

University of Manchester
School of Computer Science
Final Year Project Report
Trail Running Website

Author: Stefan Pristoleanu

Supervisor: Dr. David Lester

April 28, 2019

I. Abstract

The aim of this project is to create a website where a user can view a trail (off-road) running route in specific areas, these routes can be provided with details of how long it is, the difficulty, where to park, etc. Once they have run a route they can log their time and compete against other users who have also completed it. A rating system for routes could be implemented so that new runners can determine which routes are the best in a particular area and level. A map with the routes outline should also be provided. If a member runs a new route which is not on the website already then they should be able to create it and upload it to the site for others to try out.

II. Acknowledgements

I would like to thank my supervisor Dr David Lester for all the help and information provided for the whole duration of this project.

Also special thanks to my second marker Gareth Henshall for providing valuable feedback on my take of the project.

Contents

I. Abstract	2
II. Acknowledgements	2
1. Introduction	5
1.1 My idea	5
1.2. Report Structure	6
2. Analysing Project Structure	7
2.1. Types of databases	7
2.2. Scalability:	7
2.3. Availability:	8
2.4. Setting up and access database:	8
2.5. Types of servers	9
2.5.1. SpringBoot using Apache Tomcat	9
2.5.2. Node.JS	10
2.6. Types of Web Servers	11
2.6.1. Arbitrary Web Services	12
2.6.2. REST-compliant Web Services	12
3. Types of Frontend Frameworks	13
3.1. React	13
3.2. AngularJS	13
4. Mobile Application	14
5. Similar Applications	15
6. Overview and analysis of the approach	17
6.1. Database	17
6.2. Backend structure	19
6.2.1. Authentication and user authorisation	23
6.2.2. Third Party Integration	23
6.2.3. Backend Testing & Project Documentation	24
6.3. Frontend Application	26
6.3.1. Frontend Base Functions	27
6.3.2. Adding Trails Manually	28
6.3.3. Automatic Trail Recording	33
6.3.4. Community-focused features	34
7. Mobile Application	37
8. Security	39
9. Ethics	41

10.	Future development	42
11.	Conclusion	43
12.	References	44
13.	Appendix	47
13.1.	Appendix	47
13.2.	Appendix:	47
13.3.	Appendix	48
13.4.	Appendix	49
13.5.	Appendix	50
13.6.	Appendix	51
13.7.	Appendix	51
13.8.	Appendix	52

1. Introduction

Let us start with a little imagination game in order to show the functionality of my third-year project: Trail Running. Pretend that you are a fan of outdoor activities, like walking, hiking, jogging or cycling. You have a favourite route that you always go on, however it is little known, potentially in a remote area through a forest or on a mountain. You want to share your trail to the whole world, but to your surprise Google Maps doesn't quite cover all that area. Or maybe you want to check whether a trail is downhill or uphill. If the area is safe or filled with crime. All of this, and more, can be done using the Trail Running interactive application, which combines two features present in any mobile phone or browser: the ability to determine real-time coordinates through GPS and to send data across the internet to a server where it is held in a secure database.

1.1 My idea

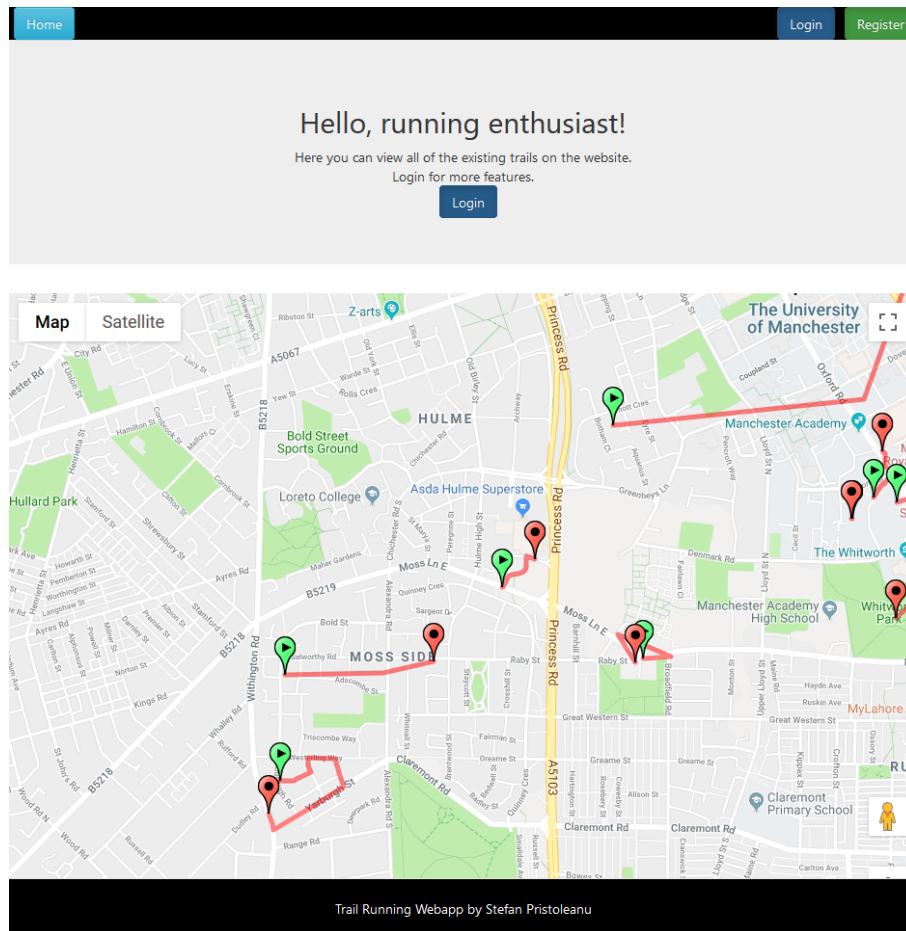
In order to achieve all of this, Trail Running had been scaled up to be a full-stack application consisting of front-end and back-end frameworks, a database as well as a mobile component. Its main purpose is to be an easily accessible tool for outdoor activities fans offering various services and striving to create a long-lasting community behind it.

Essentially, Trail Running is a mobile and web platform for sharing and viewing user-made trails. It has the capability to create off-road routes while maintaining the usual snap-to-road functionality, if the user wants it to. Moreover, it provides a wide range of statistics for each trail, some of which are computed by the application like the distance, elevation and level of safeness. Additionally, a core feature of the project is the ability to record routes in real-time using the GPS, and then upload the newly-created trail to the server. Alternatively, a user can manually create trails on the web platform with maximum accuracy, due to the fact that the chosen library for Google Maps still provides all of the core functionalities such as levels of zoom and Google Street View, which are fully integrated into Trail Running. The community aspect of the application is reinforced by the fact that you can see other user's trails on the map, as well as the specific routes that a user liked and created, respectively.

I believe that I have successfully developed the project in a structured and easily scalable way, so all of the practices described in this paper could be applicable to other similar projects.

Among these, of particular interest are:

- Choosing a database that can manipulate heterogeneous data for a trail's GPS coordinates
- Dealing with communicating with Google Maps API via a HTTPS server
- Improving GPS accuracy for off-road routes
- Porting the web application into a native Android mobile app



1.2. Report Structure

This report discusses the implementation of all of Trail Running's components by comparing different tools for acquiring the best possible result.

Chapter 2 covers the literature survey which inspired me to choose the proposed structure, taking into consideration that similar applications already exist, using different methods and tools. It will follow the most commonly used technologies for a full stack project, and a comparison between the features of three applications from the same domain.

Chapter 3 presents an overview of the approach, together with the requirements that need to be fulfilled in order to successfully complete the project. An analysis is also included, related to the tools selected from the ones proposed in Chapter 2.

Chapter 4 discusses the chosen design for each of the components: the database, back-end, front-end and mobile, each based on the literature survey and the project requirements.

Chapter 5 shows the implementation details and the unexpected facets of my approach. It also focuses on the security and performance trade-off and the ethical aspects of Trail Running.

Chapter 6 covers the goals achieved and the proposed future improvements and features, based on the current limitations.

The final chapter represents the conclusion, with an overview of the whole approach.

2. Analysing Project Structure

Let us start with the way in which we store information. This varies depending on the category of the information: for general queries about users and trails, types of databases have been researched. For storing larger amount of information though, such as the array of coordinates of a route, some other strategies have been considered, with the results presented below.

2.1. Types of databases

The main feature of a database should be the ease of access and its speed to store ratio. With that in mind, two options have been taken into consideration regarding the best database choice: relational and object-oriented.

- Relational Databases:

Databases of this type “have dominated the database world for decades”, according to M. Kofler in his book “The Definitive Guide to MySQL” [\[1\]](#) and are ideal for structuring data into tables with multiple fields. One of the most common open-source databases of this type are MySQL and PostgreSQL, both being object-relational database management systems with an emphasis on extensibility and standards compliance. Between the two, I chose PostgreSQL because it supports indexing JSON data for faster access, unlike MySQL [\[2\]](#). Being ACID-compliant, transactional and using tables with columns and constraints as the core components we would be working with, PostgreSQL proved to be a capable database with all the tools that we would need for our project. That is, apart from solving the problem of efficiently storing a lengthy array of coordinates for each trail, but more on this later.

- Object-oriented Databases:

The object-oriented database MongoDB is the opposite, where data does not have to be split into tables, but rather stored as pairs of key-values, thus not following the structure of the typical relational database. This type is called NoSQL, which stands for “Not only SQL” and is considered useful when a large amount of data is required, as stated in “NoSQL Database: New Era of Databases for Big Data Analytics – Classification, Characteristics and Comparison” [\[3\]](#). But apart from the different amount of data usually stored into these two types of databases, there are other reasons just as important as to why software developers might decide to give up using relational databases in favour of NoSQL, illustrated below.

- Comparison between PostgreSQL and MongoDB

The following features have been taken into consideration while comparing these two types of databases, from the presentation “SQL vs. NoSQL” by M. A. Khan [\[4\]](#).

2.2. Scalability:

This property represents the ability of handling a growing amount of data, which is successfully managed by NoSQL and less accepted by PostgreSQL. The high level of flexibility that MongoDB offers is undoubtedly the main advantage over PostgreSQL, through the fact that any type of data can be easily linked via the key-value system.

2.3. Availability:

No matter the type of project that uses a database, it is always expected that data will be available at any time. This is precisely what MongoDB assures, with an architecture that is distributed so “high availability, horizontal scaling, and geographic distribution are built in and easy to use” as stated on their website [\[5\]](#). Regardless of whether the data is stored in the cloud or on multiple servers, this solution provides continuous availability and less chances of data loss.

2.4. Setting up and access database:

One key advantage of PostgreSQL would be the community and support available. In contrast, setting up a NoSQL database requires a significantly larger amount of work, and is also more difficult to find online support for issues or suggestions. The syntax used with PostgreSQL is clear and simple to understand, while the HashMap-like key-value relationship that predominates the NoSQL represents a lack of standardisation which, according to researchers at Microsoft, “Can cause a problem during migration” as A. Gajani states in his article “The key differences between MySQL and NoSQL DBs” [\[6\]](#).

- Manipulating heterogeneous data for a trail’s GPS coordinates

In our situation, not only does the database need to be scalable and available, but to also operate over hundreds of records in the form of GPS coordinates. In MongoDB’s case this is easily done, as its NoSQL structure facilitates this very thing. However when it comes to the user’s details and a trail’s statistics, MongoDB operates slowly because of its document-style structure, and it is not fitted well for relational-type structures. While in PostgreSQL’s case, the reverse takes place: it is fast when it manipulates the user and trail data, while slightly slower when it comes to the route coordinates in JSON format.

In order to effectively test the two database contenders, I have created a temporary database structure, having only a primary key field and the list of GPS coordinates.

Furthermore, I have developed a tool for comparing the speed of the two databases by creating two projects each having a Java backend but connecting to a different database with similar structure: one with PostgreSQL and one with MongoDB. In PostgreSQL’s case the JSON type was used initially, as it could hold the entirety of the coordinates array. The databases themselves were placed in docker containers on a server and set to operate with the same amount of resources in terms of memory and CPU power.

[The speed test](#), written in Java, is a multi-threaded program which bombards test server connecting to the database with a variable number of requests on a given number of threads. Each request consists of two commands: an insert of new data and an update on said data.

Having ran the speed test on the heavy load between 100000 and 250000 requests, in this specific test the MongoDB project was significantly faster, as seen by the table below:

Database Type	Number of Threads	Number of Method calls	Total test requests	Total time	Requests per sec
MongoDB	20	5000	100000	24.725 sec	4044.49
PostgreSQL	20	5000	100000	44.878 sec	2228.26

After a little investigation I have come to the conclusion that the main problem for PostgreSQL was the JSON type, which is stored in its plain text format, and chose to switch to the JSONB data type,

stored in binary representation instead which uses a GiN (Generalized Inverted Index) for each JSON key. Because of this change, the database column ended up taking slightly more space and increased time to build from its input representation, but it significantly decreased the time it took to operate on the data, as parsing needed to be done after each operation for the JSON data type.

Furthermore, I have updated the test database's structure to be similar to the one of the final version of my project: two tables, one for USERS with some classic fields such as name and email, and one for TRAILS with the more interesting properties of the GPS coordinates and a link to the USERS table through which we can map the trail's owner.

Additionally, I returned to the speed test program and modified the request to include, beside an insert and update, a find by owner id test. This structure simulates better where the strain of the final database would lie – the ebb and flow between the two USERS and TRAILS tables, struggling to display only a selection of trails that belong to a number of users.

I have further optimised both databases with the addition of an index over the trail's owner, which essentially creates a self-balancing binary tree during the insert of data and allows us to search for a user's trails in the complexity $O(h)$. Therefore, in the second batch of tests, we have arrived to the following stats which tipped the balance in favour of the PostgreSQL database, as in the following table:

Database Type	Number of Threads	Number of Method calls	Total test requests	Total time	Requests per sec
MongoDB	20	5000	100000	72.096 sec	1387.04
MongoDB	25	10000	250000	439.494 sec	568.84
PostgreSQL	20	5000	100000	66.165 sec	1511.37
PostgreSQL	25	10000	250000	183.279 sec	1364.04

Based on all of the previous results, I ended up choosing the relational database PostgreSQL with JSONB data type for storing information, situated in a docker container as the database for Trail Running, due to the more familiar syntax and speed, as well as the improved prospect of scalability.

2.5. Types of servers

Continuing our project structure, let us analyse the place where we process the data: the backend of the project. A server is required in order to make the connection between different parts of the program, and it needs to be reliable, fast and secure. Therefore, two different options were taken into account to decide which best fits these attributes.

2.5.1. SpringBoot using Apache Tomcat

SpringBoot became a standard for creating “standalone, production-grade” [\[7\]](#) Java applications. For the embedded application server, in SpringBoot I chose the default option, which is Apache Tomcat and a couple of open-source libraries which will be described in the project structure chapter.

Apache Tomcat is described on the official website as an “open source software implementation of the Java Servlet and JavaServer Pages technologies, intended to be a collaboration of the best-of-breed developers from around the world” [\[8\]](#), and has had numerous releases in the past few years. In the book “Tomcat: The Definitive Guide”, J. Brittain and I. F. Darwin [\[9\]](#) illustrate the idea behind the name Tomcat: created by James D. Davinson, he chose the name of the animal that was “self-sufficient”, just like the server, and easy to be taken care of.

Some of the advantages of using Apache Tomcat are that a diverse range of organisations and industries use it, with more than 10 million downloads, as stated on their official website. This solid following is due to Tomcat being open-source, scalable and supporting a cluster architecture in order to have multiple instances running at the same time, if need be.

2.5.2. Node.JS

Another option for the project's server would be Node.JS which, according to the official website, "is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications." [10] Through the use of asynchronous I/O, Node.JS is significantly faster than other types of servers due to the fact that it is capable of performing more than one task at a time, which is "a key requirement for real-time apps", as M. Cantelon stated in his book "Node.js in Action" [11].

In the same publication we are presented with the types of application Node.JS is ultimately designed for: DIRTy applications or "data-intensive real-time" apps. Node.JS allows holding multiple connections opened for handling more than one request, and this could prove a useful feature for a mobile application used by multiple people simultaneously.

Having the Trail Running project in mind, I initially thought this would be exactly what was required. However, after a bit of research and planning I realised there was no particular need for the project to be a DIRTy application, as all of the modules that could put a heavy load onto the server, namely the recording of a new trail, would be computed in the input device, with a single final call towards the server to upload it.

In order to choose between the two technologies I created two projects: one in [Java SpringBoot](#) and one in [Node.JS](#), both being connected to the same PostgreSQL testing database that yielded the better results in the earlier phase. In doing so I got accustomed to the way Node.JS worked, being pleasantly surprised with the fact that "Node supports the eventing model at the language level", as noted in S. Tilkov's book on the matter [12], where other programming languages rely on external libraries to achieve this.

I recognised the Model View Controller architecture being the most applicable to the Trail Running project, where objects of different classes take over the operations related to the application domain, the display of the application's state, and the user interaction with the model and the view. This early decision was beneficial as otherwise it would have been hard to incorporate this design pattern in later stages of the project, as stated in A. Leff et al in "Web-application development using the Model/View/Controller design pattern" [13]: "Applying the Model/View/Controller design pattern to web-applications is therefore complicated by the fact that current technologies encourage developers to partition the application as early as in the design phase."

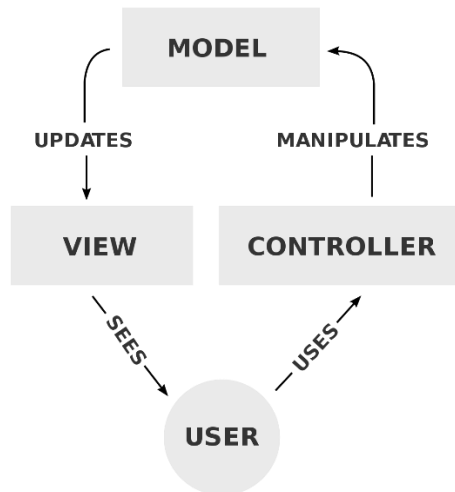


Figure2: The Model-View-Controller Design Pattern

Having more experience in Java, this was relatively easily done using the standard Entity-Repository-Controller structure. However, when it came to Node.JS more research had to be done, and so I came across node-postgres [\[14\]](#). It is a collection of Node.JS modules for interfacing with a PostgreSQL database, which was exactly what we needed. Soon enough a similar structure was achieved in both projects and the speed test could begin.

The tool used in this situation was the same which compared the two databases earlier, however now that the database was identical, the difference would lie in the technology used for the server. Same as before, it simulates the final structure of Trail Running with requests comprising of an insert, update and bind by user id. After extensive testing under heavy load, the following data emerged:

Server Type	Database Type	Number of Threads	Number of Method calls	Total test requests	Total time	Requests per sec
SpringBoot	PostgreSQL	20	5000	100000	66.165 sec	1511.37
SpringBoot	PostgreSQL	25	10000	250000	183.279 sec	1364.04
Node.JS	PostgreSQL	20	5000	100000	104.261 sec	959.13
Node.JS	PostgreSQL	25	10000	250000	259.511 sec	963.35

In the end, I chose Java SpringBoot as a backend framework for Trail Running, not only being the faster of the two but also due to having a relational database, in which Java specialises. If it were a NoSQL database, Node.JS would work easier on it with json-like document stores. Furthermore, there are some security aspects such as authentication tokens which are easily done using SpringBoot, and would require expensive manual implementation in Node.JS.

2.6. Types of Web Servers

The server itself would be built as a web service. In their book called “Web Services”, G. Alonso et al [\[15\]](#) state that these are “self-contained, modular business applications that have opened, internet-oriented, standards-based interfaces”, or more specifically according to the World Wide Web Consortium definition “a software application identified by an URI, whose interfaces and bindings are capable of being defined, described and discovered as XML artefacts” [\[16\]](#). In addition, it is explained

that a web service provides direct interaction through XML-based messages with other software agents, transmitted via Internet-based protocols.

There are two major classes of Web services that can be identified, each presented below.

2.6.1. Arbitrary Web Services

In this case, service may expose some arbitrary operations using a Web Services Description Language (WSDL). WSDL describes services as collections of network endpoints, used together with SOAP and an XML Schema to provide Web services across the Internet. SOAP, or Simple Object Access Protocol “provides a way to communicate between applications running on different operating systems”, but it is stated that a better way of achieving this is using HTTP according to www.w3schools.com.

In short, the operations available on the server can be determined by such a web service using a WSDL file, which contains an XML with the specifications of each datatype. SOAP can then be used to call over HTTP one of these datatypes and create the connection.

2.6.2. REST-compliant Web Services

For this type of services, no client context is stored on the server between two requests, as a set of stateless operations is used to achieve the connection. Therefore, the operation is done smoother than with an arbitrary web service. One type in particular was reviewed in more detail, the RESTful web service.

In an article on the Oracle page, entitled “RESTful Web Services”, S. Tyagi states that “In the web services world, Representational State Transfer (REST) is a key design idiom that embraces a stateless client-server architecture in which the web services are viewed as resources that can be identified by their URLs” [17].

Thus, the connection between the different components of a system can be done using RESTful in an easy and intuitive way, just by giving an URL with parameters. These are then processed accordingly, and the result is returned through the web service.

3. Types of Frontend Frameworks

For the front-end part of the Trail Running project, I decided that I was going to need a powerful JavaScript framework that supports the integration of the maps services, all the while being able to create a responsive and intuitive design for the web application, which would later be ported into the mobile component. After some research I have narrowed my search to two well-established frameworks, both of which could easily achieve this.

3.1. React

React is a relatively new JavaScript framework initially developed by engineers at Facebook as a solution to the complex user interfaces with datasets that change over time. It is fundamentally different from the other frameworks, featuring one-way data binding with props, stateful components and lifecycle methods. As emphasised in C. Gackenhaimer's book, "Introduction to React", "When React was released in 2013, the web development community was both interested and seemingly disgusted by what React was doing" [\[18\]](#).

In the end, it has received praise for its rigorous structure but is ultimately a simple, yet effective JavaScript framework for building user interfaces. Because of that, complex React applications require the implementation of additional libraries for state management, routing and interaction with APIs.

3.2. AngularJS

AngularJS is thought of as the world's most popular model-view-wildcard framework. It has the backing of Google as well as the many developers who joined and are using it, providing great support in terms of tutorials. It is a heavy weighing framework, with a more declarative nature in its templating. It has concepts like controllers, directives and services which are linked to the application developed. Dr. A. Bhansali keenly notes that "AngularJS is built around the philosophy that declarative code is better than imperative code while building UIs and wiring different components of web applications together" [\[19\]](#).

A large difference between React and AngularJS is one-way vs. two-way binding when it comes to events and properties. As a project becomes larger, React's binding simpler system results in a better data overview, thus making debugging much easier too. Both concepts have their advantages and disadvantages.

In the end, I chose React mainly because my rate of development in this frontend framework was better than in AngularJS, which has many additional features which I would not have used. I could not agree more with A. Fedosejev's uplifting statement in his book "React.js Essentials": "Get ready to be surprised by the simplicity, predictability, and thoughtfulness of React.js" [\[20\]](#).

4. Mobile Application

It was an ambition of mine to extend the functionality of the Trail Running project to the Android mobile platform. However, writing code in Android Studio would have taken precious time, and in the end the mobile app does not have to be all that different from the web browser one.

This is where Apache Cordova comes into play. It is an open source framework in which one can develop a native mobile app, or alternatively through which a web developer can deploy a web app that is packaged for distribution, meaning it is in working order and fully operational. From their website, they advertise the fact that Apache Cordova “allows you to use standard web technologies - HTML5, CSS3, and JavaScript for cross-platform development. Applications execute within wrappers targeted to each platform, and rely on standards-compliant API bindings to access each device's capabilities such as sensors, data, network status, etc.” [\[21\]](#)

It essentially would wrap around my web application and use the mobile phone's browser core in order to render the app as a standalone native Android app, with access to the GPS and all of the other features of the phone, if need be.

5. Similar Applications

There are applications on nearly every domain imaginable, and the exercising area is no different. Three such apps, similar in scope with aspects of Trail Running, stood out:

- [Mapometer](#) – a tool for planning routes

Mapometer is an online map-based route planner, where one can map out runs and calculate the distance and elevation profile of a route. It inspired me to think about the type of routes, as one could select from different categories like running or swimming. Another useful element was the toggleable option of following roads, in case one were cycling in the city or trekking on a mountain.

- [UKClimber](#) – the climbing equivalent app

UKClimbing is a complex website which handles news and reviews on climbing, climbing gear and their community in general. They also have a system in place for finding new climbing spots, as well as viewing pictures, reviews and other user's performance on them.

Despite the differences between climbing spots and trails, this has been helpful in terms of the layout of the map, the representation of spots as well as the introduction of various additional features such as weather in the area and a corresponding gallery of photos, which motivated me to implement similar features in Trail Running.

However, the lack of recording or tracking a new climbing spot in real time convinced me to look at other options, too.

- [Strava](#) – the intersection of social media and exercise

Perhaps the best example of a successful exercise app is Strava, a fitness application that tracks your activities via your smartphone. It is self-described as a “social network for athletes” and in their annual report, Strava boasts 36 million users from 195 different countries worldwide, having logged 6.67 billion miles from September 2017 through August 2018. The data showed an average of 15 million activities uploaded per week, and this convinced me to scrutinize the application in order to gather information regarding any outstanding feature.

Its detailed route statistics were something I strived towards, going beyond the usual distance and pace to include competing times between athletes and the introduction of splits and a liking system. It also features a “Segment Explore” section, which takes an area and displays all of the user-uploaded trails in that region, on which my project focuses and tries to improve.

The trail recording was good for the most part, as it warned me of the potential inaccuracies of the GPS and determined me to think of a solution.

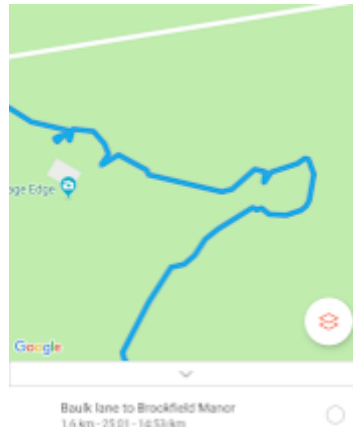


Figure 3: Strava off-road GPS example

6. Overview and analysis of the approach

This chapter presents the overview of the required system, together with an analysis regarding which tool would be the best for this project, from those reviewed beforehand.

System requirements can be split into two categories: functional and non-functional. The former refers to the specific system's behaviour, while the latter explains how those results are achieved. They are mainly related to the quality and documentation of work and have been kept in mind over the course of the project. Furthermore, system requirements can be split into three levels of priority:

- M – mandatory
- D – desired
- O – optional

Task Id	Description	Priority
1.	Database	
1.1	Database Table SQL	M
1.2	Heterogeneous Data Handling	M
1.3	Relational Links & Index Optimisation	O

Development started soon after the research phase, having adopted the Agile Model as a way to deliver content in a fast and flexible manner. My preferred version control system was Git, which allowed me to manage the tasks of my project, as well as keep track of the various iterations of Trail Running and the supporting projects developed for testing purposes. I have also emphasised documenting every command needed to replicate the steps explained here, which can be found in README.md files in the [GitHub project](#).

The following section will cover the system requirements, explaining their development.

6.1. Database

First things first, the PostgreSQL database structure was made. It contains two tables USERS and TRAILS, connected by the fact that every trail has an owner user, which is invaluable when it comes to finding specific information such as a user's trails. In order to achieve this, [DBeaver](#) was used: a free multi-platform database tool for developers, SQL programmers, database administrators and analysts, as stated on their website. It supports all the popular databases including PostgreSQL and provided me with invaluable tools for SQL insertion and data management.

The database UML diagram (Figure 3) shows the practices explained earlier for managing heterogeneous data, namely the JSONb data_json field which contains all of the trail's GPS coordinates. The index optimisation can be seen in the create table SQL (Figure 4), as well as the later addition of a JSON type field: LIKED_TRAILS. For this field it would be more appropriate to use JSON instead of JSONB as this way we can preserve a user's chronological order of likes. We have also established a foreign key one-to-many relation between the trail_created_by and user_id.

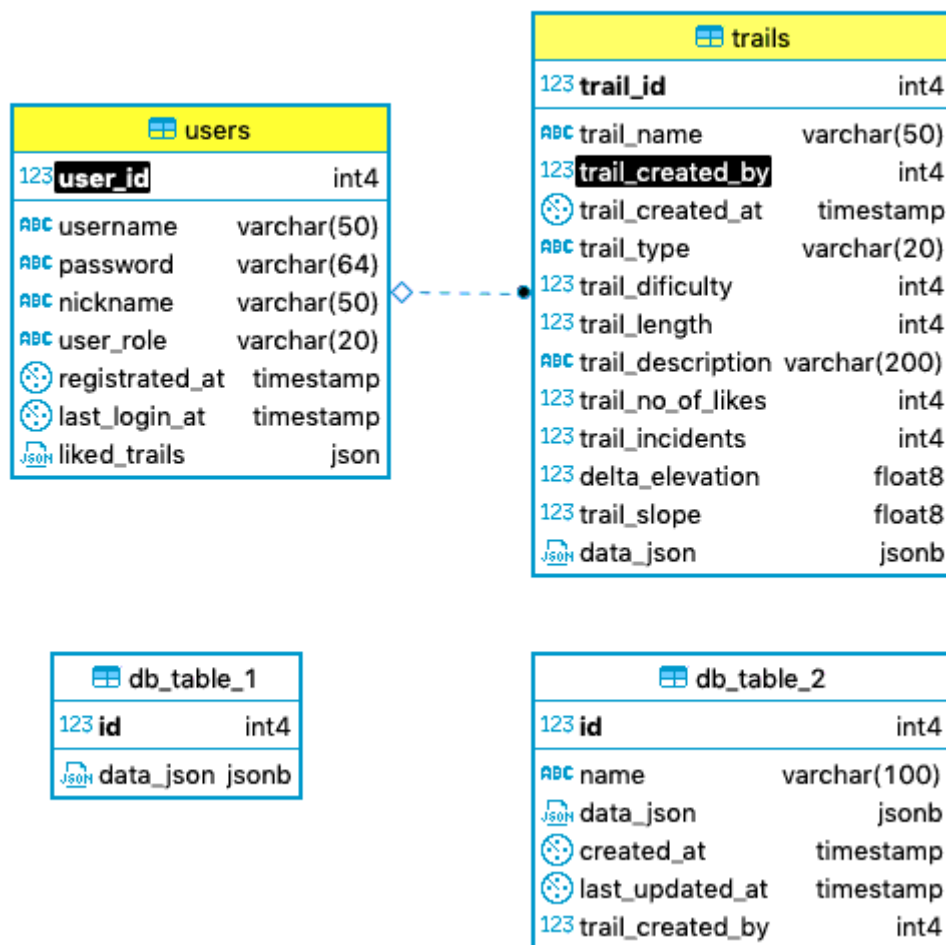


Figure3: Trail Running database structure & test tables

```
CREATE TABLE USERS (
    USER_ID SERIAL, --an auto incremented field for primary key
    USERNAME VARCHAR(50) NOT NULL UNIQUE, -- user's email address
    PASSWORD VARCHAR(64) NOT NULL, -- PASSWORD: min 8 and max password chars:
24 and it will be encryped one-way SHA in 64 chars
    NICKNAME VARCHAR(50) NOT NULL,
    USER_ROLE VARCHAR(20) NOT NULL, --default all registration users are with
role USER
    REGISTRATED_AT TIMESTAMP NOT NULL,
    LAST_LOGIN_AT TIMESTAMP NOT NULL,
    LIKED_TRAILS JSON, --Using JSON instead of JSONB in order to keep the
chronological order of likes
    CONSTRAINT users_pk PRIMARY KEY (USER_ID));

CREATE TABLE TRAILS (
    --an auto incremented field for primary key
    TRAIL_ID SERIAL,
    TRAIL_NAME VARCHAR(50) NOT NULL,
```

```

TRAIL_CREATED_BY INTEGER, -- = USER_ID from TABLE_USERS
TRAIL_CREATED_AT TIMESTAMP NOT NULL,
TRAIL_TYPE VARCHAR(20), --bicycle, by foot
TRAIL_DIFICULTY INTEGER,
TRAIL_LENGTH INTEGER,
TRAIL_DESCRIPTION VARCHAR(200),
TRAIL_NO_OF_LIKES INTEGER,
TRAIL_INCIDENTS INTEGER,
DELTA_ELEVATION DOUBLE PRECISION,
TRAIL_SLOPE DOUBLE PRECISION,
DATA_JSON JSONB,
CONSTRAINT trails_pk PRIMARY KEY (TRAIL_ID));
CREATE INDEX TRAILS_GIN_DATA ON TRAILS USING GIN (DATA_JSON);

ALTER TABLE trails
ADD CONSTRAINT owner_id_fk FOREIGN KEY (trail_created_by) REFERENCES users
(user_id);

```

Figure 4: SQL Commands for creating the Database Structure

6.2. Backend structure

Task Id	Description	Priority
2.	Server backend	
2.1	Connect to the database	M
2.2	Create JPA entity and repository	M
2.3	Create RESTful controller classes	M
2.4	Integrate Spring Security	M
2.5	Integrate Spring Swagger UI	D
2.6	Query Police Data API for nearby crimes	O
2.7	API ready for third party use	D
2.8	Unit Testing	M
2.9	Integration Testing	D

Development on the backend started with taking the previously created project featuring Java Springboot with PostgreSQL and expanding on it. The connection to the database was setup through Spring's application.properties file, in which we placed the database URL, username, password and port.

SpringBoot allowed us to easily integrate useful libraries using Apache Maven, a software project management tool:

- Spring Security [\[22\]](#) & JWT – handled roles, authentication and authorization of users when trying to access most controller methods. I used the technique of having the login method return a JWT which is placed in the header of every subsequent request to the server, and without which the methods are unavailable. [\[23\]](#)

- Spring Data JPA – gives us the luxury of dynamic query execution and the ability to integrate custom data access code. This significantly improves the implementation of data access layers by providing automatic wiring to repository interfaces, including custom finder methods, which Trail Running uses plenty of. [\[24\]](#)
- [Unirest](#) – a lightweight HTTP Request Client Library for communicating effortlessly with APIs. With only a few lines of code we are able to connect to the UK’s police data API, forwarding the trail’s coordinates to receive an array of several crimes from its immediate vicinity. This array with data could be manipulated in many ways, and ultimately we display the number of crimes to the user.
- [SwaggerUI](#) – an open source software that facilitates the documentation of RESTful controllers, by providing working links to all of the controller’s functions, which can be easily tested on the web

The Model-View-Controller architecture came to fruition, linking the two database tables to their corresponding entities: UserEntity and TrailEntity. Each database field is a variable in the entity class ([Appendix 1](#)) and the specialised constructor takes a Data Transfer Object to populate the non-changeable fields such as user role or last login date ([Appendix 2](#)).

Furthermore, the entity classes are linked to their respective JPA repositories. These classes extend CrudRepository which enables actions like save, delete, find and count data in the respective database table. [\[25\]](#) By extending them, Spring Data JPA allows us to insert custom SQL native queries in order to make more complex operations, like finding trails by their user id.

Sample of using native query:

```
@Query(value = "SELECT * FROM TRAILS t WHERE t.TRAIL_CREATED_BY = ?1 ",
nativeQuery = true)
public List<TrailEntity> findById(Long userId);
```

The final piece of the puzzle is the controller class, which uses a repository to create various API functions. In our case, there are two Spring RestControllers, one for users and one for trails (Figure 5). The visual representation of these is due to the implementation of SwaggerUI, whose usefulness in testing made it worth the effort.

The controller classes are linked to the JPA repository through the “@Autowired” annotation, which provides a more fine-grained control over where and how autowiring should be accomplished. Each of the two controllers use the “@RequestMapping” annotation to map web requests onto methods in request-handling classes with flexible method signatures. [\[26\]](#)

UserController and TrailsController contain various RESTful API methods, each with its own annotation “@PostMapping” or “@GetMapping” based on their business logic and presented below.

Trail Running RestAPI ^{0.1}

[Base URL: sa306.saturn.fastwebserver.de:9090/]

<http://sa306.saturn.fastwebserver.de:9090/v2/api-docs>

This application display the methods for user and trail controllers.

Authorize



Trail management Trails Controller



POST /api-server/trail/add Add a specific trail



POST /api-server/trail/delete Delete a specific trail



GET /api-server/trail/find-all Retrieve all trails. It is public function displayed on the home page.

POST /api-server/trail/like Like an existing trail



GET /api-server/trail/trail-details Find a specific user's certain trail



POST /api-server/trail/update Update an existing trail by the trail owner



GET /api-server/trail/user-trails Find a specific user's trails



User management Users Controller



GET /api-server/user/details User Details info



POST /api-server/user/login login

POST /api-server/user/register register

GET /api-server/user/unsubscribe Unsubscribe user



POST /api-server/user/update Update an existing user



Figure5: RESTful Controller APIs

I have created APIs for nearly all of the functional requirements - for UsersController we have:

- /login
Verifies the user credentials and returns the aforementioned JWT in the header of the response.
- /register
Creates a new user in the database provided all of the constraints are met. Otherwise, appropriate error messages are displayed.
- /update
Updated a user's own details, only working on one's own account as the user id is a primary key.
- /details
Returns an array with useful user details such as email, username, role, registered date and list of liked trails. It has limited response for other users.
- /unsubscribe
Deletes a logged in user from the database. Can only be performed through the command line or the SwaggerUI interface.

These API functions vary greatly in difficulty, introducing several checks for constraints, security aspects and database queries in the functions other than login and register. ([Appendix 3](#))

In TrailsController's case, there are functions for:

- /add
Adds a new trail in the database provided all of the mandatory fields were filled. It also automatically calls the police API and computes the number of crimes for the start of the trail.
- /delete
Delete one of your own trails.
- /find-all
Returns all of the available trails in the database. Google maps balances well displaying even immense numbers of trails.
- /like
Like an existing trail by increase their number of likes counter.
- /trail-details
Returns the details of a targeted trail.
- /update
Updates the details of one the trails.
- /user-trails
Returns the array of trails which are owned by a specific user

All of the developed API functions are ready for third party use, behaving as intended even when called upon through the command line. Naturally, an additional parameter is to be put when the Authorization Bearer is needed. ([Appendix 4](#))

I have taken deliberate care when writing the controllers' API functions, especially when it comes to the parameters that they require. That is why we are using Data Transfer Objects (DTOs) in most cases.

In the context of communication in web services, every call is an expensive operation. One way of reducing the number of calls is to use an object (the DTO) that aggregates the data that would have been transferred by several calls normally. It also ensures that minimum amount of data is sent, thus not risking sending unnecessary clutter that otherwise may potentially be critical data like the username of someone.

6.2.1. Authentication and user authorisation

Being an application which requires user login, their authentication is required. Therefore some API methods need to be available without login. In order to solve this problem I implemented Spring Security by overriding the http Cross-Origin Resource Sharing configuration in order to expose the header and be able to store and then later read the authentication bearer ([Appendix 5](#)).

If the login method was successful, the auth bearer is sent to the current user and the client application has to send it in every subsequent API request in order to validate its credentials.

The bearer in itself is a string of 256 random characters in which are hashed the username, role and user id of the user using the SignatureAlgorithm.HS512 and a secret key of my choosing. It is set to last five minutes for security reasons, so that the user does not lose the current session on browser refresh or mild loss of internet connection. ([Appendix 6](#))

6.2.2. Third Party Integration

Using the Unirest library we are able to query police data API in order to determine nearby crimes:

```
// Connects to an URL and returns the http response in JsonNode format
// Example URL:
// https://data.police.uk/api/crimes-at-location?date=2017-
02&lat=52.629729&lng=-1.131592
public static Integer composeTrailIncidents(GpsCoord currentPoint) {
    try {
        // ?date=2017-02&lat=52.629729&lng=-1.131592
        String policeAPI = "https://data.police.uk/api/crimes-at-location?date=" +
dateForPoliceAPI;
        String lat = "&lat=" + currentPoint.lat;
        String lng = "&lng=" + currentPoint.lng;
        policeAPI = policeAPI + lat + lng;
        // System.out.println("#### " + policeAPI);
        HttpResponse<JsonNode> jsonResponse;
        jsonResponse = Unirest.get(policeAPI).header("accept",
"application/json").asJson();
        // System.out.println("Response: " + jsonResponse.getBody().toString());

        return jsonResponse.getBody().getArray().length();
    } catch (UnirestException ex) {
        Logger.getLogger(TrailsController.class.getName()).log(Level.SEVERE, null,
ex);
        return 0;
    }
}
```

```
}
```

The police response contains a detailed report of each crime which passes through the backend and could be contained in the database, but for the purpose of this project I decided to only write down the number of crimes. In the future, this aspect of Trail Running can be further iterated upon.

6.2.3. Backend Testing & Project Documentation

Moving on to the testing aspect of the SpringBoot project, SwaggerUI proved to be an invaluable addition. The convenience of testing any feature of the RESTful API at a click's distance is something I came to appreciate greatly (Figure 6) . These manual tests represent the integration or acceptance testing which verifies end-to-end slices of the project. For instance, I would login, use the given authentication bearer to add a trail and then delete it, thus simulating a prospective series of actions of a regular user.

POST

/api-server/user/login login

Parameters

Cancel

Name	Description
request required (body)	request <div> <div>Example Value</div> <div>Model</div> </div> <pre>{ "password": "s1234567P", "username": "t.t@t.com" }</pre> <div>Cancel</div> <div>Parameter content type</div> <div>application/json</div>

Execute

Clear

Responses

Response content type

application/json; charset=UTF-8

Curl

```
curl -X POST "http://sa306.saturn.fastwebserver.de:9090/api-server/user/login" -H "accept: application/json; charset=UTF-8" -H "Content-Type: application/json" -d "{ \"password\": \"s1234567P\", \"username\": \"t.t@t.com\"}"
```

Request URL

```
http://sa306.saturn.fastwebserver.de:9090/api-server/user/login
```

Server response

Code	Details
200	<div>Response body</div> <pre>{ "responseCode": 0, "message": "OK login: Test" }</pre> <div>Download</div> <div>Response headers</div> <pre>authorization: Bearer eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJ0LnRAdC5jb28iLCJyb2xlijoivVNFUjIsInVzZXJJZCj6IjQjLCJleHAiOiE1NTg2OTk0ODd9.YIRYtuyiCWifUMSDGbC2oLRW9iPqLCSJvlb506H6tIb fomP360TRI0kEF6o uvimKeKqNVD4hsmU7BLItgbwDJw cache-control: no-cache, no-store, max-age=0, must-revalidate content-length: 45 content-type: application/json; charset=UTF-8 date: Mon, 29 Apr 2019 15:33:23 GMT expires: 0 pragma: no-cache x-content-type-options: nosniff x-frame-options: DENY x-xss-protection: 1; mode=block</pre>

Figure 6: Login through SwaggerUI example

There are also several Unit Tests made for the various controllers' functions, which are the backbone of the Agile Methodology. They are all being ran after a change in code, to ensure that previous code is still in working order.

```
@Test
public void testLikeTrail() {
    String existedLikes = "[]"; // myUser.get().getLikedTrails()
    JSONArray jsonLikes = new JsonParser().parse(existedLikes).getAsJsonArray();
    Long trailId = 1L;
    JsonElement trailIdElem = gson.fromJson(trailId.toString(),
    JsonElement.class);
    assertNotNull(trailIdElem);
}
```

And with that, the backend workings of Trail Running were up and running. The React web application followed the development cycle, which would make use of the various APIs created earlier.

6.3. Frontend Application

Task Id	Description	Priority
3.	Frontend	
3.1	Homepage	M
3.2	Login & Register functionality	M
3.3	Register Validation & GDPR	D
3.4	Google Maps Integration	M
3.5	Add Trail Manually	M
3.6	Record Trail Automatically	D
3.7	Snap-To-Road Feature	O
3.8	Basic Trail Statistics	M
3.9	Trail Slope & Elevation	D
3.10	Third Party service for Nearby Crimes	O
3.11	Liking Trails module	D
3.12	Delete & Update Trail	M
3.13	Offline Caching	O
3.14	User Details Page	D
3.15	View Own & Liked Trails	D
3.16	Display Corresponding Messages for User Actions	D

Sadly, several problems followed trail. My own inexperience in the matter was one, as I had the least amount of experience working in this area of the project. But by going in-depth into the research found in my literature survey on React, this issue was soon no longer. By making good use of a couple external libraries, Trail Running's full functionality was ensured:

- [Gmap](#): Google Maps JavaScript API is the most widely spread library for utilising embedded maps in client applications such as the web component of Trail Running.

- [HashRouter](#): Is a ReactJS component which enables page navigation. Its most common use is in routing single page applications between different screens.
- [React-bootstrap](#): Every frontend application needs components for building its interface. Bootstrap is one of the most well-known libraries being the standard de facto in responsive design. This will further be useful with respect to the client's mobile application.
- [PrimeReact](#): this is another open source component library used for a couple components which were not available in React-bootstrap, such as a gmap wrapper and dialog modules for notifications.
- [Axios](#): A well-established lightweight HTTP client for JavaScript. It was used to call the backend API methods via GET or POST methods.

6.3.1. Frontend Base Functions

The frontend module was developed using Microsoft Visual Code Studio and is organised in Pages, Components and common Utility files. (Figure 8)

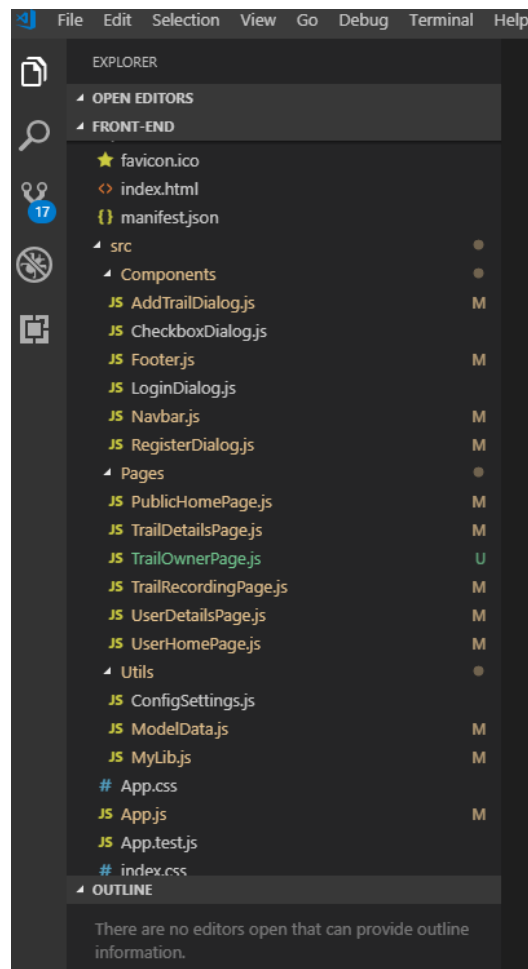
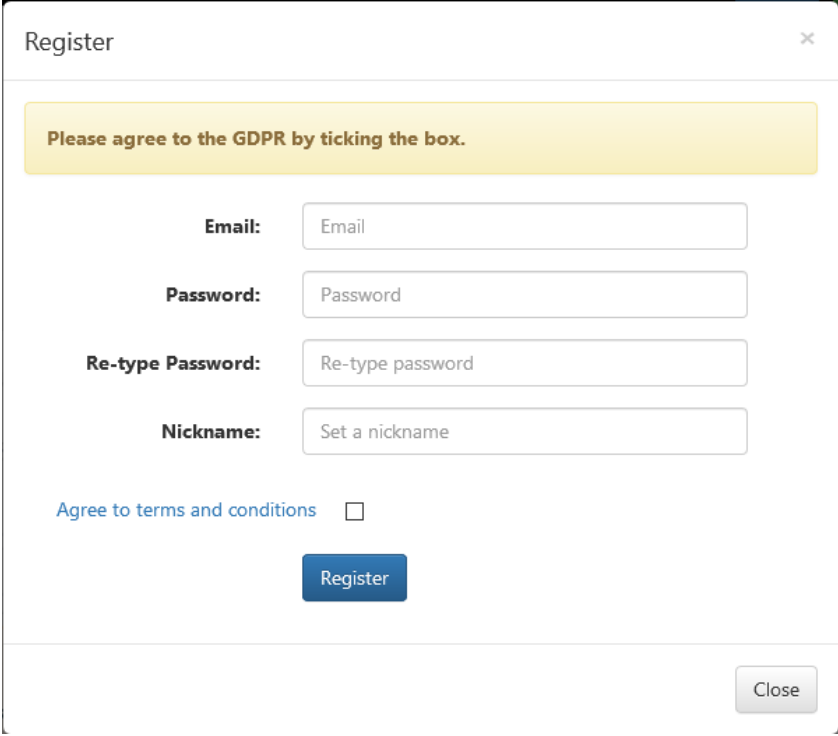


Figure 8: The Frontend Structure

The first interfaces built were Register and Login. They are singular instances of code each in separate files, which facilitates inserting them as components in the Navbar, which is present on every page of the website. This modular architecture ensures future proofing this part of Trail Running, as the common parts of the pages, like the Footer or the form for adding trails can be inserted into newly-created pages with minimal work. (Figure 9)



Register

Please agree to the GDPR by ticking the box.

Email:

Password:

Re-type Password:

Nickname:

Agree to terms and conditions ☐

Register

Close

Figure 9: Register Dialogue

Early design choices were also made when it comes to common page functionality, in particular the geolocation functions which are used in most parts of Trail Running. All of these reside in a custom library which can be imported in any file that would need it. The same practices were followed for configuration settings such as the server URL or the inactivity timer.

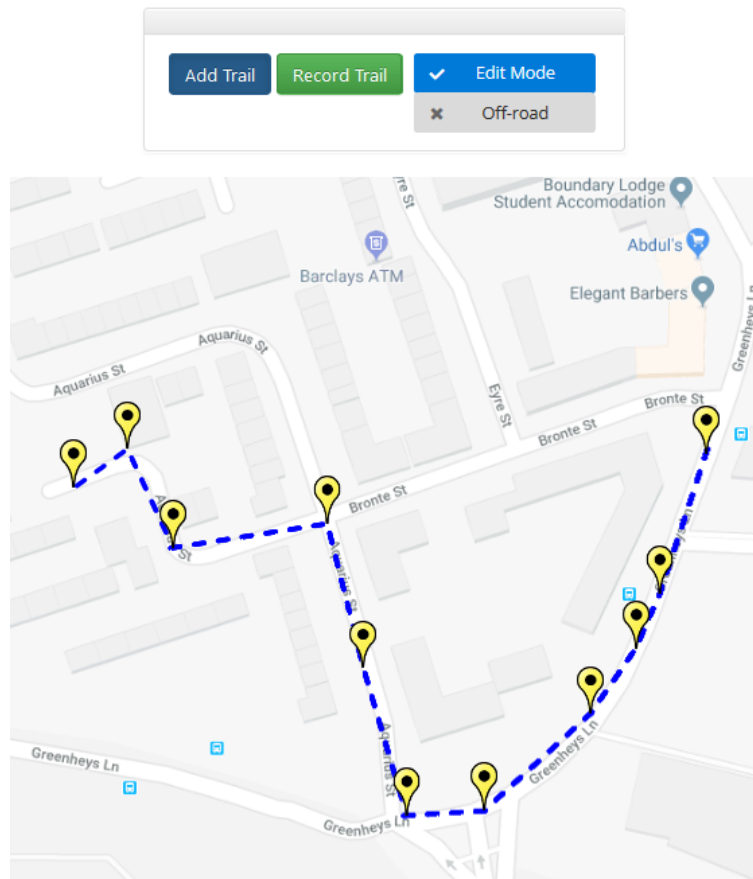
Trail Running's website also uses the browser's local storage to the fullest, thus recovering the active session and remembering the user login. This has proved exceptionally useful for the Trail Recording feature when the internet connection is not available. By storing the current location's coordinates, the program computes the trail in real time no matter what. The user then can upload their trail to the server when the internet connection has been restored.

Continuing development, the main features of the project soon followed:

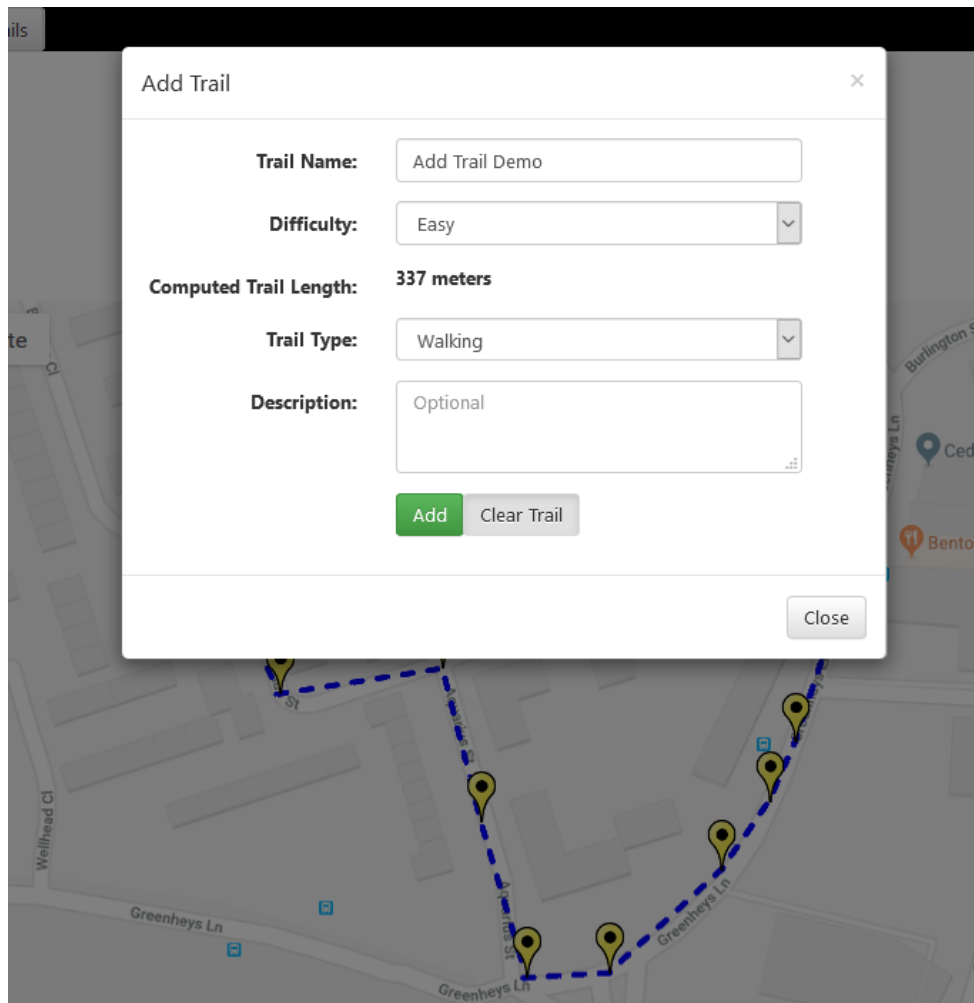
6.3.2. Adding Trails Manually

This is a useful feature to quickly load a new route in the server, by entering Edit Mode to add points by clicking on the map. Alternatively, this method of trail making can be used to plan routes onto which a user can run at a later time.

User Home Page



When the trail has the desired length, a user can choose to either add or discard it. One is prompted with customisable fields such as the next screen capture shows.



Of note is that the length of the trail is automatically computed using geolocation through a custom function.

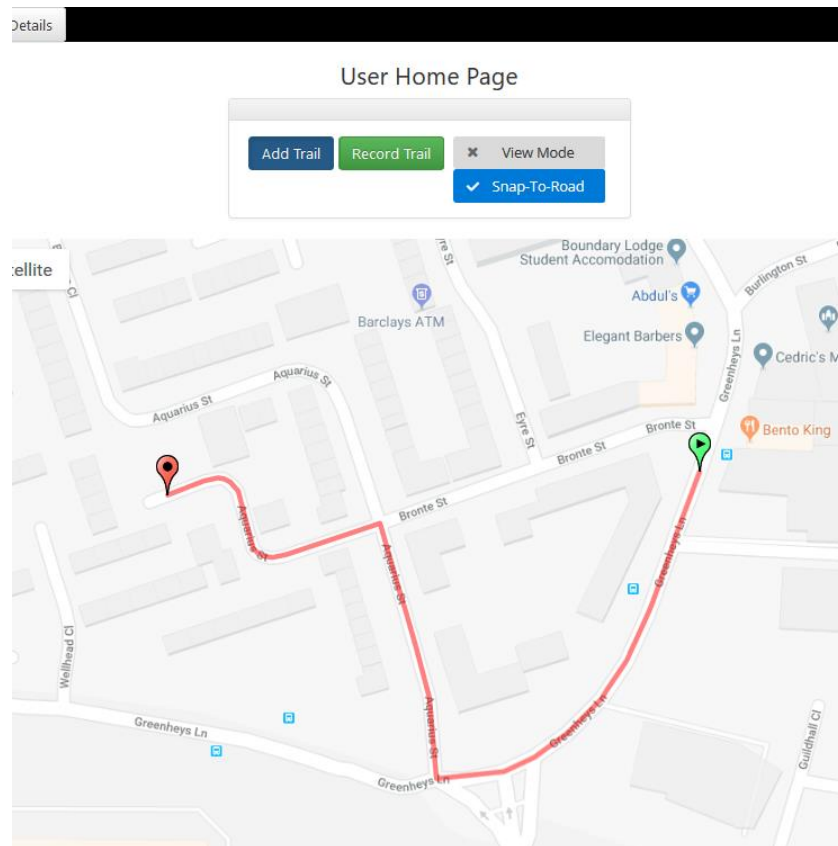
```
export function distanceBetweenCoordinates(lat1, lon1, lat2, lon2) {
  console.log("distanceBetweenCoordinates: " + lat1 + ", " + lon1 + " and " +
    lat2 + ", " + lon2);
  var earthRadius = 6371000;

  var dLat = degreesToRadians(lat2 - lat1);
  var dLon = degreesToRadians(lon2 - lon1);

  lat1 = degreesToRadians(lat1);
  lat2 = degreesToRadians(lat2);

  var a = Math.sin(dLat / 2) * Math.sin(dLat / 2) +
    Math.sin(dLon / 2) * Math.sin(dLon / 2) * Math.cos(lat1) * Math.cos(lat2);
  var c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));
  return earthRadius * c;
}
```

By choosing the snap-to-road toggleable, the user-made trail is sent to a Google Maps API which streamlines it and places it onto the nearby roads automatically:



This function which communicates with Google API for adjusting the trail onto nearby roads is:

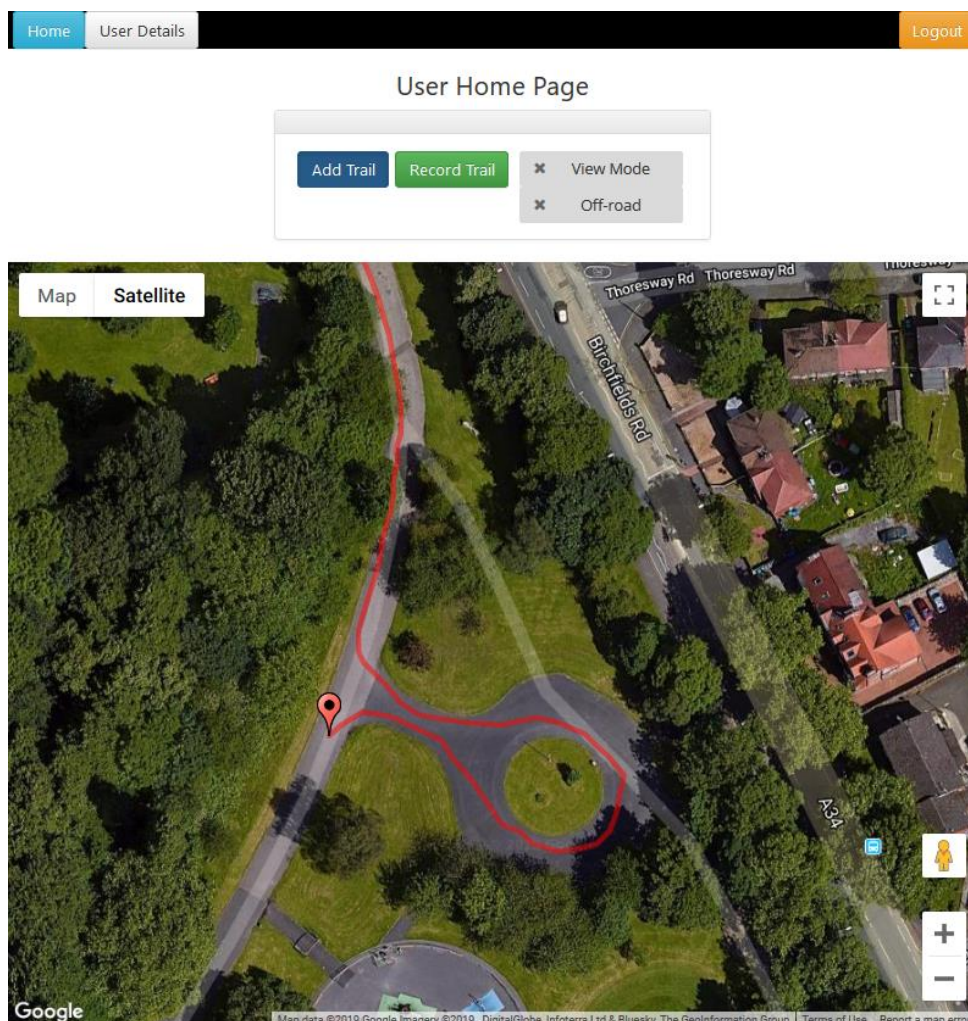
```
// Snap a user-created polyline to roads and draw the snapped path
//https://roads.googleapis.com/v1/snapToRoads?path=-35.27801,149.12958|-
35.28032,149.12907|-35.28099,149.12929|-35.28144,149.12984|-
35.28194,149.13003|-35.28282,149.12956|-35.28302,149.12881|-
35.28473,149.12836&interpolate=true&key=YOUR_API_KEY
export async function runSnapToRoad(path) {
  const google = window.google;
  let pathValues = "";
  for (let i = 0; i < path.length; i++) {
    pathValues += path[i].lat + "," + path[i].lng + "|";
  }
  pathValues = pathValues.substring(0, pathValues.length - 1);
  pathValues +=
"&interpolate=true&key=AIzaSyCdfU7kT27TmWiEAYDmallIcebbeBe5M_E";
  // console.log("New path: " + pathValues);
  await axios.get("https://roads.googleapis.com/v1/snapToRoads?path=" +
pathValues)
  .then(await function (response) {
    //console.log("New path " + JSON.stringify(response));
```

```

let point;
global.newTrailSnapToRoad = [];
for(let i = 0; i < response.data.snappedPoints.length; i++) {
    //console.log(JSON.stringify(response.data.snappedPoints[i]) + "/n");
    point = { lat: response.data.snappedPoints[i].location.latitude, lng:
response.data.snappedPoints[i].location.longitude };
    global.newTrailSnapToRoad.push(point);
}
//console.log("snapped: " + JSON.stringify(global.newTrailSnapToRoad));
})
.catch(function (error) {
    console.log(error);
});
}

```

Alternatively, one may choose to keep the original off-road functionality of the Add Trail feature, which allows the placement of the coordinates regardless of terrain and road infrastructure. A synergy with the chosen library for embedding Google Maps also arises, as the options for Satellite and Street View are available too.



The Street View feature is especially useful when paired with a mobile device, as it makes full use of its compass.



6.3.3. Automatic Trail Recording

This feature allows the user to start recording their trail with variable pace, all the while showing the distance of the current sprint.

Recording Page

Current lat:	<input type="text" value="53.4615352"/>
Current lng:	<input type="text" value="-2.2317147999999998"/>
Last sprint distance:	<input type="text" value="3.434295993917083"/>
Select pace:	<div><div>Walking</div><div>Off-road</div></div>
<div><div>Add Trail</div><div>Start Recording</div><div>Stop Recording</div></div>	



The pacing system had to be designed because the GPS in itself is fairly inaccurate, with deviations of up to four meters. Thus, even though the application is polling the current location of the user every five seconds, it does not take it into account as a new valid trail point if it does not pass a certain threshold. After extensive testing, I chose the following thresholds based on the type of trail:

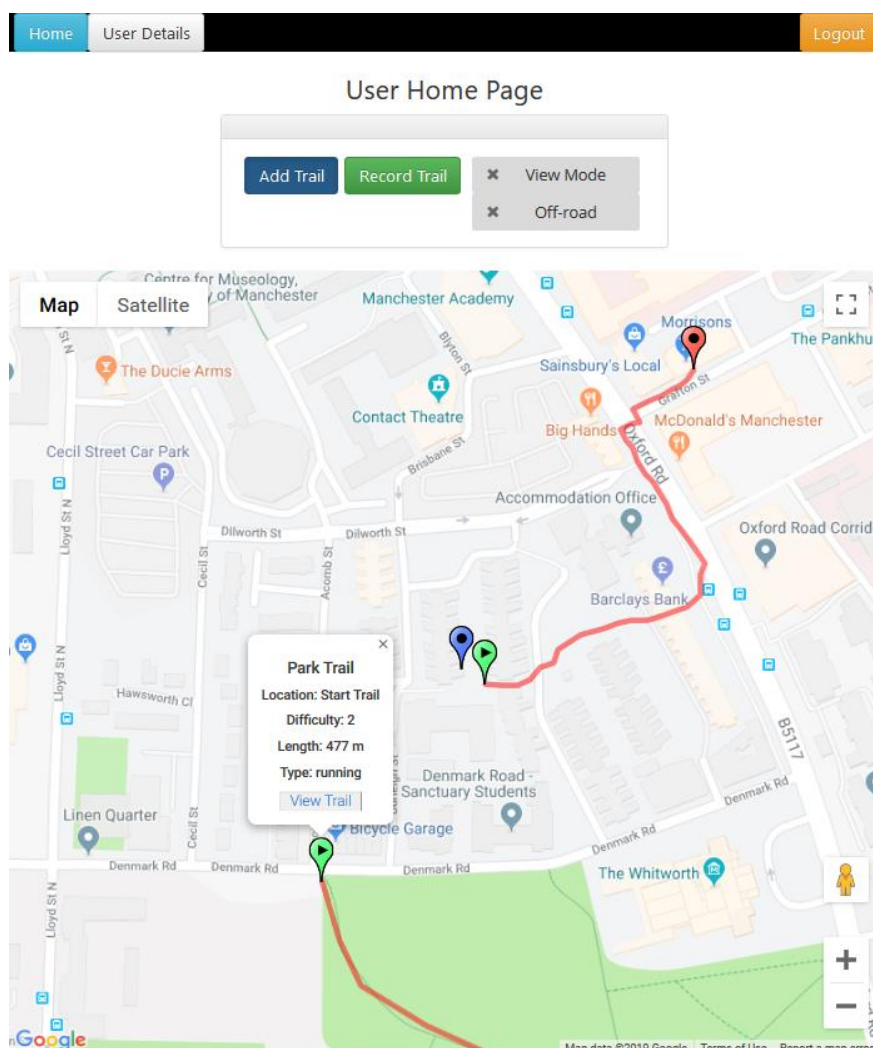
- Walking – 5 meters
- Running – 9 meters
- Cycling – 12 meters

For any added trail, we automatically store the elevation matrix for each point along the route, using Google Maps Elevation Service API. Using this extra data we calculate the gradient between the maximum and minimum elevation of the trail, as well as the trail's slope, displaying them in the trail details page.

6.3.4. Community-focused features

Once the main functionalities for adding trails were in place, it was time to focus on the community aspect of Trail Running.

The first step taken in this direction was the Homepage map, which was modified to show all of the existing trails, initially hardcoded on Manchester. Once a user logs in and allows access to their location, the User Page pans to be re-centred on the user's current location, thus showing the most relevant nearby routes.



Following that, the ability to select a trail was developed, as well as the specific page with its details where one can view all of its statistics and which provides a focused map with only the respective trail. In the case where one owns the respective trail, additional functionality is unlocked, such as updating its details and deleting it.

[Home](#) [User Details](#) [Logout](#)

Trail - Jogging in the Park Details

Trail Name:

Difficulty:

Computed Trail Length: **298 meters**

Trail Type:

Description:

Likes:

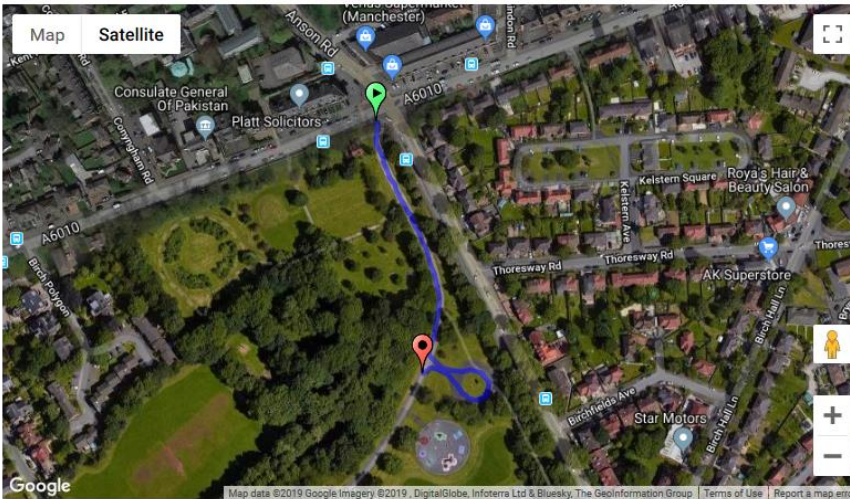
Creator: [View User](#)

Number of incidents: [View Details](#)

TrailSlope:

Maximum elevation: [Uphill](#)

[Like Trail](#) [Delete Trail](#) [Update](#)



Map data ©2019 Google Imagery ©2019 DigitalGlobe, Infoterra Ltd & Bluesky, The GeoInformation Group | Terms of Use | Report a map error

Trail Running Webapp by Stefan Pristoleanu

A liking system was also set in place, with the ability to view all of the liked trails in one's user details page. Moreover, one can update their own details in this page.

[Home](#) [User Details](#) [Logout](#)

User Test's Details Page

Username:

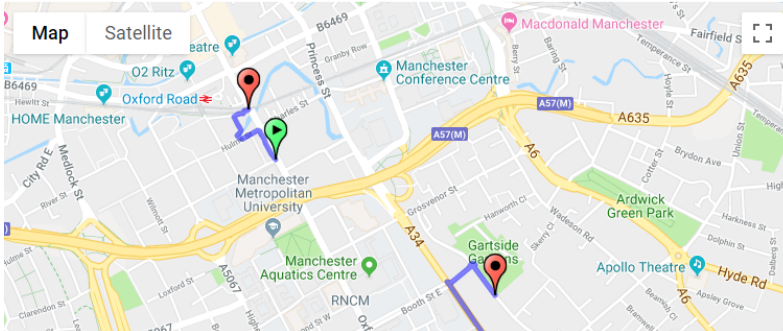
Nickname:

Registered On:

Total trail likes received:

Liked trails: [Road To Morrisons](#) | [Jogging in the Park](#) |

Test's own trails:



Surprisingly, a real obstacle came when communicating with Google Maps API. Because of our desire to have a mobile component as well, we could not rely on using localhost as a server anymore. A lengthy process was undertaken to put the database inside a docker container onto a server, with an instance of our server running permanently in the background thanks to 'mvn build'. The web and mobile applications would use this server's APIs in order to communicate and everything seemed to be in order - all of the Unit tests were passing and the SwaggerUI functionality checked out.

However, when connecting the React application to the new set of APIs, the Google Maps API would refuse to work. In truth, they only connect to HTTPS servers, with localhost being accepted as no harm can be done that way. We were faced with a new problem: either make our server secure or discard the plans of having a working mobile component.

Thankfully, I managed to find a way to make the server HTTPS, by creating and authenticating a self-issued certificate through the command line.

Finally, the SpringBoot backend was running on the server and the React frontend was successfully connected to it. Now that the end-to-end functionality was done, the Trail Running prototype was finished.

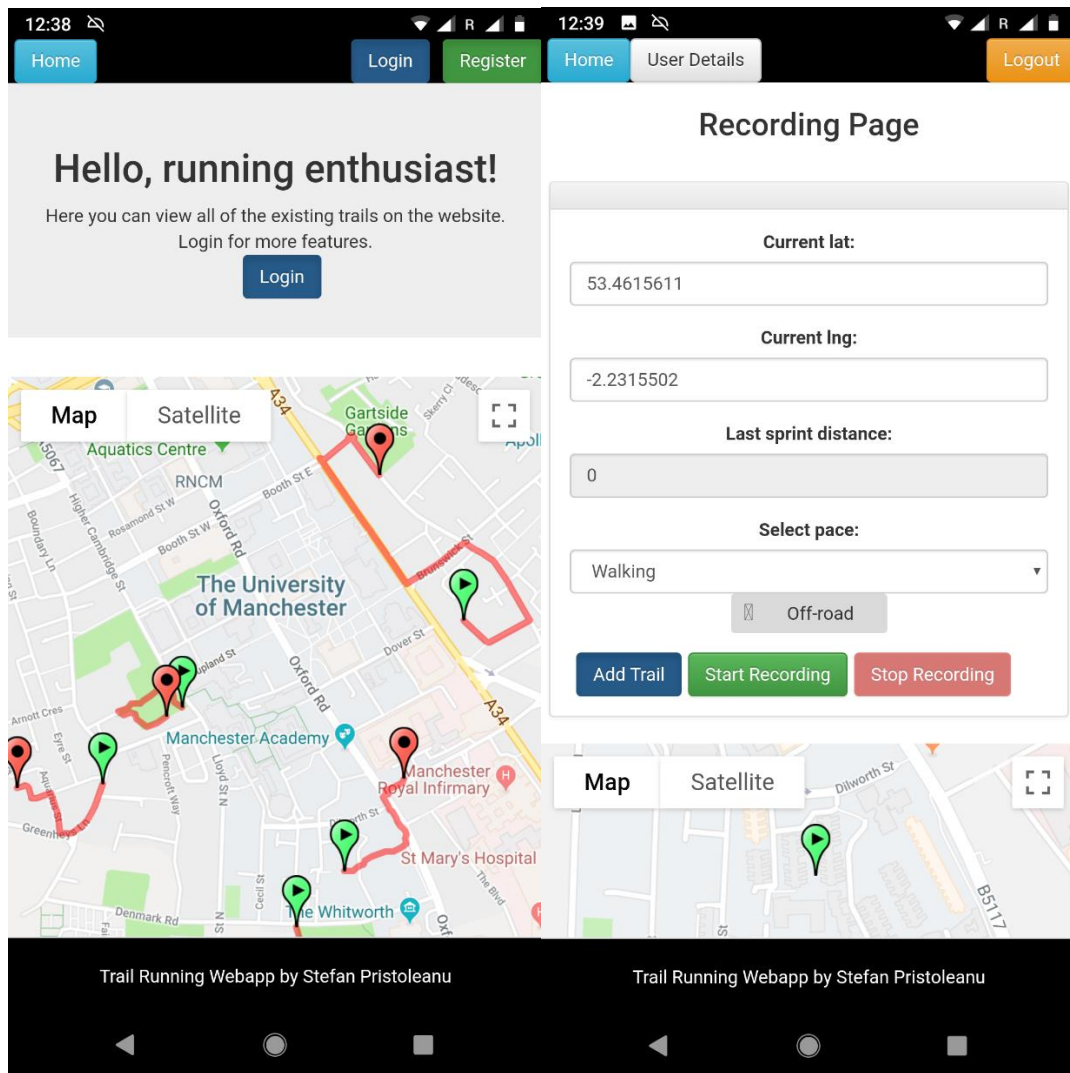
7. Mobile Application

Task Id	Description	Priority
4.	Mobile Application	
4.1	Porting the application with Cordova	D
4.2	Extensive testing	D

Although the frontend application has a responsive design, thus being fully functional on a mobile device's browser by going to the site URL: <https://sa306.saturn.fastwebserver.de:9093>, I considered it would be useful to create a dedicated smartphone application – for Android in particular.

In order to achieve this, we used Apache Cordova framework which wrapped the frontend production build into a mobile hybrid application, which is being ran by the WebKit engine of the device, which stands at the core of its browser.

The porting process required the installation of Android Studio, as well as the Apache Cordova framework with geolocation plug-ins. In the new Cordova project we dropped the final build from the React component, and edited the “index.html”. We added permissions towards third party services for google maps and the font library used. Then, we generated the Android apk, which I thoroughly tested on my mobile phone as seen below.



```
<meta http-equiv="Content-Security-Policy" content="default-src 'self'
'unsafe-inline'
    data: gap: https://ssl.gstatic.com
https://sa306.saturn.fastwebserver.de:9093 https://maps.google.com
https://maps.googleapis.com https://fonts.googleapis.com 'unsafe-eval'; style-
src 'self' 'unsafe-inline'; media-src ; img-src * 'self' data: content::">
    <meta name="format-detection" content="telephone=no">
    <meta name="msapplication-tap-highlight" content="no">
    <meta name="viewport" content="user-scalable=no, initial-scale=1,
maximum-scale=1, minimum-scale=1, width=device-width">
    <link type="text/css" href="css/2.a1660956.chunk.css"
rel="stylesheet">
    <link type="text/css" href="css/main.f6ca92c9.chunk.css"
rel="stylesheet">
```

8. Security

I have used quite a few practices meant to guarantee and maintain the security aspect of Trail Running.

To begin with, the data is being held onto a database and the critical data of users, namely their password is encrypted in the backend using the SHA-256 secure hashing algorithm. The one-way hashing function was designed by the National Security Agency (NSA) to be part of the Digital Signature Algorithm, and the database stores the result of this function, which cannot be traced back to the original password. ([Appendix 7](#))

Furthermore, the backend code was written with security in mind, making good use of private variables and protected methods.

The use of Json Web Token creates another layer of protection, preventing users to access methods which they are not allowed to, as presented earlier.

In addition, Trail Running uses Hypertext Transfer Protocol Secure, the extension over the usual HTTP for all communication purposes. This yields protection of the privacy and integrity of the exchanged data while in transit. It protects against man-in-the-middle attacks, which are some of the most common occurrences of stolen data in cybernetic attacks. HTTPS essentially nullifies eavesdropping and tampering of the communication, through the bidirectional encryption of the packages between a client and our server.

Moving more towards the frontend, it is always the user's choice to allow the sharing of their location, and the website functions on default settings if they choose to not do so. Speaking of the user, when one registers onto the Trail Running website, they are presented with number of security protocols. Starting with the EU General Data Protection Regulation (GDPR) checkbox, which has to be checked in order to continue.

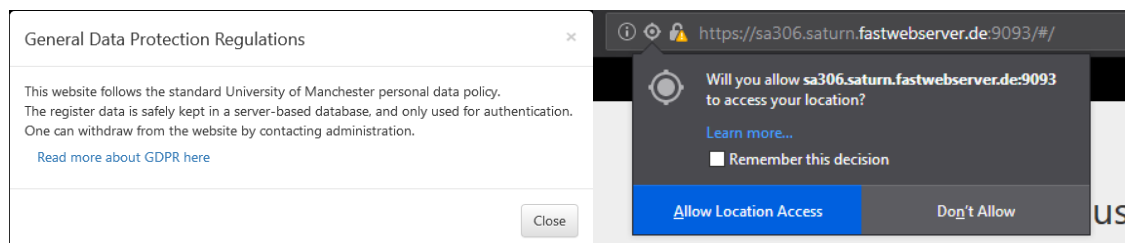


Figure: GDPR notice and Location Permission

Furthermore, I have made functions in order to ensure that the user's credentials are all in order: the email has an email format, and their password is between 7 to 15 characters which contain only characters, numeric digits, underscore and first character must be a letter.

```
export function validateEmail(email) {  
  var re =  
  /^(^<>()\[\]\.\.,;\s@"]+(\.^[^<>()\[\]\.\.,;\s@"]+)*|(".*"))@((\[[0-9]{1,3}\. [0-9]{1,3}\. [0-9]{1,3}\. [0-9]{1,3}\]|((a-zA-Z\-[0-9]+\.)+[a-zA-Z]{2,}))$)/;  
  return re.test(String(email).toLowerCase());  
}
```

```
export function validatePassword(inputPassword) {  
  var re = /^(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{6,}/;  
  return re.test(inputPassword);  
}
```


9. Ethics

Ethics are essentially moral principles that govern the behaviour of a group or individual. In the computer science domain, ethics are more often than not related to intellectual property rights, privacy concerns and how computers affect society.

Particularly in the fitness app area, there have been some several privacy infringements in the past. The most recent one would be Strava's heatmap, released in November 2017 and built from one billion activities in the past two years. It showed the paths its users would use as they run or cycle, and it accidentally showed the clear location of the largest US military facility in Afghanistan [<https://www.bbc.co.uk/news/technology-42853072>]. Ever since I heard about the news, I became aware of the various ethical aspects of my project.

I would like to think that Trail Running will have a positive impact upon society, as it promotes virtues of comradery through its ability to share trails to the world, or view and like other user's routes. Also, its unique feature of signalling the amount of crime in the trail's area is undeniably useful to individuals who exercise at night or in early mornings, when the aforementioned crimes usually take place.

Regarding privacy concerns, Trail Running only holds in its database data that the user sent of his own accord. The current location of a user could be problematic too, however it is registered on the user's device and only sent towards the server as a trail format when the user chooses to end the route.

Another interesting ethical aspect were trail images, which I was keen on implementing at the beginning of the project. Soon enough I came to realise that the my implementation would pose serious ethical questions, as there was no system of detecting what the actual photo contained. What if it contained the faces of people who did not give their permission, or maybe a completely different part of the town instead. Its proper implementation would require serious work in the image processing department, which would take up quite some time.

10. Future development

There are many ways in which Trail Running can be taken forward. It could certainly use more polish in the web design department, with the caveat of keeping the design responsive so that the mobile iteration would still be in working order.

I have dedicated a bit of time to thinking about one of my presentation's suggestions, that is to add the ability to port trails from other already established providers such as the fitness application Strava. After a bit of research, it seems like most providers use the Encoded Polyline Algorithm Format, an eleven step algorithm meant to compress a trail to a single string, encoded using signed values.[\[27\]](#) ([Appendix8](#))

The reasoning for using this technique is mostly security-related, as most applications hold one's current location, however this represents an easier way of storing coordinates. Extensive further work would have been required to implement this in the time left, as it would imply radical changes to the database structure which ripples into both the backend and the frontend of Trail Running.

Another future ambition of mine, which I mentioned in the lightning talk, was a chatting module between users accessing the same trail. I believe that this would strengthen the community aspect of Trail Running.

Having successfully created an easily expandable structure, more statistics such as calorie burns and pacing would benefit the application in future updates.

At the same time, on the security side, it may be worthwhile to implement a Refresh Token mechanism (from OAuth2), which would enable us to seamlessly create new a JWT when the previous access bearer had expired.

In addition, it would be nice to have an IOS equivalent mobile app, however this required a MacOS platform for development and an iPhone for testing.

11. Conclusion

To sum up, Trail Running is a full-stack application with a database, an application server and two frontend clients, one for web and the other for mobile devices. For its development a lot of work has gone into the research phase, not only during the literature survey, in which I gathered crucial information regarding previously unknown technologies, but also during the testing phase between said technologies. In my opinion the speed testing tool had a worthwhile addition to this project, and it can be used for similar projects in the future.

In retrospect, I believe that the chosen technologies have been appropriate for this complex project and they allow for easy insertions of additional features, which Trail Running stands living proof of.

12. References

1. Kofler, K., 2005, The Definitive Guide to MySQL, Second Edition – accessed online from www.books.google.co.uk
2. PostgreSQL vs MySQL - <https://www.2ndquadrant.com/en/postgresql/postgresql-vs-mysql/>
3. Moniruzzaman, A.B.M. and Hossain, S.A., 2013, NoSQL Database: New Era of Databases for Big Data Analytics – Classification, Characteristics and Comparison, Daffodil International University, Vol 6, No 4 – available online at: <http://arxiv.org/ftp/arxiv/papers/1307/1307.0191.pdf>
4. Khan, M. A., 2012, SQL vs. NoSQL, available online at: <http://www.cse.yorku.ca/~jarek/courses/6421/F12/presentations/NoSQLDatabases.pdf>
5. MongoDB – available online at: <https://www.mongodb.com/what-is-mongodb>
6. Gajani, A., The key differences between MySQL and NoSQL DBs - available online at: <http://blog.monitor.us/2013/05/cc-in-review-the-key-differences-between-mysql-and-nosql-dbs/> [accessed on 02nd April 2019]
7. Gutierrez, F., 2018, Pro Spring Boot 2: An Authoritative Guide to Building Microservices, Web and Enterprise Applications, and Best Practices – available online at: <https://link.springer.com/content/pdf/10.1007/978-1-4842-3676-5.pdf>
8. Apache Tomcat – available online at: <https://tomcat.apache.org/>
9. Brittain, J., Darwin, I. F., 2007, Tomcat: The Definitive Guide, O'Reilly Media, Inc.
10. Node.js – available online at: <https://nodejs.org/>
11. Cantelon, M., Holowaychuk, T. J., Harter, M., Rajlich, N., 2013, Node.js in Action,

12. Tilkov, S., Vinoski, S., 2010, Node.js: Using JavaScript to Build High-Performance Network Programs, IEEE Internet Computing
13. A. Leff, Rayfield, J., 2001, Web-application development using the Model/View/Controller design pattern – available online at: <https://dl.acm.org/citation.cfm?id=650161>
14. Node-Postgres – available online at: <https://node-postgres.com/>
15. Alonso, G., Casati, F., Kuno, H., Machiraju, V., 2004, Web Services: Concepts, Architectures and Applications – available online at: <https://www.springer.com/gb/book/9783540440086>
16. World Wide Web Consortium – available online at: <https://www.w3.org/Consortium/>
17. Tyagi, S., 2006, RESTful Web Services – available online at: <http://www.oracle.com/technetwork/articles/javase/index-137171.html>
18. Gackenheim, C., 2015, Introduction to React – available online at: <https://www.apress.com/gb/book/9781484212462>
19. Bhansali, A., 2015, AngularJS: A Modern MVC Framework in JavaScript – available online at: www.jgrcs.info/index.php/jgrcs/article/download/952/610
20. Fedosejev, A., 2015, React.js Essentials – available online at: <https://books.google.co.uk/books?hl=en&lr=&id=Rhl1CgAAQBAJ&oi=fnd&pg=PP1&dq=react+js+essentials&ots=JjwvtyAXOJ&sig=KHy94K9erY8UxNTRvSW3X67BzWE#v=onepage&q=react%20js%20essentials&f=false>
21. Apache Cordova – available online at: <https://cordova.apache.org/>
22. Spring Security – available online at: <https://spring.io/projects/spring-security>
23. Spring JWT – available online at: https://www.tutorialspoint.com/spring_boot/spring_boot_oauth2_with_jwt.htm
24. Spring Data JPA – available online at: <https://spring.io/projects/spring-data-jpa>

25. Spring Data CRUD – available online at: <https://docs.spring.io/spring-data/data-commons/docs/1.6.1.RELEASE/reference/html/repositories.html>
26. SpringBoot Request Mapping – available online at: <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org.springframework.web.bind.annotation.RequestMapping.html>
27. Encoded Polyline Algorithm Format – available online at: <https://developers.google.com/maps/documentation/utilities/polylinealgorithm>

13. Appendix

13.1. Appendix: User Entity variable link example

```
@Entity
@Table(name = "USERS")
@XmlRootElement
public class UserEntity implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(columnDefinition = "serial")
    @Basic(optional = false)
    private Long userId;

    @Column(name = "USERNAME")
    private String username;

    @Column(name = "PASSWORD")
    private String password;

    @Column(name = "NICKNAME")
    private String nickname;

    @Column(name = "USER_ROLE")
    private String userRole;

    @Column(name = "REGISTRATED_AT")
    @Temporal(TemporalType.TIMESTAMP)
    private Date registratedAt;

    @Column(name = "LAST_LOGIN_AT")
    @Temporal(TemporalType.TIMESTAMP)
    private Date lastLoginAt;

    @Column(name = "LIKED_TRAILS")
    private String likedTrails;
```

13.2. Appendix: Constructor with DTO example

```
public TrailEntity(TrailDTO trailDTO, Long userId) {

    this.trailName = trailDTO.getTrailName();
    this.trailCreatedBy = userId;
```

```

this.trailCreatedAt = new java.sql.Timestamp(System.currentTimeMillis());
this.trailType = trailDTO.getTrailType();
if (trailDTO.getTrailDifficulty() != null) {
    this.trailDifficulty = trailDTO.getTrailDifficulty();
} else {
    this.trailDifficulty = 1;
}
this.trailLength = trailDTO.getTrailLength();
this.trailDescription = trailDTO.getTrailDescription();
this.trailNoOfLikes = 0;
this.deltaElevation = trailDTO.getDeltaElevation();
this.trailSlope = trailDTO.getTrailSlope();
this.dataJSON =
TrailsController.gson.toJsonTree(trailDTO.getCoordinates()).toString();
System.out.println("get coord: " + trailDTO.getCoordinates());
}

```

13.3. Appendix: API function complexity example

API function for register :

```

// alias curlj='curl --header "Content-Type: application/json" -w "\n" -v'
// curlj http://localhost:9090/api-server/user/register -d '{"username":"s1",
"password":"123",
// "nickname":"test"}'
@PostMapping(value = "/register", produces =
MediaType.APPLICATION_JSON_UTF8_VALUE)
public ResponseEntity<?> register(@RequestBody RegisterDTO request) {
    try {
        UserEntity newRec = new UserEntity(request);
        usersRep.save(newRec);
        JsonObject jsonResponse = new JsonObject();
        jsonResponse.addProperty("responseCode", 0);
        jsonResponse.addProperty("message", "OK register: " +
request.getUsername());
        return new ResponseEntity(jsonResponse.toString(), HttpStatus.OK);
    } catch (JsonSyntaxException ex) {
        LOG.severe(ex.toString());
        return new ResponseEntity(ex.toString(), HttpStatus.BAD_REQUEST);
    }
} // end register

```

API function for update:

```

@ApiOperation(value = "Update an existing user",
authorizations = {@Authorization(value = "Bearer")})

```



```

@PostMapping(value = "/update", produces =
MediaType.APPLICATION_JSON_UTF8_VALUE)
public ResponseEntity<?> updateUser(@RequestBody UpdateUserDTO userDTO) {
    try {
        Long userId = 0L;
        try {
            Authentication auth =
                SecurityContextHolder.getContext().getAuthentication();
            String username = auth.getPrincipal().toString();
            String userRole = auth.getAuthorities().toArray()[0].toString();
            userId = Long.parseLong(auth.getAuthorities().toArray()[1].toString());
        } catch (Exception e) {
            LOG.severe(e.toString());
        }
        Optional<UserEntity> myUser = usersRep.findById(userDTO.getUserId()); //
        findById(userId);
        if (!myUser.isPresent()) {
            return new ResponseEntity("User not found", HttpStatus.NOT_FOUND);
        }
        myUser.get().setUsername(userDTO.getUsername());
        myUser.get().setNickname(userDTO.getNickname());
        usersRep.save(myUser.get());
        JsonObject jsonResponse = new JsonObject();
        jsonResponse.addProperty("responseCode", 0);
        jsonResponse.addProperty("message", "OK updated user: " +
myUser.get().getUsername());
        return new ResponseEntity(jsonResponse.toString(), HttpStatus.OK);
    } catch (JsonSyntaxException ex) {
        LOG.severe(ex.toString());
        return new ResponseEntity(ex.toString(), HttpStatus.BAD_REQUEST);
    }
} // end updateUser

```

13.4. Appendix: Command line API call examples

```

curl -X GET "http://sa306.saturn.fastwebserver.de:9090/api-server/trail/find-
all" -H "accept: application/json;charset=UTF-8"

```

```

curl -X GET "http://sa306.saturn.fastwebserver.de:9090/api-
server/user/details" -H "accept: application/json;charset=UTF-8" -H
"Authorization: Bearer
eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJ0LnRAdC5jb20iLCJyb2x1Ijo1VWVFNUIsInVzZXJJZCI6I
jQiLCJleHAiOiE1NTg3MjUzMjU1LjUwS9hNVnHne-
IqkiebH1U1FnFnH330XXi0e0072w7B8d2StFTDoy0WMiwQtXRy00YQ0t1ksZ_GyTnhRCU9h_Q"

```

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    // private final BCryptPasswordEncoder passwordEncoder = new
    BCryptPasswordEncoder();
    // @Autowired private UserDetailsServiceImpl userDetailsService;
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        // Override the http Cross-Origin Resource Sharing configuration in order
        // to expose the header
        // with the authentication bearer
        http.cors()
            .configurationSource(
                (HttpServletRequest request) -> {
                    CorsConfiguration config = new CorsConfiguration();
                    config.setAllowedHeaders(Collections.singletonList("*"));
                    config.setAllowedMethods(Collections.singletonList("*"));
                    config.addAllowedOrigin("*");
                    config.setAllowCredentials(true);
                    config.addExposedHeader("authorization");
                    return config;
                });
        http.csrf()
            .disable()
            // make sure we use stateless session; session won't be used to store
            // user's state:
            .sessionManagement()
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
            .and()
            .authorizeRequests()
            // permit all access to /rest-jpa/ for speed test and also for
            // register and login api:
            .antMatchers(
                "/api-server/user/register",
                "/api-server/user/login",
                "/api-server/trail/find-all",
                "/v2/api-docs/**",
                "/webjars/**",
                "/swagger-resources/**",
                "/swagger-ui.html**",
                "/swagger-ui/**")
            .permitAll()
            // all other api methods must be authenticated:
            .anyRequest()
            .authenticated()
            .and()
    }
}
```

```

        // add filter for all other requests to check the presence of JWT in
header:
        .addFilterBefore(new AuthenticationFilter(),
UsernamePasswordAuthenticationFilter.class);
    }

```

13.6. Appendix: JWT Authentication

```

static final long EXPIRATIONTIME = 300_000; // 5 minutes = 300_000

static final String SECRET_KEY = "my_secret_key_for_JWT";

static final String TOKEN_PREFIX = "Bearer";
static final String HEADER_STRING = "Authorization";

public static void addAuthentication(
    HttpServletResponse res, String username, String role, Long userId) {
    res.addHeader(
        HEADER_STRING, TOKEN_PREFIX + " " + generateAuthentication(username,
role, userId));
}

public static String generateAuthentication(String username, String role,
Long userId) {

    Claims claims = Jwts.claims().setSubject(username);
    claims.put("role", role);
    claims.put("userId", userId + "");
    String jwt =
        Jwts.builder()
            .setClaims(claims)
            .setExpiration(new Date(System.currentTimeMillis() +
EXPIRATIONTIME))
            .signWith(SignatureAlgorithm.HS512, SECRET_KEY)
            .compact();
    return TOKEN_PREFIX + " " + jwt;
}

```

13.7. Appendix: Password Hashing

```

/** @author stefan */
public class UtilForUser {

    public static String hashEncryption(String mytext) {
        String cryptText;
        try {

```

```

        java.security.MessageDigest encryptionAlgo =
            java.security.MessageDigest.getInstance("SHA-256");
        encryptionAlgo.reset();
        encryptionAlgo.update(mytext.getBytes("UTF-8"));
        byte[] sha_digest = encryptionAlgo.digest();
        StringBuilder hexString = new StringBuilder();
        for (int i = 0; i < sha_digest.length; i++) {
            if ((0xFF & sha_digest[i]) < 16) {
                hexString.append("0");
            }
            hexString.append(Integer.toHexString(0xFF & sha_digest[i]));
        }
        cryptText = hexString.toString().toUpperCase();
    } catch (java.security.NoSuchAlgorithmException |
UnsupportedEncodingException ex) {
        System.out.println("SHA algorithm was not found ! Error: " +
ex.toString());
        return mytext;
    }
    return cryptText;
} // end encryptPassword(String mytext)
}

```

13.8. Appendix: Encoded Polyline Algorithm Format

1. Take the initial signed value:
-179.9832104
2. Take the decimal value and multiply it by 1e5, rounding the result:
-17998321
3. Convert the decimal value to binary. Note that a negative value must be calculated using its [two's complement](#) by inverting the binary value and adding one to the result:
00000001 00010010 10100001 11110001
11111110 11101101 01011110 00001110
11111110 11101101 01011110 00001111
4. Left-shift the binary value one bit:
11111101 11011010 10111100 00011110
5. If the original decimal value is negative, invert this encoding:
00000010 00100101 01000011 11100001
6. Break the binary value out into 5-bit chunks (starting from the right hand side):
00001 00010 01010 10000 11111 00001
7. Place the 5-bit chunks into reverse order:
00001 11111 10000 01010 00010 00001
8. OR each value with 0x20 if another bit chunk follows:
100001 111111 110000 101010 100010 000001
9. Convert each value to decimal:
33 63 48 42 34 1
10. Add 63 to each value:
96 126 111 105 97 64
11. Convert each value to its ASCII equivalent:
`~oia@