

BABEŞ–BOLYAI TUDOMÁNYEGYETEM KOLOZSVÁR
MATEMATIKA ÉS INFORMATIKA KAR
INFORMATIKA SZAK

Szakdolgozat

**Kiránduló Web Applikáció, valós idejű
követéssel**



TÉMAVEZETŐ:

**DR. ING. VARGA LEVENTE,
EGYETEMI ADJUNKTUS**

SZERZŐ:

BARABÁS BALÁZS

2024

BABEȘ–BOLYAI UNIVERSITY OF CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND INFORMATICS
SPECIALIZATION: COMPUTER SCIENCE

Diploma Thesis

Hiking Web Application with Real Time tracking capabilities



ADVISOR:
LECTURER DR. ING. LEVENTE VARGA

AUTHOR:
BALÁZS BARABÁS

2024

UNIVERSITATEA BABEȘ-BOLYAI, CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ

Lucrare de licență

**Aplicație Web pentru drumeții, cu
funcționalitate de urmărire în timp real**



CONDUCĂTOR ȘTIINȚIFIC:
LECT. UNIV. DR. ING. LEVENTE VARGA

ABSOLVENT:
BALÁZS BARABÁS

2024

BABEȘ–BOLYAI UNIVERSITY OF CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND INFORMATICS
SPECIALIZATION: COMPUTER SCIENCE

Diploma Thesis

Hiking Web Application with Real Time tracking capabilities

Abstract

The lack of digital infrastructure in our countries Tourism Industry means that while foreign and domestic tourists can find a variety of information online, personalized offers are hard to come by. On the other side of the business spectrum, local guides are also met with difficulties when trying to engage these potential customers.

To better bridge the gap between tourists, tour guides, and companies in the tourism industry, we decided to develop a hiking app that addresses the needs of all parties involved.

The resulting project will be the subject of the following paper.

This work is the result of my own activity. I have neither given nor received unauthorized assistance on this work.

2024

BALÁZS BARABÁS

ADVISOR:
LECTURER DR. ING. LEVENTE VARGA

Contents

- 1 Introduction** **1**
- 2 Technologies** **2**
 - 2.1 TypeScript 3
 - 2.2 Frontend 3
 - 2.2.1 Vue 3
 - 2.2.2 Tailwind CSS 4
 - 2.2.3 Leaflet 6
 - 2.2.4 SocketIO 6
 - 2.3 Backend 7
 - 2.3.1 Node.js 8
 - 2.3.2 Fastify 8
 - 2.3.3 MongoDB 9
 - 2.3.4 JWT 9
- 3 Infrastructure** **11**
 - 3.1 Microsoft Azure 11
 - 3.1.1 Static Web Apps 11
 - 3.1.2 App Service 12
 - 3.2 GitHub 12
 - 3.2.1 GitHub Actions 12
 - 3.3 MongoDB 14
 - 3.3.1 Clusters 14
 - 3.4 Testing 14
 - 3.4.1 Vitest 14
- 4 Presenting the Application** **16**
 - 4.1 Roles 16
 - 4.2 Registration 16
 - 4.3 Login 17
 - 4.4 Account settings 18
 - 4.5 Interacting with trails 19
 - 4.6 Real-time tracking functionalities 20
- 5 Implementation** **22**

CONTENTS

- 5.1 Database Overview 22
- 5.2 Client Side App 22
 - 5.2.1 Components 24
 - 5.2.2 Pages and routes 25
 - 5.2.3 Stores and Services 26
 - 5.2.4 Composables 28
 - 5.2.5 Leaflet 29
- 5.3 The API 30
 - 5.3.1 The server instance 30
 - 5.3.2 Routes 31
 - 5.3.3 Controllers 32
 - 5.3.4 Services 32
- 5.4 The WebSocket Server 32
- 6 Potential Improvements 33**
- 7 Conclusions 34**

1. Chapter

Introduction

The past couple of years has shown a rapid increase of foreign nationals living in Romania. From students on University campuses, to guest workers or tourists, they can be found all across the country. Targeting this demographic could be the source of potentially huge revenue streams for our countries growing Tourism Industry, but there are a couple of challenges, the biggest one being: the lack of both physical and digital infrastructure. This lack of digital infrastructure means that while foreign and domestic tourists can find a variety of information online, personalized offers are hard to come by. On the other side of the business spectrum, local guides are also met with difficulties when trying to engage these potential customers.

To better bridge the gap between tourists, tour guides, and companies in the tourism industry, we decided to develop a hiking app that addresses the needs of all parties involved, and this project will be the subject of the following document.

At first we will take an in depth look at all the technologies used throughout development. This section of the paper includes an overview of both front and backend side of our app, and the tools that have been used to implement these. After that comes a presentation about the infrastructure, meaning where and how we host the project, what the testing flows are and how deployment happens. We will also take a detailed look at all the functionalities present in the project, and examine how they were implemented step by step from the database level all the way to the User Interface. At the end we will discuss some potential improvements that can be added in the future and summarize some conclusions about the whole project.

2. Chapter

Technologies

The technologies used to create the Application have been chosen carefully based on the intended goals of this project. In order to have a faster development cycle on both Client and Server side, the code has been written in TypeScript, a modern and safer superset of the JavaScript programming language. Using the same language allows developers to write more compatible code and saves effort and time, that comes from learning and adapting to different technologies.

The frontend has been created with the goal of having a clean, simple-to-use UI, that is stable and works as intended. The choice of using Vue was made based on these criteria, since it's a mature framework, with excellent documentation and a thriving ecosystem. Vue also integrates easily with other tools used in this project, such as: Tailwind CSS (CSS framework for easier styling) and TypeScript.

The rendering of the maps and trails has been done using LeafletJs, an open source JavaScript library. We chose to use this library, since it offers a high degree of customization with a well designed and easy to use API.

Real time tracking functionalities have been implemented with SocketIO, also a JavaScript based wrapper library for managing publish/subscribe events using the WebSocket communication protocol.

Data is stored in a MongoDB database. It currently is one of the most popular document based, NoSQL database system. The choice to use it for this project was made because it is easy-to-setup, and integrates well with JavaScript based systems that communicate through JSON formatted data.

The Backend code itself was also written in TypeScript, using Fastify a low overhead and stable Node.js web framework. It uses Mongoose ORM in order to handle data manipulation and JWT tokens for authentication.

2.1 TypeScript

JavaScript has transformed in the last 20 years from a relatively simple scripting language, to being the go-to solution for Web based systems, not just on the frontend but also on the Server side thanks to Node.js. While the language matured a lot, the humble origins of it can still be observed in the fact that it lacks a coherent type system to this day, which is the source of many bugs.

TypeScript's main goal is to remedy this exact problem by introducing a Static Type checking system. Originally conceived by engineers at Microsoft [1], TypeScript has become the de facto standard for modern web development.

TypeScript is a typed superset of JavaScript, meaning that it adds rules and requires the developer to specify how different kinds of values can be used [2]. Type checking happens statically, meaning that the detection of errors happens without running the code. Errors are instead identified by comparing the code against previously defined schemas on how data should look and behave. This kind of behaviour allows code written in TypeScript to retain JavaScript's runtime behaviour while also avoiding potential bugs stemming from a weak type system [2].

In order to best benefit from the properties mentioned above, both the frontend and backend of the project was written using TypeScript.

2.2 Frontend

The client side implementation of our project was done in Vue. It provided all the core functionalities needed here: UI reactivity, a routing implementation for multiple pages and a way to programmatically render HTML. The capabilities of this framework were augmented by using a wide range of external libraries and frameworks, the most important among them being: TypeScript, Tailwind CSS, LeafletJS and SocketIO.

2.2.1 Vue

Vue is an open-source JavaScript framework and ecosystem used for creating user interfaces and single-page applications [3]. It was created by Evan You, and since its original release in 2014, it has become one of the most popular frontend frameworks alongside Facebook's React and Google's Angular.

2. CHAPTER: TECHNOLOGIES

The framework builds on top of the basic web technologies: HTML, CSS and JavaScript and organises them into components, this way allowing the developer to write declarative code and to easily scale the scope of his project.

Vue organizes it's source code into Single File Components (SFC) that provide a structured way of declaring structure(HTML), functionality(Javascript) and styling(CSS) of a component in one file as seen in Source Code 2.1.

```
<script setup>
import { ref } from 'vue'
const count = ref(0)
</script>

<template>
  <button @click="count++">Count is: {{ count }}</button>
</template>

<style scoped>
button {
  font-weight: bold;
}
</style>
```

Source Code 2.1: Example of Single File Component (SFC) in in Vue.

Just like in React, Vue uses a Virtual DOM (Document Object Model) to create a reactive system. The concept behind Virtual DOMs is that the framework stores a virtual copy of the DOM in it's memory. Each DOM element is a self contained component that has it's one state.

These elements can also be viewed as nodes of a tree structure. Whenever a change is detected, the Virtual DOM tree is traversed again and if a change has happened in the state of that node, it will be updated. This allows the system to only update specific components, whenever changes occur, instead of re-rendering the whole page [4]. Only updating specific nodes leads to smaller memory usage and faster reactivity.

2.2.2 Tailwind CSS

Web based User interfaces usually use CSS (Cascading Style Sheet), a programming language to customize the appearance of it's HTML elements. From positioning to text colors, all of these attributes are specified by using this language.

2. CHAPTER: TECHNOLOGIES

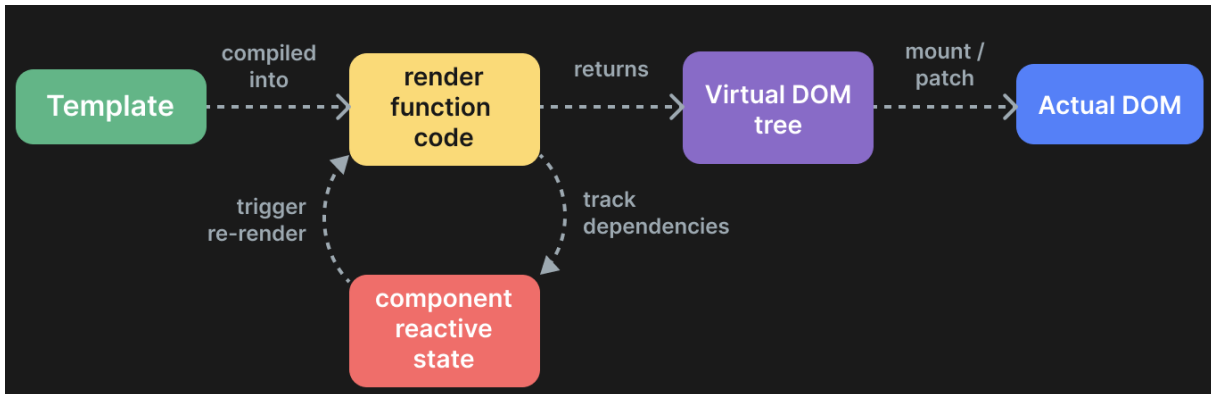


Figure 2.1: Visualization of the Virtual DOM to real DOM transformation in Vue. <https://vuejs.org/guide/extras/rendering-mechanism>.

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      .custom-class {
        padding: 4px;
        margin: 4px;
        background-color: rgb(255 255 255);
      }
    </style>
  </head>
  <body>
    <p class="custom-class">This paragraph uses my own custom class</p>
    <p class="p-4 m-4 bg-white">This paragraph uses tailwind classes</p>
  </body>
</html>
```

Source Code 2.2: Example of a custom CSS class can be seen on the first p tag. The same functionality can be achieved with Tailwind CSS utility classes as seen in the second tag.

Since a lot of CSS attributes are reused across multiple elements, these can be grouped into so called *CSS classes*. These allow us to create uniform UI elements, that all share the same kind of design. While CSS provides a lot of useful utility, managing and grouping classes can become rather overwhelming and cumbersome, especially on larger projects, and this is where Tailwind CSS comes into play. Tailwind CSS [5] provides a large number of predefined utility classes that can be tied directly to HTML elements in the template.

The framework works by scanning all of your HTML files or in this case Vue files for class

2. CHAPTER: TECHNOLOGIES

names, generating the corresponding styles and then writing them to a static CSS file. This way there's no overhead for the application and makes styling the app much easier.

Source Code 2.2 illustrates how using utility classes makes it easier for the developer to style HTML elements.

2.2.3 Leaflet

Since the project aims to create a platform where custom trails can be viewed, a tool is needed that can help illustrate these as interactive map components. This is where Leaflet comes into play. Leaflet [6] is a fully open-source JavaScript library, created for mobile-friendly interactive maps. It combines both styling and functionality, allowing developers to customize both look and behaviour of their maps.

Leaflet uses so called tile layers to render maps onto web interfaces. These layers were originally invented by Google for Google Maps, to reduce loading times on online maps. Using this approach means that maps are broken into several smaller units. Instead of loading the whole map as one, the system takes into account which map tiles are currently needed and only loads those that are visible in an asynchronous manner. Combined with the ability to cache tiles, this technique greatly reduces the computational power needed to render maps. It also makes maps more fault tolerant, since if a tile fails to load, it doesn't compromise the whole map [7].

Maps created with Leaflet are bound to a respective HTML element and declared as classes. These classes can be customized by using the JavaScript API of the library. A map class contains all the specifics of a map, which are at its core: the position, size, zoom level and the corresponding tile layer. Thanks to the open-source mapping community Leaflet has the ability to bind countless different tile layers to web maps, expanding the range of customization.

2.2.4 SocketIO

Real time capabilities for tracking users, are handled by SocketIO [8], a library that enables low-latency, bidirectional and event-based communication between a client and a server.

Under the hood the library uses a couple different protocols to achieve bidirectional communication, but the preferred method is to create WebSocket connections. If these fail, the library will use HTTP long polling as fallback [9].

2. CHAPTER: TECHNOLOGIES

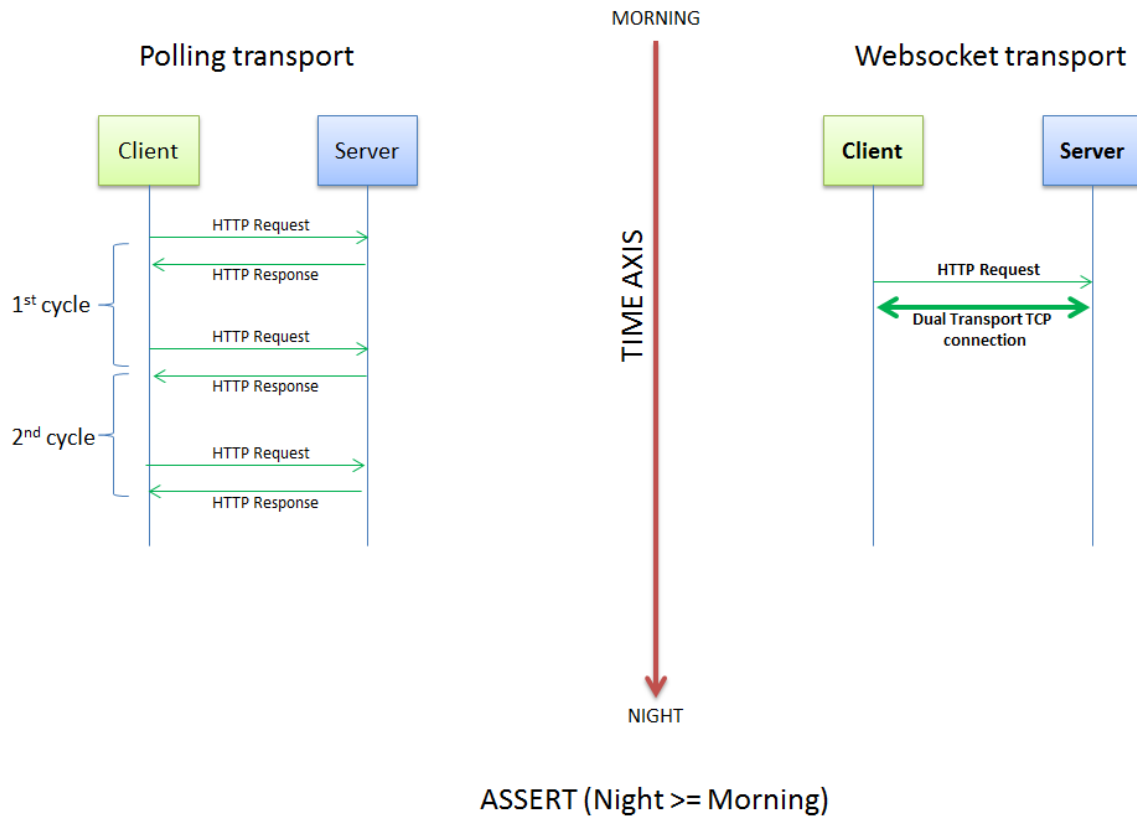


Figure 2.2: Comparing HTTP Polling to WebSocket connections. <https://mashhurs.wordpress.com/2016/09/30/polling-vs-websocket-transport/>

The WebSocket protocol allows for fully real time communication between a web server and a browser. It does this by providing a way for the server to send data to the client, without the client requesting it. It is faster and results in lower overhead compared to HTTP polling, which uses successive HTTP requests to achieve the same real time behaviour [9].

2.3 Backend

The server side handles a wide range of tasks, but it's main purpose in our application is data manipulation and transmission to the frontend. It encompasses a RESTful API, a real-time server and a database.

The RESTful API written in Node.js, allows the client side to interact with the data stored in our applications MongoDB database. It also handles authentication/authorization with JWT, thus allowing our system to have a robust and well defined user role system.

2. CHAPTER: TECHNOLOGIES

The real-time server was also written in Node.js and uses SocketIO's server side implementation to stream the location data to the frontend.

2.3.1 Node.js

Node.js [10] is a JavaScript runtime environment that allows developers to write server side applications such as: web servers, CLI tools, and automated scripts.

It runs on a single thread and is asynchronous thus making it lightweight and perfect for non-blocking, event driven systems with high data traffic. By transporting JavaScript to the server it also allowed a wide range of frontend developers to write server side code, without needing to learn a new language. Under the hood it is powered by Google's V8 engine, the same tool used to parse and execute JavaScript in browser such as Chrome and Microsoft Edge.

Since it's release it has become one of the most popular choices for web servers, and was chosen for our application because of the high degree of compatibility it offers with our client side code.

2.3.2 Fastify

```
await server.register(fastifyEnv, configOptions);

await server.register(authPlugin);
server.register(router, { prefix: "/v1" });

server.register(fastifyCors, corsOptions);
```

Figure 2.3: Example of Fastify Plugins being declared in an ordered manner. The `authPlugin` has access to the added functionalities of the previous plugin but not the other way around.

Fastify [11] is a JavaScript web framework, chosen for our project because of it's focus on developer satisfaction, low overhead and plugin based ecosystem.

The framework provides all the functionalities needed in order to create a powerful REST API. It has a built-in router that can redirect HTTP requests to their corresponding routes, supports custom middlewares for authentication and authorization, allows the developer to configure the servers CORS (Cross Origin Resource Sharing) settings, supports JSON based schemas

2. CHAPTER: TECHNOLOGIES

for data validation and output serialization, provides TypeScript support for extra safety and has an extendable logging system for better debugging.

Most of these functionalities can be extended or customized through plugins. Fastify plugins aim to be encapsulated and inheritable while also avoiding problems caused by cross-dependencies. This behaviour is achieved because each new plugin creates a new scope, which allows the changes to be applied in the descendant plugins but not its ancestors [12]. As seen in Fig. 2.3 for example by declaring the plugin that handles environment variables first and the plugin that handles authentication after, it will allow the `authPlugin` to access the added functionality of the previous plugin (environment variables), but not vice-versa.

2.3.3 MongoDB

Since its release, MongoDB [13] has become the most popular NoSQL database program. It stores data as documents in its own custom format called BSON which is a binary representation of JSON documents. Documents then are gathered into collections that roughly resemble tables in SQL databases. A database is comprised of at least one collection.

The fact that the data is stored and also queried in a format almost identical to JSON makes it an ideal database choice to use with Node.js.

2.3.4 JWT

For the authorization system, the project uses a JWT implementation on the server side.

JSON Web Token or JWT [14] is an open standard, by which applications can transmit data securely using the JSON format.

The basic idea is that the token is digitally signed with a key. The tokens themselves are encrypted using this key and can contain a wide range of information. By using the same key, the receiving end can decrypt the data then and read it.

A JWT token is made up of three parts: header, payload and signature.

1. The header consists of two parts: the type of the token, and the algorithm that was used to sign it.
2. The payload contains the transmitted data.

2. CHAPTER: TECHNOLOGIES

3. The signature is generated using the hashing algorithm provided in the header, a secret, the encoded payload and header. Its role is to provide an additional layer of safety, because by using it the party receiving the message can check whether the token has been tampered with.

All three parts of the token are Base64 encoded. JWT's are easy to integrate into our backend code by using the `jsonwebtoken` library for Node.js.

| Encoded | Decoded |
|---|--|
| <pre>eyJhbGciOiJIUzI1NiIsI nR5cCI6IkpXVCJ9.eyJz dWIiOiIxMjM0NTY3ODkwI iwibmFtZSI6IkpvaG4gRG9 lIiwiaWF0IjoxNTE2MjM5 MDIyfQ.SflKxwRJSMeKKF 2QT4fwpMeJf36P0k6yJV_ adQssw5c</pre> | <pre>HEADER: { "alg": "HS256", "typ": "JWT" } PAYLOAD: { "sub": "1234567890", "name": "John Doe", "iat": 1516239022 } VERIFY SIGNATURE HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), your-256-bit-secret) <input type="checkbox"/> secret base64 encoded</pre> |

Figure 2.4: Structure of a JSON Web Token. The left side shows the encoded form with all three parts and the right side the decoded form.

3. Chapter

Infrastructure

The following chapter is about the infrastructure behind our application. The term infrastructure used here refers to all the operating systems, tools and storage-arrays that helped us manage, test and deliver the application.

3.1 Microsoft Azure

Microsoft Azure is a platform developed and maintained by Microsoft, through which it offers a wide range of cloud computing products.

The deployed versions of the project, both on client and server side are hosted on this platform. In the following section we'll present the services used by the application.

3.1.1 Static Web Apps

Azure Static Web Apps [15] are a service provided by Microsoft through its Azure cloud platform, in order to help developers deploy web apps from their code repositories.

The service is specifically tailored towards hosting apps developed with web technologies or frameworks such as: plain HTML with vanilla JavaScript, React, Angular or Vue. Since all of these technologies can be bundled into static files (HTML, JavaScript and CSS), they all get served from a single server.

Static Web Apps are also well integrated with code hosting platforms such as GitHub and Azure DevOps, thus allowing for easier building and deployment for projects hosted on this platform. It also comes with a Free SSL certificate and the option to add custom domains through third-party DNS providers. The UI of this project is served through this service.

3. CHAPTER: INFRASTRUCTURE

3.1.2 App Service

Azure App Service [16] is a Platform as a Service offering for building HTTP-based web apps, REST APIs or any type of backend project in either Windows or Linux based environments. Like most Azure products it come with full DevOps capabilities: continuous deployment from GitHub or other code hosting platforms, TLS/SSL certificates, multiple environments, and package management. The runtime environment support containers trough Docker, or it can be set to handle a specific language.

Both REST API and WebSocket server of this project have been deployed trough this service.

3.2 GitHub

GitHub is a development platform, primarily used for hosting repositories of projects created with the Git version controlling software. Alongside repositories it also hosts wikis, and offers a wide range of services for developers such as: bug tracking, task management and continuous integration. The whole projects source code is currently hosted on GitHub in multiple repositories.

3.2.1 GitHub Actions

GitHub Actions [17] is a continuous integration and deployment platform, that has been used to build, test and deploy the source code of this project trough a customized pipeline.

GitHub Actions contain workflows that get triggered when an event occurs in the code's repository, such as a push or a pull request being merged. These workflows are defined trough a YAML file in the projects corresponding folder. A single workflow contains at least one job which can run jobs in sequential order or in parallel. Each job will run inside its own virtual machine runner, or inside a container, and has at least one step that runs a small script. The runner is a server that when the correct event is triggered, runs the workflow in a virtual machine.

By using this service, we where able to define a sequence of events that would at build, run the corresponding tests and deploy the source code onto the projects cloud hosting platform, whenever a new commit has been pushed onto a certain branch.

3. CHAPTER: INFRASTRUCTURE

```
name: Build and deploy Node.js app to Azure Web App - api-for-map-app

on:
  push:
    branches:
      - main
  workflow_dispatch:

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4

      - name: Set up Node.js version
        uses: actions/setup-node@v3
        with:
          node-version: '18.x'

      - name: yarn install, build
        run: |
          yarn install
          yarn build

      - name: Zip artifact for deployment
        run: zip release.zip ./* -r

      - name: Upload artifact for deployment job
        uses: actions/upload-artifact@v3
        with:
          name: node-app
          path: release.zip

  deploy:
    runs-on: ubuntu-latest
    needs: build
    environment:
```

Figure 3.1: A YAML file describing a Github Action.

3.3 MongoDB

Primarily being a database management system, MongoDB also offers cloud based solutions in order to help developers deploy their databases alongside their apps.

3.3.1 Clusters

MongoDB clusters [18] are replicated sets. Replicated sets in turn mean multiple servers, populated with the same data. This approach of duplicating database servers is widely used in production grade apps, in order to ensure a high availability of the stored data, and to minimize potential losses by added redundancy.

MongoDB clusters are hosted on cloud platforms such as Microsoft Azure or Amazon Web Services and can be managed through MongoDB's Atlas program or their web interface.

The database of this project was chosen to be hosted through this service, in order to minimize the effort that comes from maintaining and setting up a custom environment for a NoSQL database.

3.4 Testing

Thorough testing of the project was necessary in order to see how well the application held up against its requirements. While manual testing has played a big role in verifying whether certain functionalities were working right, it also proved to be quite time consuming and hard to manage.

The biggest issue with it stems from the fact, that once a flow and behaviour is confirmed to be working well, it doesn't really get tested again after new features have been introduced. This can lead to unpredictable bugs appearing whenever a new feature has been introduced. In order to avoid this, we added automated tests to the deployment pipeline, that would check if previous functionalities were not compromised by new features.

3.4.1 Vitest

The testing framework used for running our automated tests is Vitest.

3. CHAPTER: INFRASTRUCTURE

All files

91.24% Statements 2565/2811 84.96% Branches 356/419 77.57% Functions 128/165 91.24% Lines 2565/2811

Figure 3.2: Coverage report generated by Vitest.

It comes with an easy to use API, that aims to be Jest compatible (the most popular JavaScript testing framework), TypeScript support, easy configuration and a fast execution time [19]. It was chosen for this project specifically because it integrates really well with Vue and its development server Vite on the frontend, while also being easy to setup for API testing on the backend.

To measure the quality of our tests we used Vitest's coverage provider. This allowed us to better identify the parts of our code that have not been properly tested.

4. Chapter

Presenting the Application

The application provides a couple different functionalities that aim to help tourists and their guides to better explore certain areas, whether it be in nature or a big city.

Since the use cases differ based on whether the user is a guide, an admin or a tourist, we created a role based system to best separate these. Thus user accounts in our app belong to one of the three categories and the corresponding functionalities have been crafted accordingly.

4.1 Roles

For better understanding of the different functionalities, first we have to understand the different types of roles users can have. These are as follow:

Admins - These have the ability to view, edit, change roles and delete other accounts. It is the only type of user role that can respond to incoming requests from other users that also want to become admins.

Guides - These can create trails, update existing ones and track users in real time that are hiking on trails created by them.

Users - These can view, filter and sort trough trails created by guides. Users can also opt to subscribe to specific trails, enabling them to simulate hikes on these specific trails.

4.2 Registration

The initial action available to users in our application is to register a new account. The application only requests necessary data for its functionalities, no additional information is collected. The necessary fields are: an *email account*, a *password*, a *role* and optionally the users *name*.

Balázs's Map App

Register

Name: Email:

Password: Confirm Password:

Role:

Already have an account? [Return to Login](#)

Figure 4.1: Registration form of the application.

To ensure accurate registration, the sign up form includes a second password field for confirmation as seen in subsection 4.1. The registration process will fail if the passwords entered do not match. The system also checks whether the password is secure enough and whether there are already users in the system with the given email address.

If a user chooses to be an admin, at first his account will be created only with *user* permissions since admin privileges are granted only upon approval. *User* and *Guide* accounts gain automatic access to their respective functionalities. After successful registration, users are automatically logged in and redirected to the landing page.

4.3 Login

The login system was designed with simplicity in mind. Proper authentication only requires the user's email address and password. Upon receiving a valid input combination, users are

4. CHAPTER: PRESENTING THE APPLICATION

redirected to the app's landing page, where they can access functionalities corresponding to their account roles.

4.4 Account settings

Admin Page

Account Settings

| | |
|---|---|
| Email: | Name: |
| <input type="text" value="balazsadmin@mail.com"/> | <input type="text" value="updateName12"/> |
| <input type="button" value="Edit"/> | |

Administration Requests

| |
|--|
| Request to become Admin from adminrequest@mail.com |
| <input type="button" value="Approve"/> <input type="button" value="Reject"/> |

Users

| | |
|---|--|
| Email: | Name: |
| <input type="text" value="krisztauser@mail.com"/> | <input type="text" value="Kriszta 1"/> |
| Role: | |
| <input type="text" value="Guide"/> | |
| <input type="button" value="Edit"/> | <input type="button" value="Delete"/> |

Figure 4.2: Administration page seen by an admin. The first card shows the admins information which can be edited, after that there's an admin request and another users information.

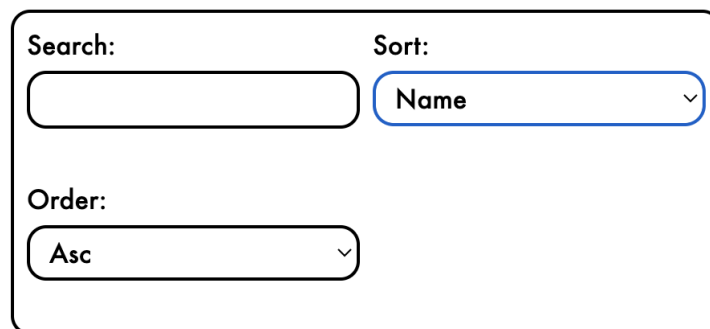
4. CHAPTER: PRESENTING THE APPLICATION

Users, including non-administrators, can edit their account information on the administration page. This includes updating usernames, email addresses, and the option to delete their account if they choose to no longer use the hiking app.

Administrators get additional privileges as seen in Fig 4.2. They can not only edit their own information but also manage other accounts. This includes changing usernames and email addresses, modifying roles, and even deleting accounts if necessary, as shown in Fig. 4.2.

This is also the page where administration requests are displayed for the admins. These get generated when a new user signs up and tries to become an admin. If the administrator accepts the request the user will become an admin as well, if not the request is deleted and the account remains a simple user.

4.5 Interacting with trails



The image shows a rectangular box containing three search and filtering controls. At the top left is a text input field labeled 'Search:'. To its right is a dropdown menu labeled 'Sort:' with 'Name' selected. Below the search field is another text input field. Below the 'Sort:' dropdown is a third dropdown menu labeled 'Order:' with 'Asc' selected.

Figure 4.3: Search box and filtering options as seen on the trails page.

Users registered as guides have the ability to create trails, which consist of a name, a location, and a minimum of two path points. Path points include coordinates (latitude and longitude) and a name.

Guides have the ability to edit trails they've created if mistakes were made during initialization or if updates to the data are necessary. They alongside with administrators can delete trails as well.

The trails page not only showcases all trails but also offers users various options to filter or refine their search, in order to find specific results. These results in turn can be further sorted by name, location, or creator, allowing users to arrange them in either ascending or descending order as seen in Fig 4.3. A personalized version of this page exists for both users and

4. CHAPTER: PRESENTING THE APPLICATION

guides. Guides can access trails specifically created by them through the /my-trails path, while tourists can view trails they've subscribed to via the /subscribed-trails path.

4.6 Real-time tracking functionalities

Balázs's Map App Simulator

Based on the input data, a path of coordinates will be generated. These can be tracked on the Tracking Page

Interval between steps: ⓘ

Send

Number of points: 47

Latitude: 52.214338608258224,
Longitude: -2.691650390625

Latitude: 52.241287943507956,
Longitude: -2.516543552055907

Latitude: 52.26823727875769,
Longitude: -2.341436713486871

Latitude: 52.29518661400742,
Longitude: -2.1663298749177784

Latitude: 52.32213594925715,
Longitude: -1.9912230363487424

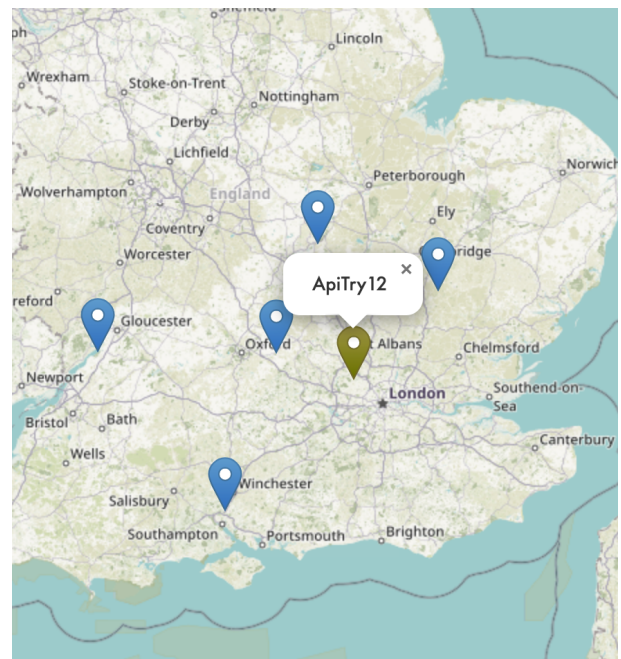


Figure 4.4: Real-tracking pages. On the left is the simulator page with the generated coordinates, and on the right is the live feed of the user's position, viewed by the tour guide.

To showcase the real-time capabilities of our app more effectively, we developed a special simulator page. Here, subscribed users can simulate hiking on that specific trail.

After providing the speed in km/h, the app will generate a set of coordinates that would take the user through each of the path points on that trail. Once these coordinates are generated, the user can set the interval at which they are streamed. After clicking the send button as seen on the left in Fig. 4.4 the app will start streaming the generated coordinates.

4. CHAPTER: PRESENTING THE APPLICATION

Guides can access specific views for the trails they created, allowing them to see the real-time movements of all the users currently hiking on that trail as seen on the right in Fig. 4.4.

5. Chapter

Implementation

After taking a look at the various functionalities of our application, let's dive a bit into the specifics of how it was implemented.

5.1 Database Overview

Since the project uses MongoDB to persist its data, entities are stored as documents. The following types of documents or collections can be found in our system:

users - This collection stores all necessary user data, including a unique ID, email address, name, hashed password, user role, metadata about account creation and last update, and a list of trail IDs. For guides, the trail IDs represent all the trails created by that account, while for users, it holds the list of subscribed trail IDs.

trails - The collection contains data specific to each trail, including a unique ID, name, location, the ID of the user who created it, metadata about its creation and updating, a list of user IDs subscribed to it, and an array of path points. Path points include the name of the location and its coordinates.

admin-requests - This collection stores admin requests created during the registration process. It contains a unique ID, the ID of the user requesting administrator status, the creation date, and the last update date.

5.2 Client Side App

The purpose of the client side is to offer users a clean and user-friendly interface, enabling them to interact with the API and, consequently, with the database.

5. CHAPTER: IMPLEMENTATION

```
_id: ObjectId('66158b4fcf722195e9837f0e')
email: "krisztauser@mail.com"
name: "Kriszta1"
password: "$2a$10$EBu0Tb//IdJ4U6hYQfRXWe7Mza0Y/B/m./qs1ul6QQhBI4CGprIi+"
role: "guide"
▼ trails: Array (1)
  0: ObjectId('661562f8ffaf2566a0403b10')
  createdAt: 2024-04-09T18:39:11.548+00:00
  updatedAt: 2024-05-10T15:01:06.261+00:00
  __v: 0
```

```
_id: ObjectId('6638fdb40156280eb6a1c52b')
name: "ads"
location: "asd"
creator: ObjectId('662409c96370e45eaa136c71')
▼ path: Array (5)
  ▼ 0: Object
    ▼ coordinates: Object
      lat: 51.883272964437474
      lng: -0.9832763671875001
      _id: ObjectId('6638fdb40156280eb6a1c52d')
      name: "das"
      _id: ObjectId('6638fdb40156280eb6a1c52c')
    ▶ 1: Object
    ▶ 2: Object
    ▶ 3: Object
    ▶ 4: Object
  ▼ users: Array (2)
    0: ObjectId('6627d34f87387b4ba3ffdcc5')
    1: ObjectId('6645d49d57a96e7d0d9dc2b5')
  createdAt: 2024-05-06T15:56:36.052+00:00
  updatedAt: 2024-05-16T09:40:56.825+00:00
  __v: 0
```

```
_id: ObjectId('6645030a57a96e7d0d9dc25e')
user: ObjectId('6645030a57a96e7d0d9dc25c')
createdAt: 2024-05-15T18:46:34.312+00:00
updatedAt: 2024-05-15T18:46:34.312+00:00
__v: 0
```

Figure 5.1: Instances of the database entities: the top panel is a user document, the middle panel shows a trail, and the bottom panel is an admin request.

5. CHAPTER: IMPLEMENTATION

The entry point of the application is located in the root folder's `src/` directory. This includes a `main.ts` file for loading all plugins and basic configurations, a `style.css` file for global CSS classes, and an `App.vue` file that serves as a gateway to all the different routes of the app. The following section will provide an overview of all the subsystems that make up our frontend architecture.

5.2.1 Components

The UI is divided into several smaller components, each fulfilling a specific role. These components are isolated from one another, which significantly reduces the potential for bugs. This approach also facilitates code reuse across multiple areas [20].

During development, we organized components based on the area of the app they were used in. The components are divided into the following categories: *authentication-related*, *map-related*, *simulator-related*, and *trail-related*. Some components were needed in more than one area and thus were placed in the shared folder. These are:

BaseButton - A general-purpose button component used throughout the app.

BaseInput - An input component supporting multiple types, including numbers, strings, and booleans, with built-in error handling.

BaseModal - A modal component used to warn users before they take certain actions.

LoadingAnimation - An animation displayed while data is loading from the API.

PageLayout - A wrapper component that provides a unified structure for various pages across the app.

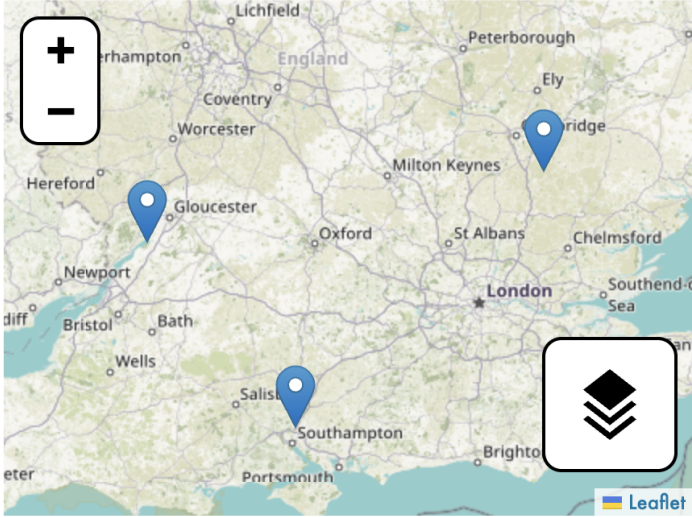
SelectInput - An input component designed for selecting a single value from multiple options.

ToolTip - A simple tooltip component that displays additional information when hovering over an icon.

The other components were more specifically tied to certain areas of the app. For example, the `TrailDisplay` component as seen in Fig. 5.2, is used throughout the app to visualize a trail. It is employed during trail creation and also displays existing trails for both users and guides. Most other components are used in only one or two places and serve to better encapsulate small sections of code.

5. CHAPTER: IMPLEMENTATION

Name: Location:



Name: Latitude: Longitude:

Name: Latitude: Longitude:

Name: Latitude: Longitude:

Figure 5.2: Instance of a TrailDisplay component in edit mode.

5.2.2 Pages and routes

By utilizing multiple components, we were able to construct entire pages. Each page is designed to enable specific actions related to a particular area of the application. By customizing the Vue Router plugin, we connected each page to a specific URL and added redirect rules to

```

const routes = [
  { path: "/", name: "Trails Page", component: GeneralTrailsPage },
  { path: "/tracking/:id", name: "Tracking Page", component: TrackingPage },
  { path: "/simulator/:id", name: "Simulator Page", component: SimulatorPage },
  { path: "/login", name: "Login Page", component: LoginPage },
  { path: "/my-trails", name: "My Trails Page", component: GeneralTrailsPage },
  {
    path: "/subscribed-trails",
    name: "Subscribed Trails Page",
    component: GeneralTrailsPage,
  },
  {
    path: "/administration",
    name: "Administration Page",
    component: AdministrationPage,
  },
  { path: "/*", name: "NotFound Page", component: NotFoundPage },
];

```

Figure 5.3: The pages available on the frontend and their corresponding URLs.

restrict access for users without the necessary permissions. One advantage of using the Vue Router is that it dynamically changes the URL without requiring a full page reload from the server [21]. The routes defined in our application and the pages assigned to them can be seen in Fig. 5.3.

To better manage the different kind of roles and their permissions, we implemented an additional layer of logic here. When a user attempts to access a URL not defined in our list or lacks permission based on their role, the app will redirect them to a custom 404 page.

5.2.3 Stores and Services

Stores are entities that manage and hold state and logic across the app, independent of components. They provide a way to handle global state and share it across multiple components. The recommended library for handling state in Vue is called Pinia [22]. Pinia allows us to write stores that manage instances of all our main entities, from fetching them from the API to handling mutations.

The app contains the following stores: *administration-store*, *auth-store* and *trails-store* and each store in turn is split into three sections:

5. CHAPTER: IMPLEMENTATION

```
// stores/counter.js
import { defineStore } from 'pinia'

export const useCounterStore = defineStore('counter', {
  state: () => {
    return { count: 0 }
  },
  // could also be defined as
  // state: () => ({ count: 0 })
  actions: {
    increment() {
      this.count++
    },
  },
})
```

Figure 5.4: Example of a simple Pinia store. <https://pinia.vuejs.org/introduction.html/>

state - The central part of any store is the state definition. It is declared as a function that returns the initial value of the state [23].

actions - This is where the definitions of business logic and mutations are placed. We use it to communicate with the API and then modify the store's state accordingly [24].

plugins - Pinia store functionalities can be significantly extended through additional plugins. We used the pinia-plugin-persist plugin to persist the state in `localStorage` on the frontend, effectively using it as a makeshift cache [25].

Communication with the API was handled using services. Each service is associated with specific API endpoint and has a separate method for each HTTP verb. These services are then utilized in the stores to fetch or transmit data to the backend using the browser's native `fetch` API.

5.2.4 Composables

```
import { useTrailsStore } from "../stores/trails-store";
import { useRoute } from "vue-router";
import router from "../routing/router";
import { ref, onMounted } from "vue";

export function usePermissionRerouting() {
  const route = useRoute();
  const trailsStore = useTrailsStore();

  const isLoading = ref(true);

  onMounted(async () => {
    const trailResponse = await trailsStore.getTrail(route.params.id as string);
    if (!trailResponse) {
      return router.push({ name: "NotFound Page" });
    }
    isLoading.value = false;
  });

  return { isLoading };
}
```

Source Code 5.1: Example of a composable: This composable fetches the trail based on the ID from the URL. If an error occurs, it reroutes to a 404 page; otherwise, it updates the loading status.

To minimize code redundancy, developers typically write functions whenever they have to reuse logic. These functions accept certain parameters and return values, encapsulating *stateless logic*. However, in some cases within the application, we also need to write helper functions that manage state, making them not entirely side-effect free. When used with Vue-specific APIs, these are called *composables* [26].

We used this programming pattern throughout the app whenever we needed to reuse stateful logic. For instance, when a user attempts to access a simulator or tracking page associated with a certain trail, if they do not have permission to view that trail, the API returns a 404 status code. Ideally, the frontend should then redirect to a 404 page as well. To handle this, we wrote a composable called `usePermissionRerouting`, as seen in Source Code 5.1. The logic in this function depends on the state of the trail fetched from the API (whether the user has permission

to view the trail). Based on this, the page either loads or redirects to another page.

5.2.5 Leaflet

```
//@ts-nocheck

#addDisplayMarker(
  id: string,
  coords: Coordinates,
  popupContent: string,
  isColored = false
) {
  const icon = L.icon({
    iconUrl: markerIcon,
    iconSize: [25, 41],
  });
  const marker = L.marker([coords.lat, coords.lng], { icon })
    .addTo(this.#mapInstance.value)
    .bindPopup(popupContent, {
      offset: [0, -7],
      className: "marker-popup",
    });
  if (isColored) {
    (marker as any)._icon.style.filter = `hue-rotate(${Math.floor(
      Math.random() * (360 - 10) + 10
    )}deg)`;
  }
  this.#markerMapping.set(id, marker);
}
```

Source Code 5.2: The addDisplayMarker function used to add markers to an existing leaflet map.

The visualization of map related data has been done with the Leaflet library. Since Leaflet binds its functionality to a specified DOM element, we had to write a custom service that provides a reusable API for managing the properties on any map we wanted to display, regardless of where it is in the app.

LeafletService is a class that accepts several arguments in its constructor. These required arguments include a reference to the HTML div where the data is bound, the focus coordinates, whether it is an editable map or not, and a list of path points to be displayed on the map. Based

5. CHAPTER: IMPLEMENTATION

on these the system can customize the number of displayed markers on the map, the initial coordinates displayed, and whether the map supports editing of markers and addition of new ones.

The class also includes a private function named `addDisplayMarker` as seen in Source Code 5.2. It is responsible for creating a new marker to be displayed on the map using the provided description and location. By way of public wrapper functions, this method is frequently used across the app whenever new markers need to be added for trail updates, map initialization, and during the live streaming process of user coordinates.

5.3 The API

The API that we implemented on the backend for our the application follows the REST (Representational State Transfer) architecture model. This implies that the server transfers data in a uniform format (in our case JSON), and that it implements a couple of other architectural constraints [27].

When talking about RESTful APIs, each resource is identified by separate requests grouped together based on a URL. For example, all actions related to users would be mapped to the `/users` endpoint, with HTTP verbs specifying the action to be performed. GET is used for retrieving data, POST for creating new entities, PATCH and PUT for updating, and DELETE for removal.

The transmitted data is also formatted in a way that makes it easy to consume on the client side, which may not be identical with its original form in the database.

Statelessness is also a crucial aspect of RESTful API design. It ensures that each request is entirely isolated from others, meaning the data transferred through these requests is independent of any previous or future transactions that the server may have had.

The upcoming chapter will delve deeper into the implementation details of our RESTful API for the application.

5.3.1 The server instance

The main entry point of our project is the `server.ts` file found in the `src` folder's root directory. This is where all the necessary plugins and additional configurations are loaded up

5. CHAPTER: IMPLEMENTATION

and registered with the server instance.

Since we're using the Fastify framework for our backend, every middleware is declared as a plugin. The benefit of this approach is that it allows each plugin to have a separate context, unaffected by changes applied in subsequent declarations. The following list of middlewares are used in our app:

Logger plugin - The Pino logger plugin enables us to customize our server logs. It is low-overhead and offers a wide range of additional features [28].

Global error handling plugin - To override Fastify's default validation errors, we declared a custom plugin that formats these errors, providing the frontend with more easily understandable error messages.

Environmental variables plugin - A plugin that checks whether the environmental variables match the declared schema, and then sets these values into the context of our application.

Authentication middleware - A custom plugin that deciphers the information encoded in the incoming HTTP request's Bearer Token. It is used to catch unauthorized requests before they reach the controller level.

Router - The router file holds the declarations to all the different routes.

CORS settings - A plugin that enables CORS (Cross-Origin Resource Sharing) in our application. This adds an extra layer of security by allowing us to specify the permitted HTTP methods and origins with which the API can communicate.

Mongoose plugin - The plugin responsible for establishing and maintaining the database connection using the Mongoose library.

5.3.2 Routes

The routes folder holds each route declaration of our system. A route in this context corresponds to a specific URL that identifies a database entity. These typically support multiple actions, and are differentiated by their respective HTTP verbs. A route also holds additional business logic, such as a schema detailing the expected structure of the request, the controller function that returns the response, and whether authentication is required for the request to be valid.

5.3.3 Controllers

Controllers are the level where most of the business logic of our API is defined. They act as a bridge between the API and the database. They typically take the form of a function that accepts the incoming request and first validates it. If there's an issue with the request's format, the controller will return a specific error. If the request is valid, the controller interacts with the service layer to perform an action with the database and then returns the result in a format that is easy to interpret on the frontend.

5.3.4 Services

The service layer is where our app interacts with the database. Methods in this layer use Mongoose, a library that simplifies writing queries and formatting results. Each service is specific to a database entity, and each function within a service interacts with the database differently, either by querying it, modifying it, or removing data from it.

5.4 The WebSocket Server

The purpose of the WebSocket server is to facilitate the transmission of data from the tourist to the corresponding guide during the streaming process. It is a simple server running on Express, a minimalist JavaScript framework.

The WebSocket server has two namespaces declared: one for listening to updates and the other for transmitting them to the appropriate destination. Namespaces are specific communication channels that allows us to separate logic over the same connection. The system works as follows: when the server detects a new connection on the `/updates` namespace, it will take the received ID, create a new event called `update-location-{connection.id}`, and transmit the received data to the connections listening on that event.

6. Chapter

Potential Improvements

During development our main focus was to satisfy the needs of all user types. This led to a wide range of functionalities being implemented across the app, but also took time away that could have been used to better flesh out the streaming process and the map visualization.

One potential additional feature would be the ability to persist the streamed data of the users hiking on trails. Currently the data is only used to present the real time location, but it isn't saved anywhere. Once this data is accessible, it would unlock a whole range of potential features. We could analyze it in various ways and allow the guides to review the results. Guides could then optimize the trails that they created based on this information and thus deliver better options for tourists.

Another feature that could improve the app would be an offline option for recording hikes on certain trails. Mobile signal isn't always available, especially in more remote areas, and this way users could still record hikes, and then the app would persist it in the database once the connection is established again. This feature could be iterated on, by adding an option for users to set the frequency of their transmission. This way they could customize the performance of the app on their device.

And at last, the map visualization could be improved by displaying the lines between the different path points on a map, as well as the route taken during hikes streamed by users. This would give more accurate data about the preferences of the different users and whether or not they manage to follow the paths or not.

7. Chapter

Conclusions

The purpose of creating this hiking app was to develop a platform that connects potential tourists interested in a specific area with local guides who want to promote it. In order to achieve this goal, we leveraged all the possible use cases that could come up for the different types of parties interested in a solution like this.

Tourists are provided with a simple and elegant platform where they can browse multiple trails, search by specifics, and filter or organize the results. A personal list of preferred trails can be created by subscribing to them. Hikes on these trails can then be streamed to the guides, adding a layer of security for tourists unfamiliar with the area.

For guides the app presents a clean User Interface through which they can manage existing trails, create new ones or even delete old trails if they wish to do so. By monitoring the real time movements of users on these trails, they get a better idea about user behaviour, which paths are popular and what areas are hard to access.

Users who would like to monitor the community and manage the accounts associated with tourists and guides, can apply for an admin role. Administrators get unlimited access to the applications features, they can also approve other admins and edit user accounts.

The resulting software was built with these varying requirements in mind, and by utilizing a wide range of modern web development technologies, delivers an easily accessible platform with low performance overhead, featuring a responsive and intuitive UI for all its potential users.

Bibliography

- [1] IDG News Service staff. Microsoft augments javascript for large-scale development. <https://www.infoworld.com/article/2614863/microsoft-augments-javascript-for-large-scale-development.html>, 2012. Visited on: 2024.05.04.
- [2] Typescript official documentation. <https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html>, 2024. Visited on: 2024.05.04.
- [3] Vue official documentation. <https://vuejs.org/guide/introduction.html>, 2024. Visited on: 2024.05.04.
- [4] Vue rendering mechanism. <https://vuejs.org/guide/extras/rendering-mechanism>, 2024. Visited on: 2024.05.04.
- [5] Tailwind css documentation. <https://tailwindcss.com/docs/installation>, 2024. Visited on: 2024.05.04.
- [6] Leaflet overview. <https://leafletjs.com/index.html>, 2024. Visited on: 2024.05.05.
- [7] Matt Forrest. A brief history of web maps. *towardsdatascience.com*, 2021.
- [8] Introduction to socket.io. <https://socket.io/docs/v4/>, 2024. Visited on: 2024.05.05.
- [9] How socket.io works. <https://socket.io/docs/v4/how-it-works/>, 2024. Visited on: 2024.05.05.
- [10] Node.js documentation. <https://nodejs.org/en/learn/getting-started/introduction-to-nodejs>, 2024. Visited on: 2024.05.05.
- [11] Fastify documentation. <https://fastify.dev/>, 2024. Visited on: 2024.05.05.
- [12] Fastify plugins. <https://fastify.dev/docs/latest/Reference/Plugins/>, 2024. Visited on: 2024.05.05.
- [13] Mongodb core documentation. <https://fastify.dev/docs/latest/Reference/Plugins/>, 2024. Visited on: 2024.05.05.
- [14] Jwt documentation. <https://jwt.io/introduction>, 2024. Visited on: 2024.05.05.
- [15] What is azure static web apps? <https://learn.microsoft.com/en-us/azure/static-web-apps/overview>, 2024. Visited on: 2024.05.07.
- [16] App service overview. <https://learn.microsoft.com/en-us/azure/app-service/overview>, 2024. Visited on: 2024.05.07.
- [17] Understanding github actions. <https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions>, 2024. Visited on: 2024.05.07.
- [18] Mongodb clusters. <https://www.mongodb.com/resources/products/fundamentals/clusters>, 2024. Visited on: 2024.05.07.
- [19] Vitest features. <https://vitest.dev/guide/features.html>, 2024. Visited on: 2024.05.12.
- [20] Components basics. <https://vuejs.org/guide/essentials/component-basics>, 2024. Visited on: 2024.05.19.
- [21] Vue router documentation. <https://router.vuejs.org/guide/>, 2024. Visited on: 2024.05.19.
- [22] Pinia documentation. <https://pinia.vuejs.org/getting-started.html>, 2024. Visited on: 2024.05.20.
- [23] Pinia states. <https://pinia.vuejs.org/core-concepts/state.html>, 2024. Visited on: 2024.05.20.
- [24] Pinia actions. <https://pinia.vuejs.org/core-concepts/actions.html>, 2024. Visited on: 2024.05.20.
- [25] Pinia persisted state. <https://seb-l.github.io/pinia-plugin-persist/advanced/strategies.html>, 2024. Visited on: 2024.05.20.

BIBLIOGRAPHY

- [26] Vue composables. <https://vuejs.org/guide/reusability/composables>, 2024. Visited on: 2024.05.19.
- [27] What is a restful api? <https://aws.amazon.com/what-is/restful-api/#:~:text=RESTful%20API%20is%20an%20interface,applications%20to%20perform%20various%20tasks.>, 2024. Visited on: 2024.05.20.
- [28] Pino logger documentation. <https://getpino.io/#/docs/pretty>, 2024. Visited on: 2024.05.20.