Design report
Project eTROM

Author:
Lieke Turenhout
Max Jeltes
Niels Kruk
Ruben de Koning
Sybe de Oude

Academic supervisor:
Tom van Dijk

Clients:
Sarah Onrust
Lotte Steenmeijer

Department of Computer Science

November 8, 2023

Abstract

This report gives a technical insight into the development of a fail-resistant, server-based solution to handle real-time data for the Batavierenrace. The Batavierenrace is the world's largest relay race with over 8500 runners participating each year and is completely organised by students from the University of Twente and Saxion Enschede. This project is undertaken as part of the "Design Project" module of the bachelor Technical Computer Science at the University of Twente.

As of now, the data management system that takes care of all data generated by the runners and volunteers was created in 2003 and last updated in 2014. Being a Windows application that runs locally and requires someone to be physically behind a computer to ensure everything runs smoothly. The eBART committee of the Batavierenrace asked us to redesign the so-called "eBPC" and create a server-based solution that can be accessed from anywhere by a select group of organisers. This new system must ensure that no data is lost during a race, since stages of the relay race cannot be redone by the participants.

The new system, which will carry the name "eTROM: enhanced Time Registration Optimised for Monitoring", is a complex system starting with a message-server containing all logic for the data flows and ensures the data is sent to the right places in the right format. To gain insight into all this data processed by the message-server, we developed a web-application consisting of a React front-end and a Spring Boot back-end connected to a PostgreSQL database. The technical details of these four main components are discussed in this report together with the test plan used and an overview of all stakeholders, abbreviations and requirements. At the end of the report, it was made clear which requirements the final product was able to meet and which things could be improved in the future.

Contents

1. Introduction

The Batavierenrace (Bata) is the world's largest relay race, where thousands of students run 175 kilometres from Nijmegen to Enschede in 25 stages every year. The stages are divided into a night, morning and afternoon shift. The first nine stages form the night shift, which runs from Nijmegen to Ulft. In Ulft there is a restart after which the next eight runners run towards Barchem. After this, the last segment heads towards Enschede, where the final restart occurs for the last two stages. The restarts are used to reduce the distance between the runners and to ensure that the entire route does not have to be closed off for traffic throughout the entire race.

To record the times of all runners, the Bata uses a self-developed time measurement system. All runners wear a vest equipped with an RFID tag that is read by the registration cabinet (RK) at each exchange point. The times are transmitted via RabbitMQ servers to the central computer in Enschede (eBPC). From here the stage intermediate time is calculated and sent back to the RK via RabbitMQ. The eBPC also forwards the information to the race secretary who keeps an overview of the race and is responsible for the validity of the results. Finally, the eBPC shares the data with the public website of the Bata.

Currently, the eBPC is running locally on a Windows laptop. This application has to be started manually every year and someone has to stay with it to monitor it. In addition, the data is only viewable on the screen itself and cannot be viewed by other people from the organisation during the race. Also, the application contains a lot of legacy code considering it was created around 2003. Therefore, it would be good to recreate the application and use current technologies and standards.

The project will consist of replacing the eBPC for a web application. In addition to displaying the available data, the web app will need to continue to support the current situation by receiving data from all devices positioned in the race and by forwarding the data in the correct format to the race secretariat and the Bata public website. In addition to this, the new application should support the existing communication protocol through all the different formats of messages. By developing a web-app, we aim to enhance its functionality, accessibility and user-experience. In addition, the aim of this project is to scrutinise the existing eBPC application and see if improvements can be made to it. This could be both in the internal code and the way certain connections are made. Or, for example, in the visual representation of the application.

2. Terminology and stakeholders

This section gives an overview of all abbreviations used during the Batavierenrace by either the eBART committee or the board of the Batavierenrace to establish a clear definition of all abbreviations, acronyms and definitions used. Furthermore, an overview of the different stakeholders is provided.

2.1 Abbreviations, acronyms and definitions

The board and committees of the Batavierenrace utilise various abbreviations and acronyms for ease of communication. These are also applied throughout the time registration system. To guarantee a common understanding, the following list provides explanations for these abbreviations and acronyms.

Abbreviation/ Acronym	Meaning	Explanation
AK	Accukist (battery box)	Large battery used at a WP to power several devices.
Bata	Batavierenrace	The world's largest relay race with over 8500 runners.
BataID	Batavierenrace Identifier	The BataID identifies which Batavierenrace component created a log entry. Various components can create logs, such as the different WP's, the eBPC or the WS.
BE	Beeper	Device which makes a loud noise and bright light when a runners tag is scanned by the RK at a WP.
BPW	Bata public website	Public website of the bata accessible worldwide. During the race, times of the runners can be viewed here once they are made official.
CC	Commando centrum	eBART headquarters during the race. From here all WPs are managed and problems solved during the race.
CDB	Central database	Main database used by the bata to store all race data. Multiple components forward their data to this DB during the race.
DB	Database	-
DI	Display	Display used at each WP to view race times, team information and the current time.
eBART	extended Batavierenrace Automatisch Registrerend Tijdwaarnemingssysteem	All parts responsible for the time registration system, such as the eBPC, RK's and RFID tags. The eBART committee is responsible for the functionality of this

		system.
eBPC	eBART Personal Computer	The central computer in Enschede that used to run the data management application.
eTROM	enhanced Time Registration Optimised for Monitoring	The new system that replaces the eBPC.
KK	Kabelkist (cable box)	Used by a WP to power several devices and connect them to each other.
KP	Keypad	Device used to enter the number of the runner that is approaching the WP. This number is shown on the DI and seen by the next runner of the team so he/she can be prepared to take over.
LB	Log (Logbook)	Used to save all actions of WPs during the race.
LBI	Logboekitem (Log book item)	Data type used during the race to communicate from the WS to the eBPC and all RKs.
LBR	Logboekregel (Log book line)	Data type used during the race to communicate from a RK to eBPC.
PDA	Personal Digital Assistant	Small hand-computer used at a WP to add notes or penalties to the system. This system is being replaced by a tablet.
RFID tag	Radio Frequency Identification tag	Identification tag inside the vest of a runner used to register stage times.
RK	Registration Cabinet (Registratie Kast)	Hardware component containing a Raspberry Pi and RFID scanner which is used at each WP to scan runner tags and forward the data to the eBPC via 4G.
RMQ	Rabbit MQ	Message queueing software used in the bata to buffer messages between several applications.
TA	Tablet	Tablet used to display the RK web-interface. See WI for more info.
UC	Universiteitscompetetie (university competition)	All university teams taking part in the race.
WI	Web interface	Website running on a tablet at each WP to monitor the RK and add notes or penalties for certain teams.
WL	Wedstrijdleiding	Committee in the bata responsible for the safety of all

	(race control)	runners and volunteers.
WP	Wisselpunt (waypoint)	Each stage has a checkpoint at which the runner his tag is scanned and registered to save a stage time.
WPP	Wisselpunt ploeg (waypoint team)	Group of about 8 volunteers responsible for managing one WP.
WS	Wedstrijdsecretariaat (contest secretariat)	Committee in the bata responsible for validating run times and handing out any penalties.

Table 2.1.1: Abbreviations/Acronyms with their meaning and an explanation

2.2 Stakeholders

During the Batavierenrace, multiple end-users work with the time registration system. The primary stakeholders who interact with the application most extensively are the eBPC administrators from the eBART committee. Their responsibility is to maintain an overview of all the runners, waypoints and devices throughout the entire race, using the application. Therefore, it is essential that the application is clear and well-organised to ensure optimal user experience for the administrators when working with the system.

In addition to the eBPC administrators, there are also several indirect stakeholders. Firstly, there are the runners who wear the vests with the RFID tag and run through the waypoints where the tags are scanned. Secondly, there are the individuals who set up the devices at the waypoints and ensure everything is working properly. Finally, there is the race secretariat which tracks whether all registered times are valid and administers penalties when necessary. They also ensure that the times are accurately displayed on the public website of the Batavierenrace.

3. Current system

The current system used by the eBART committee will be explained in this section. It should give a clear overview of the system as a whole and address its pitfalls. We'll give more in-depth explanations on the RabbitMQ architecture, the setup of the Batavierenrace and the logic inside the current eBPC.

3.1 Architecture

Currently, the bata uses the following system architecture (see Figure 3.1.1). At the top you see several devices sending messages to several RMQ exchanges (arrow symbols). Based on a routing key, these messages are forwarded to the correct RMQ queue (yellow symbols). The green arrows represent consumer data streams where devices retrieve messages from the queue for processing. By the bata, RMQ was chosen to temporarily buffer all messages. This allows an application to process messages when it is available and has time. Using RMQ within the architecture provides stability, a smaller error rate and better storage of messages in case something goes wrong. The key components for this project are the eBPC, the RKs, the queues lbr-rk* and the queue eBPC.

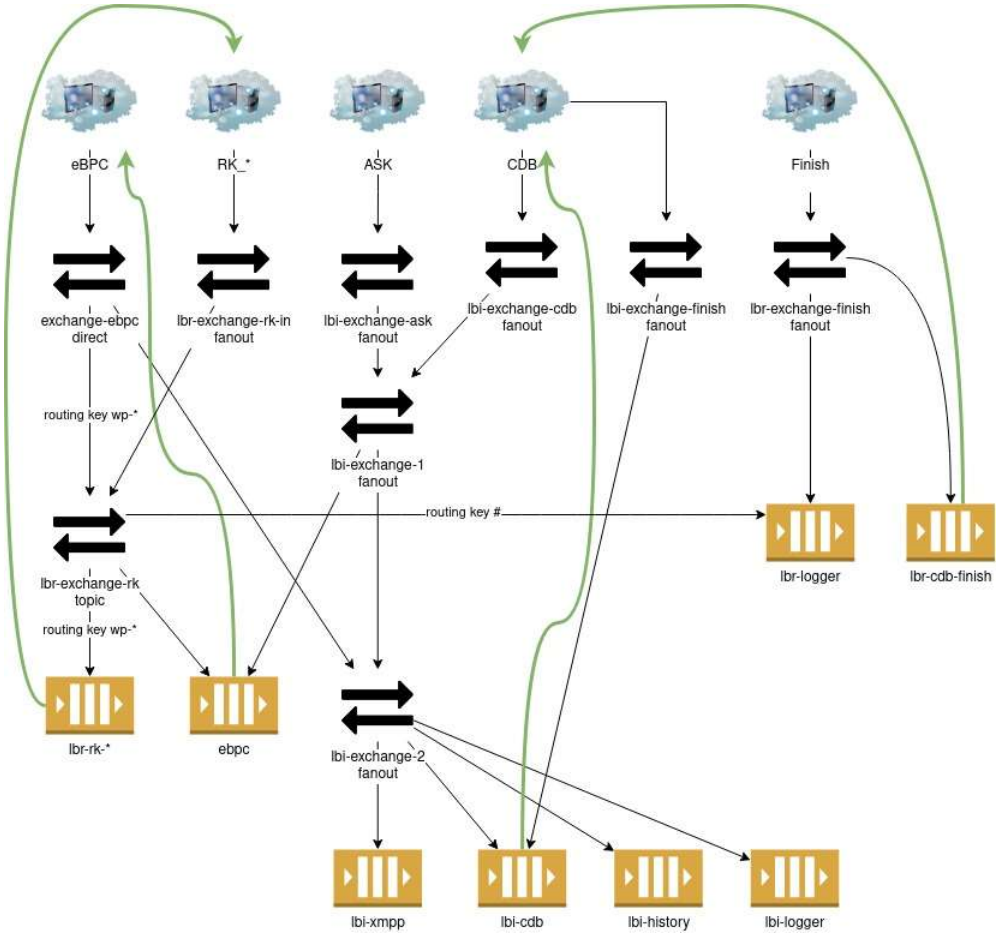

Figure 3.1.1: RMQ Architecture of the current system

3.2 Functionality

The diagram in Figure 3.2.1 shows how the eBART system currently works. During the race, there are 9 WPP driving around in a van carrying an eBART set. This set includes an RK, DI, KP, TA, BE, KK, AK and PDA. All of these components are needed to facilitate timekeeping at a WP. As soon as a runner approaches a WP, the team's number is entered into the keypad. The team number is then displayed on the DI so the next runner can get ready to start. Once the runner passes the RK, the tag is scanned and an LBR message is sent to the eBPC via intermediate exchanges. This message is sent from the RK over a 4G connection. The eBPC continuously checks if there are LBR messages ready in the eBPC queue. As soon as a message is ready, the eBPC retrieves it and processes the message. The message is converted to an LBI message (in XML) that is sent to the WS and the BPW via several exchanges. The message is also sent to the CDB. Finally, the time registration of the runner on WP 1 is forwarded to WP 2 (via queue lbr-rk-2). In this way, WP 2 can use the registered time of WP 1 to calculate a provisional stage time once the runner arrives at WP 2. This time is not official and is only displayed on the DI at WP 2. In addition to automatic timekeeping, the RK can also be used to transmit penalties, start information, tag information and messages to the eBPC. All this information is created on the TA and processed by the eBPC once it is received. Based on the type of the LBR message, data may be forwarded to the WS via an LBI message.

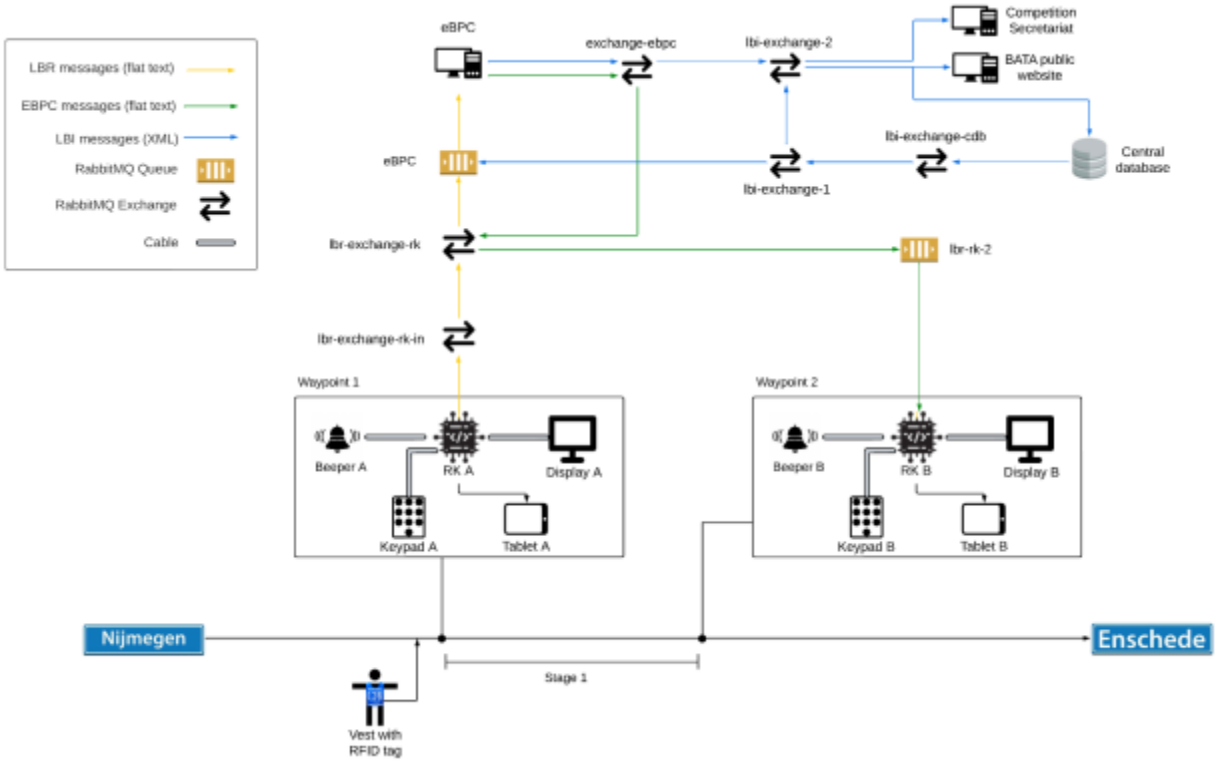

Figure 3.2.1: the eBART architecture with two WP's.

3.3 eBPC

The eBPC is an import application within the current bata time registration system. The current eBPC application is a local Microsoft .NET app. This application is used every year during the race to process all race messages. The application is being run on a Windows laptop. The application was built more than 10 years ago and is therefore very outdated. In the meantime, the application has been maintained but nothing has changed in the core code. The application connects to RabbitMQ to communicate with all the RKs, the WS and the BPW. The user interface shows the status of all waypoints/RKs, incoming LBR messages and registered RFID tags. Also, an existing RFID tag can be linked to a particular team. To store data, the application connects to a local SQLite DB. This stores all incoming and outgoing messages. The messages are also stored here in local text files as an additional backup.

One drawback of the current application is that it has to be physically started every year on a Windows laptop and someone has to be there the entire race to see if it is still running properly. Another issue is that the data in the user interface can only be viewed by the person sitting in front of the Windows laptop. A solution more appropriate to the current world would be a website where multiple people could see the data at the same time from any location.

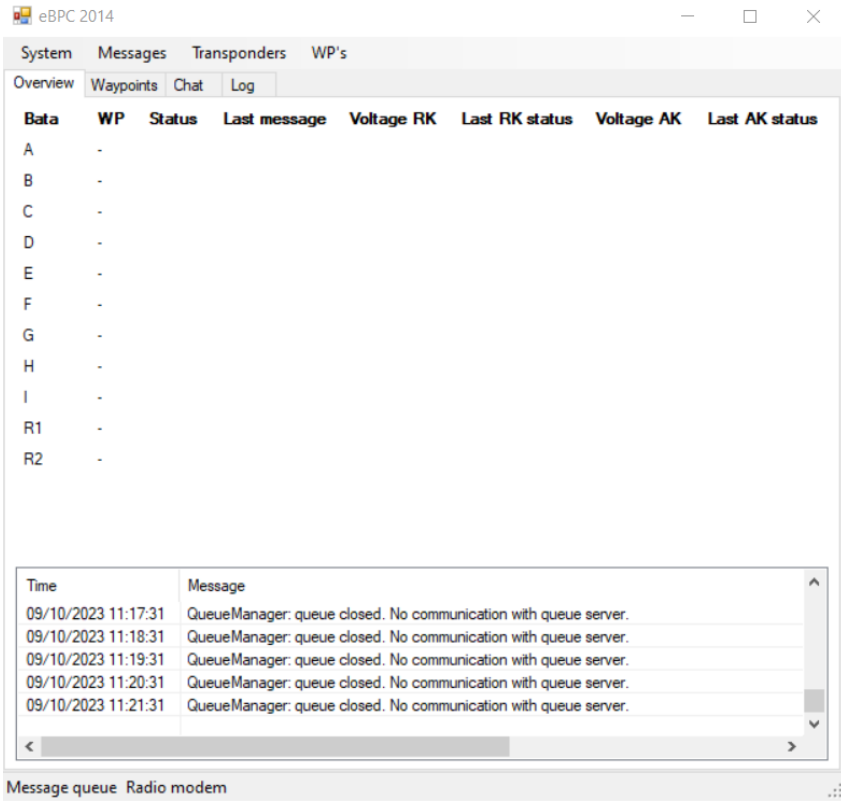

Figure 3.3.1: Screenshot of the current eBPC

4. System proposal

The new system will have some major changes compared to the current system. Here, we will go more in-depth on how the four main components of the new system will interact with each other and show the initial mockups of the front-end.

4.1 Architecture

The new system will replace the current eBPC application. For an overview of the new system architecture see Appendix A. The main driver for developing the new eBPC is to make it more accessible and easier to update. That is why we chose to develop a website instead of a local Windows application. The website was realised in the form of a React web server. In order to retrieve data from the front-end, a Spring Boot back-end was developed. This back-end will retrieve data from a PostgreSQL DB. Processing and sending all the different messages from all the RKs and the WS will be handled by a headless Python application (message server). This application will connect to RabbitMQ to retrieve and send messages. All the data coming from the messages will be stored in the PostgreSQL DB.

The 4 applications (front-end, back-end, PostgreSQL DB and message server) will each run in a separate Docker container on the bata server. This will eliminate the need to manually boot up the Windows laptop and run the application. Starting all 3 applications should be easy with just a few commands via SSH.

4.2 Mock-ups

Before the development of a fully-fledged front-end, firstly some mock-ups were made. These mock-ups were made using the web-based user interface design tool “Balsamiq Cloud”. This design phase allowed us to visualise and plan the user interface of the web application. With the mockups, a concept of the system and its eventual implementation was created. The idea was to design a user-friendly interface that aligns with the needs and expectations of the eBART committee. By presenting these mockups early in the project and getting feedback on them, an efficient development of the front-end was achieved. The input of the eBART committee provided valuable insights and refined the design. An example of a mock-up page can be seen in Figure 4.2.1

Figure 4.2.1: A Mock-up page for the waypoint specification

For an in-depth discussion about the design choices for the mockups/front-end, see Section 7.1.2. The entire design of the mock-ups can be found in Appendix B.

5. Requirement specification

In this section, the requirements of the system are listed. These are divided by application to be developed, including the database. To prioritise the requirements, the MoSCoW method has been used. This means that each requirement has been given a certain category. The categories are: must have (M), should have (S), could have (C) and won't have (W).

5.1 Message server (MS)

ID	Priority	Description
MS-1	M	Communicate with RMQ server by consuming messages from the RK queues and publishing messages to the eBPC queue.
MS-2	M	Store and retrieve data from a PostgreSQL DB.
MS-3	M	Process log entries (LBR format) coming from each active RK via RMQ.
MS-4	M	Forward received LBR messages as LBI messages to the correct RMQ exchange.
MS-5	M	Store incoming and outgoing messages in the DB.
MS-6	M	Store incoming and outgoing messages in local text files.
MS-7	M	Forward time registrations and transponder registrations to the correct RKs.
MS-8	M	Convert an LBR to an LBI message.
MS-9	M	Process the received LBR and LBI messages based on the existing eBART protocol.
MS-10	S	Log all actions done in the MS.
MS-11	S	Restart the application without losing any data.
MS-12	S	Configure RMQ and database settings in a local configuration file.
MS-13	S	Create unit tests for all developed features.
MS-14	C	Save information in a structured way inside the DB.
MS-15	C	Being able to test the MS by processing old race data (containing LBR lines) from a text file.
MS-16	C	Run inside a docker container.
MS-17	C	Handle losing DB connection by retrying the connection till it succeeds.
MS-18	C	Use multiple environments (production, testing, development) to switch between different configurations.

Table 5.1.1: MoSCoW requirements for the message server

5.2 Front-end (FE)

ID	Priority	Description
FE-1	M	Show both manual and automatic runner registrations.
FE-2	M	Show WP status information (used RK, open/closed, # runners passed, time since last message received).
FE-3	M	Show log of received messages by the MS.
FE-4	M	Register a back-up transponder and connect it to a certain team.

FE-5	M	Data from/to the back-end is formatted in JSON.
FE-6	M	MS environment can be configured.
FE-7	M	All data in the DB can be cleared.
FE-8	S	Retrieve and send data from/to the back-end via HTTPS requests.
FE-9	S	The website should not be publicly available and therefore protected by a login.
FE-10	C	Custom log entries can be registered.
FE-11	C	Status of components can be managed (in which set they are currently).
FE-12	C	Notes can be viewed and created. This note could contain a device and/or waypoint.
FE-13	C	Store user and session information of users that login into the website.
FE-14	C	Show the battery status of a RK.
FE-15	C	Show log of received messages per WP.
FE-16	C	All log files of the MS can be cleared.
FE-17	C	Run inside a Docker container.
FE-18	W	Keep track of the status of components like working or defective.

Table 5.2.1: MoSCoW requirements for the front-end

5.3 Back-end (BE)

ID	Priority	Description
BE-1	M	Retrieve and insert data from/in the DB via a JDBC connection.
BE-2	M	Process HTTPS requests from the BE and answer in JSON format.
BE-3	M	Configure DB settings inside a local configuration file.
BE-4	M	Store data inside the DB in a clear and structured way.
BE-5	C	Run inside a Docker container.

Table 5.3.1: MoSCoW requirements for the back-end

5.4 Database (DB)

ID	Priority	Description
DB-1	M	Store received LBR and LBI messages.
DB-2	M	Store sent LBI and EBPC messages.
DB-3	M	Store devices and status of each device (should be updatable).
DB-4	M	Store status of a waypoint (should be updatable).
DB-5	M	Store runner registration information.
DB-6	S	Store received messages that have an incorrect format/structure.
DB-7	S	Store team information.
DB-8	S	Store transponder information.
DB-9	S	Store battery status information (should be updatable).
DB-10	S	Store notes created in the front-end. Optionally connected to a WP or RK.
DB-11	C	Store team notes of normal teams.
DB-12	C	Store team notes of university teams.

DB-13	C	Store display info messages.
DB-14	C	Store start times of teams.
DB-15	C	Store start procedures of teams.
DB-16	C	Store device messages received from both an RK and the WS.
DB-17	C	Store user information for the front-end.
DB-19	C	Store status of components (in which set they are currently, should be updatable).

Table 5.4.1: MoSCoW requirements for the back-end

6. Global design

In this section, the global architecture of the system is explained in order to get a clear understanding of the overall system and its components. Additionally, the frameworks, programming languages and libraries that we used, are elaborated upon.

6.1 System architecture

In Figure 6.1.1 you find a schematic diagram of the software system architecture. All applications that are implemented during this project can be run as separate Docker containers inside a Docker stack. Using one docker-compose file the entire stack can be easily configured and started. The RabbitMQ server pictured in the schema is already existing and maintained by the bata. All applications are explained in more detail in the subparagraphs 6.2.1 - 6.2.4.

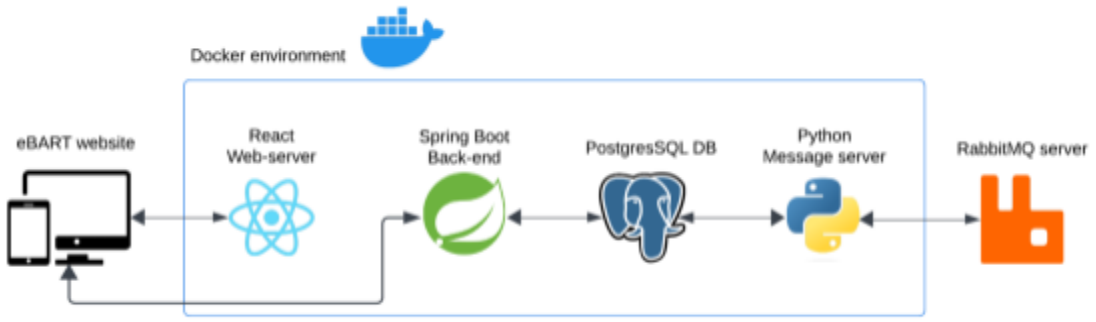

Figure 6.1.1: Schematic diagram of the software system architecture

6.2 Implementation technologies

As can be seen in the system architecture diagram in Figure 6.1.1, every component of the system is developed using a specific framework or programming language. In this section, the choices for these are explained.

6.2.1 Message Server

The message server is developed in Python. The reason for using this programming language is that each member of our project team was already familiar with Python and there are many packages available to use. Furthermore, the functionality of the message server does not require the application to have a visual interface so Python fits perfectly in the goal to make a headless server application. To connect to the existing RabbitMQ broker on the bata server we have used the package Pika. This provides an out-of-the-box synchronous client which can consume and publish messages from/to a RabbitMQ server. The PostgreSQL DB connection is set up using the psycopg2 package. Logging application status and

error messages is done using the default logging Python package. XML parsing is done using ElementTree. Unit Testing is implemented using the default unittest Python package.

6.2.2 Front-end

The front-end is developed using the React framework. React is a highly popular open-source library that uses JavaScript as the underlying programming language. The reason for using this framework is that the eBART committee as well as one of our team members were already familiar with it. Additionally, a reason for using React is that it is relatively easy to work with due to its simplicity and flexibility. To simplify and enhance the development process, the Devias Material Kit Pro[1] from GitHub was used as a template. This kit provides a set of pre-designed components and styles but also allows to build upon those in order to customise the user-interface to meet the client's needs. Utilising this kit enabled the team to quickly understand the fundamentals of React, leading to the implementation of a basic design in a short amount of time.

6.2.3 Back-end

The back-end is developed using Spring Boot. Spring Boot is commonly used in the industry to make web applications and microservices with the Spring framework faster and easier. As Java is known to a lot of programmers, including the whole project group and the eBART committee of the Batavierenrace, we decided to use Spring Boot for the project.

6.2.4 Database

The database is created/maintained using PostgreSQL. Several team members already had previous experience using this so it seemed like a logical choice to use this.

7. Detailed design

In this section, we explain the design choices for each component of the system and how they work together. Furthermore, we describe the choice for using certain security measures.

7.1 Justification of design choices

Compared to the current eBPC application, the proposed architecture is quite different and more comprehensive. Whereas the current eBPC is a Microsoft GUI application running locally on a Windows laptop, in the new system this has been replaced by 3 applications (message server, back-end and front-end). The main reason for this is that we wanted to separate the handling and processing of messages from their display to the eBART committee. Reason for this is the required robustness and reliability of the system. If we were to process both message handling and visualisation in 1 application and then something goes wrong with the visualisation, the MS crashes and thus message handling stops as well. This may absolutely not happen as the race may then have to be stopped as the times of the runners cannot be handled. In the new system, both back-end and front-end can crash while the message server remains active and continues to handle and forward messages. Should the message server still fail for whatever reason, all RKs continue to store and send their data locally to the RMQ server. Once the message server is active again, all messages in the queue will be processed. However, this is far from ideal as several parts of the system would not work in the meantime such as forwarding run times to the next RK to calculate a provisional stage time and show it at the next WP.

7.1.1 Message Server

The two main desired functionalities of the message server are receiving and sending messages via RMQ and extracting/saving data from these messages to the database. To connect to both the DB server and the RMQ server, the MS contains two Python files that each take care of a connection. The DB connection is set up when the MS is started and maintained for the duration of the run. Once the connection is lost, a new connection is attempted. If this fails, a log is made of this and the MS continues to try to connect. Without a connection to the DB, no messages can be handled from RMQ. This is because the database connection is required to store all the information in the messages so that it can be displayed in the front-end. The RMQ connection on the other side works pretty much the same. It too is set up at MS startup and will reconnect once the connection is lost. For now, a synchronous connection has been chosen where handling an incoming message occupies the only main thread available within the programme. Therefore, multiple messages cannot currently be handled simultaneously. Should it later turn out that handling the incoming messages is not fast enough, it is always possible to switch to an asynchronous connection and parallel threading. However, this requires more programming knowledge and is more difficult to debug.

To get a better understanding of how the MS works, appendix E of this document contains an application state diagram. This diagram broadly shows the logic incorporated in the MS and the order in which actions are performed.

7.1.2 Mockups

The first step towards designing a front-end was by making some mockups. The designed mock-ups represent all (important) pages of the web application. Both minor and major design choices were discussed with the eBART committee. An example of a minor design choice is the difference between appendix Figure B.1 and Figure B.2. In these two images, two simplistic designs for a login page are shown. With such minor, primarily visual adjustments, the eBART committee determined that the project team had the flexibility to exercise creative freedom in choosing an optimal layout.

During the mockup discussions with the client, another design decision emerged: The eBART committee expressed that the “Home page”, as in Figure B.3, was not needed. Given the ease of navigation between the pages using the sidebar, a distinct homepage was deemed unnecessary. Instead, their preference was for the default page to be the waypoints overview page.

Speaking of the waypoints overview page: Another major design choice concerned the visualisation of data. The current system mainly uses tables to visualise data (see Figure 3.3.1). The eBART committee showed great interest in the design of the proposed aesthetical visualisation of the WP overview page (see Figure B.4, compared to Figure B.5). It was made clear that the tabular layout from the current eBPC was not a necessity for the future. So, in the new design, these tables are replaced with a visual, more aesthetically pleasing, layout where possible.

7.1.3 Front-end

The initial meetings with the eBART, during which the scope of the project and later on the mockups were discussed, offered a clear view of the desired front-end. These early meetings provided insights into both the visual and functional requirements. This led to a quick and efficient development of the front-end.

As explained at the global design (see Section 6.2.2), React was used for the front-end. React is a JavaScript library that can be used as a framework. Furthermore, we used the Material UI component library (https://mui.com/) to have a good design base and use existing well-defined components.

The first page that the user sees when accessing the website is the login page. The authentication is done via Google Oauth2. With the eBART it was discussed that all members should already have a Google account since the committee is currently also using Google Drive. Because of this it was no problem to have the authentication done by whitelisting Google emails. For more details about the security aspect see Section 7.2.

In Figure 7.1.3.1 a screenshot of the waypoint overview page is shown. On the left, there is a side navigation menu that allows the user to easily navigate between the different pages. In the middle, an overview of the waypoints is shown. As discussed in the detailed design section about the mockups (see Section 7.1.2), this visual representation was preferred over the tabular format. All elements from the current system are shown in a visual representation. Note that the current overview page of the eBPC, as shown in Figure 3.3.1, has fields for battery statuses of the RK. Currently, however, this field is not in use

as the RK's do not send their battery information to the eBPC. Therefore, it was decided to leave this out in the front-end. We did add a function in the back-end that can handle this in case the eBART decides to implement the sending of battery information in the future.

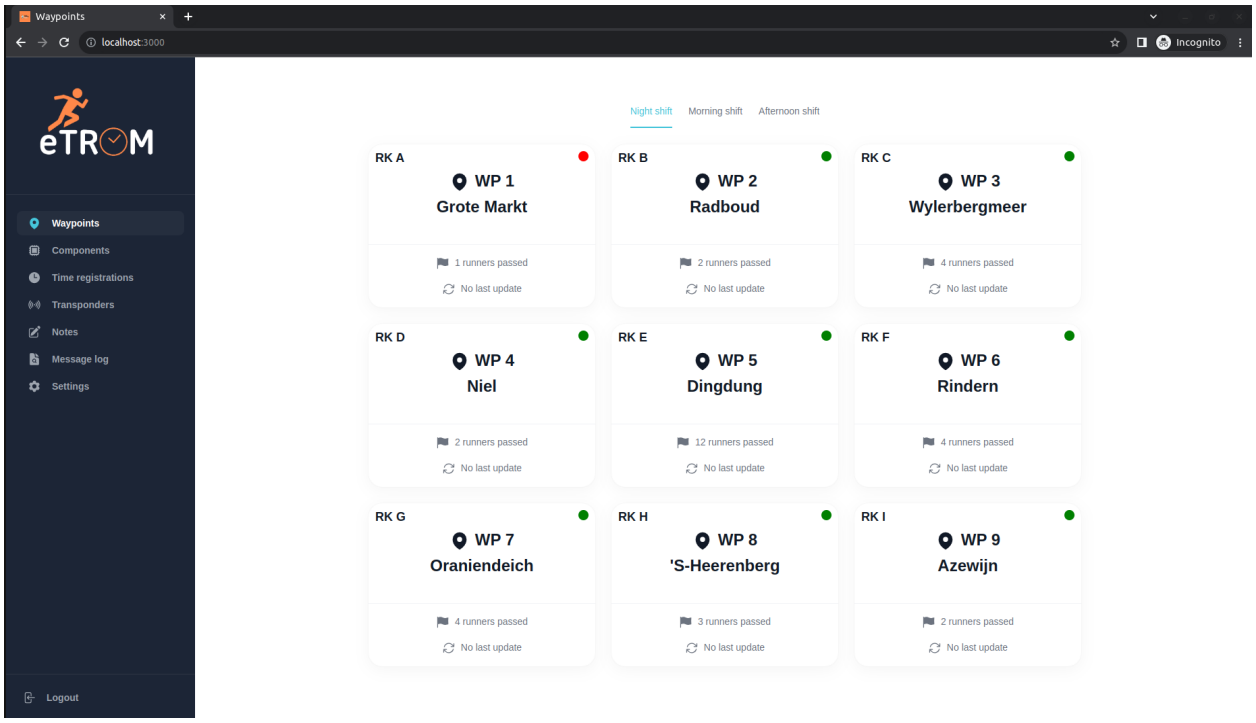

Figure 7.1.3.1: The waypoint overview page of the front-end

The Batavierenrace consists of 25 stages, which are divided into a night, morning and afternoon shift. The user can select a shift at the top menu, which will show the corresponding waypoints. This design cleverly avoids the issue of screen congestion that would occur if all 25 waypoints were squeezed onto a single display. Throughout the Batavierenrace, it is impossible for two shifts to be active simultaneously. So, this design also makes sure that only relevant information is shown. In the top left corner of each waypoint, the BataID is matched with the WP. Here, the BataID shows which RK is placed at which WP. In principle, the Bata starts with 'RK set A' at WP-1, set B at WP-2, et cetera. For the morning shift the RK's are moved towards the next waypoints, such that set A normally ends up at WP-10. During the race, however, it is possible that RK's are placed at a different location. It can also be the case that a spare RK is used. Therefore, the overview page neatly shows which RK is at which WP.

Furthermore, the waypoints overview page shows whether a certain waypoint is opened/closed with a green/red dot in the top right corner. The waypoint shows the amount of runners that passed the checkpoint and when the last message was received. Clicking on the box of a waypoint brings the user to the waypoint specification page of that specific waypoint.

The full overview of all pages in the front-end can be found in Appendix C. For a detailed page-by-page breakdown of the front-end and its functionality, refer to the eTROM manual.

7.1.4 Back-end

For the back-end, a piece of software that could handle the communication between the DB and the front-end was required. Hence, we developed a RESTful API. This API serves as an interface that manages the communication between the front-end and the back-end of a system. In our case, it handles the requests from the webpage and fetches the required data from the DB. There are different types of requests possible (see Table 7.1.4.1)

Request type	Usage
GET	Used to retrieve a resource
POST	Used to create a new resource
PATCH	Used to update an existing resource, including partial updates
DELETE	Used to delete a resource

Table 7.1.4.1: HTTP methods of the RESTful API

After outlining the possible types of requests in the Table 7.1.4.1, it is important to emphasise that each request serves a specific purpose in handling the communication between the front-end and the database. For instance, when looking at the note endpoint, a GET request is used to fetch all notes, a POST request is used to add a note, a PATCH request updates an existing note, and a DELETE request is used to delete a note. Each request has a specific response structure, specifying the information to be fetched from the DB. To fully document all different requests, our team has created an extensive API specification document (see file attached to this report). This document outlines all possible requests, providing details for each request, including the type of request, the response structure and an example response. This ensures a detailed and consistent understanding of the communication protocols in our system.

To ensure the proper functioning of the system, it is crucial to implement error handling. This is necessary in case a request fails or if the API encounters difficulties fetching the required information from the DB. When implementing error handling, the team tried to align closely with the standard HTTP and REST conventions. This is done in order to maintain clarity and understanding in the error messages generated (see Table 7.1.4.2).

Error code	Meaning
400 Bad Request	The request was malformed. The response body will include an error providing further information
401 Unauthorised	The provided authorization token is not valid (anymore)
404 Not Found	The requested resource did not exist

500 Internal Server Error	Either the request was invalid and the server cannot process the request or the request was valid but the server encountered an unexpected condition that prevented it from fulfilling the request

Table 7.1.4.2: HTTP error codes

The implementation of the API contains different components. Each component has a specific role in the application. These components are organised in a Model-View-Controller (MVC[2]) pattern, where the Model represents the data, the View represents the user interface, and the Controller manages the user input and updates the Model and the View accordingly. In our application, five components are implemented. Firstly, the controller. This component handles incoming requests and makes use of the other components to compose a response if necessary. Secondly, the Data Transfer Object (DTO[3]). A DTO exchanges data between the client and the database. They can contain data from multiple entities and are returned to the client upon request. Thirdly, the entities. Entities represent data objects that are directly connected to the database. Entities can also be seen as models. Lastly, the repository. This component is responsible for the connection between the application and the back-end containing SQL queries and is able to immediately convert data to an entity. By default, Spring Boot creates simple queries for you. However, when working with a lot of data, it is beneficial to overwrite these queries with your own. Spring Boot always consists of at least these components, but when the back-end also has to contain a lot of logic, the logic can be moved to Service classes. In our case, the logic is minimal and we mostly return data immediately from the database through DTO's. Therefore, we decided to keep the minimal logic we have inside the controllers. Additionally, this saves us some time as we only have to test the controllers.

7.1.5 Database

Before creating the DB design, we first looked at the DB structure of the current eBPC. It turned out that it had few relational tables, requiring many filters to visualise the data in the GUI. Additionally, certain data, such as incorrect messages and user information, was not stored. The new DB is based on a relational structure where each type of data has its own table. In addition, we chose to store the meanings of all kinds of data from the eBART message protocol. An example is that a record can have type A. According to the protocol, this means that it is an automatic registration. To show this in the front-end, the back-end can extract this information directly from the database by having a table registration_type in which this information is defined.

In Table 7.1.5.1 is an explanation of each table's function and what other tables it links to.

Table	**Links to**	**Contains/function**
battery_status	received_message, battery_status_type	Status (actual voltage) of batteries used at waypoints during the race.
battery_status_type		Descriptions of battery_status types.
component	component_type	Keep track of which components are used at every WP

		and in which component set they are contained.
component_set		Set of components.
component_type		Descriptions of component types.
device	device_type, battery_status, received_message, sent_ebpc_msg_id	Mapped to BataID in the eBART protocol. Physical devices which are able to send/receive RMQ messages.
device_message	device	Messages broadcasted from a device during the race. This will mainly be the WS sending messages.
device_message_fragment	device, device_message, received_message	If a RK sends a device_message this message is split into multiple LBRs due to size constraints of the LBR message. Split fragments of the message are stored in this table.
device_type		Descriptions of device types.
device_waypoint	device, waypoint	Keep track of which waypoint a RK is located during the race. Contains both historical and actual data.
display_info	received_message	Information (sponsors, rankings, etc.) which is displayed at a WP on a screen during the race.
note	device, waypoint	Used by the front-end to add notes during the race. Optionally connected to a waypoint and/or device.
ping	ping_type, device	Ping tests are done by devices via RMQ. The results of ping tests done with the MS are stored here.
ping_type		Descriptions of ping types.
received_incorrect_message		Messages received via RMQ (both LBR and LBI) that have an incorrect format or structure.
received_message	device, team	All received LBR and LBI messages with correct format.
registration	registration_type, received_message	Runner time registrations received from all RKs during the race.
registration_type		Descriptions of registration types.
sent_message_ebpc	device	EBPC messages still to be sent or already sent to an RK.

sent_message_lbi		LBI messages still to be sent or already sent to the correct RMQ exchange. This is mostly LBR messages that are converted to LBI and forwarded to WS, CDB and BPW.
setting		Used to store several settings from the front-end like reset of DB, reset of MS logs and changing MS environment.
start_procedure	start_procedure_type, received_message	Start procedures performed by a WP during the race.
start_procedure_type		Descriptions of start procedure types.
start_time	received_message	Start times of all teams in a certain group for a specific stage during the race.
start_time_team	start_time, team	Many-to-many couple table between start_time and team.
team	team_type	Team of runners that is participating in the race.
team_note	team_note_type	Penalties for non-university teams. Can be added by a WP or by WS.
team_note_type		Descriptions of team note types.
team_type		Descriptions of team types.
transponder	transponder_type, team, received_message	Transponder registrations done by either an WP or manually added in the front-end.
transponder_type		Descriptions of transponder types.
university_team_note	university_team_note_type, received_message	Penalties for university teams. Can be added by a WP or by WS.
university_team_note_type		Descriptions of university team note types.
user		Log-in sessions of front-end users.
waypoint		Waypoints used in the race.
waypoint_status	waypoint, waypoint_status_type, received_message	Keep track of the actual status of a waypoint during the race. Does not save historical data.

waypoint_status_type	Descriptions of waypoint status types.

Table 7.1.5.1: Explanation and links of tables in the database

7.2 Security

As mentioned earlier, currently the eBPC operates on a Windows laptop, requiring the physical presence of the eBART committee to login to the computer and start the application. Given that the new system is web-based, it requires a secure login mechanism. Without this, there is a risk of unauthorised access to the Batavierenrace data, which should only be accessible to members of the eBART committee and the WS. To address this, the new system implements a secure login page. Initially, a simple password system was considered. However, the team eventually opted for a more robust option. Hence, the new system uses Google OAuth2[3], ensuring an elevated level of security and control over who has access to the system.

OAuth2, denoting Open Authorisation 2.0, allows users to log in using their Google credentials. The database stores a whitelist of the email addresses of all the users that are allowed to access the system. When a user attempts to login to the system, they are redirected to the Google login page to enter their Google credentials. Google then authenticates the user and generates a Google Oauth2 identity JSON Web Token. This token is sent to the Google OAuth2 verifier in the back-end, which checks if the token is valid and matches with an email address in the whitelist. If the email is on the whitelist, the user is granted access, otherwise access is denied. Following this initial authentication and verification against the whitelist, the user receives a session token. When the user navigates within the web application, this token is included in the request parameters. When the back-end receives a request it verifies the session token by checking if it is the same one as issued during the user's authentication and if it has not expired yet. If valid, the server grants access to the requested resource, otherwise the server denies access and the user will need to re-authenticate.

The reason for choosing Google OAuth2 is because it is relatively easy to implement and yet provides a secure authentication protocol. Additionally, other applications within the Bata system already utilise Google authentication, and the eBART committee uses Google Drive for documentation. This means that everyone who will work with the new system should already have a Google account, making Google OAuth2 a secure, easily implementable option which will work for all intended users.

8. Testing

To verify that all applications within the system work as we expect, we used 3 different types of testing. Using unit tests, we tested functions at a low level within the code. Using an integration test, we tested the integration of the different systems together. Finally, we used (user) acceptance testing to verify with the client (Batavierenrace) that the system meets the predefined requirements.

8.1 Resources

To make testing as efficient as possible, we chose to use an online server for both the PostgreSQL and RabbitMQ servers. This way, we have access to a working test server whenever we want and it allows us to easily share test data within the team. In addition, Docker was used to easily run all applications locally side by side. With the push of a button, the software can be built and the processes can be started.

8.2 Test plan

8.2.1 Unit testing

8.2.1.1 Message server

The standard Python unittest package was used for unit testing in the MS. This allows the creation of a test class in which test cases can be defined. Before and after the test, certain actions can be performed such as establishing a connection to a server or clearing a database. Also, before and after each test case a certain action can be performed. A total of three different tests were implemented within the message server: a test for the RabbitMQ connection, a test for message handling and a test for the connection to the PostgresQL database. All test cases within the 3 test classes can be run within a Python development environment at the click of a button. Afterwards, the command line shows the result indicating which tests passed and which failed.

In Figure 8.2.1.1.1 is an example of a test case within the RMQ test class. This test case is used to test sending a message to the RMQ server.

```
def test_publish_message(self):
    self.assertTrue(rmq.publish_message('Test publishing of LBI message with empty routing key'))
    self.assertTrue(rmq.publish_message( msg: 'Test publishing of ebpc message to RK A with routing key wp-a'
                                        , routing_key: 'wp-a'))
```

Figure 8.2.1.1.1: Unit test for sending a message to the RMQ server

In Figure 8.2.1.1.2 is an example of a test case within the DB test class. This test case tests the creation and retrieval of a record within the table note.

```
def test_note(self):
    registered = datetime.now()
    n_id = dbc.insert_note( message: 'Testmessage',  device_id: 'A',  waypoint_num: '1', registered)
    self.assertEqual( first: (n_id, 'Testmessage', 'A', 1, registered), dbc.get_note(n_id))
```

Figure 8.2.1.1.2: Unit test for creating a note

In Figure 8.2.1.1.3 is an example of a test case within the message processing test class. This test case tests the processing of a runner registration.

```
def test_lbr_command_w(self):
    lbr = 'A\t0001\t01\tW\t10323700\t\t0       \r\n'
    self.assertTrue(mp.process_received_msg(lbr))
    self.assertEqual( first: 'O', dbc.get_waypoint_status(1)[1])
    racetime = timedelta( days: 0,  seconds: 37,  microseconds: 0,  milliseconds: 0,  minutes: 32,  hours: 10,  weeks: 0)
    self.assertEqual( first: [(1, 'A', 1, 'W', racetime, None, 'O', lbr)],
                      [i[1:-1] for i in dbc.get_received_messages()])
    self.assertEqual( first: [('0001', 'A', '01', 'W', 'PT10H32M37S', None, None, 'O')],
                      [i[1:-2] for i in dbc.get_sent_message_lbis()])
```

Figure 8.2.1.1.3: Unit test for registering a runner

8.2.1.2 Front-end

For the front-end, we made the choice at the beginning of the project not to implement unit testing. The reason for this is that in our view, the only good way to test it properly is to use an automated testing framework such as Selenium. However, the initial learning curve of using Selenium is rather steep and most of the team members had no experience with it. Also, with the development of 3 different applications and a database, the amount of workload was already quite high. It would be a good add-on to add unit tests to the front-end after this project.

8.2.1.3 Back-end

As mentioned before, we made the choice to not make use of Spring Boot's Service classes. Therefore, we only have to write tests for the controllers as the logic is there. Spring Boot also offers its own testing framework, so we decided to use this. Instead of writing JUnit tests Spring Boot allows you to test all endpoints by using a MockMVC. Writing a series of tests where we execute "get -> add -> get -> delete -> get" on a specific object we can test if the various endpoints work correctly and see if the data is returned correctly according to the defined DTO's. To execute all tests, make sure to switch to the test-database in the applications.xml and run the project using the command: "./gradlew test" instead of "./gradlew bootRun" inside the project.

8.2.2 Integration testing

To test that all 4 parts of the system (message server, database, front-end and back-end) work together correctly, the integration test below was written. It is important to do all test cases. As soon as a step cannot be performed or the result is not as expected, this should be noted down in the result and comment. If any of the test cases do not succeed, the system should be changed and all tests should be done again.

Test case	Action	Expected result	Result	Comment
1	Run 'docker-compose -f docker-compose-dev.yml up –build' to start all applications using docker	All apps should start	✓	-
2	Check in the docker logs of the back-end	Back-end should be connected to the DB and fully started without any errors	✓	-
3	Check in the docker logs of the front-end	Front-end should be started and reachable on http://localhost. The logs should not contain any errors and should be compiled.	✓	Several compile warnings are listed because of un-used resources
4	Login in the front-end using a whitelisted Google email address.	The back-end should process the HTTP request and return a session token. The front-end should redirect to the homepage.		
5	Check the docker logs of the message server	Message server should be connected to both the DB and the RMQ server and fully started	✓	-
6	Insert a test message into the RMQ ebpc queue	The message should be consumed by the message server, this should be visible in the logs	✓	-
7	Check the lbi-cdb queue of the RMQ server	This queue should contain the message previously inserted, now in XML format. The data should be the same.	✓	-

8	Check message log page inside the front-end	The previously inserted message should be visible in the table and should contain the same data as the message.	✓	-
9	Reset the database in the front-end on the settings page.	All non-static data in the DB should be removed.	✓	-
10	Reset the message server logs in the front-end on the settings page.	All log files of the message server should be empty.	✓	-
11	Create a new user on the settings page in the front-end.	The page should reload and show the new user in the table. Also, the new user should be existing in the user table inside the database.	✓	-
12	Insert another test message into the RMQ server which requires ebpc messages to be sent via RMQ (e.g. start times).	The message server should process the message and show this in the logging. It should also insert records in the sent_message_ebpc table. Then the message server should send the messages to the correct wp queue.	✓	-

Table 8.2.2.1: Integration test for the whole system

8.2.3 Acceptance testing

To verify that the system functions as expected, an acceptance test was written. This test is mainly intended to test the functional operation of the front-end. The test consists of several test cases in which a certain action must be performed in each step. The application should return a certain response, such as updating a table with data or adding a row to the table. The table also contains references to established requirements. This means that once this test case can be completed successfully, the stated requirement has been met. Before running the test, the DB must be populated with test data. The test plan was completed together with the client (the batavierenrace) and the results of this have been listed.

Test case	Feature	Requirement	Action	Expected result	Result	Comment
1	Secure login incorrect email	FE-9	Login using a non-whitelisted Google email address	Pop-up should appear saying email address is incorrect	✓	-
2	Secure login	FE-9	Login using a whitelisted Google email address	Redirect to homepage	✓	-
3	Shome time registrations	FE-1	Go to the time registrations page.	Table should be filled with time registration entries	✓	-
4	Show status of a waypoint	FE-2	Go to the homepage	Per WP the used RK, open/closed, # runners passed and last update time should be visible	✓	-
5	Show log of received messages.	FE-3	Go to the message log page	Table should be filled with message log entries	✓	-
6	Assign back-up transponder	FE-4	Go to the transponders page, click on add and fill in a team and transponder number	The transponder assignment should be added and shown in the table	✓	-

7	Configure MS environment	FE-6	Go to the settings page and change the MS environment	Data in all the tables should change since the database also changed.	X	Development of this feature is not finished yet
8	Reset database	FE-7	Go to the settings page and click the reset database button	All non-static data in the database should be cleared so most tables on pages should now be empty	✓	-
9	Show components	FE-11	Go the components page	All component sets should be visible and the components that are currently in there	✓	-
10	Edit components	FE-11	Edit a component set	The component should disappear from the other set and re-appear in the set you made the edit in.	✓	-
11	View notes	FE-12	Go to the notes page	There should be note entries visible in the table	✓	-
12	Create a note	FE-12	Create a new note	The new note should appear in the table displaying the WP, set and message	✓	-
13	Store user info	FE-13	Go to the settings page and scroll down to users	The table should display the email address and last activity timestamp of each user	✓	-
14	Message log per waypoint	FE-15	Go to the detail page of waypoint 1	There should be a table showing messages from only waypoint 1	✓	-
15	Reset MS log files	FE-16	Go to the settings page and click on reset MS logs	The MS logs should be emptied	✓	-

Table 8.2.3.1: Acceptance tests with the linked requirement

9. Conclusion

9.1 Requirements verification

To verify which requirements we met or partially met, we went through the requirements in Tables 9.1.1.1 - 9.1.4.1 with the client (Batavierenrace). The requirements we were unable to meet were underlined. If applicable, the reason for non-compliance is indicated in blue below the requirement. In the end, we were able to implement all requirements with a 'must have' priority. In addition, we managed to develop many requirements with a 'should' and 'could' priority as well.

9.1.1 Message server (MS)

ID	Priority	Description
MS-1	M	Communicate with RMQ server by consuming messages from the RK queues and publishing messages to the eBPC queue.
MS-2	M	Store and retrieve data from a PostgreSQL DB.
MS-3	M	Process log entries (LBR format) coming from each active RK via RMQ.
MS-4	M	Forward received LBR messages as LBI messages to the correct RMQ exchange.
MS-5	M	Store incoming and outgoing messages in the DB.
MS-6	M	Store incoming and outgoing messages in local text files.
MS-7	M	Forward time registrations and transponder registrations to the correct RKs.
MS-8	M	Convert an LBR to an LBI message.
MS-9	M	Process the received LBR and LBI messages based on the existing eBART protocol.
MS-10	S	Log all actions done in the MS.
MS-11	S	Restart the application without losing any data.
MS-12	S	Configure RMQ and database settings in a local configuration file.
MS-13	S	Create unit tests for all developed features.
MS-14	C	Save information in a structured way inside the DB.
MS-15	C	Being able to test the MS by processing old race data (containing LBR lines) from a text file.
MS-16	C	Run inside a Docker container.
MS-17	C	Handle losing DB connection by retrying the connection till it succeeds.
MS-18	C	Use multiple environments (production, testing, development) to switch between different configurations.

Table 9.1.1.1: Summary of requirements for the message server

9.1.2 Front-end (FE)

ID	Priority	Description
FE-1	M	Show both manual and automatic runner registrations.
FE-2	M	Show WP status information (used RK, open/closed, # runners passed, time since

		last message received).
FE-3	M	Show log of received messages by the MS.
FE-4	M	Register a back-up transponder and connect it to a certain team.
FE-5	M	Data from/to the back-end is formatted in JSON.
FE-6	M	MS environment can be configured.
FE-7	M	All data in the DB can be cleared.
FE-8	S	Retrieve and send data from/to the back-end via HTTPS requests. Currently requests are HTTP, it is possible to change this to HTTPS but this requires certificates to be added.
FE-9	S	The website should not be publicly available and therefore protected by a login.
FE-10	C	Custom log entries can be registered. Since all data in the front-end is split into multiple pages were if necessary you can also add data we decided during the design phase that it should not be possible to make message log entries in the front-end.
FE-11	C	Status of components can be managed (in which set they are currently).
FE-12	C	Notes can be viewed and created. This note could contain a device and/or waypoint.
FE-13	C	Store user and session information of users that login into the website.
FE-14	C	Show the battery status of a RK. During the project, it turned out that an RK does not yet transmit battery status. As we do not have this data, we could not implement this function.
FE-15	C	Show log of received messages per WP.
FE-16	C	All log files of the MS can be cleared.
FE-17	C	Run inside a Docker container.
FE-18	W	Keep track of the status of components like working or defective. We keep track of which components are in a component set but not their status.

Table 9.1.2.1: Summary of requirements for the front-end

9.1.3 Back-end (BE)

ID	Priority	Description
BE-1	M	Retrieve and insert data from/in the DB via a JDBC connection.
BE-2	M	Process HTTPS requests from the BE and answer in JSON format.
BE-3	M	Configure DB settings inside a local configuration file.
BE-4	M	Store data inside the DB in a clear and structured way.
BE-5	C	Run inside a Docker container.

Table 9.1.3.1: Summary of requirements for the back-end

9.1.4 Database (DB)

ID	Priority	Description
DB-1	M	Store received LBR and LBI messages.
DB-2	M	Store sent LBI and EBPC messages.

DB-3	M	Store devices and status of each device (should be updatable).
DB-4	M	Store status of a waypoint (should be updatable).
DB-5	M	Store runner registration information.
DB-6	S	Store received messages that have an incorrect format/structure.
DB-7	S	Store team information.
DB-8	S	Store transponder information.
DB-9	S	Store battery status information (should be updatable). This information can be stored and the message server is also programmed to do so but as earlier mentioned this data is not available yet.
DB-10	S	Store notes created in the front-end. Optionally connected to a WP or RK.
DB-11	C	Store team notes of normal teams.
DB-12	C	Store team notes of university teams.
DB-13	C	Store display info messages.
DB-14	C	Store start times of teams.
DB-15	C	Store start procedures of teams.
DB-16	C	Store device messages received from both an RK and the WS.
DB-17	C	Store user information of the front-end.
DB-19	C	Store status of components (in which set they are currently, should be updatable).

Table 9.1.4.1: Summary of requirements for the database

9.2 Future work

Despite the fact that we managed to create a beautiful system with many functionalities within 10 weeks, there are certain things that could be improved or that could still be researched. Some of these things are necessarily essential before you could use the system in production during the batavierenrace, other things are simply nice to add.

- Together with the batavierenrace set-up a production environment on one of the batavierenrace servers. This will take some effort since you have to deal with lots of configuration like SSH connections, vpn, SSL certificates, Docker environment variables etc.
- Run the system during the next batavierenrace as a shadow next to the current system to test its performance. In this way it is not yet mission critical since the system can fail without the race itself being in danger.
- Run more tests with the message server regarding its connection to the Rabbit MQ broker. Examples include losing the server connection, receiving a message with an incorrect format and sending many messages to the message server at once. Currently, a synchronous RabbitMQ connection is used. This means that receiving and sending a message takes up the only thread within the application. It is possible that due to the amount of data within a short period of time, an asynchronous connection should still be used. More research should reveal this.
- Add unit testing to the front-end by using Selenium, for instance. This will probably take a lot of time to build but once you have it, it is a quick, validated way to test the front-end after making changes.

- Implement HTTPS requests between the back-end and the front-end. Since the React application runs client-side, the client must be able to send requests to the back-end in a secure way. This requires the use of HTTPS requests instead of HTTP which in turn requires the implementation of certificates.
- Visualise extra data in the front-end that is already available in the database as extra pages. Examples include start times, start procedures, team penalties and device messages.
- Add more internal tests inside all applications to sanitise user inputs, check types before parsing types (e.g. string to int conversion) and add more logging to make all apps more understandable for a future developer.

9.3 Evaluation

Looking back on the project, we are incredibly proud of what we managed to put together in 10 weeks. Despite the complex existing protocol and architecture into which the new system had to be incorporated, we managed to create a concrete plan within the first 2 weeks. Then we started designing the database and once it was almost finished, we divided the tasks and started developing the applications. The message server, back-end and front-end were developed separately but through continuous consultation with each other. An API specification was created for the connection between the back-end and front-end. Because of this, both the back-end developer and front-end developer knew what the HTTP headers and responses should look like. For the design of the front-end, several mock-ups were made which we then showed to the batavierenrace. After some feedback, we made some more adjustments to the design. Next, we were quite easily able to translate the design into components in React.

During the project, we had weekly meetings with our supervisor Tom van Dijk. In these we discussed the progress, went through technical problems such as optimising SQL queries in the back-end and went through the planning. Every two weeks we also met with Lotte and Sarah from the Batavierenrace. Here the focus was mainly on the progress and current functionalities of the front-end. We are satisfied with the communication within the project. There was clear communication to each other about what needs to be done, who is going to do it, when and who is going to review it. Also, the daily stand-ups were a nice way to start the day and hear from each other what you did the day before and/or if there are things you need help with.

A discussion about the back-end framework Spring Boot arose at the beginning of the project. The learning curve turned out to be quite high, making it difficult to create something working at first. Therefore, Niels indicated that he is familiar with Rust and could easily build a back-end in it. He started working on it right away. We were of course pleased with this new option, but within the group Max expressed his concerns about using Rust. One disadvantage of it is that few people know it and it is therefore difficult to maintain. This is critical if we were to eventually use the system during the Batavierenrace. Because of this reason, it was finally decided to continue with Spring Boot. This is a good example of some discussion and disagreement within the group that was correctly handled and resolved through good communication.

9.4 Individual contributions

Below is an overview of the division of tasks within the project.

- **Lieke Turenhout**: front-end, presentation slides designer, slides presenter, poster designer.
- **Max Jeltes**: database, message server, front-end, slides presenter.
- **Niels Kruk**: front-end and back-end.
- **Ruben de Koning**: front-end and back-end.
- **Sybe de Oude**: front-end, presentation slides designer, slides presenter, poster designer.

10. Bibliography

[1] CleverDeveloper0929. (2023). *Devias-Material-Kit-Pro*. GitHub. https://github.com/CleverDeveloper0929/Devias-Material-Kit-Pro

[2] *MVC Design Pattern*. (2023, September 27). Geeksforgeeks. https://www.geeksforgeeks.org/mvc-design-pattern/

[3] Ranjan Rout, A. (2022, May 22). *Data Transfer Object (DTO) in Spring MVC with Example*. Geeksforgeeks. https://www.geeksforgeeks.org/data-transfer-object-dto-in-spring-mvc-with-example/

[4] *Using OAuth 2.0 to Access Google APIs*. (2023, October 18). Google. https://developers.google.com/identity/protocols/oauth2

Appendices

A. New system architecture

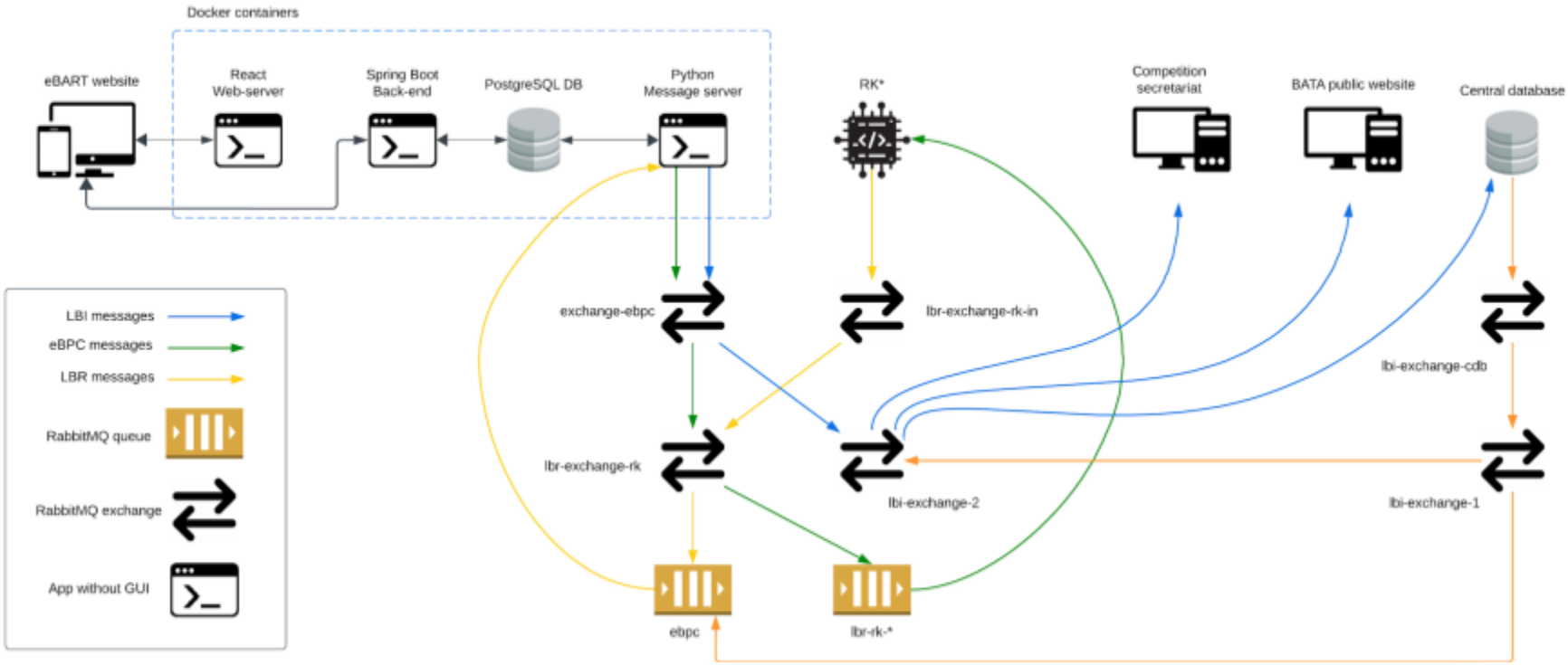

Figure A.1: Architecture of the new system

B. Front-end mock-ups

Figure B.1: Mockup of the login page

Figure B.2: Mockup of the login page version 2

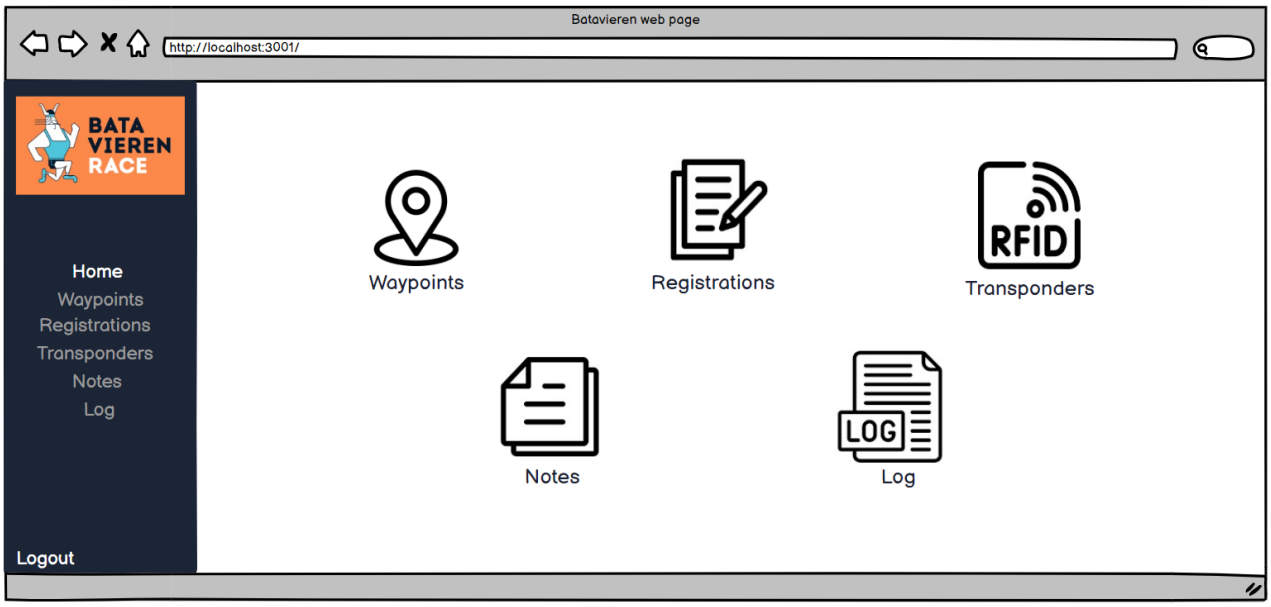

Figure B.3: Mockup of a 'home' page

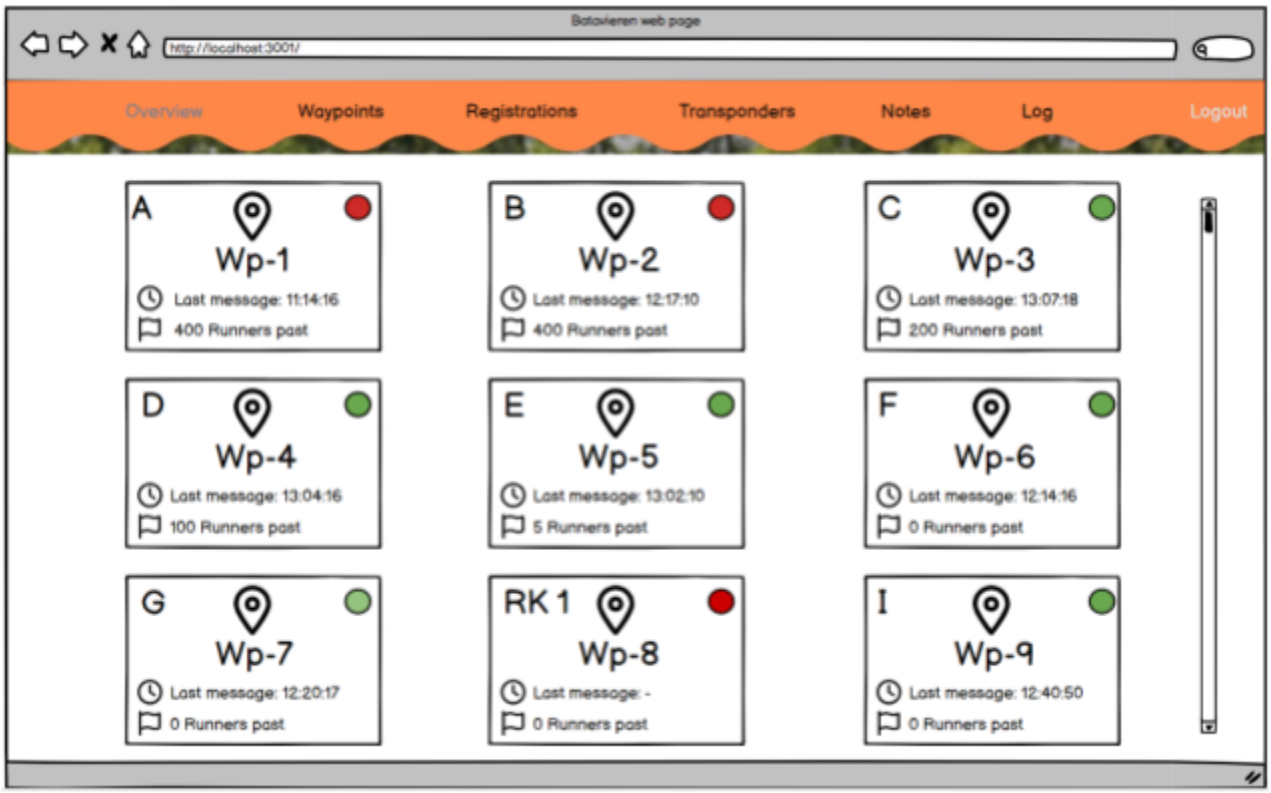

Figure B.4: Mockup of the overview page (visual representation)

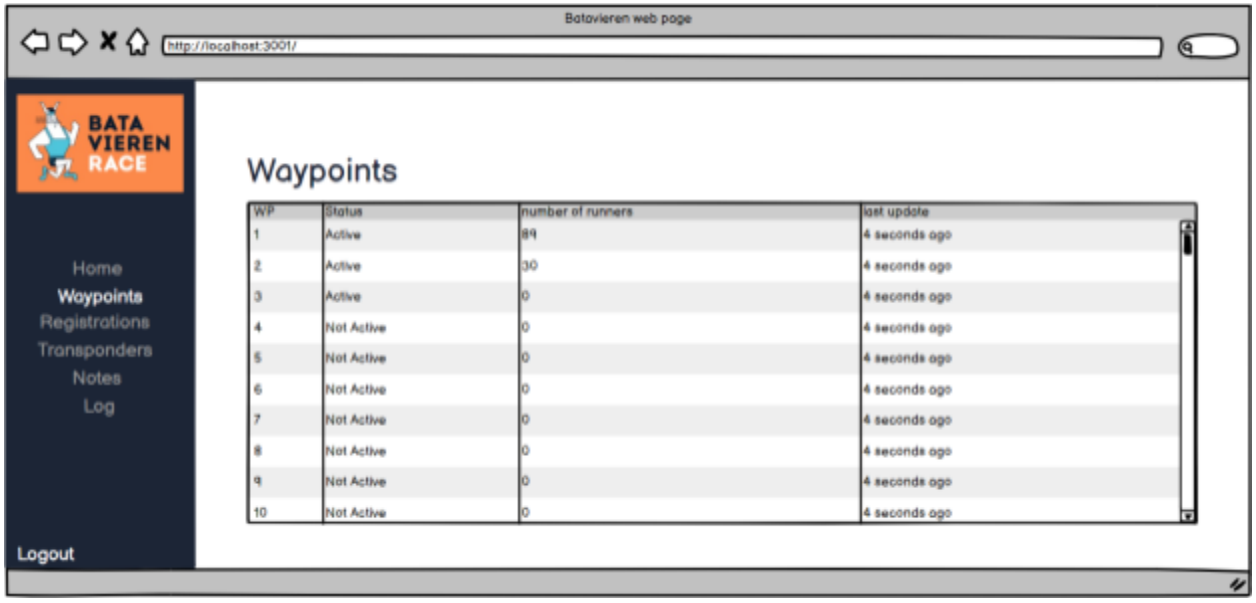

Figure B.5: Mockup of the overview page (tabular representation)

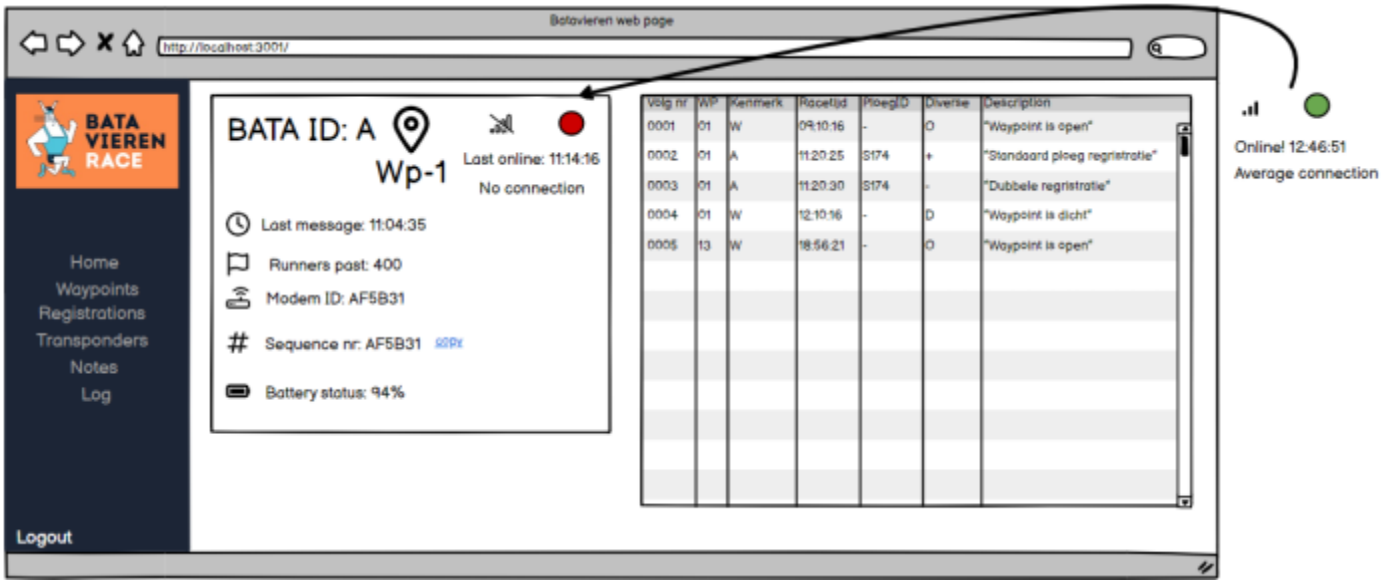

Figure B.6: Mockup of the WP specification pages

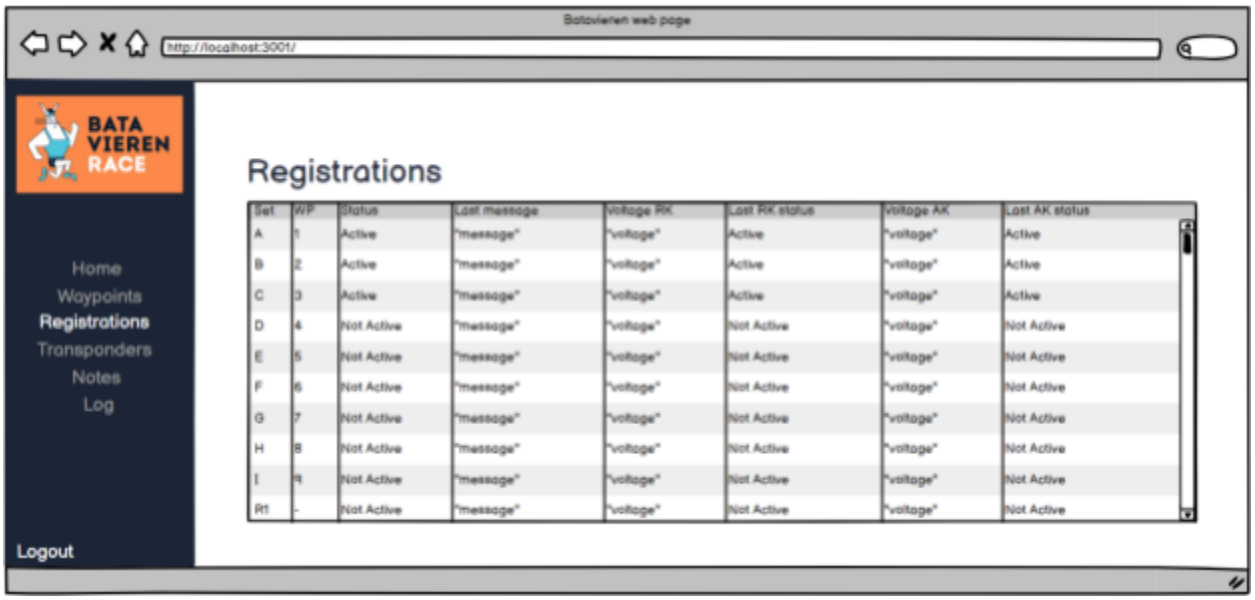

Figure B.7: Mockup of the registrations page

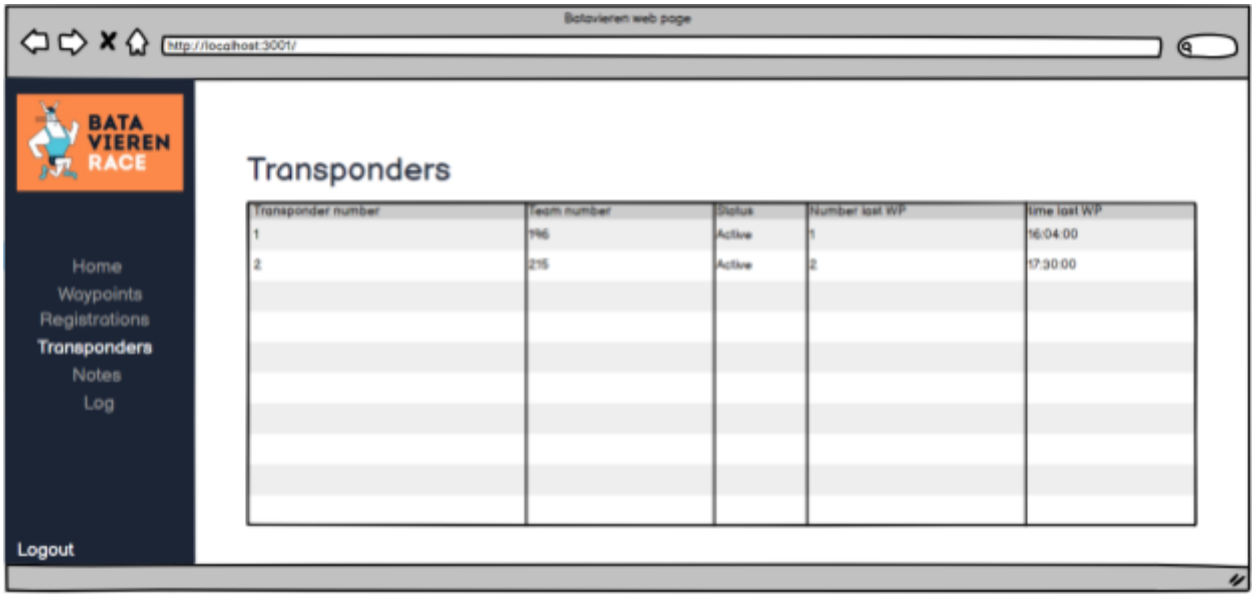

Figure B.8: Mockup of the transponders page

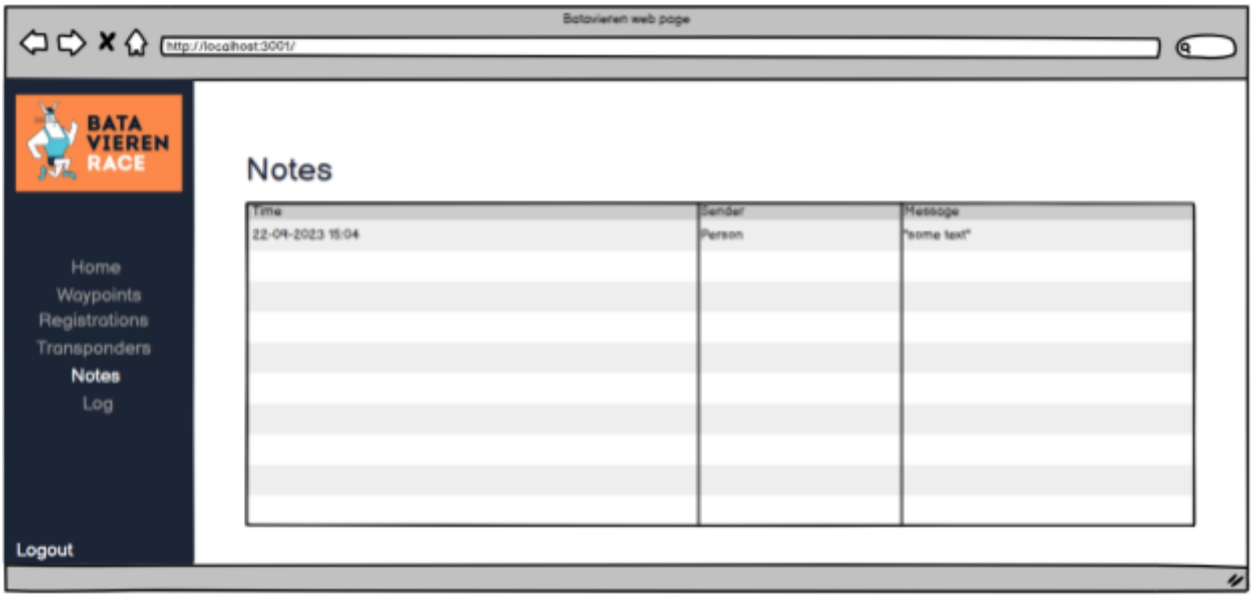

Figure B.9: Mockup of the notes page

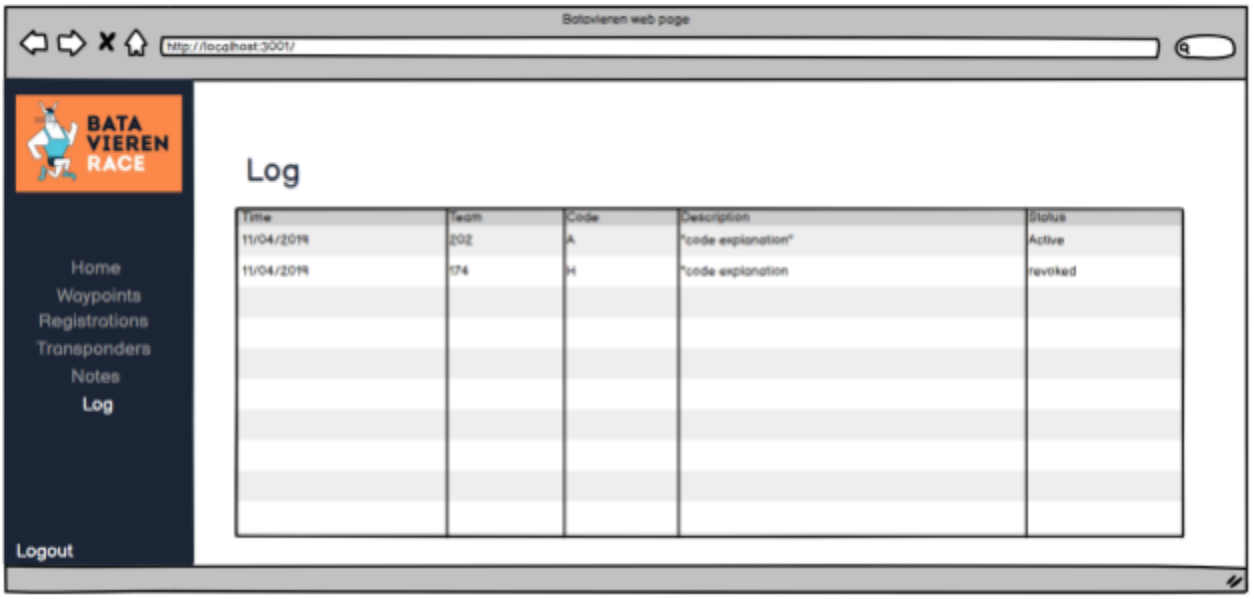

Figure B.10: Mockup of the log page

C. Front-end screenshots

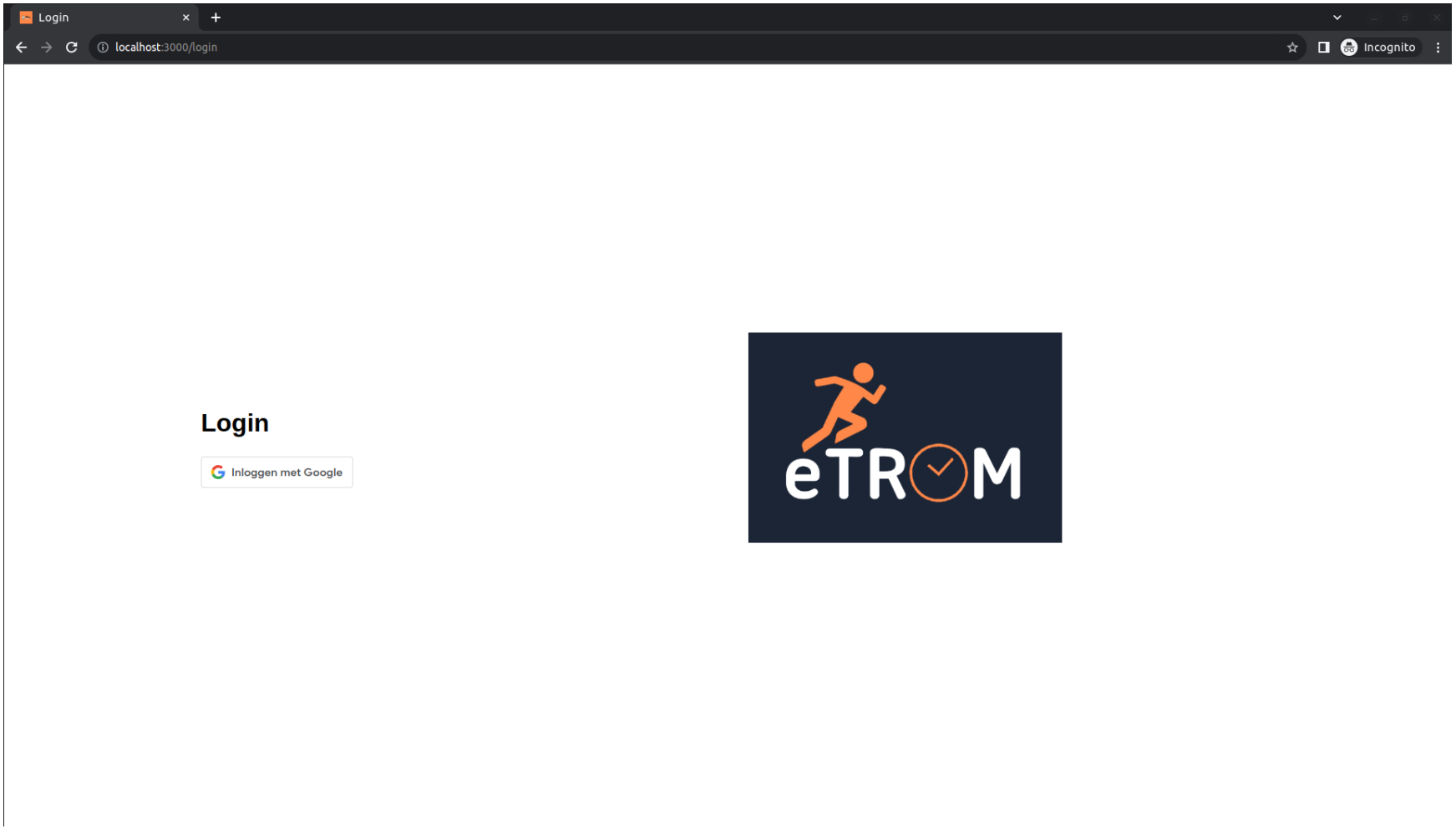

Figure C.1: Screenshot of the login page

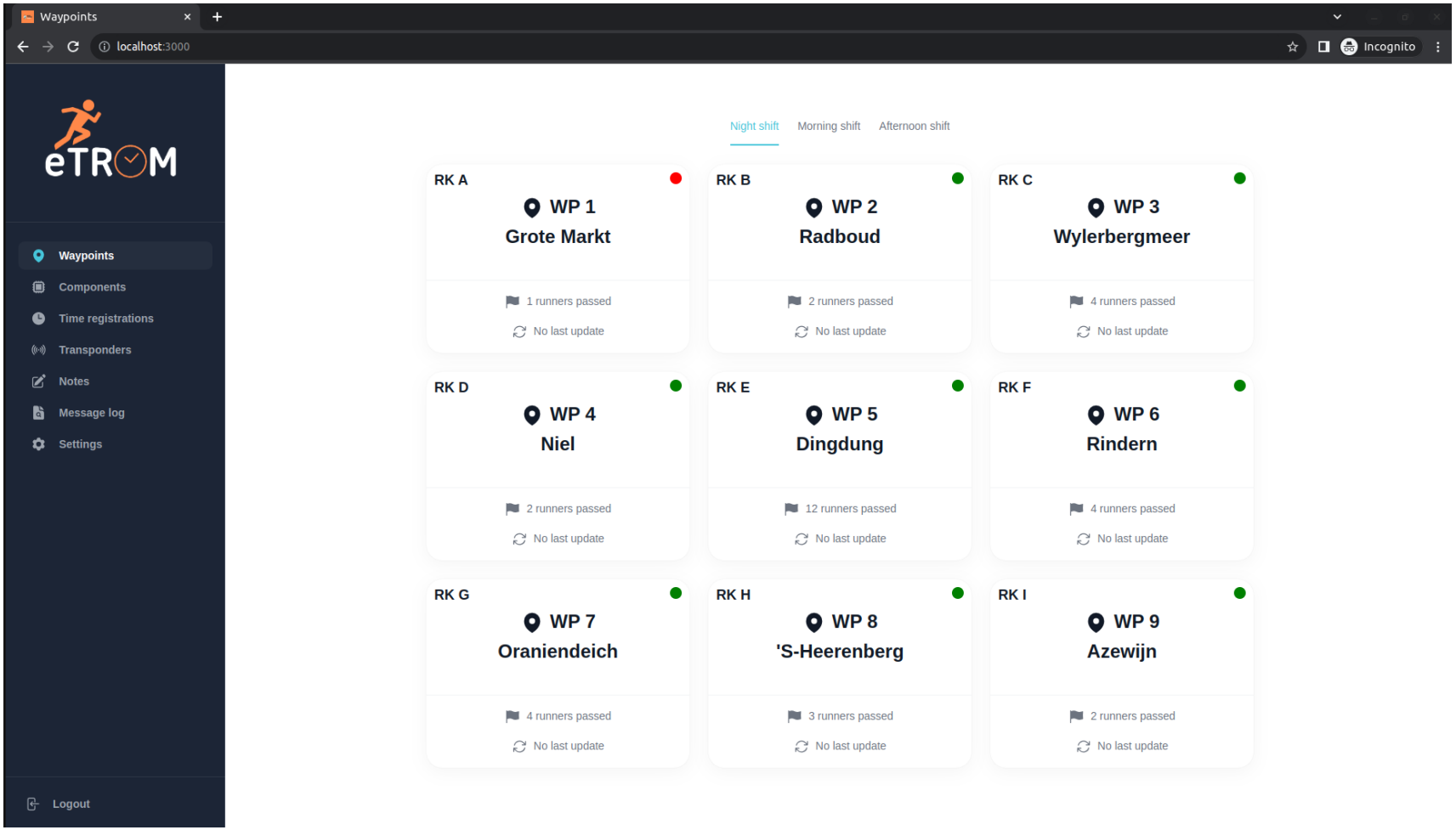

Figure C.2: Screenshot of the homepage (waypoint overview)

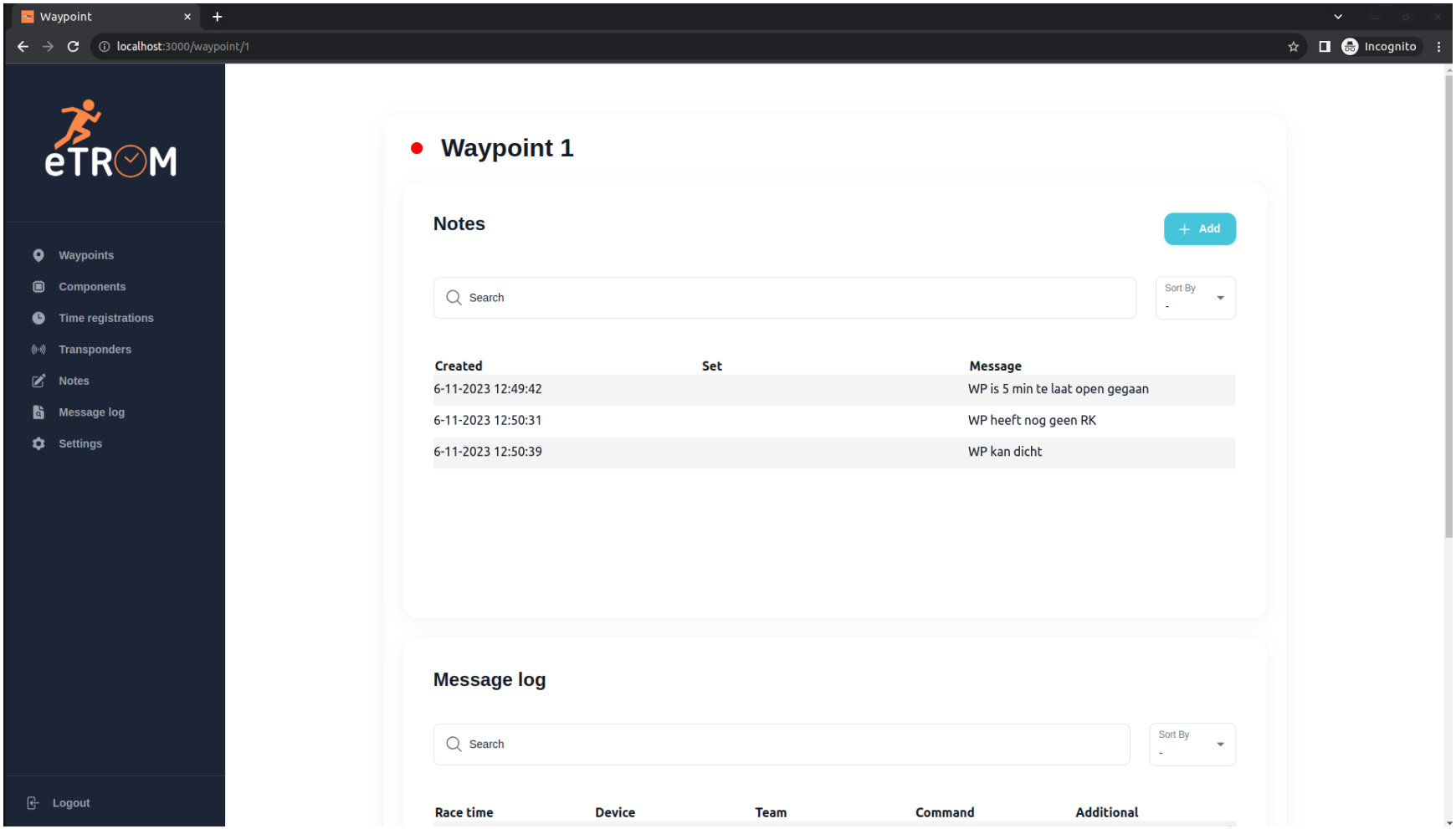

Figure C.3: Screenshot of the waypoint specification page

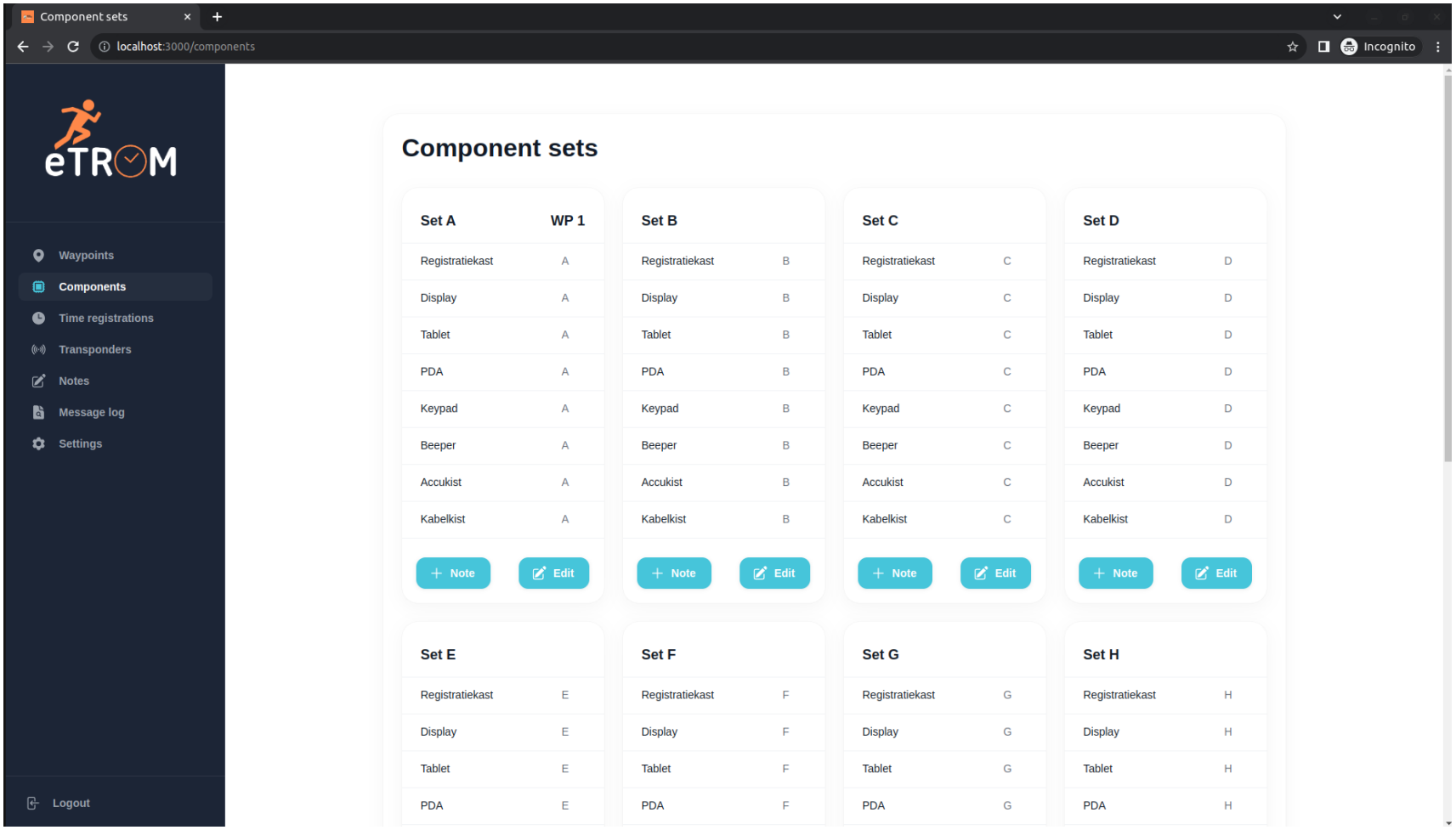

Figure C.4: Screenshot of the components page

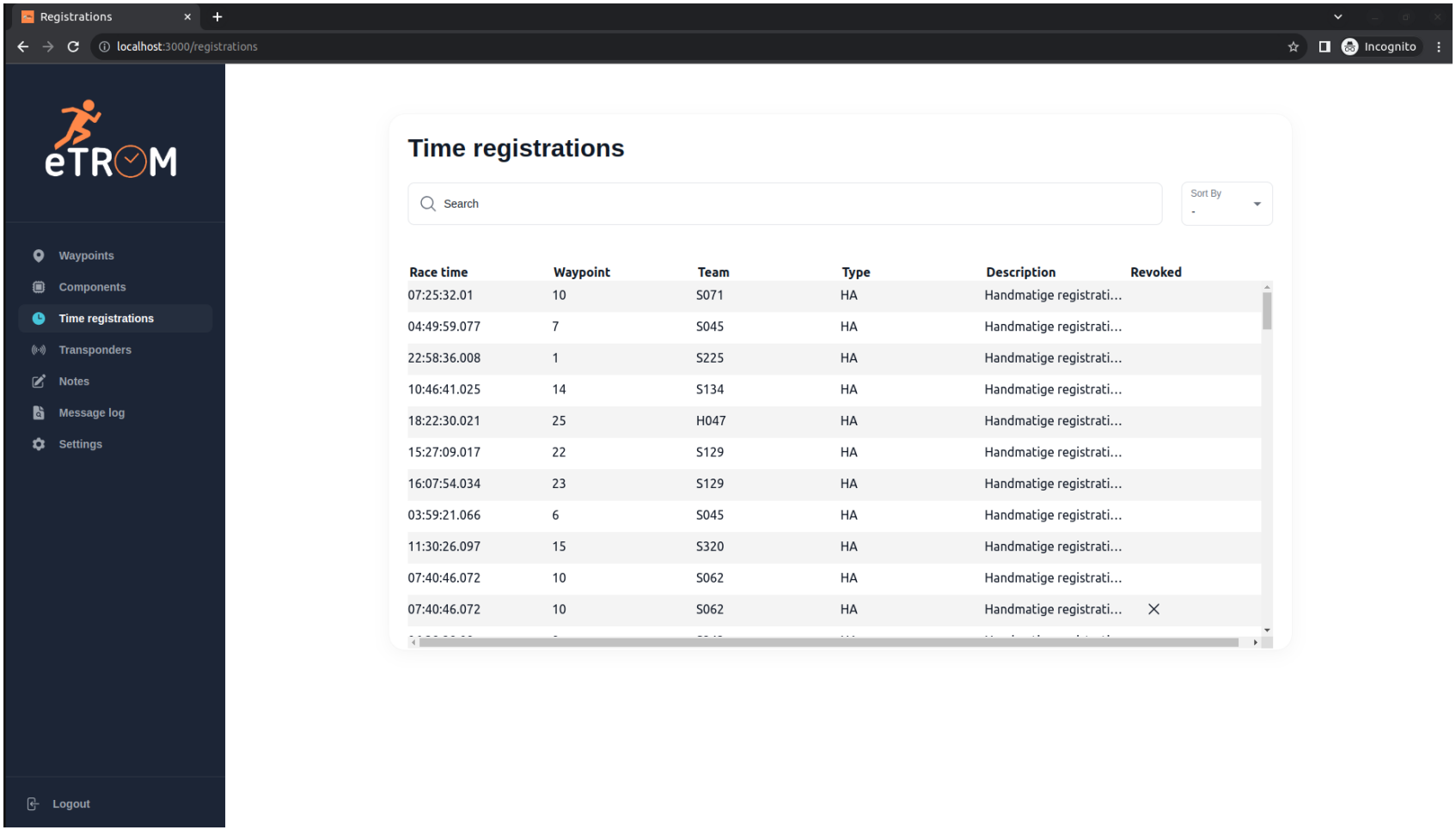

Figure C.5: Screenshot of the time registrations page

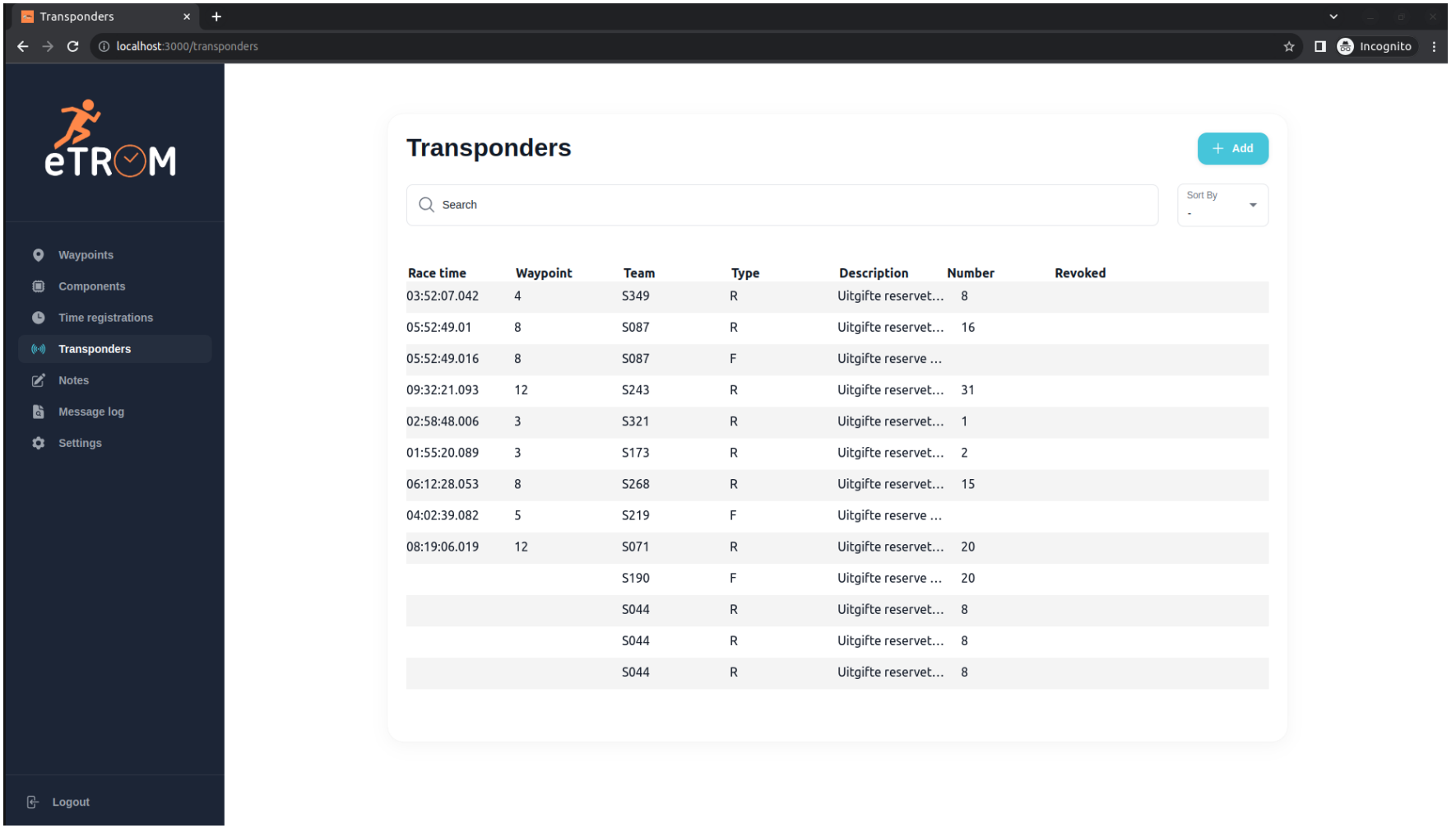

Figure C.6: Screenshot of the transponders page

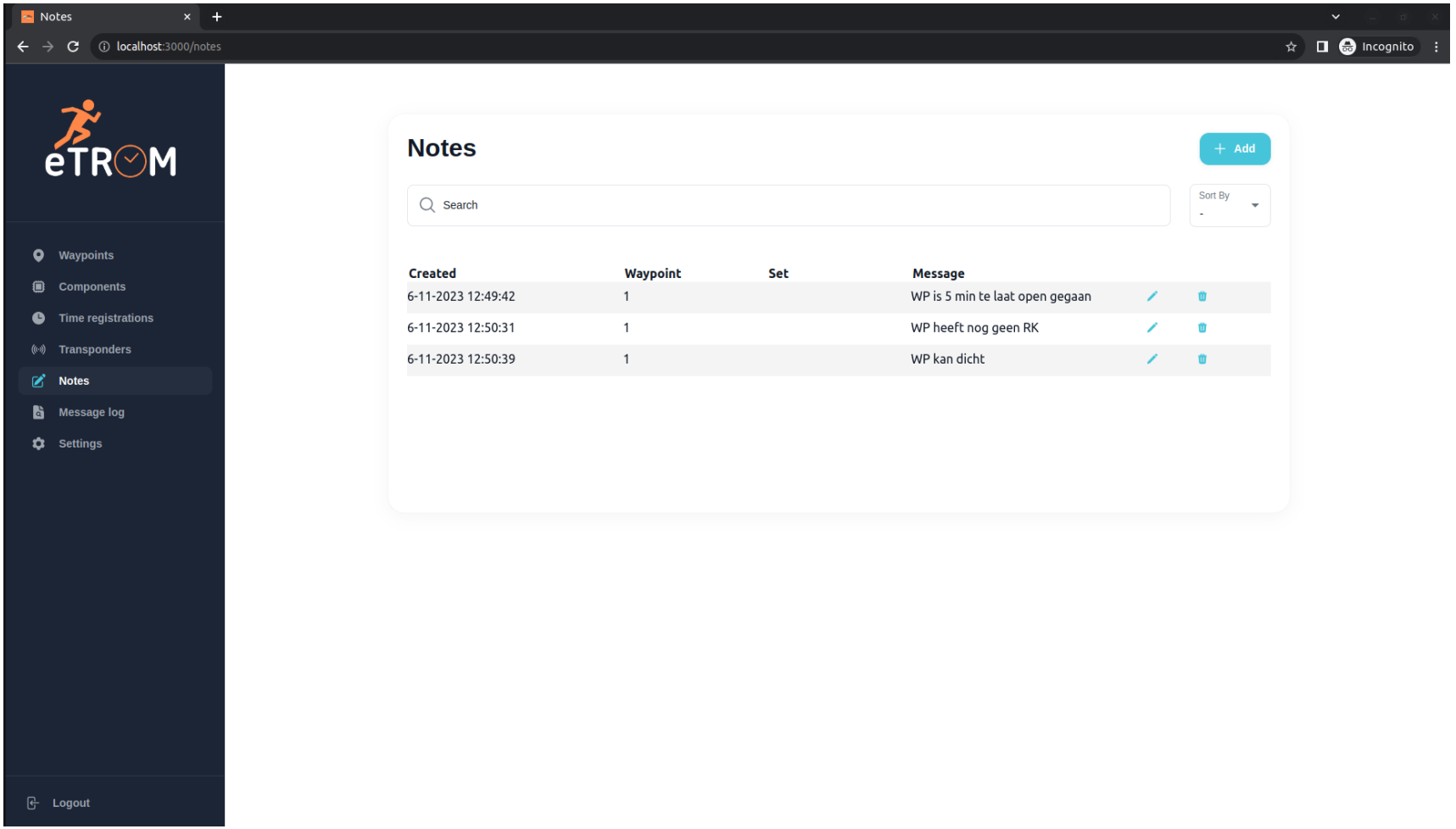

Figure C.7: Screenshot of the notes page

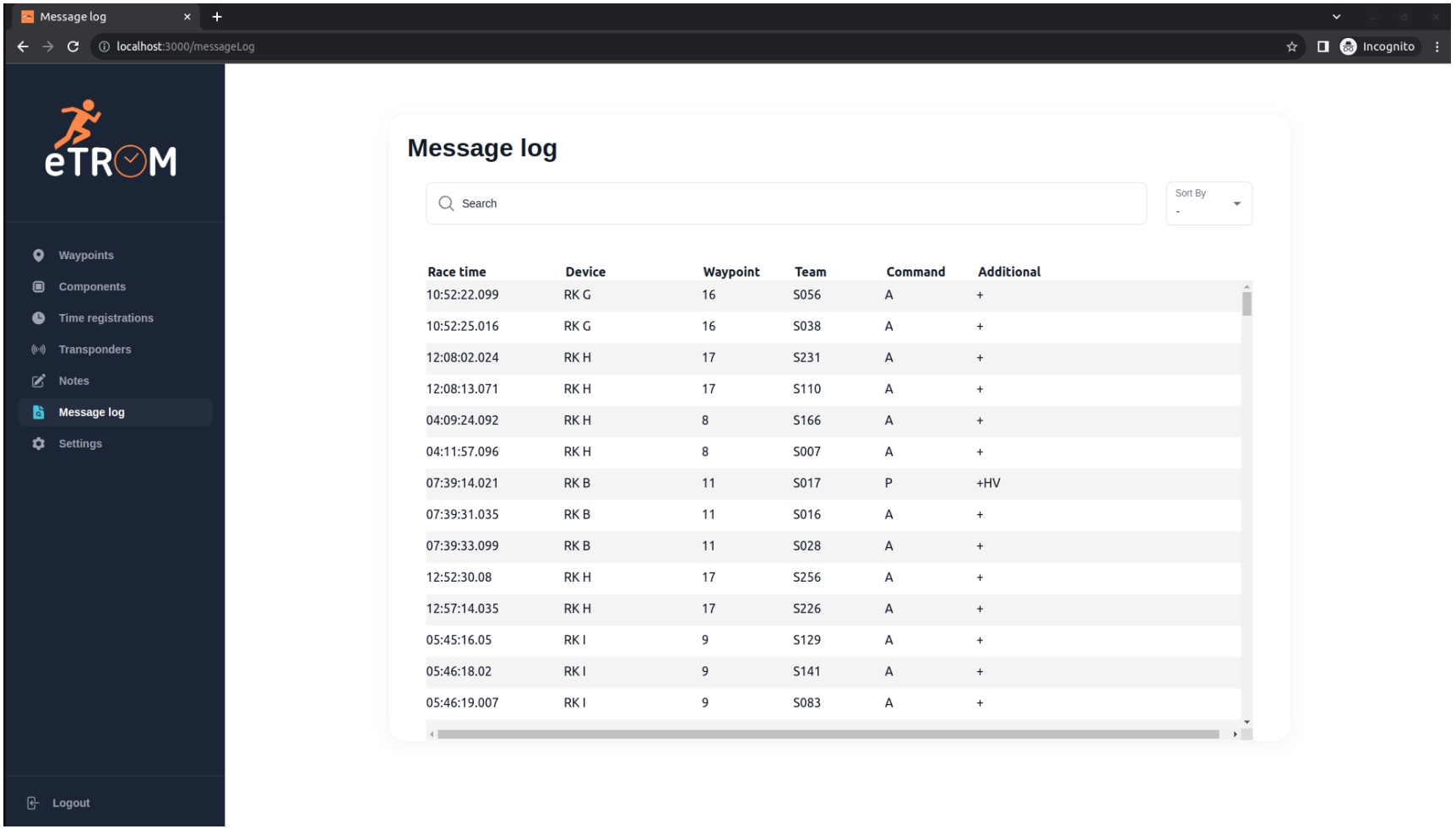

Figure C.8: Screenshot of the message log page

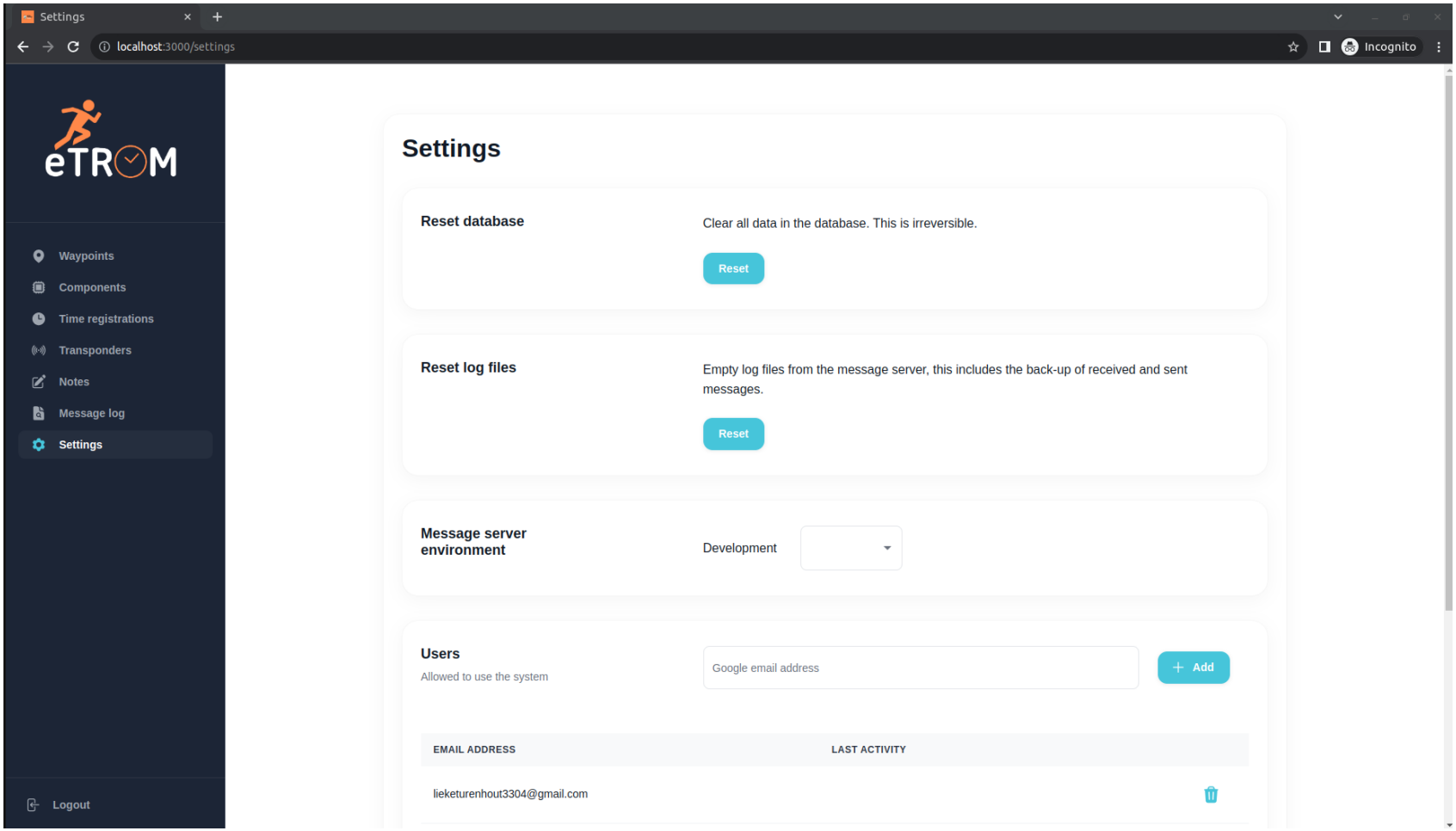

Figure C.9: Screenshot of the top half of the settings page

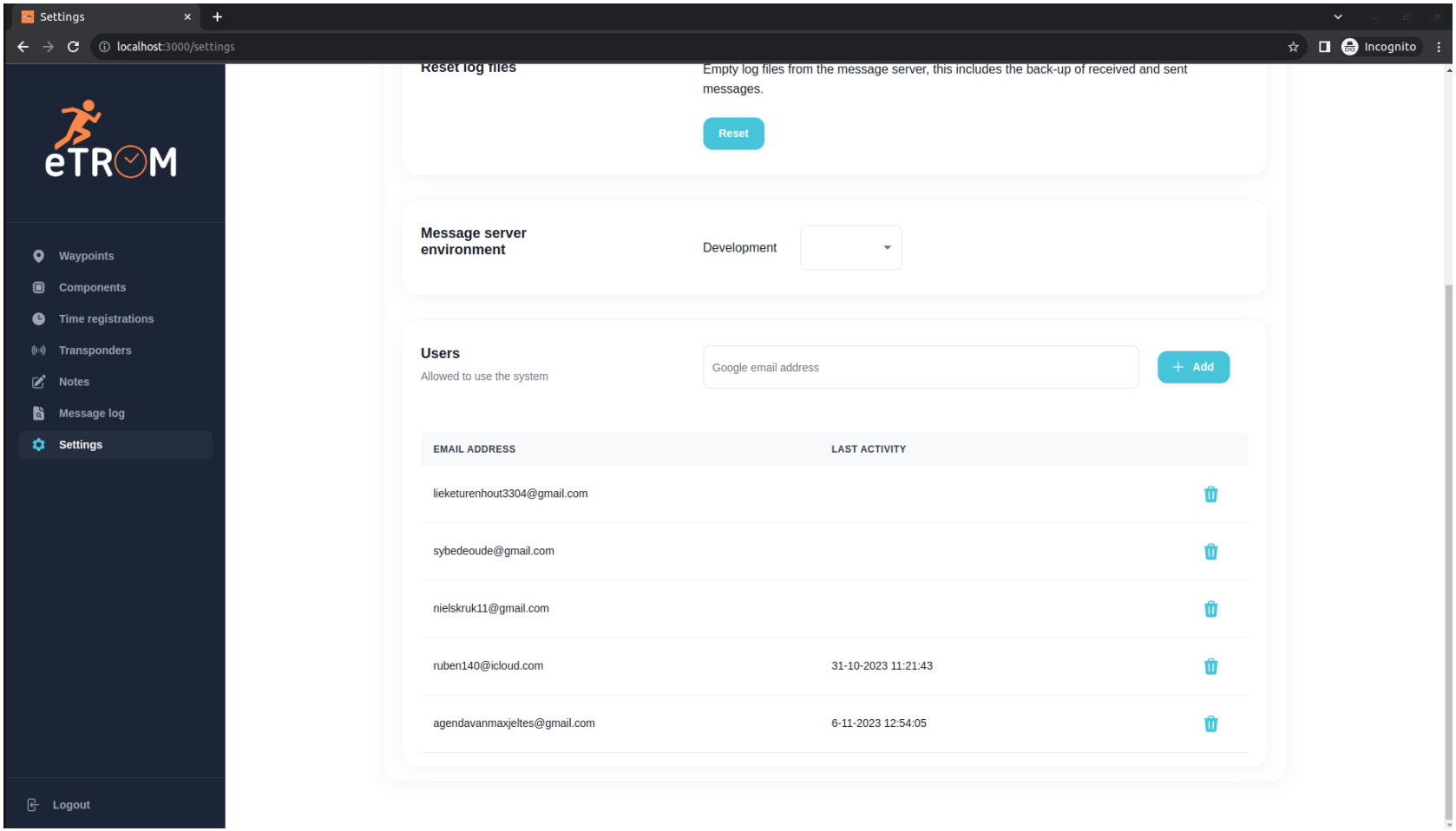

Figure C.10: Screenshot of the bottom half of the settings page

D. Database ER diagram

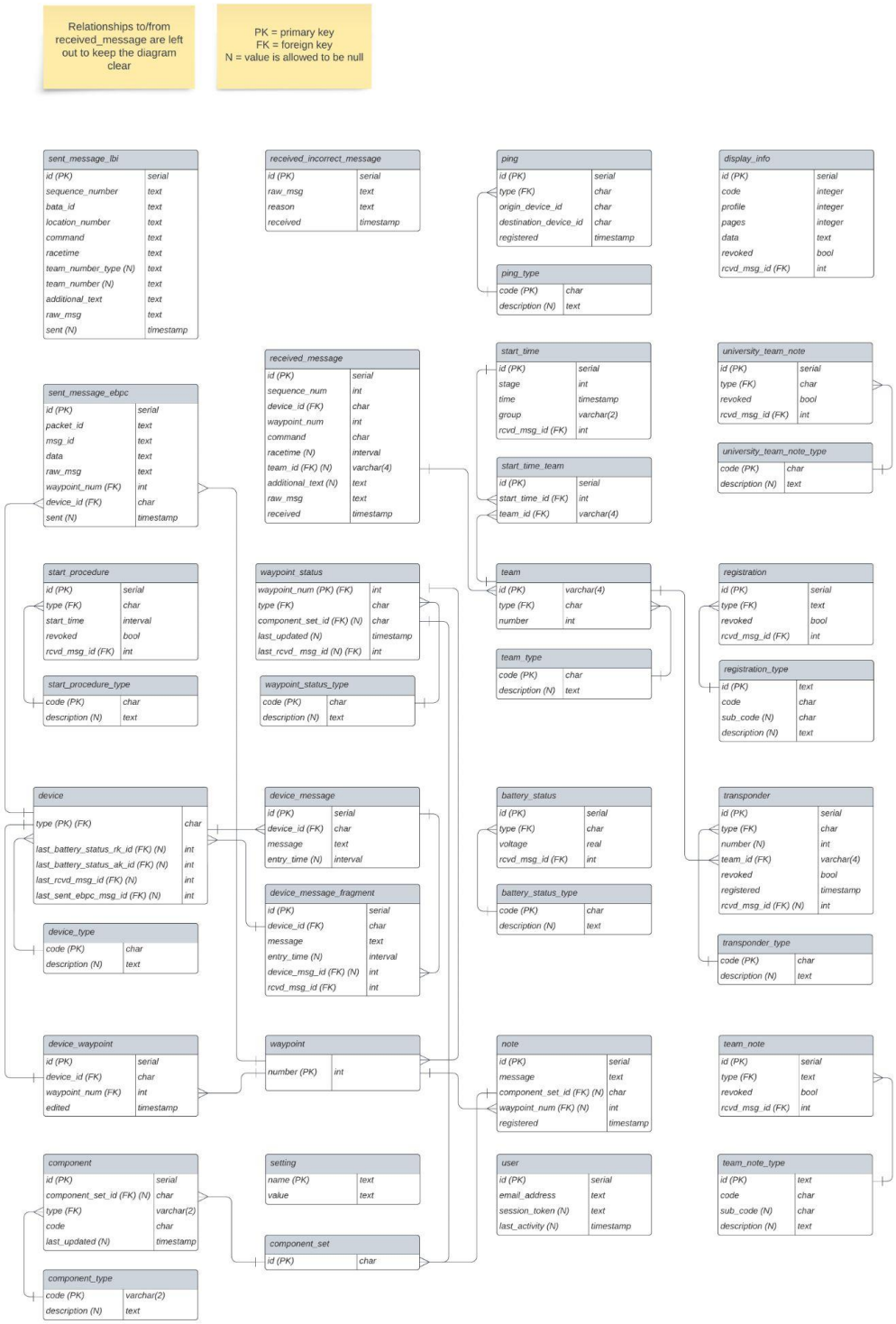

Figure D.1: The database entity relation diagram

E. Message server application state diagram

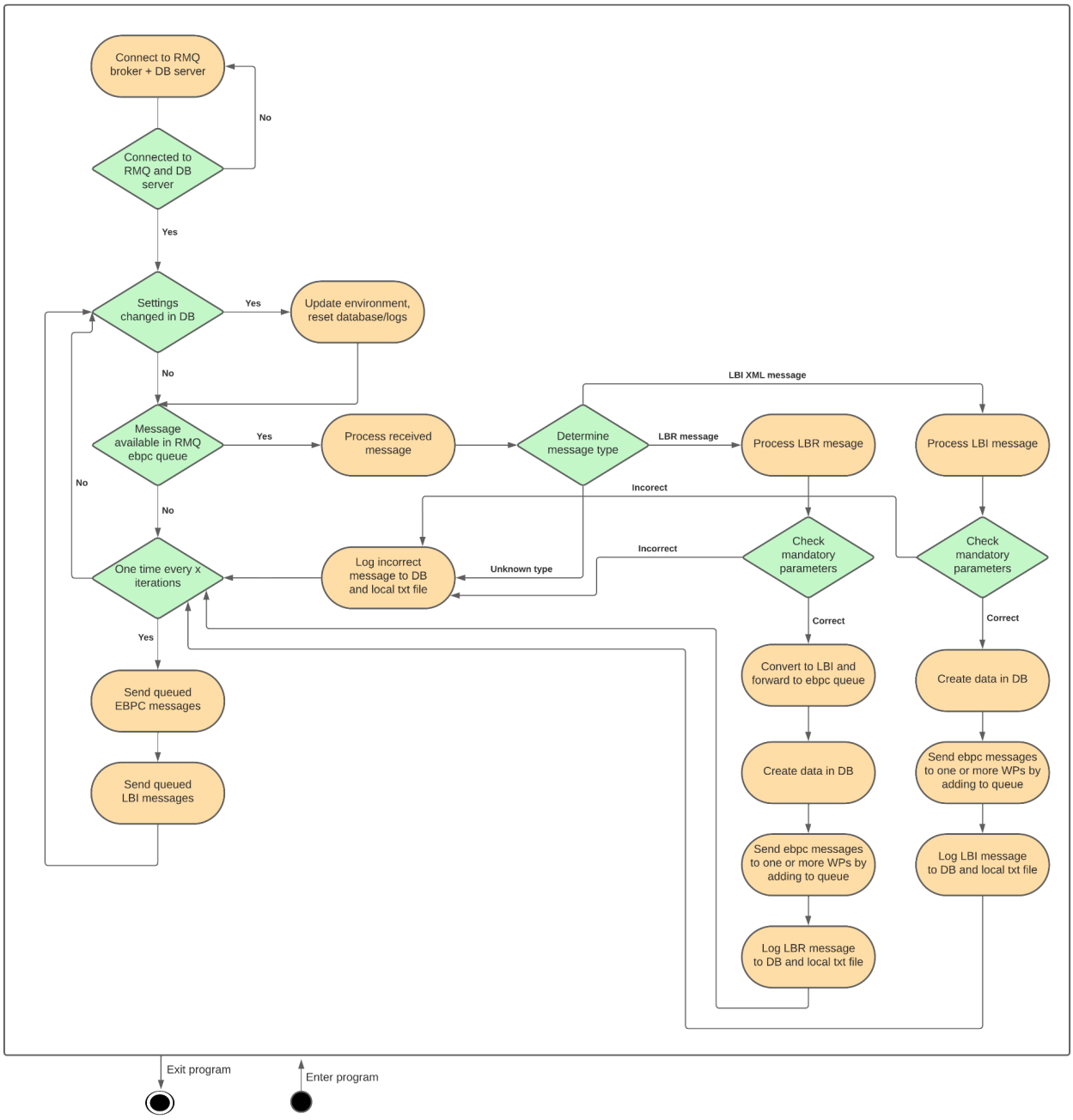

Figure E.1: The message server application state diagram