FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Sports equipment tracker
Student:	Karel Zanáška
Supervisor:	Ing. Filip Glazar
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2023/2024

Instructions

Cílem práce je vytvořit webovou aplikaci, která bude sloužit ke správě sportovního vybavení. V aplikaci si uživatel bude moci vytvořit různé typy sportovního vybavení jako bicykl, běžeckou obuv, tenisovou raketu a další. K vybavení bude možné přiřadit sportovní aktivity a sledovat tak využití a možné opotřebení daného vybavení. Tyto aktivity mohou být zadány jak manuálně, tak pomocí integrace s platformou Strava. Uživatel bude například schopen zjistit, že už by si měl vyměnit řetěz na jízdním kole, protože je opotřebený, jen na základě jeho sportovních aktivit. Hlavním zdrojem těchto dat by měla být právě platforma Strava.

Ideálně postupujte dle následujících kroků:

1) Analyzujte možnosti integrace s platformou Strava.
2) Specifikujte funkční požadavky aplikace.
3) Proveďte volbu vhodných technologií pro daný typ softwaru např. programovací jazyk TypeScript a framework React.
4) Navrhněte a implementujte alespoň prototyp aplikace.
5) Otestujte vhodnými testy důležité komponenty aplikace.
6) Zhodnoťte výsledky testování a navrhněte potřebné úpravy aplikace.
7) Připravte aplikaci pro nasazení v produkčním prostředí.

Electronically approved by Ing. Michal Valenta, Ph.D. on 23 February 2023 in Prague.

Bachelor's thesis

SPORTS EQUIPMENT TRACKER

Karel Zanáška

Faculty of Information Technology
Department of Software Engineering
Supervisor: Ing. Glazar Filip
May 11, 2023

Citation of this thesis: Zanáška Karel. *Sports equipment tracker*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Contents

List of Figures

List of Tables

List of code listings

I would like to take this opportunity to express my gratitude to my supervisor, Ing. Filip Glazar, for his guidance and support throughout this work. His invaluable feedback and advice have been instrumental in shaping this thesis.
I would also like to express my heartfelt thanks to my family, especially my mother, for her constant encouragement, love, and support. Their belief in me has been a source of strength and inspiration throughout my educational journey.
Finally, I would like to sincerely thank my friends for their encouragement and support during my studies.

Declaration

In Prague on May 11, 2023 . . . .

Abstract

This bachelor thesis aims to design and develop a web application that allows users to track the wear and tear of their sports equipment. The application allows users to manually create activities or sync new activities from the Strava platform. It also allows the creation of sports equipment and its components. Wear and tear of the equipment or its components is monitored based on distance and time used in activities. The thesis includes a theoretical section that analyses the application requirements, integration with Strava, and architecture and technologies for web application development. It also includes a practical section that describes the development process, testing, and deployment of the application. The thesis concludes with a summary of the work and possible extensions to the application.

Keywords Sports equipment tracking, Single-page application, Strava API, React, Django REST framework, Docker, GitHub Actions

Abstrakt

Tato bakalářská práce si klade za cíl navrhnout a vyvinout webovou aplikaci, která umožní uživatelům sledovat opotřebení jejich sportovního vybavení. Aplikace poskytuje uživatelům možnost ručně vytvářet aktivity nebo synchronizovat nové aktivity z platformy Strava. Umožňuje také vytváření sportovního vybavení a jeho komponent. Opotřebení vybavení nebo jeho komponent se monitoruje na základě vzdálenosti a času stráveného při aktivitách. Práce obsahuje teoretickou část, která analyzuje požadavky na aplikaci, integraci se Stravou, architekturu a technologie pro vývoj webové aplikace. Dále obsahuje praktickou část, která popisuje proces vývoje, testování a nasazení aplikace. V závěru práce je uvedeno shrnutí práce a možná rozšíření aplikace.

Klíčová slova sledování sportovního vybavení, jednostránková webová aplikace, Strava API, React, Django REST framework, Docker, GitHub akce

List of abbreviations

API	Application Programming Interface
CSS	Cascading Style Sheets
DOM	Document Object Model
DRF	Django REST framework
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
JSON	JavaScript Object Notation
ORM	Object-relational mapping
REST	REpresentational State Transfer
SPA	Single-page-application
URL	Uniform Resource Locator
URI	Uniform Resource Identifier

Introduction

Sport is an essential part of the lives of many people. There are numerous positive effects of sport on our health and overall fitness. For various sports, we need specific equipment. For running, we need running shoes; for playing tennis, we need a tennis racket; for cycling, we need a bike, helmet, cycling shoes, etc. To prevent injuries during sports, it is essential to maintain and monitor the wear and tear of the equipment. As an illustration, it is recommended to replace running shoes after covering a distance of 500 kilometers. For a bike, it is also necessary to monitor the wear and tear of its individual components, such as the chain, which should be generally replaced after 3,000 to 6,000 km.

Thanks to current technology, it is possible to use smart devices, such as a smartwatch, a smartphone, or a speedometer with GPS, to record the activity and then upload it to a preferred platform. One of these most popular platforms is Strava, which offers a lot of additional functionality beyond activity recording but does not offer advanced sports equipment management.

We decided to create a web application that helps the user to monitor the condition of sports equipment. Equipment wear and tear will be measured based on activities that can be automatically imported from the Strava platform or created manually.

In the first chapter, we deal with the requirements and capabilities of the application. Then we analyze the Strava platform and possible integration with it. We also describe alternative solutions and their limitations.

In the second chapter, we introduce the general principles of how web applications work and describe different approaches to building web applications. We explain which architecture we have chosen. Then, we discuss the appropriate technologies for developing the web application.

In the third chapter, we describe the implementation of a web application using selected technologies based on the given requirements. We present in detail how to integrate the Strava platform with our application.

In the fourth chapter, we deal with testing the application. We introduce ways of testing web applications, and we implement appropriate ones.

In the fifth chapter, we describe how the application is securely deployed. Application deployment is automated as much as possible, so it is easy to deploy a new release.

In the last chapter, we summarize the work and describe possible future application extensions.

Aim of the thesis

The aim of this work is to design a web application that allows people to track the wear and tear of their sports equipment based on activities from the Strava platform.

Within the application, the user will be able to optionally link their account to the Strava platform. If they choose to do so, the activities uploaded to Strava will be automatically synchronized with their account. Additionally, they will be able to manually create activities within the app. The user will have the option to create different types of sports equipment, such as running shoes, a tennis racket, or a bike. They will also be able to track individual components of the equipment, such as the chain on a bike. The equipment or its components will be able to be associated with activities, as wear and tear on the equipment are tracked based on distance or time used in activities.

The first part of the thesis is theoretical. It deals with the analysis of the application requirements and possible integration with the Strava platform. The selection of suitable technologies for the creation of web applications is also presented.

The second part of the work is practical. This part deals with the process of developing an application using the selected technologies, testing, and deployment.

Chapter 1

Analysis and design

1.1 Requirements

Software requirements describe the features, functions, and capabilities of the software to meet the needs of its users. Requirements can be divided into two categories. [1]

1.1.1 Functional requirements

Functional requirements describe the interactions between the system and its environment.

FR1: The user should be able to register and log in to the application.

FR2: The user should be able to link the Strava account to the app to be able to receive new activities from Strava. They should also be able to remove the Strava account.

FR3: The user should be able to manually create and delete activities.

FR4: The user should be able to create and delete sports equipment.

FR5: The user should be able to assign and remove components to each sports equipment.

FR6: The component should be able to be present on multiple sports equipment at the same time.

FR7: The user should be able to assign sports equipment to activities.

FR8: The user should be able to track the wear and tear of sports equipment and its components based on distance or time used in activities.

1.1.2 Non-functional requirements

Non-functional requirements describe general software characteristics such as usability, reliability, performance, and error handling.

NFR1: The application uses a token-based secure authentication method. All communication between the application and the client is encrypted.

NFR2: The application's graphical user interface is in English.

NFR3: The application is publicly available on the Internet and is accessible through a domain name.

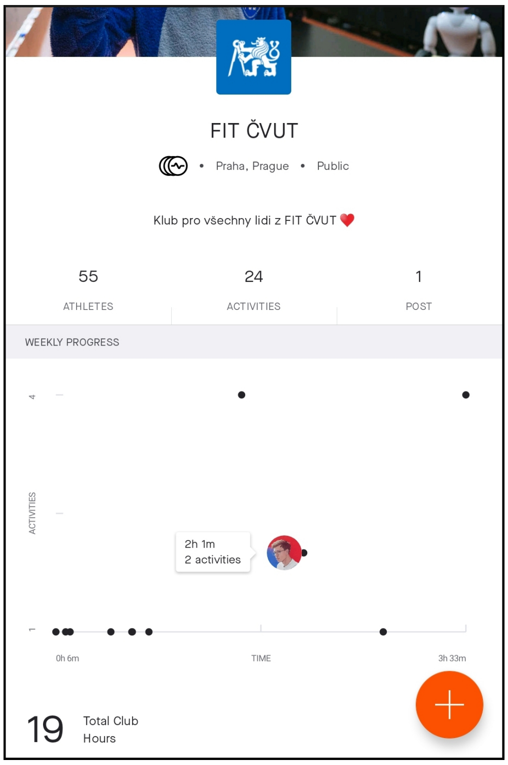
NFR4: The application is usable on various devices, such as a computer or smartphone.

1.2 Strava service

1.2.1 Overview

Strava is a fitness app that has become popular among sports hobbyists and athletes. The app allows users to track and analyze sports activities such as cycling, running, swimming, hiking, etc. It is available on the Android and iOS platforms, as well as on the Web. The user can record activity directly via the Strava app on their mobile phone or using a sports watch or bike speedometer. The application also has social functions that allow users to share their activities with friends. The user may also join various clubs, such as the FIT CTU club[1], as shown in the figure 1.1. One of Strava's unique features is its leaderboard, where users can compete with other athletes who have successfully completed similar activities. In addition to its basic features, Strava also offers premium subscriptions that provide advanced analysis tools and personalized training. [2]

Figure 1.1 Screenshot of the FIT CTU club on the Strava platform [3]



1.2.2 Strava API

Strava offers an API for public use. Unlike a web or mobile app, where data for public profiles are publicly available, the API only provides data about the logged-in user. Therefore, the application requires the necessary user permissions, which are obtained using the OAuth 2.0[2] protocol.

[1]https://www.strava.com/clubs/fit-cvut
[2]OAuth 2.0 is the industry-standard protocol for authorization.

The utilization of Strava API is restricted for each application with a cap on the number of requests that can be made in a 15-minute interval and on a daily basis. The default rate restriction allows up to 100 requests every 15 minutes and a maximum of 1,000 requests per day. To avoid these limits and to meet the login and logout requirements, it is necessary to implement webhooks. It is possible to see graphs of request rates there. [4]

1.3 Existing solutions

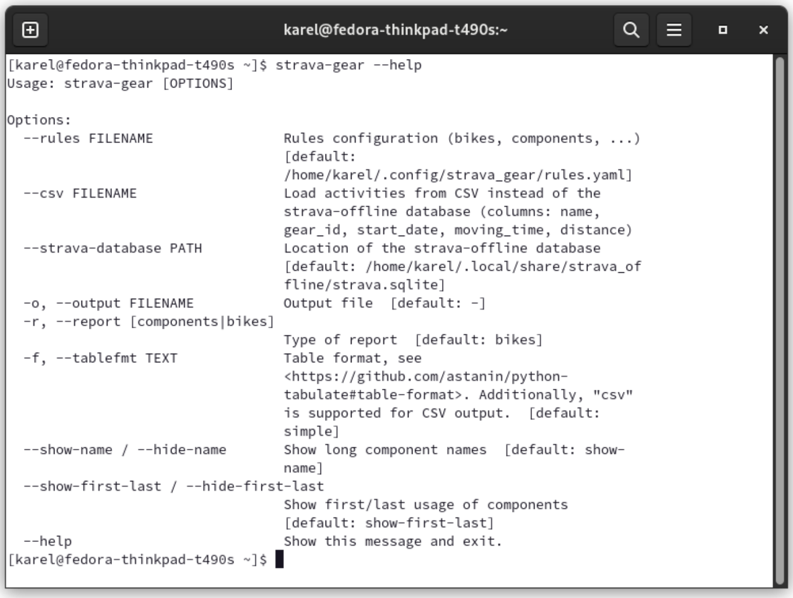
There are no applications that meet the specified requirements. There exist alternative applications that integrate the Strava platform, but they are only related to bike management.

1.3.1 Strava

The Strava platform itself provides simple sports equipment management. The user can assign a bicycle or shoes to their account, which they can then add to their activities. Other kind of equipment are not supported. [5] The Strava API 1.2.2 offers an endpoint from which information about the user's equipment can be retrieved.

1.3.2 strava-gear

strava-gear is a command line application that allows users to manage their bikes and their components based on activities from the Strava platform. The main advantages of the application are that it is open source under the MIT license[3] and does not require a third-party server to be running, as the data are stored locally by the user. The main drawbacks are that the application offers only a text-based user interface and also that it is limited to bike management only. [6]



```
[karel@fedora-thinkpad-t490s ~]$ strava-gear --help
Usage: strava-gear [OPTIONS]

Options:
  --rules FILENAME                Rules configuration (bikes, components, ...)
                                  [default:
                                  /home/karel/.config/strava_gear/rules.yaml]
  --csv FILENAME                  Load activities from CSV instead of the
                                  strava-offline database (columns: name,
                                  gear_id, start_date, moving_time, distance)
  --strava-database PATH          Location of the strava-offline database
                                  [default: /home/karel/.local/share/strava_of
                                  fline/strava.sqlite]
  -o, --output FILENAME           Output file  [default: -]
  -r, --report [components|bikes]
                                  Type of report  [default: bikes]
  -f, --tablefmt TEXT             Table format, see
                                  <https://github.com/astanin/python-
                                  tabulate#table-format>. Additionally, "csv"
                                  is supported for CSV output.  [default:
                                  simple]
  --show-name / --hide-name       Show long component names  [default: show-
                                  name]
  --show-first-last / --hide-first-last
                                  Show first/last usage of components
                                  [default: show-first-last]
  --help                          Show this message and exit.
[karel@fedora-thinkpad-t490s ~]$
```

Figure 1.2 Screenshot of the strava-gear console app

[3]Software license allowing broad reuse, modification, and distribution.

1.3.3 ProBikeGarage

ProBikeGarage is a mobile application for Android and iOS platforms. Its main advantages include an extensive database of available bike components and a user-friendly interface. Drawbacks include being tied to a mobile device and being limited to bike management. [7]

1.3.4 Conclusion

Based on the analysis of existing solutions, there is a space for a web application that allows the management of sports equipment beyond bikes. There are applications for bike management, but they are limited to use on a mobile phone or in the command line. They also do not allow managing all kinds of sports equipment. A web application would allow the user to track their sports equipment and its components on different platforms.

Chapter 2 Architecture and technology analysis

2.1 Architecture

This section first looks at the general concepts used in all web applications. These include the Hypertext Transfer Protocol, Hypertext Markup Language, Document Object Model, JavaScript Object Notation, and Web API types. Then possible approaches to creating web applications are described and compared. Finally, the selection of a preferred approach that would best suit the requirements of the application is justified.

2.1.1 HTTP

HTTP (Hypertext Transfer Protocol) is a client-server protocol. HTTP is widely used in web applications. In these applications, requests are sent by a user agent that acts on the user's requests. The role of the user-agent is usually performed by the web browser. HTTP is a stateless protocol, which means that each client request contains all the information needed for the server to process the request without any state being stored on the server. When processing requests, the server does not cache any data, and each request is processed independently. [8]

The HTTP protocol defines several methods that describe the operation to be performed on a resource. Each method serves a different purpose. [9]

GET: Retrieves a representation of the specified resource.

HEAD: Is identical to GET but without response points.

POST: Submits an entity to the specified resource.

PUT: Updates a current representation of the specified resource.

DELETE: Deletes the specified resource.

CONNECT: Establishes a connection with the server by the target resource.

OPTIONS: Retrieves communication options for the target resource.

TRACE: Conducts a diagnostic test for the target resource.

PATCH: Performs a partial modification for the target resource.

There are four versions of HTTP: HTTP/0.9, HTTP/1.0, HTTP/1.1, and HTTP/2.0. HTTP messages defined in HTTP/1.1 and earlier versions are human-readable, unlike HTTP/2 where HTTP messages are embedded in a binary structure. [10]

2.1.2 HTML

HTTP is a protocol that can be used to transfer various formats such as text or binary files. The most widely used format is HTML.

HTML (Hypertext Markup Language) is a language used to describe a document and its formatting. Documents are composed of elements that are defined by tags.

Tags are used to define the beginning and end of elements, and they can be nested within each other. For example, they can define headings, paragraphs, lists, tables, or images. Tags can contain attributes that define additional information about the element, such as the element id or size.

HTML allows the user to create links between the documents using the tag `<a>` and its attribute `href`. For example, `<a href="http://fit.cvut.cz/en">FIT CTU</a>` is a link that redirects the user to the FIT CTU website[1]. See the listing 2.1 for an illustration of use within the whole HTML document. [11]

2.1.3 CSS

CSS (Cascading Style Sheets) is a style sheet language used to describe the appearance of elements in an HTML document, although it can also be used for other languages such as XML[2]. CSS can be used to specify the layout of elements, fonts, colors, and other graphical details. CSS can be either inside the file itself or it can be embedded in the HTML document, as shown in listing 2.1. [11]

2.1 Sample of HTML document with CSS

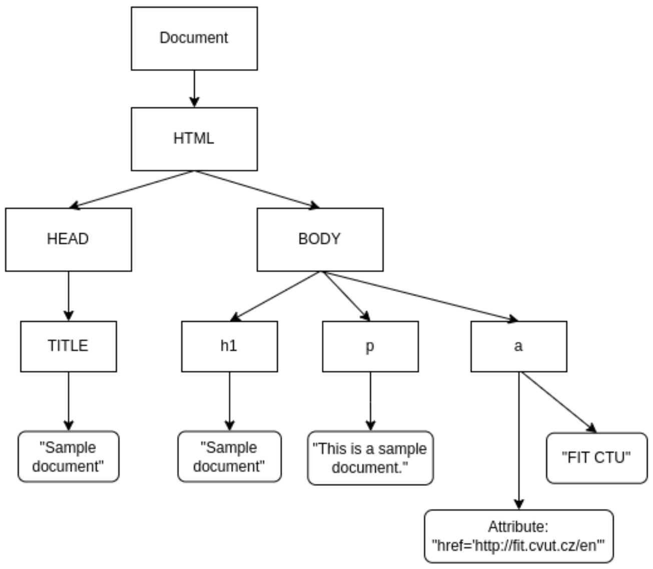
```html
<!DOCTYPE HTML>
<html>
    <head>
      <title>Sample document</title>
      <style type="text/css">
        body {
            background-color: gray
        }
        h1, h2, h3, h4, h5, h6 {
            text-align: center;
        }
      </style>
    </head>
<body>
    <h1>Sample document</h1>
    <p>This is a sample document.</p>
    <a href="http://fit.cvut.cz/en">FIT CTU</a>
</body>
```

[1] https://fit.cvut.cz/en
[2] Markup language for data exchange.

2.1.4 DOM

DOM (Document Object Model) is a standard developed by W3C[3] that specifies a Document interface for accessing and modifying individual elements of an HTML or XML document. [12] The Document interface treats the document as a tree structure, where each node represents a part of the document. The DOM allows dynamic manipulation of the content on a page using methods. Methods can be called on a specific DOM node or a collection of nodes. For example, the `getElementById()` method is used to retrieve a document element with a specific `ID`. The DOM also provides properties, which are values of the node that can be read or written. For example, the `innerHTML` property is used to retrieve or set the content of an element. [13] Web browsers use engines such as Gecko, Webkit, and Blink to parse HTML into the DOM. [14]

Figure 2.1 DOM diagram of the HTML document 2.1



2.1.5 JSON

JSON (JavaScript Object Notation) is a lightweight text data format that is easy to read and write for humans and computers. JSON data consist of key-value pairs, where the key is always a string and the values are strings, numbers, booleans, arrays, or other JSON objects. Arrays are enclosed in square brackets, and JSON objects are enclosed in compound brackets. The objects are separated by commas. Most modern programming languages support JSON. [15]

2.1.6 API

API (Application Programming Interface) is a software interface that allows the communication of various applications over the network. It provides a set of rules on how data can be exchanged. The API can be used for various needs, such as the integration of third-party applications, client-server applications, or communication with IoT devices [4]. [16]

[3]World Wide Web Consortium – https://www.w3.org/
[4]Internet-connected devices that collect and transmit data.

2.1.6.1 REST

REST (REpresentational State Transfer) is an architectural style devised by Roy Fielding in his 2000 Ph.D. thesis. REST must meet the following principles: client-server architecture, stateless communication, uniform interface, layered system, and optionally code on demand. In RESTful systems, clients send HTTP requests to the server that responds with a representation of the requested resource usually in JSON or XML format. The resource is identified by a unique URI. REST decouples the client and server as much as possible and allows a high level of abstraction. REST is also flexible enough to allow easy system extensibility. The disadvantage is that there is no uniform pattern to how resources are modeled, it always depends on the specific use case. [17], [18], [19]

2.1.6.2 SOAP

SOAP (Simple Object Access Protocol) is a protocol for exchanging information using XML. The specification was publicly released by the W3C in 1999. SOAP can be used with various transport protocols such as HTTP, FTP[5], or SMTP[6]. SOAP uses WSDL to describe the functionality of a service, where WSDL is a language built on top of XML. This built-in functionality makes the SOAP platform and language independent. One of its main advantages is that very high security can be ensured thanks to the WS-Security extension[7]. The disadvantage is that SOAP fundamentally only works with XML files, which can be large in size, and therefore SOAP requires a lot of network bandwidth. [19], [20]

2.1.6.3 GraphQL

GraphQL is a query language that was released by Facebook in 2015 as open-source. GraphQL uses the Schema Definition Language to create schemas. GraphQL allows the client to get only specific data, which makes it very efficient. Moreover, the client can verify that his query matches the data schema. The disadvantages of GraphQL are that it is relatively computationally intensive and not as easy to learn as REST API. [19], [20]

2.1.6.4 gRPC

gRPC is an open-source framework developed by Google. gRPC uses the Protobuf (protocol buffers) messaging format, which provides high efficiency of structured data transfer. Protobuf describes the service interface, and gRPC then generates code for the specific programming language. It supports many languages such as Python, Java, C++, Go, or C#. gRPC requires HTTP/2 for communication between the server and the client. in addition, gRPC offers other functionalities such as streaming and bidirectional communication. gRPC is well-suitable for microservice-based applications. [19], [20], [21]

2.1.7 CRUD

CRUD (Create, Read, Update, Delete) are operations that are used for communication with a database application. It is a popular acronym as it provides an overview of possible manipulative operations. Each language or framework provides its own operations that correspond to CRUD. The table 2.1 shows the equivalent CRUD operations for the REST standard and the SQL language[8]. [22]

[5]File transfer protocol for exchanging files.
[6]Email protocol for message transmission.
[7]https://www.ibm.com/docs/en/app-connect/11.0.0?topic=security-ws
[8]Language for managing relational databases.

Table 2.1 CRUD operations

CRUD	Description	SQL command	HTTP method
Create	Add one or more new entries	Insert	POST / PUT
Read	Retrieves entries that match certain criteria	Select	GET
Update	Change specific fields in existing entries	Update	PUT / POST / PATCH
Delete	Entirely removes one or more existing entries	Delete	DELETE

2.1.8 Webhooks

Webhook refers to a callback function that uses HTTP communication between two applications. To set up the webhook, it is necessary to send the API URI to the server and specify which data should be received. The client then does not have to ask the server for new updates, as the server automatically sends the data to the client when the requested change occurs. Webhooks are known as push APIs or reverse APIs because the server is responsible for the communication, rather than the client, as with standard API. [23]

2.1.9 Web applications

Web applications are applications that can be controlled using a Web browser. They provide a convenient way for users to interact with data and services without the need to install software on the device. The majority of applications use JavaScript, HTML, and CSS technologies on the client side, and on the server side use languages such as Python, C#, and Java to run the program on the server. [24], [25]

For this work, we chose a client-side dynamic web application because it provides a smooth user experience, lower server utilization, and offers high flexibility. For completeness, the possible approaches to creating web applications are described in the order in which they have evolved.

2.1.9.1 Static web apps

Static web apps are simple applications that work on the basis of fixed and immutable content. They do not have any interactive elements. Source files are stored on the server and they are commonly written in HTML with the same content displayed to all users. Each link in the application corresponds to exactly one type of HTTP request to which the server responds with a new page. The primary role of the web browser is to act as a simple terminal for the remote display of content, while the server acts as an index and data storage system. [25]

2.1.9.2 Server-side dynamic web apps

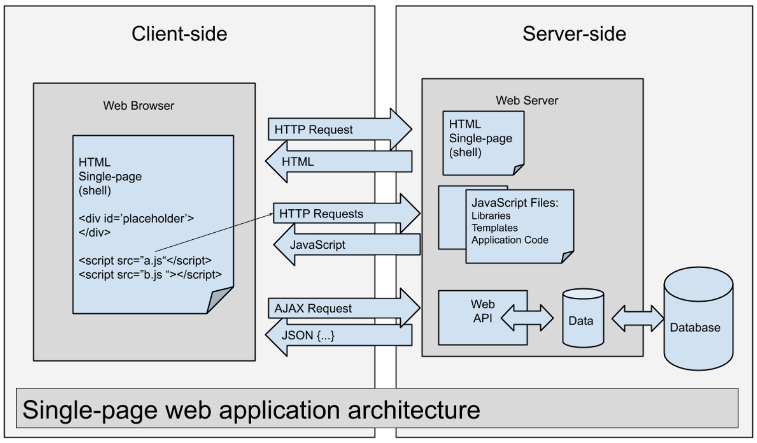
Server-side dynamic web applications are dynamic web pages that generate dynamic content on the server side based on a request from the client. The template engine on the server is used to achieve dynamic behavior. A template is stored in the source code, into which a dynamic value is placed using a script, and then a page with this value is returned by the server. Therefore, the web page is constructed using a server-side script. For the development of server-side web applications, technologies such as Django, ASP.NET, or Spring Boot, which provide templating, are used in addition to standard HTML. [25], [26]

2.1.9.3 Client-side dynamic web apps and Single-page apps

Client-side dynamic web applications are applications that are dynamically generated on the client side in the web browser. Compared to server-side dynamic applications, the changes to the page do not strictly rely on the server. They work on the principle that the web browser fetches the initial HTML page from the server and then uses a programming language to dynamically modify the DOM based on user interaction. The programing language used for client-side scripting is mostly JavaScript. This approach allows a more interactive and seamless user experience, as no server is required for each user interaction.

SPA (Single-page applications) are web applications that dynamically update the content on a single page instead of loading several pages from the server. This allows changing the content of a page without reloading it. As single-page applications depend on client-side technologies, and they are strongly linked to client-side dynamic web applications. [24], [25], [26]

Figure 2.2 SPA diagram [27]



2.1.10 Conclusion

We have described the concepts used in web application development in the order they evolved. For this work, we have chosen the Single page application architecture, which is a type of dynamic client-side web application, due to the fact that these applications provide a seamless user experience. Another reason why we chose to use a client-side dynamic web application is that the client and server components can be developed independently. These components communicate with each other over the network using the standardized HTTP protocol. By splitting the workload between server and client, client-server applications provide efficient resource sharing. In this approach, the client initiates the connection to the server, while the server remains in a state of listening to the requests of each client.

We chose REST API as the web API for communication between server and client components because of its high flexibility and simplicity. The Strava platform also offers a REST API. Webhooks are used to receive new activities from Strava so the server is not burdened with unnecessary requests for activity updates. Data between the server and client as well as between Strava and the server are exchanged using JSON data format.

2.2 Technologies

This section describes the technologies chosen for application development. As we use SPA architecture, we divide the technologies into client side, server side, data layer, and deployment sections.

2.2.1 Client side

The client side of a web application is part of the application that runs on the user's device, typically in a web browser. This part is supposed to process user interactions, send requests to the server via API and render the user interface.

2.2.1.1 JavaScript and TypeScript

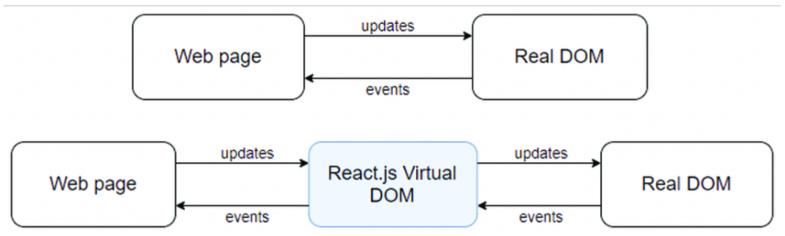
JavaScript is an open-source programming language, which is mainly used to execute complex tasks in a web browser. It allows users to create dynamic content on a web page, 3D animated graphics, and control multimedia content. JavaScript has changed the web application development industry. Nowadays, JavaScript is not only popular on the client side for creating web applications, but is also suitable for server-side use. [28]

TypeScript is an open-source programming language developed by Microsoft. It is based on JavaScript. TypeScript provides static type checking, which is a useful feature in application development. It also provides functionality for better code management, such as classes or interfaces. [29]

2.2.1.2 React

React is an open-source JavaScript library used for creating user interfaces. It was developed by Facebook. React allows the development of complex user interfaces using declarative programming. React's performance is optimized by using a virtual DOM instead of directly manipulating the real DOM. The virtual DOM acts as an intermediary between the application and the actual browser DOM. React is a suitable tool for developing single-page applications. It uses JSX, which allows writing HTML-like code inside JavaScript. JSX is not standard HTML, it is code that has a similar structure to HTML, but it is translated into JavaScript in the background. This is useful since error checking is done at the compiler level instead of in the runtime. React fully supports TypeScript, so the developers can use static type checking. [30]

Figure 2.3 Real DOM and React Virtual DOM [31]



2.2.1.3 Material UI

Material UI is an open-source React UI library that provides premade components based on Google's Material Design. Material UI offers a lot of components like buttons, formulas, and tables. Material UI gives the developer a lot of flexibility to customize the appearance and layout of the page. [32]

2.2.2 Server side

The server side of a web application is part of the application that runs on the server. This part is primarily responsible for processing incoming requests from the client.

2.2.2.1 Python

Python is an open-source interpreted programming language, which is very easy to understand and very powerful. Its big advantage is that it is easily portable on most platforms. Python supports both procedural and object-oriented programming approaches. The ecosystem of Python is wide, and it is suitable for many domains, such as web development, data analysis, or network programming. [33]

2.2.2.2 Django and Django REST framework

Django is a framework written in Python designed for creating web applications. It is primarily targeted for server-side development. Django follows the MVT architectural style. MVT (Model View Template) stands for the application which is divided into these separable parts:

Model: Represents data and business logic of the application.

View: Handles user input, retrieves data from the Model, and renders the response using the Template.

Template: Represents user interface and presentation layer.

Django includes a built-in logging system that gives users the ability to monitor the state of the application. The logger has severities of DEBUG, INFO, WARNING, ERROR, and CRITICAL. [34], [35]

DRF (Django REST framework) is a library that is built on top of the Django library. The library is used for creating the REST APIs. It handles things like serialization, object relation mapping, and authentication. It supports various data formats like JSON, XML, and HTML. [36] The Django REST framework uses the MVS (Model View Serializer) architecture, which differs from MVT in that it uses Serializers instead of Templates to convert data from the database to JSON or XML format. [37]

DRF offers ORM. ORM (Object-Relational Mapping) is a technique for mapping database tables and their relationships to a model in Django. It allows the developers to manipulate the database using Python code instead of writing SQL queries.

2.2.3 Data layer

The data layer of a web application is part of the application responsible for retrieving and storing application data. This part should provide the needed data to the server side. The data layer can be integrated within the server-side layer or separated.

2.2.3.1 DBMS

DBS (Database management system) is a computer system for organizing and managing data. Users of this system can perform various operations when working with data in the database. A database is a collection of data divided into logical units. [38] There are various types of databases, such as:

Hierarchical database: Hierarchical database is useful for managing hierarchical data, such as for managing the hierarchical structure of a company. It works on the principle of creating a tree, where a parent can have several children, and a child has just one parent.

Relational database: A relational database is a type of database that is useful for managing structured data. It works on the principle that data is stored in tables, where each table consists of rows and columns. Columns represent data attributes and rows represent individual records. The relationship between multiple tables is defined by common attributes or keys. [39]

Non-relational database: : A non-relational database is a type of database that is useful for managing unstructured data. It works on various principles such as key and value pairs, graph databases, and document-oriented databases.

In this work, we use a relational database because we work with well-structured data, and the relational database satisfies these features [39]:

Data consistency and integrity: Relational database ensures consistency and structural and logical integrity of data. Consistency means that the data is valid and meets the constraints for each change. Structural integrity ensures that the data conforms to a given schema, that each table has a primary key, that foreign keys are used to establish relationships between tables, and that there are no invalid references in the database. Logical integrity ensures that the data in the database satisfy certain constraints, such as that records have a unique identifier.

Flexibility: Flexibility indicates that the database is not implemented in a rigid way, which means it should be easy to modify or extend the database schema without disrupting the existing data.

Query performance: The query performance is high for relational queries. The SQL language is used for working with the relational database because it is very well designed for working with data.

2.2.3.2 PostgreSQL

PostgreSQL is a relational database system that is open source and offers many of the benefits of proprietary database systems. It provides high data integrity and excellent scalability. PostgreSQL fully complies with the ANSI SQL standard, which defines the syntax and semantics of SQL and specifies rules for creating, filtering, or modifying data in a relational database. [40], [41]

2.2.4 Deployment

Deployment is an essential part of the software development cycle. It includes the steps and processes necessary to make the software available to the users.

2.2.4.1 Docker

Docker is a platform for creating and managing containers. Containers are isolated environments for software applications. They simplify the development and delivery of applications. A DockerFile is a text file that contains instructions for building an image. A Docker image is created based on a DockerFile and contains all the files needed to run the application in the container. A Docker container is an instance of a Docker image. [42]

Docker Compose is a tool that allows developers to manage multi-container applications via a configuration file. Moreover, it allows us to define features like persistent volumes or service dependencies. [42]

2.2.4.2 GitHub

GitHub[9] is a web platform for developers to store and manage their Git repositories. Git is a versioning system that helps developers track changes in a project. Using Git, developers can modify code in a central repository, streamlining collaboration. GitHub offers a web interface for developers to manage Git repositories easily.

Moreover, GitHub offers GitHub Actions. GitHub Actions automates the process of building, testing, and deploying software. They provide official Docker actions that can be used for building, tagging images, or managing containers. [43]

2.2.4.3 NGINX

NGINX[10] is open-source software that operates as a web server, reverse proxy, cache, load balancer, and media streaming platform. It was designed for maximum performance and stability and can function as a reverse proxy for HTTP, TCP[11], and UDP[12] servers.

A reverse proxy is a type of server used within a private network. It sends requests from the client to a proper backend server. It provides another layer of abstraction between the server and the client. [44]

2.2.5 Conclusion

As a client-side technology, we chose TypeScript for its dynamic functionality and static type checking. We selected React as the main library because of its declarative approach, efficiency due to the virtual DOM, and TypeScript compatibility. Material UI was chosen as the UI library because it provides prebuilt components based on Google's Material Design.

On the server side, we chose Python for its ease of use and cross-platform portability. For creating a RESTful API, we selected the Django REST framework because it supports features such as data serialization and ORM. PostgreSQL was chosen as the database system because of its robustness and performance.

We chose Docker, GitHub, and NGINX technologies for application deployment because they provide several benefits and facilitate application development. Docker was chosen because it simplifies application development and deployment by containerization. GitHub was selected because it allows tracking changes during the development process, and GitHub Actions automates the process of building and deploying the application. NGINX was chosen as a reverse proxy server because it is relatively simple to configure and it performs well.

[9] https://github.com/
[10] https://www.nginx.com/
[11] Transport protocol for reliable communication on the internet.
[12] Connectionless transport protocol for communication on the internet.

Chapter 3

Implementation

This section demonstrates the details of the application implementation. As we described above, the architecture of the application is a single-page web application. 2.1.10 It uses React 2.2.1.2 with TypeScript 2.2.1.1 on the client side and Django REST framework2.2.2.2 on the server side. The application development was done according to the functional and non-functional requirements described in section 1.1.

3.1 Server side

A Django project can consist of several applications. Each application can contain views, serializers, models, and other components. In our project we have created the following applications:

`sports_equipment_tracker`: The main DRF application, the other applications are defined here.

`SportsEquipmentTracker`: The app provides management of activities, equipment, and its components.

`SportsEquipmentTrackerAuth`: The app provides user account management and security.

`SportsEquipmentTrackerStrava`: The app implements webhooks and listens for incoming events from the Strava.

As described above 2.2.3.2, we use PostgreSQL as our database system. One of the factors we chose PostgreSQL is that Django has excellent support for it. In the global settings of the project in the `settings.py` file, the configuration for PostgreSQL should be set as shown in the listing 3.1. The parameters are stored in environment variables.

The Django REST framework follows the Model-View-Controller pattern which is intended for creating REST APIs. The REST API was designed to meet the functional requirements 1.1.1. It supports CRUD operations 2.1.7 for equipment, components, activities, alerts, users, and strava accounts. Responses are in JSON format. For full API documentation, see figure B.1 in the appendix.

3.1 PostgreSQL configuration for Django

```
DATABASES = {
  "default": {
    "ENGINE": "django.db.backends.postgresql",
    "NAME": os.environ.get("SQL_DATABASE" "SportsEquipmentTracker"),
    "USER": os.environ.get("SQL_USER"),
    "PASSWORD": os.environ.get("SQL_PASSWORD"),
    "HOST": os.environ.get("SQL_HOST", "172.22.0.2"),
    "PORT": os.environ.get("SQL_PORT", "5432")
  }
}
```

3.1.1 Models

The model represents the data and defines how it is stored in the database. It provides a view of the data in an object-oriented way. In DRF, model classes inherit from the `models.Model` class provided by Django. Each attribute in the module corresponds to the field in the database table. Relationships between models are defined using foreign keys or many-to-many relationships. We have created the `user`, `activity`, `equipment`, and `component` models. The database schema can be seen in figure C.1.

3.2 Equipment model

```
class Equipment(models.Model):
  user = models.ForeignKey(User, on_delete=models.CASCADE)
  kind = models.CharField(max_length=100)
  name = models.CharField(max_length=100)
  reg_date = models.DateTimeField(auto_now_add=True)
  weight = models.FloatField(null=True)
  brand = models.CharField(max_length=100, null=True)
  model = models.CharField(max_length=100, null=True)
  year = models.IntegerField(null=True)
  notes = models.TextField(null=True)
  retired = models.BooleanField(default=False)

  source = models.CharField(max_length=100, null=True)
  strava_equipment_id = models.CharField(
    null=True,
    unique=True,
    max_length=50
  )
  components = models.ManyToManyField(Component,
    related_name='equipmentcomponent_set')
  alerts = models.ManyToManyField(Alert,
    related_name='equipmentalert_set')
```

3.1.2 Serializers

Serializers are used to convert objects into JSON format so it is possible to send objects via HTTP request. They are also used to deserialize data into objects so data can be handled by the application. In DRF, Serializer classes inherit from the `serializers.ModelSerializer` class provided by Django.

Each model has its own serializer in the application. As an illustration of its use, see the listing 3.3 showing `EquipmentSerializer`, which serializes the `Equipment` model. Besides the model's own attributes, it also adds information about the total distance and elapsed time according to the use of the specified equipment in activities.

3.3 Serializer of Equipment model

```
class EquipmentSerializer(serializers.ModelSerializer):
  activities = ActivitySerializer(many=True, read_only=True)
  reminders = ReminderSerializer(many=True, read_only=True)

  # Calculate total distance for all activities using aggregate function
  distance = serializers.SerializerMethodField()
  def get_distance(self, obj):
    return obj.activityequipment_set.aggregate(
      distance=models.Sum('distance')
    )['distance']

  # Calculate total elapsed time for all activities using aggregate function
  elapsed_time = serializers.SerializerMethodField()
  def get_elapsed_time(self, obj):
    return obj.activityequipment_set.aggregate(
      elapsed_time=models.Sum('elapsed_time')
    )['elapsed_time']

  class Meta:
    model = Equipment
    fields = ('id','kind','name','reg_date','weight','brand','model',
              'year','notes','retired','source','strava_equipment_id',
              'distance','elapsed_time','activities','components','alerts')
    depth = 2
```

3.1.3 Views

In DRF, views are used to define the logic of individual endpoints. They are responsible for processing HTTP requests and returning HTTP responses. DRF provides several classes that can be used as base classes for defining views. In this work, we use the `APIView` class, from which views are inherited. As an implementation illustration, see the listing 3.4 where the view for retrieving the user equipment is implemented.

3.4 View for retrieving user equipment

```
class EquipmentView(APIView):
  # set authentication and permission classes
  authentication_classes = [TokenAuthentication]
  permission_classes = [IsAuthenticated]

  def get(self, request, user_id=None):
    # Check if the user ID in the URL matches the authenticated user's ID
    if request.user.id != user_id:
      return Response("User mismatch", status=HTTP_400_BAD_REQUEST)

    # Get all equipments associated with the authenticated user
    equipments = Equipment.objects.filter(user=request.user)

    # Serialize the equipments data and return as a response
    equipments_serializer = EquipmentSerializer(equipments, many=True)
    return Response(equipments_serializer.data, status=HTTP_200_OK)
```

3.1.4 Authentication

Token-based authentication is appropriate for SPA because it provides secure and efficient communication between the client and server. Once user identity is verified, the client receives a token from the server that can be stored in local storage[1] or a cookie[2]. Then, each time the client requests the server, the client attaches this token to the request header. The client can use the token as long as it is valid. [45] This principle preserves the statelessness of the application because the server does not have to maintain any session.

In this application, we use access tokens with infinite expiration for communication between the client and server. We decided to do so because of the simplicity of use and user-friendliness. The disadvantage is a slightly reduced security of the application compared to the approach where access tokens have limited validity. If the user logs out from the application, the token is destroyed, and a new token is generated when the user logs in again.

DRF offers the possibility of token authentication. The listing 3.5 shows a fragment of `UserManager` class for the `User` model, which inherits from Django built-in `UserManager` class. The `create_user` method is used to create a new user based on username, password, and other additional parameters. The password is encrypted using the `set_password` method, and the created user is saved to the database using `user.save()`. We can also see `User` class, which is the model for the user itself. By setting `objects` to `UserManager`, we ensure that all operations with the user will be performed using `UserManager`.

[1] Client-side storage of key-value data.
[2] The small text file stored in a browser

3.5 UserManager and User model

```
class UserManager(models.Manager):
  def create_user(self, username, password=None, **extra_fields):
    if not username:
      raise ValueError('Users must have an username')
    user = self.model(username=username, **extra_fields)
    user.set_password(password)
    user.save(using=self._db)
    return user
  ...
class User(AbstractUser):
  objects = UserManager()
```

To let DRF know to use our `User` model, we need to set the `AUTH_USER_MODEL` variable to our `User` model in the global application settings, as shown in the listing 3.6.

3.6 Sets custom user model for authentication

```
...
AUTH_USER_MODEL = 'SportsEquipmentTrackerAuth.User'
...
```

3.1.5 Strava

The integration of Strava into our application can be divided into two parts. The first part is about managing the Strava account. The second part is about managing activities.

3.1.5.1 Strava account

Our REST API provides endpoints with the prefix `/api/user/<userId>/strava/account/` for working with the Strava account, as shown in the appendix in the figure B.1.

When a user logs in to Strava, the client sends a post request containing a refresh token, access token, access token expiration, and Strava account identifier to the server using our REST API. Implementation of View for creating a new Strava account can be seen in the listing. 3.7

■ **3.7** View for managing users' Strava account

```
class StravaAccountView(APIView):
  authentication_classes = [TokenAuthentication]
  permission_classes = [IsAuthenticated]
  ...
  def post(self, request, user_id=None):
    user = request.user
    if user.id != user_id:
      return Response("User id does not match", status=HTTP_403_FORBIDDEN)

    access_token = request.data.get('accessToken')
    refresh_token = request.data.get('refreshToken')
    expires_at = request.data.get('expiresAt')
    athlete_id = request.data.get('athleteId')
    # check if the user already has Strava account
    try:
      strava_account = StravaAccount.objects.get(user=user)
      if strava_account:
        return Response("Strava account already exists",
                        status=HTTP_403_FORBIDDEN)
    except StravaAccount.DoesNotExist:
      pass

    strava_account = StravaAccount.objects.create(user=user,
          refresh_token=refresh_token, athlete_id=athlete_id)
    # save access token in cache
    set_access_token(strava_account, access_token, expires_at)
    return Response({"message": "strava account created"}, status=HTTP_200_OK)
  ...
```

After creating the Strava account, the access token and its expiration are cached using the `set_access_token` method. As a cache, we use `LocMemCache` provided by Django. The data are stored in local memory, so it does not require external dependencies like a database server. [46] When communicating with Strava, such as when getting activity details, a valid access token is obtained using the `get_access_token` method, which checks the validity of the access token. If expired, it establishes a connection to Strava and using the refresh token requests a new access token, which is then cached and returned. The implementation of the `get_access_token` and `set_access_token` methods can be seen in the appendix in the listing C.2.

If the user decides to remove their Strava account, the client sends a DELETE request to the server using the REST API B.1. The server then processes the request and sends a request to deauthorize the account to the Strava URL: `https://www.strava.com/oauth/deauthorize`.

3.1.5.2 Activities

We use webhooks to receive new activities from Strava. Our REST API provides the endpoint `/api/user/<userId>/strava/webhook/`. To register a webhook subscription, we need to send a POST request to Strava with the address of our endpoint, to which Strava will send a registration GET request. [47] This request can be made using cURL[3], as shown in the listing 3.8. Only one subscription is allowed per application. To sign up for a new subscription, we must first delete the previous subscription. You can accomplish this by using a similar command, but instead of using the POST method, use the DELETE method and include the ID of the subscription that we want to delete.

3.8 cURL command to register a Strava webhook subscription

```
curl --location 'https://www.strava.com/api/v3/push_subscriptions' \
  --form 'client_id="<CLIENT_ID>"' \
  --form 'client_secret="<SECRET>"' \
  --form 'callback_url="<APP_URL>/api/strava/webhook/"' \
  --form 'verify_token="<TOKEN>"'
```

For working with webhooks we implemented the `StravaWebhookView` class, which inherits from `APIView`, and implements the `get` and `post` methods. The entire implementation of the class can be seen in the appendix in the listing C.1.

The `get` method is used to register for event subscriptions from Strava. It checks if the mode is set to `subscribe` and if the token matches the token we sent with the POST request to Strava. If the authentication is successful, it sends a response with the challenge parameter included in the GET request.

The `post` method is used to handle the events sent by Strava. In our application, we work with activity-type objects. Activities are created, updated, and deleted in synchronization with Strava. This means that if, for example, a user changes the name of their activity on the Strava platform, the change is also propagated to our application.

In the listing C.3 in the appendix, we can see the `get_activity_details` function, which retrieves detailed information about the activity from Strava based on the activity id received from the webhook.

[3]Command-line tool for transferring data.

3.2 Client side

As mentioned in the previous chapter 2.2.1.2, React is a JavaScript library with TypeScript support. It is designed for creating client-side web applications. This section describes the features used in implementing the client-side part of the application.

3.2.1 Component-based architecture

React uses component-based architecture. That means that the developer can split the components into smaller reusable components. Components can be class-based or function-based. Our application uses function-based components because they are simpler and more elegant. A functional component is a function that returns a value of React element or JSX. The container is a component that is responsible for manipulating states and making API requests. [48], [49]

The listing 3.9 shows the functional component `GenericTable`, which we use on several application pages. The component takes an object of properties named `title`, `headers`, and `data`. It returns a formatted table. The `GenericTable` component consists of other components like `Root`, `Grid`, and `Table`, which are Material UI library 2.2.1.3 components with custom styles added.

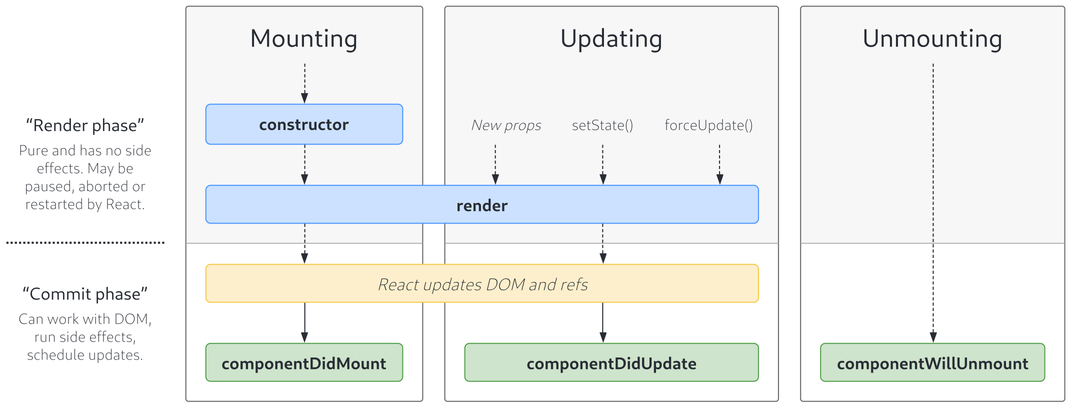
3.9 Functional component that represents a table

```
const GenericTable = ({title, headers, data}: GenericTableProps) => (
  <Root>
    <Title variant="h4">{title}</Title>
    <Grid container spacing={2}>
      <Grid item xs={12}>
        <StyledTableContainer>
          <Table aria-label={`${title} table`}>
            <TableHead>
              <TableRow>
                {headers.map((header, index) => (
                  <StyledTableCell>{header}</StyledTableCell>))}
              </TableRow>
            </TableHead>
            <TableBody>
              {data.map((row, index) => (
                <TableRow>
                  <>
                    {row.map((detail, index) => (
                    <StyledTableCell>{detail}</StyledTableCell>
                    ))}
                  </>
                </TableRow>))}
            </TableBody>
          </Table>
        </StyledTableContainer>
      </Grid>
    </Grid>
  </Root>
);
```

3.2.1.1 Lifecycle of React component

The lifecycle of a React component consists of three parts: mounting, updating, and unmounting. The mounting occurs only once when the component is created and inserted into the DOM. The updating occurs when the props or state is changed, and the component is re-rendered. The unmounting occurs when the component is no longer needed and is removed from the DOM. [50], [51]

Figure 3.1 React class-based component's lifecycle diagram [50]



Class-based components have different methods for each stage of the component's lifecycle. We use functional components that use React hooks instead of these methods. Listing 3.10 shows the `useEffect` hook we use in the `App` container to get information about the user's Strava account, activities, equipment, and components after the user logs in. The `useEffect` hook takes two arguments: a function and an array of dependencies, where the dependency is the `loggedIn` state. When `loggedIn` changes, the function from the first parameter is called. This hook is equivalent to the combination of the `componentDidMount` and `componentDidUpdate` methods in class-based components. [50], [51]

3.10 Retrieves user data if the user is logged-in

```
React.useEffect(() => {
  if (user && loggedIn) {
    const { token, id: userId } = user;

    retrieveStravaAccount(token, userId);
    retrieveActivities(token, userId);
    retrieveEquipment(token, userId);
    retrieveComponents(token, userId);
  }
}, [loggedIn]);
```

3.2.2 Routing

Routing is one of the fundamental principles used in web applications. Our application is SPA type, which means we have one HTML page on which logical pages and content are dynamically rendered without the need to refresh the page. Routing helps us to navigate between these logical pages. It also offers the ability to capture the application's state, save it to a bookmark, or share it with others. [52], [53] In our application the routing is connected to a side drawer, as seen in the listing 3.11.

To enable routing, we use the React Router library. This library provides all the functionality needed to implement routing. For navigation between pages we use `useNavigate` hook [54], and for implementation of routing we use the following components:

BrowserRouter: The `BrowserRouter` component is used to activate routing in our application. It wraps up the entire application. [55]

Routes: The `Routes` takes Route components as children. Route elements can be nested. [56]

Route: The `Route` component is used to define a specific route. It takes `path` and `element` as props. The path is a URL that should match this route. The element takes a JSX element that should be rendered. [57]

3.11 Definition of routing nested in sidedrawer

```
<SideDrawer body={drawerBody} items={drawerItems} >
  <Routes>
    <Route path="/" element={
      <RequireAuth usedLocalStorage={usedLocalStorage}>
        <Home usedLocalStorage={usedLocalStorage} />
      </RequireAuth>
    } />
    <Route path="/auth" element={<Auth />} />
    <Route path="/strava-redirect/:code" element={
      <RequireAuth usedLocalStorage={usedLocalStorage}>
        <StravaRedirect />
      </RequireAuth>
    } />
    <Route path={"/equipment"} element={
      <RequireAuth usedLocalStorage={usedLocalStorage}>
        <Equipment />
      </RequireAuth>
    } />
    <Route path={"/activities"} element={
      <RequireAuth usedLocalStorage={usedLocalStorage}>
        <Activities />
      </RequireAuth>
    } />
    ...
  </Routes>
</SideDrawer>`
```

3.2.3 State management

As mentioned above 2.2.1.2, in React, the user interface is defined in a declarative way. This means we describe how a component should look depending on its state. React handles the re-rendering of the component when the state changes. As the complexity of the application grows, it becomes more challenging to manage the state of the application by manually passing the state between individual components.

In our application, we have four separate contexts. The first two are for managing the user account and the user's Strava account. The second two are for managing activities, equipment, and components. This section demonstrates how to manage the activities. [58]

3.2.3.1 Context creation

First, we have to create a new context object. The context maintains the data. Since we have a separate context for reading and writing state, we have to create two context objects. In the listing 3.12, we can see the type definitions and the creation of two context objects `sportsContext` and `sportsDispatchContext`.

Also, we can see the functions `useSports` and `useSportsDispatch`, which are hooks that allow components to consume `sportsContext` and `sportsDispatchContext`. These hooks hide the implementation details of managing the context, and they provide a clean interface for consuming or updating the activities.

```
type SportsContextType = {
  activities: Activity[];
  selectedActivity: Activity | null;
  ...
};
type SportsDispatchContextType = {
  dispatchActivities: React.Dispatch<ActivityAction>;
  setSelectedActivity: React.Dispatch<React.SetStateAction<Activity | null>>;
  ...
};

const sportsContext = createContext<SportsContextType>({
  activities: [],
  selectedActivity: null,
  ...
});
const sportsDispatchContext= createContext<SportsDispatchContextType | null>({
  dispatchActivities: () => {},
  setSelectedActivity: () => {},
  ...
});

export const useSports = () => useContext(sportsContext);
export const useSportsDispatch = () => useContext(sportsDispatchContext);
```

■ **3.12** Creation of context objects

3.2.3.2 Action and reducer creation

Then we have to create an action and reducer, as seen in the listing 3.13. An action is an object describing a specific change in the application state. In this case, `ActivityAction` is a type defining three actions: ADD, DELETE, and CLEAR. The ADD action is used to add a new activity. DELETE action is used to delete a specific activity. CLEAR action is used to delete all activities. As a reducer the `activityReducer` function is used, which handles these actions.

3.13 Creation of action and reducer

```
export type ActivityAction =
  | { type: 'ADD', id: number, kind: ActivityKinds, distance: number | null,
      moving_time: number, elapsed_time: number, name: string | null,
      source: ActivitySources, date: string | null }
  | { type: 'DELETE', id: number }
  | { type: 'CLEAR' }
const activityReducer: Reducer<Activity[], ActivityAction> =
  switch (action.type) {                 (state, action): Activity[] => {
    case 'ADD': return [...state, { ...action }]
    case 'DELETE': return state.filter((activity) => activity.id !== action.id)
    case 'CLEAR': return []
    default: return state
}}
```

3.2.3.3 Provider creation

The provider is a React component that allows other components to access the context and its changes. The provider accepts value and children as props. Value is data that are available to other components. Children is a tree structure of components that have access to the context.

The listing D.1 in the appendix shows the `SportsProvider` component that creates and manages the application's global state using the Context API. Specifically, it shows the state for all activities and the selected activity. The component uses the `useReducer` hook to create reducer functions and the `useState` hook to create state variables. Then the component wraps the child components with providers. The first provider `sportsContext.Provider` passes the application state as an object to the other components. The provider `sportsDispatchContext.Provider` passes dispatch methods.

The listing 3.14 shows a section of the `index.tsx` file, which renders the main component `App` of the application. It wraps it in `UserProvider`, `SportsProvider`, and `BrowserProvider`. This implies that all components used in the application have access to the app state.

3.14 UserProvider and SportsProvider usage

```
root.render(
  <UserProvider>
    <SportsProvider>
      <BrowserRouter><App /></BrowserRouter>
    </SportsProvider>
</UserProvider>);
```

3.2.3.4 Context consumption

In the listing 3.15 we can see the section on the implementation of the `Activities` container, which is a logical page for displaying and manipulating activities. This component uses the `useSports` and `useSportsDispatch` hooks to retrieve activities. The `dispatchActivities` function is used to update the application state with activities.

When the user chooses to add an activity, the `onAddActivity` function is called, which uses the `postActivity` function to communicate with the server side of the application using the REST API B.1. If the activity was saved successfully, this function dispatches the `ADD` action with the data received from the server. This will change the global state of the application, and all components dependent on the activities will be re-rendered.

3.15 Activities page

```
export const Activities = () => {
  const activities = useSports()?.activities
  const dispatchActivities = useSportsDispatch()?.dispatchActivities
  const selectedActivity = useSports()?.selectedActivity
  const setSelectedActivity = useSportsDispatch()?.setSelectedActivity
  ...
  const onAddActivity = () => {
    postActivity(activityFormData, token, userId)
      .then((createdActivity: ActivityRequestData) => {
        ...
        dispatchActivities({
          type: 'ADD',
          ...createdActivity,
          date: formatDate(new Date(createdActivity.date || ""))
        } as ActivityAction);})
      .catch((error) => {...})}
  ...
  return (<>...</>);
};
```

3.2.4 Authentication

As we described in the server-side implementation section 3.1.4, the application's authentication is token-based. Our REST API B.1 provides endpoints for user signup, login, and logout. The listing 3.16 shows the `loginUser` function, which is used for user login. This function takes the user id and password and calls the `makeHTTPRequest` function, which encapsulates communication with the API. The `makeHTTPRequest` function can be seen in the listing D.2 in the appendix.

3.16 User login function

```
export const loginUser = async (username: string, password: string):
                                        Promise<AuthResponseData> => {
  return makeHTTPRequest<AuthResponseData>(
    '',
    'POST',
    `api/auth/login/`,
    JSON.stringify({username: username, password: password})
  )
}
```

The `loginUser` and `signupUser` functions return a promise of type `AuthResponseData`, which contains the id and access token, as seen in the listing 3.17. The id and token are then stored in web storage. Web storage API allows storing the data in key-value format in the browser. Web storage offers two types of storage: `sessionStorage` and `localStorage`. We use `localStorage` as it persists data after the browser is closed and reopened. [59] We can see the implementation of the functions for working with local storage in the listing in the appendix D.3.

3.17 Type defining the user id and its access token

```
export type AuthResponseData = {
    id: number;
    token: string;
}
```

When the page is refreshed, the application tries to find the user id and token in the local storage so the user does not have to log in again. This process is implemented using the `useEffect` hook, as seen in the listing 3.18.

3.18 Hook to get user id and token from local storage

```
useEffect(() => {
  const authValue: AuthData | null = getLocalStorage<AuthData>('auth');
  if (authValue) {updateAuth(authValue)}
  setUsedLocalStorage(true)
}, [])
```

The application automatically redirects to the login page if the user id and token are not stored in the application context or local storage. This is achieved by our RequireAuth component, which ensures that the user can only access the components when logged in.

The implementation of the `RequireAuth` component is shown in the appendix in the listing D.4. Its use can be seen in the listing defining the application router 3.11.

3.2.5 Strava

This section describes how to link a Strava account to the user account. As described in the server-side section of the Strava implementation 3.1.5.1, the client gets a refresh token, access token, and access token expiration and sends them to the REST API B.1.

Firstly, the client must obtain an authorization code, which is then used to request access and refresh tokens from Strava. The obtained tokens, along with the access token expiration, are sent via the REST API to the server.

When the user decides to connect their account to a Strava account, the application redirects them to the Strava OAuth2 page. For the redirection, we use the `onHandleStravaLogin` method, which sets the `window.location` to a URL containing configuration parameters. The parameters include:

CLIENT_ID: CLIENT_ID is the application id generated within the Strava management console A.2.

REDIRECT_URL: REDIRECT_URL is the URL to which Strava redirects the user with the authorization token after they log in. In the listing 3.11, we can see `/strava-redirect/:code` route, which corresponds to the `REDIRECT_URL`. This route redirects to the `StravaRedirect` component.

SCOPE: SCOPE is the type of authorization the client requests from the user. In our case, the client requests permission to read the profile and activities.

3.19 Redirects to Strava authentication page

```
const onHandleStravaLogin = () => {
    window.location = `http://www.strava.com/oauth/authorize?
        client_id=${CLIENT_ID}&
        response_type=code&
        redirect_uri=${REDIRECT_URL}/exchange_token&
        approval_prompt=force&
        scope=${SCOPE}`;
};
```

After the user approves the login on the OAuth2 page, they are redirected with the authorization code to the `StravaRedirect` component. StravaRedirect component loads the user id and token from local storage, it parses the authorization code, and sends a POST request to Strava to get the Strava id, access, and refresh token.

Once it receives these data, it sends them to the server via the REST API and updates the Strava id in the application's user context. The partial implementation of the `StravaRedirect` component can be seen in the listing D.5 in the appendix. A user can remove their user account if they do no longer want to receive data from Strava.

Chapter 4

Testing

Testing is an essential part of software development. It ensures that the application meets functional and non-functional requirements and is reliable and secure.

4.1 Requirements

To verify that the functional and non-functional requirements defined in the analysis and design chapter 1.1 are met, we can summarize the requirements as follows.

4.1.1 Functional requirements

Functional requirements include the ability for users to register and log in, link their Strava account, manually create and delete activities, create and delete sports equipment, assign components to equipment, track wear and tear of equipment, and assign equipment to activities.

For each functional requirement, a specific page was created on the client-side along with proper endpoints on the REST API on the server-side B.1. These pages include authentication, activity management, equipment management, component management, alerts management, equipment-component association, and activity-equipment association, equipment-alert association, component-alert association. The sample pages can be seen in appendix F.

4.1.2 Non-functional requirements

Non-functional requirements include the use of a token-based secure authentication method to protect user data, the encryption of all communication between the client and server to ensure confidentiality, the availability of the application in the English language to cater to a broader audience, the accessibility of the application via a public domain name, and ensuring the usability of the application on different devices.

The application uses a token-based authentication method as described in the server-side implementation chapter 3.1.4. Both the server-side and client-side use HTTPS for secure communication 5.3, and the application is publicly available on the internet. The client interface is in English. The application is designed to be responsive, so it can be used on desktops, tablets, and smartphones with ease, as shown in appendix F.

4.2 Server-side unit testing

To test a View of the Django application, the tool `pytest-django`[1] was used. The pytest-django is a plugin for `pytest`[2], a tool suitable for testing Python applications.

As an illustration of testing using `pytest`, we can see the test of `ActivityView` in the listing 4.1. The test verifies the integration of the `User`, `Equipment`, `Component`, and `Activity` models.

```
@pytest.fixture
def user():
    return User.objects.create_user(username='testuser', password='testpass')

@pytest.fixture
def equipment(user):
    return Equipment.objects.create(user=user, kind='bike', name='My Bike')

@pytest.fixture
def component(user):
    return Component.objects.create(user=user, kind='frame', name='My Frame')

@pytest.fixture
def activity(user, equipment, component):
    activity = Activity.objects.create(user=user, name='My Ride', kind='ride',
                                       distance=5000, moving_time=30, elapsed_time=40)
    activity.equipment.add(equipment)
    activity.components.add(component)
    return activity

@pytest.mark.django_db
def test_activity_equipment(activity, equipment):
    assert activity.equipment.count() == 1
    assert activity.equipment.first() == equipment

@pytest.mark.django_db
def test_activity_components(activity, component):
    assert activity.components.count() == 1
    assert activity.components.first() == component
...
```

4.1 Activity view test

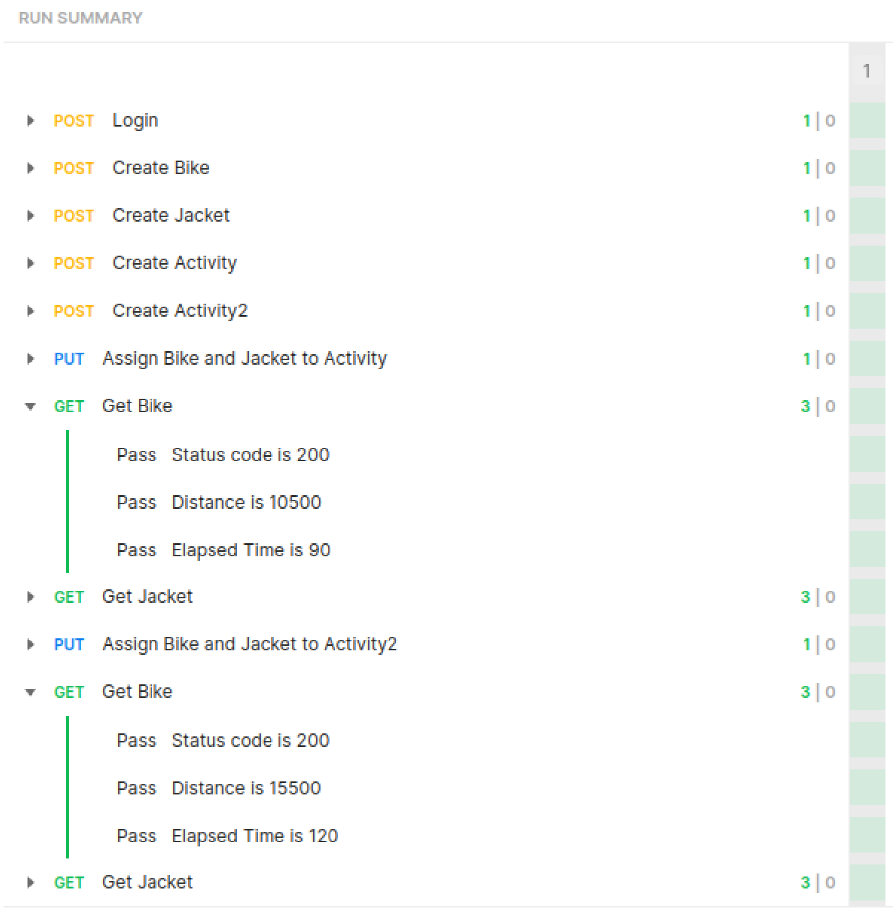
[1] https://pytest-django.readthedocs.io/en/latest/
[2] https://docs.pytest.org/

4.3 REST API testing

In this section, the functionality of the REST API is tested using tool `Postman`[3], which is a tool useful for testing REST APIs. This testing aims to verify the correct functionality of the REST API. Postman makes it easy to create and execute automated tests that can reveal potential bugs in the API. During testing, we simulated different user scenarios and analyzed the APIs' responses to these scenarios.

The figure 4.1 shows a collection of happy path[4] tests performed by the Postman tool. First, the log-in is tested using the user created for testing purposes. The return code is verified, and the token and user ID are stored in a global variable. Then a bike, a jacket, and two activities are created, and their IDS are stored in global variables. Each activity has a different distance and elapsed time. Then the bike and jacket from the previous step are assigned to the first activity. Then the correct distance and time of using the jacket and bike are verified. The bike and jacket are assigned to the second activity, and it is verified that the time and distance of the bike and jacket usage have increased correctly.

Figure 4.1 Happy path test of REST API in Postman



```
RUN SUMMARY
                                                          1
▸ POST  Login                                      1 | 0
▸ POST  Create Bike                                1 | 0
▸ POST  Create Jacket                              1 | 0
▸ POST  Create Activity                            1 | 0
▸ POST  Create Activity2                           1 | 0
▸ PUT   Assign Bike and Jacket to Activity         1 | 0
▾ GET   Get Bike                                   3 | 0
          Pass  Status code is 200
          Pass  Distance is 10500
          Pass  Elapsed Time is 90
▸ GET   Get Jacket                                 3 | 0
▸ PUT   Assign Bike and Jacket to Activity2        1 | 0
▾ GET   Get Bike                                   3 | 0
          Pass  Status code is 200
          Pass  Distance is 15500
          Pass  Elapsed Time is 120
▸ GET   Get Jacket                                 3 | 0
```

[3] https://www.postman.com/
[4] A test scenario that checks if a system works correctly under ideal conditions.

4.4 User testing

In the first part of the user testing, the work was tested by the author of this work. The author wanted to test the functionality of the application and see if it meets the functional requirements. He focused mainly on the connections with the Strava platform and the correct synchronization of activities from the Strava platform.

The author found that the application works correctly, and the activities are synchronized immediately after uploading to the Strava platform. He came up with a new functional requirement that it would be useful to be able to synchronize activities that have already been completed and to be able to assign equipment to them.

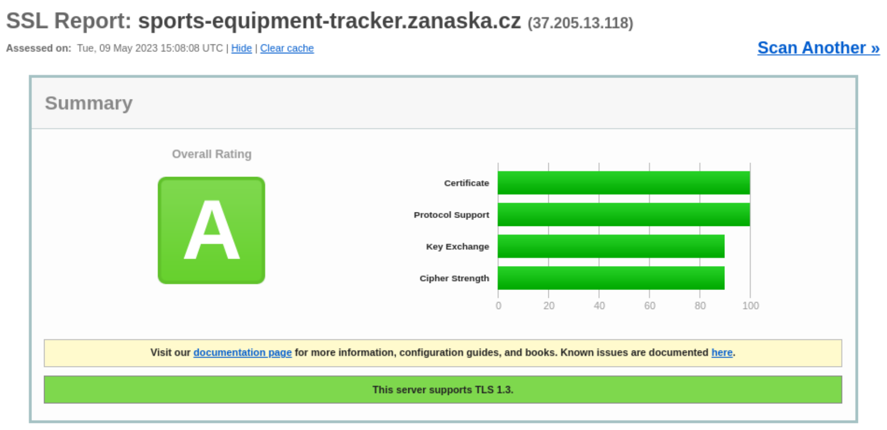
After the initial testing by the author, the application was tested by two other users. Both users had a Strava account. They were asked to use the application and then provide feedback based on their experience.

One of the users found a bug where the wear of a bike component was not evaluated correctly. This bug was fixed afterward. Both users would also like it if the application supported the display of a graph showing the use of sports equipment and components over time.

4.5 Security testing

The SSL Labs tool[5] was used to verify the security of the application. SSL Labs allows testing of the security of SSL/TLS[6] implementation. The tool performs an advanced check of SSL/TLS configuration, protocols, ciphers, and algorithms for key exchange. It provides a detailed report with a grade from A+ to F. Our application received a grade of A, as seen in the figure 4.2, which is a very good result. Security testing is an important step in order to ensure that sensitive data, such as login credentials, are sufficiently secured.

Figure 4.2 SSL test result



SSL Report: **sports-equipment-tracker.zanaska.cz** (37.205.13.118)

Assessed on: Tue, 09 May 2023 15:08:08 UTC | Hide | Clear cache

Summary

Overall Rating

A

Visit our documentation page for more information, configuration guides, and books. Known issues are documented here.

This server supports TLS 1.3.

[5]https://www.ssllabs.com/ssltest/
[6]The protocols used for secure communication over the internet.

Chapter 5

Deployment

This section describes how the application is deployed. The application is composed of Django REST framework, PostgreSQL, and React applications. Containerization is used, which means each application is containerized. These containers are managed on the server using an orchestration tool.

The deployment workflow can be described in the following steps:

1. The developer commits the code changes and pushes them to the application's GitHub repository 2.2.4.2. Afterward, they merge the commits into the main branch.
2. The GitHub Action 2.2.4.2 is configured to be triggered when the code is delivered to the main branch. The action builds Docker images 2.2.4.1 for the client and server parts of the application. It then pushes these images to the server.
3. Docker Compose 2.2.4.1 is used on the server to orchestrate containers created from the images.
4. NGINX 2.2.4.3 is used on the server as a reverse proxy to route requests to the proper container based on the URL. This allows accessing both the client and server sides on the same domain and IP address.
5. Communication between the user's browser and the server is secured using a certificate obtained from Let's Encrypt[1], enabling the HTTPS protocol.

For deployment of the application on the server, we use a virtual private server provided by vpsFree.cz[2]. As an operating system, we use Ubuntu 22.04[3], an open-source operating system based on the Linux kernel with high stability and long-term support.

The client part of the app is available at `https://sports-equipment-tracker.zanaska.cz`. The API is available at `https://sports-equipment-tracker.zanaska.cz/api/`.

5.1 Docker image

As we mentioned above 2.2.4.1, DockerFile is a file describing instructions for building a Docker image. In the listing 5.1, we can see the DockerFile for our Django application. At the beginning of the file, we define a base Docker image for the Python application. Afterward, we copy

[1] https://letsencrypt.org/
[2] https://vpsfree.cz/
[3] https://releases.ubuntu.com/jammy/

the `requirements.txt` file from our system to the image containing all the requirements needed to run the application, and then install them. We expose port 8000, on which the application listens. Finally, we run the application using Gunicorn. Gunicorn[4] is an HTTP server designed for Python WSGI applications. WSGI[5] is a standard that specifies communication mechanisms between web servers and web applications.

The DockerFile for the React application can be seen in the appendix in the listing E.1. For the PostgreSQL image, we use the official image as seen in the Docker Compose file in the listing E.2.

5.1 Dockerfie for Django app

```
FROM python:3.9-alpine

COPY backend/sports_equipment_tracker/requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY backend/sports_equipment_tracker/ ./
EXPOSE 8000

CMD ["gunicorn", "sports_equipment_tracker.wsgi", "--bind", "0.0.0.0:8000",
     "--workers", "4"]
```

5.2 Docker Compose

As mentioned above 2.2.4.1, Docker Compose is a tool that allows developers to manage multi-container applications via a configuration file. Moreover, it allows to define features like persistent volumes or service dependencies.

In the appendix in the listing E.2, we can see the Docker Compose file for our application. It defines three services:

db: The `db` service uses the official Postgres Docker image from the Docker Hub[6]. We configure the volume[7] for persistent data so that the data stored in the database remains after the container is restarted. Afterward, we set the environment variables for the database and the network to `backend_network`.

django_server: The `django_server` is a service that uses a custom image `tracker_django` created within a GitHub Action 5.1. The service exposes the application on port 200 and it depends on the `db` service with which it is connected in the same `backend_network` network. It sets the environment variables for the hostname and database port required by the Django application. 3.1.

react_client: The `react_client` service uses a custom image `tracker_react` E.1. The service exposes the application on port 201. It does not use the shared network with `db` and `django_server` services. The communication between `react_client` and `django_server` is via a reverse proxy.

[4]https://docs.gunicorn.org/
[5]Web Server Gateway – https://wsgi.readthedocs.io/
[6]https://hub.docker.com/
[7]Persistent data storage for Docker containers.

The registration of a Strava subscription is not automated. After the `django_server` service is started, sending a POST request for the provision of a subscription is necessary, as described in the server-side implementation section 3.1.5.2.

5.3 NGINX

As mentioned above 2.2.4.3, NGINX is software that operates as a web server. We use NGINX on the server to receive client requests and proxies them. In the listing 5.2, we can see the server's configuration, which shows how to use NGINX as a reverse proxy.

The first section of the file defines two upstreams, `backend` and `frontend`, which correspond to our services from the `docker-compose` file E.2. The `backend` block proxies requests to the `django_server` service running on port 200. The `fronted` block proxies requests to the `react_client` service running on port 201. The second section of the file defines a block that listens on port 443, the default port for HTTPS. In this section, we specify the SSL certificate created using the Certbot utility[8], as seen in the figure E.1 in the appendix. We specify two locations. The `/api/` location proxies requests to the `backend` upstream. The `/location` proxies requests to the `frontend` upstream.

5.2 Configuration of NGINX as a reverse proxy

```
upstream backend {
        server localhost:200;
}
upstream frontend {
        server localhost:201;
}
server {
    server_name sports-equipment-tracker.zanaska.cz;
    location /api/ {
        proxy_pass http://backend/;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }
    location / {
        proxy_pass http://frontend/;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }
    listen [::]:443 ssl ipv6only=on; # managed by Certbot
    listen 443 ssl; # managed by Certbot
    ssl_certificate /etc/letsencrypt/sports-equipment-tracker.zanaska.cz/
                    fullchain.pem; # managed by Certbot
    ssl_certificate_key /etc/letsencrypt/sports-equipment-tracker.zanaska.cz/
                    privkey.pem; # managed by Certbot
    include /etc/letsencrypt/options-ssl-nginx.conf; # managed by Certbot
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem; # managed by Certbot
}
...
```

[8]https://certbot.eff.org/

5.4 GitHub Actions

The listing in the appendix2.2.4.2 shows a GitHub Actions workflow file that describes the automation of building a Docker image for a Django application, saving the image as a tar file, copying it to the VPS, and loading the image on the VPS. This workflow is triggered when changes are pushed to the main repository branch. The workflow contains a `build_and_push_django` job that runs on the Ubuntu operating system. The job for building a React application is very similar, it differs in the DockerFile that is used. In the figure E.2 in the appendix we can see a screenshot from GitHub showing the individual steps of the `build_and_push_django` job.

Chapter 6

Conclusion

This thesis aimed to design and implement an application that allows users to monitor the wear and tear of sports equipment and its components. It was necessary to analyze existing solutions, the possibility of integration with the Strava platform, and choose appropriate technologies for developing the web application. Subsequently, it was crucial to use these technologies in practice to create a functional prototype, which had to be tested and deployed.

While implementing the application, we encountered a few challenges that required careful consideration. One of these challenges was correctly integrating the Strava platform using webhooks.

In conclusion, we have successfully analyzed, designed, developed, tested, and deployed a web application allowing users to monitor their sports equipment's wear and tear. We believe that this application will be useful for many people who are involved in sports and want to take proper care of their equipment.

6.1 Possible future extensions

It is important to note that the web application presented in this work is a working prototype that serves as a proof of concept for tracking wear and tear on sports equipment. Although the application is fully functional and provides valuable features to users, there is still room for improvement. The app was designed and developed on a limited scale as a prototype, focusing on integrating the Strava platform and basic equipment tracking features.

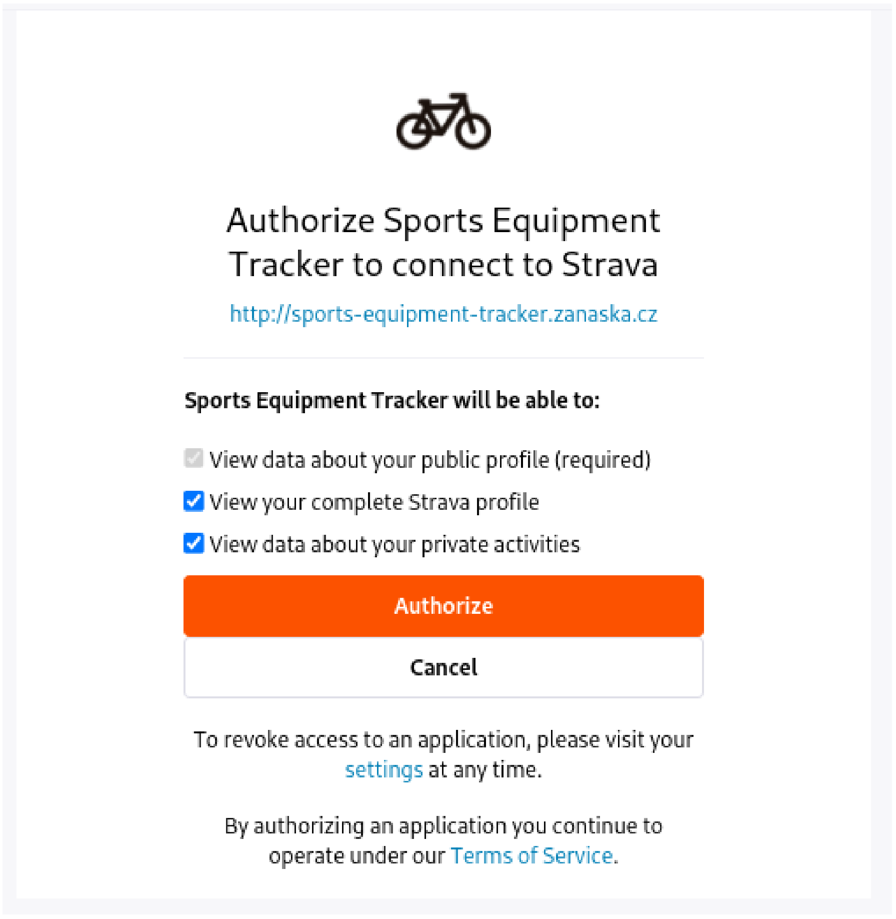
In addition to the basic features of the app, a few features came out of the testing of the app, that were good to have and could improve the user experience and overall functionality. These features include:

- Allow synchronization of activities that have already happened in history.
- Enhance the client side of the application and add graphs showing wear and tear over time.
- Add support for sending alert notifications via push notifications or email.
- Adding support for testing in GitHub actions would be helpful to ensure that all changes are tested before deploying to production. Also, automation of Strava webhook subscription registration when launching the app would be useful.
- Add support for other sports tracking services such as Garmin[1].

[1] https://connect.garmin.com/

Appendix A
Strava API

Figure A.1 Screenshot of Strava authentication page



Authorize Sports Equipment Tracker to connect to Strava

http://sports-equipment-tracker.zanaska.cz

Sports Equipment Tracker will be able to:

✓ View data about your public profile (required)

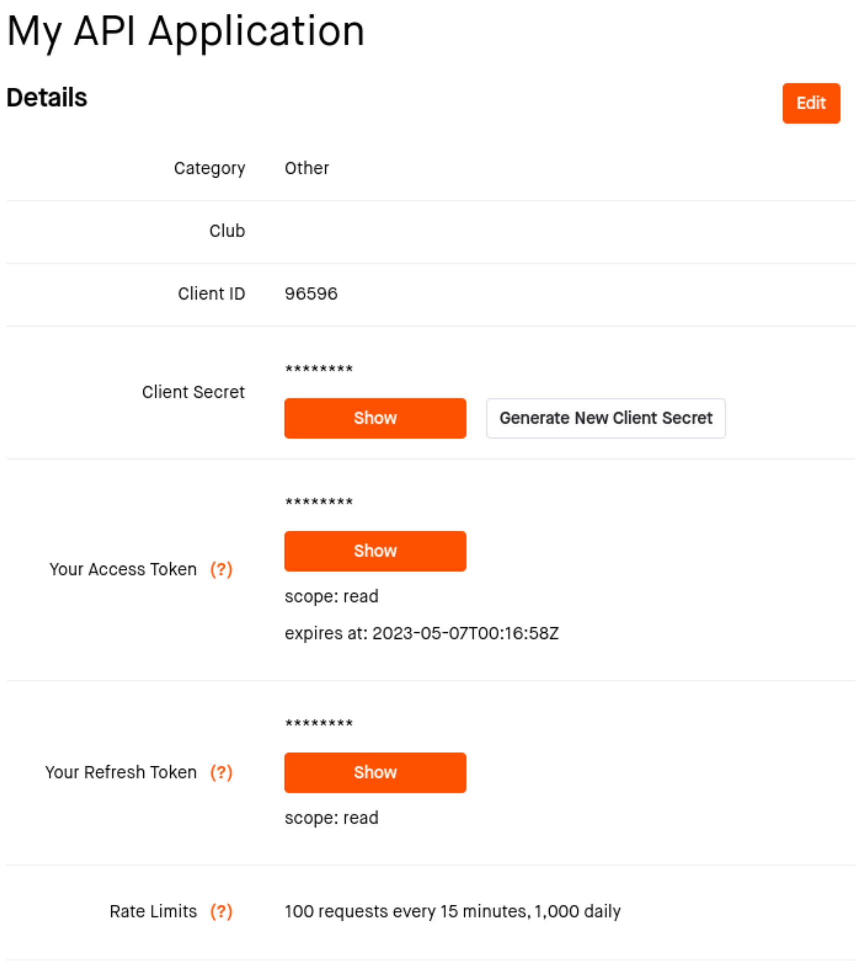
✓ View your complete Strava profile

✓ View data about your private activities

To revoke access to an application, please visit your settings at any time.

By authorizing an application you continue to operate under our Terms of Service.

Figure A.2 Screenshot of Strava management console

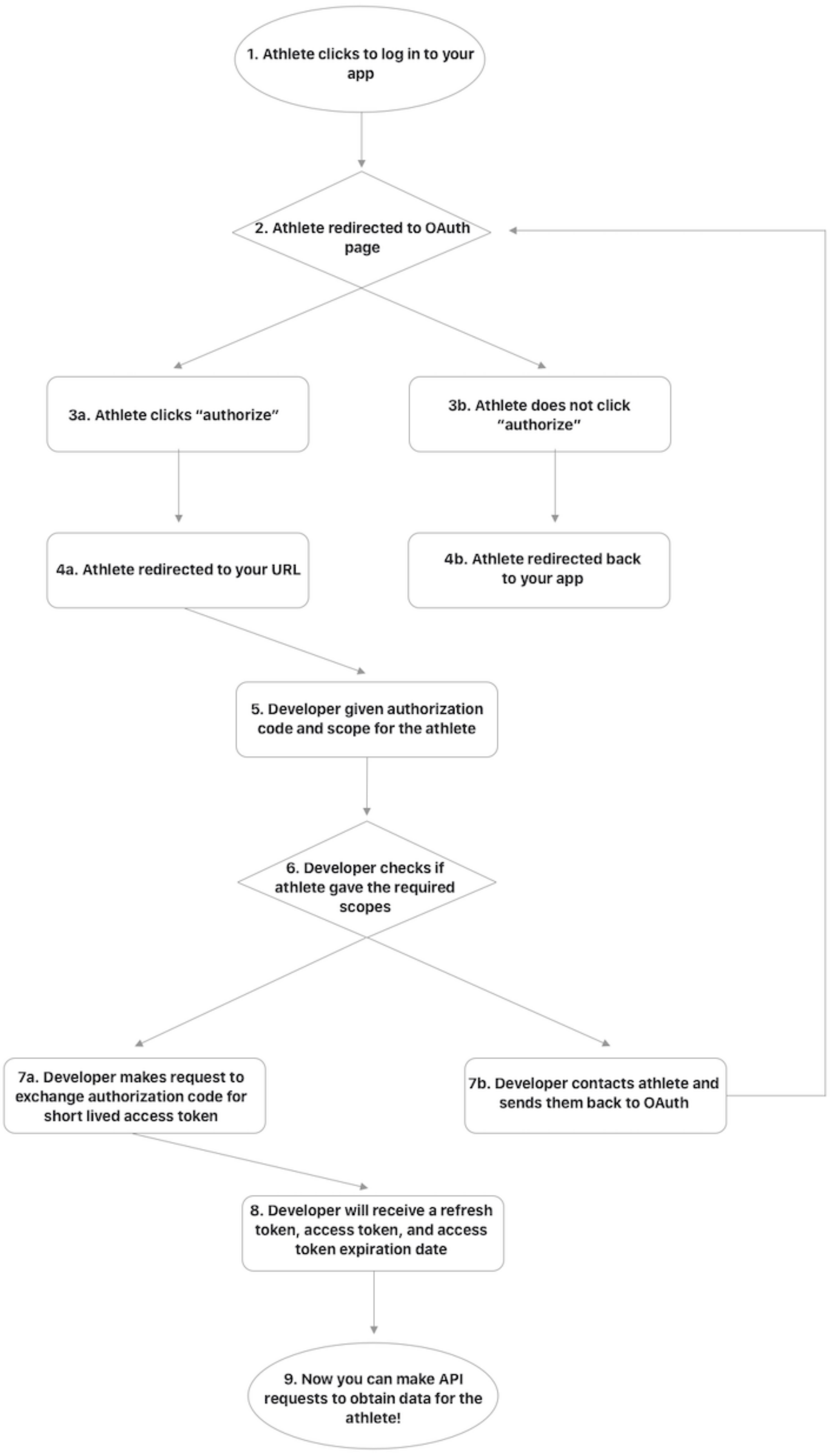

My API Application

Details

Edit

Category	Other
Club	
Client ID	96596
Client Secret	******** Show Generate New Client Secret
Your Access Token (?)	******** Show scope: read expires at: 2023-05-07T00:16:58Z
Your Refresh Token (?)	******** Show scope: read
Rate Limits (?)	100 requests every 15 minutes, 1,000 daily

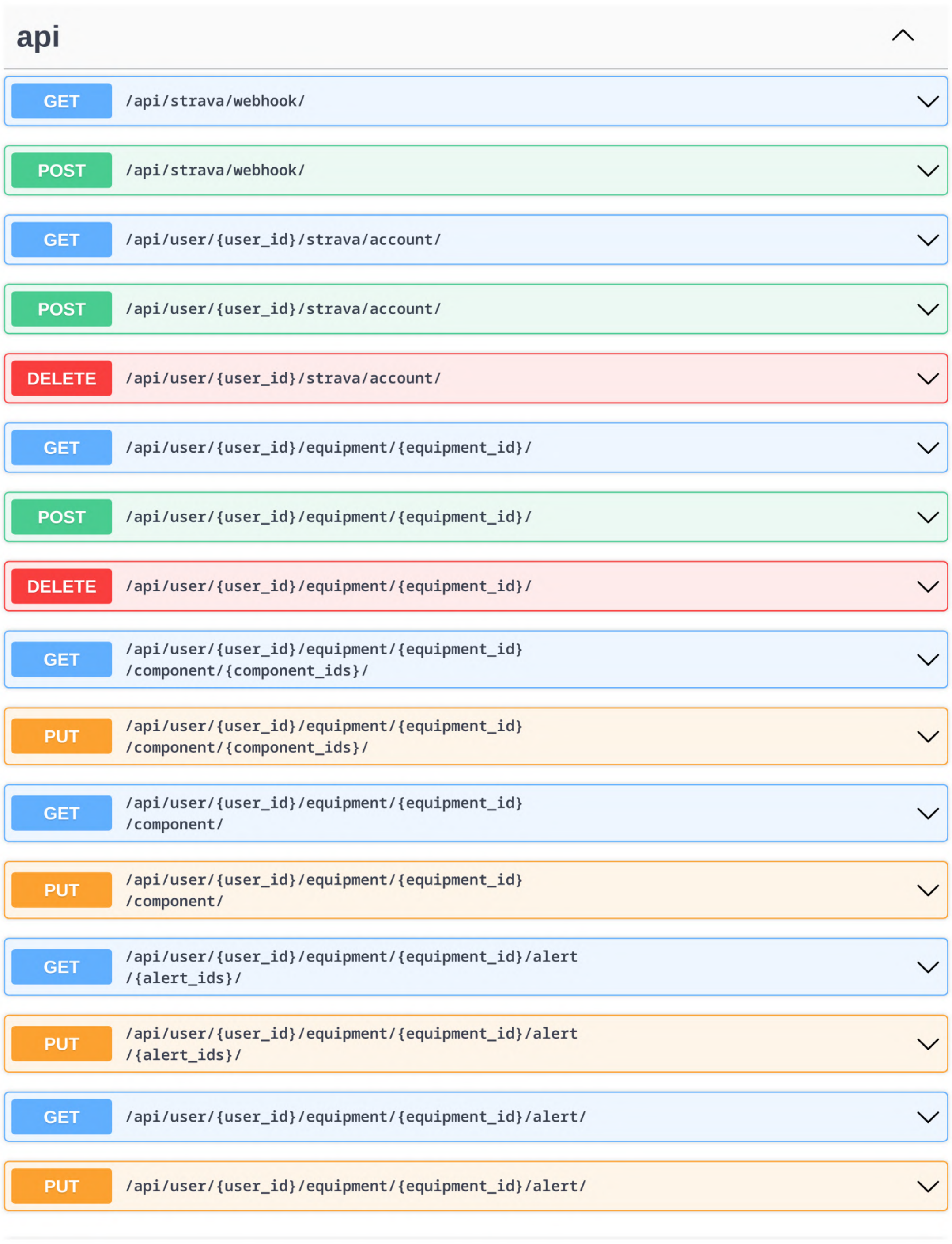
Figure A.3 Activity diagram Strava authenticaton [4]


1. Athlete clicks to log in to your app
2. Athlete redirected to OAuth page
3a. Athlete clicks "authorize"
3b. Athlete does not click "authorize"
4a. Athlete redirected to your URL
4b. Athlete redirected back to your app
5. Developer given authorization code and scope for the athlete
6. Developer checks if athlete gave the required scopes
7a. Developer makes request to exchange authorization code for short lived access token
7b. Developer contacts athlete and sends them back to OAuth
8. Developer will receive a refresh token, access token, and access token expiration date
9. Now you can make API requests to obtain data for the athlete!

Appendix B

REST API

Figure B.1 REST API Documentation



api

GET /api/strava/webhook/

POST /api/strava/webhook/

GET /api/user/{user_id}/strava/account/

POST /api/user/{user_id}/strava/account/

DELETE /api/user/{user_id}/strava/account/

GET /api/user/{user_id}/equipment/{equipment_id}/

POST /api/user/{user_id}/equipment/{equipment_id}/

DELETE /api/user/{user_id}/equipment/{equipment_id}/

GET /api/user/{user_id}/equipment/{equipment_id}/component/{component_ids}/

PUT /api/user/{user_id}/equipment/{equipment_id}/component/{component_ids}/

GET /api/user/{user_id}/equipment/{equipment_id}/component/

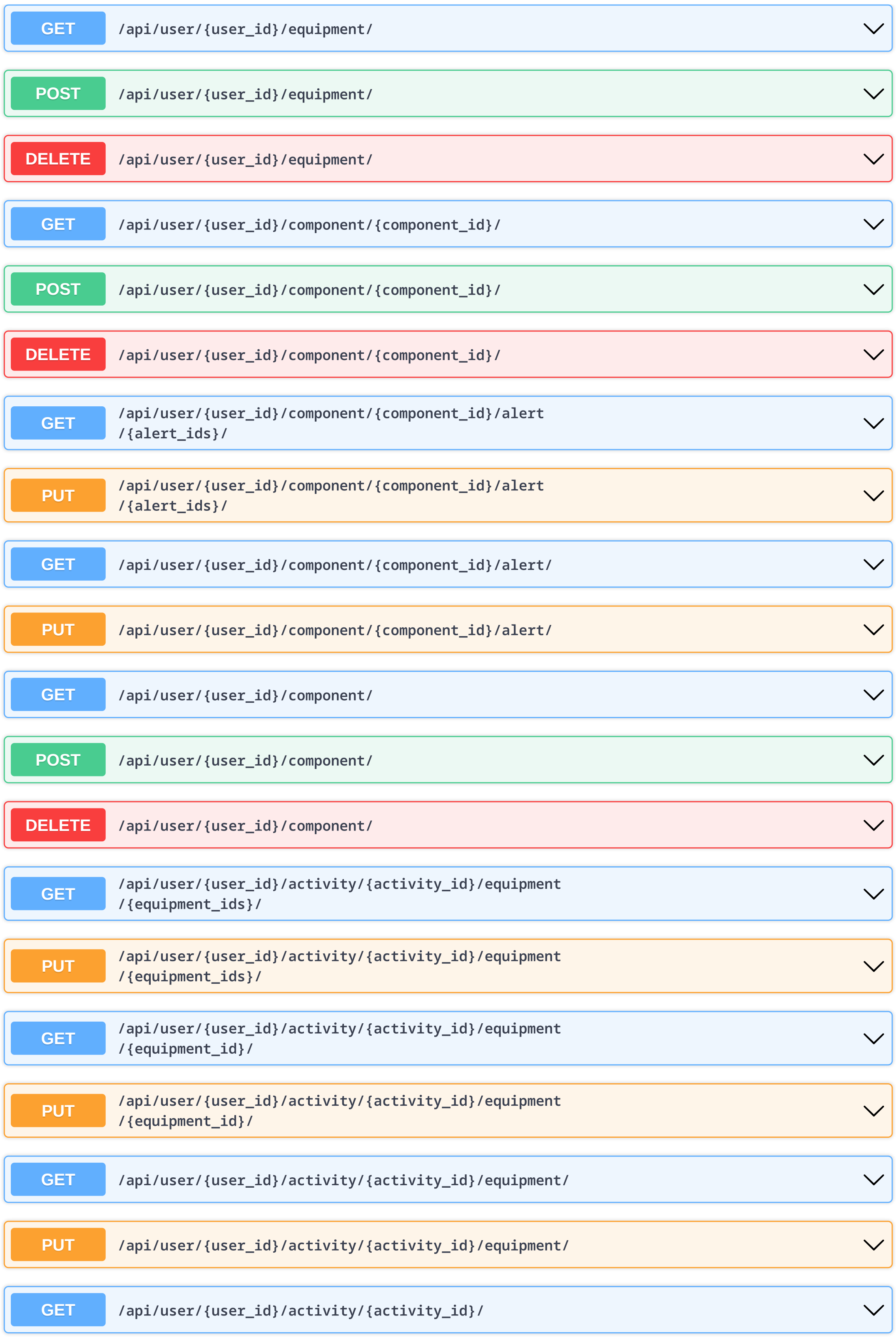
PUT /api/user/{user_id}/equipment/{equipment_id}/component/

GET /api/user/{user_id}/equipment/{equipment_id}/alert/{alert_ids}/

PUT /api/user/{user_id}/equipment/{equipment_id}/alert/{alert_ids}/

GET /api/user/{user_id}/equipment/{equipment_id}/alert/

PUT /api/user/{user_id}/equipment/{equipment_id}/alert/



GET /api/user/{user_id}/equipment/

POST /api/user/{user_id}/equipment/

DELETE /api/user/{user_id}/equipment/

GET /api/user/{user_id}/component/{component_id}/

POST /api/user/{user_id}/component/{component_id}/

DELETE /api/user/{user_id}/component/{component_id}/

GET /api/user/{user_id}/component/{component_id}/alert/{alert_ids}/

PUT /api/user/{user_id}/component/{component_id}/alert/{alert_ids}/

GET /api/user/{user_id}/component/{component_id}/alert/

PUT /api/user/{user_id}/component/{component_id}/alert/

GET /api/user/{user_id}/component/

POST /api/user/{user_id}/component/

DELETE /api/user/{user_id}/component/

GET /api/user/{user_id}/activity/{activity_id}/equipment/{equipment_ids}/

PUT /api/user/{user_id}/activity/{activity_id}/equipment/{equipment_ids}/

GET /api/user/{user_id}/activity/{activity_id}/equipment/{equipment_id}/

PUT /api/user/{user_id}/activity/{activity_id}/equipment/{equipment_id}/

GET /api/user/{user_id}/activity/{activity_id}/equipment/

PUT /api/user/{user_id}/activity/{activity_id}/equipment/

GET /api/user/{user_id}/activity/{activity_id}/

POST `/api/user/{user_id}/activity/{activity_id}/`

DELETE `/api/user/{user_id}/activity/{activity_id}/`

GET `/api/user/{user_id}/activity/`

POST `/api/user/{user_id}/activity/`

DELETE `/api/user/{user_id}/activity/`

GET `/api/user/{user_id}/alert/{alert_id}/`

POST `/api/user/{user_id}/alert/{alert_id}/`

DELETE `/api/user/{user_id}/alert/{alert_id}/`

GET `/api/user/{user_id}/alert/`

POST `/api/user/{user_id}/alert/`

DELETE `/api/user/{user_id}/alert/`

POST `/api/auth/login/`

POST `/api/auth/signup/`

POST `/api/auth/logout/`

Schemas

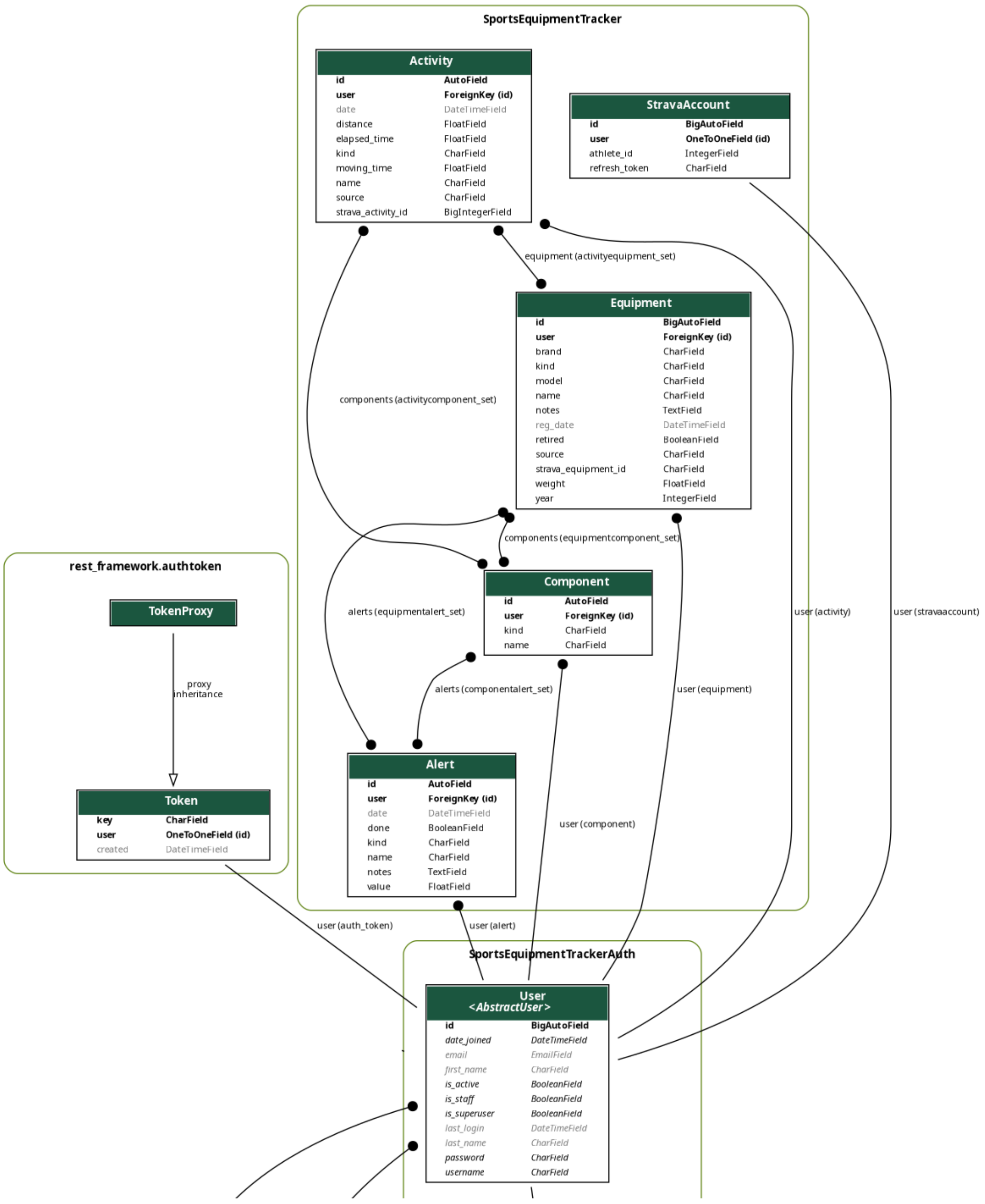
AuthToken ›

Appendix C
Server side implementation

C.1 Implementation of webhooks for receiving activities from Strava

```
class StravaWebhookHandlerView(APIView):
  authentication_classes = [TokenAuthentication]
  permission_classes = [AllowAny]

  def post(self, request):
    data = request.body
    data = json.loads(data)
    aspect_type = data.get('aspect_type')
    object_id = data.get('object_id')
    object_type = data.get('object_type')
    owner_id = data.get('owner_id')

    if not object_type == "activity":
      return JsonResponse({"message": "EVENT_RECEIVED"}, status=200)

    if owner_id:
      try:
        strava_account = StravaAccount.objects.get(athlete_id=owner_id)
        user = strava_account.user
        access_token = get_access_token(strava_account)
        if not access_token:
          return JsonResponse({"message": "EVENT_RECEIVED"}, status=200)

        try:
          activity = Activity.objects.get(
                user=user,
                strava_activity_id=object_id
          )
        except (Activity.DoesNotExist, ObjectDoesNotExist, AttributeError):
          activity = None

        serializer = ActivitySerializer()
      if aspect_type == "create":
        if not activity:
          activity_details = get_activity_details(object_id, access_token)
          if activity_details:
            activity = serializer.create({
              "user": user,
              "name": activity_details.get("name"),
              "distance": activity_details.get("distance"),
              "moving_time": activity_details.get("moving_time"),
              "elapsed_time": activity_details.get("elapsed_time"),
              "source": "strava",
              "kind": activity_details.get("type"),
              "strava_activity_id": activity_details.get("id")
            })
            activity.save()
      elif aspect_type == "update":
        activity_details = get_activity_details(object_id, access_token)
        if activity and activity_details:
          serialized_activity = serializer.update(activity, {
            "name": activity_details.get("name"),
            "distance": activity_details.get("distance"),
            "moving_time": activity_details.get("moving_time"),
            "elapsed_time": activity_details.get("elapsed_time"),
            "kind": activity_details.get("type")
          })
          serialized_activity.save()
      elif aspect_type == "delete":
        if activity:
          activity.delete()

    except StravaAccount.DoesNotExist:
      pass

  return JsonResponse(data={}, status=HTTP_200_OK)

def get(self, request):
  print("webhook verification received!", request.GET, request.body)
  mode = request.GET.get('hub.mode')
  token = request.GET.get('hub.verify_token')
  challenge = request.GET.get('hub.challenge')
  if mode and token:
    if mode == 'subscribe' and token == VERIFY_TOKEN:
      return JsonResponse({"hub.challenge": challenge}, status=HTTP_200_OK)
    return JsonResponse(status=HTTP_403_FORBIDDEN)

  return JsonResponse(status=HTTP_400_BAD_REQUEST)
```

Figure C.1 Main section of the database schema



C.2 Functions for working with the cache

```
def get_access_token(strava_account):
  access_token = cache.get(f"strava_account_token_{strava_account.user.id}")
  expires_at = cache.get(f"strava_account_expires_at_{strava_account.user.id}")
  if access_token and expires_at:
    if expires_at > time.time():
      return access_token

  # refresh the token
  refresh_token = strava_account.refresh_token
  response = requests.post(
    url="https://www.strava.com/oauth/token",
    headers={"Accept": "application/json"},
    data={
      "client_id": CLIENT_ID,
      "client_secret": CLIENT_SECRET,
      "grant_type": "refresh_token",
      "refresh_token": refresh_token
    }
  )
  if response.ok:
    data = json.loads(response.text)
    access_token = data.get("access_token")
    expires_at = data.get("expires_at")
    if access_token and expires_at:
      set_access_token(strava_account, access_token, expires_at)
      return access_token
  return None

def set_access_token(strava_account, access_token, expires_at):
  cache.set(f"strava_account_token_{strava_account.user.id}", access_token,
            timeout=None)
  cache.set(f"strava_account_expires_at_{strava_account.user.id}", expires_at,
            timeout=None)
```

C.3 Function to get activity details

```
def get_activity_details(activity_id, access_token):
  response = requests.get(
    f"https://www.strava.com/api/v3/activities/{activity_id}",
    headers={
      "Authorization": f"Bearer {access_token}",
    }
  )
  if response.ok:
    return json.loads(response.content)
  return None
```

Appendix D

Client side implementation

D.1 Provider creation

```
export const SportsProvider: React.FC<SportsProviderProps> = ({children}) => {
  const [activities, dispatchActivities] = useReducer<Reducer<Activity[],
                                                              ActivityAction>>
    (activityReducer, []);
  ...
  const [selectedActivity, setSelectedActivity] = useState<Activity |
                                                           null>(null);
  ...
  return (
    <sportsContext.Provider
      value={{
        activities: activities,
        selectedActivity: selectedActivity
        }}
       >
        <sportsDispatchContext.Provider
          value={{
            dispatchActivities: dispatchActivities,
            setSelectedActivity: setSelectedActivity
          }}>
            {children}
        </sportsDispatchContext.Provider>
    </sportsContext.Provider>
  );
};
```

D.2 Function that performs HTTP request

```
export const makeHTTPRequest = async <T>(
        token: string,
        method: 'GET' | 'POST' | 'PUT' | 'PATCH' | 'DELETE',
        uri: string, body: string=''
 ): Promise<T> => {
  const fetchOptions: RequestInit = {
    method: method,
    headers: {
      'Content-Type': 'application/json',
      'Authorization': `Token ${token}`
    },
  };

  if (method === 'POST') {
    fetchOptions.body = body;
  }

  const response = await fetch(concatUrl(baseAPIUrl, uri), fetchOptions)
  if (response.ok) {
    const data: T = await response.json()
    return data
  }
  else {
    const error = await response.text()
    throw new Error("makeHTTPRequest error: " + error)
  }
}
```

D.3 Functions for working with local storage

```
export const setLocalStorage = <T>(key: string, value: T): void => {
  try {
    const storedValue = localStorage.getItem(key);
    if (storedValue !== null) {
        const existingValue = JSON.parse(storedValue);
        const newValue = { ...existingValue, ...value };
        localStorage.setItem(key, JSON.stringify(newValue));
    }
    else {
        localStorage.setItem(key, JSON.stringify(value));
    }
  }
  catch (error) {}
};

export const getLocalStorage = <T>(key: string): (T | null)  => {
  const storedValue = localStorage.getItem(key);
  return storedValue ? JSON.parse(storedValue) : null;
}

export const removeLocalStorage = (key: string): void => {
  localStorage.removeItem(key);
}
```

D.4 Component which prevents access for unauthenticated users

```
export const RequireAuth = (props: { children: any;
                    usedLocalStorage: boolean; }) => {
  const usedLocalStorage = props.usedLocalStorage
  const user = useUser()?.user
  const navigate = useNavigate()
  const loggedIn: Boolean = !!user && (user.id !== 0)
  const [shouldRedirect, setShouldRedirect] = React.useState(false)

  React.useEffect(() => {
    if (usedLocalStorage && !loggedIn) {
      navigate("/auth")
      setShouldRedirect(true)
    }
  }, [usedLocalStorage])

  return shouldRedirect ? navigate("/auth") : props.children
}
```

D.5 The component to which the Strava auth page redirects after the user logs in

```
const StravaRedirect = () => {
  const location = useLocation();
  const query = new URLSearchParams(location.search);
  const code = query.get('code');
  const navigate = useNavigate();
  ...
  const addStravaAccount = (data: StravaAuthData) => {
    ...
    postUserStravaAccount(data, token, userId)
      .then((response: StravaAuthResponseData) => {
        ...
        userDispatch({
          type: 'ADD_STRAVA_ACCOUNT',
          stravaId: response.athlete_id,
        })
      })
      .catch((error) => {...})
    navigate("/")
  }
  ...
  useEffect(() => {
    if (!usedLocalStorage) return;
    fetch(
      `https://www.strava.com/oauth/token?
      client_id=${CLIENT_ID}&
      client_secret=${REACT_APP_CLIENT_SECRET}&
      code=${code}&
      grant_type=authorization_code`,
      { method: 'POST' }
    )
      .then(response => response.json())
      .then((data: StravaAuthData) => {
        addStravaAccount(data)
      })
      .catch((error) => {...})
  }, [usedLocalStorage])
  ...
  return (<>...</>)
}

export default StravaRedirect;
```

Appendix E

Deployment

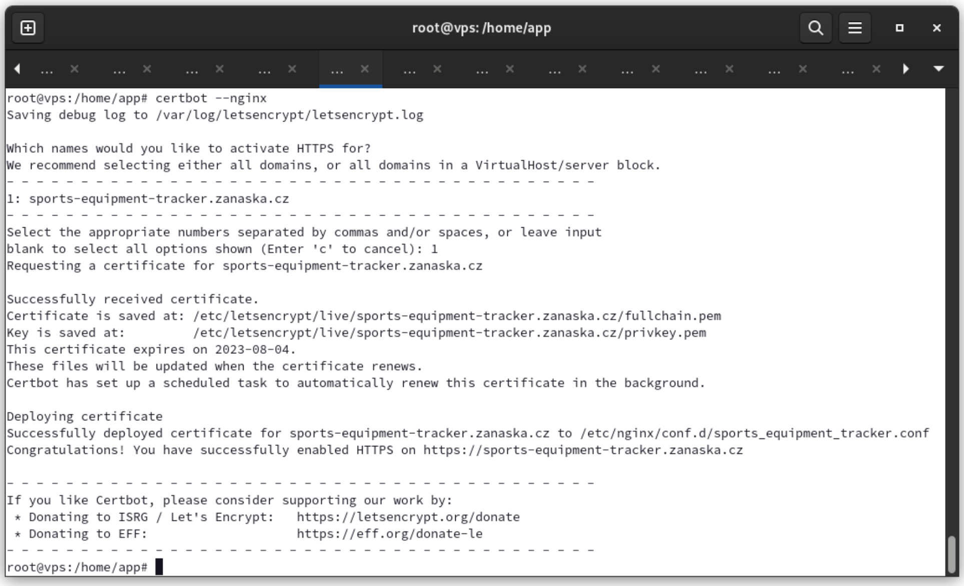
E.1 Dockerfile for React app

```
FROM node:20-alpine3.16 as node_build
WORKDIR /usr/app
COPY frontend/sports_equipment_tracker/ /usr/app

RUN npm ci --legacy-peer-deps
RUN npm run build

FROM nginx:1.23.1-alpine
EXPOSE 80
COPY ./react_nginx_config.conf /etc/nginx/conf.d/default.conf
COPY --from=node_build /usr/app/build /usr/share/nginx/html
```

E.2 Docker compose file

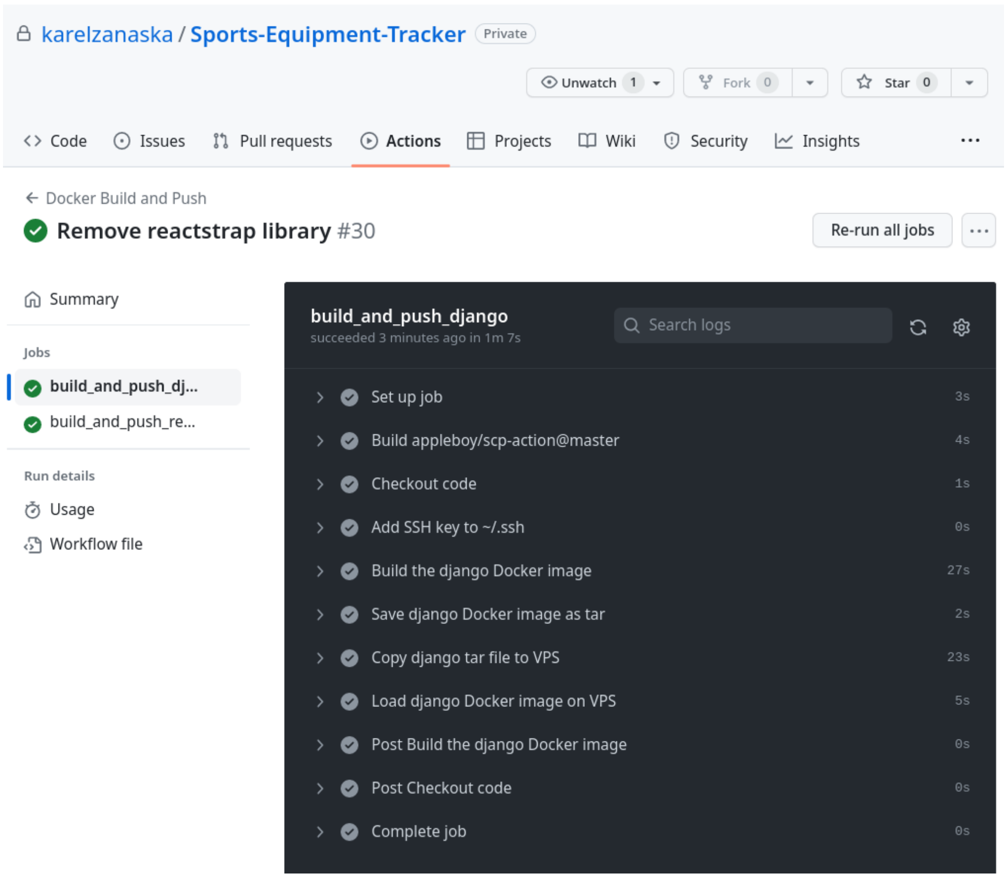
```yaml
version: '3.7'

services:
  db:
    image: postgres:15.1-alpine
    volumes:
      - postgres_data:/var/lib/postgresql/data/
    ports:
      - "5432:5432"
    environment:
      - POSTGRES_DB=SportsEquipmentTracker
      - POSTGRES_USER=root
      - POSTGRES_PASSWORD=whateverpassword
    networks:
      - backend_network
  backend:
    image: tracker_django
    volumes:
      - .:/app
    ports:
      - "200:8000"
    depends_on:
      - db
    environment:
      SQL_HOST: db
      SQL_PORT: 5432
    networks:
      - backend_network
  frontend:
    image: tracker_react
    ports:
      - "201:80"

volumes:
  postgres_data:

networks:
  backend_network:
```

Figure E.1 Screenshot of the Certbot tool that generates the certificate



```
root@vps:/home/app# certbot --nginx
Saving debug log to /var/log/letsencrypt/letsencrypt.log

Which names would you like to activate HTTPS for?
We recommend selecting either all domains, or all domains in a VirtualHost/server block.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
1: sports-equipment-tracker.zanaska.cz
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Select the appropriate numbers separated by commas and/or spaces, or leave input
blank to select all options shown (Enter 'c' to cancel): 1
Requesting a certificate for sports-equipment-tracker.zanaska.cz

Successfully received certificate.
Certificate is saved at: /etc/letsencrypt/live/sports-equipment-tracker.zanaska.cz/fullchain.pem
Key is saved at:         /etc/letsencrypt/live/sports-equipment-tracker.zanaska.cz/privkey.pem
This certificate expires on 2023-08-04.
These files will be updated when the certificate renews.
Certbot has set up a scheduled task to automatically renew this certificate in the background.

Deploying certificate
Successfully deployed certificate for sports-equipment-tracker.zanaska.cz to /etc/nginx/conf.d/sports_equipment_tracker.conf
Congratulations! You have successfully enabled HTTPS on https://sports-equipment-tracker.zanaska.cz

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
If you like Certbot, please consider supporting our work by:
 * Donating to ISRG / Let's Encrypt:   https://letsencrypt.org/donate
 * Donating to EFF:                    https://eff.org/donate-le
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
root@vps:/home/app# 
```

Figure E.2 Screenshot of the GitHub platform showing GitHub actions



karelzanaska / Sports-Equipment-Tracker Private

Unwatch 1 Fork 0 Star 0

Code Issues Pull requests Actions Projects Wiki Security Insights

Docker Build and Push

Remove reactstrap library #30

Re-run all jobs

Summary

Jobs

build_and_push_dj...

build_and_push_re...

Run details

Usage

Workflow file

build_and_push_django

succeeded 3 minutes ago in 1m 7s

Search logs

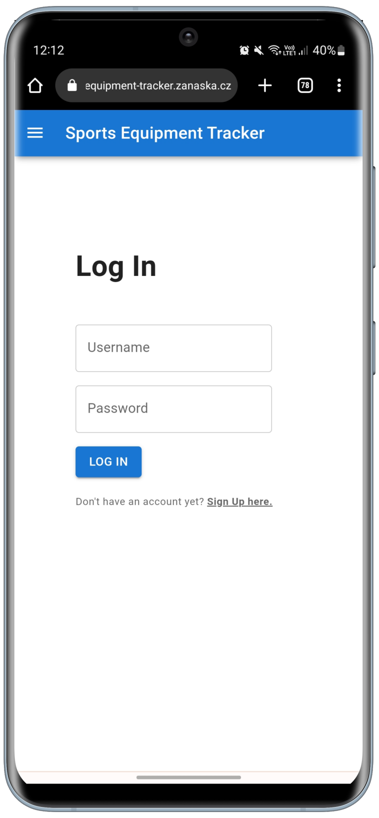
Set up job	3s
Build appleboy/scp-action@master	4s
Checkout code	1s
Add SSH key to ~/.ssh	0s
Build the django Docker image	27s
Save django Docker image as tar	2s
Copy django tar file to VPS	23s
Load django Docker image on VPS	5s
Post Build the django Docker image	0s
Post Checkout code	0s
Complete job	0s

E.3 The docker-build.yml file defining GitHub actions

```
name: Docker Build and Push to VPS

on:
  push:
    branches:
      - main
jobs:
  build_and_push_django:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout code
      uses: actions/checkout@v2

    - name: Add SSH key to ~/.ssh
      run: |
        mkdir -p ~/.ssh
        echo "${{ secrets.VPS_SSH_KEY}}" > ~/.ssh/id_rsa
        chmod 600 ~/.ssh/id_rsa

    - name: Build the django Docker image
      uses: docker/build-push-action@v2
      with:
        context: .
        file: ./Dockerfile-django
        push: false
        tags: tracker_django:latest

    - name: Save django Docker image as tar
      run: |
        docker save tracker_django:latest -o tracker-django-image.tar

    - name: Copy django tar file to VPS
      uses: appleboy/scp-action@master
      with:
        host: ${{ secrets.VPS_HOST }}
        username: root
        port: 22
        source: ./tracker-django-image.tar
        target: /home/app/
        key: ${{ secrets.VPS_SSH_KEY }}
        overwrite: true

    - name: Load django Docker image on VPS
      run: |
        ssh -o "StrictHostKeyChecking=no" root@${{ secrets.VPS_HOST }}
               "docker load < /home/app/tracker-django-image.tar"
  ...
```

Appendix F
Application showcase

Figure F.1 Login page


12:12
equipment-tracker.zanaska.cz
Sports Equipment Tracker

Log In

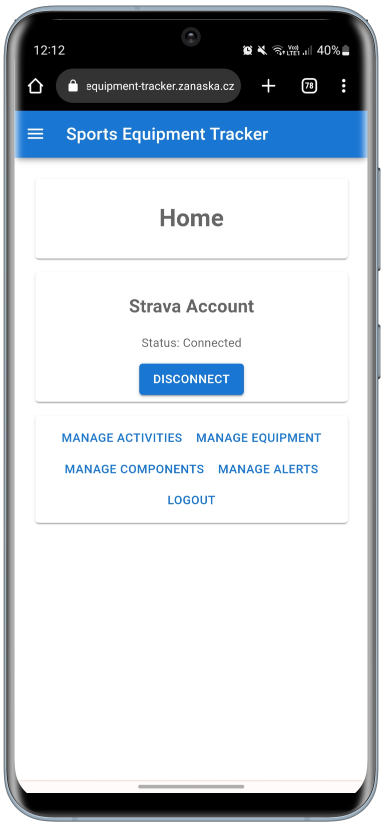
Username

Password

LOG IN

Don't have an account yet? Sign Up here.

Figure F.2 Home page


12:12
equipment-tracker.zanaska.cz

Sports Equipment Tracker

Home

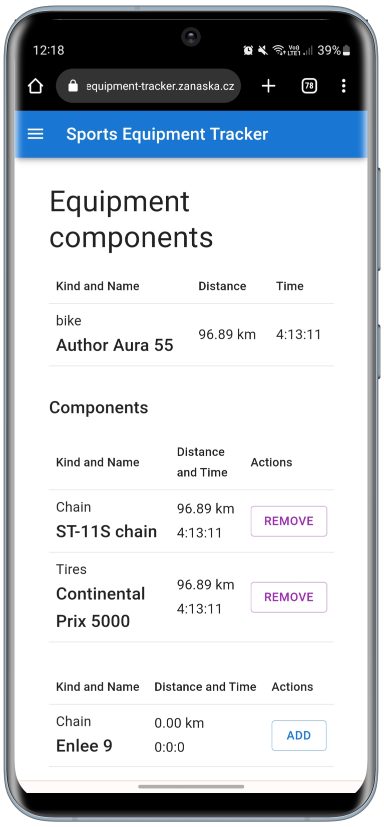
Strava Account

Status: Connected

DISCONNECT

MANAGE ACTIVITIES MANAGE EQUIPMENT MANAGE COMPONENTS MANAGE ALERTS LOGOUT

Figure F.3 Equipment components page


12:18
equipment-tracker.zanaska.cz

Sports Equipment Tracker

Equipment components

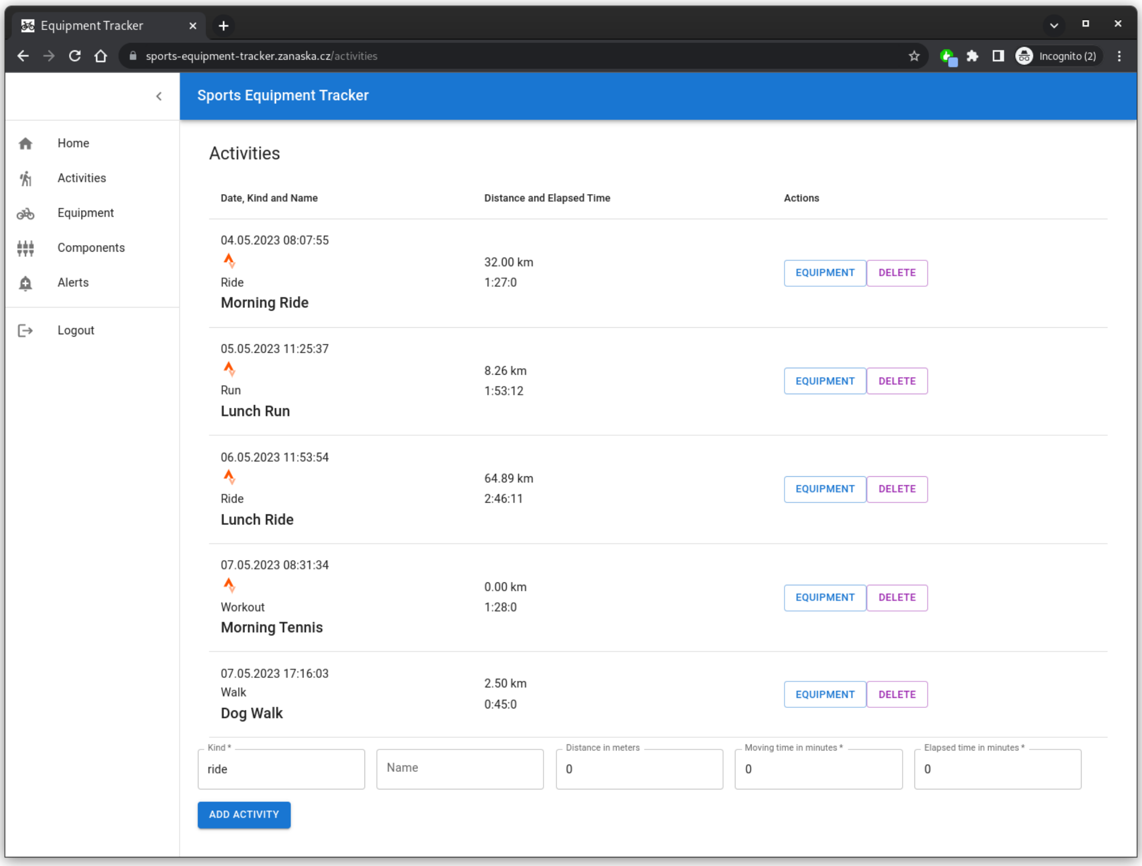
Kind and Name	Distance	Time
bike Author Aura 55	96.89 km	4:13:11

Components

Kind and Name	Distance and Time	Actions
Chain ST-11S chain	96.89 km 4:13:11	REMOVE
Tires Continental Prix 5000	96.89 km 4:13:11	REMOVE

Kind and Name	Distance and Time	Actions
Chain Enlee 9	0.00 km 0:0:0	ADD

Figure F.4 Activities page


Equipment Tracker

sports-equipment-tracker.zanaska.cz/activities

Sports Equipment Tracker

Home
Activities
Equipment
Components
Alerts
Logout

Activities

Date, Kind and Name	Distance and Elapsed Time	Actions
04.05.2023 08:07:55 Ride **Morning Ride**	32.00 km 1:27:0	EQUIPMENT DELETE
05.05.2023 11:25:37 Run **Lunch Run**	8.26 km 1:53:12	EQUIPMENT DELETE
06.05.2023 11:53:54 Ride **Lunch Ride**	64.89 km 2:46:11	EQUIPMENT DELETE
07.05.2023 08:31:34 Workout **Morning Tennis**	0.00 km 1:28:0	EQUIPMENT DELETE
07.05.2023 17:16:03 Walk **Dog Walk**	2.50 km 0:45:0	EQUIPMENT DELETE

Kind *	Name	Distance in meters	Moving time in minutes *	Elapsed time in minutes *
ride		0	0	0

ADD ACTIVITY

Bibliography

1. TRAJANOV, Tosho. Functional and non-functional requirements. In: *Adeva* [online]. Adeva, 2021 [visited on 2023-04-15]. Available from: `https://adevait.com/software/functional-non-functional-requirements`.
2. Running, Cycling amp; Hiking App – Train, track amp; share. In: *Strava* [online]. [N.d.] [visited on 2023-04-15]. Available from: `https://www.strava.com/features`.
3. FIT ČVUT. In: *Strava* [online]. Strava, [n.d.] [visited on 2023-04-15]. Available from: `https://www.strava.com/clubs/fit-cvut`.
4. Getting Started with the Strava API. In: *Strava Developers* [online]. [N.d.] [visited on 2023-04-15]. Available from: `https://developers.strava.com/docs/getting-started/`.
5. In: *Strava* [online]. 2021 [visited on 2023-05-01]. Available from: `https://www.strava.com/clubs/231407/posts/17891665?hl=en-GB`.
6. strava-gear. In: *GitHub* [online]. [N.d.] [visited on 2023-04-15]. Available from: `https://github.com/liskin/strava-gear`.
7. Bicycle maintenance tracker app. In: *ProBikeGarage* [online]. [N.d.] [visited on 2023-04-15]. Available from: `https://www.probikegarage.com/`.
8. An overview of HTTP. In: *MDN* [online]. 2023 [visited on 2023-04-15]. Available from: `https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview`.
9. HTTP request methods. In: *MDN* [online]. 2023 [visited on 2023-04-15]. Available from: `https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods`.
10. Evolution of HTTP. In: *MDN* [online]. 2023 [visited on 2023-04-15]. Available from: `https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Evolution_of_HTTP`.
11. DUCKETT, Jon. *HTML amp; CSS design: Design and build websites*. John Wiley amp; sons, inc., 2011. ISBN 978-1-118-00818-8.
12. Introduction to the DOM. In: *MDN* [online]. 2023 [visited on 2023-04-15]. Available from: `https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction`.
13. Document. In: *MDN* [online]. 2023 [visited on 2023-04-15]. Available from: `https://developer.mozilla.org/en-US/docs/Web/API/Document`.
14. BIRO, Jonathan. Browser engines... chromium, V8, Blink? gecko? webkit? In: *Medium* [online]. Medium, 2019 [visited on 2023-04-15]. Available from: `https://medium.com/@jonbiro/browser-engines-chromium-v8-blink-gecko-webkit-98d6b0490968`.

15. Introducing json. In: *JSON* [online]. [N.d.] [visited on 2023-04-15]. Available from: https://www.json.org/json-en.html.

16. What is an API? In: *RedHat* [online]. 2022 [visited on 2023-04-15]. Available from: https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces.

17. FIELDING, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures*. 2000. PhD thesis. University of California, Irvine.

18. RESTful web API design. In: *Azure Architecture Center — Microsoft Learn* [online]. [N.d.] [visited on 2023-04-15]. Available from: https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design.

19. RESELMAN, Bob. An architect's guide to APIs: Soap, rest, GraphQL, and grpc. In: *Enable Architect* [online]. Red Hat, Inc., 2020 [visited on 2023-04-15]. Available from: https://www.redhat.com/architect/apis-soap-rest-graphql-grpc.

20. Comparing API Architectural Styles: Soap vs REST vs GraphQL vs RPC. In: *AltexSoft* [online]. AltexSoft, 2020 [visited on 2023-04-15]. Available from: https://www.altexsoft.com/blog/soap-vs-rest-vs-graphql-vs-rpc/.

21. H, Jeremy. gRPC vs. REST: How Does gRPC Compare with Traditional REST APIs? In: *FreamFactory Blog* [online]. 2022 [visited on 2023-04-15]. Available from: https://blog.dreamfactory.com/grpc-vs-rest-how-does-grpc-compare-with-traditional-rest-apis/.

22. BUSH, Thomas. CRUD vs. REST: What's the Difference? In: *Nordic APIs* [online]. 2020 [visited on 2023-04-15]. Available from: https://nordicapis.com/crud-vs-rest-whats-the-difference/.

23. What is a webhook? In: *Red Hat - We make open source technologies for the enterprise* [online]. 2022 [visited on 2023-04-15]. Available from: https://www.redhat.com/en/topics/automation/what-is-a-webhook.

24. What Is A Web Application? In: *Amazon* [online]. [N.d.] [visited on 2023-04-15]. Available from: https://aws.amazon.com/what-is/web-application/.

25. Single Page Applications (SPA). In: *AppCheck* [online]. 1978 [visited on 2023-04-15]. Available from: https://appcheck-ng.com/single-page-applications.

26. Server-side web frameworks. In: [online]. 2023 [visited on 2023-04-15]. Available from: https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Web_frameworks.

27. Architecture. In: *Hands on React* [online]. [N.d.] [visited on 2023-04-15]. Available from: https://handsonreact.com/docs/architecture.

28. What is javascript? In: *MDN* [online]. 2023 [visited on 2023-04-15]. Available from: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript.

29. TypeScript for JavaScript Programmers. In: *TypeScript* [online]. 2023 [visited on 2023-04-15]. Available from: https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html.

30. DESHPANDE, Chinmayee. The Best Guide to Know What Is React. In: *Simplilearn* [online]. 2023 [visited on 2023-04-15]. Available from: https://www.simplilearn.com/tutorials/reactjs-tutorial/what-is-reactjs.

31. OLIVEIRA, Domingos F.; GOMES, João; PEREIRA, Ricardo; BRITO, Miguel; MACHADO, Ricardo-J. Development of a Self-Diagnostic System Integrated into a Cyber-Physical System. *Computers*. 2022, vol. 11, p. 131. Available from DOI: 10.3390/computers11090131.

32. Material UI – Overview. In: *Material UI* [online]. [N.d.] [visited on 2023-04-15]. Available from: `https://mui.com/material-ui/getting-started/overview/`.

33. H., Swaroop C. *A Byte of Python*. Open Textbook Library, 2013.

34. ELMAN, Julia; LAVIN, Mark. *Lightweight Django*. O'Reilly Media, 2015.

35. Django introduction. In: *MDN* [online]. 2023 [visited on 2023-04-15]. Available from: `https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Introduction`.

36. SINGHAL, Gaurav. How to create a REST API with Django REST framework. In: *LogRocket Blog* [online]. 2022 [visited on 2023-04-15]. Available from: `https://blog.logrocket.com/django-rest-framework-create-api/#what-is-django`.

37. NATHANAEL, Dave. Design Pattern: Django Rest Framework. In: *Medium* [online]. Medium, 2020 [visited on 2023-04-15]. Available from: `https://davenathanaeld.medium.com/design-pattern-django-rest-framework-1e8c17946bce`.

38. Types of Databases. In: *GeeksforGeeks* [online]. GeeksforGeeks, 2021 [visited on 2023-04-15]. Available from: `https://www.geeksforgeeks.org/types-of-databases/`.

39. SUMATHI, S.; ESAKKIRAJAN, S. *Fundamentals of Relational Database Management Systems*. Springer, 2007.

40. In: *PostgreSQL* [online]. PostgreSQL, 2023 [visited on 2023-04-15]. Available from: `https://postgres.cz/wiki/PostgreSQL`.

41. What is PostgreSQL? In: [online]. 1999 [visited on 2023-04-15]. Available from: `https://aws.amazon.com/rds/postgresql/what-is-postgresql/`.

42. In: *IBM* [online]. [N.d.] [visited on 2023-05-01]. Available from: `https://www.ibm.com/topics/docker`.

43. In: *Docker Documentation* [online]. 2023 [visited on 2023-05-01]. Available from: `https://docs.docker.com/build/ci/github-actions/`.

44. In: *NGINX* [online]. 2023 [visited on 2023-05-01]. Available from: `https://www.nginx.com/resources/glossary/nginx/`.

45. What Is Token-Based Authentication? In: *Okta* [online]. 2023 [visited on 2023-04-15]. Available from: `https://www.okta.com/identity-101/what-is-token-based-authentication/`.

46. Django's cache framework. In: *Django Project* [online]. [N.d.] [visited on 2023-05-01]. Available from: `https://docs.djangoproject.com/en/4.2/topics/cache/`.

47. Webhook Events API. In: *Strava Developers* [online]. [N.d.] [visited on 2023-05-01]. Available from: `https://developers.strava.com/docs/webhooks/`.

48. Component Architecture. In: *Hands on React* [online]. [N.d.] [visited on 2023-05-01]. Available from: `https://handsonreact.com/docs/component-architecture#container-and-presentation-components`.

49. SHARMA, Rohit. Class Components and Functional Components in Reactjs. In: *Let's React* [online]. 2022 [visited on 2023-05-01]. Available from: `https://www.letsreact.org/class-components-and-functional-components-in-reactjs/`.

50. MAJ, Wojciech. React Lifecycle Methods diagram. In: *GitHub* [online]. [N.d.] [visited on 2023-05-01]. Available from: `https://github.com/wojtekmaj/react-lifecycle-methods-diagram`.

51. MARANAN, Menard. The React lifecycle: methods and hooks explained. In: [online]. 2022 [visited on 2023-05-01]. Available from: `https://retool.com/blog/the-react-lifecycle-methods-and-hooks-explained/`.

52. MONSANTO, Marco. Routing in SPAs. In: *DEV Community* [online]. 2020 [visited on 2023-05-01]. Available from: `https://dev.to/marcomonsanto/routing-in-spas-173i`.
53. Tutorial V6.11.1. In: *React Router* [online]. [N.d.] [visited on 2023-05-01]. Available from: `https://reactrouter.com/en/main/start/tutorial`.
54. useNavigate V6.11.1. In: *React Router* [online]. [N.d.] [visited on 2023-05-01]. Available from: `https://reactrouter.com/en/main/hooks/use-navigate`.
55. BrowserRouter V6.11.1. In: *React Router* [online]. [N.d.] [visited on 2023-05-01]. Available from: `https://reactrouter.com/en/main/router-components/browser-router`.
56. Routes V6.11.1. In: *React Router* [online]. [N.d.] [visited on 2023-05-01]. Available from: `https://reactrouter.com/en/main/components/routes`.
57. Route V6.11.1. In: *React Router* [online]. [N.d.] [visited on 2023-05-01]. Available from: `https://reactrouter.com/en/main/components/route`.
58. BOATENG, Dickson. How to Use the React Context API in Your Projects. In: *freeCodeCamp.org* [online]. 2023 [visited on 2023-05-01]. Available from: `https://www.freecodecamp.org/news/context-api-in-react/`.
59. Using the Web Storage API. In: *Web APIs — MDN* [online]. 2023 [visited on 2023-05-01]. Available from: `https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API/Using_the_Web_Storage_API`.

Contents of the attached medium

```
readme.md ............................ description of the contents
src ..................................................................
├── backend .............................. server side source codes
├── frontend ............................. client side source codes
└── deployment ............................ deploment source codes
thesis ...............................................................
├── text ....................................... thesis in PDF
└── src .................................... thesis source codes
```