of their disease: they cannot live like this anymore. With an application like *HAppi*, doctors have concrete data and an impartial point of view.

3.1 Overview

The figure 3.1 shows an overview of the solution we have built during this master thesis. We describe it from the upper left to the bottom right.

On the left, we see the frontends. The first one is a mobile application, used by patients to keep track of their symptoms as they occur. This application is synchronized with the *Ruby on Rails* backend. The second frontend is a web application designed for doctors. It allows them to receive and analyze reports, sent by their patients. They access it via a URL received by email.

In the middle part of the figure, we first see that there is a bot, built using the *Hubot* technology. This bot is used for many things but its main tasks are to warn us, the developers, when our applications are broken and to simulate patients. This simulation feature is quite important to make sure our data analyzing tools work fine.

The central element of the figure is the *Ruby on Rails* REST API. This application is the backend of our solution, exposing a REST API to the world. It is responsible for collecting and storing the occurrences of the users, creating reports for doctors, pushing data to *Elasticsearch* and asking external services for information about the context in which occurrences appear.

The backend uses a *PostgreSQL* database as main storage.

The last column of the figure is dedicated to Data Analysis and visualization. The first element is *Kibana*. It is a tool dedicated to data visualization of *Elasticsearch* storage. We use it to have global view of where and which symptoms appear the most in the world. *Kibana* is very good at aggregating data and produces nice charts. It reads data from *Elasticsearch*, the next element on the figure.

Elasticsearch is both a storage and a powerful search engine. It exposes a REST API to write and query data. Our *Rails* backend sends to it an anonymous version of each occurrence

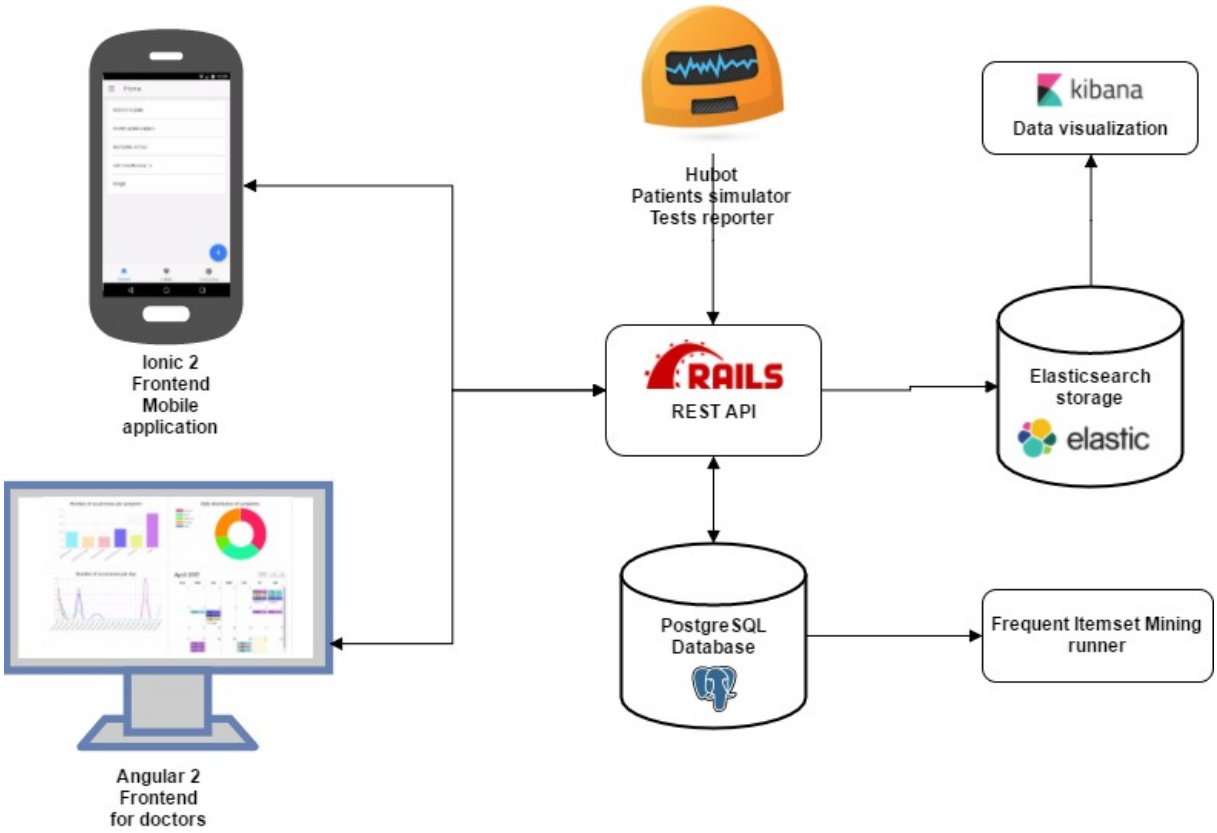


Figure 3.1: Solution overview

received.

Finally, we have developed a web interface for a data mining tool called *LCM*, addressing the problem of *Frequent Itemset Mining*. Thanks to this tool, we are able to see how many users had the same symptoms during a given time range.

All those components are described in details in the following sections.

3.2 Frontend for patients

Thanks to this mobile application, patients can keep track of their symptoms as they appear. They can also access the history of their symptoms and see some statistics. Finally, they can share them with their doctors to ease his diagnosis. This application mainly targets patients with chronic diseases to help them when they have to write down each symptom. Indeed, in this kind of disease, it is very common that doctors ask their patients to keep an history of all symptoms.

In this section, we start by presenting the technologies used for the mobile application. Then we explain in details the features.

3.2.1 Technologies

To develop this mobile application, we explored different possibilities. We wanted a technology that allows us to build a multi-platform mobile application easily.

At first, we thought about *Xamarin*[1] or *React Native*[2] because of their popularity. But finally we chose *Ionic 2*[3]. This choice is mainly motivated by the learning curve which is inherent to

[1] `https://www.xamarin.com/`
[2] `http://www.reactnative.com/`
[3] `http://ionicframework.com/docs/v2`

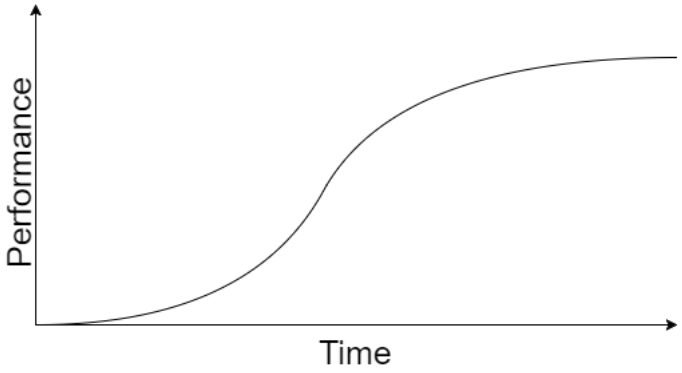


Figure 3.2: A typical S-curve representing the evolution of performance over time

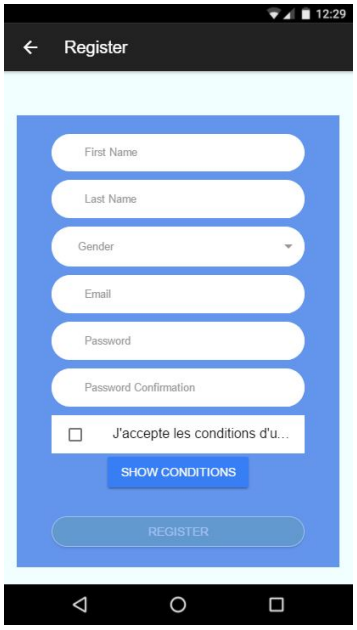


Figure 3.3: Registration page

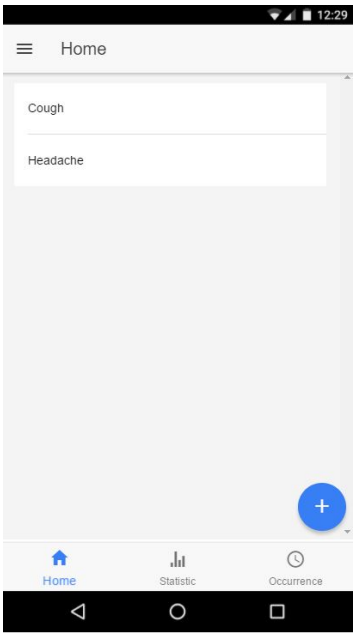


Figure 3.4: Home page

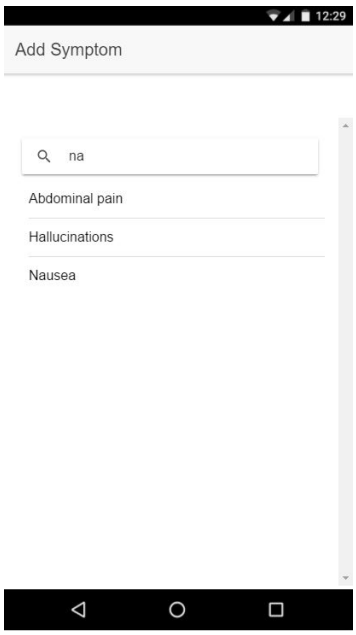


Figure 3.5: Adding a new symptom - Search results for "na"

every new programming language or framework. Investopedia gives the following definition: "A learning curve is a concept that graphically depicts the relationship between cost and output over a defined period of time"[10]. This concept is illustrated on figure 3.2. Most programming languages have a learning curve with this S-shape. At first, developers are not efficient because they have many things to learn. Over time, they become more and more efficient. Finally they reach a ceiling where they cannot be more efficient.

Because we were already familiar with *AngularJS* (see section 5.1.1), *Ionic 2* was a good choice since it is based on this framework. On the other side, to use *Xamarin* we should have learned C# and for *React Native*, we should have learned the concepts of *React*[4].

3.2.2 Mobile app features

The mobile application is dedicated to registering the symptoms of patients and sharing them with doctors. It also provides an history and a statistic page. In this section we go through the features proposed by the application by following the typical usage scenario of a new patient.

[4] `https://facebook.github.io/react`

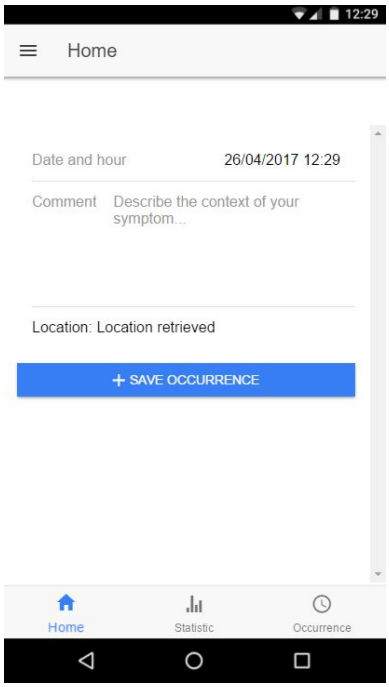


Figure 3.6: Adding a detailed occurrence of a symptom

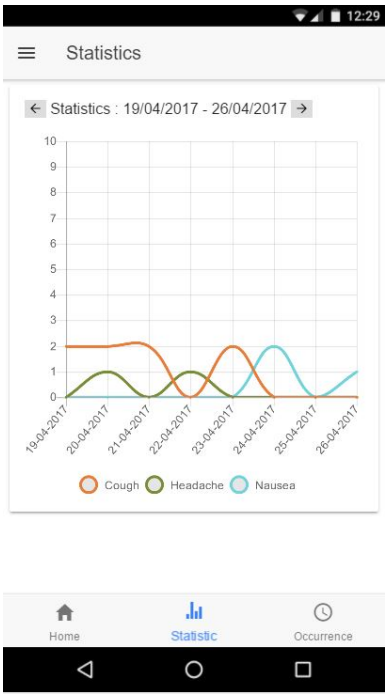


Figure 3.7: Statistics page showing the number of occurrences per day for each selected symptom

Figure 3.8: History of occurrences

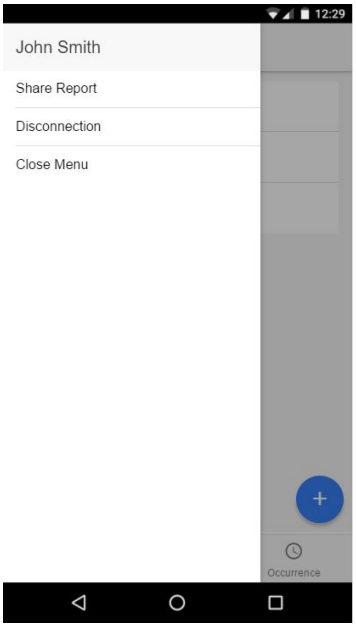


Figure 3.9: Sliding menu revealed

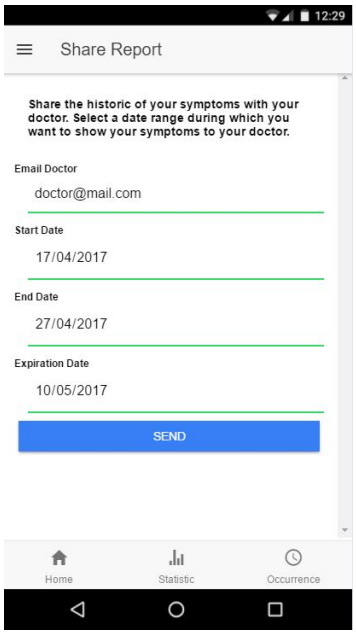


Figure 3.10: Share report with a doctor

Registration

Because our user does not have an account yet, he cannot simply log in[5].

The figure 3.3 shows the registration page. The form is quite simple, with classical inputs: first name, last name, gender, email, password. The final checkbox is actually very important because our application is still in beta so it is important that users are warned.

Once all the inputs are filled and the *"Register"* button is pressed, the user accesses the home page if the creation of the user account succeeded.

Home

On the home page, shown on figure 3.4, the patient has already added a few symptoms: *Cough* and *Headache*. He can do many things on this page:

- Register a new occurrence of a symptom by simply taping the corresponding symptom's name
- Register a detailed occurrence of a symptom by doing a long press on its name
- Add a new symptom with the blue "+" button on the bottom right
- Delete a symptom from the list by swiping the symptom's name to the left

When a new occurrence of a symptom is created, the GPS location is retrieved, if possible, as well as the date and time. The position is anonymized (see section 3.2.3 to have more details about anonymization) and the occurrence is sent to the server. The patient cannot click on the same symptom as long as the previous request is not finished.

Add a new symptom

Our patient now feels nausea coming. To add this new symptom, he simply clicks the blue "+" button on the bottom right and search for *nausea*. Then he taps the corresponding and the symptom is added to the list of symptoms of the home page. The figure 3.5 illustrates the search of a symptom. The search is done while the user is typing.

Add a detailed occurrence

Our patient wants to register more information about a symptom. He does a long press on a symptom name that brings the *detailed-occurrence* page, shown on figure 3.6. As soon as he enters this page, the location is being retrieved in the background. The patient can use this screen to change the date and time. Useful if he forgot to register a symptom earlier for example. He can type a free comment about this particular occurrence. A great improvement would be to allow him to give an estimation of the intensity of the symptom from 1 to 10.

Statistics

Our patient uses the app for a few days now. When he feels nauseous again, he can access the statistics page to have an overview of how many times this symptom appeared per day during the last week. In figure 3.7, the patient sees that he was nauseous two times two days ago and one time today. He can filter out the other symptoms that are also shown in the chart. Finally, he can navigate per week to see further statistics.

[5]The login screen is not shown in this document since it is a classical login form with username and password inputs.

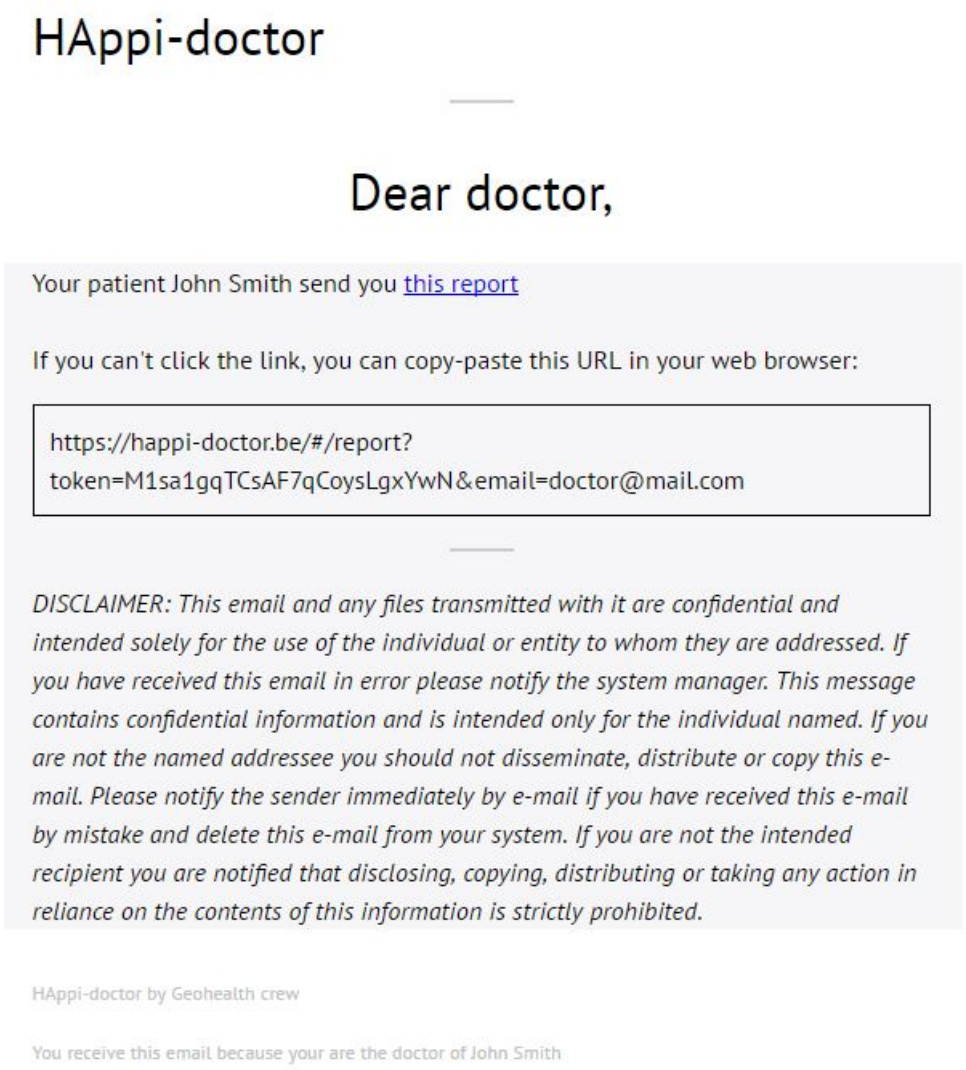


Figure 3.11: Email received by a doctor from a patient. It contains a unique link to consult the report.

Occurrences history

If the patient wants more details about a specific occurrence, the third tab shows a list of past occurrences. The patient can delete an occurrence from here by swiping it to the left, if he made a mistake for example.

Figure 3.8 shows the history of our patient.

Share report

A very important feature of the mobile app is to share a list of symptoms with a doctor. For example before or during an appointment, to ease the diagnosis of the doctor.

Our patient accesses the *Share report* page via the left sliding menu (shown in figure 3.9).

The figure 3.10 shows the screen from where the user gives the mail address of his doctor, select a date range and an expiration date. The report is generated and an email is sent to the doctor, containing a unique link with a secret token (example of such email is shown on figure 3.11). The report can only be consulted with the correct token associated to the email address of the doctor to which it was sent. For privacy reason, it is no longer available after the expiration date.

3.2.3 Security

When designing this application, we tried to keep security in mind since we are touching very sensitive data here.

GPS anonymizer

An important thing that needs to be anonymized is the position of the user. Since we record his location for every occurrence, if an attacker has access to this data, he could deduce private information about the user, like where does he live and work, what are his habits, ...

To reduce this risk, the location of the user is anonymized **before** it is sent on the network. This operation is done locally, on the smartphone of the user. To do so, we have taken a formula from Stackoverflow[6]. It takes a random position within a radius of 100 meters starting from the real location. And because we do this operation before sending the occurrence to the backend, even if an attacker was looking at the traffic, he would not retrieve real positions.

Report

The reports might lead to security issues because they give access to a partial history of the symptoms of a user without any authentication mechanism. However, we think that there are not a lot of possible attacks.

An attacker might try to guess what is the next generated token. To generate unique random tokens, we use *HasSecureToken* gem[7] which is based on *ActiveSupport::SecureRandom* class. This class is said to be "suitable for generating session key in HTTP cookies, etc." according to the official documentation[5]. So we can fairly suppose that it is extremely hard for an attacker to guess the next generated token.

An attacker could try to find a valid token using a *bruteforce attack*[8]. To do this, he would call directly our API by doing an HTTP GET request on */reports*. If the response has the status code 200 then the attacker found a valid token. But to prevent this kind of attack, we have linked each report to the email address of the doctor to which it was sent. So the attacker would have to guess both the token and the linked email address, which is much more difficult. Because even if he finds a valid token, he would not notice it since the response from the server is no different if the token is invalid or if it is valid but the email is incorrect.

Finally, even if the attacker successfully retrieve a valid token and its associated email, he must do it within the validity period of the report, i.e. before the expiration date is reached.

One last attack, that is harder to prevent, is the one where the attacker takes possession of the mailbox of a doctor. He could have access to all the emails containing links to reports. There is not much we can do against this type of attack. However, thanks to the expiration date of reports, the attacker would not be able to consult all of them. This is not a perfect solution but it is a mitigation of this attack.

3.2.4 Translations

Making a multi-language application can sometime be hard unless you think about it in the first place. That is what we have done, quickly adding a way to translate the application. Note that we did not translate our application entirely

Because we were developing an *Ionic* application, we could have used *angular-translate*[9]: a module for *AngularJS* that aims to make pluralization easier. But we preferred to use a tool that is widely used for pluralization: *gettext*. This tool is developed by the GNU project, an operating

[6] `http://stackoverflow.com/q/31192451/2179668`
[7] `https://github.com/robertomiranda/has_secure_token`
[8] `https://en.wikipedia.org/wiki/Brute-force_attack`
[9] `https://angular-translate.github.io/`

```
1 {{translation.gettext("Share Report")}}
```

Listing 3.1: Example of usage of TranslationProvider

system that uses only free software[6]. It has many implementations in different programming languages. We used the Javascript implementation of *gettext*[10].

To integrate this tool in our application, we created an angular service called *Translation-Provider* that can be injected in other components of the app. This service follows the original API of *gettext*: in order to retrieve translations, it exposes a method *gettext* receiving the original text in the default language (English in our case). Internally, the library has loaded the translations from a JSON file for the preferred language of the user. It performs a look up among the loaded translations for the given text. If a translation exists, it is returned, else the parameter is returned as is (i.e. the default text in English is returned). This mechanism allows developers to write inside the code the original text in the default language and translators to extract those texts and translate them in other languages. This provides a better readability of the code than having just a unique key identifying the text to retrieve. Here the key **is** the text to retrieve.

The snippet 3.1 illustrates how the translation provider is used inside an HTML page by passing the default English text to the *gettext* method.

Adding translations

Because all texts that have to be translated use the exact same method called *gettext*, it is possible for tools like *Poedit*[11] to extract directly from the source code all string to translate. This kind of tool is used by translators because they highlight the texts that are not translated yet. To ease the collaboration of translators, *gettext* (and *Poedit*) uses *PO* files[12]. This is a special format of file dedicated to translations. There is one *PO* file per language, containing the translation for this language.

Adding a new language is very easy. A translator only need to start its favorite *PO* file editor (like *Poedit*), select the project and add a language. This creates a new *PO* file corresponding to the language (example: *fr.po* for French, *en-US.po* for American English).

Compiling PO file

Finally, because the implementation of *gettext* we use receives JSON as input file, we have to convert the *PO* files containing the translations to JSON files. We use a tool called *po2json*[13] in order to do so. We simply need to run the command `npm run build-translation` which is an npm shortcut for a longer command invoking *po2json* on each *PO* file we have.

3.2.5 Google Store

The final step for the mobile application was to deploy it on the Google Store. It was not actually as easy as just building an APK because we needed to obtain a valid SSL certificate for our backend to make real smartphones able to access it via HTTPS. The section 5.12 explain how we obtained a valid SSL certificate and why it was mandatory.

With the certificate added to the backend, we were able to build and run our application for production using the command `ionic build android --prod --release`. We are not showing

[10] urlhttps://sourceforge.net/projects/jsgettext.berlios/

[11] `https://poedit.net/`

[12] `https://www.gnu.org/software/gettext/manual/html_node/PO-Files.html`

[13] `https://www.npmjs.com/package/po2json`

here the arguments that are needed to sign the application but of course we also had to generate a developer certificate and sign the application with it.

Deploying the application to the store was just a formality after we paid the 25$ one-time fee asked by Google to have a developer account. The application is available for every smartphone running Android 4.4 and more at the address: `https://play.google.com/store/apps/details?id=com.geohealth.happi`.

It is however important to note that our application will stop working on the 6 of August 2017 because we rented the *VPS* at OVH until this date and at the time we are writing this thesis, we do not plan to renew it.

We found it very interesting to deploy our app to a real store and were excited about getting feedback from real users.

3.3 Frontend for doctors

The frontend for doctors is an *AngularJS 2* web application available at `https://happi-doctor.be`. It is dedicated to providing reports to doctors containing the symptoms of their patients. Just accessing the URL of the website is not useful since the home page does not give any relevant information. It becomes interesting when accessing the website from a URL received in an email, containing a token and the associated email address. This URL is of form: `https://happi-doctor.be/#/report?token=TOKEN&email=EMAIL`.

We started the development of this web application after the development of the basics features of the mobile application and the backend (login, saving occurrences, showing statistics). During the development of this frontend for doctors, we were also developing the mobile application and the backend to make them able to share reports with doctors. So we naturally oriented our technology choice to *AngularJS 2* because of the experience we had with this framework.

In this section, we go through the features proposed by this frontend and some interesting aspects of the application.

3.3.1 Features

The home page has no features. It is a simple placeholder showing a demo dashboard of a fake report.

The report page is much more elaborated. An example can be consulted at `https://happi-doctor.be/#/report?token=3QknCrY7QxGLPwCyUuBizZtd&email=doctor%40mail.com`

First, it provides a list of symptoms (figure 3.12) along with the number of occurrences that the patient had during the period of the report. Each symptom in this list can be (de)selected to filter the rest of the report.

Then, there are several charts showing information, for example during which period of the day the occurrences appeared the most, the number of occurrences per day per symptom, a calendar showing the exact hour of each occurrence, a map and more. All those charts are updated every time the doctor (de)select a symptom from the main list.

In this section, we go through the different charts to explain their pertinence and how they can help doctors in their diagnosis.

Also, it is nice to note that we had a discussion with a doctor (more detail about this collaboration in section 3.7) to evaluate the pertinence of each chart.

Number of occurrences per symptom

This chart shows the number of occurrences that appears for each selected symptom. The figure 3.13 is an example of this chart. It was implemented using *ChartJS* that is described in section 5.5.1.

This chart is useful to quickly see what symptoms the patient had most frequently.

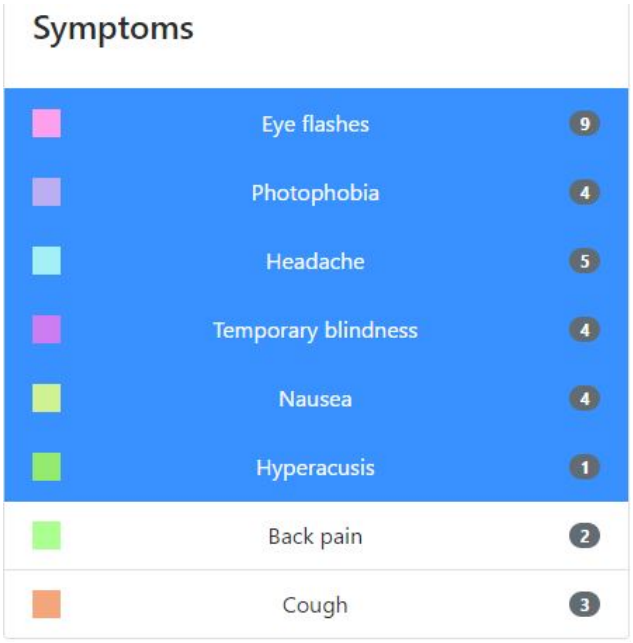


Figure 3.12: List of symptoms with number of occurrences. Blues are selected. Whites are deselected.

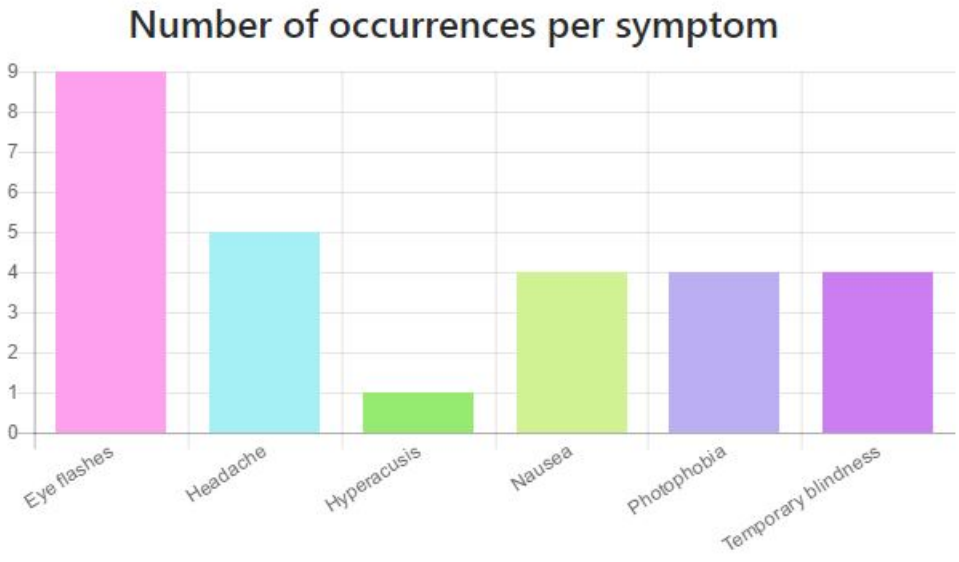


Figure 3.13: Chart showing the number of occurrences per symptom

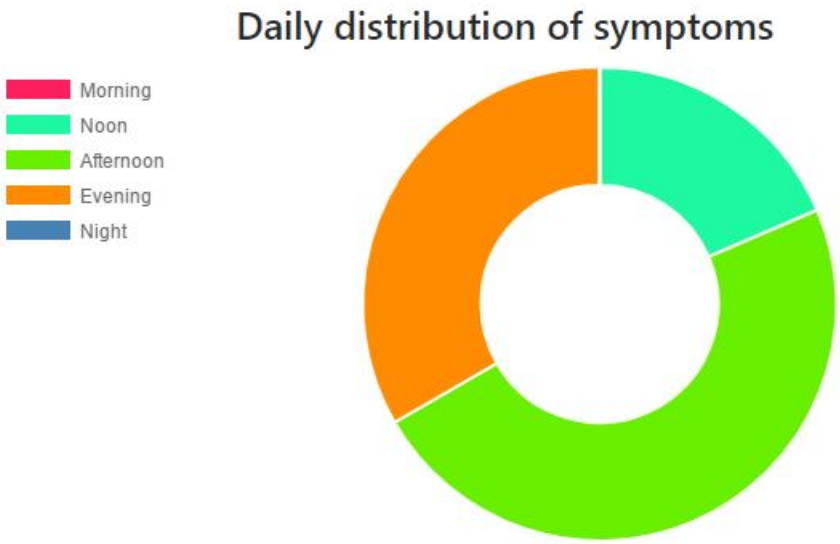


Figure 3.14: Chart showing the daily distribution of symptom

	Start hour	End hour
Morning	8	12
Noon	12	14
Afternoon	14	19
Evening	19	2
Night	2	8

Table 3.1: Periods of the day

Daily distribution of symptoms

The chart shows the proportion of symptoms that appeared during the different periods of the day. We have defined those periods as indicated in table 3.1.

This chart is useful to identify at a glance if there is a strong correlation between a period of the day and the appearance of some symptoms. For example, the figure 3.14 shows that half of the symptoms appeared during the afternoon. It was also implemented using *ChartJS*.

Number of occurrences per day

On this chart, illustrated on figure 3.15, the doctor can see the same kind of chart that the user has using the mobile application. This is the number of occurrences that appeared per day for each symptom. This chart is some kind of a visual history and can be used to visually see if some symptoms are related because they appear a lot together.

This chart was also implemented with *ChartJS*.

Calendar

Like shown on figure 3.16, a calendar is available to give details about the date and time of appearance of occurrences.

To construct this calendar, we used *FullCalendar* this is described in section 5.5.2.

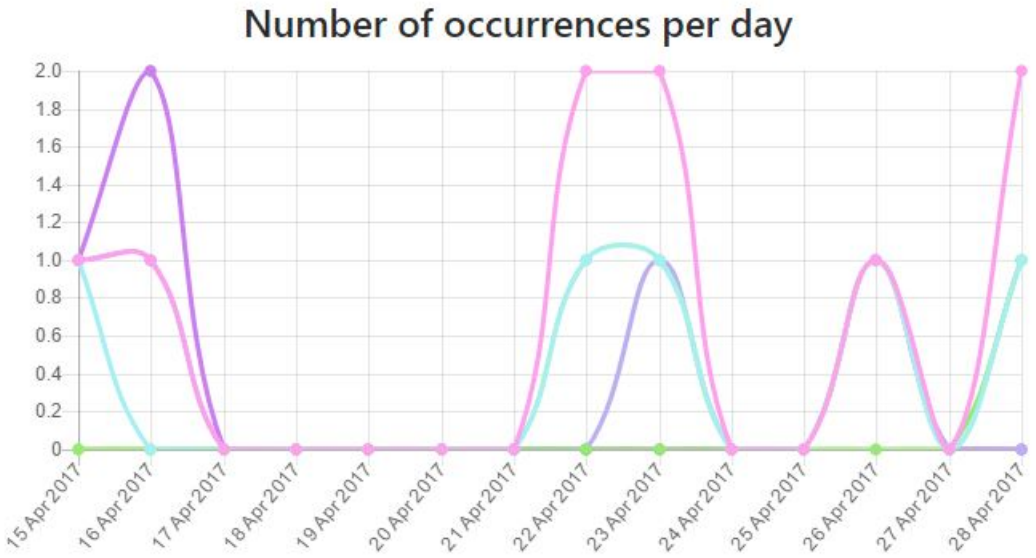


Figure 3.15: Chart showing the number of occurrences per day for each symptom

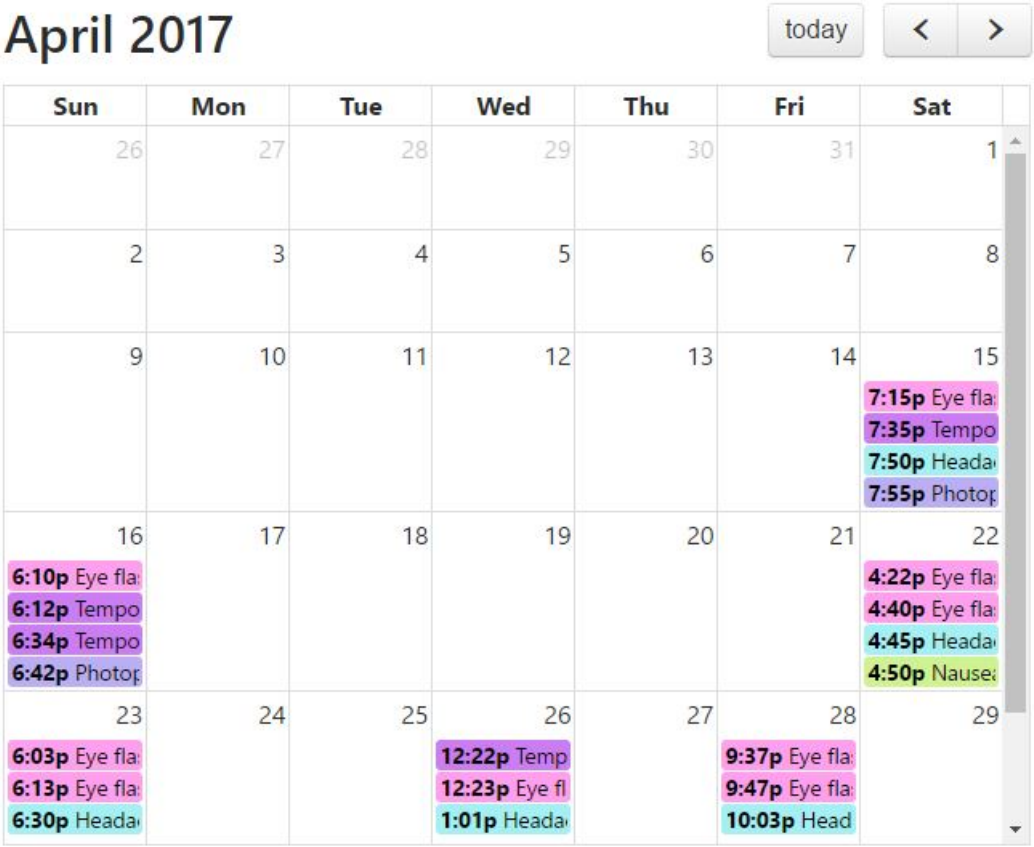


Figure 3.16: Calendar showing the date and time of appearance of occurrences

Figure 3.17: Map showing the gps locations where occurrences appeared

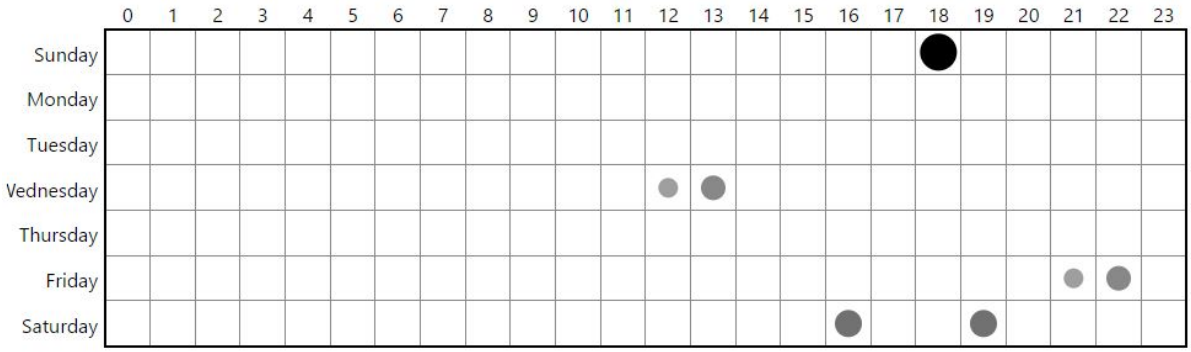


Figure 3.18: Punchcard (Github like) counting the number of occurrences grouped per hour and per day of the week

Map

The map (illustrated on figure 3.17) shows where the occurrences appeared. Depending on the zoom level, occurrences are (un)grouped. Clicking on a group of occurrences increases the zoom level and unfolds the group to see more precisely the location of occurrences.

The map is useful because symptoms can be caused by environmental factors.

To build this dynamic map, we used *MarkerClusterer* that is described in section 5.5.3.

Punchcard

A punchcard is a type of chart showing the number of occurrences that appeared per hour and per day of the week. Each day of the week is represented by a row and each hour of a day by a column. In each square, the bigger the black circle is, the more occurrences appeared at this hour and day. Hovering a circle with the mouse reveals the number of occurrences. The figure 3.18 illustrates this concept in our application. We were inspired by the punchard from Github[14]

A punchard allow doctors to reveal if occurrences occur especially on some days of the week, at some particular hours.

Full history and detailed factors

The final section of the report is a complete history of occurrences with their detailed factors. For now the detailed factors are only about the weather: description of weather, temperature and humidity percentage. But we designed the backend so that it is easy to add more factors.

The occurrences are grouped by symptom and ordered by date and hour.

[14]See an example of a punchard on Github at `https://github.com/GeoHealth/HAppi_mobile/graphs/punch-card`

Symptom	Day	Hour	Weather	Temperature (C°)	Humidity (%)
Eye flashes	15-04-2017	19:15:05	Overcast	10.0	66
Eye flashes	16-04-2017	18:10:05	Mostly Cloudy	10	39
Eye flashes	22-04-2017	16:22:05	Mostly Cloudy	10.0	66
Eye flashes	22-04-2017	16:40:05	Mostly Cloudy	10.0	66
Eye flashes	23-04-2017	18:03:50	Mostly Cloudy	10	33
Eye flashes	23-04-2017	18:13:50	Overcast	10.0	43
Eye flashes	26-04-2017	12:23:50	Scattered Clouds	5.0	70
Eye flashes	28-04-2017	21:37:50	Mostly Cloudy	11.0	50
Eye flashes	28-04-2017	21:47:50	Mostly Cloudy	10.0	54
Headache	15-04-2017	19:50:05	Mostly Cloudy	10.0	62

« 1 2 3 »

Figure 3.19: History of occurrences with the associated detailed factors

3.3.2 Security

Like we said in section 3.2.3, it is very difficult to guess a valid token and its associated email address. We consider that this protection mechanism is secure enough regarding the information that is available.

Also, accessing the report does not give any personal information about the user like the name or the email. The map shows the locations of the patient but each gps position is anonymized.

3.4 Hubot

Hubot (described in section 5.7.4) is a chatbot that can be used to automate almost anything. It is integrated into our Slack team.

We called our bot *Arnold* in reference to the TV show *Westworld*[15]. It lives in its own Github repository available at: `https://github.com/GeoHealth/arnold-slack-hubot`

3.4.1 Deployment

The deployment of the bot was very easy thanks to *Flynn* because it can be deployed as a classic *Node.js* project. We simply had to add a *Procfile*[16] to specify to *Flynn* how to start the bot. Also, we had to configure a custom integration in our Slack team.

3.4.2 Build and coverage warnings

Thanks to *Hubot*, we are notified in our Slack as soon as a build fails on *TravisCI* or when a commit decreases the code coverage. To achieve this, we wrote a simple custom script for *Hubot*. *Hubot* listens to all messages posted to the channel where the bot was added. Our custom script

[15] `http://www.imdb.com/title/tt0475784/`

[16] A *Procfile* is a special file used in *Heroku* (and *Flynn*) applications to describe how to start processes

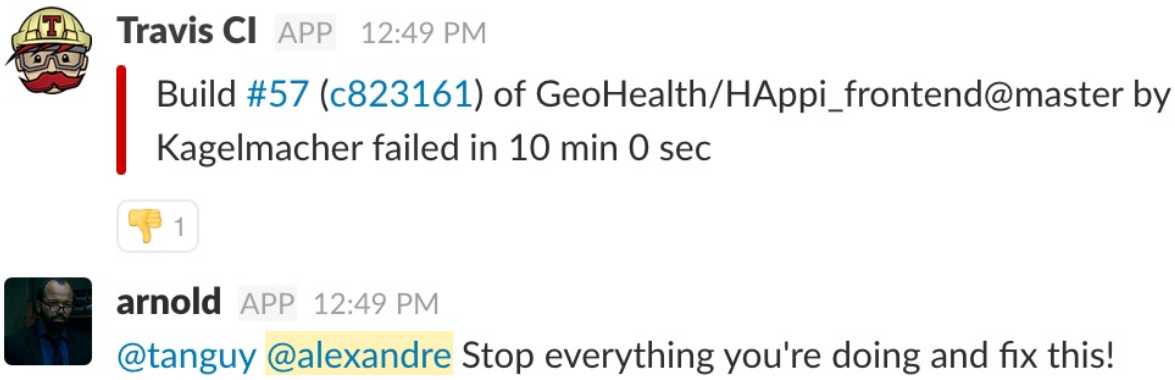


Figure 3.20: Hubot notifies the team when a build fail occurs

reacts to all messages matching the regular expressions `Build .* failed`, `Build .* errored` and `Test coverage .* has declined`. Messages of those forms are posted to Slack via the *TravisCI*'s slack integration[17] and *Codeclimate*'s slack integration[18]. In reaction, *Hubot* posts a message containing our names to force slack to notify us and encourage us to either fix the build or increase the code coverage. An example of such behavior is shown on figure 3.20.

3.4.3 Automatic deployment

When a build of the backend passes on *TravisCI*, *Hubot* automatically deploys the code to *Flynn* in the "*dev*" environment. We wanted this behavior because of the *Deployment Pipeline* described in section 2.2. The goal is to reduce human interaction during deployment to avoid human errors as much as possible [8]. Having *Hubot* automatically deploy every new version of the backend to the "*dev*" environment brings us to the *Manual testing* stage of the *Deployment Pipeline*. After having tested the backend manually, we can trigger the deployment in "*prod*" environment directly from *Flynn*. This last step corresponds to the *Release* stage.

3.4.4 Simulation

For certain types of applications simulation is needed. We need to simulate a lot of users to make some stress tests. To see how the application reacts under heavy usage. The simulation is also interesting to see if the features that we developed for data analysis are relevant. For example, to see if an influenza epidemic is detected by the application.

For ease-of-use, we decided to integrate the simulation feature into *Slack* thanks to *Hubot*. To implement the simulation, we wrote a custom script in *CoffeeScript*. As we had no experience in this language, we developed this feature by doing *Test-Driven Development*.

To not influence the different environments (*dev* and *prod*), we decided to create a specific environment for simulations called *simul*.

Hubot simulates occurrences of symptoms with the command shown in snippet 3.2. It creates the given number of users and for each of them, it posts a new occurrence to the backend using its REST API. Based on the parameters, it picks a random GPS location and a date from the given ranges. To simulate occurrences, the following fields must be given:

- **users**: the number of users.
- **symptoms**: a list of symptom names and the numbers of occurrence to create for each symptom.

[17] `https://slack.com/apps/A0F81FP4N-travis-ci`
[18] `https://slack.com/apps/A0F81R5AB-code-climate`

```
@arnold simulate users=4 symptoms=Abdominal Pain;10; latitude
   =50.673856 longitude=4.23655 radius=100 start_date=10-04-2017
   end_date=20-04-2017
```

Listing 3.2: Example of request for simulation: creation of 4 users with 10 Abdominal Pain as symptom in a circle area with a radius 100 meters between 10-04-2017 and 20-04-2017

- **geolocalization**: the latitude and the longitude with a radius in which the symptoms occur.
- **dates**: a start date and an end date during which the symptoms occur.

3.5 Backend

The backend is the cornerstone of our thesis. It is the connection between all the applications. It collects users information, it provides reports to the doctors and finally it supplies data to our data mining applications.

3.5.1 Technologies

We have decided to use *Ruby on Rails* to implement the backend because this language combines well with Agile methodology: it provides the flexibility level needed in an Agile project. Indeed, with *Ruby on Rails*, changing an application's feature is fast and easy. For example, database evolution is made easy thanks to a *migration* mechanism: the database architecture is defined by a list of migrations that must be ran in a particular order. When a developer needs to change the database structure, he can do so by adding a new migration that his coworkers can apply later on to upgrade their own local database. These migrations must be checked out in the version control tool just like the rest of the *Rails* application.

Rails can be used to build complete websites but we only used its REST API feature since we wanted independent applications (This is not entirely true because we made a web interface for a data-mining tool. See section 3.6.2 for more details).

3.5.2 Database

A detailed diagram of the database is available in the appendix C . In this subsection, we present the models used by the backend.

User

The model **User** represents a user of the mobile application (a patient). It inherits of the attributes from the model user defined inside the *Devise* gem except the attributes *omniauthable* and *confirmable* that are removed.

Symptom

This model represents a **Symptom** with its name, a short and a long description. It has also a gender filter to indicate if the symptom appears to female, male or both.

Occurrence

An **occurrence** represents an instance of a symptom at a specific *gps_coordinate*, for a *user*, with a list of *factors*.

GpsCoordinate

A **GpsCoordinate** represents the latitude and longitude of an occurrence.

Factor

A **Factor** represents a thing that can be measured when an occurrence occurs. For example, it could be the temperature of the patient, the intensity of the pain, the heart beats of the patient, ...

FactorInstance

A **FactorInstance** links a factor to a occurrence.

Report

A **report** is sent by the patient to his doctor. It contains a list of **SharedOccurrence** and a period during which the doctor may consult this report.

SharedOccurrence

This model allows the user (the patient) to indicate which occurrences he wants to share with his doctor.

SymptomsUser

The model **SymptomsUser** allows us to remember the favorite symptoms of the user (i.e. the ones he pinned to his home screen).

SymptomsCounts

The model **SymptomsCounts** is used to show statistics on the patient's mobile application. It represents the number of symptoms appearances for a given period and a given unit. The period is defined by a start date and an end date. The unit can be "hours", "days", "months" or "years". An instance of *SymptomsCounts* contains a list of *SymptomCount*.

SymptomCount

The model **SymptomCount** is the number of occurrences for a symptom that occurred during a period. It contains a list of *CountPerDate*.

CountPerDate

An instance of **CountPerDate** associates a date to a number of occurrences that occurred at that date.

3.5.3 Development

The development of the backend was the most organized part of our master thesis. When we developed a new feature, we proceed as follows:

1. Define the request and response specifications in *Swagger Editor* (see section 5.7.5)
2. Develop the feature using *Test-Driven Development* methodology

3.5.4 Versioning

We used namespaces to be able to manage different versions of the API. When someone uses our API, he can specify explicitly a version. This allow us, the developers, to update the API (for example modify the response format of a request) and still be compatible with other applications that use an older version of the API. It is all about having a stable contract. Specifying the version is optional since the latest version is implicitly used if none is provided. For example, one can access the list of all symptoms by doing an HTTP GET request to `https://api.happi-doctor.be/symptoms` (without explicit version) or `https://api.happi-doctor.be/v1/symptoms` (with the *v1* version specified explicitly) [19].

We have chosen to put the version of the API in the URL but other solutions exist: the version can be specified in a custom header or inside the content type of the request [9]. Each method has some pros and cons. In the end, we have chosen the URL method because in our opinion, this is the most obvious solution and it was very easy to implement: we have simply added the *Versionist*[20] gem. Also it organizes our routing policies in a nice way by using namespaces, one per version. Namespaces are reflected in the *Rails* application by a module of the same name and a module is under a folder of its own name. So the file structure follows the routing organization. We find it clear and well organized.

Currently, we have only developed the version *V1* of the backend so we did not use much this versionning mechanism but it was important in term of maintainability to implement it as soon as possible.

3.5.5 Authentication

When a user logs in or creates an account, the mobile application sends requests to the backend. To manage authentication, we used a gem called *devise_token_auth*[21]. This gem creates the new model *User* (and the associated migration to create the corresponding table in database) and adds several routes to our API. These routes are described in our *Swagger* file (see section 5.7.5) with the *Authentication* tag.

Based on the famous authentication gem *Devise*[22], we consider that this gem provides a satisfying level of security. The authentication is based on a token, changing after each request or not. We have chosen not to make this token changing after each request to make the client's implementation simpler. However, we have found that there is a library called *angular2-token*[23] that integrates very well with the *devise_token_auth* gem. So a nice security improvement would be to use this library instead of managing the token ourselves. Indeed, using this library would allow us to easily take advantage of the changing-token security policy. Thus, a hacker intercepting a single HTTP request and stealing the token would not be able to impersonate a user's identity.

Another security aspect is the token expiration. The default value is two weeks, meaning that any generated token is no longer usable after this delay. We have expended this value to one year for development purpose because we did not want our token to expire while we were developing. Again, for security reason, this value should be reduced back to two weeks if the application begin to be widely used.

[19] Note that if you actually try these queries, they will fail with a status *401 Unauthorized* because you must also send proper headers to be authenticated.

[20] `https://github.com/bploetz/versionist`

[21] `https://github.com/lynndylanhurley/devise_token_auth`

[22] `https://github.com/plataformatec/devise`

[23] `https://github.com/neroniaky/angular2-token`

3.5.6 Asynchronous jobs

Some tasks performed on the backend can take time (for example: call the api to get weather information or send an email). We do not want the backend to be blocked by this kind of process and prevent it from answering other requests. *Sidekiq*, described in section 5.3.1, is a tool that allows to perform asynchronous operations is *Ruby* by defining *Workers*.

Thanks to this tool, the backend can handle many concurrent jobs or delay some of them.

We have defined two workers related to the recording of a new occurrence. The goal of these workers is that when an application adds a new occurrence to the backend, the response is as fast as possible and every operation that can be asynchronous is actually asynchronous to avoid slowing down this response.

In the two next sections, we give a description of the workers we have built.

Wunderground

When a user creates a new occurrence, thanks to its position and the date of the occurrence, we can retrieve information about the weather. To get this information, we need to make a call to *Wunderground* (see section 5.10.2 for a description of this tool). Since making an API call is something that can take some time and is not mandatory in order to provide a response to the application which has added the occurrence, we have moved this task into a *Sidekiq* worker.

The job is queued in the default queue of *Sidekiq* and processed as soon as a thread is available in the thread pool. So each occurrence that has got gps coordinates will eventually be decorated with some weather information. Those additions take place as *FactorInstances*.

The weather information is something really useful to help the doctor to make his diagnosis. The appearances of symptoms may depend on meteorological factors.

Elasticsearch

To be able to use the data analysis tool *Kibana* (see section 5.11.2), we have to push each occurrence to an *Elasticsearch* instance (*Elasticsearch* is presented in section 5.11.1). This operation is not mandatory to add an occurrence correctly and can take some time (because it is made through an HTTP POST request). So, like the *Wunderground* API call, we have moved this operation to a *Sidekiq* worker.

Thanks to this data, *Elasticsearch* supplies information to *Kibana*.

3.5.7 Testing

To be able to certify the reliability of our backend is really important because other applications rely on it. Tests, when performed correctly, can prove this reliability with a certain a level of confidence depending on how well tests cover the different aspects of the system.

To develop the backend, we used the *Test-Driven Development* methodology. It is an iterative process starting by writing a test that fails (red test). Then, writing the minimum needed code to make the test pass (green test). Finally, we iterate over this red-green loop.

Rspec

There are multiple testing frameworks in *Ruby*. We have chosen to use *Rspec* (see section 5.8.3) because it has a good documentation and a big community. It also has lots of helpers to do assertion on different things we could need.

With *Rspec*, we wrote both unit and integration tests. Unit tests make assertions on models, some helpers classes and the *Sidekiq* workers. Integration tests make assertions on multiple modules working together. Typically to test our controllers, we make high level tests on the final result of the controller's actions.

Emails testing Because our application is able to send emails to doctors, we must also test that the emails are sent to the correct recipients with the correct content. *RSpec* allows us to write unit tests for *Mailers*[24]. No emails are actually sent during these tests but we can be sure that the emails are built and processed the way we want.

Guard

When doing *Test-Driven Development*, it is important that tests run very fast because they are run many times: once when writing the initial red test, potentially multiple times when writing code to make the test green. We used *Guard::RSpec* (see section 5.8.2) to automatically and intelligently run our tests when a file is modified. *Guard* is able to detect that a file has changed and run only the tests for this file, providing quick feedback to the developer.

When running tests twenty times per minute (or more), this kind of tool is really helpful.

Spring

A task that slows down the testing process of a *Rails* application is the initialization of the *Rails* environment. This task loads all the requirements for *Rails* and it can take a few seconds to perform. So we have added *Spring* (explained in section 5.8.4), a *Rails* preloader that speeds up our tests by loading only once the *Rails* application and keeping it running in the background. So the very first time we run our tests, the entire environment is loaded and it can take more than ten seconds. Subsequent tests run much faster because the environment is already loaded. Typically, a test suite for a model runs in more or less 300 milliseconds.

Mutation testing

As developed further in section 4.4, mutation testing allows us to be sure that the tests are really useful, that they cover all possible cases. In short, it gives us confidence about our tests.

When developing the backend, we really wrote a lot of tests (more than actual code actually). So we wanted to have this confidence that our tests were "right", that they could handle all possible cases. We have added the gem *mutant-rspec*[25] that integrates with *Rspec* and creates mutations for any file it receives.

We run the mutation tests less frequently than the unit tests. Typically we let *TravisCI* run them for us because it can take a few minutes to run.

However, not every mutation is interesting. Sometimes we end up with some mutation tests failing because we are using the method *Time.zone.parse* to parse a date but it also works with the mutation *Time.parse*[26]. This is the kind of mutations that would make us loose time if we were trying to write tests to kill this mutant.

Factory Girl

When writing tests for our controllers, we quickly realized that we needed to create instances of our models very often. In *Rails*, there is a solution to ease the creation of placeholder models used in tests: *fixture*. A *fixture* is a sample data, specified in a YAML file that is automatically used to populate the database before a test run. However, we decided to use a gem called *Factory Girl* (described in section 5.8.5) as replacement of *fixtures* because it provides nice additions like multiple build strategies and factory inheritance.

[24] A *Mailer* is a special class in *Rails* to is able to send an email based on some associated views

[25] `https://github.com/mbj/mutant`

[26] It is highly recommended to use *Time.zone* to manipulate dates in *Rails* [12]

3.5.8 Hosting

Students at UCL doing their master thesis can ask for a dedicated machine and use it as they want. We requested one to host our backend. The default proposed Operating System was CentOS and we were fine with it. We did not care about having a specific linux distribution. However, we should have looked better at the requirements of the hosting tool we wanted to use, namely *Flynn*. The installation of *Flynn* is only possible on Ubuntu (for now). So we were stuck with our dedicated machine.

We really wanted to use a Platform as a Service (PaaS) tool to deploy the backend because this kind of tool is widely used in companies: they allow to deploy applications easily, with almost no configuration and to scale them dynamically. Other PaaS tools exist of course but only a few of them can be self-hosted. We wanted this self-hosting option to avoid being restricted by a free plan or being charged too much.

Finally, we decided to rent a *Virtual Private Server* (VPS) on OVH[27]. We first took the smallest instance called *VPS SSD 1* with 1 vCore, 2 GO of RAM and 10 GO of storage on an SSD disk but we ended up with the *VPS SSD 2* instance because we were running low on storage and RAM. On this *VPS*, we have installed *Ubuntu 14.04 Server 64 bits* and the installation of *Flynn* became easy. With *Flynn* in place, we deployed three versions of the backend: `simul`, `dev` and `prod`.

Another reason for using OVH is that with the dedicated machine provided by the UCL, we could not have a port accessible from the internet so our app would have worked only from the inside of the UCL network. This was a big problem if we wanted to test our application with real users.

3.5.9 Environments

We deployed initially three versions of the backend on *Flynn*.

- **prod**: production environment used by real users. This environment is never reset and its data must be treated with care.
- **dev**: development environment that we (the developers) can use as we want. The data of this environment can be deleted without any hurt.
- **simul**: simulation environment used by *Hubot* to simulate users and occurrences of symptoms. This environment exists to isolate the simulations from the other environments.

When we started doing end-to-end testing with the mobile application we deployed a fourth version of the backend. This environment is called **tests** and is used only for the end-to-end tests. It was created at the very end of this master thesis. That is why we often talk about "three environments" while there are now four of them.

3.5.10 SSL certificate

Flynn generates a self-signed SSL certificate when it is installed. In order to access the *Flynn* dashboard and connect to the applications with SSL, one needs to install locally the self-signed SSL certificate. This is not very convenient and they are working hard[28] to make the installation pick up a valid SSL certificate from *Let's Encrypt*[29]. In the meantime, we can generate manually an SSL certificate and configure *Flynn* to use it for some routes.

We realized we were going to need a valid certificate when we first tried to build our mobile application in production mode and tested it in an Android device: it was not working at all. The

[27] `https://www.ovh.com`

[28] `https://github.com/flynn/flynn/issues/1995`

[29] `https://letsencrypt.org/`

server could not be reached. We never had the problem before because we were deploying our application in debug mode and in this mode, no certificates are checked [30]. So before deploying the application to the Google Store, we used *Certbot*[31] to generate a valid certificate for the following domains:

- `happi-doctor.be` Our basic domain name routing to the doctors frontend
- `doctors.happi-doctor.be` A subdomain also routing to the doctors frontend
- `api.happi-doctor.be` A subdomain routing to the *prod* backend

We have multiple routes pointing to our applications. For example, the production backend can be accessed by `api.happi-doctor.be.` or `geohealth-happibackend-prod.happi-doctor.be` but we did not see the need to generate a certificate for all routes. Indeed, only the *prod* backend and the frontend for doctors need a valid SSL certificate. The other environments (*simul* and *dev*) are only used by developers having the self-signed SSL certificate of *Flynn* installed locally.

The certificates generated by *Certbot* are provided by *Let's Encrypt* and last for 90 days. *Certbot* is able to renew them automatically but we did not configure this because we do not plan to keep the server up after our master thesis.

3.6 Data Analysis

With the frontend for doctors, we can consult data for only one patient. But it could also be very interesting to have a global view of this data. We might be able to detect areas where an epidemic is taking place, make the correlation between symptoms, see the impact of the environment on the symptoms... This feature could be very interesting for governmental organizations for example.

To bring this analysis feature, we worked on two strategies. The first one is based on existing aggregation tools: *Elasticsearch* and *Kibana*. In this solution, we simply had to deploy an instance of each tool and configure them. The second approach is based on a data mining tool called *LCM*, addressing the problem of *Frequent Itemset Mining*, built by Takeaki Uno, Tatsuya Asai, Yuzo Uchida and Hiroki Arimura [14]. For this solution, we have built a web interface on top of the *LCM* executable and some business logic to generate the input and parse the output of *LCM*.

In the following sections, we present the two solutions in details.

3.6.1 Solution based on existing aggregation tools

Kibana and *Elasticsearch* are both developed by the Elastic company. *Kibana* reads data from an *Elasticsearch* instance and generates charts and other data visualizations diagrams.

Elasticsearch

Elasticsearch (more details about this tool in section 5.11.1) is a search and analytics engine. It can receive data from any sources, in any format and analyze it.

We host our *Elasticsearch* instance on *Bonsai*[32], using their free plan. We did not deploy it on our VPS because it can consume lots of resources and we did not want it to impact on our backend infrastructure.

Elasticsearch allows us to store occurrences in order to produce statistics. When the mobile application sends an occurrence of a symptom to the backend, the backend sends it to *Elasticsearch*. We created a mapping, shown in snippet 3.3, for the occurrences. Thanks to this mapping,

[30] `https://forum.ionicframework.com/t/from-http-to-https-preparing-the-app/26926/5`
[31] `https://certbot.eff.org/`
[32] `https://bonsai.io/`

```
{
   "mappings": {
       "occurrences": {
           "properties": {
               "occurrence_id": { "type" : "string", "index" : "
   not_analyzed" },
               "symptom_id": { "type" : "string", "index" : "
   not_analyzed" },
               "user_id": { "type" : "string", "index" : "
   not_analyzed" },
               "location": { "type": "geo_point" },
               "symptom_name": {"type": "string", "index" : "
   not_analyzed" }
               }
        }
   }
}
```

Listing 3.3: Mapping for the occurrences in Elasticsearch

Elasticsearch is able to index all the occurrences based on their location which is useful to show occurrences on a map in *Kibana*.

Kibana

We used *Kibana* (more details in section 5.11.2) to display information about the data stored in our *Elasticsearch* instance. We deployed it on an *Heroku* dyno, using the free plan. Again, we did not deploy it on our VPS because it can be quite heavy and resources consuming. Furthermore, we were not going to use it very often so a free instance from Heroku was perfect for our needs: it sleeps after 30 minutes of inactivity and wake up when it is solicited[33].

Thanks to *Kibana*, we created a dashboard in which we added the following elements:

- The total number of occurrences stored in *Elasticsearch*
- A pie chart to show the percentage of each symptom
- A bar chart to count the occurrences of each symptom
- A map to show the distribution of symptoms across the globe
- A bar chart to show the number of occurrences added per day

The aim of this dashboard, shown in figure 3.21, is to consult the medical data in its entirety. This can be useful for institutions such as governments to make decisions on the field of health. The dashboard can highlight an area where the number of symptoms is unusually high, show an epidemic such as an unusual number of influenza, ... Note that the pie chart and bar chart can be clicked to filter the rest of the dashboard.

Something that disappointed us, is that with the areas on the map, we cannot make the distinction between the symptoms: they are all grouped together.

You can consult the *Kibana* dashboard with this url: `http://happi-static-kibana.getforge.io`

[33] `https://www.heroku.com/free`

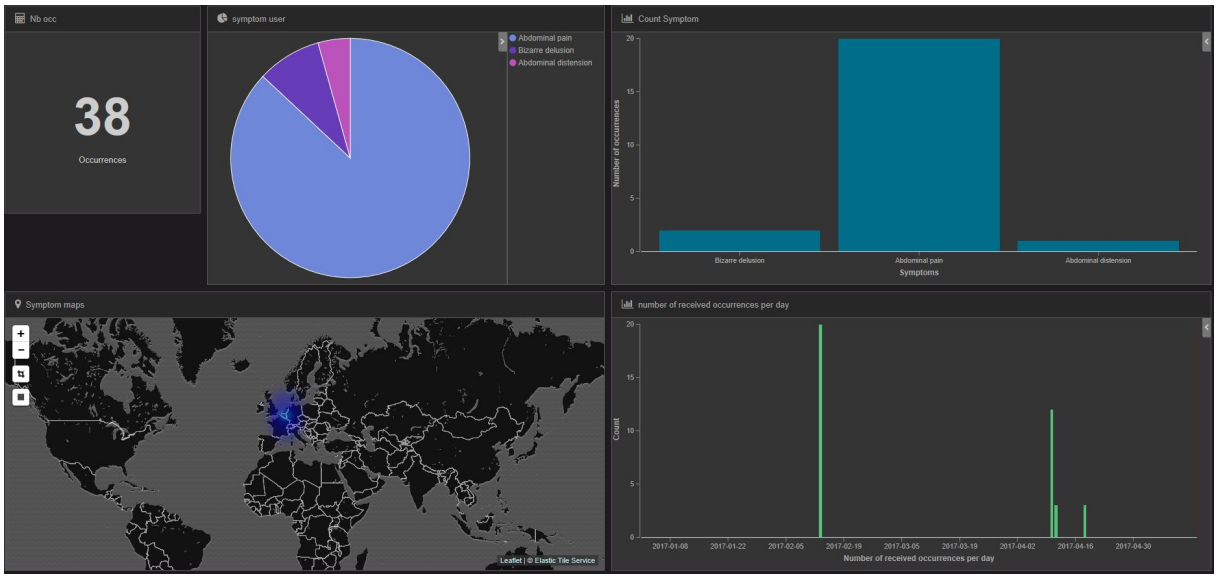


Figure 3.21: Dashboard inside Kibana

Privacy

We do not think that this dashboard presents any privacy issue. It could only be a problem if there were data from a single user since it would become possible to track his position and symptoms appearances. But people having access to this dashboard are not able to know the number of users nor their personal data so it is not an issue.

However, a potential risk that exists is that insurance companies use this data to identify dangerous areas and that they refuse to cover particular diseases in those areas.

3.6.2 Solution based on LCM

LCM (presented in section 5.11.3) is a data-mining tool available for academic use only.

It is in the form of a binary executable, which is not very convenient to use. Furthermore, we cannot "simply" run the program, we must generate the correct input for it. To be able to use *LCM* easily and in a comfortable way, we built a web interface on top of it. The goal is to be able to produce a valid input file, call the executable, parse and store the output file and display the result as easily as possible.

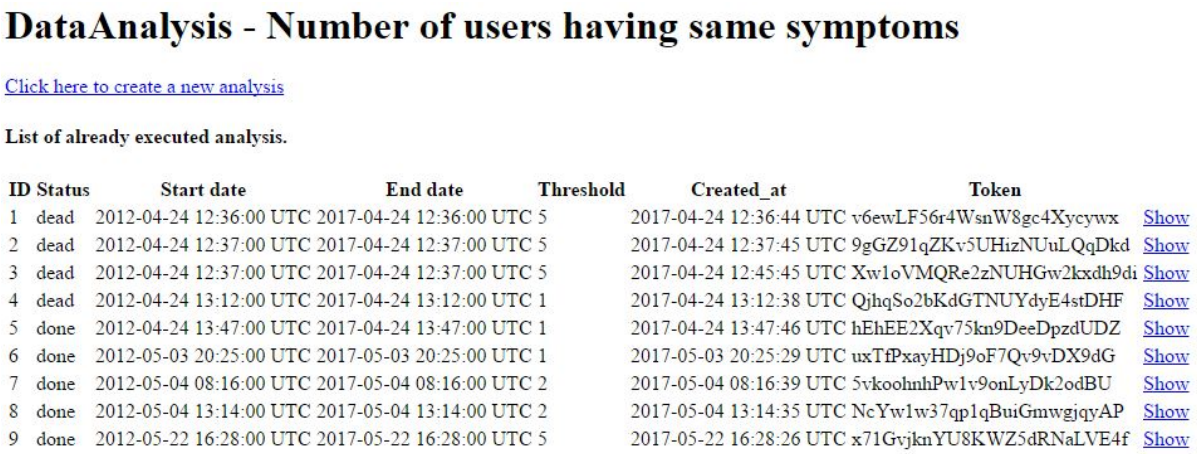
We had many ideas of applications of *LCM* in our context: output the number of users having the same symptoms, output the number of symptoms that occurred per region, output the number of symptoms having similar factors. We have only implemented the first one but we have done it in such a way that it is easy to extends and add new capabilities.

Implementation of a web interface

We implemented the web interface that is available at `https://api.happi-doctor.be/data_analysis/users_having_same_symptoms`. The figures 3.22, 3.23 and 3.24 illustrate this web interface. The first one shows the list of all *LCM* jobs and their status. The second one shows the details page of a job, with its start date, status, parameters and results. The last figure shows the page to create a new job.

The first choice we had to do in order to implement this interface was answer the question *Where do we create this web interface?* We had two solutions:

- Create a new, independent project
- Add the web interface into the existing backend



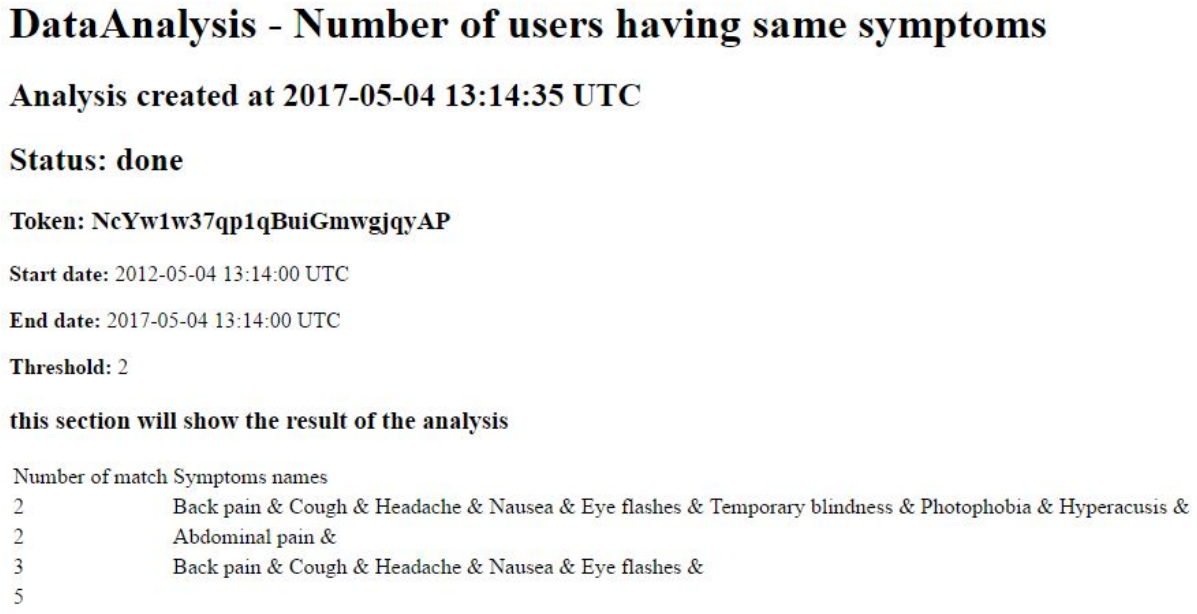
DataAnalysis - Number of users having same symptoms

Click here to create a new analysis

List of already executed analysis.

ID	Status	Start date	End date	Threshold	Created_at	Token	
1	dead	2012-04-24 12:36:00 UTC	2017-04-24 12:36:00 UTC	5	2017-04-24 12:36:44 UTC	v6ewLF56r4WsnW8gc4Xycywx	Show
2	dead	2012-04-24 12:37:00 UTC	2017-04-24 12:37:00 UTC	5	2017-04-24 12:37:45 UTC	9gGZ91qZKv5UHizNUuLQqDkd	Show
3	dead	2012-04-24 12:37:00 UTC	2017-04-24 12:37:00 UTC	5	2017-04-24 12:45:45 UTC	Xw1oVMQRe2zNUHGw2kxdh9di	Show
4	dead	2012-04-24 13:12:00 UTC	2017-04-24 13:12:00 UTC	1	2017-04-24 13:12:38 UTC	QjhqSo2bKdGTNUYdyE4stDHF	Show
5	done	2012-04-24 13:47:00 UTC	2017-04-24 13:47:00 UTC	1	2017-04-24 13:47:46 UTC	hEhEE2Xqv75kn9DeeDpzdUDZ	Show
6	done	2012-05-03 20:25:00 UTC	2017-05-03 20:25:00 UTC	1	2017-05-03 20:25:29 UTC	uxTfPxayHDj9oF7Qv9vDX9dG	Show
7	done	2012-05-04 08:16:00 UTC	2017-05-04 08:16:00 UTC	2	2017-05-04 08:16:39 UTC	5vkoohnhPw1v9onLyDk2odBU	Show
8	done	2012-05-04 13:14:00 UTC	2017-05-04 13:14:00 UTC	2	2017-05-04 13:14:35 UTC	NcYw1w37qp1qBuiGmwgjqyAP	Show
9	done	2012-05-22 16:28:00 UTC	2017-05-22 16:28:00 UTC	5	2017-05-22 16:28:26 UTC	x71GvjknYU8KWZ5dRNaLVE4f	Show

Figure 3.22: LCM web interface - List all the jobs



DataAnalysis - Number of users having same symptoms

Analysis created at 2017-05-04 13:14:35 UTC

Status: done

Token: NcYw1w37qp1qBuiGmwgjqyAP

Start date: 2012-05-04 13:14:00 UTC

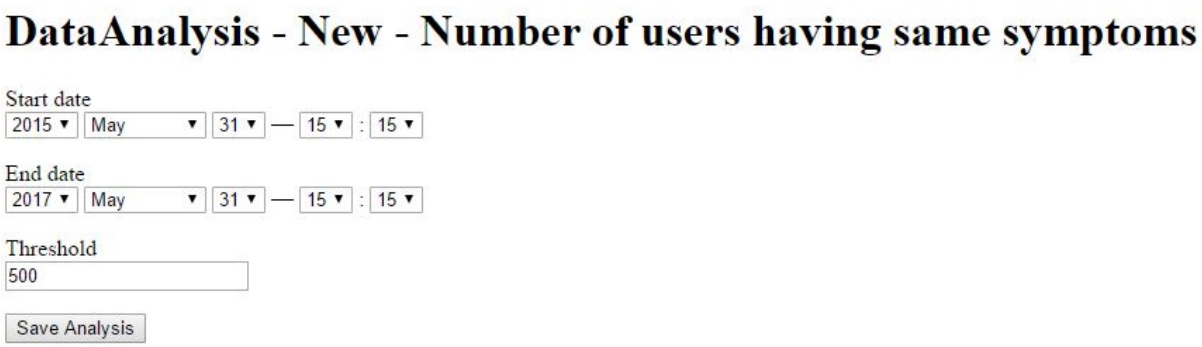
End date: 2017-05-04 13:14:00 UTC

Threshold: 2

this section will show the result of the analysis

Number of match	Symptoms names
2	Back pain & Cough & Headache & Nausea & Eye flashes & Temporary blindness & Photophobia & Hyperacusis &
2	Abdominal pain &
3	Back pain & Cough & Headache & Nausea & Eye flashes &
5	

Figure 3.23: LCM web interface - Details about a job



DataAnalysis - New - Number of users having same symptoms

Start date

2015 ▾ | May ▾ | 31 ▾ — 15 ▾ : 15 ▾

End date

2017 ▾ | May ▾ | 31 ▾ — 15 ▾ : 15 ▾

Threshold

500

Save Analysis

Figure 3.24: LCM web interface - Create a new job

This question was hard to solve because on one hand we wanted our backend to stay as simple as possible and not to add any web page to it. But on the other hand, if we created a new project, we would have lots of duplication in our models because the *LCM web interface* would have to be able to access the same database as the backend: read symptoms, users, occurrences, etc. So we would duplicate those models and maybe even the database migrations, which is even worst. If we go with this solution, and have a duplication of the migration, then which project is responsible for running the migration? Is it be the backend or the new project (or both)? How to maintain those duplicated models and migrations synchronized? A solution to the duplication problem is to isolated the models (and the related migrations) to a dedicated gem. Then by adding the gem to both projects we have the correct version of the models and the corresponding migrations. But the problem of who runs the migration persist. Actually we are reaching a severe limitation of *Rails*: a *Rails* application must have its own, dedicated database because *Rails* handle lots of stuff for the developers, in particular it generates the database structure. So we cannot have multiple *Rails* application connected to the same database.

Actually, we have a third solution: create a new application *that does not use Rails*. The problem of who is responsible for running the migration vanishes but we still have the problem of duplication: we are loosing the models we have already created in *Rails* and must recreate them in another technology. It becomes even harder to keep the two projects synchronized.

So we have chosen the second solution: integrate the web interface into the existing backend. But we have done it in a specific way to separate as much as possible the API itself from the *LCM* web interface. We started by creating a new namespace in our routing table. A namespace is a prefix common to a list of routes and it is reflected in the structure of the *Rails* application by a module of the same name (and a module is inside a folder of the same name). So we have a very nice separation between our backend, that is under a module (and so a folder) specifying the version of the API (see section 3.5.4 about the versionning of the API) and the *LCM web interface* that is under the module (and folder) *data_analysis*.

The next step was to create a controller, some web pages and a *Sidekiq* worker responsible for creating the input file, running *LCM* and parsing the result. We have done all this doing *Test-Driven Development* to be sure at every moment that we were on the good track.

In the end of the process, we had created two new models: *BasisAnalysis* and *AnalysisUsersHavingSameSymptom* extending the first one. We have the same inheritance hierarchy for the *Sidekiq* worker: *AnalysisUsersHavingSameSymptomWorker* inherits from *BasisLCMAnalysisWorker*. With those abstractions, we tried to make it easy to add a new kind of worker for another type of analysis, for example an analysis that outputs the number of symptoms per region.

The container problem

Our solution was working in our development machine but when the time came to push it to *Flynn*, we could not see the results of the *LCM* jobs. We realized that it was because on our local machine, we were running *Rails* and *Sidekiq* side by side but on *Flynn*, they run in two different containers. Because *LCM* outputs the results into a file and we were creating the output file from the *Sidekiq* process but parsing it from the *Rails* process to generate the web page, the *Rails* process could not access the file.

The solution was actually easy: we needed to parse the output file in the *Sidekiq* process, store the results in the database and simply read the results from the database in the *Rails* process. So we created two new models: *AnalysisResultSymptom* and *AnalysisResult*. We moved the parsing logic of the output file into the *Sidekiq* worker and the *Rails* controller generating the web page with the result only has to read the results from the database. It is actually much better to do it this way because with the first solution, we were parsing the output file every single time a user wanted to access the result page. Here the parsing is done only once and the file is deleted when it is done, saving disk space.

Job status

Since *LCM* is running in a *Sidekiq* worker, it is executed asynchronously. This is, of course, much better because the *LCM* executable can take lots of time to run. But how can the user know if the job is done, still running or dead?

Each *LCM* job is represented in the database with its threshold, a unique ID and a **status**. The status can be either *created*, *running*, *done*, *aborted* or *dead*. We set the status to the correct value when the job is created, running in *Sidekiq* or finished (successfully or not). The *aborted* status is the only one that is never used because we did not implemented a way to kill a *LCM* process from the web interface.

Security

As you might notice, the web interface is accessible to anybody for the moment. It is deployed in all our backend's environment: `simul`, `dev` and `prod`. The URL we gave in the introduction of this section is the one for the `prod` environment so you can see data about real users.

This is a security issue to make this kind of interface accessible to anyone. A mandatory improvement to make is to add an authentication mechanism to access this interface and be able to run *LCM* jobs. We think about adding an *admin* field to the *User* model and create a web login form granting access to administrators only.

3.7 Feedback

3.7.1 Doctors

During our thesis, we made a lot of hypotheses to develop our applications. For example, we assumed that the gps position and the weather are relevant information to collect. We wanted to validate these hypotheses so we met several doctors. We have especially worked with a doctor from UCL, Dr. Coralie Bonnier. We organized different meetings with her. At the beginning to present our project and know what she thought about it. And at the end to make a presentation of the applications. She was very interested by the project and told us she could use our solution. She also asked to add some features to the doctor frontend such as the creation of a PDF with the report of a patient to be able to download it and store it in other software.

After the deployment of the mobile application on Google Play, we received good feedback from health professionals through the Linkedin platform: we posted[34] a description of our solution and a link to our mobile application in a group dedicated to medical concerns and lots of people posted comments. Globally, they were very happy with our solution for several reasons:

- They saw that it could help for "distance patient care" (aka "tele-medecine") especially for people living in areas with limited medical care
- They hope that it could improve the relation between patients and doctors.
- They think that the application could help the interactions between doctors and patients. Because the patients often interpret their symptoms and describe them not clearly. This complicates the doctor's diagnosis.

They also encouraged us to continue developing the application and they suggest us some improvements for our application such as :

- Be able to locate symptoms on a human's body diagram.
- The patient could indicate his feelings during the apparition of the symptom. Add emoticons to indicate feelings.

[34] `https://www.linkedin.com/groups/120372/120372-6274680442075643906`

- Indicate the range of the pain.
- Indicate duration of symptom.

Chapter 4

Testing

We wrote lots of tests for our projects to avoid regression issues and reach a good level of confidence into our code. In fact, for some projects, we wrote more tests than lines of code (see appendix A about code statistics).

There are different types of tests. Most people know about unit tests and integration tests. A bit less know about end-to-end tests. And only a few know about mutation tests. We used all those types of tests during the realization of our projects. Each type of test is useful for something different and the tests as a whole give us a good level of confidence that our solution works as expected.

4.1 Unit testing

The aim of unit tests is to test that each individual public method works as expected: for a given input, it reacts as required. They are fast to run and cover most of the code. They should not use any database or dependency modules: if any dependencies exist, they should be stubbed. Stubbing is a technique used to simulate a component's behavior to avoid using this component during the tests.

When writing unit tests, we can observe multiple metrics like the *statement coverage*, *branch coverage*, *condition coverage*, etc. The higher a coverage is, the more confident we can be that there is no undetected bug in our code.

For each commit published to one of our public github repositories, unit tests are run by *TravisCI* and a *test coverage* report is published to *Codeclimate* (see section 5.7.2). The *test coverage* indicates the lines of code that are called by our tests. This metric is useful to determine which part of an application is not tested yet. You can consult the *Codeclimate*'s dashboard for our projects by looking at the table 4.1.

Actual test coverage

We tried to keep the *test coverage* metric very high for all of our projects. We only achieved a 100% test coverage for the backend. We can easily explain this result by the fact that we were doing *TDD* for this project so each line is tested at least once. It means that every single line of

Project's name	**URL**
arnold-slack-hubot	`https://codeclimate.com/github/GeoHealth/arnold-slack-hubot`
HAppi mobile	`https://codeclimate.com/github/GeoHealth/HAppi_mobile`
HAppi frontend (for doctors)	`https://codeclimate.com/github/GeoHealth/HAppi_frontend`
HAppi backend	`https://codeclimate.com/github/GeoHealth/happi_backend`

Table 4.1: Codeclimate URLs of repositories

code of the backend is executed at least once when the unit tests run. All those tests are very useful because the backend has evolved a lot and thanks to them, we were sure that we were not breaking anything when adding, removing, modifying or refactoring code.

The mobile application has a coverage close to 80% because we did not test the trivial parts of the application. Also we encountered errors while adding some of the tests like a memory leak or some random timeout. Because of those non-deterministic errors, we did not write as many tests as we wanted. Indeed, we were losing more time trying to make the tests work than if we were testing the app manually. So the test coverage is high enough to be confident in the fact that the mobile application could go to production. In this project, we mainly tested the Angular services and providers that we wrote and the main methods of the controllers.

The frontend for doctors does not have any unit tests because it mainly displays information and does not perform any complicated computation. Hence, it was faster for us to test manually the behavior of the application than to write unit tests for it.

The project for *Hubot* has a *test coverage* of 53%. This seems to be quite low but this can be explained easily: we have added multiple custom scripts to our bot (for example it reminds us about deadlines, post some comments on our Slack, etc.) but we did not test all those (simple) scripts because they are not essential. They just add some fun functionalities to the bot. The only script that has been intensively tested (by the way, it was written by doing *TDD*) is the one to simulate users. And we can see on *Codeclimate* that this script has a coverage of 99%[1]. So we can be quite confident that this script works as expected. Of course, the unit tests must stub all the server's calls so this level of confidence stays as long as the server behaves also as expected.

4.2 Integration testing

Integration tests are used to check that different components work together as expected. These components are dependent to each other and during unit testing, these dependencies are stubbed. Here the goal is to assert, at a higher level, that the dependencies actually work fine together.

We wrote integration tests in the backend only. The controllers tests are integration tests, making assertion on the result of an HTTP request made to a controller. We test the returned HTTP code and value, verify the state of the database after the test and make sure that dependencies are called with the expected parameters.

These integration tests are ran along with the unit tests by *Rspec*.

4.3 End-to-end testing

End-to-end testing (E2E) corresponds to testing the application like the user would use it. So a website is tested from a browser, an application is tested from a mobile phone (emulator),... The goal is to simulate the same behavior as a user to check if the entire flow of the solution works as expected. The advantage of E2E testing is that it can test the application as it appears to the user. Indeed we can reproduce stories asked by the customers. For example, with E2E testing, we can represent the story: "add a symptom to my list of symptoms' and validate it easily.

Concretely we implemented E2E testing only for the mobile application. We did not make any E2E testing on the doctor application because this application is really simple and does not need a lot of interactions by the user. We integrated the E2E testing to our continuous delivery pipeline. When writing the E2E test for the mobile application, we deployed a new environment of our backend to *Flynn* dedicated to the tests. This instance of the backend is never used unless for testing. *TravisCI* builds, executes the mobile application and run the end-to-end test suite.

[1] `https://codeclimate.com/github/GeoHealth/arnold-slack-hubot/coverage?sort=covered_percent&sort_direction=desc`

4.4 Mutation Testing

We write tests to give us confidence about our code. But what about testing our tests? How can we be sure that our tests are good? After all, tests are code just like any other code.

Mutation testing is another kind of testing that evaluates the quality of existing tests. It creates mutants by modifying a program in small ways. A mutant is a modification of the source code by, for example, switching operators, deleting an instruction, etc. Here are a few examples:

- $== \Rightarrow !=$
- $++ \Rightarrow --$
- $true \Rightarrow false$
- ...

To generate the mutants, the source code is scanned and on each code lines a test is performed to know which mutations are applicable. After that, a pool of mutations is created. For example, if we take this line of code.

```
if ((@report && (@report.expiration_date > Time.zone.now)) &&
(@report.email == params.fetch(:email)))
```

We get the following mutations:

1.
```
    if ((@report || (@report.expiration_date > Time.zone.now))
    && (@report.email == params.fetch(:email)))
```

2.
```
    if ((@report && (@report.expiration_date <
 Time.zone.now)) && (@report.email == params.fetch(:email)))
```

3.
```
    if ((@report && (@report.expiration_date > Time.now
)) && (@report.email == params.fetch(:email)))
```

4.
```
    if ((@report && (@report.expiration_date > Time.zone.now))
    && (@report.email !=
 params.fetch(:email)))
```

5. ...

We can observe (in red) several mutations. The test suite is initially run on the unmutated code to be sure it passes. Then, the test suite run on each mutant. At least one test must fail to "kill" the mutant because this indicates that the tests are appropriate and they are able to detect changes in the source code. Otherwise, the mutant stays "alive" and it indicates that we have to write a new test or to refactor the code to kill it.

Chapter 5

Technologies

During our master thesis, we used many different technologies. In this chapter, we present the tools we used the most.

5.1 Frameworks

A framework is a bunch of tools and software components that provides generic functionality. They help developing IT solutions faster by taking care of a lot of aspects that are common to a family of software. During this thesis, we discovered and used *AngularJS 2* to develop a website for doctors, *Ionic 2*, which is based on *Angular 2*, to build a multi-platform mobile application and *Ruby on Rails*, a *Ruby* framework to build websites, that we used to create our REST API.

5.1.1 AngularJS 2

AngularJS 2[1] is a TypeScript-based open-source front-end web application framework developed by Google. The goal of *AngularJS* is to ease the creation of a web application. The framework uses a MVC (Model-View-Controller) architecture. It adapts and extends traditional HTML to present dynamic content.

An *Angular 2* application is composed of *components* that are some kind of small self-contained applications. These components interact with each other to create the global application.

Typescript is a language developed by Microsoft. It is free and open-source. It is built on top of Javascript to which it is transpiled. It brings lots of features from Object-Oriented Programming that are not present in Javascript such as static typing, interfaces, etc.

Angular2 Webpack Starter

To start a new project in *AngularJS 2*, we can use the *angular-cli* tool[2] or we can clone an existing, pre-configured project. To build the frontend for doctors, we opted for the second approach and started from *Angular2 Webpack Starter*.

Angular2 Webpack Starter[3] is a pre-configured *AngularJS 2* project. It comes with several tools already installed and helps to kick-start a project. It includes (among other tools) *Karma*, *Protractor* and *Jasmine* to test the application, *Typescript* to write it and *Webpack* to build it.

It follows a file structure based on the component approach, a new standard for *Angular* applications that organizes the app in components. Each component is some kind of self-contained application living in its dedicated file or folder. All resources needed by the component (style sheets, HTML files, tests, etc.) are grouped in the same folder.

[1] `https://angular.io`
[2] `https://cli.angular.io/`
[3] `https://angularclass.github.io/angular-starter/`

Starting from this project instead of using the command line generator was a time saving solution because we did not have to install the different tools ourselves and make sure the version were not clashing with each other.

5.1.2 Ionic 2

Ionic[4] is a framework, based on *Cordova*[5] and *AngularJS*, to build multi-platform mobile applications. We use the second version of this framework, that is based on *AngularJS 2*.

With *Cordova*, developers can develop mobile applications using the web technologies: *HTML*, *JavaScript*, *CSS*. They can also access the native API of mobile devices like the GPS, fingerprint reader, camera, ... by adding a plugin into their applications. Developers write the code once and build it for many platforms like *Android*, *iOS*, *Firefox OS*, *Windows*, *Mac OS* and more. Once deployed, the application runs as a web page, inside a webview component.

Ionic adds a layer on top of *Cordova* by integrating *AngularJS* and bringing new visual components that look like natives ones. Like *Cordova*, it provides a Command-Line Interface (CLI) tool, extending *Cordova CLI*, to easily manage the application lifecycle: create an application, add a platform and some plugins, build and deploy the application.

5.1.3 Ruby on Rails

Ruby on Rails[6], also called *Rails*, is a framework written in *Ruby*. It uses *Model-View-Controller* (MVC) as architectural pattern and allows to quickly develop web applications. It can also be used to develop REST API. *Rails* helps developers a lot by handling lots of things automatically.

The migration mechanism is useful to manage a database architecture. Each migration is represented in a file marked with a timestamp. The migrations suite must be run in order to produce the latest database structure. A developer can add migrations easily thanks to the CLI assistant. Other developers of the team can then run the migrations to update their local database to the latest version. Thanks to this mechanism, it is easy to collaborate and manage the version of the database.

Ruby has a package manager (RubyGems). It allows to install libraries in the form of "gems". The gems make the power of *Rails* because there are many interesting libraries available.

5.2 Package manager

To manage the dependencies of our projects, we used packages managers.

5.2.1 NPM

NPM[7] is a package manager for Javascript. It allows developers to define the dependencies of a project in a file called *package.json*. It provides a CLI and has a huge database of public packages. Like every good package manager, it is able to manage different versions of a package and gives the ability to developers to specify exactly which one they need.

Working with a team becomes easier once the file *package.json* has been added to a version control system. Every developer can install the dependencies in one single command: *npm install*.

[4] `http://ionicframework.com`
[5] `http://cordova.apache.org`
[6] `http://rubyonrails.org`
[7] `https://www.npmjs.com`

5.2.2 Bundler

Bundler[8] is a package manager for *Ruby* projects. It manages the dependencies of a project for different environments (development, staging, production) and takes care of installing the right version of the gems.

Thanks to this tool, installing all the gems of a project is as simple as running *bundle install*.

5.3 Asynchronous tools

5.3.1 Sidekiq

Sidekiq[9] is an efficient background processing tool for *Ruby*. It can manage a pool of threads and multiple queues of jobs to execute.

When working with *Sidekiq*, one can define a *Worker* by creating a class extending the module *Sidekiq::Worker* and override the *perform* method. A worker defines a job that can be queued, delayed and processed by *Sidekiq*.

5.4 Translation tools

5.4.1 gettext

This tool is developed by the GNU project, an operating system that uses only free software[6]. It has many implementation in different programming languages. We used the Javascript implementation of *gettext*[10].

This implementation exposes a *gettext* method that receives a String. This String is the text to display in the default language (in our case, English). Internally, the library has loaded the translations from a JSON file for the preferred language of the user. It performs a search among the loaded translations for the given text. If a translation exist, it is returned, else the parameter is returned as is (i.e. the default text in English is returned). This mechanism allows developers to write inside the code the original text in the default language and translators to extract those texts and translate them in other languages. This provides a better readability of the code than having just a unique key identifying the text to retrieve. Here the key **is** the text to retrieve.

5.4.2 Poedit

Because all texts that have to be translated use the exact same method called *gettext*, it is possible for tools like *Poedit*[11] to extract directly from the source code all strings to translate. This kind of tool is used by translators because they highlight the texts that are not translated yet. To ease the collaboration of translators, *gettext* (and *Poedit*) uses *PO* files[12]. This is a special format of file dedicated to translations. There is one *PO* file per language, containing the translation for this language.

5.5 Data visualization libraries

In this section, we briefly explain the tools we used to implement the charts, map, calendar and punchard.

[8] `https://bundler.io`
[9] `https://github.com/mperham/sidekiq`
[10] `https://sourceforge.net/projects/jsgettext.berlios/`
[11] `https://poedit.net`
[12] `https://www.gnu.org/software/gettext/manual/html_node/PO-Files.html`

```
// HTML
<chart [type]="nbOfOccurrencesPerSymptomGraph.type"
       [data]="nbOfOccurrencesPerSymptomGraph.data"
       [options]="nbOfOccurrencesPerSymptomGraph.options"></chart
   >

// Typescript
public nbOfOccurrencesPerSymptomGraph = {
    type: 'bar',
    data: {
      labels: ['Eye flashes', 'Headache', 'Hyperacusis', 'Nausea'
   , 'Photophobia', 'Temporary blindness'],
      datasets: [{ data: [9, 5, 1, 4, 4, 4] }]
    },
    options: {
      responsive: true,
      maintainAspectRatio: true,
      scales: {
        yAxes: [{
          ticks: {
            beginAtZero: true
          }
        }]
      },
      legend: {
        display: false
      }
    }
  };
```

Listing 5.1: Example of ChartJS for Angular2

5.5.1 ChartJS

ChartJS[13] is a library available as an npm package. It is open source and easy to use but still provides an advanced configuration capability.

We used the npm package specially crafted for *AngularJS 2*[14], coming with a custom angular component. To design a chart like the one shown on figure 3.13, one can simply use the *HTML* and *Typescript* code shown on snippet 5.1.

Using this library, we made the charts shown on figures 3.13, 3.14 and 3.15.

5.5.2 FullCalendar

To display the calendar shown on figure 3.16, we used the *FullCalendar* library[15]. At the time we were developing this feature, no package for *Angular2* was available so we used the pure *Javascript* version of the library. If we had to redo this feature now, we would use the npm package for *Angular2*[16] that has been released on April 2017.

[13] http://www.chartjs.org
[14] https://www.npmjs.com/package/angular2-chartjs
[15] https://fullcalendar.io/
[16] https://www.npmjs.com/package/angular2-fullcalendar

We created our own angular component that can be used with the *full-calendar* HTML selector. This component receives a list of symptoms and a list of colors and adds the occurrences of each symptom to the calendar at the date it occurred, using the color associated to the symptom.

5.5.3 MarkerClusterer v3

To build a map like shown on figure 3.17, we needed a library able to group points in a map based on the zoom level. We found a library called *MarkerClusterer v3*[17] that does exactly what we were looking for. Again, this library has no special version for *Angular2* so we created our own component accessible with the *bubble-map* HTML selector.

Receiving a list of symptoms, this component adds to a Google Maps the location where occurrences occurred.

In order to use the Google Maps API, we had to create a Google Developer account on `https://console.developers.google.com` and activate the Google Maps Android API. We obtained a key that we restrained to be only usable on the domain `happi-doctor.be` for security reasons (typically avoid that someone steals our key and consumes all our Google API requests quota).

5.5.4 D3.js

To create a punchcard like the one shown on figure 3.18, we found a snippet of code on `http://bl.ocks.org/kaezarrex/10122633` that is based on *D3.js*. *D3.js*[18] is a famous library used to visualize data by binding them to *Document Object Model* (DOM)[4]. Based on this code, we built an angular component, accessible with the *punchcard* HTML selector, dedicated to showing a punchard by receiving a list of symptoms.

5.6 Reporting tools

When an application is used by a lot of people, it is necessary to receive fast feedback when something happens. Reporting tools make it easier. Thanks to these kind of tools, we do not have to wait a feedback from the user himself, the application sends it automatically.

We used reporting tools for the backend, to watch our different environments but especially the *prod* one and for the mobile application.

5.6.1 Sentry

Sentry[19] tracks errors and exceptions that happen during the execution of applications and provides instant notification with detailed information needed to prioritize, identify, reproduce and fix each issue.

When errors occur, *Sentry* can send notification via email, sms, or chat. We integrated *Sentry* into our Slack, so when something is broken, we receive a notification via Slack: the figure 5.1 shows an example of exception reported by *Sentry*.

5.6.2 Fabric

Fabric[20] is a reporting tool that provides crash reporting for Android and iOS applications through *Crashlytics*. This library is available as a plugin for *Cordova* named *cordova-fabric-plugin*[21] so it

[17] `https://github.com/googlemaps/js-marker-clusterer`
[18] `https://d3js.org`
[19] `https://sentry.io`
[20] `https://fabric.io`
[21] `https://github.com/sarriaroman/FabricPlugin`

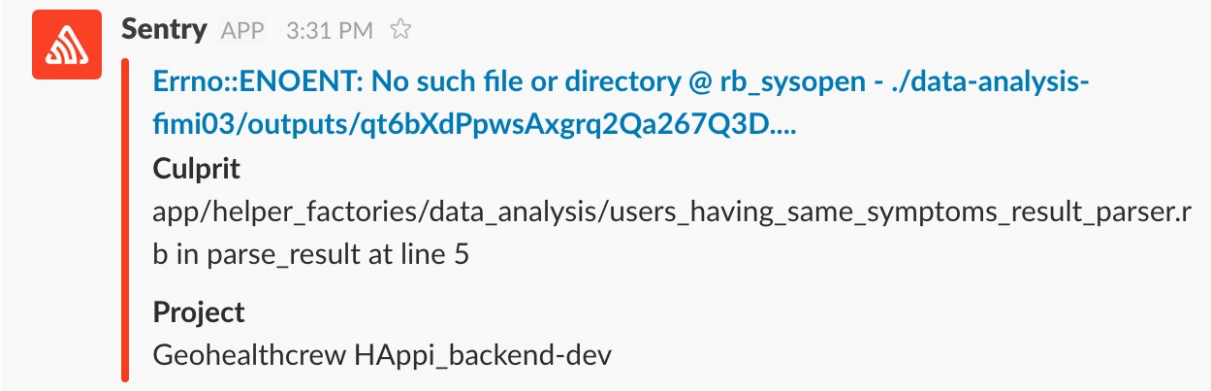


Figure 5.1: Alert from Sentry into Slack

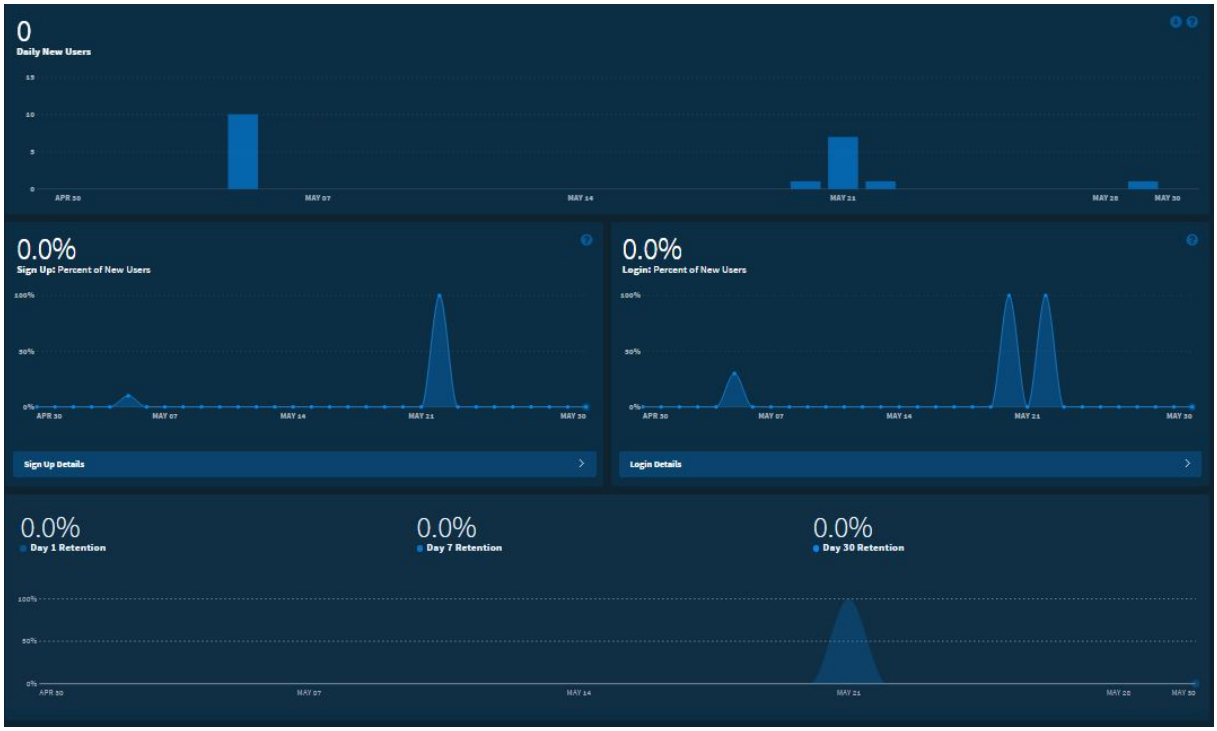


Figure 5.2: Fabric dashboard showing the number of users per day

was easy to integrate it into our *Ionic* application.

Fabric also provides a users tracking tool called *Answers*. Thanks to this tool, we can see in live the number of users using our app, the time they spend on the app and which version they are using. This service is also included in the *cordova-fabric-plugin* library.

The figure 5.2 shows an example of dashboard that *Fabric* generates.

5.7 Continuous Integration tools

5.7.1 Travis CI

TravisCI[22] is a distributed Continuous Integration service used to build and test software projects. It is integrated into Github repository. The configuration is done by adding a file named *.travis.yml* to the root directory of the repository. On each commit, Github triggers the build on

[22] `https://travis-ci.org`

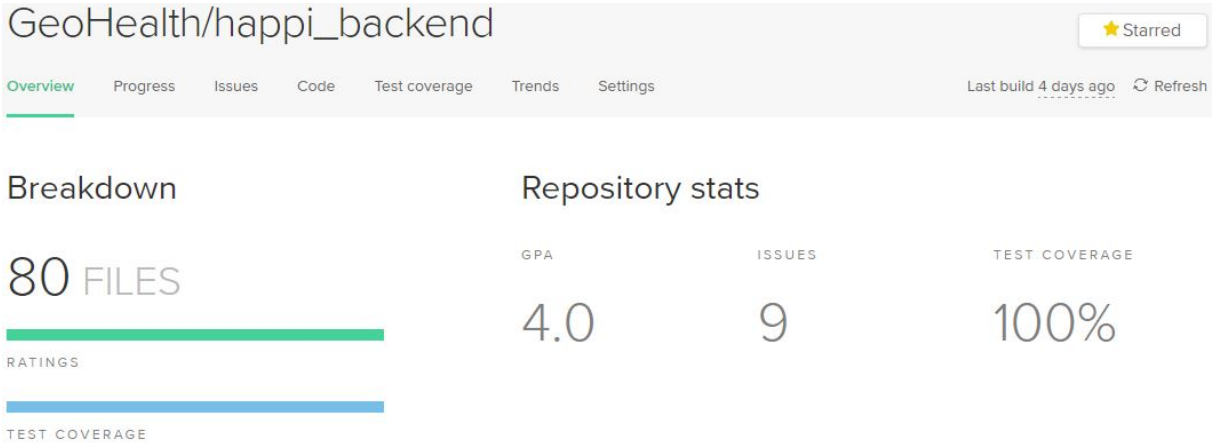


Figure 5.3: Codeclimate - Dashboard for the backend

TravisCI.

We use *TravisCI* on each of our projects, you can check the builds at the following address: `https://travis-ci.org/GeoHealth`

Thanks to this tool, we can run our entire test suites on each commit pushed to Github and be warned of the results into our Slack.

TravisCI is free for open-source projects like ours. For private projects, another version of *TravisCI* exists at `https://travis-ci.com`

5.7.2 Codeclimate

Codeclimate[23] is a code review tool that analyzes for free the public repositories on Github. It provides a dashboard for each of our projects, the figure 5.3 illustrates the dashboard of our backend.

It checks the quality of our code based on rules that we defined. It evaluates our project thanks to some ranking systems :

- Quality GPA: GPA (Grade Point Average) is an overall of all individual grades of files present in the project which ranges from 0.0 to 4.0. The grades are calculated based on the quality of the code which is itself based on the issues detected by some analyses engines. The figure 5.4 shows an example of GPA attribution to some files of the backend.
- Test coverage: indicates the percentage of code covered by the tests.
- Churn vs. quality: *Codeclimate* attribute a churn metric that is approximately the number of times a file changed during the last 90 days. A file of low quality that changes a lot should receive special attention. A graph is available (example shown on figure 5.5) showing the churn of the files on the x axis and the quality on the y axis. A file that is on the bottom left is safer than a file on the upper right side.

Codeclimate is also able to detect issues. To do so, it uses analyzing engines. These engines (like *ESlint*, *RuboCop*, etc) inspect the code and detect any potential issues or bad practices. *Codeclimate* can be configured to report those problems directly into Github as issues.

Codeclimate can automatically detect the languages used in a project and generate a default configuration but we can also explicitly specify the configuration in a YAML file named *.codeclimate.yml*. In this file, we can specify the analyze engines to use, which paths should be analyzed or ignored, etc.

[23] `https://codeclimate.com`

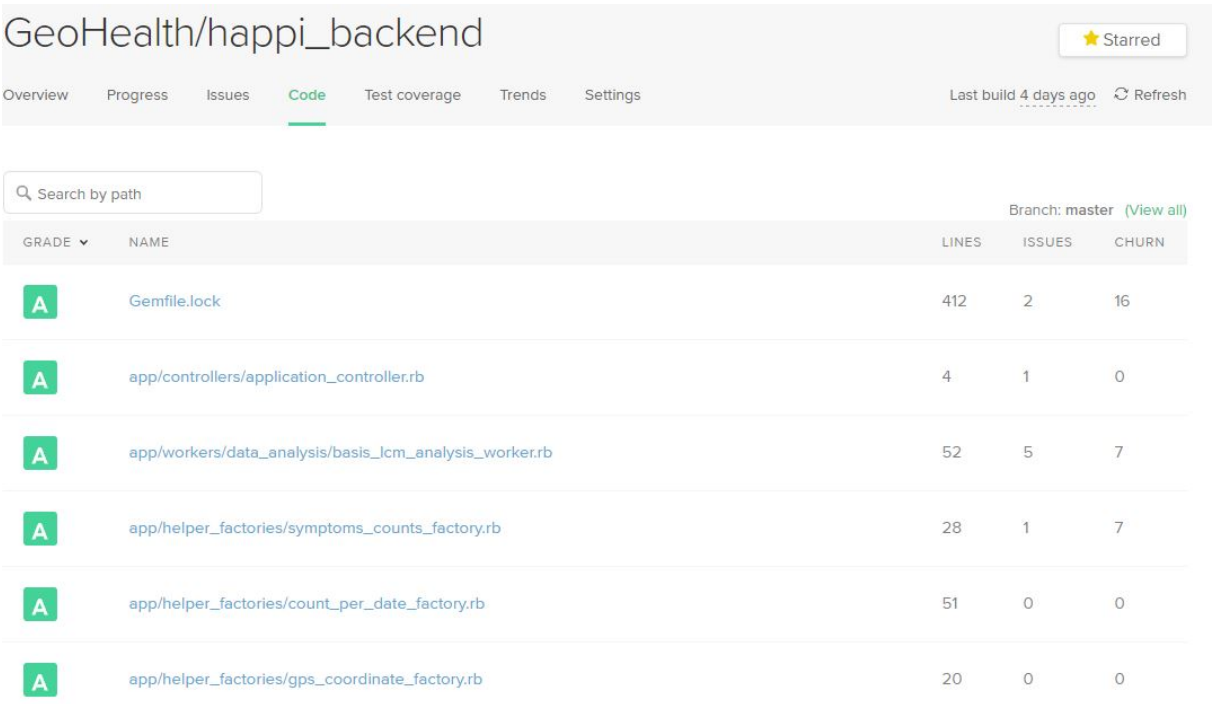


Figure 5.4: Codeclimate - GPA attribution to files of the backend

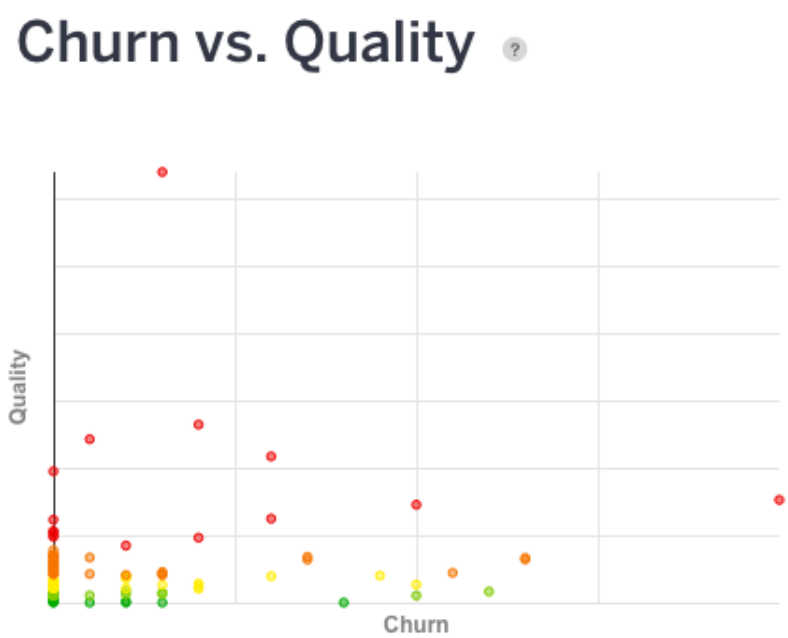


Figure 5.5: Codeclimate - Churn versus Quality - Indicate the churn and quality of files. The lower the better

5.7.3 Flynn

Flynn[24] is an open source *Platform as a Service* (PaaS) that can easily deploy, load-balance and scale applications. It creates running applications directly from source code. With this kind of tool, we avoid manual deployment of applications. This is a good thing because manual deployment are error-prone since it is easy to forget adding a dependency or an environment variable.

Also, *Flynn* includes built-in *Postgres*, *MySQL*, *MongoDB* and *Redis* databases with high availability that are very easy to deploy and link to existing applications. It can deploy many different applications based on *Heroku buildpacks*[25] (a set of scripts responsible for transforming source code into a running application).

Flynn works by creating a cluster of instances and performing load-balancing among these instances. In our case, we did not build a cluster because we only have one instance of *Flynn*.

Flynn makes app scalable thanks to the container mechanism. It is easy to add or remove instances of an application by adding or removing a running container. It is also easy to add new routes for existing application, i.e. create a new subdomain pointing to that application. We used this functionality in particular to add our *Let's Encrypt* SSL certificate to the backend and frontend for doctor.

Thanks to a special *release* mechanism, *Flynn* is able to deploy a new version of an application with no down-time: it builds the new version of the application, deploys it into a new container and redirects traffic with no down-time to this new version. If everything goes fine for a few seconds, it shutdowns the previous version and keeps only the new container. If the new version fails somehow, it switch back to the previous stable release.

We installed *Flynn* on our VPS from OVH. On this instance, we deployed *Hubot*, the frontend for doctors and three instances of the backend. Once *Flynn* is installed, it can be managed using a CLI or a web interface[26]. The figure 5.6 shows the dashboard provided by *Flynn* to manage the *prod* environment of our backend. We can scale the app up or down, update environment variables, deploy a specific commit (from Github) or a previous release, manage the routes of the application, add a new database or access the logs.

5.7.4 Hubot

Hubot[27] is a chat bot developed by Github that automates tasks. It can be integrated in a chat like *Slack*. *Hubot* is open source, written in *CoffeeScript* and runs using *Node.js*.

It listens to messages posted on channels in which the bot was added and, using pattern matching against the posted messages, it executes a custom script. In this script, it can do anything a classic *Node.js* script can do. In addition, it can also interact with Slack (in our case) by posting messages, adding reactions to existing messages, etc.

One can easily add custom scripts to *Hubot* in order to extend the commands the bot understand. It is as simple as writing a script in *Coffeescript* and shipping it with the source code of the bot. We can also use scripts developed by other users by adding them as dependencies of the bot. Since it is based on *Node.js*, we can benefit of *NPM*.

With *Hubot*, we can automate almost everything we want. It can simply remind us how much time we have before a deadline or deploy our backend on *Flynn*.

[24] `https://flynn.io`

[25] `https://devcenter.heroku.com/articles/buildpacks`

[26] Our instance of *Flynn* has a management web interface at `https://dashboard.happi-doctor.be`

[27] `https://hubot.github.com`

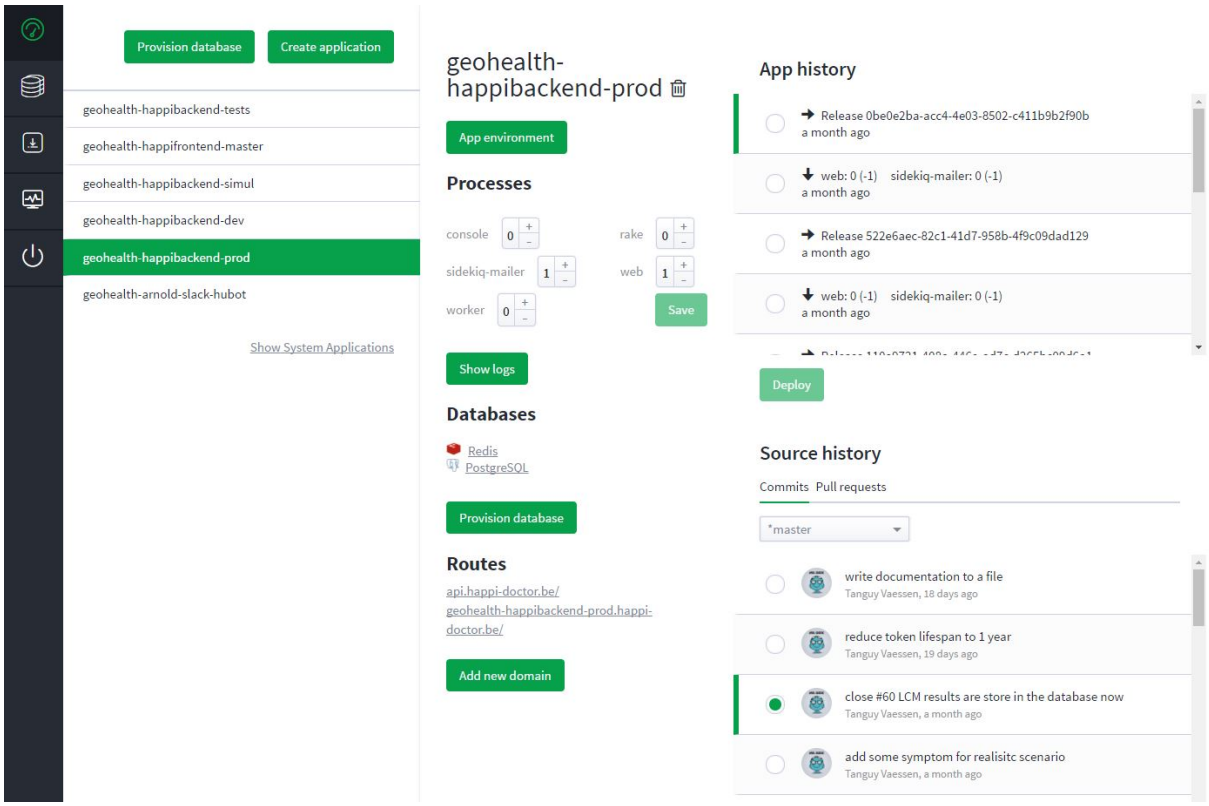


Figure 5.6: Flynn dashboard to manage the backend in *prod*

5.7.5 Swagger

Swagger[28] is a powerful open source framework that we used to design and document our REST API. A good practice is to define the headers of requests before developing them. That is what we have done, working with *Swagger Editor*[29]. We define in a YAML file the specification of each request (path, headers, parameters) and response (headers, status, body).

The swagger file of our API is available on Github at `https://github.com/GeoHealth/HAppi_backend/blob/master/swagger.yaml`. We have also generated an HTML version of the REST API that is available at the following address: `https://geohealth.github.io/HAppi_backend`

5.8 Testing tools

We wrote lots of tests during this thesis hence we used many tools to help us.

5.8.1 MailCatcher

MailCatcher runs a SMTP[30] server and a web interface where we can see the emails that are received by the server. We used *MailCatcher* during our development process to verify that emails were correctly sent.

This tool is not used during automatic tests like unit tests or integration tests. It is only used for development purposes. During automatic tests, no emails are sent.

[28] `http://swagger.io`
[29] `http://editor.swagger.io`
[30] *Simple Mail Transfer Protocol*

```ruby
RSpec.describe DataAnalysis::BasisAnalysis, type: :model do
  describe 'attributes' do
    it {should validate_presence_of(:threshold)}
    it {should validate_presence_of(:status)}
    it {should validate_inclusion_of(:status).in_array(%w(
     created running done aborted dead ))}
  end

  describe 'after save' do
    before(:each) do
      @analysis = build(:basis_analysis)
      @analysis.save
    end

    it 'has a generated token' do
      expect(@analysis.token).not_to be_nil
    end
  end
end
```

Listing 5.2: Rspec unit test example on a Model

5.8.2 Guard

Guard and more specifically *Guard::RSpec*[31] allows to automatically launch tests when a file is created or modified. It is helpful to speed up *Test-Driven Development* because it avoids us to have to manually start the tests suite.

5.8.3 Rspec

Rspec[32] is a testing framework for *Ruby* with which it is easy to make *TDD*. It allows to test all the parts composing a *Rails* application: models, controllers, mailers, *Sidekiq* workers, etc.

Rspec makes *TDD* easy by giving us the ability to write high level tests. The snippet 5.2 shows an example of unit test we wrote for a model. We can see that the test can be read in natural language. For example, for the first test: *"BasisAnalysis attributes should validate presence of threshold"*, or for the last test: *"BasisAnalysis, after save, it has a generated token"* Actually, *Rspec* can be used to generate some documentation using the command `rspec --format documentation --out rspec.txt`. A documentation generated from the tests is what we call a *"Living Documentation"* because it is always up-to-date by nature. Indeed, the tests must always pass, otherwise the application is broken.

The figure 5.7 shows the output provided by *RSpec* when running all the tests on our backend. On this figure, we can see that the last line talks about randomization. We configured *RSpec* to run our tests suites in random order to be sure that the order of the tests does not impact the success.

5.8.4 Spring

Spring[33] is a *Rails* application preloader. It speeds up the development by keeping your application running in the background so you do not need to boot it every time you run a test, rake task or

[31] `https://github.com/guard/guard-rspec`
[32] `http://rspec.info`
[33] `https://github.com/rails/spring`

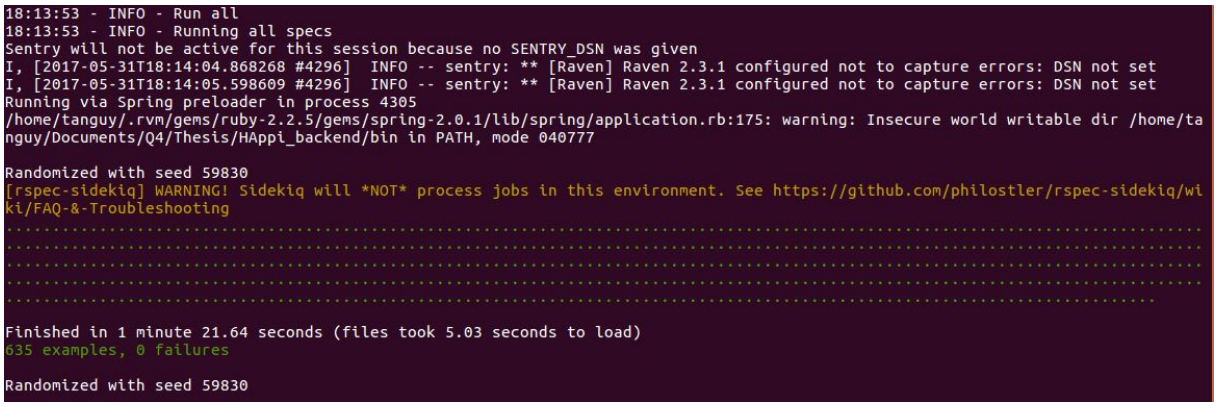


Figure 5.7: RSpec - All tests suites of the backend

```ruby
factory :symptom do
    sequence(:id)
    name 'Abdominal pain'
    gender_filter 'both'
end
```

Listing 5.3: Factory Girl - Declaration of factory for Symptom model

migration.

If any file of the application or the tests is changed while *Spring* is up, you do not need to restart the background process. However, if a file that is used to start the application (like a configuration file) is changed, you need to restart the background process. To do so, *Spring* provides a CLI to see the status of the background process and stop it.

5.8.5 Factory Girl

Factory Girl[34] is a solution to replace default fixtures in *Rails*. It has a straightforward definition syntax, supports multiple build strategies and multiple factories for the same class. It also supports factory inheritance. It is available as a gem for *Rails*.

To declare a *factory*, we add a code like the one shown on snippet 5.3. On this snippet, we declare a *factory* for the *Symptom* model. Then in our test, we can `build` or `create` one or multiple symptoms. Of course, every attribute specified in the *factory* can be overridden. For example, the snippet 5.4 shows the creation of ten occurrences that belong to the same user. These occurrences are automatically added to the database.

We could have used another build strategy to only build the occurrences without adding them to the database as shown in snippet 5.5

5.8.6 Mocha

Mocha[35] is a *JavaScript* test framework running on *Node.js*. It allows us to make asynchronous testing. We used it to run tests suites for the mobile application and the frontend for doctors.

[34] `https://github.com/thoughtbot/factory_girl_rails`
[35] `http://mochajs.org`

```ruby
create_list(:occurrence, 10, user_id: @user.id)
```

Listing 5.4: Factory Girl - Creation of 10 occurrences with an explicitly specified `user_id`

```
1 build_list(:occurrence, 10, user_id: @user.id)
```

Listing 5.5: Factory Girl - Build 10 occurrences with an explicitly specified `user_id`

5.8.7 Jasmine

Jasmine[36] is an open source testing framework for *JavaScript*. This is the recommended test framework for *Angular*. It supports *Test-Driven Development*, allows to stub Javascript objects, handles asynchronous code execution, etc.

5.8.8 Protractor

Protractor[37] is an end-to-end test framework for *Angular JS* applications. It runs tests in a browser by simulating user interactions and making assertions on the elements present in the web page.

5.9 Scrum tools

During a project, tools for communication, information sharing,... have a very important place. In the case of this thesis we were just a team of two so communication was not such a big problem but for bigger teams, it can be very hard to share information efficiently. We used a bunch of tools to manage our project and ease the communication.

5.9.1 Slack

Slack[38] is a famous tool that organizes team communications into channels dedicated to specific topics and also allows direct messaging. It can be extended with lots of integrations.

We used this tool for all our exchanges with our supervisor but also for our private debates about the projects. We integrated *TravisCI*, *Codeclimate*, *Sentry*, *Fabric* and *Hubot* to have relevant information about the status of our projects.

5.9.2 Github

Github[39] is a very popular website for managing project's versions using *Git*. We used it to host our code publicly, communicate about specific lines of code, report issues, etc.

5.9.3 ZenHub

We used the Kanban[40] methodology to display different cards representing stories and their progress. We decided to use *ZenHub*[41], a tool that integrates natively with Github's user interface. It allows us to view the Github issues in a dashboard. We have therefore defined our stories as Github issues. This is very useful because we can directly reference a story in a commit by using the Github notation (for example #75 references the issue 75).

Thanks to *Zenhub*, a team can assign "story points" to issues, define milestones (this is a Github functionality) and keep track of the velocity of the team across sprints.

[36] `https://jasmine.github.io`
[37] `http://www.protractortest.org`
[38] `https://slack.com`
[39] `https://github.com`
[40] Kanban is methodology to manage projects composed of a visual board, divided in column containing cards. A card represents a task. Each card moves from column to column depending on its progress state.
[41] `https://www.zenhub.com/`

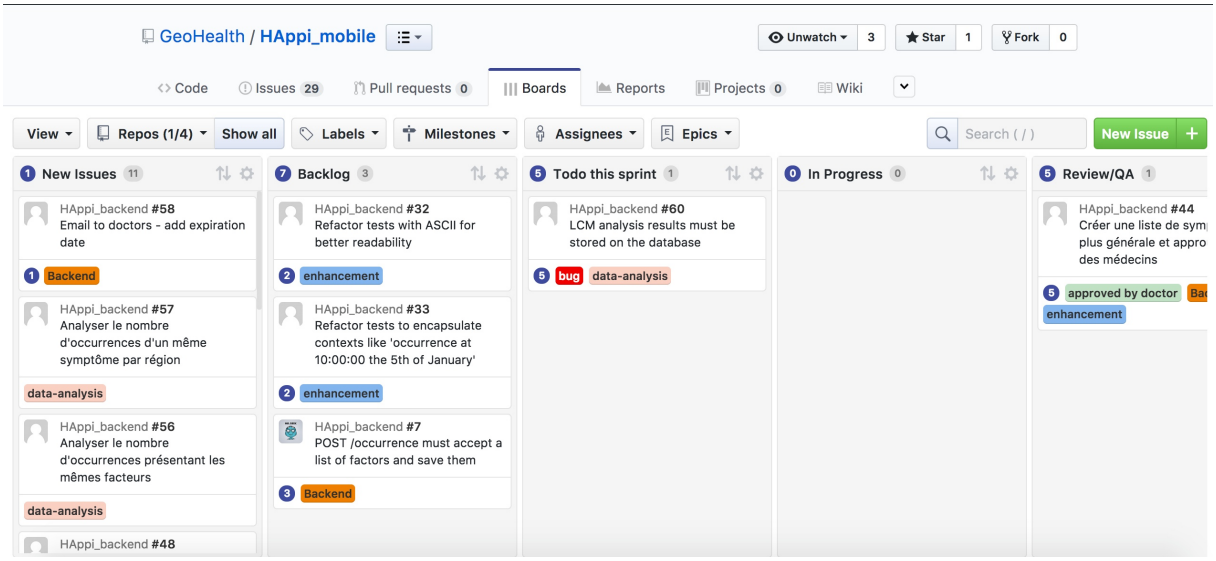


Figure 5.8: Zenhub dashboard for the backend

The figure 5.8 shows a version of our *Zenhub* dashboard and the figure 5.9 shows the velocity of our team.

5.10 External services

The backend uses external services for two tasks: sending mails and getting weather information.

5.10.1 Mailgun

Mailgun[42] is the email service that allows us to send and track emails. In our thesis, we used *Mailgun* to send emails to doctors containing reports from the users. We used a free account so we have some restrictions: we cannot send more than 200 emails per day and the recipients email addresses must be pre-validated[43].

5.10.2 Weather Underground

Weather Underground[44] provides an API thanks to which we can get information about the weather. For a date and a position, the API gives us the percentage of humidity, temperature and other details about the weather (cloudy, sunny,...).

5.11 Data analysis

To analyze big volumes of data, we decided to use *Elasticsearch*, *Kibana* and *LCM*. *Elasticsearch* and *Kibana* are linked because *Elasticsearch* provides the data to *Kibana*. *LCM* is not really a tool but an algorithm that solves the problems of Frequent Itemset Mining.

5.11.1 Elasticsearch

Elasticsearch[45] is an open source search engine. It provides a distributed, multi-entity search engine through a REST interface. It is a free software written in *Java* and published in open

[42] `https://www.mailgun.com`
[43] If you want to test to send a report, ask us to register your email address
[44] `https://www.wunderground.com/weather/api`
[45] `https://www.elastic.co/products/elasticsearch`

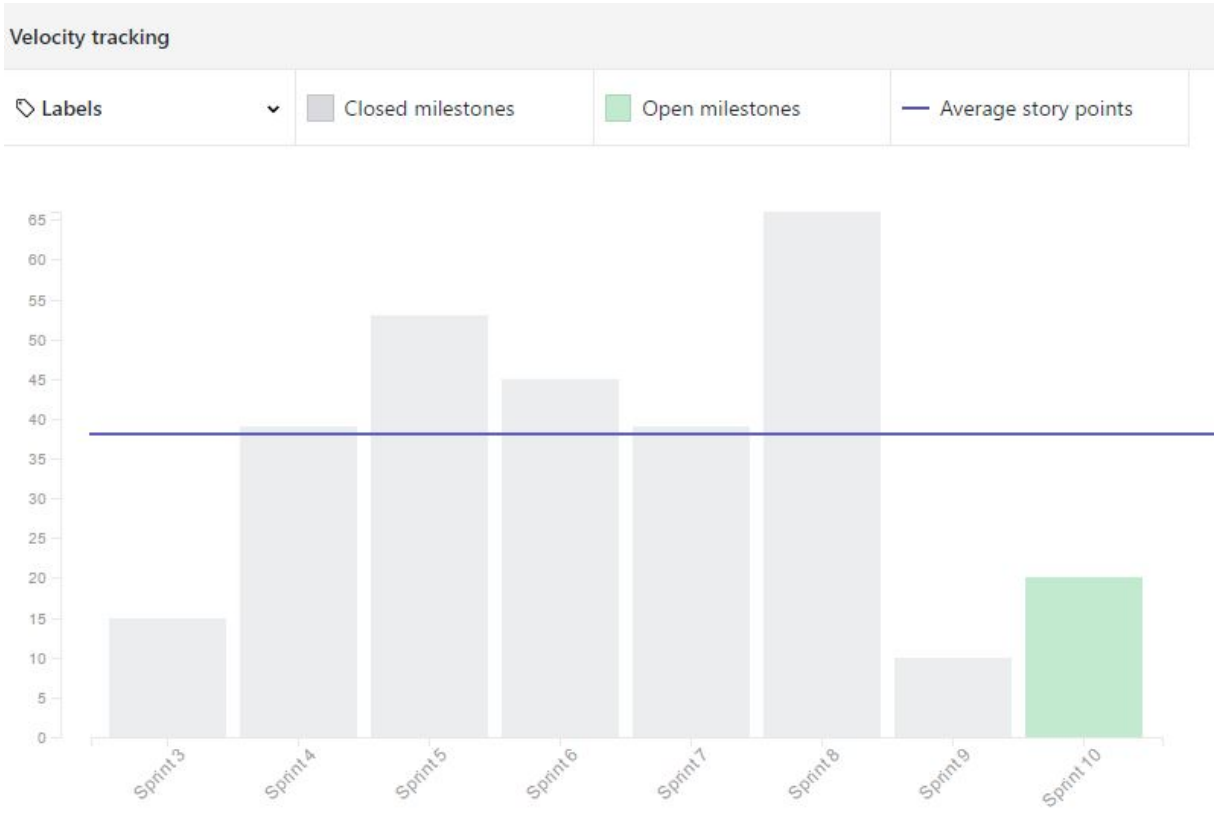


Figure 5.9: Zenhub velocity tracker

source under Apache license.

It can receive data in any format from anywhere and is able to analyze and search it.

5.11.2 Kibana

Kibana[46] is an open source tool that allows to visualize data from *Elasticsearch*. It allows the user to create many types of charts and show maps based on large volumes of data.

5.11.3 LCM

LCM (short form for *Linear time Closed item set Miner*) is an algorithm that addresses the problem of Frequent Itemset Mining, defined in the paper *LCM: An Efficient Algorithm for Enumerating Frequent Closed Item Sets* writen by Takeaki Uno, Tatsuya Asai, Yuzo Uchida and Hiroki Arimura [14]. It is available only for academic use.

The algorithm is available in three versions described below.

1. **All** No filter on the output sets
2. **Closed** Output only the closed sets[47]
3. **Max** Output only the maximal sets[48]

We built the program from its sources using the option `DFREQSET_PROBLEM=2` to produce the *closed* version of the algorithm. This is the version that gives us the most relevant results

[46] `https://www.elastic.co/products/kibana`
[47] A closed set is either a maximal set or a subset that appear more often than its superset
[48] A maximal set is a set that is included in no other sets

```
1 2 10 20
1 2 25 22
1 2 10 22
```

Listing 5.6: *LCM closed* - small input

```
22 1 2 (2)
10 1 2 (2)
2 1 (3)
```

Listing 5.7: *LCM closed* - output with a threshold of 2

when analyzing the frequency of symptoms. It gives us the most frequent sets of symptoms that appear together and also the biggest sets. It was compiled for Ubuntu 14.04.

LCM can be run using the following command line: `fim_closed input-file threshold output-file` The format of the input file is a list of sets of integer, each number of a set is separated by a space and each set by a line feed. The threshold is a positive value used by the algorithm to restrict the output length. *LCM* reads the input file and outputs the subsets of values that appeared in at least *threshold* sets. The output file is composed of lines respecting the format `[value1] [value2] [...] (X)` where X is the number of time the subset of values has appeared in different sets.

For example, running *LCM closed* with the input shown in snippet 5.6 and a threshold of 2 produces the output shown in snippet 5.7.

5.12 Certificates

During our master thesis, we deployed a backend and a website. Because we are manipulating sensitive information, we wanted it to be secured by TLS. Because we are using *Flynn* to host those softwares, we already had a self-signed certificate. But a self-signed certificate is not trusted by any device so we needed to obtain a valid certificate in order to deploy our mobile application to the Play Store and make our website accessible through HTTPS.

5.12.1 Let's Encrypt

Let's Encrypt[49] is a *Certificate Authority* (CA) that provides SSL/TLS certificates accepted by most of modern browsers. To obtain a certificate, one must prove its control over a domain name by hosting a special file or adding a DNS entry to a subdomain. Additionally, during the process, a public-private key pair is generated and the server must sign with its private key a nonce provided by *Let's Encrypt* to prove its control over the key pair[50]. The generated certificate is valid during 90 days[51].

Obtaining a certificate from *Let's Encrypt* is free because it is a nonprofit organization funded by sponsors. Their goal is to promote the usage of HTTPS.

5.12.2 Certbot

To retrieve a TLS/SSL certificate from *Let's Encrypt*, we used *Certbot*[52]. This software is used to generate, renew and revoke a certificate and is compatible with lots of existing web servers

[49] `https://letsencrypt.org`
[50] `https://letsencrypt.org/how-it-works/`
[51] `https://letsencrypt.org/2015/11/09/why-90-days.html`
[52] `https://certbot.eff.org`

like *Apache* and *Nginx*.

Unfortunately, it does not support *Flynn* natively. So to add a SSL certificate to our routes in *Flynn*, we had to generate the certificate manually using the command `certbot certonly --manual --preferred-challenges dns`. It starts an interactive client to generate a certificate that will be valid for some specific domains and it verifies that we are the owner of the domains by doing a DNS validation. Once we obtained the certificate, we need to add it to the routes of *Flynn*. This can be done easily with this command: `flynn route update ROUTE_ID --tls-cert cert.pem --tls-key privkey.pem`

Chapter 6

Improvements

If you take a look at our *Zenhub* dashboard [1] , you might see that we still have a lot of "stories" to do. Indeed, this kind of project is never completely finished because we can always add new features, develop new ideas and improve existing functionalities.

In this chapter we talk about some points that could be improved or added.

6.1 Development environment

6.1.1 Docker

During the development of our project, we got a problem with dependencies. Sometimes the application, backend, ... works on a machine and not on another. That is a serious problem and a waste of time.

With *Docker*[2], we can easily avoid this kind of problem. *Docker* is an open-source project that can run applications inside software containers. *Docker* can package an application and its dependencies in an isolated container, which can be run on any Linux server. Anyone can run the application thanks to docker without needing to install all the dependencies.

Furthermore *Docker* provides more automation. We do not need to install the technologies. Thanks to *Docker*, we stop wasting time installing and maintaining software on servers and developer machines. All dependencies run in containers, eliminating “works on my machine” problems. Finally we can easily deploy the application by providing the containers to the server without any installations.

6.2 Security

Even though our master’s thesis is not focused on the security aspect, we tried to provide a minimum of security. In this section, we explain what can be added to improve even more security.

6.2.1 Vault by Hashicorp

In our applications, we manipulate sensitive information. This data must be as secured as possible. A solution is to use *Vault*[3]. *Vault* is a tool for securely accessing data. It generates a certain number of keys, in addition to the initial root token. These keys unseal the vault. The protection is based on key sharing, according to the *Shamir algorithm*[4], it is necessary to have a

[1] You need to use Google Chrome, install the Zenhub extension and create an account to be able to see the dashboard. However, an example is illustrated on figure 5.8

[2] `https://www.docker.com/`

[3] `https://www.vaultproject.io`

[4] `https://en.wikipedia.org/wiki/Shamir's_Secret_Sharing`

certain number of keys to access the vault. The interest is to encrypt these keys with the keys of different people to ensure that a quorum is necessary to (un)seal the vault.

Thanks to *Vault*, users information could have an optimal protection. Someone who wants to access the data should have access to the keys. Each key is stored on a different device.For the hacker, it is almost impossible to get these keys.

6.2.2 One-time token

As discussed in the section 3.5.5, an improvement that could be made to improve security of the mobile application is to make the token provided by the backend via *Devise* change at each request. This is actually the default behavior but we deactivated it to make it easier for us to manage the authentication headers and not to have to update them after each request. We should have used the *Angular* library called *angular2-token*[5] to avoid managing the token ourselves and benefit from the changing-token policy built in *Devise*.

This improvement would reduce the probability that an attacker who eavesdrops on communication can impersonate a user because the token that the attacker heard would not be valid anymore for the next request.

6.2.3 SSL pinning

"SSL pinning is a mechanism to limit the number of Certificate Authorities (CAs) trusted to sign for a specified website" [7]. SSL pinning can also be used to configure an application to trust only some specifics certificates (instead of trusting certificates signed by a particular CA). Basically, we add locally into the configuration of an application the list of SSL/TLS certificates it can trust when accessing a remote server. Thanks to this list of certificates, the application does not have to trust any of the CAs that exist. When it connects to a server and validates the server's certificate by looking at its own list of trusted certificates.

It is mainly used to reduce Man-in-The-Middle attack where an attacker somehow obtained a certificate from a trusted CA or was able to manipulate the client's system Truststore[6] to add a bad CA. This is a security feature that we would like to implement in our application because we have sensitive information going through the network and this solution seems easy to implement.

Actually, it is not that easy to pin a certificate in an *Ionic* application. While Android provides a native way to do so [7], *Cordova* does not [8]. Luckily for us, *Ionic2* does support SSL pinning natively [9] using its *HTTP* component but this is **not** the component that we used to manage HTTP connections: we used the *HTTP* component provided by *Angular2*. The problem with the *HTTP* component from *Ionic* is that it only supports the *GET* and *POST* HTTP methods while we also need the *DELETE* method.

So to implement SSL pinning into our application, either we use both components simultaneously, the one from *Ionic* with SSL pinning for *GET* and *POST* and the one from *Angular* for *DELETE* but this is not very maintainable. Or we get rid of the *DELETE* methods and transform them into *GET* or *POST* but it reduces the coherence of our API.

6.2.4 Local encryption

Smartphones can be attacked and private information can be stolen. To prevent this kind of leak, we could encrypt the data stored in the local databases. To be able to use the application,

[5] `https://github.com/neroniaky/angular2-token`

[6] The Truststore is the store that contains the certificates that an application trusts or CAs that it trusts to identify other parties.

[7] `https://developer.android.com/training/articles/security-ssl.html#Pinning`

[8] `https://cordova.apache.org/docs/en/latest/guide/appdev/security/#certificate-pinning`

[9] `https://ionicframework.com/docs/native/http/#enableSSLPinning`

the user would have to provide a password known only by himself, otherwise the data would be unreadable.

6.2.5 One-Time Password

To push further the previous improvement about local encryption, our mobile application could ask, in addition to (or instead of) a password, a One-Time Password. Only then, the local files would be decrypted.

A "One-Time Password" or OTP[10] is a password that is valid only during a short period of time. After this period, it expires and another password must be used. Some famous companies (like Google, Facebook, Microsoft, etc) suggest to use this kind of protection. They call it the "two-factors authentication" because the password of the user (first secret) is not enough to get access to his account, an attacker would also need access to his unlocked smartphone (or whatever device he uses to generate his OTP) to get the OTP (second secret). This is a good security improvement because passwords can easily be stolen or guessed and are not sufficient as a mean of authentication.

The OTP is generally generated by a mobile application like *Google Authenticator*[11].

6.3 Data mining

6.3.1 LCM

Symptoms have same factor values

Currently, we use *LCM* to highlight users having the same symptoms. We could do more with *LCM*: we could regroup symptoms having the same factor values. This would demonstrate correlations between factors and symptoms and highlight a factor that potentially cause the appearance of a symptom.

Symptoms in same region

Another usage of *LCM* could be to highlight the symptoms that appear in the same geographic region. However there is a limitation if we implement this. The implementation of *LCM* we used can only take a matrix of integers as input. So we would have to divide the Earth in regions represented by integers. If two occurrences are very close but in different regions, the algorithm would not mark them as being close.

Web interface

The web interface we have built to use *LCM* more easily is integrated into our backend for now. We realized that it could have been better to develop it in a dedicated *Ruby gem* so that it could be reused by anybody by simply adding the gem to a *Rails* project. Doing so, other people could benefit of this web interface and use *LCM* in their project.

The main limitation here is that the implementation of *LCM* is available for academic uses only so we could never publish this gem to an official repository.

A second improvement for the web interface is to restrict its access by adding an authentication mechanism. Currently, anybody knowing the URL can access the interface and run *LCM* jobs or access the results of previous jobs. They are multiple risks:

- A malicious person can access confidential data
- A malicious person can overload our server by running too many jobs

[10] `https://en.wikipedia.org/wiki/One-time_password`

[11] `https://play.google.com/store/apps/details?id=com.google.android.apps.authenticator2`

A simple login form to restrict the access to the *LCM* web interface seems to be an important improvement to make.

6.4 Continuous Delivery

We have deployed our mobile application very late to the Google Store but once it was done, a major improvement would be to make this deployment automatic each time a commit is pushed into Github and tests pass on *TravisCI*. To do so, we could use the same mechanism as we did for the backend: make a script for*Hubot*. This script would react to messages posted by *TravisCI* and push the new version of the application to the Google Store thanks to the *Google Play Console* API. This continuous delivery seems to be the next logical step to take for us.

6.5 Mobile application

The mobile application can be improved in many ways. The main improvements are the following:

- Read data from external sensors, like a heart rate tracker or a blood pressure sensor. Include this information as an occurrence's factor
- Add pictures of environment or food to help tracking symptoms' origin
- Smart symptoms re-ordering and suggestions: based on the most recently used symptom, we can order the symptoms list on the home screen and propose new ones
- Alert the patient when a correlation is established between multiple symptoms

6.6 Other ideas

Alternative usage with SMS

In some countries, especially in Africa, people do not have easy access to internet but can send SMS. A nice improvement, to make our solution usable in those conditions, would be to add a SMS layer responsible for receiving text messages and interact with our existing API. Thanks to such a layer, we could collect lots of information coming from countries where epidemic diseases can be very deadly and, maybe, help to prevent them.

Bluesquare

Bluesquare[12] is a company specialized in development of health and education tools in low and middle income countries. A nice step for our solution would be to be adapted and integrated in some of their solutions to help them collect data in those countries.

Factors

Something really useful is to add factors to a specific symptom. Such as when someone has a headache, he could indicate the intensity of the pain or if he ate before. So an improvement could be to add specific factors to each symptoms that could help even more the doctor to make his diagnosis. Adding this features would not be really hard because the model of factors is already implemented on backend side. We should only implement specific views for each symptom to display their own factors. This feature must be performed with the help of a doctor to select the factors that are relevant for each symptom.

[12] `https://bluesquarehub.com`

Chapter 7

Sprint reviews

During the realization of our master thesis, we performed different sprints as explained in the section Scrum 2.1.2. During a sprint, we implemented features explained in different stories selected at the beginning of the sprint. For each sprint, we computed our velocity and tried to improve it from sprint to sprint. The velocity is defined by the sum of the stories points selected for the sprint. We explained how we have attributed points to stories in the section 2.1.2.

To represent the stories we used issues from *Github* (5.9.2) and *ZenHub* (5.9.3). That allows us to have a *Kanban*[1] inside *Github* that displayed our stories. Moreover *ZenHub* provides a feature that represents a sprint with a chart. The X axis represents the time (for us, 2 weeks except bonus and last sprints). The Y axis represents the story points. The dotted line indicates the ideal workflow and the solid line depicts the actual workflow. So we had to aim for the solid line to be below the dotted line.

Sprint 1

Start: Nov 21th, 2016
Due: Dec 4th, 2016
Story points: 16
Burndown chart: fig B.1

Description

During this first sprint, we have set up the development environment. We have developed the following features for the mobile application: add, delete, display symptoms and register an occurrence with GPS position.

We attributed one point to all stories, because it was our first stories estimation and they seemed simple to us.

Achievement

We finished this sprint just in time. The configuration of the development environment was a little bit harder than we thought. We got some problems with the configuration of *TravisCI*.

Sprint 2

Start: Dec 5th, 2016
Due: Dec 18th, 2016

[1] `https://en.wikipedia.org/wiki/Kanban_(development)`

Story points: 23
Burndown chart: fig B.2

Description

During this sprint, we planned to develop the creation of occurrences and symptoms on the backend side. We also planned to deploy the server online and finally to add Google Maps to the application.

Achievement

We were blocked for a long time and therefore did not finish a card during this period. Initially, we had a server provided by the INGI department of UCL. *CentOS* was installed on this server. To deploy the backend we tried to install different *PaaS* on the server: *Flynn*, *Dokku*, *Tsuru* and *Open Shift*. All these technologies worked only on *Ubuntu*. Finally, we found a solution: to run *Open Shift* on *CentOS* inside a *Docker* container. But this solution did not work as we wanted. We also encountered problems using the Google Maps developer key. It was also the first time we used TDD to develop. The learning curve for *TDD* is really important, that is why at the beginning we had many problems in applying it. Because of the problems encountered, we finished this sprint a bit late.

Learning

1. The choice of the operating system on a server is really important. Before making the choice, we must look at the compatibility of the technologies we want to use with the different Operating Systems.

2. We must avoid making too broadly scoped (not precise or too global) cards (user stories) in order not to waste time developing parts that are not necessary

Sprint 3

Start: Dec 19th, 2016
Due: Jan 1st, 2017
Story points: 15
Burndown chart: fig B.3

Description

This sprint was smaller due to the beginning of the blocus and the ending years events. It also started later because of problems encountered during the previous sprint. During this sprint, we decided to write additional tests and set up support for different languages for the mobile application.

Sprint blocus

Start: Dec 26th, 2016
Due: Feb 5th, 2017
Story points: 16
Burndown chart: fig B.4

Description

As during the exams we had free time, we decided to do a special sprint. During this sprint, we decided to set up the test coverage of *Codeclimate* for the backend, to refactor some code and to add a new functionality to the backend to retrieve symptoms filtered by name. We have also populated the database with some symptoms.

Sprint 4

Start: Feb 6th, 2017
Due: Feb 19th, 2017
Story points: 39
Burndown chart: fig B.5

Description

As we encountered a lot of problems with the server provided by the INGI department, we decided to rent a VPS with *Ubuntu* on OVH. The first part of this sprint was to install the server and use *Flynn* to deploy the backend. After we developed new features for the mobile app: the home screen, intelligent search for symptoms, the fact that the user can remain logged... Finally we deployed the application Ionic View[2].

Achievement

This sprint was bigger than the previous ones. But we finished this sprint five days in advance. This is due to the fact that we had overestimated three user stories. We thought that realizing the authentication would have been more difficult. In the end it was very simple thanks to the *devise_token_auth* gem.

Sprint 5

Start: Feb 15th, 2017
Due: Mar 5th, 2017
Story points: 53
Burndown chart: fig B.6

Description

As the previous sprint went very well, we decided to increase the number of story points for this sprint. During this sprint, we decided to introduce new technologies into our project: *mutation-testing* 4.4, *Sentry* 5.6.1 and *Crashlytics*. We also created a *test* environment. Finally we developed a the statistic feature on the backend and the mobile application.

Achievement

Despite the fact that this sprint was very large, we finished it in time even if the trend of the curve is located above the optimal diagonal dotted line.

[2]Ionic View is an application that allow users to access a beta version of an application and give feedback about it: `https://view.ionic.io`

Sprint 6

Start: Mar 6th, 2017
Due: Mar 19th, 2017
Story points: 45
Burndown chart: fig B.7

Description

As the previous sprint was a success, we decided to achieve 58 story points during this sprint. We decided to manage the versionning of the application. Finally we implemented the fact that a user could consult his data on any devices (his favorites symptoms are synchronized with the backend).

Achievement

The event "Les journées de l'industrie" at UCL has disturbed this sprint. We decided to improve user accounts by adding additional data. We had overestimated this sprint, so we removed 13 points that were added to the next sprint. We closed this sprint by finishing the 16 remaining points on the last day.

Sprint 7

Start: Mar 20th, 2017
Due: Apr 2nd, 2017
Story points: 39
Burndown chart: fig B.8

Description

As the last sprint did not go well, we decided to handle less story points. During this sprint, we decided to integrate new technologies into our master thesis: *Sidekiq* 5.3.1, *Mailgun* 5.10.1 and *MailCatcher* 5.8.1. We also implemented mailing to doctors. We tried to get the weather data for the occurrences. Finally we implemented the application for the doctors to allow them to visualize their patients reports.

Achievement

This sprint went pretty well. We noticed a long period without stories finished. This is quite normal because creating an application from scratch takes a lot of time. The configuration and creation of the application took time. After finishing the application for the doctors, we quickly finished this sprint.

Sprint 8

Start: Apr 3rd, 2017
Due: Apr 16th, 2017
Story points: 69
Burndown chart: fig B.9

Description

Due to Easter holidays, we had a lot of time to work on our master thesis. So we programmed our biggest sprint of the year with 69 story points. During this sprint, we worked on the anonymization of data. We also decided to finalize the application for doctors. We carried out the statistical part with *Kibana* 5.11.2, *Elasticsearch* 5.11.1 and *LCM* 5.11.3

Achievement

At the beginning we were faster than the estimations. Then, we got delayed because of a memory leak issue in our unit tests on the *Ionic* app. So we can see a long period without any issue resolved. Finally, we managed to end the sprint on time.

Sprint 9

Start: Apr 18th, 2017
Due: Apr 30th, 2017
Story points: 10
Burndown chart: fig B.10

Description

From now on, the sprints are going to be smaller because we have to dedicate our time to the writing part of our master thesis. During this sprint we developed a feature that allows doctors to save reports in pdf format. We decided to fix *TSlint* issues. And finally we have managed errors that can happen in the statistics part.

Achievement

We finished this sprint just in time because the creation of pdf had caused us problems. The Google Maps API does not allow to save a map as a pdf easily.

Sprint 10

Start: May 1st, 2017
Due: Jun 9th, 2017
Story points: 23
Burndown chart: fig B.11

Description

During this sprint, we managed to handle a sprint with 23 story points. The most important tasks were to deploy the mobile application to Google Play and to store the results of *LCM*'s analysis to the database instead of a file to fix the problem described in section 3.6.2. We also added end-to-end tests for the mobile application.

Achievement

We finished the sprint at the same time as the written version of our thesis and we followed the ideal dotted line.

Chapter 8

How to manage a project

During this master thesis, we have gained a lot of experience in development and project management. The question on quite a few people's lips is: "*How would you manage a project now?*" That is a good question but it is also a broad one. We try to answer it in this chapter.

During the past year, we used many different technologies and methodologies. All of them meet *needs*. The first thing to point out is that a technology or a methodology must not be used unless it is needed. Otherwise, it is a waste of time to learn and implement it. In addition, it can add complexity to a project.

8.1 Defining needs

The first step, is to define the needs of the project. So let us try to define what are the needs. It can be some requirements defined by customers: they want a software that is resilient to failure, they want a website that is not vulnerable to a list of attacks, etc. So we can see that the needs can be really different. For example if the customer wants a software that has a very low failure rate, we can use very advanced testing tools and techniques to be sure that the program works as desired. Using mutation testing, in this case, can be a good choice. But if the customer's needs are focused on prompt delivery of the product and not on the failure resilience, it is not necessary to spend a lot of time on testing. In this case, we should try to use frameworks and available services to save time. This does not mean that good practices can be completely ignored, testing must still be present in the project event though it can be less important.

Another source of needs is the development team and especially each member of the team. A team as a whole can have values such as: providing software of high quality or secured programs that perfectly meet the needs of customers, ... These values turn into needs. Each member of a team also has individual needs. Some people need to communicate more than others, some need that someone validates their work, etc.

So the first step to start a project is to define its needs and find the best way to meet them. To better identify the needs it is possible the classify them into different categories such as:

- Need of performance: the client has defined performance characteristics that his application must have. The performance could be the resilience to failure, the rapidity to perform requests,... Need of performance can also be defined by the development team to ensure the quality of the product.
- Need of satisfaction: when the product is finished, it must meet the customer's needs. It is necessary to be able to correctly define the needs of the customer and to be able to satisfy them.
- Human's needs: A team is composed of humans beings. They have a lot of needs and if a team wants to be productive, it has to satisfy the needs of each member of the team.

As a project is (almost) never done by a single person, the first need to satisfy is the need of collaboration. It is clearly impossible to carry out a project if the resources are not shared between the members of the team. During the implementation of the project, everybody needs to pool its part of the project. It is maybe the most important thing to do to succeed in the realization of the project. So we need to find a way to share the project (the sources) between each member of the development team. There are already solutions to set this up, using a *Version Control System* (VCS, such as: *git*, *mercurial*, *subversion*, ...) can already help tremendously in this aspect. We proscribe the usage of archaic solutions such as: sharing the code with dropbox or an USB stick. They transform a project into an uncontrollable monster. Imagine that two members of a team work independently on the same file. How do we merge the files? One team member will obviously crush the work of the other. That could be a really huge problem! Hence to avoid it, we recommend to use a VCS that manages all these things for us.

8.2 Technologies versions

To carry out a project we need tools of course. Each member of a team needs to have the same version of the tools and technologies to avoid problems such as "*this functionality is not supported by this version*". Sometimes it is easy, for example if we develop a web site in *Rails*, we simply need to have the same version of *Rails* (and *Ruby*) on every machine. But when a team needs to use a lot of different technologies it becomes harder to have the same version on every computer. The *OS* can be different and thus the installation process can also be different (sometimes a tool is not at all available for an *OS*).

One solution is to create a Virtual Machine (VM) with all the tools installed. Every member uses the VM and have the same environment. This solution can take some time to set up but it is easier to share (especially when a new team member arrives in the middle of the project).

A lighter and faster solution is to use a technology based on containers[1] like *Docker*. *Docker*[2] eases the deployment of applications by running them inside software containers. We just need to create a docker image with the technologies we need to run our project and share it with our teammates. In addition, *Docker* extensively uses a cache system to make the build of images very fast.

8.3 Team organization

When the environment is set up, we can get to work. But we must be careful because we can always meet serious problems. Especially working in team can be really hard. We can meet a lot of problems with our teammates.

First, how can we avoid that different members of the team perform the same work? We can lose a lot of time if the work is done two or three times by different team members. Hence it is really necessary to have someone that coordinates the team's work. when all the developers are on the same room, it is actually easy. But when people work from different places, what can we do?

Another problem that each team member can have is to know on what they can work. The whole team is not always together to decide who does what.

Also when we develop a feature such as "*Creation of a user account*": this is really a broad feature. We do not know what information we must ask to the user or if we have to send an email after the creation of the account. That makes a lot of questions.

[1] Containers are like VMs but they are lighter and faster because they do not emulate all the components of a computer. Instead, they share the resources of the host. However, containers are still completely isolated from the host's system.

[2] `https://www.docker.com`

Luckily, some people raised these questions and they created a lot of types of methodologies to respond to different needs.

There are a lot of *Agile* methodologies such as *Scrum*, *Extreme Programming*, etc. We do not need to stick to the letter of *Agile* principles, we just need to pick the principles which meet our needs.

In this section, we try to answer these questions: "*How to communicate effectively with your team?*" and "*How to perform daily work?*"

8.3.1 Team communication

Predicting what a team is going to do is something important. *Agile* helps us doing it. With the team, we define stories. Stories are features which are going to be developed during the project. They must be clear so that when a developer reads a story, he knows exactly what to do. For example the story *"Create an user"* is not clear enough. The first thing to do is to split the feature in many others like *"Create a login page"*, *"Save the user inside the database"*,... But they are not clear enough because a developer does not know if he must ask the name, the surname, the pseudo,... of the user. We should add comments to the stories to detail that for the creation of a user we need his pseudo and his password. In summary, we need to write clear stories with all the needed information. We can also add mockups to represent the desired screen. In fact, we can do whatever we want if it facilitates the understanding of the story.

Something important about stories is that the entire team must agree with them. So when we write stories, we need to do it with the whole team or at least the members concerned by the stories.

Another interesting thing to do is to estimate the stories by giving a score to stories based on their estimated difficulty. Easier stories have lower scores than harder stories. Adding estimations is really useful because we can estimate how much effort is needed for that specific story in comparison to other stories. And if we use cycles (iterations) to carry out our project, we can estimate how many *story points* our team is able to handle per cycle.

In practice stories can by written on post-it and put on a board following the *Kanban*[3] methodology. To improve the flexibility of the team, we can use online tools such as *Trello*[4] or *Zenhub* [5].

Communication is maybe the most important thing during the realization of a project. Do not hesitate to say when something is wrong but do not hesitate to say when something is good too. Having a good atmosphere allows a team to overcome obstacles. A lack of communication can make us lose a lot of time. When someone is stuck on a problem and communicates with nobody, he wastes his time and the time of the team. Because a project is like a puzzle, at some point the rest of the team needs the feature that this developer currently works on. So he should not waste time and, when he is stuck, he should ask the team for help.

8.3.2 Daily work

Now we are closer to the goal. But how do we deal the daily work? If we use a methodology we must think about its pros and cons. Also, we do not need to follow exactly the methodology, we can pick what we want. So do not hesitate to pick some stuffs inside different methodologies.

For example *Extreme Programming* promotes pair programming. This practice can be useful in many cases: when we use a technology for the first time, it is easier to learn it with someone else. The two developers can help each other understand things that the other would be missing otherwise. In addition, one person may already have an experience with the technology and the other can learn faster thanks to him.

[3] `https://en.wikipedia.org/wiki/Kanban_(development)`
[4] `https://trello.com`
[5] `https://www.zenhub.com`

Another case where pair programming is useful is when we develop a critical feature. Having two pairs of eyes on the screen will allow us to avoid more errors.

Finally, being with someone else will allow us to have a direct feedback on our code. The two developers can discuss about the code and learn some stuff.

Of course when we do trivial stuff pair programming can be a waste of time and we should not always use it. As explained before, we must ask the good questions before using something, see the pros and cons and speak about it with the team.

8.4 Testing strategies

In this section, we talk a bit about testing. Testing in computer science is something that is often debated: many people say anything and everything about it. Some people say *"that works so why testing? It is a waste of time!"*. They understand nothing about testing: adding tests is never a waste of time! The success of a project is linked to testing. The bigger the project, the more important tests are.

We can write tests before or after writing the code. But we must be sure that our tests are useful. Using *Test-Driven Development* techniques is good for that. But doing TDD takes a lot of time. So we encourage developers to use TDD when the code to write is critical. For example our backend is the cornerstone of our solutions and other applications use it. So a crash or a failure could create a disaster. Also when working in team with each member responsible of a different feature, tests allow them to see if something is broken when the different sources are merged.

8.5 Automate everything

In the previous sections, we can observe that the success of a project depends mostly on the time (and the budget but we were not dealing with money issues during this thesis). So to be able to win some time can be really helpful for the realization of a project. Sometimes we repeat again and again the same tasks. For example running tests. It would be really convenient to run them automatically.

There are solutions to automate tasks such as Continuous Integration. We can easily develop a bot that helps us in different tasks such as alerting the team when the build fail.

Our advice here is to automate everything that can be. If we perform a task multiple times and it is not done automatically, it is likely that errors will appear because someone forgot to do something (like deploying to production without the correct environment variables). If a script is in charge of a task, it will always behave the same way. For example we will use a script to deploy an application on our local machine and we will us the same script to deploy it to production. The script was executed a lot of times so it becomes reliable: we are sure that it works as expected.

8.6 Start simple

One last advice: when doing a project, one should start with the basic features. Try to keep these features as simple as possible, do not waste time on developing something really complicated. The most important cause of projects failures is the waste of time. When all the basic features are developed, one can try to improve his solution if he has enough time.

All these pieces of advice are based on the experience that we acquired during the development of our master thesis. We are aware that we do not give the perfect solution here and these pieces of advice are likely to change during our professional life's.

Chapter 9

Conclusion

After 10 months of work, we conclude this master thesis. We built a complete solution to help tracking symptoms of patients and ease the diagnosis of doctors. To achieve this, we have worked on four projects: a mobile application for patients, a website for doctors, a backend and a chatbot to simulate patients. But more importantly, we did those projects by using some methodologies and practices in order to improve the quality of our code, enhance the communication between us and reduce errors.

We were mainly inspired by Agile methodologies like *Extreme Programming* and *Scrum* but also by the principles of *Continuous Delivery*. We studied these methodologies from a theoretical point of view and then we applied them to our projects.

Something really important that we noticed, is the fact that during a project we did not use all the concepts of the methodologies but we picked the ones that interested us. For example, *pair programming* from *Extreme Programming*, was really useful when we used a technology for the first time because we were learning faster. But once we gained some experience and the tasks to perform were not really hard (for example creating the view of a page), pair programming became a waste of time.

Another example is *Test-Driven Development* (TDD): at first, we applied it for all our projects (backend and mobile application). But we quickly noticed that it was more efficient on the backend than on the mobile application. For the mobile application, TDD was not really necessary because all the computations and data manipulations took place on the backend. So the mobile application did not make complicated tasks. To implement simple views and models, TDD is not necessary as it can take extra time. However, we noticed that it influenced positively the quality of our code for the backend: we had almost no error and the code "looks good"[1]. That is something really nice because we know by our experience on past projects that we can spend a lot of time tracking and fixing errors in a backend.

During the development, we interacted a lot together. An IT project cannot be well performed alone. It is a team work. In a team, the communication and the comprehension are essential. If we wanted to finish all the projects in time, we needed to go in the same direction. Some methodologies helped us to manage these challenges. *Poker planning*, for example, allowed us to better understand each other: when a story was not really clear for one of us, we spoke about it and finally we had a greater understanding of the story. *Pair programming* also allowed us to improve ourselves. When we coded together, we had a lot of discussions about how to code, how to solve problems or which technology to use for example. It showed us that working in team was (and is) really interesting if we want to become better developers.

Applying all these new methodologies and technologies has not always been easy. We spent a lot of time on the configuration of *Travis CI* for example because configuring it to execute *End-to-End testing* with *Protractor* was not simple. We also encountered problems with *mutations testing*.

[1] We refer here to the feeling that an experienced person have when taking a glance at some code.

If we had to do this master thesis again, we would mostly do it in the same way but we would change few things. We now have a better understanding of the methodologies we used. So we know when to use them or adapt them and when they are not necessary.

Test-Driven Development is not always useful. It is interesting when the developed application is critical or simply when you have enough time to afford it. But for simplest tasks, it is not mandatory and we do not always have to reach a high percentage of test coverage to have confidence into our code. For some projects, like our mobile application, only specific parts of it needed to be tested, not the application in a whole.

We learned that there are many different testing techniques. Each technique must be used when it is appropriate. So for the frontend that only manipulates data, it is more interesting to use *End-to-End testing* because the views and the features must match with the needs of users (or/and clients). But for the backend, that has many classes and performs more complex tasks, it is important to have lots of unit tests and integration tests to be sure that each individual method is working as expected and that components work fine together.

We realized during this master thesis that the more important a project is, the more important time is. A project is above all a mater of resources management. If there is no more time to work on a project, using a lot of methodologies will not change this fact. Finally we must evaluate if a methodology is useful and does not waste our time, we need to consider its pros and cons. This is why we produced a table to indicate when to use the different methodologies used during this master thesis in the appendix D.

Chapter 10

Glossary

Acronyms

API Application Programming Interface

CA Certificate Authority

CD Continuous Delivery

CI Continuous Integration

CLI Command Line Interface

GPA Grade Point Average

LCM Linear time Closed item set Miner

MVC Model View Controller

OS Operating System

OTP One-Time Password

PaaS Platform as a Service

REST REpresentational State Transfer

SMTP Simple Mail Transfer Protocol

TDD Test Driven Development

VCS Version Control System

VPS Virtual Private Server

XP eXtreme Programming

Definitions

Factor is a characteristic (pain, intensity, ...) that is present when a symptom appears.

Occurrence is the appearance of a symptom at a particular time and place (GPS Coordinate).

Patient is a person who is potentially ill and has symptoms.

Report is all data concerning the appearance of symptoms for a patient that is transmitted to his doctor.

RESTful API or simply REST API is an Application Programming Interface that uses HTTP requests to access and modify data. It follows the REST architecture style that defines some constraints (like *Stateless* and *Client-Server*) and properties (like *Scalability* and *Simplicity*): `https://en.wikipedia.org/wiki/Representational_state_transfer`

Symptom is a departure from normal function or feeling which is noticed by a patient.

User denotes a patient if he uses the mobile application or a doctor if he used the doctor application.

List of Figures

Bibliography

[1] Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[2] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2Nd Edition)*. Addison-Wesley Professional, 2004.

[3] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for agile software development, 2001.

[4] Mike Bostock. D3.js Data-Driven Documents. `https://d3js.org/`, consulted 03-04-2017.

[5] API dock. ActiveSupport::SecureRandom. `https://apidock.com/rails/ActiveSupport/SecureRandom`, consulted 27-03-2017.

[6] Free Software Foundation. The GNU Operating system and the Free Software Movement. `https://www.gnu.org`, consulted 09-12-2016.

[7] Anthony Guida. MiTM Attacks & SSL Pinning - Ionic Security's Blog. `https://www.ionic.com/blog/mitm-attacks-ssl-pinning-what-is-it-and-why-you-should-care`, consulted 17-04-2017.

[8] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader)*. Pearson Education, 2010.

[9] Troy Hunt. Your API versioning is wrong, which is why I decided to do it 3 different wrong ways. `https://www.troyhunt.com/your-api-versioning-is-wrong-which-is/`, consulted 20-03-2017.

[10] Investopedia. Learning Curve. `http://www.investopedia.com/terms/l/learning-curve.asp`, consulted 23-04-2017.

[11] R. Jeffries and G. Melnik. Tdd - the art of fearless programming. *IEEE Software*, 2007.

[12] Nicklas Ramhöj. Working with time zones in Ruby on Rails. `https://www.varvet.com/blog/working-with-time-zones-in-ruby-on-rails/#parsing`, consulted 03-04-2017.

[13] Ken Schwaber and Jeff Sutherland. The scrum guide. 2014.

[14] Takeaki Uno, Tatsuya Asai, Yuzo Uchida, and Hiroki Arimura. LCM: An Efficient Algorithm for Enumerating Frequent Closed Item Sets. In *FIMI*, volume 90. Citeseer, 2003.

Appendices

Appendix A

Metrics

In this appendix, we indicate a quick summary of the lines of code, lines of test code, number of classes, number of methods,... For the backend we used the command *rake stats*. For the other projects we used *cloc*[1]

A.1 Backend

Name	**Lines**	**LOC**	**Classes**	**Methods**
Controllers	170	148	9	18
Helpers	4	4	0	0
Models	130	109	6	4
Mailers	12	12	2	1
Javascripts	16	0	0	0
Libraries	0	0	0	0
Controller specs	1468	1154	0	1
Helper_factory specs	1582	1223	0	6
Mailer specs	35	27	1	0
Model specs	312	250	0	0
View specs	82	68	0	0
Worker specs	457	384	0	0
Total	4268	3379	28	30

Table A.1: Statistics for the backend got with *rake stats*

[1] `http://cloc.sourceforge.net/`

A.2 Frontend for doctors

Language	Files	Lines	Comment
TypeScript	42	1552	184
JavaScript	2	614	499
HTML	8	149	3
JSON	5	43	0
SASS	7	122	0
CSS	2	10	0
XML	1	2	0
SUM	67	2492	686

Table A.2: Code statistics for the frontend for doctors got with *cloc*

A.3 Frontend for patients

Language	Files	Lines	Comment
TypeScript	47	1933	166
JavaScript	2	522	509
HTML	11	284	13
JSON	7	177	0
SASS	8	154	48
SUM	75	3070	736

Table A.3: Code statistics for the frontend for patients got with *cloc*

Language	Files	Lines	Comment
TypeScript	21	1660	13

Table A.4: Test statistics for the frontend for patients got with *cloc*

A.4 Hubot

Language	Files	Lines	Comment
CoffeeScript	10	181	260

Table A.5: Code statistics for Hubot got with *cloc*

Language	Files	Lines	Comment
CoffeeScript	1	588	4

Table A.6: Test statistics for Hubot got with *cloc*

Appendix B

Burndown charts

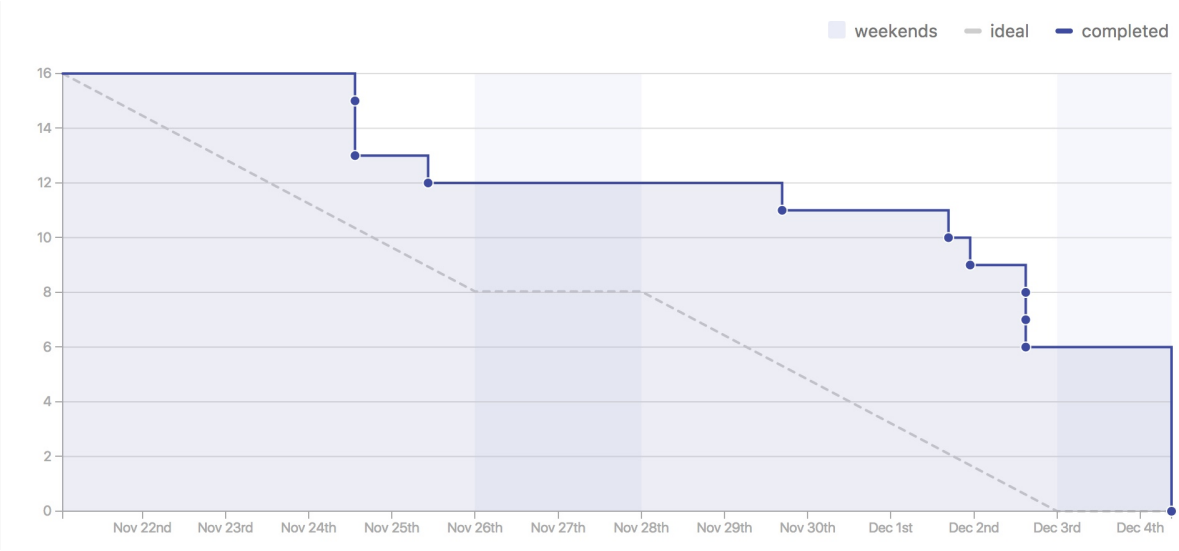


Figure B.1: Evolution of story points during sprint 1

Figure B.2: Evolution of story points during sprint 2

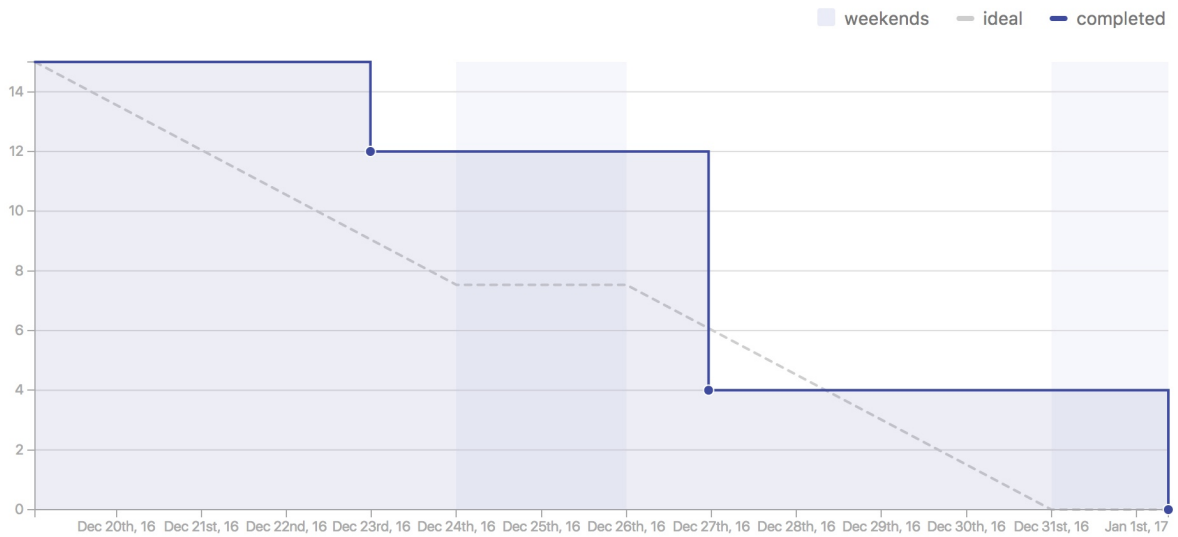


Figure B.3: Evolution of story points during sprint 3

Figure B.4: Evolution of story points during sprint blocus

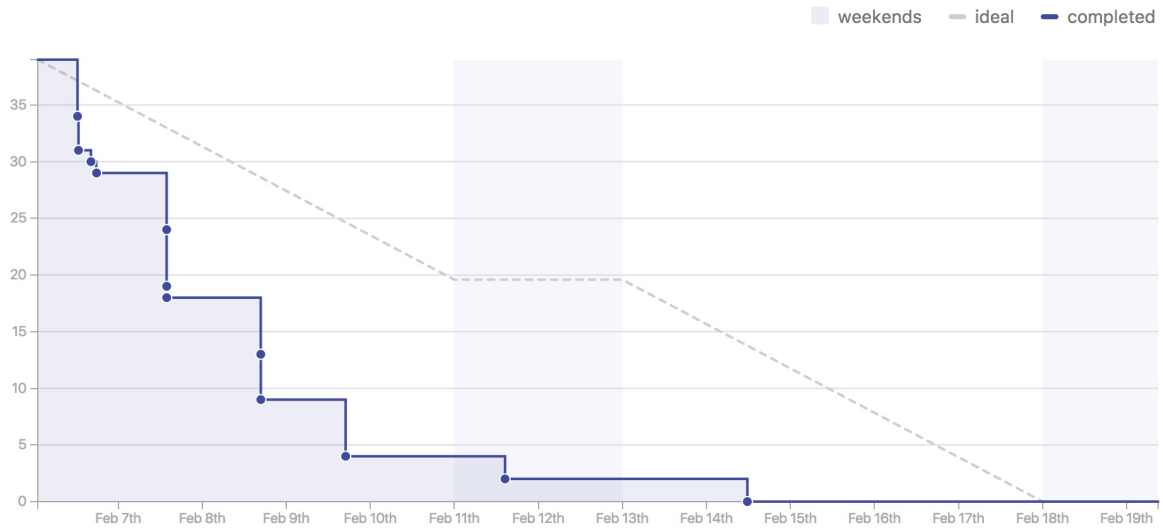


Figure B.5: Evolution of story points during sprint 4

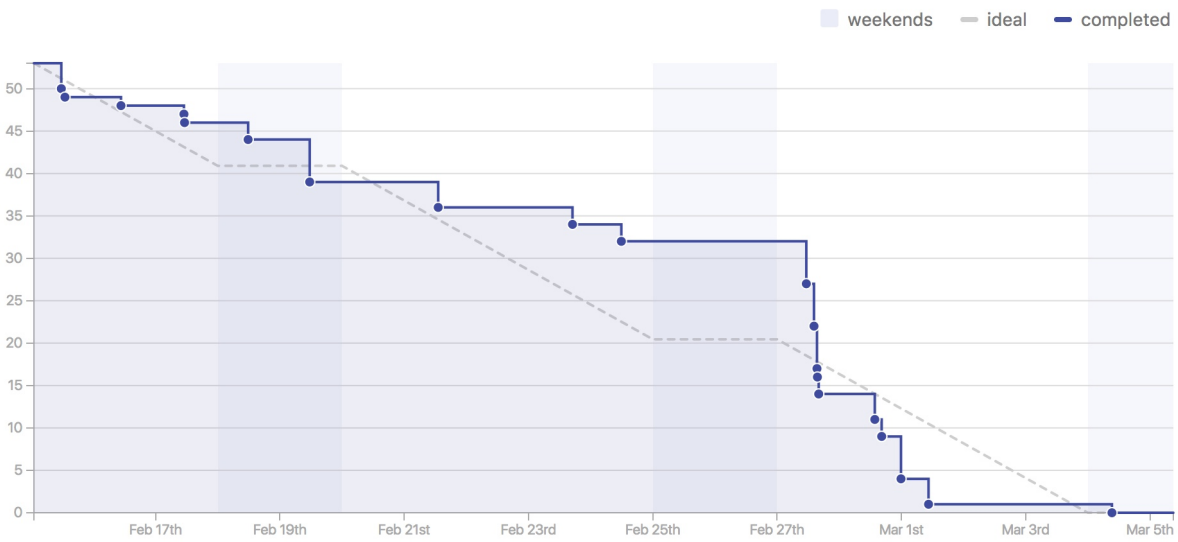


Figure B.6: Evolution of story points during sprint 5

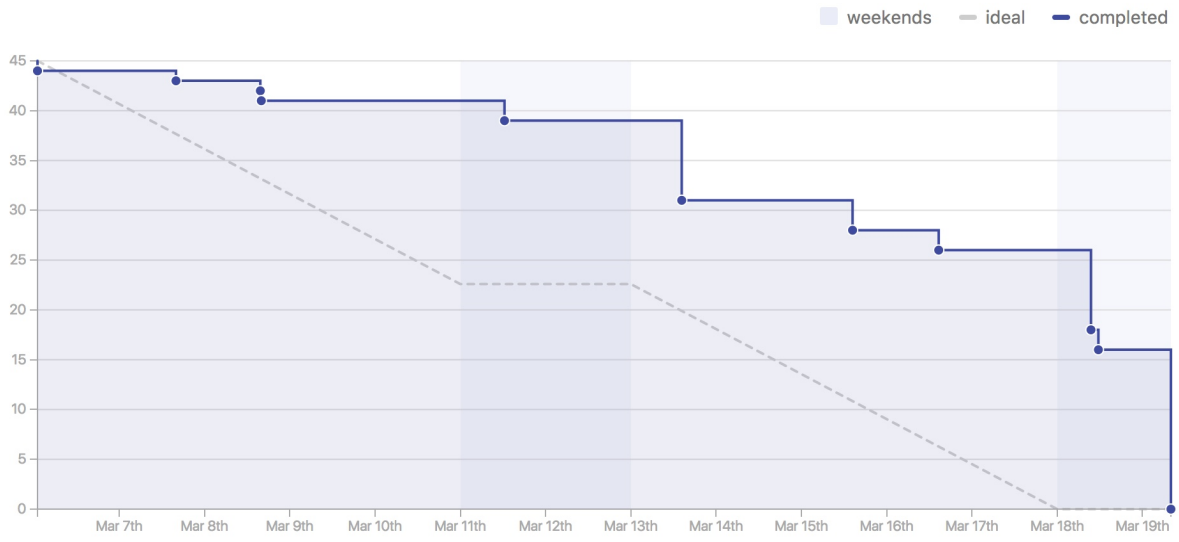


Figure B.7: Evolution of story points during sprint 6

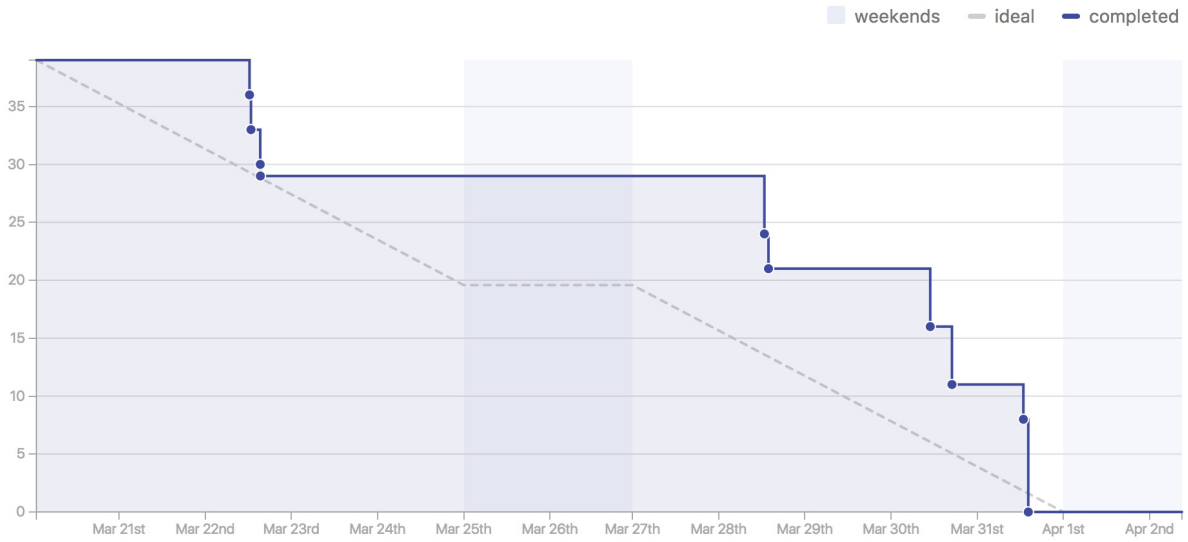


Figure B.8: Evolution of story points during sprint 7

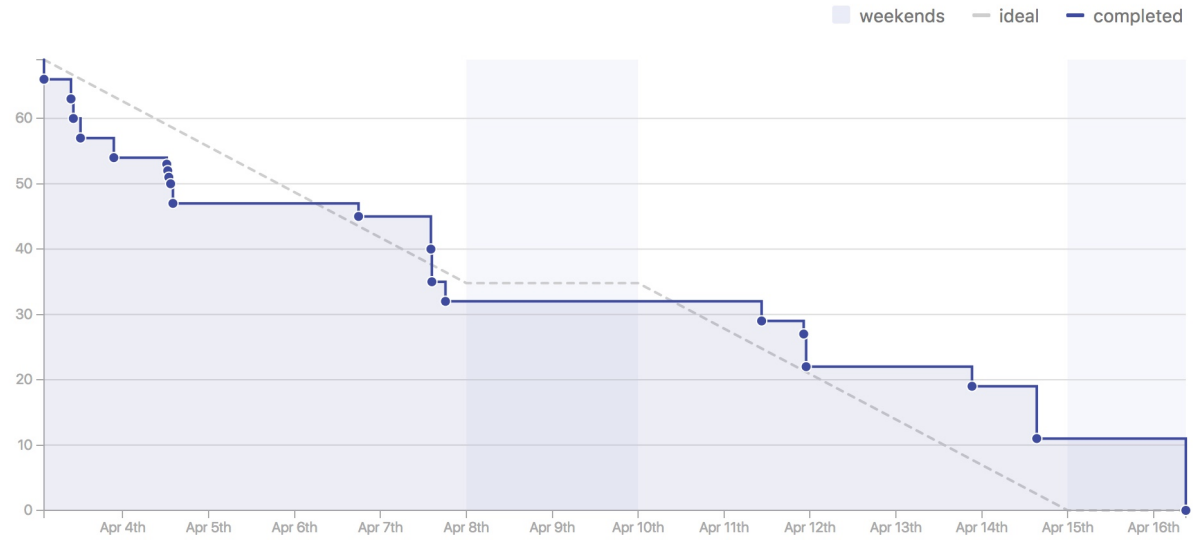


Figure B.9: Evolution of story points during sprint 8

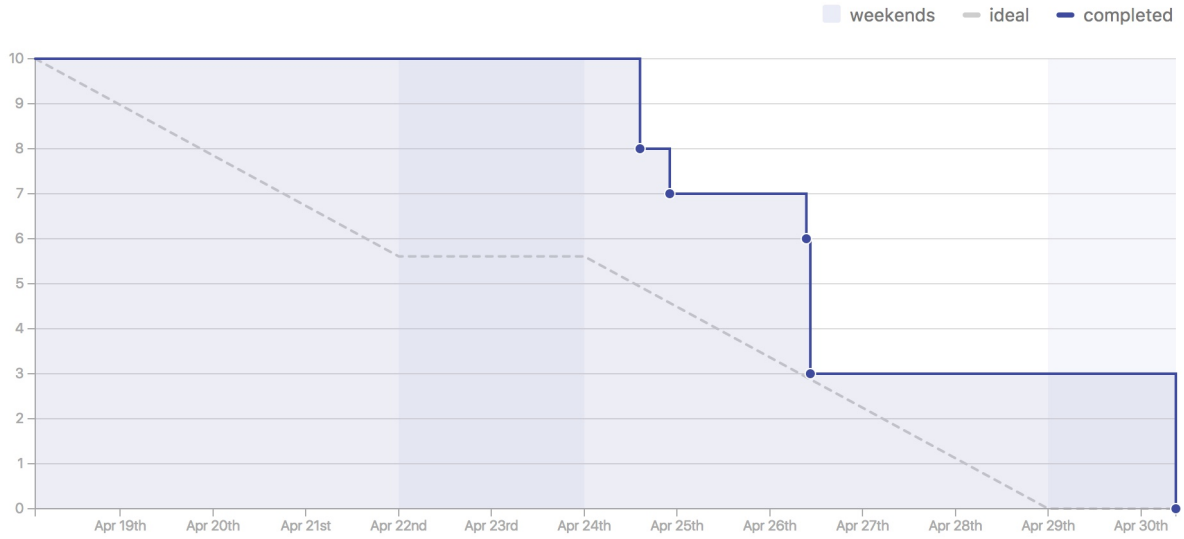


Figure B.10: Evolution of story points during sprint 9

Figure B.11: Evolution of story points during sprint 10

Appendix C

Database diagram

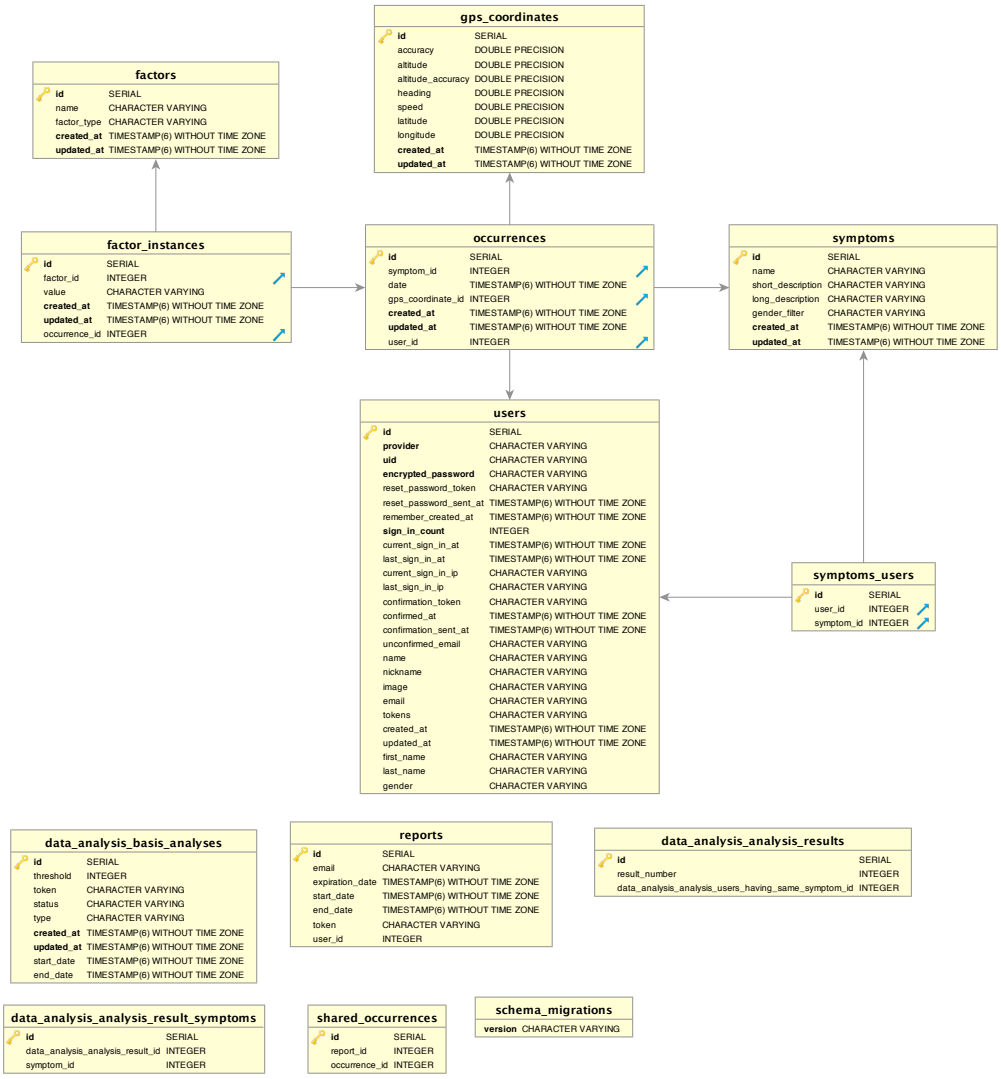


Figure C.1: Diagram of the backend database

Appendix D

Table of practices

In this appendix, we present a summary table of practices used during the realization of this master thesis. These practices are grouped by methodology[1] for easier access. In each table we indicate when the practice is useful and some remarks.

[1]The methodology in which we discovered the practice

From Extreme Programming		
Name	**When**	**Remarks**
Pair Programming	• A new technology is introduced • A new person is added to the project • You want to increase the code quality • You want a better diffusion of knowledge • You want a better transfer of skills • You are stuck on a problem • You want to reduce the coordination efforts	• Some people prefer working on their own • The environment must allow it, you need to talk and it is not possible everywhere
Energized work	• You want to be more effective • You want to have more time for yourself	
Ten-Minute Build	• You want to have feedback frequently	• Could take time to improve the build
Continuous Integration	• You want to merge often the code of the team	
Test-Driven Development	• You want to avoid bugs • You want to minimize your code • You want to have a refactored code	• Not everything has to be tested
Coding Standards	• You want a harmonious code • You do not want to be lost in the code of others	• Try to use existing standards

From Agile		
Name	**When**	**Remarks**
User stories	• You want to have an interaction with your team about the implementation, the features,... • You want to facilitate understanding between the clients and the developers • You want a structured work	• Not to be confused with Use Cases • More useful when the client is involved • Good practice: estimate the stories
Cycle Development, Iteration	• You want a fast feedback from the client • You want to be able to estimate your velocity • You want to be sure (or more confident) to match with the client's needs	• Often combined with user stories
Poker planning	• You want that each members can speak freely • You want to estimate features complexity • You want to better manage your time	• Linked to user stories
Code review	• You want to avoid bugs • You want a feedback about your code • You want to improve your skills	

From Continuous Delivery		
Name	**When**	**Remarks**
Automatic deployment	• You want that the last version of your application is directly available for users • You want quick feedback from customers	

From Testing Techniques		
Name	**When**	**Remarks**
Mutation testing	• You want to test the reliability of your tests	
Test coverage	• You want to know which part of you code is tested • You want to be confident about your code	• Test coverage can be 100% while your tests do not cover all the cases[2] • You must define the test coverage level that give you confidence into your code (20% can be enough)
Unit testing	• You want to prove that your code behaves as expected	• Not everything must be tested
Integration testing	• You want to test that different components work together	
End-to-End testing	• You want to test the features asked by the client • You want to test that your entire solution works from the point of view of a user	

[2]That is why you need mutation testing

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