"Open Source RFID Race Timing System"

Deconinck, Guillaume ; Grynczel, Wojciech

Abstract

This dissertation explains all the steps we went through to model and develop a race timing system for Game of Trails, a non-profit organization. Game of Trails organizes each year two races, which have the particularities of being muddy and having several obstacles scattered along the course. In fact, Game of Trails already had an arrangement with a company for a race timing system but it had a few drawbacks such as being expensive and giving the final results a few days after the race instead of the same day. Given those drawbacks, they wanted to change and have their own race timing system. One important requirement of this new system is the display of the results in real-time. More precisely, they wanted that anyone could see the results of the race, updated in real-time, on a website. To build this new race timing system, we first had to decide how the system would detect the runners at the finish line. Multiple technologies could be used for this and we present in this disserta...

Document type : *Mémoire (Thesis)*

UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

Open Source RFID Race Timing System

Dissertation presented by
Guillaume DECONINCK , Wojciech GRYNCZEL

for obtaining the Master's degree in
Computer Science

Supervisor(s)
Pierre SCHAUS

Reader(s)
Kim MENS, Ramin SADRE

Academic year 2016-2017

Acknowledgements

First, we would like to thank everyone who supported us throughout our thesis whether it is by advising us or simply by encouraging us. It is the sum of all those supports that made this thesis what it is now.

More specifically, we would like to thank *Game of Trails*. Without them, this thesis would not have been written and all of this would not have been possible. They believed in us and helped us from the very beginning to the very end. They were understanding that we were only students trying to do their best. In particular, we would like to thank *Damien Renier*, our main contact with Game of Trails, who was always kind and very responsive to answer any of our questions.

Secondly, we would like to thank our supervisor, *Professor Pierre Schaus*, for supporting and advising us during all this year. All of this would also not have been possible without him. He helped us many times when we were unsure of the direction to take.

Thirdly, we would like to thank all of our relatives that have always encouraged us and helped us, whether it is during difficult times or when everything worked as expected.

Finally, we would like to Bighorn Studio for allowing us to use their pictures in this dissertation.

Abstract

This dissertation explains all the steps we went through to model and develop a *race timing system* for *Game of Trails*, a non-profit organization.

Game of Trails organizes each year two races, which have the particularities of being muddy and having several obstacles scattered along the course. In fact, Game of Trails already had an arrangement with a company for a race timing system but it had a few drawbacks such as being expensive and giving the final results a few days after the race instead of the same day. Given those drawbacks, they wanted to change and have their own race timing system. One important requirement of this new system is the display of the results in real-time. More precisely, they wanted that anyone could see the results of the race, updated in real-time, on a website.

To build this new race timing system, we first had to decide how the system would detect the runners at the finish line. Multiple technologies could be used for this and we present in this dissertation what we believe are the most interesting ones. To name them, those are *QR Code*, *GPS*, *NFC* and *RFID*. After comparing them, we decided to use *UHF RFID* (Ultra High Frequency - Radio Frequency IDentification). It consists of radio waves of frequency 860Mhz to 865Mhz and allows to detect tags at a distance of maximum 12 meters with the help of antennas. Moreover, the tags are waterproof. However, it is quite expensive.

Afterwards, we did the requirements and modeled the race timing system. We show what are the different use cases and what are the expected data models, but also how the real-time feature could be modeled with the help of sequence diagrams. We also made mock-ups of graphical user interfaces to give a better idea of what the website would look like. We defined what should be the procedure for creating and managing a race in our system (non-exhaustively: create a race, import the runners, assign the RFID tags, etc.). A last important decision during the requirements was to decide what are the different parts of the system. In our case, there should be a *database*, a *back-end*, a *front-end* and *checkpoints*.

The main step of this thesis is the development which first consists of choosing the technologies. At the beginning, we decided for the database to use *Firebase* by Google for our database but we changed at the middle of the development to *MongoDB* because of its more advanced queries. Regarding the back-end, *Node.js* and more precisely the *Feathers* framework were chosen. It allowed us to rapidly build a working back-end with services, authentication and out-of-the-box real-time with Websockets. The front-end was developed with *Angular* version 2. It is a well known front-end framework that is lightweight and fast. Finally, what we call the "checkpoints" are Raspberry PIs with RFID antennas. Their role is detect the runners, store their times and send these to the back-end.

In the last chapters, we describe the actual race. We explain how we installed the system at the 3 checkpoints during both days of the race. We also explain a problem that occurred the first day which made useless our second checkpoint as it detected almost no runner. Fortunately, everything else worked as expected and Game of Trails was happy of our job.

Contents

Glossary

- **API** (Application Programming Interface) is a collection of procedures, protocols, and tools for building applications. The API defines how the application should interact.
- **GPS** (Global Positioning System) is a satellite-based navigation system, originally created for military purpose, composed of at least 24 satellites orbiting 20,200 km above sea level.
- **HTTP** (HyperText Transfer Protocol) is a web application protocol used to send queries and receive response in the client-server model.
- **HTTPS** (HyperText Transfer Protocol Secure) is a web application protocol consisting of an HTTP connection inside an encrypted connection (TLS).
- **JSON** (JavaScript Object Notation) is a lightweight format for storing and exchanging data. It is easy for the humans to read and write while still being easy for machines to parse and generate.
- **NFC** (Near Field Communication) is a radio technology based on the specification of RFID technology.
- **NPM** (Node Package Manager) is a package and dependency manager used with Node.js.
- **NPO** (Non Profit Organization) is an organization that exists without the goal of making any profit from its actions or events.
- **NTP** (Network Time Protocol) is a networking protocol used to synchronize clocks.
- **QR Code** (Quick Response Code) is a two-dimensional barcode, invented in Japan in 1994 by Denso-Wave.
- **RFID** (Radio-Frequency Identification) is a generic term used to describe a technology that allows automatic identification using radio waves.
- **RFID (LF)** (Radio-Frequency Identification Low Frequency) is a specific subset of the RFID, it operate at 125-134 KHz band with read less than 50cm.
- **RFID (HF)** (Radio-Frequency Identification High-Frequency) is a specific subset of the RFID, it operate at 13.56 MHz band with read ranges between 10 cm and 1.5 m.
- **RFID (UHF)** (Radio-Frequency Identification Ultra-high frequency) is a specific subset of the RFID, it use the 860 MHz to 960 MHz band, with read up to 10 meters.
- **RTC** (Real Time Clock) is a computer clock that keeps track of the current time. It usually has its own power source that keeps it on even when the computer is off.
- **SDK** (Software Development Kit) is a set of development tools given to help the creation of applications for a specific hardware, software package, platform, etc.

- **TLS** (Transport Layer Security) is a cryptographic protocol that provides communications security over a computer network. It is used by HTTPS.
- **WebSockets** is a technology that allows persistent two-way TCP connection between the client and the server.

Chapter 1

Introduction

For our master thesis, we wanted a subject that would consist in developing a concrete application. By concrete application, we mean an application that, once finished, would be directly used by users. It is clearly one of the main reasons why we decided to take this thesis. For us, the fact that it is a real client that proposed this thesis had a huge influence. Moreover, working for an non-profit organization that organizes mud races was an unique opportunity that directly drew our attention.

As the title of our thesis suggests, we had to develop a race timing system. But what is a *race timing system* ? It may seem obvious to some of you, readers, but let's precise it a little. A race timing system is a system that allows the organizers to have the times of all the runners at least at the end of a race. It allows them to know with certainty which runner is the first, the second, etc. Because races are usually competitive, it is often a must-have. Knowing the time of each runner allows the organizers to give rewards to the best ones and the runners to immediately evaluate their performance. Outside competitive events, such system can also be used by runners to get their time during their training.

We defined here the simplest race timing system but a lot of aspects can change from one system to another. For example, one important choice is the availability of the times: are they publicly available to everyone or are they just for the organizers ? Another main choice to make is how the system detects that the runners have finished the race and thus passed the finish line. Indeed, this is an obvious decision to make but we will show you that a lot of different solutions exist.

The structure of this dissertation is divided as follows: first, we introduce the client, Game of Trails, and its uncommon races. This will give a little more context for this subject and will help to understand some of the requirements linked to the mud races. We also explain what is the old solution that was already in place and what were the main demands of Game of Trails. The goal is to give you a broad idea of what was already there and what was asked. Afterwards, we look into the different existing solutions that we proposed to Game of Trails. Each one of them will be presented with their advantages and drawbacks. Once the chosen solution explained, the next step is to give the requirements. Their goal is to give a much clearer and defined idea of what will be possible to do with the system, how it will behave, how it will look like, and what are the different data. After the requirements comes logically the development, we show how the system has been structured, what are the hardware that we bought, what technologies have been used, etc. Once the system was developed and thoroughly tested, it was time for the production phase, which in our case was synonym with the actual race. We explain how the system was installed and what are the results of this important step in our thesis. The last chapter before the conclusion talks more about the future: what are the possible improvements for this system? Finally, we will end this dissertation with a conclusion.

Chapter 2

The client

Game of Trails is a non-profit organization (NPO) that organizes mud races in Belgium since 2014. Those mud races currently occur two times a year, and go each time through an entire weekend. As much as 3000 runners come at each race.

However, the story of Game of Trails started earlier, in 2013. It all began first as a student project for Télévie. They were 20 students and had only 3 weeks to organize a race. Even with the time constraint, the race was a success and saw as much as 200 runners for their first edition. The next year, 6 of the 20 students decided to continue the adventure and created what is known now as Game of Trails.

2.1 The races

The first race, which exists since the creation of the organization, is the race of *Sart Tilman*. It usually takes place during the last weekend of the Easter holidays. The second race, meanwhile, is quite recent. It has only taken place once in the city of *Flémalle* in 2016 but the second edition is already scheduled for September 2017.

The runners are divided into two categories which are the *Fun* runners and *Elite* runners. As the names suggest, the *Fun* category is for the runners that come to the race for having fun without any competitive intent. On the other hand, the *Elite* category exists for the runners that are more competitive and are willing to pay to have their time tracked. The runners choose their category themselves when they buy their places (20€ Fun, 25€ Elite).

Regarding the races themselves, they are of variable length. Indeed, the *Fun* runners have the choice during the race to take the longest path or the shorter one. The *Elite* runners, on the other hand, have to take the longest path. Indeed, the *Elite* runners participate for the competition and therefore they all have to take the same path in order for the times to make sense.

One aspect we didn't talk about yet is the fact that the races are said to be *mud* races. First, it means that the course of the race goes through roads, fields and forests. Secondly, several obstacles are scattered along the course. Those obstacles range from escalating a high wood barrier to having to crawl in a tunnel filled with mud. As you can see in the figure 2.1, you should not wear brand new clothes during those races !

Figure 2.1: One of the muddy obstacles

Finally, note that all the runners are divided into waves. Usually, there is only one wave for the *Elite* runners. On the other hand, the *Fun* runners are divided into several waves. The reason behind this division is that it would be too complicated to let all the runners start at the same time (imagine 1500 runners starting at the same time !). The waves are launched with an arbitrary interval, which is usually 15 min.

2.2 The old solution

Until our master thesis began, Game of Trails had an arrangement with a company to do the timing of the *Elite* runners. It worked well but it had a few drawbacks and it missed some features that could be seen as mandatory nowadays. The deal was that the company managed everything for the race timing system by themselves. They brought their own hardware and they installed the system on their own during both days of the races. After the races, they sent the results to Game of Trails.

A few points have to be noted about this solution. First, they used UHF RFID to do the detection. This technology will be explained in a later chapter, but note that it is also what we decided to use. Secondly, it was expensive for Game of Trails. Because it is a non-profit organization, the budget is always quite tight and money is saved wherever it is possible. Thirdly, the runners were only detected at the finish line (which is obviously the minimum for a race timing system). Having the detection only at the finish line does not allow to identify possible cheaters (i.e. runners that take shortcuts). Indeed, this would need some kind of "checkpoints" along the course. Lastly, as already mentioned, the results were sent to Game of Trails after the race. However, it was after a few days and not directly at the end of the race. That meant for Game of Trails that it was not possible to organize a small event at the end of each day for giving rewards to the best runners.

2.3 The request

Directly related to the drawbacks and issues of the old solution explained in the previous section, Game of Trails had four main goals that we had to fulfill.

First, one of the main features requested was to get the results at least on the same day as the race. This was the minimal requirement. However, they wanted if possible to go further and have the results displayed and updated in real-time on a website. The primary goal is clearly to be able to organize a small event at the end of each day to congratulate the best runners and give rewards to them. Moreover, having the results in real-time on their website is a plus not negligible. It's interesting and fun for the friends of the *Elite* runners to be able to see when they arrive at the finish line. Finally, note that, with a website, the runners can see their time whenever they want and compare their performance with others.

The second objective is that they wanted to stop depending on another company for the timing system. More simply, they wanted to become more independent. Depending on another company involves a lot of coordination with it such as ensuring its availability during the races. Moreover, Game of Trails had to transmit in advance the names of the *Elite* runners in order for it to assign the RFID tags.

The third goal is related to the lack of detection of possible cheaters. Game of Trails wanted to add some kind of "checkpoints" along the course of the race. Such checkpoints could allow the detection of cheaters among the *Elite* runners by forcing the runners to be detected at intermediary checkpoints. A runner that has not been detected at all the checkpoints and has a time really low compared to others can logically be presumed guilty of cheating. In addition to this, having checkpoints gives more data for the runners. They can know in which part of the race they were the fastest and where they had the most trouble.

Lastly, they wanted to reduce the costs associated to the race timing system in the long run. This reduction is of course directly profitable to them, but it is also profitable to the runners. Indeed, as it is a non-profit organization, it will naturally reduce the price of the race participation for the runners. Moreover, the perspective of paying less while having more features (real-time results, etc.) is obviously appealing. Note that the investment has only to be made once for the minimum hardware needed. Afterwards, the future costs will only be linked to the maintenance or the expansion of the system. For example, it could be possible in the long run to allow everyone, *Fun* and *Elite*, to have their time tracked (and thus merge the *Fun* and *Elite* categories).

To summarize the goals of Game of Trails, we can say that they wanted to have their own race timing system which would allow them to reduce the costs in the long run while adding more features than what they already had (such as a website updated in real-time). They told us at the same time what was their maximum budget for the race timing system, which was of 1500€. We had to keep in mind this budget during the whole thesis as it is an important limit that ideally should not be exceeded. You will see later that we first developed a prototype to reduce the risks for Game of Trails and for us.

Chapter 3

Proposed solutions

To satisfy the customer's needs, we have explored several practical solutions. Our main objective was to design a reliable system that could be easily used, by non technical people, for several years and at the same time with a reasonable price. One of the main challenges was to create a system that could resist the extreme conditions of the race, everything must work perfectly even in contact with water or mud. Moreover, we have set a goal to design a system that will provide the results of the runners in real time.

3.1 QR Code

What is QR Code? [1]

The QR Code (Quick Response Code) is a two-dimensional barcode, invented in Japan in 1994 by Denso-Wave [2]. At the beginning, these codes were used mainly in professional environments like factories, warehouses, retail sales, etc. Nowadays, QR codes are becoming more popular among the general public, all thanks to mobile devices such as smartphones and tablets.

Compared to standard barcodes, QR codes have several advantages, such as:

- QR Code can store up to several hundred times more information.
- QR Code has an error correction capacity which allows decoding from 7% up to 30% of damaged data [3].
- QR Code can be read from any direction in 360 degree.
- QR Code can be easily scanned with a camera (e.g. mobile device).

Figure 3.1: An example of QR Code

The idea

The idea was to create a system easy to use, composed of 3 elements (see figure 3.2):

1. A smartphone with a special application that uses the integrated camera to scan the runners. Thanks to the 4G connectivity of the smartphone, the scanned runners can be sent directly to the server.
2. An A4 sheet of paper with an unique QR Code.
3. A server that retrieves scanned runners and processes the results in real time.

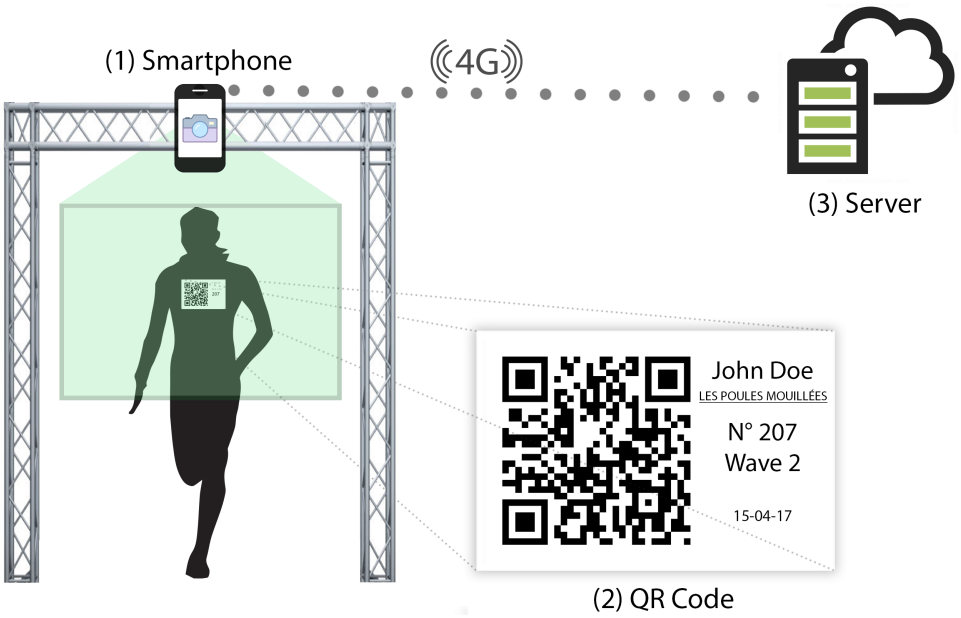


Figure 3.2: Diagram of system using QR Code

3.2 GPS

What is a GPS [4]

The GPS (Global Positioning System) is a satellite-based navigation system, originally created for military purpose, composed of at least 24 satellites orbiting 20,200 km above sea level.

The idea

This second idea was to develop an application for smartphones that uses integrated GPS to track runners in real-time. In addition to the timing of the runners, this solution makes it possible to have several interesting information such as the exact path or the speed at various stages of the race.

3.3 NFC

What is NFC [5]

NFC is a radio technology based on the specifications of the RFID technology. Unlike the RFID technology, a NFC device may act as both a tag and a reader. NFC uses the same frequency as RFID HF, 13,56 MHz, and allows wireless data exchange at a maximum distance of 20 centimeters.

The idea

A system using NFC is composed of 3 main components (see figure 3.3).

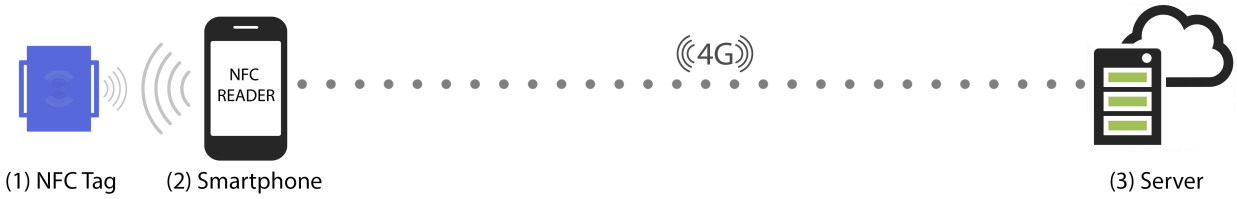


Figure 3.3: Diagram of system using NFC technology

Each participant receives an NFC wristband, containing an unique identifier. We have found several models that are water-proof with a cost varying between 0.70 € and 1.00 €. When a runner crosses a checkpoint, he must scan his tag on an NFC antenna. To reduce the cost of this solution to the maximum, we thought to use our smartphones as NFC antennas. Indeed, the majority of modern smartphones have an integrated NFC antenna which is usually used to make payments. At the same time, we can use the Internet connection of the smartphone to send the results directly to the server as soon as a runner is scanned.

3.4 RFID

What is RFID [6] [7]

RFID (Radio-Frequency Identification) is a generic term used to describe a technology that allows automatic identification of objects using radio waves. RFID is divided into several specialized subsets:

Band	LF (Low Frequency)	HF (High Frequency)	UHF (Ultra-High Frequency)
Frequencies	125-134 kHz	13.56 MHz	865 - 956 MHz
Read range	Less than 10cm	Up to 1m	Up to 12m
Multiple reads capability	Usually only single reads	Good	Excellent multiple reads capability
Tag Cost	Relatively expensive	Varies depending on type of tag	Very low cost (at high volumes)
Typical use	Animal ID Access control	Ticketing Payment Smart Labels	Inventory management Tracking Logisitcs

Table 3.1: Comparison of RFID Technologies [8] [9]

Between these 3 types of RFID, the UHF RFID technology is the most interesting for us because it has the best parameters for the case of a race timing system. A large read range combined with the excellent multiple reads capability allow to scan efficiently multiple runners at the same time. Note that the RFID UHF is divided into 2 categories, active and passive [10]. The active version is very expensive because the tags need their own power source in order to continuously broadcast the signal. The advantage is that this kind of tags can be scanned up to 100 meters. The passive version of RFID UHF tag is much cheaper because it has no internal power source. It uses the power of the radio waves of the antenna to emit data. The reading distance is much lower than in the active version but can still reach up to 12 meters.

Between the active and the passive versions of the tag, we decided to use the passive one because the tags are much cheaper and we do not need such long range of read.

The idea

This solution is composed of 5 elements (see figure 3.4). Each participant receives an RFID UHF wristband (a "tag"), containing an unique identifier. We found a reusable and waterproof sport wristband costing about ~1€ per unit.

At each checkpoint, we need to install an UHF RFID antenna that will scan the runners. In order to reduce the risk that some runners are not scanned, we think that 2 antennas at each side of the checkpoint are a must-have. The chosen hardware must be water resistant to ensure proper operation even when it's raining. We found an antenna that meets all requirements for about ~120€

In addition, we need some kind of computer that processes all the data that have been scanned with the antenna. For this task we chose the Raspberry Pi 3B because we wanted to have a reliable machine with Linux and its rich developer community. In addition to this, thanks to integrated Wi-Fi, we can easily connect to the internet (in the worst case, using a smartphone as a mobile access point) and send the results directly to the server.

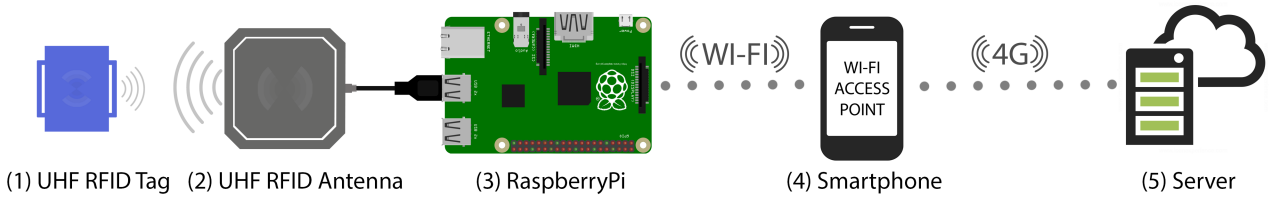


Figure 3.4: Diagram of system using UHF RFID technology

3.5 The decision

After some thorough researches, we decided to use the RFID technology. The choice was not easy, each of the four technologies has its own advantages and drawbacks. We have taken into account the different criteria like the cost, range, the ability to read multiple tags simultaneously, re-usability, etc. (see table 3.2).

The solution using the QR Codes was very interesting, the low cost per runner was the main advantage. Indeed, the cost of printing sheet of papers plus punched pockets to protect against mud is very low. The purchase of a smartphone can also be avoided if we use the devices of the volunteers. But even if this solution is very cheap, it has 3 serious problems (see figure 3.5):

1. First, despite the error correction system, the QR Code can become unreadable if it is covered with a lot of mud (which can happen easily in the case of Game of Trails).

2. Secondly, if several runners cross the checkpoint at the same time, some will not be scanned if their QR Code is hidden by another runner.

3. Thirdly, the sheet of paper/the punched pocket containing the QR Code can be easily detached or even lost during this kind of races with multiple obstacles to overcome.

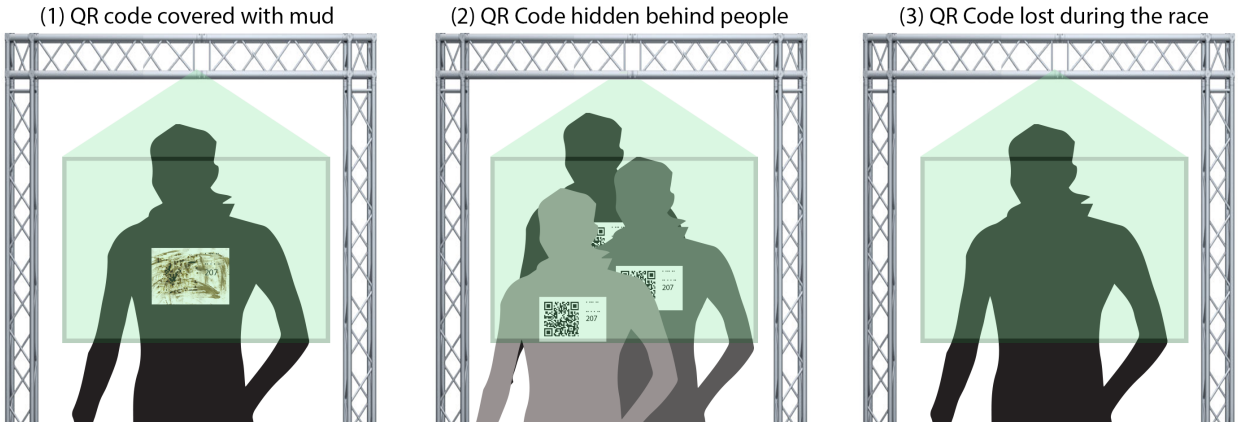


Figure 3.5: The main problems of QR Code based system

Therefore, we have decided to give up this technology because the use of the QR codes did not allow to have a reliable results. The risk of not scanning a runner was too big. On top of that, wearing a sheet of paper and taking care of it throughout the race is not very convenient for the runners.

Next, we analyzed the solution using the GPS, which on paper was very attractive. Thanks to an Internet connection, we could have for each runner the traveled path, the current position, the speed, and many other interesting information in real time. We quickly abandoned this technology because it is not practical for runners to have their devices with them during the race. All smartphones are not water (mud) resistant and they can easily break during the race. Note that GPS tags exist but they are far too expensive for the allocated budget.

The solution using NFC has almost met all our needs. It is cheap, we only need NFC tags and some smartphones equipped with a NFC reader. We found models of NFC tags that are water resistant and which can be comfortably worn on the wrist or ankle. Unfortunately, the NFC have one serious drawback, the range of an NFC antenna is far too short for being use in a serious timing system. We can not accept a technology that forces people to totally stop to perform a scan. In addition to that, with such a short range it is impossible to scan several runners at the same time. Some people may have to wait before performing a scan. Therefore, we had to drop this solution because the scanning distance was too short and it did not allow to scan several runners at the same time.

Finally, the RFID technology is perfect for a timing system. Thanks to its anti-collision system, this technology allows to scan tens of tags simultaneously and instantly. The long range of the antenna allows to scan the runners even when they move at a high speed (without stopping). Unfortunately this solution is not perfect, it also has 2 big drawbacks. Firstly, because of the large amount of hardware needed this solution is much more expensive than solutions using QR Codes or NFC. It requires the purchase of tags, antennas and Raspberry PIs. Secondly, an external power supply is required to provide power to each antenna and Raspberry PI.

Undoubtedly, the RFID technology is the most complex solution, also, like in the other solutions, a smartphone with an Internet connection is needed to provide the results in real time.

Despite these disadvantages, we decided to use the RFID because this technology is the most reliable and met all our requirements.

You can find in the table 3.2 a summary of the comparison of the different solutions.

Solution	**QR Code**	**GPS**	**NFC**	**RFID UHF**
Cost	Low	/	Medium	High
Hardware needed	Smartphone Server	Runner's smartphones Server	Smartphone NFC Tag Server	Antenna Tag Raspberry Pi Smartphone Server
Range of lecture	High	High	Low	High
Multiple read	Possible but very risky	Possible	Impossible	Possible
Confort for runner	Medium	Low	High	High
Assignation of runners	Easy	Easy	Hard	Hard
Reusable each year	No	/	Yes	Yes
Water/mud resistance	Medium	Bad	Good	Good

Table 3.2: A comparison of the proposed solutions [8]

Chapter 4

Requirements

We have explained what were the main features requested by Game of Trails and what kind of technology we chose to detect the runners. Now, it is time to express the whole system more formally. In this section, we first explain the procedure to organize a race according to the information we were provided. Then, we specify in more technical terms what are the different parts of the system. Next, we talk about the use cases and the different actions the system should allow. It is a must-have in any analysis. Afterwards, we discuss about the data models and what types of data are needed. As the real-time is one of the main features, we explain a little how it can be modeled with the help of a sequence diagram. The goal is to have a better idea how this particular feature can be implemented and what will be the interactions between the different parts of the system. To conclude this chapter, we show what are the expected graphical user interfaces for the visitors and the administrators.

4.1 Procedure

The procedure of creating and managing a race in this new race timing system involves a few steps. Those steps have to be done by the administrators before the actual race takes place.

For the first step, the administrators have to create a new race in the system. This is a mandatory step because the system needs to know during which days the race occurs in order to show the results' for each day to the visitors. If a race already exists in the system when adding a new one, it is deleted with all its related data. It might sound radical, but Game of Trails clearly stated that they don't need any history for the results and therefore preferred to reduce the complexity of the system.

The second step consists in importing the runners into the system with an Excel file. This file would have to be in a specific format for it to be parsed. After this step, the database should have been populated with all the runners and all the waves. Indeed, the information about the runners in the Excel file also include, for each runner, his wave type (Fun or Elite) and his wave number (1,2,...). We can therefore deduce the different waves (Fun 1, Elite 1, ...).

The third step is optional, the administrators can register or remove tags in the system if they feel the need to do so (for example if there are not enough tags, or if some tags were lost during the previous race). The tags we are talking about are the *RFID tags*.

For the fourth step, the administrators have to assign the tags to the runners in order for the system to know which tag belongs to which runner. Ideally, this should be an operation easy to make. In other words, it means that the administrators should be able to assign the tags to all the runners with a single click of a button. It would be too long and annoying to assign the tags

one by one to the runners. After assigning the tags, the administrators can export and print a PDF file of the timed waves. This file should be sorted by day, by wave and by team name and should show the name of the runners and their tags.

The last step is also of huge importance, the administrators have to "launch" the waves. At the beginning of the actual race, as explained in the presentation of Game of Trails, waves of runners are launched at a regular interval. The system needs a way to know that the wave W was launched at time T in order to be able to compute the times of each runner afterwards. This should be easily done like with a click of a button, one button for each wave.

4.2 The structure

We can divide the race timing system into four main parts: the **database**, the **back-end**, the **front-end** and the **checkpoints**.

Database

The database is where we store all the data, such as the runners and their times. It is the foundation of the system. It has to be accessible from anywhere on Earth because all of the other parts rely on it and therefore it needs to be hosted on a web server. However, mainly for security reasons, the database is usually not directly accessible by the outside world. The back-end is there to solve that problem.

Back-end

The back-end is usually a web server that serves the front-end (the website) and presents an API. It defines what are the different possible actions for the users and ensures that the actions are correctly secured depending on the user's role. Therefore, it often runs some verifications before executing each request. This API also acts as a gate to the database by securing the access to it. Note that the back-end can do some computations on the data coming from a request before saving, querying, updating or deleting it in the database. It is also possible in the other way, which means before returning some data to the user that initiated the request. A back-end, and more precisely the web server hosting it, is often associated to a domain name.

Front-end

The front-end is equivalent to the website. It is where the visitors can see the results and where the administrators can interact and manage the system. It is usually here that the requests for the actions proposed by the API of the back-end are sent. It is hosted on the back-end.

Checkpoints

Checkpoints consist of devices that are used to detect the runners with an antenna and to store the data such as the times of the runners. When an access to the Internet is available, those devices should also send the times directly to the back-end (without relying on the front-end (as there are no browser). For the real-time feature, they need to be connected at all time to the Internet as precised in the chapter about the possible solutions.

4.3 Use cases

Our use cases are quite simple, they involve a lot of CRUD (Create, Read, Update, Delete) operations which are made nearly exclusively by the administrators. Anyone who comes on the website is called a "visitor". A visitor who has authenticated himself is called an "administrator".

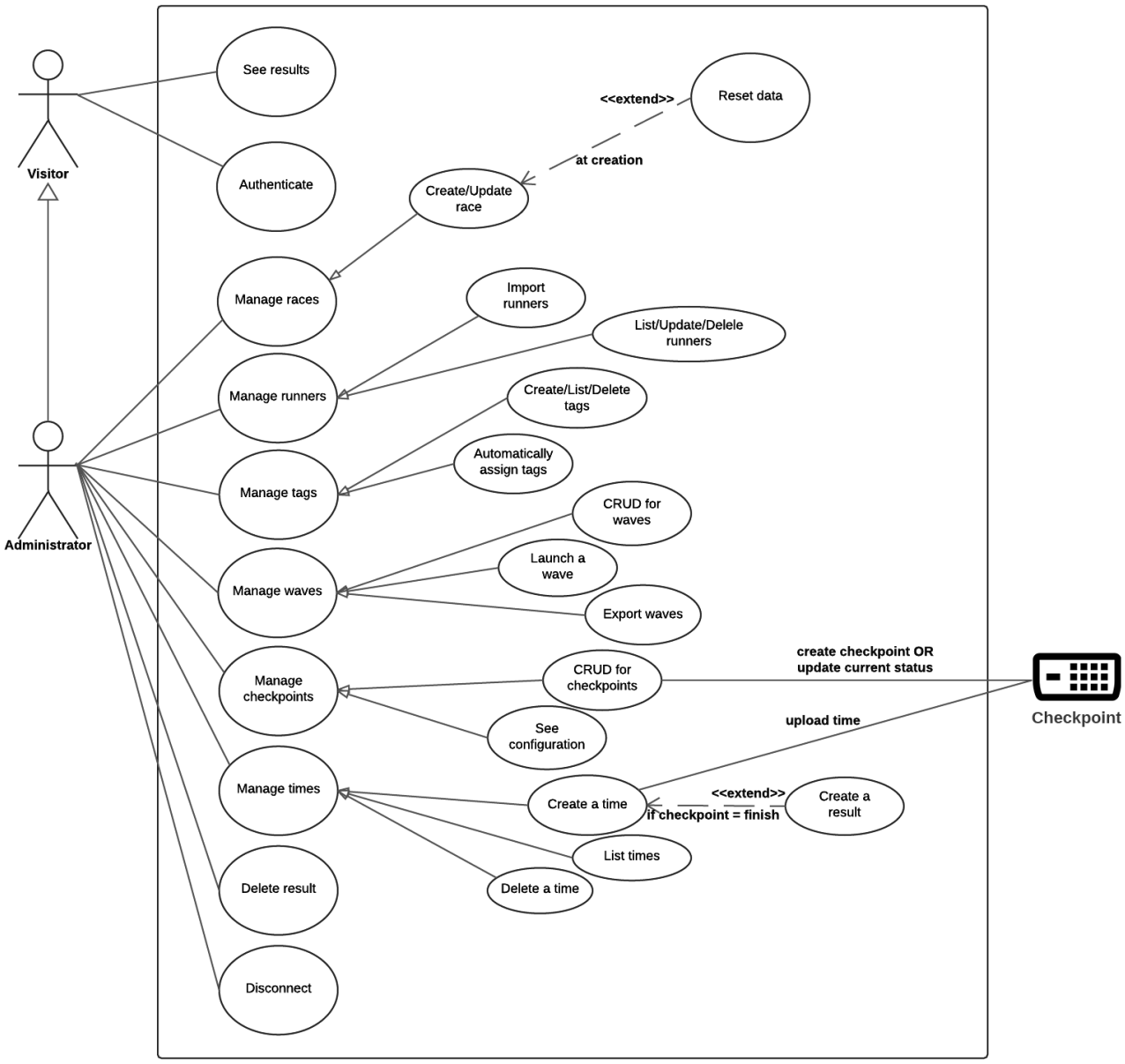


Figure 4.1: Diagram of use cases

Explanation of the use cases

- Visitor
  - **See results**: visitors can see the results of the current race.
  - **Authenticate**: visitors can authenticate. When they succeed to do so, they become *administrators*.

- Administrator
  - **Manage the races**:
    - **Create a new race** replaces the current race and resets the data (it deletes all the runners, waves, times, results and un-assigns all tags).
    - **Update the current race**
  - **Manage the runners**:
    - **Import runners** consists of importing a formatted Excel file containing all the runners' data (and thus the waves' data)
    - **List the runners** shows all the runners in a table.
    - **Update a runner**
    - **Delete a runner**
  - **Manage the tags**:
    - **Create tags** by providing a range of numbers (1-100, 1-1 for one) and a color.
    - **List the tags** shows all the tags in a table.
    - **Delete tags** by providing a range of numbers and possibly a color.
    - **Assign automatically the tags** assigns all the tags to the runners that are in a timed wave (*Elite*).
  - **Manage waves**:
    - **Create a wave**
    - **List the waves**
    - **Update a wave**
    - **Delete a wave**
    - **Launch a wave** consists of setting the start time of a wave to the current time (now).
    - **Export waves** allows the administrator to export the waves and their runners in a PDF file.
  - **Manage checkpoints**:
    - **Create a checkpoint**
    - **List the checkpoints**
    - **Update a checkpoint**
    - **Delete a checkpoint**
    - **See configuration** allows to see the current configuration of a checkpoint.
  - **Manage times**:
    - **Create a time** is useful if a checkpoint does not work correctly. If the checkpoint is the finish (id equals to "99"), it also creates a result.
    - **Delete a time** is usually used only for testing purpose before a race.
  - **Delete a result**: is used to delete results. This functionality is usually used for testing purpose before a race.
  - **Disconnect**: self-explanatory. When administrators disconnect, they become *visitors*.
- Checkpoint
  - **Create a checkpoint**: when configuring a checkpoint, it can create its "representation" by itself on the server.
  - **Update a checkpoint**: a checkpoint needs to tell at regular intervals that it is connected (update its "online" status).
  - **Create a time**: a checkpoint needs to be able to create ("upload") a time. If the checkpoint is the finish, it also creates a result.

4.4 Data models

As you will see later, we began our development with **Firebase** as the database. Firebase is a Database as a Service proposed by Google which is essentially a NoSQL database that pushes in real-time the changes as soon as they happen to its clients. After two months of development, we changed our mind and decided to use MongoDB, which is also a NoSQL database (but this time self-hosted). Why ? Because this change allowed us to have more flexibility and do more complex queries while reducing the costs in money for Game of Trails. The reasons behind this change are explained more thoroughly in the development chapter.

Therefore, our collections (somehow the equivalent of the tables in a relational database) don't have any explicit foreign key. NoSQL prefers to copy the same data in multiple places and thus have redundancy than to have relations between tables and foreign keys. *In our diagram, the relations between the collections are therefore only there to facilitate the comprehension. Indeed, they show where the duplicated data come from.*

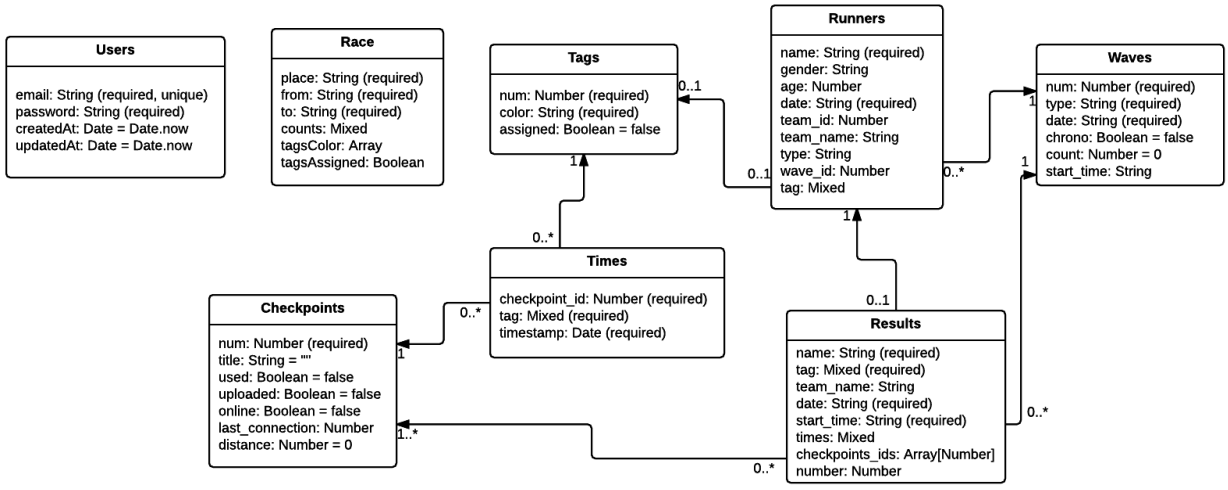


Figure 4.2: Diagram of collections

Explanation of the collections

- **Users** contains all the data related to the administrators. It has an *email*, a *password* and two timestamps to know when it was created and when it was last modified. The *email* and the *password* are mandatory, and the *email* has to be unique.
- **Race** contains the data related to the current race. This consists of the *place* of the race (i.e. Sart Tilman), the *from* and *to* dates, the *counts* of runners for each day, the colors of the different tags and a boolean to indicate if the tags have been assigned or not. The *place* and the *from*/*to* dates are mandatory.
- **Tags** represents the tags in the system. A tag is represented by a *number* and a *color* (i.e. 22 Orange). It also has a boolean to indicate if it is assigned to a runner or not. The *number* and the *color* are mandatory.
- **Runners** represents the runners in the system. A runner has a *name*, a *gender*, an *age*. The *date* is the date at which the runner is expected to participate to the race. A runner

may have a *team name*, a *wave type* and a *wave number*, and also a *tag*. The *name* and the *date* are mandatory.

- **Waves** consists of all the waves known in the system. A wave has a *number*, a *type* (i.e. fun or elite), a *date* (the day the wave belongs to), a *count* to know how many runners are in that wave, a *start time* that is necessary to compute the results and finally a boolean to know if the wave is timed or not. The *number*, the *type* and the *date* are mandatory.
- **Checkpoints** represents the checkpoints during the course of the race. A checkpoint consists of a *number*, a *title* (i.e. 'Arrivée') and a *distance* from the start (i.e. 14km). It also has a boolean to know if it is *online* and a *timestamp* to know when was the last time it was connected. The *number* is mandatory.
- **Times** is the collection containing the times of the runners. A time has a *checkpoint number* (where the time has been generated), a *tag* (the tag of the runner that went through that checkpoint) and a *timestamp* (at which time the tag was scanned). All the data here are mandatory.
- **Results** is the last collection. It contains the results computed from the times for each runner. A result has a *name*, a *tag*, a *team name* and a *date* which all come from the runner the result belongs to. It also contains a *start time* (equals the start time of the wave of the runner). Finally, it has a dictionary of *times* (one for each checkpoint the runner went through), an *array of checkpoints' number* (contains the number for each checkpoint the runner went through) and a *number* (which represents the place of the runner in the ranking, i.e. 1 (winner), 2, etc.).

As you will see in the improvements' section, we think that the choice we made of using NoSQL might not be the best. In fact, a relational database such as PostgresSQL may have been more suited and more appropriate in the long run.

4.5 Real-time feature

The real-time feature deserves its own section because it is still a rare feature in the world of web development. As a reminder, the goal is to have the results, which are visible for all the visitors on the front-end (the website), updated in real-time. That means that when a runner passes the finish line, his result should be visible on the website as soon as his time has been sent to the back-end and his result has been generated.

There are two different methods to implement this real-time feature. We will model, explain and show both of them with the help of sequence diagrams. The purpose of those diagrams is to give you a better view of how all the components of the system work together and what should be their interactions.

Long polling

The first way, which is how real-time has been implemented on websites for a long time, is known as *long polling*. The idea is that the front-end periodically asks the back-end if anything new happened since the last time. In this method, it is the front-end that does everything. The back-end does not notify by itself the front-end that new data exists.

Note that there are no state kept in the back-end to store the visitors in a way or another. The back-end simply answers to the requests and afterwards directly forgets what it did. Therefore,

no connections are left open.

We can model this method by the sequence diagram shown in the figure 4.3.

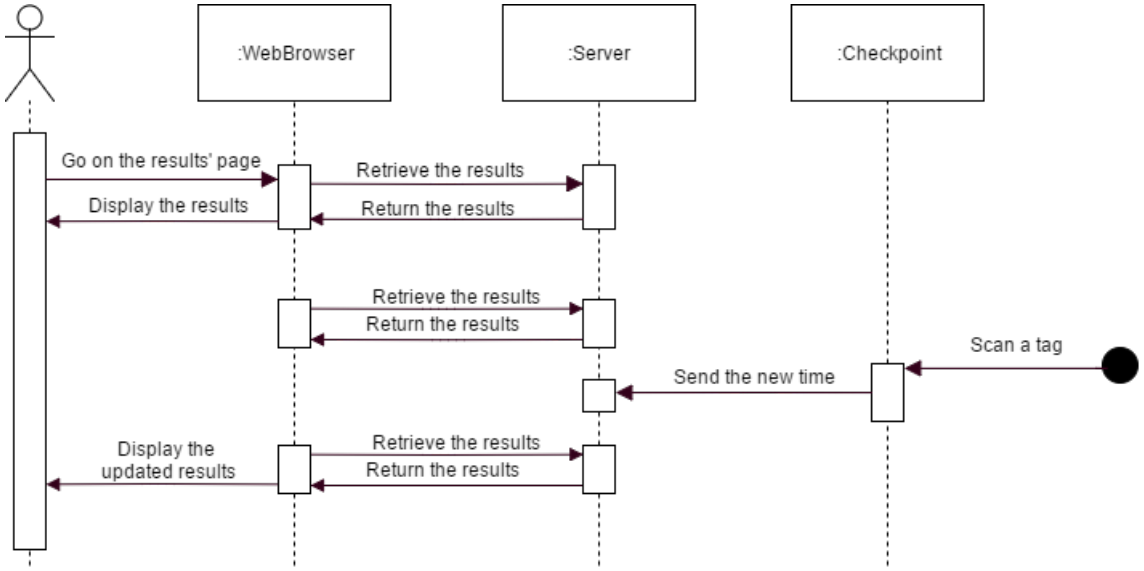


Figure 4.3: Sequence Diagram of the results' page and its live updates

As you can see, when a visitor accesses the results' page, the web browser retrieves the results by sending a request to the back-end. The server returns to the browser the results asked. Afterwards, the browser sends a request at an arbitrary interval to get the new results, if there are any. That means that requests are sent even if there are nothing new. This method can lead to a lot of overhead if there are a lot of visitors waiting on the results' page.

WebSockets

The second method for implementing real-time is to use *WebSockets*. WebSockets consist of maintaining the connection alive between the browser and the server instead of stopping it when all the resources have been loaded. This type of connection enables a full-duplex communication. Indeed, HTTP usually only allows the browser to send requests to the server and not the reverse. With Websockets, in our case, that means that the server can send new information at anytime to all the browsers currently connected. Therefore, it can send to all the visitors the new results as soon as they are generated on the back-end.

You can see the model of this method in the figure 4.4.

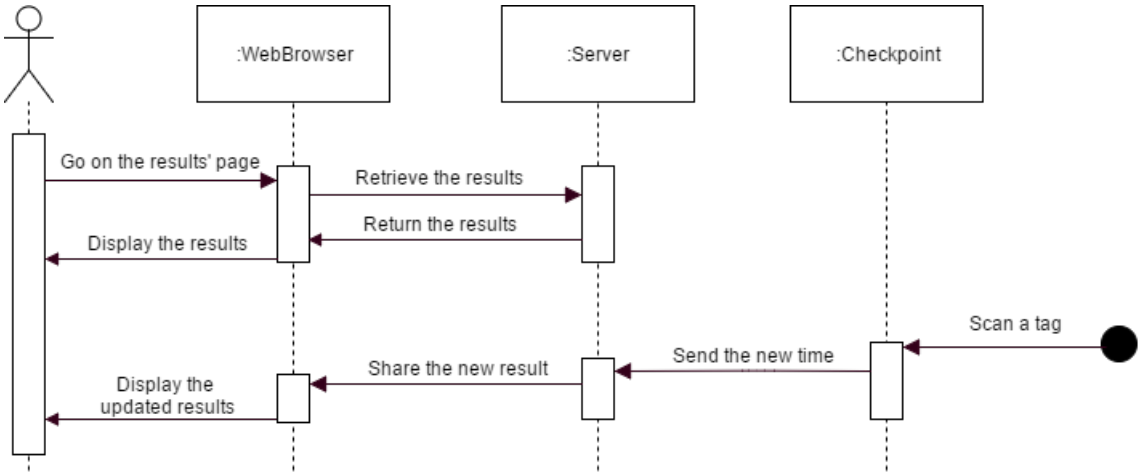


Figure 4.4: Sequence Diagram of the results' page and its live updates

The sequence is quite simple. First, when a visitor accesses the results' page, the web browser of that visitor asks to the server to get all the results. The server returns to the web browser all the results and the web browser shows them to the visitor.

Whenever a tag is scanned at a checkpoint, and if that checkpoint is connected to the Internet, the checkpoint sends the information to the server (what we call a "time"). Considering that the checkpoint is the checkpoint at the finish line, the server parses the time received and creates a new result for the runner with that tag. The server then sends to the web browser of the visitor the new result that has just been created. Finally, the web browser updates the web page to show to the visitor the updated table of results.

However, compared to the long polling, the server needs to keep in memory the visitors that are currently connected to it in order to send the new data to them. It therefore increases a little the memory usage.

4.6 Graphical User Interfaces

Lastly for the requirements, let's see what are the mock-ups of the graphical user interfaces. In fact, they are quite straightforward. First, there are the mock-ups for the pages accessible to everyone.

Visible to everyone

The **home page** is the page where visitors land first when coming to the website. As you can see in the figure 4.5, it just welcomes the visitors and shows buttons that allow to go to the results' page of each day.

Figure 4.5: Mock-up of the home page

The **results' page**, as you can see in the figure 4.6, is used to display the results to the visitors. Ideally, it is live-updated. The template of the page is the same for each day of the race (usually Saturday and Sunday), only the data of the results change. The visitors should be able to search for a specific runner by specifying his name or his team name.

Figure 4.6: Mock-up of the page of results

The last page accessible to everyone is the **login page** visible in the figure 4.7. Its goal is self-explanatory.

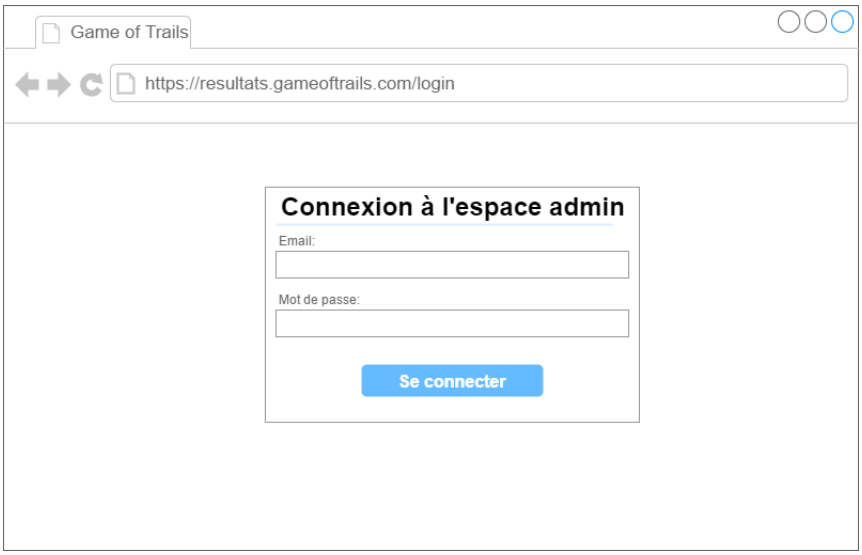


Figure 4.7: Mock-up of the login page

Administrators only

We are not going to show all the pages here, as it would take too much place, but only the main ones. Still, we will provide a small description for each one of those not shown.

The **dashboard** is the main page of the admin section. We call the "admin section" all the pages that are accessible only to the administrators. As you can see in the figure 4.8, the dashboard allows the administrators to have a glimpse of the current race planned/ongoing. It has a pie chart showing the number of runners for each day and a bar chart showing the number of runners in each wave for each day. Some useful and important buttons are also accessible on this page such as the button to import runners with the help of an Excel file and the button to export the waves as a PDF file.

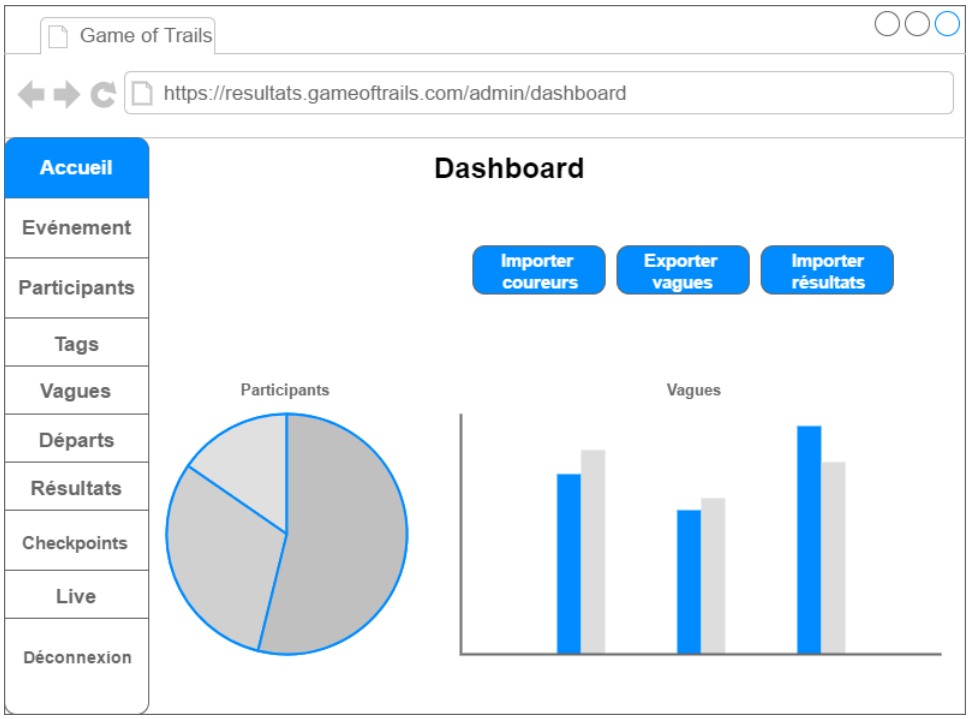


Figure 4.8: Mock-up of the dashboard

Once the runners have been imported thanks to the button in the dashboard page, the administrators can see the runners and the waves in their respective pages. The **runners' page** allows to see the details of all the imported runners. The administrators can modify the information of any runner or delete a specific runner if needed. In the figure 4.9, the M button allows to modify the runner of that line while the X button allows to delete that runner. Like the results' page, the administrators should be able to search for a specific runner with his name, team name or tag ID.

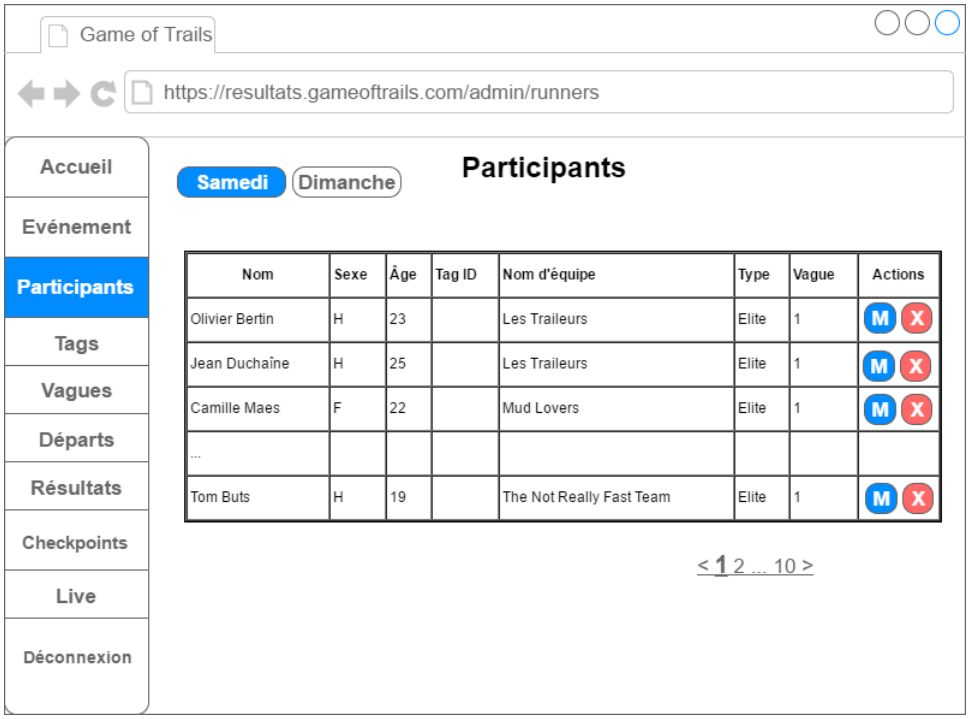


Figure 4.9: Mock-up of the runners' page

To avoid having too much mock-ups and thus figures in this chapter, the remaining pages in the admin section will be described.

- The **Evenement** page allows the administrators to create a new race or modify the current one. The goal of creating or modifying a race is to allow the system to know on which days the actual race will occur.
- The **Tags** page gives the possibility to the administrators to register or to remove tags from the system. They can do so by providing a range of numbers and a color (for example adding 1 to 100 orange tags). This page also has a button to automatically assign the tags.
- The **Vagues** page is the page that allows the administrators to manage the waves. It shows a table summarizing the waves and gives the possibility to modify each one of them.
- The **Départs** page is an important page as it allows the administrators to launch a specific wave during the days of the actual race. This start time is afterwards used to compute the results of the runners. This page therefore lists the waves of each day and has a button to launch (or reset) each wave.
- The **Résultats** page is a simple page that lists the results already generated by the back-end. It allows to delete a result if the administrators think it's erroneous and allows the creation of a time (which may generate a new result). The idea is to give some possibilities of correcting a problem during the race.
- The **Checkpoints** page lists the checkpoints known by the system. For each checkpoint, the administrators can see if it is online, how many runners have already go through it and see what is its configuration. The administrators can also specify a name for each checkpoint or delete them.
- The **Live** page allows to see in live the times as their are uploaded in the back-end. It lists those times starting by the most recent one. It also allows to delete a time if the administrators think it's erroneous or if it was just a test.

Chapter 5

Development

In this chapter, we talk about the development of the race timing system modeled in the requirements. More precisely, we explain what are the technologies we decided to use, how we implemented the system by using those and what are the hardware bought to build it.

First, we have to precise that we changed our choice for the database at the middle of the thesis and added at the same time a back-end. This is principally due to the evolution in the requirements from the client, Game of Trails. We explain this in more details in the database section. You can see in the table 5.1 a summary of the technologies used initially and after the change.

	Initial stack	**Final stack**
Front-end	Angular	Angular
Back-end	/	Feathers
Database	Firebase	MongoDB
Checkpoints	Python	Python

Table 5.1: The two stacks

Our master thesis, as the name suggests, is open source. That means that all the source code of our race timing system is freely available. This source code is hosted on Github in public repositories, one per part of the system (except the database, as there are no source code). If interested, those repositories can be found at the following link: `https://github.com/osrts`.

The chapter is structured as follows: we first look at the global architecture of our race timing system. This is the best way to give you a broad idea of how it is composed and how the different parts of the system interact with each other. Afterwards, we give more concrete details about the main parts of the system. We begin with the database and explain what are the two choices we made and why we changed. Then, we dive into the back-end which is one of the most important part of the system. Afterwards, we continue with the front-end and therefore the website in itself. Finally, we explain the checkpoints, in particular the hardware and the implementation.

5.1 Architecture

As presented in the requirements, our system is composed of four parts. Those are the **database**, the **back-end**, the **front-end** and the **checkpoints**. You can see a global view of the architecture and the relations between those parts in the figure 5.1.

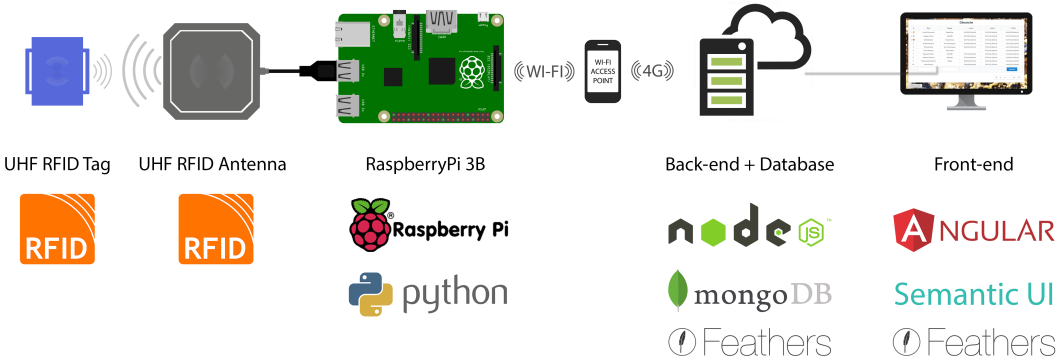


Figure 5.1: Architecture of the race timing system

5.2 Database

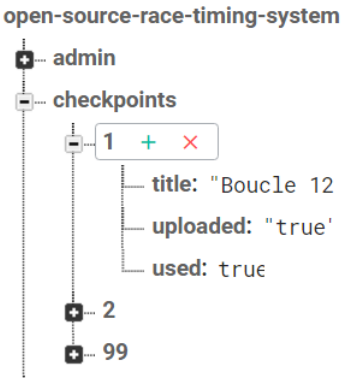
We introduce here the two choices we made for the database. First, we present *Firebase*, the technology we decided to use at the beginning. Afterwards, we explain what are the reasons behind the change we made. Then, we present *MongoDB*, the current database of the race timing system.

Firebase

Firebase [11] is a NoSQL cloud-hosted database that gives real-time updates on all the devices/clients connected to it. In other words, it directly informs all of its clients when a piece of data in the database is created, updated or removed. The fact that is has a free tier and the existence of the out-of-the-box real-time feature were the main points that interested us. This database is quite unconventional as it is self-sufficient. Usually, a database is not accessible from the clients, it is only accessible on a local machine or in a private network and the data is only really accessed by the back-end. However, here, Firebase acts as a database and somehow as a back-end too by providing hosting, authentication and direct access. However, note already that it is a proprietary service and that it can't be self-hosted.

Of course, this cloud-hosted database provides libraries in multiple programming languages and for multiple frameworks. Our front-end framework, Angular, has therefore a library specially conceived to work with it.

The database in itself is stored as a large JSON[1] document [12]. It is usually the case for NoSQL databases. You can see in the figure 5.2 the structure of it in both the view of the web console of Firebase and as a JSON document. Note that document can be inside other documents (except the root document) and that there are no "schema" specifying what is the expected data. In fact, Firebase allows to add any field wherever and whenever it is needed.



(a) In the web console

```
{
    admin: {}
    checkpoints: {
        1: {
            title: "Boucle 12",
            uploaded: true,
            used: true
        },
        2: {},
        99: {}
    }
}
```

(b) As a JSON

Figure 5.2: Firebase's real-time database

The real-time feature in Firebase is implemented thanks to Websockets [13] (as a reminder, Websockets allow to keep a duplex connection between the browser and the server). We think that Websockets are the best way to implement the real-time feature as it exchanges data only when needed and without the intervention of the clients.

You can see the free plan and the "pay as you go"[2] plan in the table 5.2. We only included the two plans that are the most interesting (the *Blaze* plan costs $25/month and therefore was excluded). Moreover, we included only the parts that are interesting to us such as the real-time database and the hosting for the front-end.

	Spark Plan	**Blaze Plan**
Price	Free	Pay as you go
Real-time database		
Simultaneous connections	100	Unlimited
GB stored	1GB	$5/GB
GB downloaded	10GB/month	$1/GB
Hosting		
GB stored	1GB	$0.026/GB
GB transferred	10GB/month	$0.15/GB
Custom domain & SSL	✓	✓

Table 5.2: Comparison of the most interesting plans of Firebase

[1] JSON (JavaScript Object Notation) is a lightweight format for storing and exchanging data. It is easy for the humans to read and write while still being easy for machines to parse and generate.

[2] "Pay as you go" means paying only for what you really used.

As you can see, these plans are quite interesting. The initial idea was to use the *Spark* plan during the development phase and to switch to the *Blaze* plan for the actual race. The total cost would have never be high because of the size of our system. For the real-time database, it would never go above 1GB (only 1MB with 3000 runners), and for the hosting, it takes way less than 1GB (only 85MB during development).

Why change ?

As the development was progressing and the requirements given by Game of Trails were changing gradually, we had more and more difficulties to use Firebase. Indeed, we had an increasing number of complex queries to make and we also needed more and more features that Firebase did not have but instead a real back-end could provide us. Indeed, Firebase was not, at that time, offering a back-end or a possibility to run computations with its database service. It offered a storage, which we intended to use to host the front-end, but there was no possible way to do some computations outside of the browsers. Therefore, the only possible solution was to pay for a Virtual Private Server. That meant adding a cost in the equation. We were using Firebase with the free tier, but we knew that we would have to pay later when the race occurred because of the limit of simultaneous connections (100). Considering the number of Elite runners, we could assume that there would be at some time more than 100 visitors on the website.

The decision was difficult to make, but we needed more and more possibilities offered by a back-end (such as importing an Excel file or exporting a PDF file). Finally, we decided to see if we could host both a back-end and a database on a single Virtual Private Server and in the same time stop using Firebase.

We decided therefore to use MongoDB. You will see in the next section that it is also a NoSQL database and it is largely known in the Node.js ecosystem. The choice of another NoSQL database has been made for one big reason: time was running, we needed to transfer our data models on a new database rapidly. Because MongoDB is also a NoSQL database, the shift was easy to make and the data models didn't change much.

MongoDB

MongoDB is a free and open source NoSQL database software. It is cross-platform and it is maintained by MongoDB Inc. This database is said to be document-oriented and more precisely to use JSON-like documents. In its (free) community edition, it is "only" a database and does not offer out-of-the-box real-time or any hosting capabilities like Firebase. Usually, it is self-hosted and needs a back-end to act as a gate between itself and the world. Indeed, for security reasons, it is commonly not directly accessible to the public.

MongoDB, like Firebase, stores its data as a big JSON document [14]. To ease the use of MongoDB with multiple different applications, the data is split into *databases*. They have the same purpose as the databases in the relational database model. Those databases are then split into *collections*, which are somehow equivalent to the tables in relational databases. For example, in our case, we have one database called "openSourceRaceTimingSystem" and multiple collections inside it such as "runners", "times", etc.

Like Firebase, MongoDB allows documents to be nested inside one another and allows to add any field anywhere at anytime. Again, there are no schema specifying the expected data to be stored (however, you will see later in the back-end section that we use a package to accomplish that).

MongoDB allows to do more sophisticated queries than Firebase. First, the basic comparison operators are complete : *$eq* (equals), *$neq* (not equals), *$gt* (greater than), *$gte* (greater than or equal), *$lt* (lower than), *$lte* (lower than or equal), *$in* (if in given array) and *$nin* (if not in given array). In the same way, the basic logical operators, *$not*, *$and*, *$or* and *$nor*, are available when querying. It also allows to use some special operators. In those, we only used *$regex*, which, as the name suggests, allows to search with a regex. It is useful for the searches done on the website such as with the runner's name or the runner's team name. See an example of a query with a *$in* operator in the listing 5.1, it returns the documents in the collection "inventory" that have a status equal to "A" or to "D".

```
1    var cursor = db.collection('inventory').find({
2        status: { $in: ["A", "D"] }
3    });
```

Listing 5.1: Query in MongoDB

You will see later in the back-end section that we don't directly send queries to MongoDB. A special package, *Mongoose*, acts as a gate in front of the database and gives back "schemas" to enforce some fields and their types.

5.3 Back-end

As explained in the database section, we did not have a back-end at the beginning of the development. It is only when we decided to use MongoDB that we had to find a back-end to work with. Of course, we had to keep in mind the real-time feature. After a lot of research, we decided to use the Feathers framework which runs on Node.js. We first explain what Node.js is as it is the foundation of the back-end. Then, we will see how Feathers allows to easily build real-time applications.

5.3.1 Technologies

Node.js

Node.js is a JavaScript run-time built on Chrome's V8 JavaScript engine. It uses an event-driven, non-blocking I/O model and is therefore asynchronous. This model allows Node.js to be lightweight while still being efficient. It has one of the largest ecosystem of open source libraries in the world called *npm*.

As mentioned, Node.js is event-driven. It uses an *event loop* that can be seen as a queue[3]

[3]In reality, it has multiple queues with different priorities, but this is beyond the scope of this thesis.

of callbacks waiting to be processed by a single thread. It processes those callbacks one after another until the queue is empty. A *callback* is a function that is given in parameter when doing an (intensive) operation. As the name suggests, this function is called when the intensive operation has completed. It allows the developer to do something with the result of the operation. A callback usually takes an error (empty if nothing went wrong) and the result (empty if an error occurred) as arguments.

This event loop enables Node.js to perform multiple non-blocking I/O operations even though, due to the nature of JavaScript, it is single-threaded. As you can see in the figure 5.3, a callback is generated when performing an intensive operation such as writing in a file or querying a database. This callback is inserted into the event loop when its related operation has completed. In the same way, when a request arrives, it is inserted in the event loop.

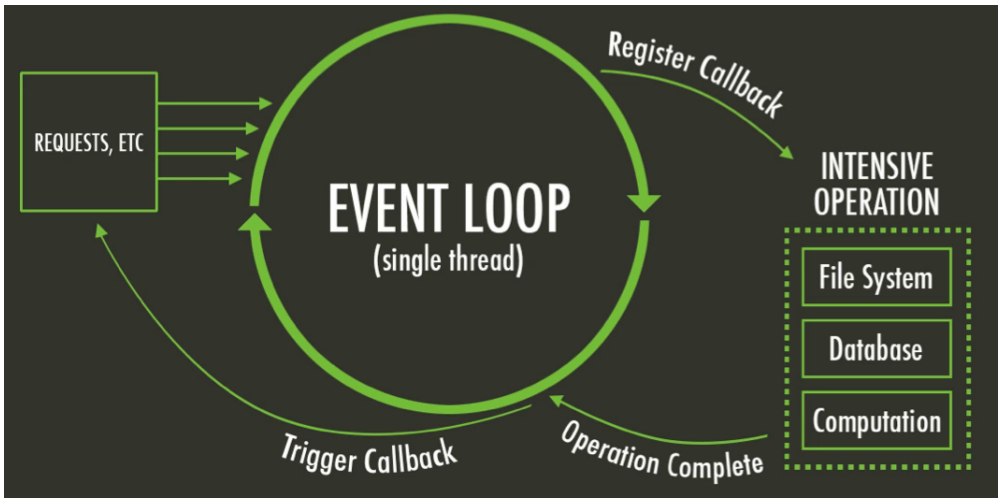


Figure 5.3: Node.js event loop

You can see the usual "Hello World" example[4] written in JavaScript for Node.js in the listing 5.2.

```
1  const http = require('http');
2
3  const hostname = '127.0.0.1';
4  const port = 3000;
5
6  const server = http.createServer((req, res) => {
7    res.statusCode = 200;
8    res.setHeader('Content-Type', 'text/plain');
9    res.end('Hello World\n');
10 });
11
12 server.listen(port, hostname, () => {
13   console.log(`Server running at http://${hostname}:${port}/`);
14 });
```

Listing 5.2: "Hello world" example for Node.js

This source code goes as follows: the first line with the *require* function allows to retrieve

[4]For launching a Node.js application, save it in a ".js" file and start it with "node your_file.js". Note that Node.js has first to be installed on your system.

the *http* package. In this case, it is an internal package of Node.js. Then, it defines two more constants, the host name and the port where the web server will run. The function *http.createServer* on line 6 allows to define the default action when receiving a request. This default action is in fact an arrow function[5] that takes the incoming request and the response builder in arguments. As you can see, we can set the HTTP status code (usually 200 for a successful request), the HTTP headers and finally we can set the body with the *end* function. This last function, *end*, also closes the current request and sends the response to the client. Finally, the last call at the line 12 is on the function *listen*. It is what actually launches the web server. The arrow function given as the third parameter is a callback usually used to perform some operations after the launch succeeded (here we simply log that it successfully launched).

The "Hello World" example is straightforward but applications are usually not implemented directly in this way. Indeed, even though it's already simple to create web servers, other packages have been built on top of the *http* package to ease even more the process of developing web servers in Node.js. A good example of such package is *Express* [16]. It is one of the most used framework in the Node.js ecosystem. The framework we used, *Feathers*, is built upon Express. Usually, for an application, you use several different packages. To add a package to a project, you first need to add it with *npm* (node package manager). You can find a list of the commands in the listing 5.3. Afterwards, as with the *http* package, *require(NameOfPackage)* is used. A project always has a *package.json* file that contains all the packages that were added to this project. Note also that some packages are only used for development (e.g. testing frameworks).

```
1 /* Initialize a Node.js project (creates a package.json file) */
2 npm init
3 /* Add a package (option --save adds it to the package.json file) */
4 npm install name_of_package
5 /* Remove a package (option --save removes it from the package.json file) */
6 npm uninstall name_of_package
```

Listing 5.3: Commands for managing packages

Finally, Node.js has a huge ecosystem for web server development, but it is also important to note that it has a huge ecosystem for the other part of the equation, which is the front-end. A lot of well-known front-end frameworks and libraries are also on *npm*, such as *Angular2*, *React*, *jQuery*, etc.

Feathers

Feathers is an open source REST[6] and real-time API[7] layer for modern applications. It is built over *Express* and *Socket.io*, which are two largely known and used packages for Node.js. As explained before, Express eases the creation of web servers. On the other hand, Socket.io eases

[5]The syntax **(params)=>{...}** creates an arrow function in JavaScript (ES6). It has a shorter syntax than the usual "function(params){...}". Note also that this syntax does not create a new context but instead keeps the enclosing one. [15]

[6]REST stands for REpresentational State Transfer. It means that it uses the different types of HTTP request such as *get*, *post*, *put*, *patch* and *delete*.

[7]API stands for Application Programming Interface. An API is an interface that can be used to interact with an application.

the use of Websockets (duplex connection between the browser and the server). The simplest example of a Feathers application can be found in the listing 5.4. It is simply the base skeleton of a Feathers application.

```
/* app.js */
const feathers = require('feathers');
const rest = require('feathers-rest');
const socketio = require('feathers-socketio');
const bodyParser = require('body-parser');

/* A Feathers app is the same as an Express app */
const app = feathers();
/* Parse HTTP JSON bodies */
app.use(bodyParser.json());
/* Parse URL-encoded params */
app.use(bodyParser.urlencoded({ extended: true }));
/* Register hooks module */
app.configure(hooks());
/* Add REST API support */
app.configure(rest());
/* Configure Socket.io real-time APIs */
app.configure(socketio());
/* Start the server */
app.listen(3030);
```

Listing 5.4: Example of a simple Feathers application

The first thing to know about Feathers is that it allows to define easily *services*. Those services are the heart of Feathers and they can be used to perform actions of any kind. In our case, they are mainly used to interact with the database, MongoDB. Each service has to be linked to a path. For example, the service "Runners" could have the path "/runners". A service can define the methods *create*, *find*, *get*, *patch*, *update* and *remove*. Those methods, if defined, are automatically called by Feathers when a request comes. Note that these methods are in fact related to their HTTP counterpart : when receiving an HTTP *post* request on "/runners", the *create* method is called. In the same way, when receiving an HTTP *get* request, it calls the *find* or the *get* methods depending if an *id* has been given in parameter or not. You can see a summary of the mapping in the table 5.3. The *id* is the string given in the path after "/runners/" and the *data* is the body of the request.

Service method	**HTTP method**	**Path**	**Description**
.find()	GET	/runners	Retrieves a list of all matching runners
.get()	GET	/runners/1	Retrieves the single runner having *id*
.create()	POST	/runners	Creates a new runner with *data*
.update()	PUT	/runners/1	Replaces the runner having *id* with the given *data*
.patch()	PATCH	/runners/1	Merges the runner having *id* with the given *data*
.remove()	DELETE	/runners/1	Deletes the runner having *id*

Table 5.3: Mapping HTTP methods to service methods

Fortunately, we don't have to implement ourselves every service we need. Feathers provides packages that contain services that already implement all these functions in a reusable way for several databases (MongoDB included). You can see an example in the listing 5.5. Note the *Mongoose* package, it is used to define a schema and a model for each MongoDB collection. Once the model created and the service running, Mongoose allows to automatically run

checks on the data such as, for example, checking that a field is present (*required: true*). It has the same purpose as the constraints in relational databases, but here it is executed on the back-end.

```
/* Inside message-model.js */
const mongoose = require('mongoose');

const Schema = mongoose.Schema;
const MessageSchema = new Schema({
  text: {
    type: String,
    required: true
  }
});
const Model = mongoose.model('Message', MessageSchema);
module.exports = Model; /* defines what is exported from this file */
```

```
/* Inside app.js */
/* Other requires */
const service = require('feathers-mongoose');
const Model = require('./message-model'); /* loads the model above */

/* Initialization of app not shown (see the simple example for it) */
app.use('/messages', service({Model})); /* Initializes the service */
app.listen(3030);
```

Listing 5.5: Example of a simple "message" service

In the previous example, if you send a *get* HTTP request to "http://localhost:3030/messages", you will receive all the messages stored in the collection "messages". Note that the collection is created by Feathers if it doesn't exist. If there are no message, it returns an empty array.

But what if we want to restrict the access to some services ? Or if we want to run some custom code before updating something ? This is where the *hooks* come in handy. *Hooks* are special pieces of code that can be executed before or after any service method (create, get, ...). For example, we can check if the current user is authenticated before doing an action or we can add a field "createdAt" when adding a new piece of data. Look at the listing 5.6 for an example.

```
const messages = app.service('messages');
messages.hooks({
  before: { /* Before the function */
    create(hook) { /* For the create function */
      hook.data.createdAt = new Date(); /* Add a field */
    }
  }
});
```

Listing 5.6: Example of a simple hook on the messages service

We talked about an authenticated user, but we have yet to specify how an user can log into our system. Again, Feathers has already everything implemented for us in its packages. It offers a few implemented authentication methods. The most common one is when you authenticate yourself with an email and a password. This is called *local authentication* in Feathers. When a visitor authenticates himself with that method, he receives in the response a token generated by the back-end. That token asserts that he has authenticated successfully and is called a JSON

Web Token (JWT)[8]. This token is usually added in the subsequent requests to prove that he has authenticated himself (otherwise the email and the password should be sent with every requests !). You can see in the example in the listing 5.7 that we initialize the application with both the *jwt* and the *local* authentications. Two other things have to be noted: first, a secret is given to the main authentication package. This is to add some randomness into the generation of the token and thus it strengthens the security. Secondly, you can see at the line 16 the hook used to protect a method of a service from unauthenticated users. This hook checks first that the visitor is authenticated with the jwt or local authentication before executing the function.

```
1  /* Inside app.js */
2  /* Other requires */
3  const auth = require('feathers-authentication');
4  const local = require('feathers-authentication-local');
5  const jwt = require('feathers-authentication-jwt');
6
7  /* Initialization not shown (see the simple example for it) */
8  app.configure(auth({"auth": {"secret": "xxxx"}}));
9  app.configure(jwt());
10 app.configure(local());
11
12 /* Protects the authentication service */
13 app.service('authentication').hooks({
14   before: {
15     create: [
16       auth.hooks.authenticate(['jwt', 'local'])
17     ]
18   }
19 });
```

Listing 5.7: Example of the authentication service

You may think now that one important feature is missing from this section: the real-time. If we retrieve some data with a *get* HTTP request, we can't have any real-time updates. This is due to the fact that the connection between the server and the browser is not left alive. However, if we use on the front-end the package *feathers-client* and use it to send requests to the service through *Socket.io* (as a reminder, Socket.io is a package to ease the use of Websockets), the connection is left open and we get the real-time updates. Note that nothing has to be added or modified on the back-end for it to work ! *Feathers-client* will be explained in more details in the front-end section.

Speaking of the front-end, where is it stored on the back-end ? *Express* and therefore *Feathers* (as it is built upon it) lets the developer specify a folder of the web server that can be used to serve static content. What is called "static content" is everything related to the front-end such as HTML, CSS and JavaScript files but also images and other files. Usually, this folder is named *public* and, as the name suggests, its content is publicly accessible.

Other packages

We have explained Node.js and Feathers, the main technologies building our back-end. Here, we introduce some other packages that were used for some specific use cases. We do not explain them in details as they are less essential to the system.

[8]JWT, it is an open, industry standard RFC 7519 method for representing claims securely between two parties.

xlsx [17] is a package that allows to parse and write easily spreadsheet formats. This is used for parsing the Excel file containing all the runners which is sent by the front-end.

EJS (Embedded JavaScript) [18] is a package that provides a language extending HTML. It gives the possibility to run JavaScript code such as *for loops*, *conditional*, etc., inside HTML. This is used to render a web page inside the back-end listing all the waves and their corresponding runners.

phantom [19] is a package that runs a scripted headless browser. In other words, it runs a browser without any graphical interface. It is used to "compute the display" and save the page rendered by EJS as a PDF.

5.3.2 Implementation

We have given general explanation about Node.js, Feathers and some other packages. Let's now see how we used them in our implementation.

Our services look for the most part like the basic service explained above. That means that the majority of them uses the service provided by *feathers-mongoose* which already implements all the service methods needed (*create*, *get*, etc.).

Hooks are used to protect the services from unauthorized accesses and to run some checks on the data before inserting or modifying it in the database. For example, we check that a time has not already been uploaded or that a tag belongs to a runner before using it. We also use hooks to update redundant data in multiple places. For instance, when updating the team name of a specific runner, we need to also update the team names inside the other runners of that team. Hooks help us achieve that.

The special features such as importing an Excel file or exporting a PDF file of the waves are done thanks to the packages explained above. More precisely, *xlsx* is used for the parsing of the Excel file and *EJS* in addition with *phantom* are used for the generation of the PDF file.

This last feature of generating a PDF is quite tricky. The process goes as follows: first, the EJS package is used to open an EJS file, which contains the template of the PDF, and then it generates the corresponding HTML without EJS. Secondly, Phantom renders this HTML in its headless browser and we save it as a PDF file in a variable. Finally, we send this PDF to the client in the response with custom headers. We had to create a special request handler with *Express* to implement this feature as *Feathers* does not allow to specify custom headers (such as 'Content-type: application/pdf') in the service methods.

5.3.3 Server

Finally, we needed a server to host the back-end and the database. Fortunately, cloud computing becomes more and more affordable nowadays. We had therefore no difficulties to find an offer that suited our needs. In fact, we had already used for ourselves the *VPS SSD 1* by OVH situated in France. For a few euros per month, it gives a VPS (Virtual Private Server) with:

- Single core CPU with a frequency of 2.4Ghz
- 2 GB of RAM

- SSD of 10 GB
- Local Raid 10

All of this comes, of course, with an unique IPv4 address (IPv6 also available) and an unlimited bandwidth. We checked once the prototype was developed that it was enough for the expected load of visitors during the actual race, and fortunately it was the case (more information will be given about the load testing in the testing section).

5.4 Front-end

From the beginning, we wanted our front-end to be a single-page application (SPA). This kind of technology has many advantages, for example:

1. The simulated different pages load much faster because each page is generated on the client side.
2. The better browsing experience since only the needed part of the page is refreshed when changing page.
3. The server load is reduced because less data must be transferred.

There are several great SPA frameworks available, in 2017 the market is dominated by 2 giants, Angular developed by Google and React developed by Facebook.

After several searches, we decided to use Angular [20] or more precisely version 2 of the framework. We chose it because one of us already knew it well which allowed the effective start of development. In addition to this, Angular 2 is highly demanded in the labor market. Even if the stable version of the framework was still not available at the beginning of our project, it has a very large community of developers and therefore it has a lot of documentations, examples and libraries.

Angular 2 uses TypeScript [21], a programming language created by Microsoft. The TypeScript language is a super set of JavaScript, this means that it has all the mechanisms that are standard JavaScript plus many others features that accelerate and facilitate the development.

One of the most important features of TypeScript is the static typing. This means that each variable have a given type and this type can not change afterwards. With this kind of information, the compiler and the developer can automatically eliminate a whole range of potential errors in the code. TypeScript also has a class concept, which allows for more object-oriented thinking and significantly simplifies the syntax. Under the hood, the whole concept of classes is based on prototypes, constructors and inheritance.

```
class Food {
    name: string;             // String
    calories: number;         // Numeric

    constructor(name:string, calories:number) {
        this.name     = name;
        this.calories = calories;
    }
    getName():string {
            return this.name;
    }
}

const burger = new Food('Burger', 650);
burger.getName(); // 'Burger'
```

Listing 5.8: Example of code written in TypeScript

When the TypeScript is translated into JavaScript, the classes, interfaces, types, etc. disappear. These elements are only used to facilitate the development stage and are not present in the generated JavaScript. In the listings 5.8 and 5.9, you can see the result of the translation. The types of the variables have been completely lost and the methods have been replaced by prototypes.

```
var Food = (function () {
    function Food(name, calories) {
        this.name     = name;
        this.calories = calories;
    }
    Food.prototype.getName = function () {
        return this.name;
    };
    return Food;
}());

var burger = new Food('Burger', 650);
burger.getName(); // 'Burger'
```

Listing 5.9: Example of the previous TypeScript code translated to JavaScript

Semantic UI

For the CSS, we used Semantic UI [22]. This framework allows to build beautiful websites with a responsive design while using human-friendly HTML and CSS class' names. It is based on LESS [9] and is much more powerful than Twitter's Bootstrap. Semantic UI has a great grid system and many very interesting and independent components that allowed us to quickly build a complete website.

ANGULAR CLI

[9]Less is a pre-processor which means that it extends the CSS to dynamic language elements such as variables, mixes, operations, and functions.

Starting a new Angular 2 project requires creating a proper folder structure, installing TypeScript, configuring the environment, etc. To facilitate this task, we have used Angular-cli [23], a command-line interface that facilitates and accelerates the creation and the management of an Angular project. With simple command '$ ng new osrts'[10] we can create a new project. Other commands exist to ease the creation of new components, pipes, modules, etc. inside the project.

The generated project allows us to work on a complete web application using the best practices provided by the Angular team. The folders are divided into application sources (TypeScript, HTML and styling codes), test files (including e2e), and static resources served in the application.

5.4.1 Implementation

In this section, we explain the implementation of the front-end. We focus on the essential elements such as the modules, the components and the services.

Here is a diagram (Figure 5.4) that represents the structure of the front-end. It contains the 2 modules that we have created as well as all the components of these modules. The blue blocks represent the pages components while the green blocks represent the small reusable UI components. You can also see the list of all the services we have implemented.

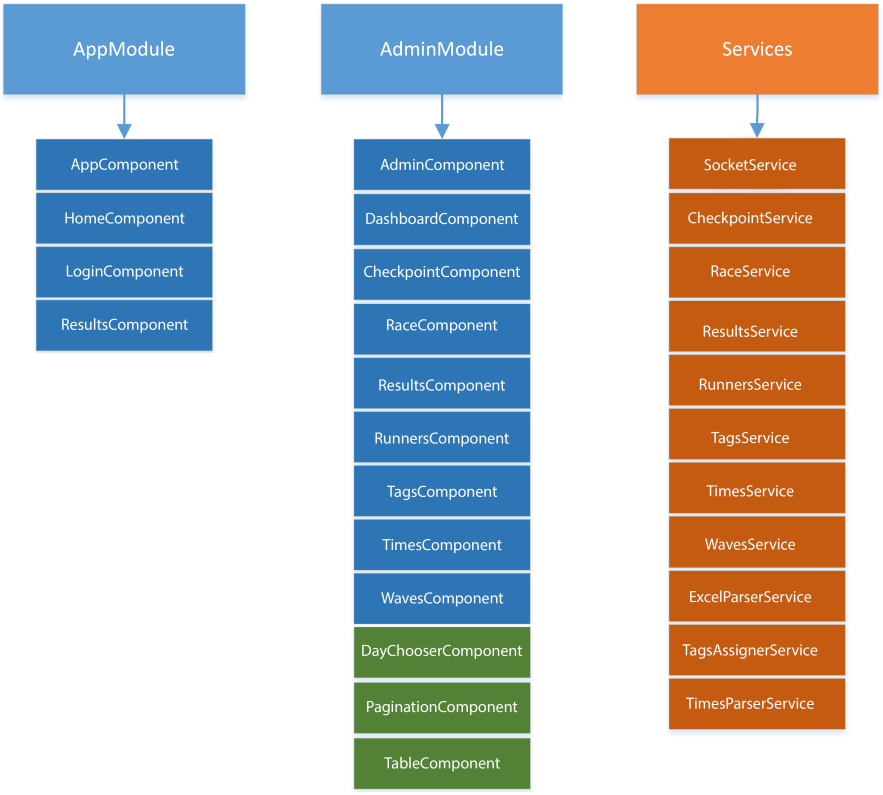


Figure 5.4: Front-end structure

Modules

First, let's see what is a module. Each Angular app has at least one root module usually named *AppModule*. The modules are used to group interrelated elements like components, directives, filters and pipes. Large applications usually are made of many modules, each responsible for a specific function. A module is composed of several elements:

[10] osrts stands for Open Source Race Timing System (our project name)

- Declarations : a list of components, directives, pipes, and filters that are part of this module. All elements declared here will be accessible throughout the module.
- Imports: a list of imported modules.
- Providers: a list of services used in this module.
- Bootstrap: a list of items that are the start points (bootstrap) of the application. In most cases, there is only one item, which is the main component of the application (AppComponent).

```
1 @NgModule({
2     declarations: [AppComponent],
3     imports: [],
4     providers: [],
5     bootstrap: [AppComponent],
6 })
7
8 export class AppModule { }
```

Listing 5.10: Example of an Angular module

We built our application in a modular way. Indeed, we have 2 modules, the **AppModule** that is loaded by default when the application is launched and the **AdminModule** that is loaded only on demand when an administrator is authenticated (this is called *lazy-Loading*). This division accelerates the loading of the application because, at first, only AppModule is loaded into memory.

The **AppModule** is very small, it allows only to consult the results of the race and to authenticate. The module **AdminModule** is much more complex, it has several pages allowing the management of the race. We have also implemented several components which are reusable within the module, these elements will be explained in the next section.

Components

In Angular, each page is a separate component and each component is an encapsulated self-contained application block. The components are used to create the UI (user interfaces) of an Angular application and they contain both the business logic and the view. A component can also consist of several small components, ideally each UI element that is used several times should be implemented as a re-usable component. This practice facilitates the development and avoids the redundancy of the code.

The listing 5.14 shows an example of a component who represents the main page of the application. The decorator *@Component* enriches the class with relevant meta-data that allows Angular to identify our classes as components. The purpose of the meta-data is to tell which selector represents our component and to indicate the HTML template and the style of the component. We can either specify a path to the files that contain the HTML / CSS, or we can place the code directly in the meta-data declaration.

```
@Component({
  selector: 'app-root',
  template: `<router-outlet></router-outlet>`,  // HTML Code
  styleUrls: ['./app.component.css']            // Path to the file
})

export class AppComponent {
  constructor() {}
}
```

Listing 5.11: Example of Angular 2 component

As we already said, a component is not necessarily a page, it can also be an independent UI element that is used in several places. In total, we have implemented 3 small components DayChooserComponent, TableComponent and PaginationComponent. Even if these component are not very complicated, it allows us to avoid unnecessary repetition of the code.

The use of components is very simple, when we create a component in the meta-data section we have to specify a selector. This selector can be used anywhere within the same module, for instance if we want to use the **DayChooserComponent** on the runners page we just have to add the **<app-day-chooser></app-day-chooser>** selector in the template HTML.

The DayChooserComponent (figure 5.5) allows us to easily choose for which day we want to consult the data. For example, when we are on the runners management page, we can see the runners' list for either Saturday or Sunday. Obviously the names of the days that are displayed depend on the date of the race, in our case it is the 15th and 16th of April 2017 which corresponds to Saturday (Samedi) and Sunday (Dimanche).

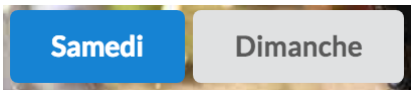


Figure 5.5: DayChooserComponent

The TableComponent (figure 5.6) handles sorting and filtering HTML tables. We can easily specify the columns that can be sortable / filterable. We can also choose the column on which we want to sort the data by default. For example, by default we sort the runners alphabetically.

N°	Nom	Tag	Team	Date	Start	Loop 1	Loop 2	Finish	Actions
1	Nathan Chantry	127 - Bleu	Wapiti	15-04-2017	12:09:54.344...	0:24:06		1:01:19	✖
2	Pierre Leso	99 - Bleu	Rock and Leso	15-04-2017	12:09:54.344...	0:22:36	0:47:41	1:02:44	✖
10	Pierre Pahaut	38 - Bleu	Familia Elite	15-04-2017	12:09:54.344...	0:26:27		1:20:15	✖
N°	Nom	Tag	Team						Chercher

Figure 5.6: TableComponent

The PaginationComponent (figure 5.7) handles the pagination of HTML tables. The operation is very simple: when a user clicks on the desired page, the component notifies its parent and the displayed data are modified.

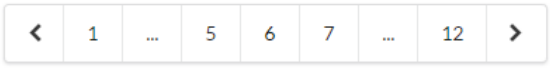


Figure 5.7: PaginationComponent

To be useful, the components must communicate with their parents. For instance when a user changes the day, the parent component must be notified in order to update the displayed data. This is possible through communication between the components [24] . The communication can be done in both directions, it means the parent component has the ability to pass the data to the child. This is possible with the **@Input()** annotation:

```
@Component({
    selector: 'app-table',
    templateUrl: './table.component.html',
    styleUrls: './table.component.css'
})

export class TableComponent {
    @Input() columns: any;
    ...
}
```

Listing 5.12: Example of @Input annotation

Now passing the data to the child is simply done with the following code:

```
<app-table [columns]="columns"></app-table>
```

Listing 5.13: Example of passing the data to the child component

The **@Output()** annotation allows us to achieve the opposite result so we can to pass values from the child to the parent. Let's take the example of the DayChooserComponent, when the user changes the day, the component notifies its parent of the change. This is done with an instance of **EventEmitter**. When we want to notify the parent of the change, we must call the **emit()** method of the changeDayEmitter object.

```
@Component({
    selector: 'app-day-chooser',
    templateUrl: './day-chooser.component.html',
    styleUrls: ['./day-chooser.component.css']
})

export class DayChooserComponent {
    @Output() changeDayEmitter = new EventEmitter();

    constructor() {...}
    changeDate(date) {
        // Notify the parent of the change
        this.changeDayEmitter.emit(date);
    }
}
```

Listing 5.14: Example of @Output annotation

This will trigger the **changeDay()** function of the parent and pass a new day as an argument.

```
<app-day-chooser (changeDayEmitter)="changeDay(event)"></app-day-chooser>
```

Listing 5.15: Example of passing the data from the child to the parent

Services

The results' page (figure 5.9) allows users to view the results of the race in real time. As soon as a runner is scanned at the finish line, we display his time at each checkpoint as well as his average speed. As explained in section 4.5 (Real-time feature), we use Websockets to maintain a connection with the server. When a new result arrives on the server, all clients are immediately informed and the results' page is updated. To connect to the server, we used the *feathers-client* [25], the client side library of Feathers which allows us to easily create a connection to the server using the Websockets (or REST API). In our Angular project, we have created a *SocketService* (listing 5.16). It is a custom service that uses the Feathers' client library to establish and maintain a connection with the server. When the service is initialized, the constructor of the service is executed which initializes the Feathers' client and configures it.

```
// Imports
...
const HOST = 'http://localhost:3030'; // Your base server URL here

@Injectable()
export class SocketService {
  public socket: any;
  public _app: any;

  constructor() {
    this.socket = io(HOST);              // Initialize socketio
    this._app = feathers()               // Initialize feathers
      .configure(socketio(this.socket)) // Fire up socketio with feathers
      .configure(hooks())                // Configure feathers-hooks
      .configure(authentication({ storage: window.localStorage }));
  }

  getService(serviceName){
    return this._app.service(serviceName);
  }
  ...
}
```

Listing 5.16: SocketService using Feathers' client

Afterwards, we created an Angular service for each Feathers service existing on the server. You can see an example of such service in the listing 5.17. Thanks to these services, we can retrieve data from the server in real time and we can send requests to add, update or delete data from the server. To summarize, the ResultsService (Angular) communicates directly (via the sockets) with the Feather service (on the back-end) that is responsible for the collection **Results** (MongoDB). To do this, we used the *getService(nameOfService)* method of the SocketService object in order to create a socket connection to the desired service on the backend. On this socket, we have added listeners who listen to the data changes (on the server). Indeed, it is the client that asks to the server to be notified when something changes for a specific service. For instance, when a new result arrives on the server, the listener that listens to the 'created' event will be triggered and the method **onCreated(result)** will be executed. In this method, we can, for example, send the new data to the components.

To exchange data between services and components, we used an object of type *RxJS Subject* [26]. The *subject* object is both an observable and an observer. This allows us to call the

subscribe() method so that we receive the new data, and at the same time we can pass new data with the **next()** method. A *subject* can also serve as a proxy between an observable and several observers. Therefore, with one subject we can receive the data from many sources and send the data to multiple observers.

```
import { Injectable } from '@angular/core';
import { Subject } from 'rxjs';
import { SocketService } from '../feathers.service';

@Injectable()
export class ResultsService {

  private _socket;
  public resultsSubject: Subject<any>;

  constructor(private _socketService: SocketService) {
    // Get the socket connected to the 'results'
    this._socket = _socketService.getService('results');
    // Listeners on socket
    this._socket.on('created', (result) => this.onCreated(result));
    this._socket.on('updated', (result) => this.onUpdated(result));
    this._socket.on('patched', (result) => this.onUpdated(result));
    this._socket.on('removed', (result) => this.onRemoved(result));
    // Store the results data
    this.resultsSubject = new Subject();
    ...
  }

  public create(result){...}        // Create new result
  public update(result){...}        // Update existing result
  public remove(id){...}            // Remove existing result
  public find(query: any) {...}     // Query the database

  // Executes when 'created' event is triggered
  private onCreated(result) {...}
  // Executes when 'updated' or 'patched' event is triggered
  private onUpdated(result) {...}
  // Executes when 'removed' event is triggered
  private onRemoved(result) {...}
}
```

Listing 5.17: Example of Results Service using Feathers client

Here is an example of how the service works. When we want to retrieve data from a service in a component, we inject the desired service and then we perform a subscribe on the subject (listing 5.18). At first, the subject is empty. We have to ask the service to retrieve the data from the server. We use the **find()** method to retrieve data and to push them in the subject object with the method *next()*. When the subscribe method receives the new data, the content of the page is automatically updated thanks to the two way data binding of Angular.

```
...
export class ResultsComponent {
  var data: [];
  constructor(private _resultsService: ResultsService) {
    // Results subscription
    this.resultsSubscription = this._resultsService.resultsSubject.subscribe((
        data) => {
        this.data = data
    });
    this._resultsService.find(); // Tells the service to retrieve the data.
  }
  ...

}
```

Listing 5.18: Component : subscription to the results' subject

5.4.2 Security

Our front-end is a single page application, which means that the visitors of the website download all the source code of the application (admin included) in order to display it in the browser. This is one of the biggest disadvantages of SPA. For this reason, it is very important to secure the back-end correctly. To do so, we have set up a token based authentication. When an administrator logs in our system with the correct user name and password, he receives in exchange a token which allows him to access private resources. When the authentication is successful, the token is used to initialize the secured connection to the server via the Websockets. To have the best possible security, we also use the HTTPS protocol. This way, when an administrator authenticates himself on the login page, the credentials are encrypted. We used a free SSL certificate from the Let's Encrypt [27] organization.

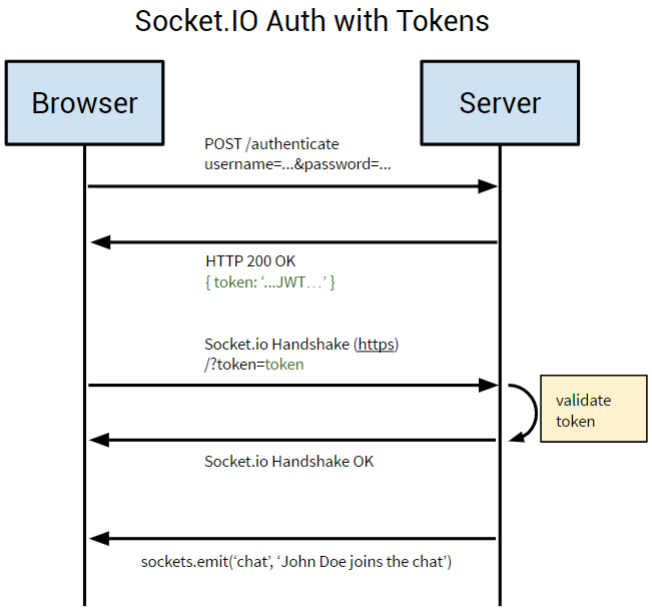


Figure 5.8: Example of Socket.IO Auth with Tokens [28]

Even if the back-end is secure (thus an unauthenticated user can not change anything in the database), we have also set up a security system that prevents unauthenticated users from accessing the administration part of our site. As we have already explained, we divided the

front-end into 2 separate modules:

1. AppModule: Displays the results of runners in real time.
2. AdminModule: The admin part that allows the administration of the race.

Unauthenticated users can only visit the AppModule. If a user wants to access the AdminModule, he must log in with an email and a password. In order to prevent unauthorized users from accessing the administration module, we have secured the route "/admin" with a navigation guard of type *canActivate*. The guard allows us to control the behavior of the route. If it returns true, the process of redirection continues. On the other hand, if it returns false, the process of redirection is canceled. When someone tries to access the "/admin" route, the canActivate function decides if the path can be activated by the current user. This function verifies if the current user is authenticated and, if it is not the case, tries to authenticate the user by using the token that may be stored in the local storage. If these two operations have failed, this means that the user does not have the right to access the admin page and is therefore redirected to the login page.

```
@Injectable()
export class AuthGuard implements CanActivate {
    constructor(private auth: SocketService, private router: Router) { }

    canActivate(): Promise<boolean> {
    return new Promise<boolean>((resolve, reject)=>{
      if(this.auth.isConnected()){
        resolve(true);
      }else if(window.localStorage['feathers-jwt']){
          this.auth.authenticateWithToken()
              .then((result)=>{resolve(true); })
              .catch((error)=>{
                this.router.navigate(['/login']);
                reject(error);
            });
      }else {
        this.router.navigate(['/login']);
        resolve(false);
      }
    });
    }
}
```

Listing 5.19: Example of canActivate guard

5.4.3 The final application

In this section, we present the final version of some pages that we have implemented. If you would like a more detailed overview of each page, we invite you to watch a short video that presents the main features of the system.

`https://www.youtube.com/watch?v=r9FqsRrWIJo`

The figure 5.9 shows the page that displays the results of the race in real time. It displays the name, the team and the time at each checkpoint as well as the average speed of the runner. There is also a filter/search system that allows the runners to easily find their own time and the times of their teammates.

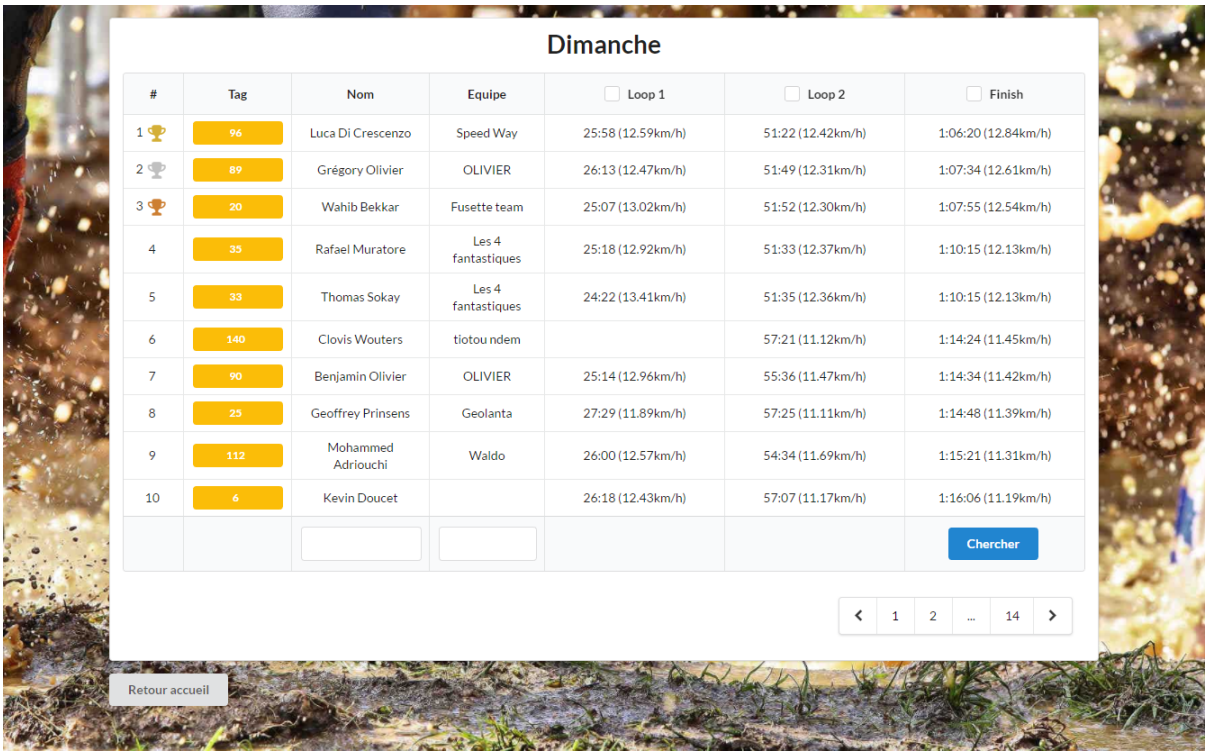


Figure 5.9: The page of the results

The dashboard page (figure 5.10) displays a recap of the race. On this page, we can see the current stage of the creation of the race, the date of the race as well as some statistics. We can also import an Excel file with the runners or a CSV file with the collected times of the checkpoints. Finally, it allows to export a PDF file with the waves, their runners, and the corresponding tags (if any).

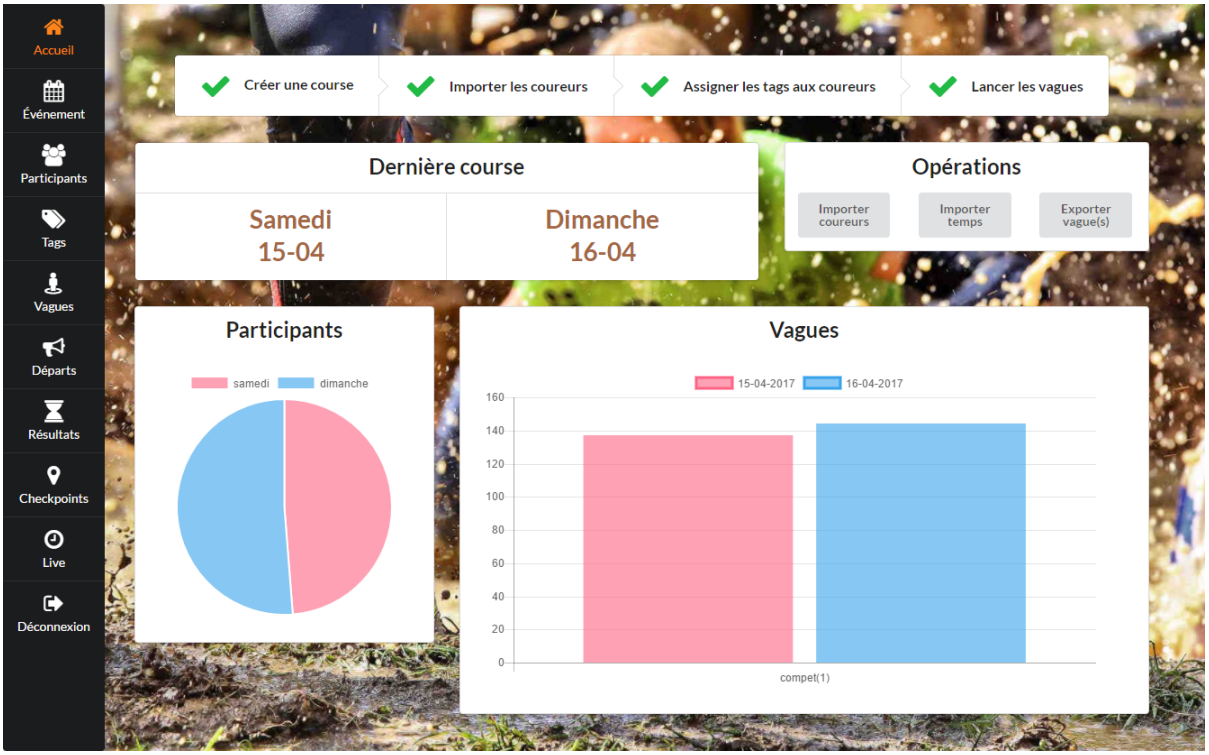


Figure 5.10: The dashboard of admin

The runners' management page (Figure 5.11) allows us to consult the list of runners and to modify them. We can edit all the data of a runner like the name, the team name or even the assigned tag number. We can also filter and sort the runners by name, age, tag, team name or wave.

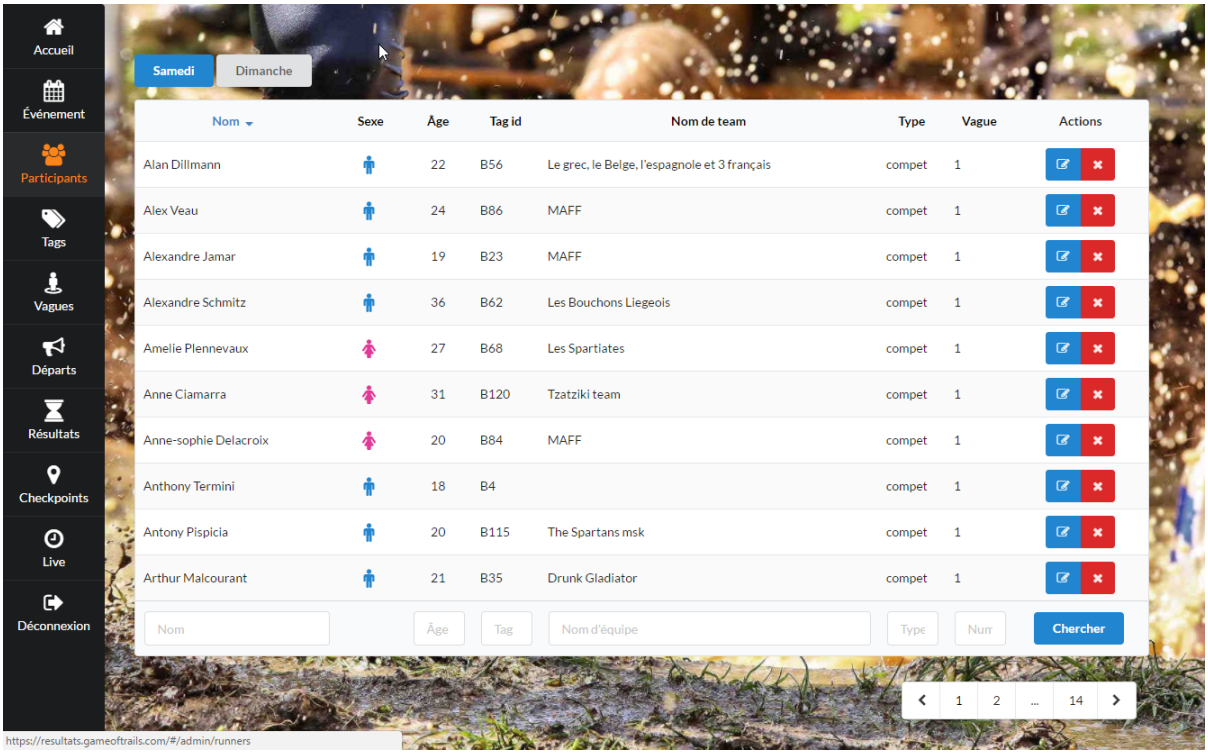


Figure 5.11: Runners management page

Checkpoints' management page (Figure 5.12) allows us to add, modify or delete the checkpoints. It is very useful because it allows us to see in real time the checkpoints that are connected to the back-end. If a checkpoint is not connected, we can see the time of its last connection. We can also see how many people were scanned at each checkpoint.

On this page, we can also manually insert a time in the system by specifying the checkpoint ID, the tag number, the day and the time of passage. Thanks to this functionality, we were able to test our system freely without plugging all the hardware all the time.

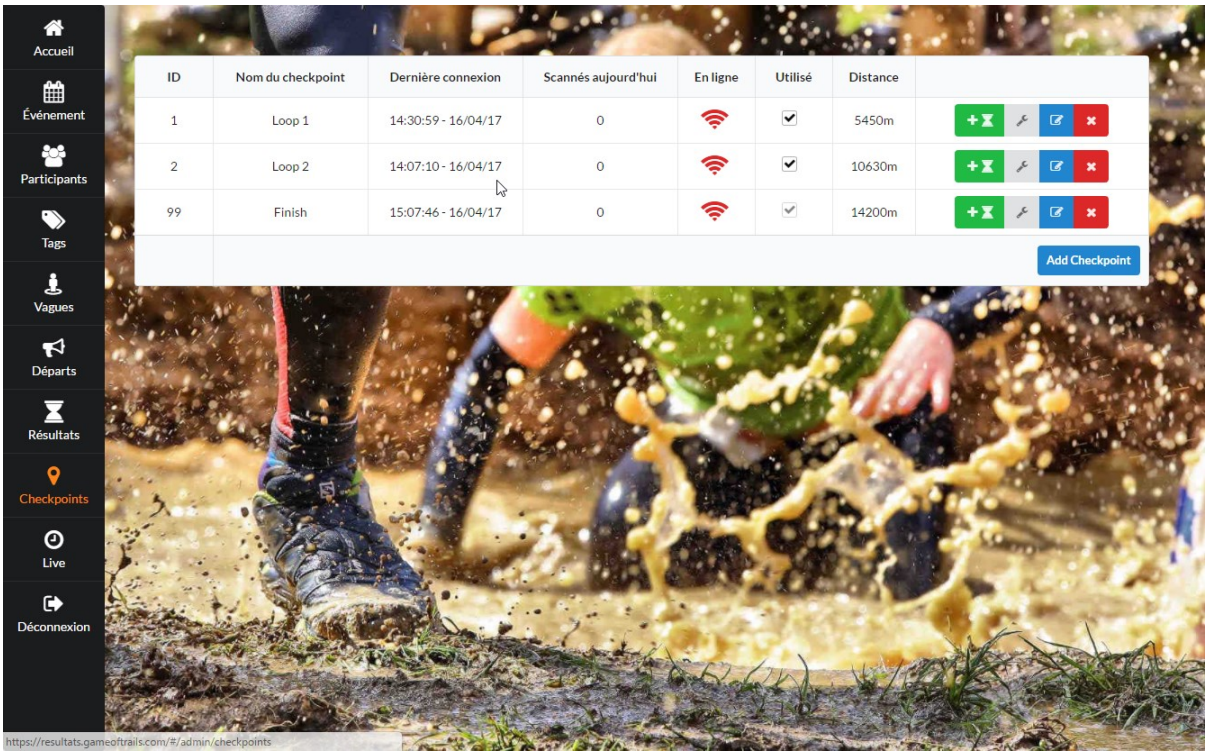


Figure 5.12: Checkpoints management page

5.5 Checkpoint

5.5.1 Hardware bought

As we chose with Game of Trails to use UHF RFID for the detection of the runners at each checkpoint, we had to chose more precisely what were the different devices needed. The first step was to create a prototype to prove that the solution we proposed worked. Afterwards, we had to buy the remaining hardware in order to have a complete system.

Creating a prototype allowed us to focus first on a small subset of the features which are mandatory to make the proof of concept. The prototype was financed by Game of Trails and fortunately could be later used in the final system. Indeed, the proof of concept was a success. It allowed us to show the viability of the project. For Game of Trails, by first doing a small investment for the prototype and checking if it works before financing the rest, they have avoided the risk of investing into something that would not work.

The main part of the hardware bought is for the checkpoints. Each checkpoint needs at least an UHF RFID antenna and a mini-computer to keep the data and send it to the server. We don't take into account here the smartphones used to give access to the Internet by sharing their cellular connection because Game of Trails counted (and still counts) on the volunteers to fulfill that job.

Finally, note that we haven't yet talk about the budget here. As said in the explanation of the requests of Game of Trails, we had a certain amount of money to develop this system. This budget did not allow us to buy most of the parts locally or at least in Europe. Indeed, the prices of the RFID antennas and the RFID tags are way higher in Europe in comparison with the prices in China.

However, due to the high price of the system, there was a big change in the idea of the RFID solution. Game of Trails preferred to have one antenna at each checkpoint and force the runners to pass their tag in front of the antenna. Indeed, they preferred to have three checkpoints scattered along the course of the race, each with one antenna, than only a checkpoint at the finish line with 4 antennas. Note that in the long run, they can have 4 antennas at each checkpoint and remove the constraint of passing the tag in front of the antenna.

Antenna

The cheapest antenna that fulfill our requirements was found on *AliExpress*. Of course, we made sure that the seller is well noted and asked beforehand to get the demo software and the SDK so that we could check that it suited our needs. This antenna is called an "integrated reader" because it includes both an antenna and a reader. Usually, a reader is connected to several antennas and play the role of managing the different inputs. However, in our case we only needed one antenna per checkpoint, therefore it is cheaper and simpler to buy an antenna with an integrated reader. You can see one of those antennas in the figure 5.13.

Figure 5.13: UHF RFID Integrated Reader

This integrated reader has the following specifications:

- **Frequency Range**: 865-868mhz (CE)
- **Protocols**: ISO18000-6B, ISO18000-6C /EPC GEN2
- **Work Mode**: FHSS hopping and fixed frequency
- **RF Power**: 0~30 dBm, adjustable, set by software
- **Antenna**: Built-in 8dbi circularly polarized antenna
- **Read Range**: 6 meters (dependent on tag and environment)
- **Read Mode**: Active, answer, trigger (software programmable)
- **Interfaces**: RS232/UART, RS485, Wiegand26/34

- **Power Supply**: +DC 9V
- **Software**: SDK, DLL and demonstration software provided

Note that this antenna can read and write tags. Therefore, we used it to write the data on the tags. This was done easily thanks to the demo software provided by the vendor.

Raspberry Pi

For the mini-computer used to store and then send the data to the server, we chose to use Raspberry PIs. It's a common choice for projects that need a small amount of computing power and memory in the Internet of Things world. We bought the latest Raspberry PIs at that time which was known as the model 3B. The fact that it has an integrated WiFi antenna is ideal for our idea of using smartphones sharing their cellular connection by acting as a WiFi access point. A picture of a Raspberry PI inside a protective case can be seen in the figure 5.14.

Figure 5.14: Raspberry PI model 3B (within a protective case)

The specifications of the 3B model are as follows:

- 1.2GHz 64-bit quad-core ARMv8 CPU
- 1 GB of RAM
- 4 USB ports
- 40 GPIO pins
- Full HDMI port
- Ethernet port
- 802.11n Wireless LAN

- Bluetooth 4.1
- Bluetooth Low Energy (BLE)
- Micro SD card slot

As you may have noticed, it does not contain any storage memory by itself. It is a micro SD card that has to be used in order to have some kind of storage. We decided to get a special kit particularly designed for the Raspberry PI. This kit contains a micro SD card of 16GB, a power supply (which is by default not provided with the Raspberry PI !) and a protective case.

To know the exact time at startup, the Raspberry PI retrieves it via the NTP Network Time Protocol (NTP). Unfortunately, it's hard to have a network connection everywhere on Earth, so we can not rely solely on this method. To resolve this problem, we decided to also buy a RTC (Real Time Clock) for each Raspberry PI. This small module has a clock that keeps track of the current time even when the Raspberry PI is off.

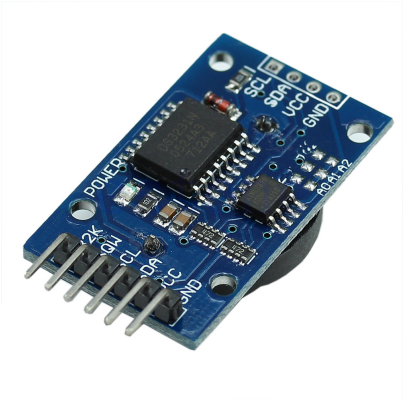


Figure 5.15: Real Time Clock board for Raspberry Pi

Tags

Since we have UHF RFID antennas, we need to have RFID tags that are at the same frequency. We decided for the prototype to buy 50 waterproof UHF tags. You can see an example of such tag in the figure 5.16.

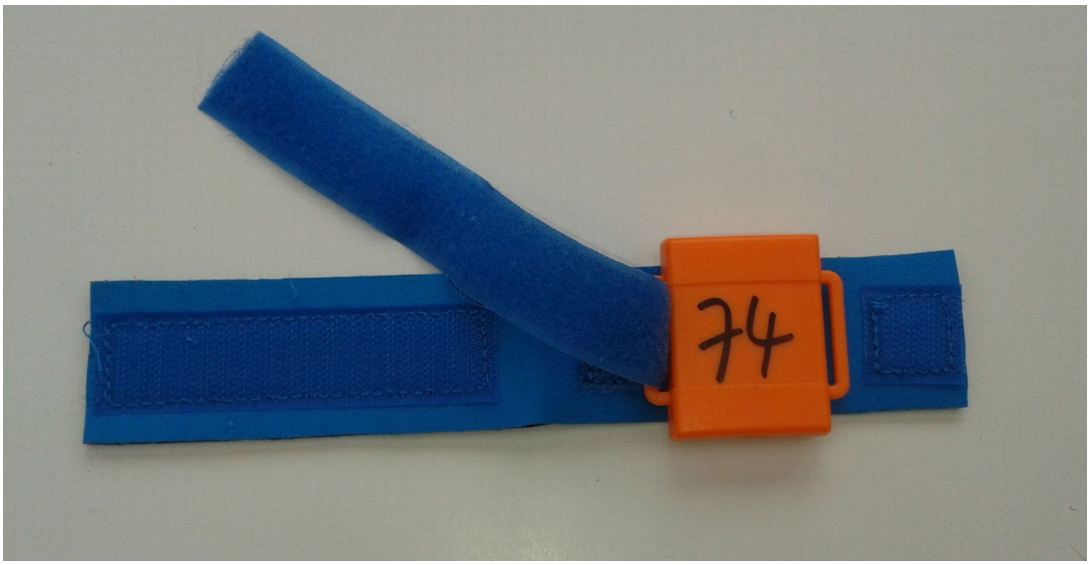


Figure 5.16: Passive UHF RFID Tag

Once the prototype was finished and fully working, we bought 500 more tags for the race. Those were equally divided into blue and orange tags (thus 250 per color). This allowed Game of Trails to have a different color for the tags for each day (blue for Saturday and orange for Sunday).

Summary of the costs

We summarize here the costs related to the hardware of the prototype and afterwards of the whole system. You can see the prototype's hardware and its cost in the table 5.4. It was ordered in October 2016.

Product	**Cost**	**Shipping**	**Remarks**	**Total**
UFH RFID Reader (antenna)	91.29€	53.95€	+27.11€ (duty) -7.45€ (discount)	164.90€
Tags (50)	68.95€	47.32€	/	116.28€
Adapter USB - RS232	10.99€	/	/	10.99€
Total				**292.17€**

Table 5.4: Summary of the costs for the prototype

As already said previously, we were fortunately able to use the hardware of the prototype in the final system. You can find the cost of the rest of the hardware that we bought in the table 5.5.

Product	**Cost**	**Shipping**	**Remarks**	**Total**
UFH RFID Reader (antenna) (x2)	240€	57.59€	+37.45€ (duty) -17.26€ (discount)	317.78€
Tags (500)	666.92€	140.53€	/	807.45€
Adapter USB - RS232 (x2)	21.98€	/	/	21.98€
Raspberry PI + Kit + RTC (x3)	174£	10£	/	221.01€
Total				**1368.22€**

Table 5.5: Summary of the costs for the rest of the hardware

Finally, the total cost of the hardware was 1660.39€. We were therefore a little over the initial budget. However, Game of Trails gave us their consent to raise a little the budget. One mistake from our side was that we forgot to take into account the duty taxes.

5.5.2 Implementation

For the checkpoints, and thus on the Raspberry PIs, we decided to use the *Python* programming language (version 3). It is a common choice for programming on small devices. The fact that it is a dynamically typed language and the existence of *pip*, the Python package manager, are often the main reasons behind this choice besides its ease of use and its good compatibility with the Raspberry PI.

Pyserial

The python library *Pyserial* [29] was our choice to read the input from the antennas. In fact, the data produced by the antenna when detecting a tag is simply outputted on an USB port. Therefore, the application has to use a loop and read at regular interval the USB port, which is where Pyserial becomes handy. You can find an example of use of Pyserial in the listing 5.20

```
import serial
serial = serial.Serial(
    port='/dev/ttyUSB0',
    baudrate=9600,
    parity=serial.PARITY_NONE,
    stopbits=serial.STOPBITS_ONE,
    bytesize=serial.EIGHTBITS,
    timeout=1
)
while True:
    data = serial.readline()
    # Do stuff with the data
    time.sleep(0.01)
```

Listing 5.20: Example of use of Pyserial

As you can see, we first initialize an object named *serial* which represents an USB port. After the initialization, we loop on the read operation. Note therefore that the Pyserial library is not event driven.

Flask

We decided to add a small web server in the Raspberry PIs' program. This web server allows to configure the Raspberry PIs easily by connecting to their local address (usually 192.168.X.X) with a browser. To create this small web server, the python "microframework" *Flask* [30] has been used. An example of use of Flask can be found in the listing 5.21.

The first few lines configure the web server. We provide a name in the constructor and a secret key afterwards. Then, to create a web page, we have to define a function (here *index()* for the index page) that returns the rendered web page. Note the annotation *@config_server.route('/')*

```
from flask import Flask, render_template, request
config_server = Flask(__name__)
config_server.secret_key = 'some_secret'
@config_server.route('/')
def index():
    error = None
    # Do some stuff like loading variables
    return render_template('index.html', a_name=a_variable, error=error)
config_server.run(debug=False, host='0.0.0.0', port=5000)
```

Listing 5.21: Example of use of Flask

that specifies which path the function responds to (therefore here "localhost/"). The web page is rendered thanks to the provided *render_template(...)* function. It takes the path of an html file and the variables that are used in the html file (for example *{{a_name}}* in the html file outputs the value of *a_variable*).Finally, the *config_server.run(...)* function launches the web server. Here we specify that it has to listen to all the interfaces (0.0.0.0) and on the port 5000.

Requests

Finally, to easily send the times to the back-end, we used the python library *Requests* [31]. It removes all the difficulty of sending custom HTTP requests with a JSON body. See the listing 5.22 for an example of use of Requests.

```
import requests
data = {checkpoint_id: 1, tag: 123,
        timestamp: datetime.now().isoformat()}
# The POST request
response = requests.post("https://resultats.gameoftrails.com/times",
    json=data, timeout=3)
```

Listing 5.22: Example of use of Requests

The imported *requests* object directly gives access to functions such as "get", "post", etc. Here, we use the "post" function to send a post request to the back-end. As you can see, the data is stored and then sent as a JSON. Note the use of HTTPS in the url (with a correctly configured certificate on the back-end), it ensures that the data won't be read or changed by an external person (man in the middle attacks). More information are given about this subject in the security chapter.

Structure and features

Now that we have introduced what are the different libraries and frameworks used in our Python program, let's see the structure of it and how it works. We have been particularly careful about the separation of concerns. You can see the structure of the project in the figure 5.17.

The starting point, and thus where the program should be launched from, is *program.py*. It starts the threads of the different parts. The second important file is the *model.py*. As the name suggests, it is our model where we store the variables needed by multiple parts. This model is implemented as a singleton to ensure that there is only one instance at all time. Our program is built as a **producer-consumer** problem.

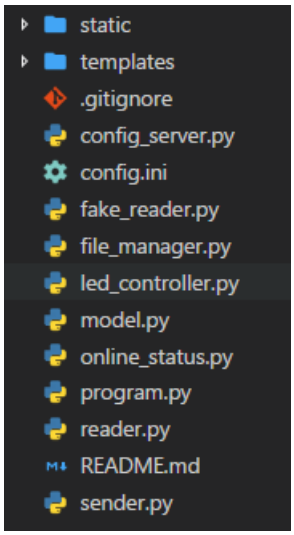


Figure 5.17: Structure of Python program

Indeed, the first main part of the program is the *reader.py*. It has its own thread that does the job of reading in a loop the input coming from the antenna in one of the USB port (see *Pyserial* above). When data have been read, it creates a time (the checkpoint id, the time stamp, the tag id) and stores it in a queue inside *model.py*. We use a *condition* object[11] to ensure that there are no race condition on the queue. This our **producer**. The *fake_reader.py* does the same thing except that it simulates the input of an antenna.

The second main part of the program, the **consumer**, is the *sender.py*. It also has its own thread and it tries to read data from the queue. If there is nothing, it sleeps. Otherwise, it checks that it is connected to the Internet, and if it is the case, it sends the time to the server (see *requests* above) and removes it from the queue. Again, the same *condition* object is used to avoid race condition. Note that if the request fails (connection to the web lost), the time is put again in the queue.

The remaining parts are used to facilitate the configuration and the use of the checkpoints. *file_manager.py* is used by the reader to store all the times in a file (times.data). *online_status.py* has the job of checking the connection to the web and to periodically notify the back-end that the checkpoint is online. The *config_server.py* launches the configuration web server with Flask (see above). The *led_controller.py* periodically turns on the LEDs on the Raspberry PI to indicate that it is connected and that it is sending data.

Now that we have explained the producer, the consumer and the remaining parts, what about the configuration of the checkpoint ? How does it know its ID ? This is done thanks to the *config.ini* file. Inside that file, the checkpoint ID, but also the URL of the back-end and the credentials to get admin access are stored. This is the file that the configuration web server modifies. The program also copies any *config.ini* file existing on a connected USB key which makes it easy to update the configuration without SSH, local network access or a keyboard and a screen for the Raspberry PI.

Note that *program.py* can take 5 possible options when launched:

- **-f n** launches the program with the fake reader (simulates the real reader with random data). It fakes scanning **n** runners.
- **-o** launches the program in offline mode (no online_status and no sender).

[11] A condition object is somehow an enhanced lock. It allows to acquire or release the lock, but it also allows threads to sleep while waiting for an arbitrary condition to become true. Moreover, the thread having currently the lock can notify and wake up the sleeping threads. [32]

- **-b** deletes the previous backup file *times.data*.
- **-s** launches the program in silent mode. That means that it deactivates the textual output.
- **-c** launches the program with the configuration web server.

5.5.3 Redundancy of the data

We try to ensure that the data of the checkpoints is never lost by adding multiple layers of redundancy. First, the times stay in the queue until they are successfully sent to the back-end. That means that if the connection fails when sending some times, they are re-added in the queue. Secondly, all the times that have not been sent are stored in a file called *times.data*. That means that if the Raspberry PI restarts for any reason, it knows what are the times yet to be sent. Lastly, all the times (therefore not only those not yet sent) are stored in an CSV file both in the current folder and on any USB key connected to the Raspberry PI.

Those Excel files containing the times can be uploaded by the administrators on the website whenever needed. When uploaded, the back-end generates the results only for the times that have not yet been received. Indeed, before inserting a time, the back-end run some checks to avoid duplicates. We hope that with all those layers of redundancy, we never lose the data.

5.5.4 Security

To ensure that only valid tags numbers are sent to the server, we perform a check on the Raspberry Pi. First, we check if the scanned tag has the right format. A valid tag number is supposed to contain (see figure 5.18):

1. (2 bytes) Security key 1
2. (1 byte) Day of the race (01 = Saturday, 02 = Sunday)
3. (4 bytes) Tag number
4. (2 bytes) Security key 2

Figure 5.18: Content of an RFID tag

This verification allows us to filter the tags that are not related to the race (eg. clothing labels equipped with a tag RFID). That way, we reduce the server load and the amount of data sent via the 4G network. When the tag number is validated on the Raspberry PI, we send it with a post request to the server via a secure HTTPS connection. In order to insert the data into the database, we have first to authenticate ourselves. Therefore, we send the login data in the same request (more precisely, we send the email, the password and the strategy). See listing 5.23 for an example. Finally, the server checks if a runner with this tag number exists. If all goes well, the runner's time is saved and the corresponding result, if it is the finish line, is generated and displayed on the results page.

```
{
    "timestamp":"1495452328",
    "checkpoint_id":"1",
    "tag":{
        "num":"42",
        "color":"Orange"
    },
    "email":"race@gameoftrails.be",
    "password":"foo",
    "strategy":"local"
}
```

Listing 5.23: Example of data sent to the server

5.6 Tools

To ease the development process, we used several different tools that are well known and often recommended. This is particularly important for us as we are two doing this thesis. We needed to avoid conflicts in the source code but also a place that allows to specify and see who worked on which feature. We won't go into much details as it is not the goal of the thesis.

Git is a version control system. Its goal is to provide a means to keep track of the changes on the source code and any other files of a project. This is the most important tool as it gives us the opportunity to work each on our side and then to merge our changes.

Github is a web-based version control repository. We host our source code on Github and use git (explained above) to commit, push and pull to the different repositories of the race timing system.

Trello is a web-based project management application. Projects are presented by boards. Each board can contain multiple lists, and each list can contain multiple cards (each representing a task). We have one board for each part of the system (except database) and we use the conventional "Backlog", "To do", "Doing" and "Done" tasks lists. We used this service to have a concrete view of what had to be done and what had already been developed.

5.7 Tests

One important part that we have yet to discuss is the tests written for the different parts of our system to ensure that each behave correctly. The back-end and the front-end have been thoroughly tested thanks to frameworks such as *Mocha* and *Protractor*. However, as we will explain in the corresponding subsection, the checkpoints had to be mainly tested by hand due to the "real world" nature of their use. Indeed, we don't have an automated way to physically pass tags in front of the antennas.

5.7.1 Back-end

We used three different packages to write and run tests on the back-end. *Mocha* is the test framework and therefore the package that runs the tests we wrote. It can be considered as the principal package. On the other hand, *Request* and *Chai* are simply packages used to ease the writing of the tests.

Mocha

Mocha [33] is a JavaScript test framework. Mocha runs its tests serially, allowing flexible and accurate reporting. The tests of mocha have ideally to be stored inside a *test* directory in the root of the project.

Mocha has a precise structure to be respected. First, the tests have to be defined inside a call to the **describe(String, Function)** function, which takes a string and a function in parameter. The string is a description while the function is the actual tests' sequence. Note that *describe* can be called inside another *describe*, which gives different levels of tests. Inside the function given in parameter to *describe*, tests are defined by calling the **it(String, Function)** function. Like *describe*, it takes a string which is the description and a function which is the actual test. Look at the listing 5.24 to have a better understanding of this. Tests are run simply by calling "mocha" in the command line.

```
1 var assert = require('assert');
2 describe('Array', function() {
3   describe('#indexOf()', function() {
4     it('should return -1 when the value is not present', function() {
5       assert.equal(-1, [1,2,3].indexOf(4));
6     });
7   });
8 });
```

Listing 5.24: Example of a simple Mocha test

Inside *describe*, a call to *before(Function)* and *after(Function)* can be made anywhere to specify if a specific action has to be made before or after running all the tests inside that sequence.

All the functions given in parameter for the *it*, *before* and *after* can take an argument which is by convention called **done**. This argument is used in the case of asynchronous actions inside a test. In that case, done must called ("done()") to specify to Mocha that the test has finished.

Mocha does not provide an assertion library and therefore lets you chose what assertion library you would like to use. We decided to use *Chai*, which is explained afterwards. You may have noticed the *assert* package in the listing 5.24. It is an assertion library internal to Node.js.

Chai

Chai [34] is an assertion library that simplifies the writing of tests by providing out-of-the-box BDD style assertion constructors. BDD (Behaviour Driven Development) is an extension of TDD (Test Driven Development) which consists of writing tests that are comprehensible to the developers but also to anyone else. To do so, it integrates natural language.

To be more concrete, look at the following two assertions in the listing 5.25. As you can see, the BDD style is more expressive by using natural language "*.to.be.true*", which can be more easily read by anyone.

```
1     /* TDD */
2     assert.equal(true, true);
3     /* BDD */
4     expect(true).to.be.true;
```

Listing 5.25: Comparison TDD and BDD

Let's look at a few more examples of assertions in listing 5.26 to have a better overview of the possibilities offered by *Chai*.

```
1 expect(2).to.equal(2);
2 expect(null).to.be.null;
3 expect(undefined).to.be.undefined;
4 expect(NaN).to.be.NaN;
5 expect([]).to.be.empty;
6 expect('foo').to.be.a('string');
7 expect('foo').to.have.lengthOf(3);
8 expect({a: 1}).to.deep.equal({a: 1});
9 expect({a: 1}).to.have.property('a');
```

Listing 5.26: Chai assertions builder

Example

You can see a shortened version of one of our tests in the listing 5.27. As you can see, we have to import our application (line 4) and launch it in the *before* function (line 11-12). Here the test simply checks that the server is running by retrieving the index page and checking that it is a valid HTML file. Once the test is finished, the *after* function does the job of closing the server.

```
1  const chai = require('chai');
2  const chaiHttp = require('chai-http');
3  const expect =  chai.expect;
4  const app = require('../src/app');
5  chai.use(chaiHttp);
6
7  const URL = "http://"+app.settings.host+":"+app.settings.port;
8
9  describe('Feathers application tests', () => {
10   before(function(done) {
11     this.server = app.listen(3030);
12     this.server.once('listening', () => done());
13   });
14
15   after(function(done) {
16     this.server.close(()=>{
17       done();
18     });
19   });
20
21   it('starts and shows the index page', done => {
22     chai.request(URL).get('/').end((err, res) => {
23       expect(res.text.indexOf('<body>')).to.not.equal(-1);
24       done();
25     });
26   });
27  });
```

Listing 5.27: Shortened version of one of our Mocha tests

Coverage

Our tests on the back-end using Mocha cover nearly all the functions (*create*, *get*, ...) of all the services. Only the uncommon services such as the import of the Excel file and the export of the PDF file are not automatically tested by our tests as it requires more complicated interactions with the back-end.

Load testing

An important part of the testing for a back-end is the load testing. The goal is to test if it can handle heavy load while still being efficient and resilient. That means that the server should work efficiently even if there is a large number of concurrent requests. Of course, any back-end (and more precisely the server hosting it) has a limit depending on the CPU, the memory and the network connection.

To test our back-end and therefore our server, we used a global package for Node.js called **Artillery** [35]. It describes itself as a modern, powerful and easy-to-use load testing toolkit. We didn't make complicated test for this part, we just wanted to see if it could handle a sufficiently large number of requests for our case.

To install a global package, the option "-g" has to be specified.

```
npm install -g artillery
```

We can launch a quick test with the following command:

```
artillery quick --duration 60 --rate 10 -n 20 http://url/resource
```

This command creates 10 virtual users per second for 60 seconds. Each virtual user send 20 *GET* request to the URL specified.

In our case, we tested with 100 new users each second for 60 seconds. Each user sent only one request. At the end, we got a median latency of 323.5 milliseconds, which is totally acceptable. Note also that all the requests received a response (we received 6000 times the HTML "ok code" 200). See the more detailed results in the listing 5.28. In any case, 100 new users every second is already a huge number considering that there are at maximum 300 *Elite* runners.

```
1   Scenarios launched:  6000
2   Scenarios completed: 6000
3   Requests completed:  6000
4   RPS sent: 99.11
5   Request latency:
6     min: 71
7     max: 1012.4
8     median: 323.5
9     p95: 740.9
10    p99: 872.7
11  Scenario duration:
12    min: 71.9
13    max: 1020.8
14    median: 326
15    p95: 742.8
16    p99: 874.3
17  Scenario counts:
18    0: 6000 (100%)
19  Codes:
20    200: 6000
```

Listing 5.28: Results of our load test

5.7.2 Front-end

To test our front-end application we decided to implement end-to-end tests. We used 2 libraries, **Protractor [36]** and **Jasmine [37]**. End-to-end tests allow us to test our application from the point of view of the end user. This kind of testing lets us test the flow of the application from the beginning (front-end) to the end (back-end) and to ensure that everything is working as expected.

To start the tests, you must place yourself in the root folder of the front-end project and run the command **"ng e2e"**. Before launching the tests, you must not forget to launch the back-end.

Protractor

Protractor is a free tool built on top of Selenium [38] which allows to test Angular applications in a real web browser and interacting with it as an actual user would. The main role of Protractor is to identify the DOM elements on the web page thanks to the following selectors [39]:

- **by.css('.myclass')** - Find an element using a css selector
- **by.id('#myid')** - Find an element with the given id
- **by.model('name')** - Find an element with a certain ng-model
- **by.binding('bindingname')** - Find an element bound to the given variable

The second role of Protractor is to execute the browser level commands. It allows operations such as clicks, texts input, URL navigation, etc.

The listing 5.29 contains an example of a test written with the Protractor framework. First, we used the selector **by.css()** to find the login and password input fields. Secondly, we used the **sendKeys()** function to insert text into those fields. Then, we ask Protractor to press the send button with the **click()** function. After a delay of 500ms, we check if the page has changed with the help of the following Jasmine test (for example: **except(value 1).toEqual(value 2)**).

```
1  it('should login successfully', () => {
2    element(by.css('.email')).sendKeys('name@mail.com');
3    element(by.css('.password')).sendKeys('foo');
4    element(by.css('form button')).click();
5    browser.sleep(500)
6    expect(browser.getCurrentUrl()).toEqual('http://localhost:4200/#/admin/
         dashboard');
7  });
```

Listing 5.29: Example of protractor test

Jasmine

Jasmine is a BDD (Behavior Driven Development) framework for testing applications created using the JavaScript language. The tests in Jasmine have a similar structure to the tests in Mocha explained in the back-end tests' section. As in Mocha, the tests must be defined in a **describe(String, Function)** function. The string describes the group of tests and the function given in parameter is used to contain the tests created with the function **it(String, Function)** [40].

The main function of Jasmine is to provide the **expect** function and the **matchers** [41]. Matcher is a function called in the expect returned object and allows you to define an assertion,

thanks to which we can test the expected value.

By default, jasmine provides a multitude of matchers [42] :

- **toBe** - expect the actual value to be === to the expected value
- **toEqual** - the actual value to be equal to the expected, using deep equality comparison
- **toMatch** - the actual value to match a regular expression
- The description of other matchers is available on the official web page

In addition, Jasmine gives us with the ability to invert the result using **not** before the matcher.

```
1         expect(false).not.toBe(true);
```

Listing 5.30: Example of inverted jasmine test

5.7.3 Checkpoints

As said in the introduction of this section, we couldn't really automate some kind of tests during the development of the checkpoints. The main features of the checkpoints are scanning tags with an antenna and sending the times to the back-end. In the first case, it needs to interact with the antenna and the tags and thus with the physical world. In the second case, it could be possible by checking if the time has successfully been uploaded on the back-end, but the back-end already verifies that a time sent through a *POST* request is successfully saved in the database.

Nevertheless, we did a lot of manual tests on our own. We installed multiple times the system in a field and always tried several cases such as when the internet is not available and becomes available after a short while. We also checked that the tags worked even with mud on them. Another important test was to check the maximum speed at which the runners can pass in front of the antennas. The vendor of the antenna claims that it works as fast as 80 km/h. Of course, we didn't have the possibility to test at that speed, but the antenna successfully detected tags tossed rapidly in front of it.

5.8 Problems encountered

Like in any other project, some problems occurred during the requirements and more principally during the development of the race timing system. Fortunately, those problems were not insurmountable.

Firebase limited possibilities

As already explained throughout the thesis, we had some difficulties with Firebase. We don't explain again here what were the problems as it has already been explained in the database section.

Lack of documentation for Feathers

The lack of documentation for Feathers was an important point that slowed us down during the development phase. Indeed, the team developing and maintaining Feathers introduced some breaking changes and did not update the documentation. We had to search in the change logs

published on their Github repositories and dive directly inside the source code of the framework to understand how some features had to be used. Hopefully, the documentation has been totally updated one month after those breaking changes.

NoSQL drawbacks

Another problem that grew during the development is related to our choice of a NoSQL database (Firebase at the beginning, MongoDB afterwards). NoSQL databases are nice for fast prototyping and fast queries because they don't enforce some kind of "schemas". The developers can insert and query the database right away without thinking about data models. Moreover, the queries are fast because there are no join. Each query only runs on one collection at a time.

However, NoSQL also has drawbacks that were more strong that what we thought. The main one is related to the consistency of the data across the collections. As there are no join, the data has to be repeated in multiple places if we need that data in all those places. This leads to a lot of redundancy which can't be avoided because it is all the essence itself of NoSQL to have redundant data. Indeed, that allows queries to be fast. However, that also means that when we update a piece of data somewhere, we also have to update the redundant data in other collections, otherwise the consistency is lost. In NoSQL, it is the developer's job to ensure that the consistency is maintained in the data at anytime. On the other hand, relational databases do this job for you (usually there are no redundant data, at least with well structured data models).

Chapter 6

The race

The race that occurred during the 15th and the 16th April 2017 in Sart Tilman was the culminating point of our thesis. It was the most important step where everything we developed and bought was going to be used in real conditions. Therefore, it meant that the 14th was, for us, synonym of deadline. Indeed, the development but also the final testing of our race timing system had to be finished before the race.

6.1 Preparations

Several days before the race, we decided to go to Sart Tilman to see with our own eyes where the mud race was going to take place. We wanted to see exactly where the checkpoints were going to be positioned along the course of the race but also what were the possibilities for us to install our hardware. We also gave the RFID tags to Game of Trails during that visit so that they could distribute them to the runners Saturday and Sunday morning. You can see the map of the race in the figure 6.1.

Figure 6.1: Map of the race (Sart Tilman 2017)

The first checkpoint was at the border of a cliff along a 2 meters wide walking path. The strength of the signal from the cellular network (thus 4G) was good and the fact that it wasn't too wide helped for the detection of the runners. It is situated at a distance of 5.450 kilometers right after the 6th obstacle at the first supply point (with food and drinks), which is indicated by an "R" on the map.

The second checkpoint was in a big plain near constructions for new habitations. It was next to the "mud zone", which deserves its name well. It was more difficult to imagine how everything would be during the race in that zone as it was just a big plain when we first came. The signal of the cellular network was also good there. This second checkpoint is situated at a distance of 10.630 kilometers from the start. The mud zone can be found as the 13th obstacle on the map.

The final checkpoint and therefore the finish line for the runners is in fact next to the start. Here, we knew that there would be a lot of barriers and thus that we could arrange them in the most efficient way for our system. The signal for the 4G was excellent as it was right next to a gym and therefore not inside the woods. The total distance of the race is 14.200 kilometers and the finish line is located right after the last obstacle (20th).

6.2 During the race

During the weekend of the actual race, we came each day at 9 o'clock in the morning to install the hardware at the different checkpoints. Of course, we couldn't let the hardware stay during the night, we had to uninstall and re-install the next day. The race started at 12 o'clock, so we had 3 hours to prepare the 3 checkpoints. Fortunately, the distribution of the tags were managed by Game of Trails themselves as they had already the tags in their possession. Note that a power generator and an arbour to protect the Raspberry PIs were provided by Game of Trails for each checkpoint. Each antenna was installed by hooking it to a barrier. Note that each antenna had a fixation provided.

Figure 6.2: Installation of the antenna at the first checkpoint

Figure 6.3: Installation of the antenna at the finish line

It was planned that 144 *Elite* runners would come Saturday and 137 *Elite* runners would come Sunday. One point to note is that there were 3 checkpoints and we were only two persons to manage them. Therefore, we had to split ourselves in order to have one person at the first checkpoint and one person at the finish line. The second checkpoint was managed by a volunteer of Game of Trails.

At 12 o'clock, one of us (the one who was staying at the finish line) had to go to the start line to see at what time the runners from the *Elite* wave started. Indeed, as in any event, the runners didn't start at 12 o'clock exactly. Therefore, it is an important step in our application to specify to the system at what time the runners started.

During the race of Saturday, a problem occurred. The problem was that the volunteer at the second checkpoint seems to not have done a good job at telling the runners to pass their tag in front of the antenna. A lot of runners had no time for the second checkpoint Saturday. Sunday morning, fortunately, we had the occasion to explain more thoroughly to the volunteer what he had to do.

Except for this problem, everything went well and the results appeared in real-time on the website[1].

6.3 Statistics

In this section we show different statistics that we collected. For this purpose we have used Google Analytics [43], it's a free audience analysis service that provides a multitude of statistics.

That being said, the current data that we possess are not representative of the actual number of visitors that we could have had from the first day of the race because the integration of Google Analytics was made only on Saturday evening.

[1]The results can be seen on `https://resultats.gameoftrails.com`

Among the different statistics that have been available, we chose two of the most interesting ones which are the number of unique visitors as well as the geographical location. Data analysis is done over a week.

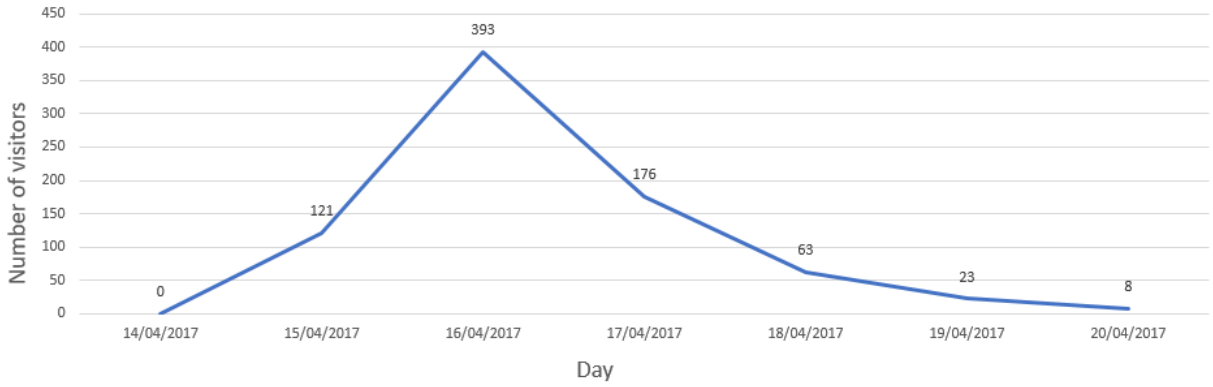


Figure 6.4: Number of unique visitors per day

The figure 6.4 shows the number of unique visitors per day on the website on which the results are available (`https://resultats.gameoftrails.com/#/`).

The integration of Google Analytics has been done on April 15, 2017 in the evening, in just a few hours we can see that the number of visitors has grown fast. On the 16th April, we can see a significant increase in the number of visitors in a single day. The next day, the evolution of the graph is normal. There are fewer visitors because the race has been over for more than 24 hours. Then, the line follows a decreasing curve in relation to the passing days.

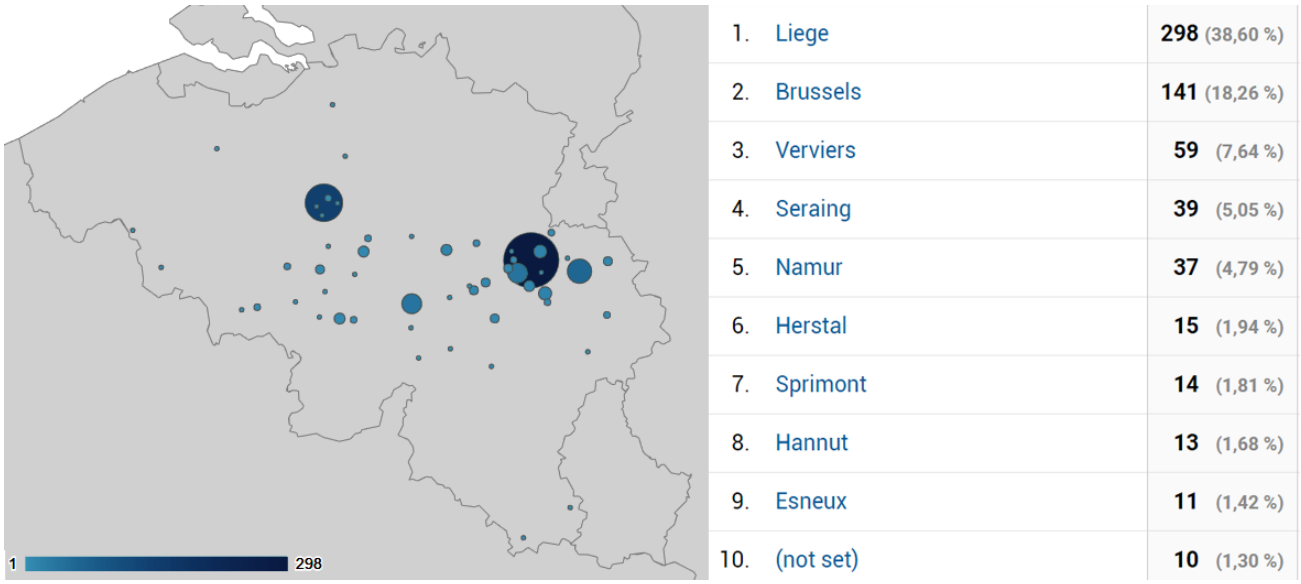


Figure 6.5: Geographic summary

On the second graph (see figure 6.5), we can see the geographical distribution of the visitors. We can assume that these data represent the residence of participants. As we can guess most visitors come from around Liège because the race was organized in this region.

We also carried out some statistics on the results of the race. On the figure 6.6, we can see the average speed of runners. The calculation is based on the distance between the start and the

finish line that was delivered by Game Of Trails and the time that was required to complete the race.

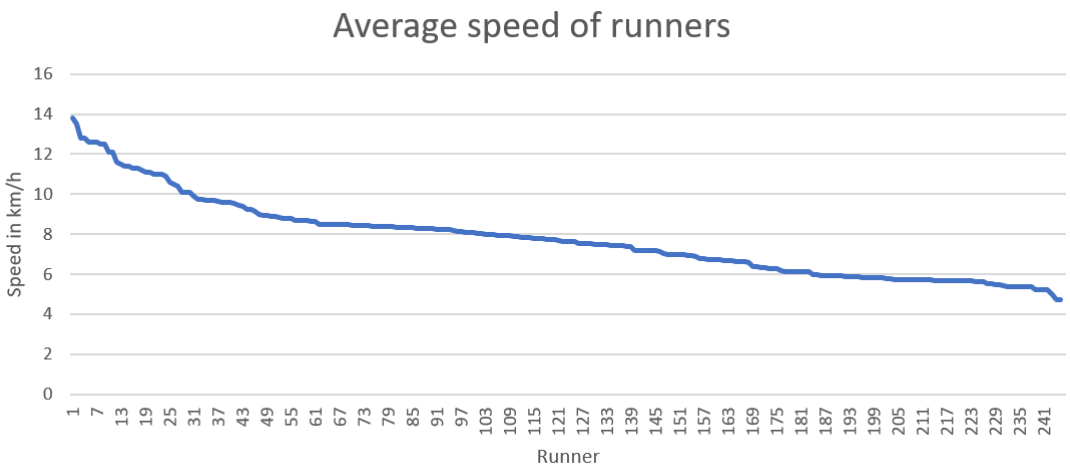


Figure 6.6: Avarange speed of runners

In the figure 6.7, we can see the time that participants took to complete the race. As you can see, the best runners were almost 3 times faster than the last runners.

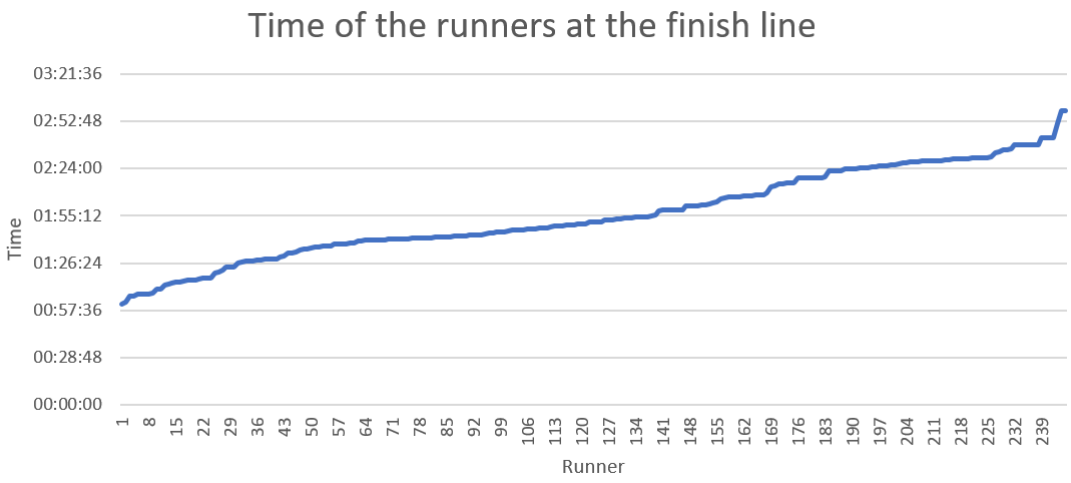


Figure 6.7: Time of the runners at the finish line

6.4 Feedbacks

Game of Trails was happy of how everything went. The results of the *Elite* runners could be seen in real-time on the website and Game of Trails has been able to organize at 4 o'clock pm its "podium" with rewards for the best runners during both days. An improvement that could be useful according to them is to have the gender displayed in the results' page (with the possibility to filter on it).

As for the future, they told us that they wish to continue their partnership with us. That implies, among others, managing the race timing system during the next race in September. However, we think that the system should be ideally usable by anyone without any computer science background. But for now, the system is far too complex. We will talk about that more thoroughly in the next section.

Chapter 7

Possible improvements

Our race timing system is only the first version of probably a lot more to come. This first version can be improved in multiple ways, such as improving the performances or adding more features. The actual race that occurred on the 15th and 16th April, as presented in the previous chapter, showed us that some parts could be improved. Here is a list of the enhancements we think should or could be made.

Gender displayed on results' page

Game of Trails wish that the gender is displayed on the results' page and that a filter can be applied on it. They need this in order to find who are the 3 best male and female runners when giving the rewards.

Page to manage the administrators

Currently, the administrators have to be created by adding them manually in the database or by modifying the source code of the back-end. In both cases, it is not a simple procedure. A page allowing to manage the administrators could be handy.

QR Code and/or NFC smartphone application

The system is currently designed to manage races with the help of RFID devices. However, the front-end and the back-end need only a few changes to be able to be linked to other technologies such as QR Code or NFC. Indeed, the RFID solution is nice but it has a high cost that can't be afforded by everyone. We could develop a smartphone application for the use of QR Code or NFC which would allow anyone to use the system. They would only need to pay for a small server, which fortunately becomes cheaper and cheaper.

Change to a relation database

As said throughout this dissertation, the choice of a relational database instead of a NoSQL database might have been wiser. Because we were interested and already committed to the fast real-time database, Firebase, we accepted the use of NoSQL without further thoughts. However, a relational database may make the system easier to maintain in the long run. This is due to the fact that NoSQL databases use redundancy to allow fast queries without the need of joins. The consistency of all the redundant data is left to the developer, which can easily lead to errors.

Encryption of the data on tags

Currently the data stored on the tags is not encrypted. We have a special format which ensures that we take into account only the tags belonging to the race. However, if a malicious

person scan one of the tags, he will find out what is the format and may introduce his own tags with our special format into the race. Encrypting the data on the tags could resolve this problem.

Websocket connection between checkpoints and back-end

The connection between the checkpoints and the back-end is currently not a Websocket connection. Indeed, the checkpoints send a new HTTP request for each new time to upload. Moreover, they send HTTP requests periodically to indicate that they are online. All this could be more intuitive and more efficient with the use of Websockets. If the connection between a checkpoint and the back-end is alive, the checkpoint is connected. When the connection is abruptly stopped, the state can be changed to offline. Of course, this Websocket connection should also be used to send the times.

Ease the use of the system by someone with no computer science background

One important point that we already mentioned in the previous section is the fact that the system can only be used by someone who understands it well. This is due to multiple reasons: first, the data on the tags have to be written, which is not an easy operation. Secondly, the configuration of the checkpoints is also an important point difficult for someone not knowledgeable. Once those two operations are done, the system is easily usable only with the website. Therefore, we may need to find a way to simplify those two operations.

Chapter 8

Conclusion

To conclude, we explained thoroughly the process behind the conception and the implementation of our race timing system. From the introduction of Game of Trails to the actual race that occurred the 15th and 16th April, a lot of decisions had to be made such as choosing the technology to detect the runners at the checkpoints and choosing the frameworks to be used to build the different parts of the system. It wasn't easy but we think that we successfully developed a race timing system for Game of Trails. Of course, there were some problems along the road, as in any other project, but we resolved them as they came. Our race timing system is far from being perfect. A lot of improvements can be made and we already think of implementing some of them.

In the improvements that can be made, the development of a smartphone application for the QR Code and/or the NFC could give new horizons to this race timing system. Indeed, with a smartphone app acting as an "antenna", the cost of the system decreases a lot as the hardware needed is less expensive (or already there). Therefore, it could be used in events that do not have such constraints as being able to resist to mud. Moreover, the QR Code smartphone application is the easiest solution and could be easily used by teachers or in small events. Only a smartphone, a printer and a small server are needed.

However, as already said, some of our choices are questionable. Particularly the choice of using NoSQL, we somehow took the easy option and we didn't think that it would be so complex to maintain the consistency of the data by ourselves.

In all cases, this thesis may be the beginning of an adventure for us. We learned a lot of things including new frameworks and libraries that we didn't already know. Moreover, working with a real client was an unique experience that we enjoyed as it gave us a real objective.

Bibliography

[1] J. Luu, "The anatomy of a qr code." http://axiomcreative.blogspot.be/2012/01/anatomy-of-qr-code-7-things-to-know.html, January 2012.

[2] "Qr code - wikipedia." https://en.wikipedia.org/wiki/QR_code.

[3] "Qr code error correction." https://blog.qrstuff.com/2011/12/14/qr-code-error-correction, December 2011.

[4] "Garmin - what is gps?." http://www8.garmin.com/aboutGPS/.

[5] "Qu'est-ce que la nfc?." http://www.identivenfc.com/fr/what-is-nfc.

[6] "What is rfid?." http://www.technovelgy.com/ct/technology-article.asp.

[7] "Les gammes de fréquences rfid." http://www.centrenational-rfid.com/les-gammes-de-frequences-rfid-article-16-fr-ruid-17.html.

[8] J. Thrasher, "Rfid vs. nfc: What's the difference?." http://blog.atlasrfidstore.com/rfid-vs-nfc, October 2013.

[9] "Rain rfid guide." https://www.impinj.com/about-rfid/.

[10] S. Smiley, "Active rfid vs. passive rfid: What's the difference?." http://blog.atlasrfidstore.com/active-rfid-vs-passive-rfid, March 2016.

[11] "Firebase." https://firebase.google.com/.

[12] R. Khanna, "Structuring your firebase data correctly for a complex app." https://www.airpair.com/firebase/posts/structuring-your-firebase-data, May 2015.

[13] "Firebase uses websocket or sse." https://groups.google.com/forum/#!topic/firebase-talk/lD91N6Zalfw, November 2016.

[14] I. MongoDB, "Nosql databases explained | mongodb." https://www.mongodb.com/nosql-explained.

[15] M. D. Network, "Arrow functions - javascript - mdn." https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions, May 2017.

[16] N. Foundation, "Express - node.js web application framework." https://expressjs.com/, 2017.

[17] S. LLC, "xlsx." https://www.npmjs.com/package/xlsx.

[18] M. Eernisse, "Ejs – embedded javascript templates." http://ejs.co/, September 2016.

[19] A. Raminfar, "phantom." https://www.npmjs.com/package/phantom.

[20] "Angular 2." https://angular.io/.

[21] "Typescript." https://www.typescriptlang.org/.

[22] "Semantic ui." https://semantic-ui.com/.

[23] "Angular-cli." https://cli.angular.io/.

[24] "Angular 2 - tutorial on components." https://angular.io/docs/ts/latest/tutorial/toh-pt3.html.

[25] "Feathersjs client." https://github.com/feathersjs/feathers-client.

[26] "Using subjects rxjs subject." https://xgrommx.github.io/rx-book/content/getting_started_with_rxjs/subjects.html.

[27] "Let's encrypt." https://letsencrypt.org.

[28] J. F. Romaniello, "Token-based authentication with socket.io." https://auth0.com/blog/auth-with-socket-io/, January 2014.

[29] C. Liechti, "Welcome to pyserial's documentation - pyserial 3.3 documentation." https://pyserial.readthedocs.io/en/latest/index.html, May 2016.

[30] A. Ronacher, "Welcome | flask (a python microframework)." http://flask.pocoo.org, May 2017.

[31] K. Reitz, "Requests: Http for humans - requests 2.14.2 documentation." http://docs.python-requests.org/en/master/, May 2017.

[32] P. S. Foundation, "17.1. threading - thread-based parallelism - python 3.6.1 documentation." https://docs.python.org/3/library/threading.html#condition-objects, March 2017.

[33] "Mocha - the fun, simple, flexible javascript test framework." https://mochajs.org/, January 2017.

[34] "Chai." http://chaijs.com/, January 2017.

[35] "Artillery, a modern load testing toolkit." https://artillery.io/.

[36] "Protractor, end to end testing for angularjs." http://www.protractortest.org/#/.

[37] "Jasmine - behavior-driven javascript." https://jasmine.github.io/.

[38] "Selenium - browser automation." http://www.seleniumhq.org/.

[39] "Protractor - using locators." https://github.com/angular/protractor/blob/master/docs/locators.md.

[40] D. Tang, "Rfid vs. nfc: What's the difference?." https://www.codementor.io/javascript/tutorial/javascript-testing-framework-comparison-jasmine-vs-mocha, January 2016.

[41] "Jasmine - introductions." https://jasmine.github.io/2.0/introduction.html.

[42] "Jasmine - matchers." https://jasmine.github.io/api/2.6/matchers.html.

[43] "Google analytics." https://www.google.com/analytics.

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