

FINAL YEAR PROJECT



Creative Media & Design B.Sc. Honours

Fortune Ndlovu - K00257222

1 May 2024

Technological University of the Shannon: Midlands Midwest - Clonmel Digital Campus

“A SOFTWARE DEVELOPMENT CONTRIBUTION INTO THE SOFTWARE DEVELOPMENT
COMMUNITY, THROUGH THE IMPLEMENTATION OF A COMPLEX WEB APP REPLICA”

Word Count: 20,000

Declaration

This thesis is my original work and has not been submitted previously for a degree at this or any other institute or university. To the best of my knowledge, it does not contain, any material published or written by another person, except as acknowledged in the text.

Signed: _____ Date: _____
Fortune Ndlovu 1 May 2024

Abstract

This thesis presents a comprehensive exploration into the world of software development, through the replication, and implementation of a complex web application. The project's highlight is the meticulous recreation of the Strava web application using modern technologies such as React and Firebase, providing an in-depth report through the complexities of the challenges and triumphs of software development.

The journey begins with a literature review that lays the groundwork for understanding Strava's functionality, the significance of coding, and the advantages of React as a development framework. The thesis then delves into the intricacies of the project, detailing the process of overcoming complexities such as data integration, real-time updates, user authentication, and social features and interactions.

Through the lens of this project, the thesis contributes to the software development community by demonstrating the practical application of theoretical concepts and the adaptability required to address unforeseen complexities. It serves as a testament to the power of perseverance and innovation in the face of complexity, offering insights and lessons learned that are invaluable.

In essence, this report is not only a narrative of replicating a complex web application but also a reflection on the personal and professional growth experienced through the journey. It underscores the importance of a methodical approach, the value of user-centric design, and the continuous pursuit of knowledge in the ever-evolving landscape of software development.

Acknowledgments

I would like to express my gratitude to my project supervisor Mr. Paul Keating, for his tremendous support, guidance, and seasoned advice throughout this project.

A special thank you to Mr. John Hannafin, for not only introducing me to modern technologies that improve user experiences but also for taking the time for professional code reviews.

I would also like to extend a sincere thank you to all the lecturers and staff of the Technological University of the Shannon: Midlands Midwest - Clonmel Digital Campus, for their exceptional teachings, and practical explorations through problem-solving, over the last four years. The hard work that all lecturers do to ensure students are learning and growing is evident and appreciated.

Moreover, I want to thank my class peers, Marie Cowan, Samantha Quinlan, Matthew Carey, and Shane O'Callaghan. Continuously improving, problem-solving, and exploring with my class peers has been a magical moment because we all supported and encouraged each other to the end which was a nice experience to be part of.

Lastly, I would like to thank my family, being the oldest in the family provided me with a mindset of wanting to work hard so my younger siblings can have an example to become even greater minds of the future. Shayla, Hayley, and Dion, my appreciation goes to my curious siblings for supporting and encouraging me throughout my studies and especially to my Mom for making everything possible.

Table of Contents

Creative Media & Design B.Sc. Honours.....	i
Declaration	ii
Abstract	iii
Acknowledgments	iv
List of Abbreviations.....	vii
Chapter 1. Introduction	1
Chapter 2. Overview and Objectives.....	2
Chapter 3. Literature Review	3
3.1. Introduction	3
3.2. Strava.....	3
3.3. Coding.....	8
3.4. React	13
3.5. Complexities in Replicating Strava in React	18
3.6. Addressing Complexities in Replicating Strava in React.....	21
3.7. Coding Design.....	24
3.8. Literature Review In Summary	27
Chapter 4. Project Complexities Process.....	28
4.1. Basic UX/UI.....	28
4.2. Activity Management	47
4.3. Multiple User Integration	79
Chapter 5. Personal & Professional Reflection - What I Learned as a Person	101
Chapter 7. Conclusion	103
Bibliography and References.....	104
Appendix 1 - List of Figures	108
Appendix 2 - List of Tables.....	111
Appendix 3 - Definitions.....	112
Architecture.....	112
Application Programming Interface (API)	112
Amazon Web Services (AWS).....	112
Authentication (Auth).....	112
Cascading Style Sheet (CSS)	113
Document Object Model (DOM)	113

Firebase.....	113
Hyper Text Markup Language (HTML)	113
HyperText Transfer Protocol (HTTP)	113
Integrated Development Environment (IDE).....	114
Infrastructure.....	114
JavaScript (JS).....	114
HyperText Processor (PHP)	114
Portable Network Graphics (PNG)	114
React	115
Software Development Kits (SDK)	115
Scalable Vector Graphics (SVG)	115
User Interface (UI).....	115
User Identifier (UID)	115
Uniform Resource Locator (URL)	116
User Experience (UX)	116
Virtual Document Object Model (VDOM).....	116

List of Abbreviations

API: Application Programming Interface

AWS: Amazon Web Services

Auth: Authentication

CCPA: California Consumer Privacy Act

COBOL: Common Business Oriented Language

CRUD: Create, Read, Update, and Delete

CSS: Cascading Style Sheet

DIV: Division

DOM: Document Object Model

EU: European Union

GCP: Google Cloud Platform

GDPR: General Data Protection Regulation

GPS: Global Positioning System

HTML: Hyper Text Markup Language

HTTP: HyperText Transfer Protocol

IDE: Integrated Development Environment

JS: JavaScript

JSX: JavaScript Extensible Markup Language

LISP: list Processing

LTM: Long-Term Memory

PHP: Hyper Text Processor

PNG: Portable Network Graphics

SDK: Software Development Kits

STM: Short-Term Memory

SVG: Scalable Vector Graphics

UI: User Interface

UID: User Identifier

URL: Uniform Resource Locator

US: United States

UX: User Experience

VDOM: Virtual Document Object Model

Chapter 1. Introduction

This report presents an in-depth exploration of the process of replicating the Strava web application using React and Firebase, two powerful tools in the current technological landscape.

The report begins with an overview and objectives, setting the stage for the detailed exploration that follows. A comprehensive literature review provides insights into Strava, its founding, history, technological architecture, and data security. The importance of coding, its origin, its evolution, and its distinction from programming are also discussed.

The report then delves into the specifics of React, discussing its features, advantages, and why it was chosen for this project. The complexities involved in replicating Strava in React are examined, including aspects like data integration, real-time updates, activity analysis, visualization, user authentication, security, social features, interactions, and handling external APIs and services.

The report further discusses how these complexities were addressed, providing a detailed look at the coding design, technology stack, and the modular component-based architecture used. The report also covers the project complexities process, discussing the design and implementation of basic UX/UI components, activity management, and multiple-user integration.

A personal and professional reflection provides insights into the learning journey of the developer. The report concludes with a summary of the findings and their implications for the software development community.

This report serves as a comprehensive guide for understanding the process of replicating a complex web application like Strava using modern technologies like React and Firebase. It provides valuable insights and lessons learned, contributing to the ongoing dialogue in the software development community.

Chapter 2. Overview and Objectives

This comprehensive report is a testament to the journey of a passionate programmer with also interest in UX/UI engineering, who looked to solidify their expertise in software development by undertaking a complex project. The project involved replicating a complex web application, Strava, using modern technologies. This endeavour was not merely an academic exercise, but a personal mission for the programmer to ensure that programming was their core centre, passion and most thought-out skill to master.

The primary Objectives of this report are as follows:

1. To Demonstrate Proficiency in Programming: The project serves as a platform to showcase the author's programming skills. Particularly in the context of Full Stack Development.
2. To apply UX/UI Principles: Given the author's background in UX/UI, an integral part of the project was to incorporate these principles into the design and engineering of the web application.
3. To Showcase Research and Planning Expertise: The extensive literature review conducted as part of the report demonstrates the author's ability to research effectively and plan meticulously. It reflects the author's understanding of the subject matter and their ability to synthesize and apply this knowledge.
4. To Understand and Overcome Complexities: Replacing a complex web application like Strava posed numerous complexities. One of the key objectives was to understand these complexities and devise effective solutions.
5. To Contribute to the Software Development Community: By documenting the process, and lessons learned, and making the artifact open source on GitHub this report and artifact aims to contribute valuable insights to the software development community.

Chapter 3. Literature Review

3.1. Introduction

This literature review shares that I have done my research and I have learned a lot from the research and will be applying the knowledge I am gaining here and now and from my past experiences all into my final year project components. This literature is also to showcase my love and interest in the field of Full Stack Development and UX/UI Engineering.

3.2. Strava

In our world where athletes are excited about keeping fit and healthy together, there are many applications out there to choose from to help us achieve our fitness goals Strava stands out as a powerful platform to achieve this because it has revolutionized the way athletes connect, compete, and track our progress together (STRAVA stories, 2023). Strava secured second position, (Figure. 1) generating approximately 7.31 million U.S dollars in total revenue from both Google Play and the Apple Store (Ceci, 2023). This report comprehensively explores Strava, offering insights into its history, purpose, key features, technological architecture, and privacy measures.

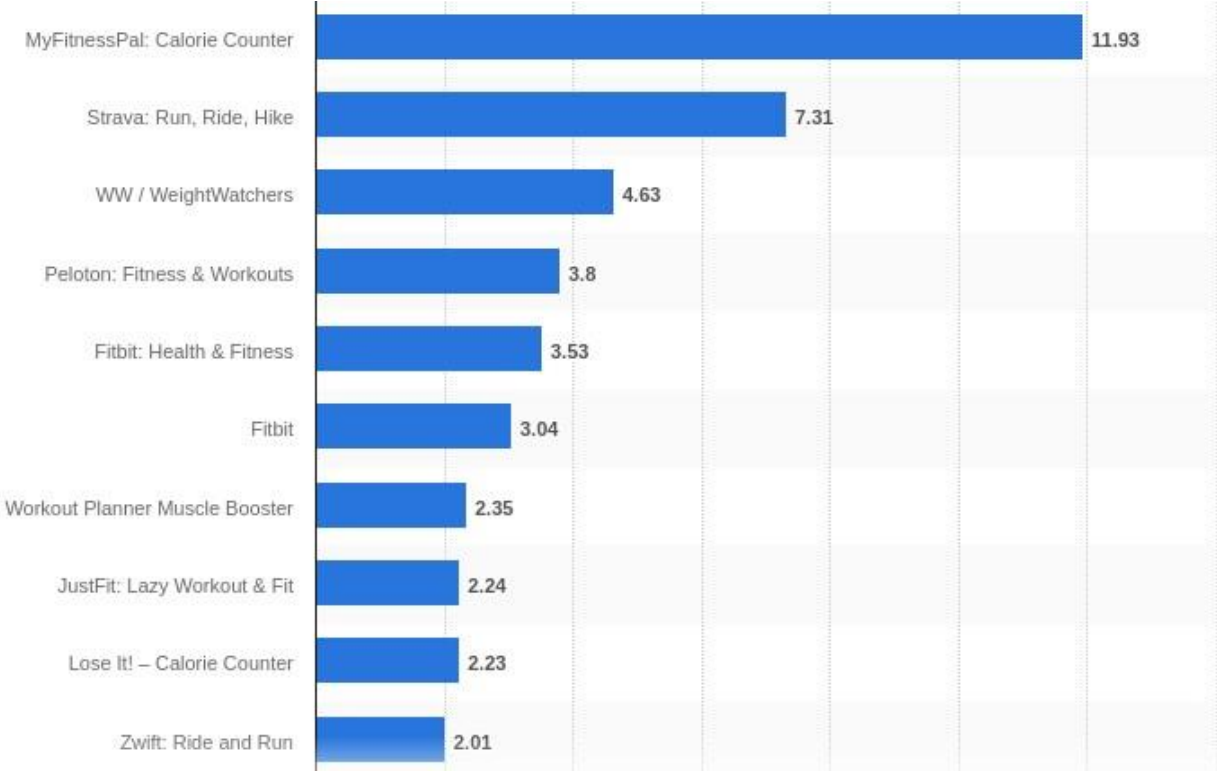


Figure 1: Leading fitness and sport apps worldwide in March 2023, by revenue (Statista 2023)

3.2.1. What is Strava?

Strava, at its core, is a social fitness network that enables athletes to track and share their activities. It unites technology and community to create an environment where users can record workouts, analyze performance, set and achieve goals, and connect with fellow enthusiasts (HALL, 2023).

One of the aspects that differentiates Strava from other social networks for athletes is its pricing model, which offers both a free and subscription version. The subscription version, also known as Strava Summit, provides additional features and benefits that enhance the user experience and performance. Table 1 shows a comparison of the primary features of the free and subscription versions of Strava, showing how they cater to different user needs and preferences (Mark, 2023).

Table 1 Strava free vs. paid features compared. The subscription is worth it for the avid rider.

Features	Free	Subscription
Activity Recording	✓	✓
Device Support	✓	✓
Social Network	✓	✓
Beacon on Phones	✓	✓
Beacon on Devices		✓
Route Planning		✓
Segment Competition		✓
Training Dashboard		✓
HR & Power Analysis		✓
Advanced Metrics		✓
Goal Setting		✓
Training Log		✓
Compare Efforts		✓
Personal Heatmaps		✓
Partner Perks		✓

~~\$7.00/mo*~~
\$5.00/mo*
* when billed annually

[Continue to Billing](#)

3.2.2. Founding and History of Strava



Figure 2: Strava founders Mark Gainey and Michael Horvath (HANSEN-GILLIS, 2020)

Founded by Mark Gainey and Michael Horvath in 2009 they were former rowing teammates at Harvard University who wanted to create a platform that would connect athletes and motivate them to improve their performance. They launched Strava as a web-based service for cyclists and later added support for runners and other sports. Strava quickly gained popularity among fitness enthusiasts. Since then, Strava has evolved significantly over the years, reaching milestones that have solidified.

Its position as a leader in the fitness tech industry. From its inception, Strava has continually refined its platform to enhance the user experience and expand its user base (Richroll, 2021). Figure 2 shows the founders of Strava.

3.2.3. Technological Architecture and Tools Used

Strava utilizes an array of components within its architecture. One of the key components of Strava's architecture is its use of cloud services, such as Amazon Web Services (AWS) and Google Cloud Platform (GCP). Strava uses AWS for its core infrastructure, such as servers, databases, storage, networking, and security. Strava also uses GCP for some of its features, such as maps, machine learning, and analytics. Using cloud services, Strava can benefit from their reliability, scalability, flexibility, and cost-effectiveness (Merchant, 2023).

3.2.4. Privacy and Data Security

Besides its technological architecture and tools used, Strava also pays attention to its privacy and data security practices. Strava states that it does not sell users' personal information for monetary value or share it with third parties that are not service providers without users' consent. Strava also provides users with various privacy controls and options to manage their visibility, location based activity, data exportation, and deletion. Moreover, Strava complies with the General Data

Protection Regulation (GDPR) in the European Union (EU) and the California Consumer Privacy Act (CCPA) in the United States (US), which grant users certain rights regarding their personal information (Strava, 2023).

3.2.5. In Summary

Strava holds a significant position in the athletic community by providing a platform that seamlessly combines technology, community, and fitness tracking. Its ability to motivate athletes, foster a sense of community, and offer personalized experiences cements its importance and relevance in the world of sports and fitness. In the next section of this literature review, we will inspect the realm of coding because building a Strava web app replica involves fundamental coding knowledge.

3.3. Coding

According to Nalea Ko (2021), coding is the process of creating instructions for computers to perform various tasks. Coding is an essential skill in the modern world, as it enables people to create and use various technologies that shape people's daily lives. In this section of the report, we will explore the basics of coding, its history, its applications, its complexities, and its opportunities, including some guidance on how to learn coding.

3.3.1. What is Coding?



Figure 3: Coding. Photograph by Chris Ried. (Unsplash, 2018)

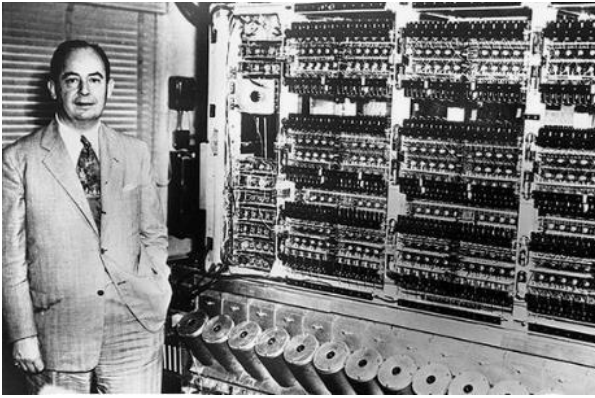
Coding is the act of writing code, which is a set of commands or instructions that a computer can understand and execute (Figure 3). Code can be written in different languages, which have different rules and syntax. A computer language is a formal system of symbols and rules that define how code should be written and structured. Some examples of computer languages are, C, Python, Java, C++, PHP, and JavaScript. (Lemonaki, 2021).

To write code, a text editor or an integrated development environment (IDE) is needed, which are software tools that allow us to write, edit, debug, and run code. There is also a need for a compiler or an interpreter, which are programs that translate code from one language to another. A compiler converts code from a high-level language (which is closer to human language) to a low-level language (which is closer to machine language), while an interpreter executes code line by line without converting it (Lemonaki, 2021).

3.3.2. Origin and Evolution of Coding

The history of coding can be traced back to the 19th century when mathematicians and engineers devised methods to automate calculations and data processing using machines. One of the

pioneers of coding was Ada Lovelace, who is considered the first computer programmer. She wrote an algorithm for the Analytical Engine, a mechanical computer designed by Charles Babbage. She also predicted that computers could perform more than just arithmetic operations (Hansel, 2018).



The first electronic computer was built in the 1940s by John von Neumann and his colleagues at the Institute for Advanced Study (Figure 4). They also developed the concept of stored programming, which means that both data and instructions are stored in the same memory unit. This enabled computers to be more flexible and efficient (Hansel, 2018).

Figure 4: John von Neumann with the stored program computer at the Institute for Advanced Study, Princeton, New Jersey, in 1945. Photograph by Getty (The Guardian, 2012)

The first programming languages were developed in the 1950s and 1960s, such as FORTRAN, LISP, COBOL, and BASIC. These languages were designed to make coding easier and more accessible for different purposes and users. They also introduced features such as variables, loops, conditionals, functions, and data structures (PARTIDA, 2022).

Since then, coding has evolved rapidly with the advancement of technology and science. New languages, paradigms, frameworks, libraries, and tools have emerged to meet the diverse and complex needs of various domains and applications. Some examples of modern coding languages are Python, Java, C#, Ruby, PHP, Swift, and JavaScript (PARTIDA, 2022).

3.3.3. Why is Coding Useful?

Coding is useful because it allows us to create and use various technologies that have significant impacts on our society and our lives. Table 1 summarizes some of the main reasons why coding is useful, along with some examples of how coding can help us in each area. The information in the table is based on the article by Siu (2022).

Table 2 Why coding is useful and how it can help us.

Solve problems	Coding can help us solve various complexities in different fields such as mathematics, science, engineering, business, education, and healthcare. For example, coding can help us analyse data, optimize processes, simulate scenarios, and design systems (Siu, 2022).
Create products	Coding can help us develop software products that provide value and functionality to users. For example, coding can help us create websites, mobile apps, games, and software applications (Siu, 2022).
Express creativity	Coding can help us express our ideas and imagination in innovative ways. For example, coding can help us create digital art, music, and animations (Siu, 2022).
Communicate	Coding can help us communicate with other people and machines using various methods and protocols. For example, coding can help us send emails, chat online, and control robots (Siu, 2022).
Learn	Coding can help us learn new skills and concepts in various disciplines. For example, coding can help us learn mathematics, physics, and biology (Siu, 2022).

Note. Adapted from “Why is coding important? 10 reasons to learn coding in 2022” by Siu, 2022.

3.3.4. Understanding Coding vs. Programming

Coding and programming are two terms that are often used interchangeably in the software development industry. However, they have different meanings and implications. Coding is the act of writing code, which is a set of instructions that a computer can understand and execute. Programming is the process of creating software using code and other tools and techniques. Programming involves more than just coding; it also includes planning, designing, testing, debugging, documenting, and maintaining software. Programming requires logical thinking, problem-solving skills, creativity, and attention to detail (Nyakundi, 2021).

Table 3 The Coding Process.

Analyse	In the analysis step, the coder understands the problem or the goal that they want to achieve with code.
Design	In the design step, the coder plans how they will write their code, using algorithms (step-by-step procedures) and data structures (ways of organizing data).
Implement	In the implementation step, the coder writes their code using a programming language and a text editor or an integrated development environment (IDE)
Test	In the test step, the coder runs the code and checks if it works as expected. If not, they find and fix errors or bugs (Programiz PRO Resources, 2021)

Note. Adapted from *“What is coding? How to code? A beginner’s guide”* by Programiz PRO Resources (2021).

Table 4 The Programming Process.

Software design	The programmer defines the overall structure and architecture of the software, such as the modules, components, interfaces, and data flow.
Software testing	The programmer verifies that the software meets the requirements and specifications and that it is free of defects and errors. Testing can be done at different levels, such as unit testing, integration testing, system testing, and user acceptance testing.
Software deployment	The programmer delivers the software to the end-users or customers, either through installation, distribution, or cloud services.
Software documentation	The programmer creates and maintains documents that describe the software, such as user manuals, technical manuals, and code comments.
Software maintenance	The programmer updates and modifies the software to fix bugs, improve performance, add features, or adapt to changing needs and environments (Nyakundi, 2021).

Note. Adapted from “Why is coding important? 10 reasons to learn coding in 2022” by Nyakundi (2021).

Coding and programming are both essential skills for software development, but they are different. Coding is a subset of programming, and programming is a broader and more complex process that involves coding and other activities. Coding requires basic programming knowledge, while programming requires advanced programming concepts and tools (InterviewBit, 2022)

3.3.5. In Summary

Coding is an essential skill in our technology-driven world. It has a rich history and a broad range of applications, from problem-solving to creative expression. Distinguishing between coding and programming is important, as both play critical roles in software development. Embracing coding is an opportunity to unlock innovation and creativity in the digital age. In the next section of this literature review, we will explore React, a JavaScript library that will be used in the building of the Strava web app replica.

3.4. React

React is a JavaScript library for building user interfaces for web applications. React was created by Facebook in 2011 as a way to improve the performance and maintainability of their web products. React has become one of the most popular and widely used front-end frameworks in the web development community (Bhatnagar, 2023).

3.4.1. Features and Advantages of React

```
import React, { useState } from "react";

Comment Code
function Counter() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}

export default Counter;
```

Figure 5: Example of a React code that uses JSX, hooks, and components to create a simple counter app. Coded by Fortune Ndlovu.

React has several features that make it a suitable choice for web application development, such as:

- The Virtual Document Object Model (VDOM), which is a representation of the actual Document Object Model (DOM) in memory, allows React to update only the parts of the User Interface (UI) that have changed, resulting in faster and smoother rendering (GeeksForGeeks, 2023).
- It uses a component-based architecture, which means that UIs are composed of reusable and independent pieces of code called components. Components can have their state, props, and lifecycle methods, and can also render other components as their children. This makes it easy to create complex and dynamic User Interfaces from simple components (GeeksForGeeks, 2023).
- It uses JavaScript XML (JSX), which is a syntax extension that allows writing HTML-like markup within JavaScript, a programming language for building interactive web pages (Ozanich, 2022). JSX makes it convenient to write and read UI code and also enables some features such as props validation and code completion (WebandCrafts, 2021)
- It supports hooks, which are functions that let you use state and other React features without writing a class component. Hooks make it easier to reuse stateful logic and manage the component lifecycle (Ikechukwu, 2022).

Figure 5 above shows an example of React features. What the code is doing is displaying a paragraph that says “You clicked 0 times” and a button that says “Click me” all displayed on a browser document and whenever the button is clicked the number 0 increases incrementally.

Under the hood, we have

- **JSX:** A syntax extension that allows writing HTML-like elements inside JavaScript code. For example, `<div>Counter App</div>` is a JSX element that renders a div element with the text “Counter App”.
- **useState:** A React hook that allows creating and updating a state variable. For example, `const [counter, setCounter] = useState(0);` creates a state variable named counter with an initial value of 0 and a function named setCounter to update it.
- **Event handlers:** Functions that are executed when a user interacts with an element, such as clicking a button. For example, `onClick={handleClick1}` assigns the function handleClick1 to the onClick event of the button, which increments the counter state by 1 when the button is clicked.

3.4.2. Component-Based Architecture

A component in React is a reusable piece of code that can render HTML elements, handle user interactions, and manage state. A component can be either a function or a class, but it must always return JSX markup, which is a syntax extension that looks like HTML but is JavaScript.

React's component-based architecture simplifies development and maintenance by allowing developers to break down complex UIs into smaller and independent pieces. Each component has its logic, data, and presentation, which makes it easier to understand, test, and reuse. Components can also communicate with each other through props and states, which are ways of passing data and events between components. By using components, developers can create modular and scalable applications that are easier to debug and maintain (React Dev, 2023).

React does not provide everything you need to build a complete web application. That's why there is a rich ecosystem of libraries and tools that complement React and help us with various aspects of web development. Here are some of the most common ones:

- **Redux:** A library for managing the state of a developer's application. Redux helps with organizing data and logic predictably and consistently and makes it easier to debug and test code. Redux works well with React (Redux, 2023).
- **React Router:** A library for handling navigation and routing in a React app. React Router allows developers to define the URL structure of their app, and dynamically render different components based on the current location. React Router also supports features like nested routes, redirects, and transitions (React Router, 2023).
- **Jest:** A testing framework for JavaScript and React. Jest makes it easy to write and run tests for a developer's code and provides features like mocking, snapshots, and code coverage. Jest is also fast and reliable, thanks to its parallel test execution and intelligent caching system (Jest, 2023).
- **Create React App:** A tool for setting up a modern React project with minimal configuration. Create React App gives developers a ready-to-use environment with features like hot reloading, code splitting, linting, and testing. Create React App also lets developers customize their projects with various options and scripts (Create React App, 2023).

3.4.3. Why React for this Project?

In selecting the appropriate technology stack for the development of the Strava web app Replica, various factors were considered, ultimately leading to the choice of React as the primary frontend library. React was deemed an ideal fit for this project due to several compelling reasons.

3.4.4. Factors Influencing the Choice of React

4.4.4.1. Modularity and Reusability

React's component-based architecture facilitates the creation of modular and reusable UI components (React Dev, 2023). This aligns perfectly with developing a complex web application like Strava, where different sections of the application can be broken down into self-contained components. Each component can be managed and tested independently, enhancing maintainability.

3.4.4.2. Efficiency in Development

React's declarative and efficient programming paradigm significantly boosts development speed. The ability to describe the UI based on state and data allows for streamlined development without micromanaging DOM manipulation (GeeksForGeeks, 2023). This characteristic is crucial for a project like Strava, where providing a responsive and dynamic user interface is of paramount importance.

3.4.4.3. Enhanced User Experience (UX)

React's virtual DOM and efficient rendering mechanism contribute to a seamless and instant user experience. Given that Strava is a platform where users expect real-time updates and interactions, React's performance optimizations play a vital role in ensuring a smooth and engaging UX (React Dev, 2023).

3.4.4.4. Previous Experience and Familiarity

Having prior experience with React was a significant influence on choosing it for this project. React has been a preferred framework due to its complexity yet ease of learning, quick development cycle, and vast ecosystem of libraries and tools. Familiarity with React allows for rapid prototyping, efficient debugging, and effective utilization of best practices, resulting in a well-structured and maintainable codebase.

3.4.5. In Summary

React was chosen for the Strava web app replica due to its modularity, efficiency, and excellent user experience. Its component-based structure allowed for modular development, while its virtual DOM and efficient rendering were crucial for real-time updates. Familiarity and positive past experiences with React, along with its ease of use and strong community support, reinforced the decision. React's capabilities aligned well with the goals of the project, making it an ideal choice for creating a responsive and maintainable web application. The next section of the report will talk about the complexities in replicating Strava in React.

3.5. Complexities in Replicating Strava in React

Replicating a comprehensive and feature-rich platform like Strava within a React-based web application introduces a multitude of complexities, each posing unique complexities to the project's implementation. These complexities span various facets, from user interface design to data integration, real-time updates, user interactions, and security measures. Understanding why these complexities make the implementation a learning opportunity is essential to grasp the depth of the project. Here, we delve into the specifics of each complexity, elucidating their impact on the project's intricacy.

3.5.1. Data Integration and Synchronization

Strava's core functionality hinges on seamless data integration with its backend systems and external services, such as GPS tracking and heart rate monitors (Mason, 2022). The complexity lies in harmonizing diverse data sources and ensuring they synchronize cohesively. The need to maintain data accuracy and consistency, despite variations in data formats and sources, adds a layer of complexity to the project.

3.5.2. Real-Time Updates

Strava's ability to deliver real-time updates on activities and social interactions is a hallmark feature (Mason, 2022). Implementing this real-time functionality in React presents a complex task. It requires setting up mechanisms for immediate notifications, handling data streams efficiently, and ensuring timely user engagement. The need for responsive, real-time interactions introduces intricacies into the application's architecture and user experience.

3.5.3. Activity Analysis and Visualization

Strava offers in-depth activity analysis, including maps, elevation profiles, and performance metrics (Mason, 2022). Replicating these features while maintaining user-friendliness and optimal performance presents a multifaceted complexity. It demands the implementation of sophisticated visualization tools and the efficient processing of vast datasets, all while preserving a responsive and intuitive user interface.

3.5.4. User Authentication and Security

The criticality of user data security cannot be overstated. Strava deals with sensitive information, including location data and personal profiles (Mason, 2022). Implementing robust user authentication and security measures is indispensable but intricate. Safeguarding user data from unauthorized access, and data breaches, and ensuring compliance with privacy regulations amplifies the complexity of the project.

3.5.5. Social Features and Interactions

Strava's social features, such as user connections, activity sharing, and user interactions like comments and likes, are integral to its community-building aspect (Mason, 2022). Replicating these social functionalities requires sophisticated user management and real-time updates for seamless interaction. The complexity lies in creating a user-friendly, real-time social ecosystem within the application, complete with notifications and activity feeds.

3.5.6. Handling External APIs and Services

Strava relies on external APIs and services for mapping, heart rate monitoring, and other functionalities (Sawh, 2020). Integrating these external services into the application introduces intricacies due to the diversity of data sources and dependencies. The project must manage external API calls, data processing, and ensure seamless interactions with these services, all while maintaining stability and performance.

3.5.7. In Summary

These complexities underscore the depth and complexity of the Strava web app replica. Successfully replicating the Strava web app in React is a testament to my proficiency in front-end development and UX/UI engineering, as it necessitates meticulous planning, thorough testing, and innovative solutions. Each of these complexities represents an opportunity for skill development, reinforcing the importance of my final year project in honing my expertise in the field. The next section of the report will talk about addressing complexities in replicating Strava in React.

3.6. Addressing Complexities in Replicating Strava in React

To tackle the complexities inherent in replicating Strava within a React-based web application, a strategic and multifaceted approach is essential. Leveraging a combination of well-chosen tools, frameworks, and development strategies will play a pivotal role in overcoming these complexities. Here's how each complexity can be addressed, along with the tools and techniques that can be employed:

3.6.1. Data Integration and Synchronization

Tool: Fetch API

Strategy: Utilize the Fetch API for efficient data retrieval and synchronization. The Fetch API allows the client application to make precise requests to the Strava API, fetching only the specific data required. This approach optimizes data transfer and ensures seamless synchronization (Ijaz, 2023).

3.6.2. Real-Time Updates

Tool: Fetch API with Polling

Strategy: Implement real-time updates using the Fetch API with polling mechanisms. Periodically polling the Strava API for updates enables the application to receive notifications and new data in near real-time, ensuring timely user engagement (Niedringhaus, 2020).

3.6.3. Activity Analysis and Visualization

Tool: Chart.js

Strategy: Utilize a charting library like Chart.js to streamline the creation of interactive and visually appealing activity analysis components. Chart.js offers a variety of chart types and customization options, reducing the coding effort required. Additionally, consider Mapbox for mapping and visualization of routes and elevation profiles, and custom solutions for handling performance metrics (Akbulut, 2021).

3.6.4. User Authentication and Security

Tool: Firebase Authentication

Strategy: Integrate Firebase Authentication for user management and authentication. Firebase offers robust security measures, including secure token-based authentication and OAuth support, ensuring user data protection and compliance with privacy regulations (Faruq, 2022).

3.6.5. Social Features and Interactions

Tool: Real-time database such as Firebase's Database named Firestore Database

Strategy: Leverage a real-time database to facilitate social features and interactions. Firestore Database allows for seamless real-time updates of user connections, activity sharing, comments, and likes (Gobo, 2020).

3.6.6. Handling External APIs and Services

Tool: Fetch API

Strategy: Implement the Fetch API to manage interactions with external APIs and services, particularly for fetching data like heart rate and other relevant information. The Fetch API simplifies HTTP requests and is a suitable choice when the integration scope primarily involves data retrieval. This approach offers a lightweight and efficient solution for handling these specific interactions (Ijaz, 2023).

3.6.7. In Summary

Addressing these complexities in replicating the Strava web app in a React-based web app requires a strategic approach:

- Data Integration and Synchronization: Utilize the Fetch API for efficient data handling.
- Real-Time Updates: Implement Fetch API with polling for real-time updates.
- Activity Analysis and Visualization: Simplify with Chart.js and consider Mapbox for mapping.
- User Authentication and Security: Integrate Firebase Authentication for data protection.
- Social Features and Interactions: Utilize a real-time database like Firestore Database.
- Handling External APIs and Services: Efficiently manage data with the Fetch API.

These tools and strategies ensure successful navigation of these complexities, leading to the realization of the Strava web app replica in React. The next section of the report will talk about the coding design.

3.7. Coding Design

In this section, we will delve into the coding design for replicating the Strava web app using React, considering the knowledge and insights gained from the literature review. We will discuss the overall approach and the key components that will come together to create the Strava web app replica.

3.7.1. Technology Stack

For my project, I have carefully selected a technology stack that aligns with my goals. React, a widely acclaimed JavaScript library for building user interfaces, will be the primary framework. React was chosen for its modularity, efficiency, and exceptional user experience, as detailed in the literature review. It provides the flexibility and robustness required for replicating the Strava web app. The component-based architecture of React will facilitate the creation of modular and reusable UI components, offering maintainability (React Dev, 2023).

3.7.2. Modular Component-Based Architecture

The core of my coding design revolves around a modular and component-based architecture using React. Each section of the Strava web app replica will be broken down into self-contained components, reflecting the components in the real Strava application. This modular approach will facilitate easy maintenance as new features are added. The components will include the header, footer, user profile, activity list, activity details, and social interaction components (Bhatnagar, 2023).

3.7.3. Data Integration and Synchronization

To provide users with up-to-date information, I will employ the Fetch API for efficient data integration and synchronization with Strava's backend systems. The Fetch API will enable my application to make precise requests to the Strava API, fetching only the specific data required. This approach optimizes data transfer and ensures seamless synchronization, aligning with the real-time updates offered by Strava (Ijaz, 2023).

3.7.4. Real-Time Updates

Real-time updates are essential for replicating Strava's user experience. To achieve this, I will implement real-time updates using the Fetch API with polling mechanisms. Periodically polling the Strava API for updates will allow my application to receive notifications and new data in near real-time. Users will be promptly informed of any new activities and social interactions (Niedringhaus, 2020).

3.7.5. Activity Analysis and Visualization

For activity analysis and visualization, I will make use of libraries like Chart.js and potentially Mapbox. Chart.js will be instrumental in creating interactive and visually appealing activity analysis components, such as distance, speed, and elevation data charts. Mapbox may be used to provide detailed maps and elevation profiles. These visualization tools are essential for replicating Strava's in-depth activity analysis features while ensuring a user-friendly and intuitive interface (Akbulut, 2021).

3.7.6. User Authentication and Security

User authentication and data security are of paramount importance. For this, I will integrate Firebase Authentication into my project. Firebase offers robust security measures, including secure token-based authentication and OAuth support. This will ensure that user data is protected from unauthorized access and data breaches. Additionally, compliance with privacy regulations, such as GDPR and CCPA, will be upheld (Faruq, 2022).

3.7.7. Social Features and Interactions

To replicate Strava's social aspects, I will employ a real-time database, specifically Firestore Database. This database will facilitate user connections, activity sharing, comments, and likes in a real-time environment. Users will be able to connect, share their activities, and interact seamlessly and responsively. Social notifications and activity feeds will ensure a dynamic social ecosystem (Faruq, 2022).

3.7.8. Handling External APIs and Services

Strava relies on external APIs and services for features like heart rate monitoring. To handle these interactions, I will use the Fetch API. This lightweight and efficient solution will manage interactions with external services, ensuring accurate and reliable data retrieval. The compatibility with various heart rate monitoring devices will be maintained (Niedringhaus, 2020).

3.7.9. In Summary

The comprehensive coding design for replicating the Strava web app in React encompasses a technology stack focused on React's modularity, efficiency, and user experience. The modular and component-based architecture will enable scalability and maintainability. The integration of the Fetch API, real-time updates, visualization libraries, and Firebase services addresses the complexities identified in the literature review. With this coding design, I am well-equipped to develop a successful Strava web app replica that aligns with the goals of my final-year project.

3.8. Literature Review In Summary

This literature review has provided a deep dive into the intersection of technology and fitness, with Strava as the focal point. We've explored Strava's history, core features, and technological architecture, setting the stage for the coding design of my final-year project.

The choice of React as my primary framework aligns with my project's goals. React's component-based architecture, real-time capabilities, and robust user interface design make it the ideal choice.

The coding design plan outlines my strategic approach, addressing complexities like data integration, real-time updates, activity analysis, user security, social features, and external APIs.

This literature review reflects my commitment to a well-researched and meticulously planned project, underpinned by a genuine passion for front-end development and UX/UI engineering. It sets the foundation for the successful execution of my Strava web app replica project. The next section of the report begins a new chapter about the project complexities process.

Chapter 4. Project Complexities Process

4.1. Basic UX/UI

The Strava web application (web app) replica project was initiated to create a user-friendly and efficient web app that replicates the core features of the Strava web app. Before implementing the core functionality of the web app, basic user experience and user interface (UX/UI) needed to be built, therefore I, the developer, focused on implementing the UX/UI. I began with the home page because this viewport would be the first visual the user would initially see.

When implementing the home page viewport I began with the header, followed by the footer. This created a solid foundation for me to build within both components, moreover, this process provided me with a top-view perspective of the inner components outlined within.

4.1.1. Header

A header in a web app plays a crucial role in the user's experience because it serves as a consistent navigation tool that is always available to the user as they navigate through the web app. Therefore, fundamentally it needs to be accessible meaning it should be easy for the user to locate and use, and it should be visually consistent meaning offering a consistent feel and look throughout the web app to avoid confusing the user.

4.1.1.1. Design and Implementation

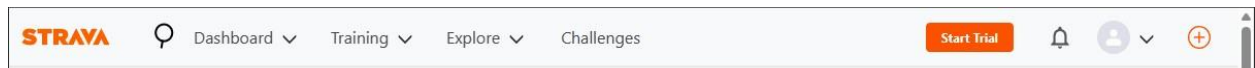


Figure 5: Header. Screenshot by Fortune Ndlovu (2024)

Figure 5 shows a screenshot of the coded header for the Strava web app replica. The header is divided into two major sections, on the left the header consists of links that are more closely related, and likewise, on the right side of the header, both sections evaluate to being the entire header. On the left, there is a PNG image for the web app logo wrapped around an anchor tag for navigating back to the home page, followed by a button with a transparent background with a search icon as its value, followed by 4 navigation links and 3/4 of these navigation links with

dropdown icons indicating dropdown executions upon hover interactions. Moreover, on the right side of the header is a button that consists of text as its value and another button that consists of an icon as its value with a transparent background, followed by a link with 2 values the first being an icon and the second being a dropdown icon, lastly, we have another button.

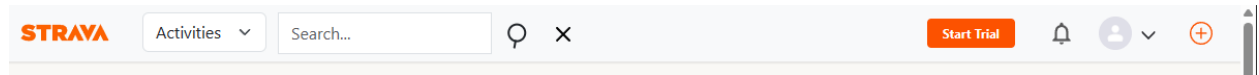


Figure 6: Header Search box open. Screenshot by Fortune Ndlovu (2024)

Figure 6 also shows a screenshot of the coded header for the Strava web app replica. When the user clicks the button with the search icon as its value two things happen. Firstly, the prior navigation links are removed from the navigation within this viewport and secondly, a search component is rendered. The search component returns a form with a select dropdown, a search input, a button with a search icon as its value for search, and lastly a button with a close icon for closing the search visual representation.

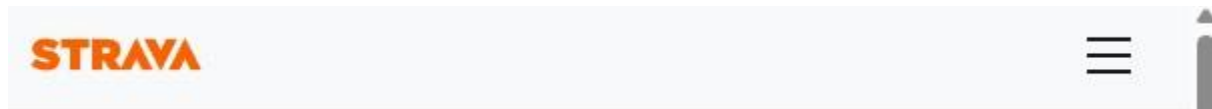


Figure 7: Header Resize Hamburger display. Screenshot by Fortune Ndlovu (2024)

Figure 7 also shows a screenshot of the coded header for the Strava web app replica. The Figure 7 screenshot displays how the navigation looks when the screen is resized to smaller screens, only the web app logo and hamburger are present and the other navigation links are removed.

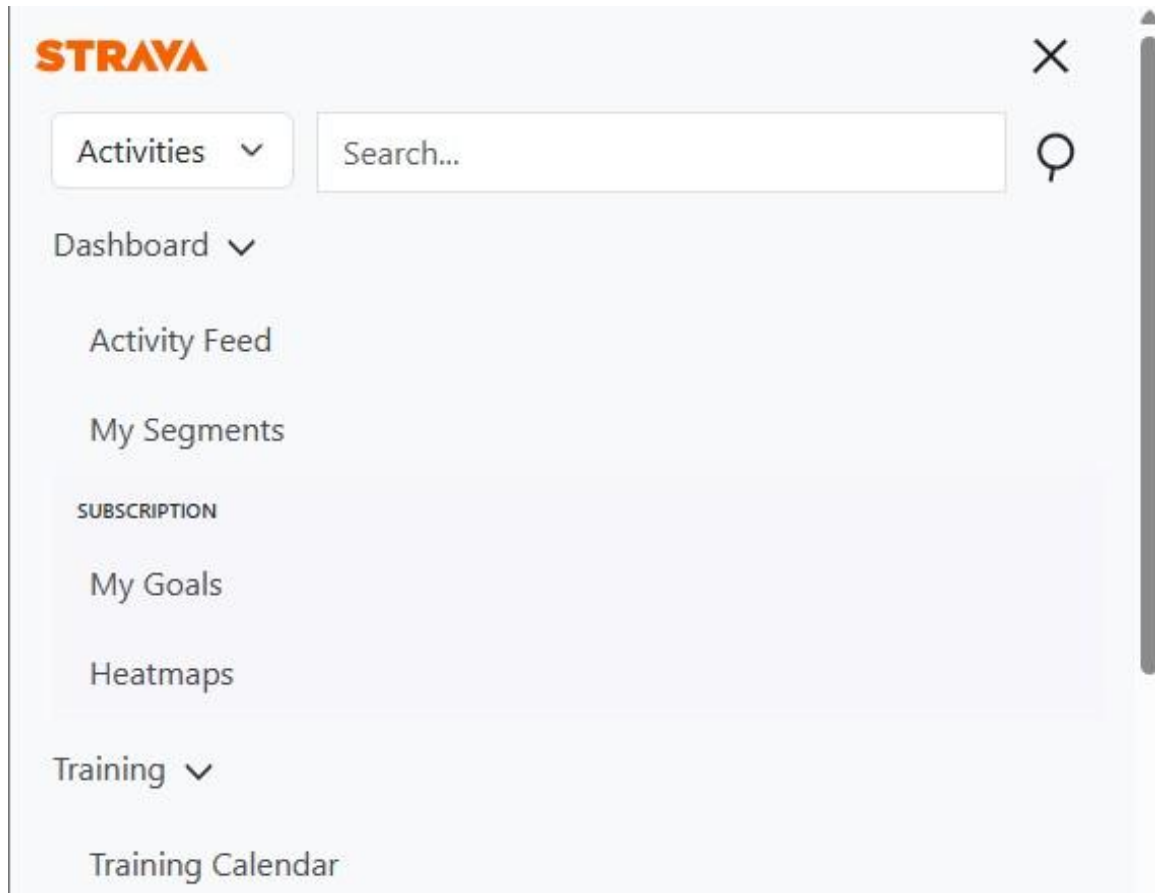


Figure 8: Header Resize Hamburger dropdown display. Screenshot by Fortune Ndlovu (2024)

Figure 8 also shows a screenshot of the coded header for the Strava web app replica. The Figure 8 screenshot displays how the navigation looks when the user clicks the hamburger button on the top right of the screen. When this interaction occurs, an event takes place causing a component with all the navigation links to render. Three major visuals can be interacted with and that are worth noting. Firstly, the search component is re-rendered again in this viewport, secondly, the navigation links are displayed vertically for smaller screen sizes and not horizontally for larger screen sizes.

The navigation links are also open by default in this viewport and can be toggled true or false (open or close) via clicking the dropdown icon next to the parent navigation item. Lastly, the hamburger button has been animated to an X icon because its dropdown navigation is now open.

4.1.1.2. Code Overview

The header coding implementation is interesting because logic has been crafted to display certain elements and components at various user interactions. At a high level, the code structure for the header defines a React functional component called Header, which represents the header/navigation bar for the web app. The header component returns JSX. JSX (JavaScript XML) is a syntax extension for JavaScript that allows developers to write HTML-like code within JavaScript. The navigation is structured using the React Bootstraps Navbar component. It contains a logo linking to the homepage, a responsive toggle button for small screens and a collapsed menu for navigation items.

The component also uses several useState hooks to manage the state of various elements such as dropdown menus (`showDashboardItems`, `showTrainingItems`, `showExploreItems`, `showUserAvatar`, `showUploadButton`), search bar visibility (`showSearch`), and hamburger menu visibility (`isHamburger` and `isMenuOpen`). The reason the component uses several useState hooks is to manage different pieces of state that affect how the header behaves and renders based on user interactions and screen size. Each piece of state corresponds to a specific aspect of the header headers functionality.

4.1.1.3. Code Breakdown

```
<Navbar.Collapse id="basic-navbar-nav">
  <Nav className="me-auto">
    {/* When we are on both small and large view-ports we want to show the search component. */}
    {(showSearch || isHamburger) && (
      <SearchBar onCancel={() => setShowSearch(false)} />
    )}

    {/* When the hamburger and the search component are not showing, then show the search icon. */}
    {!isHamburger && !showSearch && (
      <Nav.Link
        title="Search"
        href="#search"
        onClick={() => setShowSearch(true)}
      >
        <FiSearch className="open-search-icon" />
      </Nav.Link>
    )}
  </Nav>
</Navbar.Collapse>
```

Figure 9: Conditional Rendering Logical OR inside Logical AND (&&) Expression. Screenshot by Fortune Ndlovu (2024)

At the top half of Figure 9, the screencap shows the code used to display the search component when the search link is clicked. I am using a logical AND (&&) expression to conditionally render the SearchBar component. This means if either showSearch or isHamburger is true the SearchBar component renders. The search bar is a component that has its own functionality. I imported it into the main header component for clean code maintenance reasons. The reason I am using logical OR (||) inside Logical AND (&&) is because I wanted to display the searchBar component whether the search icon link has been clicked in larger screen or whether the hamburger is true this would mean we are at smaller screen sizes and we also want to show the searchBar component.

At the bottom half of Figure 9 the screencap shows another logical AND (&&) expression that only shows the search icon link when the isHamburger and showSearch state is not true. This is because in this instance we are in larger screens, and we only want to show the search icon in larger screens so the user can click it to show the search bar. This interaction is helped by the onClick handler on the link that is passed an anonymous function that sets setShowSearch to true. Notice I am using logical AND (&&) inside another logical (&&) expression because both state variables need to not be true for this execution.


```

{({isHamburger || !showSearch) && (
  <React.Fragment>
    <NavDropdown
      id="dashboardDropdown"
      onMouseEnter={(e) =>
        handleDropdownHover(() => setShowDashboardItems(true), e)
      }
      onMouseLeave={(e) =>
        handleDropdownHover(() => setShowDashboardItems(false), e)
      }
      show={showDashboardItems}
      title={
        <div className="d-flex align-items-center">
          <Link className="navDropdownHeading" to="/home/">
            Dashboard
          </Link>
          <button
            title="Expand dashboard menu"
            onClick={handleDashboardDropdownToggle}
            onKeyDown={(event) =>
              handleDropdownIconKeyPress(
                event,
                handleDashboardDropdownToggle
              )
            }
            className="icon-button"
            aria-label="Toggle Dashboard Dropdown"
            tabIndex={0}
          >

```

Figure 10: Initially Setting Dropdown Menus to true based on isHamburger. Screenshot by Fortune Ndlovu (2024)

Figure 10 explains how the dropdown menus are initially open when the screen size is small as defined by the isHamburger, this is because the initial state of isHamburger is less than or equal to 992 pixels changing the initial state to true. I specified this under my header component in the useState hook for example:

```

`const [isHamburger, setIsHamburger] = useState(window.innerWidth <= 992);`

```

When the dropdown menus are initially rendered based on the state of isHamburger using the show prop and when the isHamburger is true (i.e, on smaller screens), the dropdown menus are set to be shown. Additionally, I coded a useEffect hook with a handleResize function that listens for window resize events shown in Figure 11 below. When the window is resized, handleResize is called and updates the state of isHamburger based on the new window width. This ensures that the dropdown menus react to changes in screen size dynamically.

```

// Responsible for updating the isHamburger state based on the window width.
const handleResize = () => {
  setIsHamburger(window.innerWidth <= 992);
};

// Listen for window resize events.
useEffect(() => {
  // if the hamburger is true that means we can show the navigation dropdowns.
  setShowDashboardItems(isHamburger);
  setTrainingItems(isHamburger);
  setExploreItems(isHamburger);
  setUserAvatar(isHamburger);
  setUploadButton(isHamburger);

  // When the window is resized the handleResize func will be called.
  window.addEventListener("resize", handleResize);
  return () => {
    window.removeEventListener("resize", handleResize);
  };

  // The useEffect will be re-executed whenever the isHamburger changes.
}, [isHamburger]);

```

Figure 11: useEffect hook for listening to window resizes. Screenshot by Fortune Ndlovu (2024)

Figure 10 also highlights event handlers, when the user hovers and clicks dropdown menu items events take place. For when the user hovers, I coded an onMouseEnter and onMouseLeave event handler which both are passed an anonymous function capturing the event and setting the state of the dropdown items to either true or false based on mouse enter/leave. When the user clicks the toggle button we pass an onClick event handler of the event that calls a function with the event passed as an argument to a parameter setting the state value to the previous value shown in Figure 12, this means whatever the value is, set it to the previous value, for example, if its false being closed set it to true being open and if it's true being open set it to false being close.

```

// Setting the dropdowns to be toggled based on their previous state.
const toggleDropdownVisibility = (showStateSetter) => {
  return () => {
    showStateSetter((prevShowState) => !prevShowState);
  };
};

// Whether the navigation items I mouse clicked or tabbed toggle their dropdown menus.
const handleDashboardDropdownToggle = toggleDropdownVisibility(setShowDashboardItems);
const handleTrainingDropdownToggle = toggleDropdownVisibility(setTrainingItems);

```

Figure 12: Toggling State to its previous value. Screenshot by Fortune Ndlovu (2024)

4.1.1.4. Complexities and Solutions

The first complexity I faced was implementing the logic for displaying the hamburger menu on smaller screens while rendering the SearchBar component, which was also displayed on larger screens upon user interactions. This meant that the condition used whenever the user clicked the search icon to render the search bar, which is true, I had to also attach the value of the hamburger to also true. This meant that whenever the search icon is clicked shows the search bar or whenever the hamburger is true also shows the search bar, this was complex because the user interactions were not taking place on the same event, and I wanted to reuse my code for readability and code maintenance purposes.

The solution for this complexity was the usage of useState variables; they allowed me to control elements of code, whenever certain events took place, I could capture the value within a conditional rendering logical expression and pass the evaluated value to the next conditional rendering logical expression for evaluation. This meant instead of coding large blocks of code I was reassuring state variables and if I needed to code a block of code, I would separate it into a component and import it to the main component as I had done with the search bar.

The second complexity I faced was having the dropdown menu icons open initially on smaller screens and closed on larger screens. Including the ability to hover over the navigation items on larger screens to display the dropdowns and be able to toggle the dropdowns on smaller screens while including keyboard functionality. This was complex because as a developer we cannot add hover events and click events on one element. It's not common practice, it is best to have an element either be hoverable or clickable for accessibility reasons.

The solution for this was to initially have the navigation elements be hoverable meaning the user could mouse over the element on larger screens and the navigation dropdown elements would display. I used onMouseEnter and onMouseLeave for this. I also converted the dropdown icon into a transparent button meaning I could pass an onClick handler to it so the user could also toggle the dropdown menu in smaller screen sizes. This is a hack because in large screen sizes, the user initially mouses over the navigation elements to display the dropdown menus while the dropdown icon can also be clicked to toggle the dropdown menus. For the safekeeping of this hack, I coded an onKeyDown event on the navigation items to also display the dropdown just in case the user is using the keyboard only ensuring they are not going over both the navigation item and dropdown item.

Lastly, creating the hamburger animation was complex because initially, React Bootstrap offered a hamburger with its boilerplate navigation component, but it is not animated. The solution for this was I had to override the default hamburger using CSS and apply two-dimensional transformation through the transform property, which consists of a list of transform functions such as rotate and translate, therefore tweaking the degrees angle on the rotate and pixels on the translate functions was complex as the initial values are not priorly defined.

4.1.2. Footer and Main Dashboard Components UX/UI

The footer implementation was more straightforward than the header because there was a lack of user interactions and more of displaying static navigation links, as shown in Figure 13. However, it is worth noting the footers UX/UI because it is part of the overall UX/UI of the web app homepage.

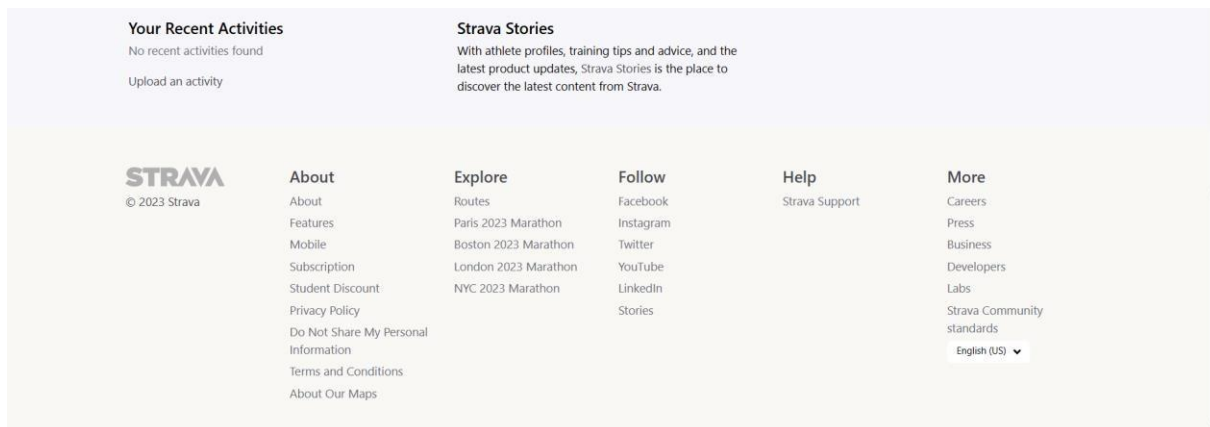


Figure 13: Footer UX/UI. Screenshot by Fortune Ndlovu (2024)

At this moment of the Strava web app replica, the header and footer have been implemented and it was now appropriate to code the main content within the home page. The home page is divided into three components that are essentially sidebars.

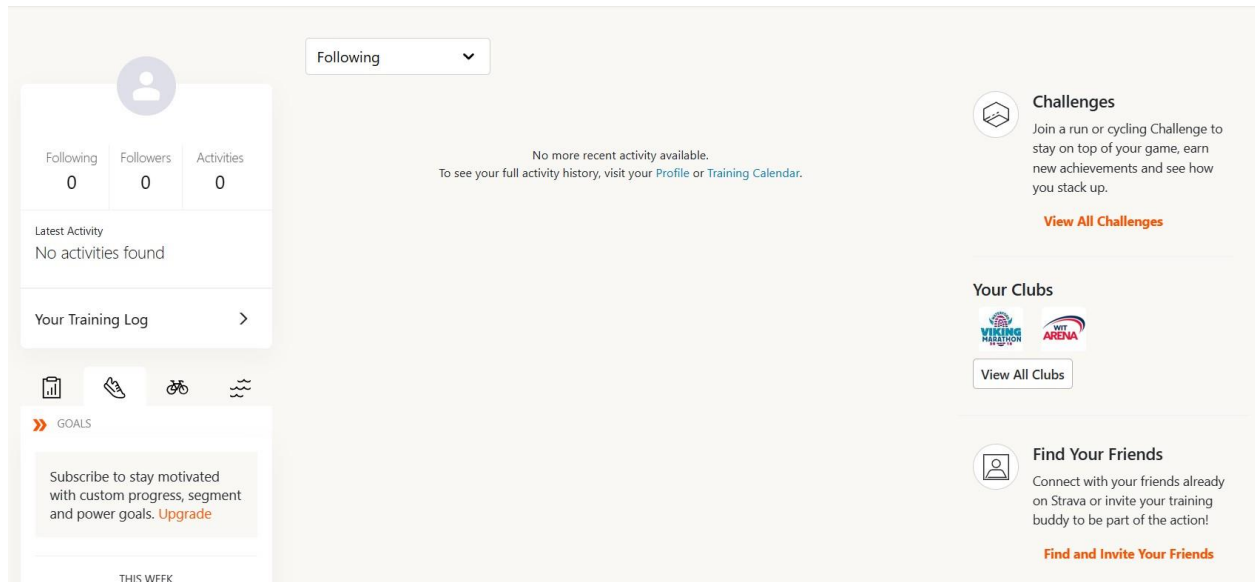


Figure 14: Home Page UX/UI. Screenshot by Fortune Ndlovu (2024)

Figure 14 shows the coded home page of the Strava web app replica. The home page has sidebars on the left and right of the screen leaving space in the middle for a component that will display the generated activity in the form of a card by the user. This structure is made possible by making use of React Bootstrap. Effectively the entire page is wrapped around a fluid container this means it is responsive to viewport sizes by default and is flexible for additional modifications. Within the container are essentially three columns. The first column starts on the left and the second is in the middle and the last column is on the far right. The middle column has been provided with double size of the side columns in width. Moreover, within each column lives a component that is used to display the visual representations.

4.1.2.1. Design and Implementation

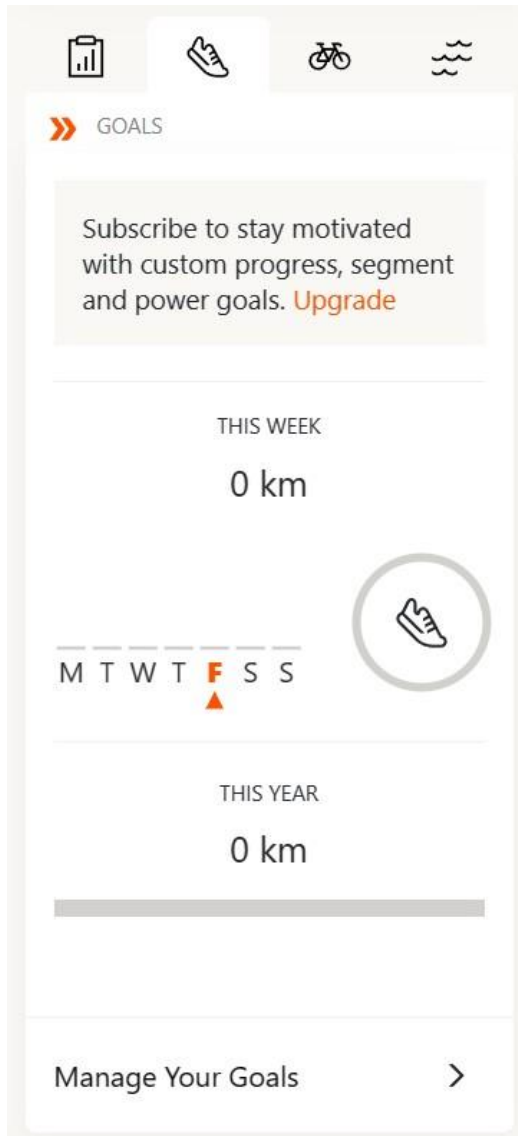


Figure 15: Home Page Left Sidebar UX/UI. Screenshot by Fortune Ndlovu (2024)

I also implemented functionality to get the current date, apply a polygon and fill in the text that represents the current day.

For the design implementation of the home page dashboard, in this section of the report, I will focus on the left sidebar because it was the most complex and interesting to implement. Figure 14 shows the top half of the left sidebar and Figure 15 shows the bottom half of the left sidebar. The top half displays a card withholding statistics that are relevant to the current user, such as following, followers, activities, and latest activity; in this instance the values are static and will later be dynamically populated.

The bottom half of the left sidebar is also a card, and this card has 4 navigation tabs. Tabs is a higher-level component for creating Nav matched with a set of TabPanels.

Initially, when the user navigates through the tabs a new tab with its content renders. This was not what I wanted because the 4 tabs contained similar content and I did not want to code the same content for each tab 4 times, because of repetition. Therefore, I figured out a solution, where only the content that changes between the tabs is updated with its unique information, upon navigation. In this instance, only the large circled SVG icons change, indicating we are in the correct tab as intended.

4.1.2.2. Code Overview

```
LIVE EDITOR

import Button from 'react-bootstrap/Button';
import Card from 'react-bootstrap/Card';
import AthleteStatsTabs from './LeftDashboardAthleteSidebarComponents/AthleteStatsTabs';

function BasicExample() {
  return (
    <div>
      <Card style={{ width: '18rem' }}>
        <Card.Img variant="top" src="holder.js/100px180" />
        <Card.Body>
          <Card.Title>Card Title</Card.Title>
          <Card.Text>
            Some quick example text to build on the card title and make up the
            bulk of the card's content.
          </Card.Text>
          <Button variant="primary">Go somewhere</Button>
        </Card.Body>
      </Card>
      <AthleteStatsTabs />
    </div>
  );
}

export default BasicExample;
```

Figure 16: Basic Example of the Left Sidebar UX/UI Code. Screenshot by Fortune Ndlovu (2024)

The code for the left sidebar begins with a functional component that returns, a division of content that has a card for the top half of the sidebar and imports another component named AthleteStatsTabs for the bottom sidebar as shown in Figure 16. The reason why Figure 16 displays a basic example of the left sidebar component is because the overall code file is too long to fit in one screenshot and therefore I coded an overview version of the code.

Inside the AthleteStatsTabs component are the 4 tabs with their dynamic content adaptation. From an overview perspective, when the AthleteStatsTabs component first renders, it sets the activeKey state to profile by default. The user can navigate between tabs by clicking on the tabs provided such as "Home", "Profile", "Cycle/Bike", and "Swim/Wave". When the Profile tab is active it displays the RunningShoeSvg imported from ./RunningShoeSvg. Similar to "Cycle/Bike", and "Swim/Wave". This way we are only changing the SVG based on the active tab.

4.1.2.3. Code Breakdown

```
import TabsCard from "./TabsCard";
import RunningShoeSvg from "./RunningShoeSvg";
import CyclingBikeSvg from "./CyclingBikeSvg";
import SwimWaveSvg from "./SwimWaveSvg";
import { Tabs, Tab } from "react-bootstrap";

const tabContentMap = {
  profile: <RunningShoeSvg />,
  cycleBike: <CyclingBikeSvg />,
  swimWave: <SwimWaveSvg />,
};

const AthleteStatsTabs = () => {
  const [activeKey, setActiveKey] = useState("profile");

  const handleSelect = (key) => {
    setActiveKey(key);
  };

  return (
    <Tabs activeKey={activeKey} onSelect={handleSelect}>
      <Tab eventKey="profile" title="Profile">
        { /* Content for the Profile tab */ }
        <TabsCard svgContent={tabContentMap[activeKey]} />
      </Tab>
      <Tab eventKey="cycleBike" title="Cycle/Bike">
        { /* Content for the Cycle/Bike tab */ }
        <TabsCard svgContent={tabContentMap[activeKey]} />
      </Tab>
      <Tab eventKey="swimWave" title="Swim/Wave">
        { /* Content for the Swim/Wave tab */ }
        <TabsCard svgContent={tabContentMap[activeKey]} />
      </Tab>
    </Tabs>
  );
};
```

Figure 17: Left Sidebar AthleteStatsTabs Code Breakdown. Screenshot by Fortune Ndlovu (2024)

Figure 17, displays the code breakdown of the AthleteStatsTabs component. The component is returning a react bootstrap Tabs component which has an activeKey that is equal to the activeKey

in question. By default, the activeKey is a profile because I declared a useState variable with an initial value being the profile activeKey. Each Tab within the Tabs component has an eventKey and a string value assigned, this means when a specific tab is clicked the eventKey is assigned to the activeKey. Each tab has content that is a TabsCard component that expects an assignment of the activeKey into its svgContent prop. The tabContentMap is a mapping object that has properties and values, in this instance, the property is the eventKey and the corresponding value is the SVG. Therefore, what we are passing to the svgContent prop is a mapped activeKey, that is we are mapping over the tabContent object and capturing the value that matches the correct activeKey based on the tab we are currently on which is the tab that is currently active.

We know which tab is currently active because of the handleSelect function that expects an argument and uses it to dynamically update the activeKey state variable which then the tabsContentMap uses to map over its object. For example, when the activeKey is profile, it will render the content mapped to the profile key in tabsContentMap. It's worth noting the mapping object tabContentMap does not determine the active tab. Instead, the activeKey state variable managed by the useState hook tracks the active tab and the mapping object is used to display the corresponding content based on the active tab's key. As the user selects different tabs the activeKey state is updated, triggering the re-rendering of the correct content through the mapping object.

To display the correct date as shown in Figure 15, I created a component that returns an SVG and with conditions applied to its curtain elements.

```
import { React, useState } from "react";

const WeeksStyleSvg = () => {
  // State to store the current day of the week (0 = Sunday, 1 = Monday, ..., 6 = Saturday)
  const [currentDay, setCurrentDay] = useState(() => new Date().getDay());

  const daysOfWeek = ["M", "T", "W", "T", "F", "S", "S"];

  return (
    <div>
      {" "}
      <svg
```

Figure 18: Left Sidebar WeeksStyleSvg Code Breakdown. Screenshot by Fortune Ndlovu (2024)

Figure 18 shows the initialization of the WeeksStyleSvg component. In this component, a state variable named currentDay is declared using the useState hook from React. This variable is

initialized with the current day of the week, represented as a number (0 for Sunday, 1 for Monday, ..., 6 for Saturday). The `currentDay` state is used to track the current day in the web app. For example, if today is Sunday, then the value assigned to `currentDay` would be 0.

Additionally, an array named `daysOfWeek` is created, with each element being a single letter representing a day of the week ("M" for Monday, "T" for Tuesday, "W" for Wednesday, etc). The rest of the component is returning an SVG, with logic at the end of the component, for dynamic `currentDay` updates, shown in Figure 19.

```
<g className="recharts-layer recharts-cartesian-axis-tick">
  /* Mapping over daysOfWeek to create text labels and polygons for each day */
  {daysOfWeek.map((day, index) => (
    <g key={index} transform={`translate(${15 + index * 20},73)`}>
      <text
        x="0"
        y="10"

        // Conditional fill color for the text based on the current day
        fill={index + 1 === currentDay ? "#fc5200" : "#2b2b2b"}
        textAnchor="middle"

        // Conditional font weight for the text based on the current day
        fontWeight={index + 1 === currentDay ? "800" : "400"}
      >
        {day}
      </text>

      /* Polygon to highlight the current day */
      {index + 1 === currentDay && (
        <svg x="-5" y="15" width="10" height="10">
          <polygon points="5,0 0,10 10,10" fill="#fc5200"></polygon>
        </svg>
      )}
    </g>
  )}
</g>
```

Figure 19: Left Sidebar WeeksStyleSvg daysOfWeek Map Code Breakdown. Screenshot by Fortune Ndlovu (2024)

Figure 19 is the remainder of the `WeeksStyleSvg` component; this part of the code is responsible for creating a visual representation of the days of the week. The outer `<g>` element with the

class names `recharts-layer` and `recharts-cartesian-axis-tick` is a group element used to encapsulate all the elements related to the days of the week.

Inside this group, the `daysOfWeek` array is mapped over to create a new group `<g>` for each day. Each group is translated horizontally based on the index of the day, creating a horizontal sequence of days. The `transform` attribute in SVG is used to move, scale, rotate, and skew SVG elements. In this case, the `translate` function is being used, which moves the element from its current position. The `translate` function takes two parameters: `translate(x,y)`. The `x` parameter specifies how much to move the element horizontally and the `y` parameter specifies how much to move the element vertically. In my code shown in Figure 19, `translate(${15 + index * 20}, 73)`, the `x` value is dynamically calculated as $15 + \text{index} * 20$. This means that each day of the week is moved horizontally by 15 units plus 20 units times the index of the day in the `daysOfWeek` array. This creates a horizontal spacing of 20 units between each day. The `y` value is a constant 73, which means that all days are aligned at the same vertical position.

Inside each day's group, a `<text>` element is created. This element is the letter representing the day of the week. The `x` and `y` attributes position the text within the group.

The fill color and font weight of the text are conditionally set based on whether the index of the day (plus one) matches the `currentDay` state. If it does, the text color is set to `#fc5200` and the font weight is set to 800, highlighting the current day. Otherwise, the text color is `#2b2b2b` and the font weight is 400. Additionally, if the index of the day (plus one) matches the `currentDay` state, a `<polygon>` element is created inside the `<svg>` element. This polygon serves as a visual highlight for the current day. The `points` attributes of the polygon define the shape and size of the polygon, and the `fill` attribute sets its color to `#fc5200`.

4.1.2.4. Complexities and Solutions

The most complex implementation in this case was implementing the tabs functionality. By default, React Bootstrap provides the tabs component but when modifying the default component, it can get complicated depending on the intended use case. For my use case, I wanted the user to tab through the tabs and the content for each tab does not refresh, the entire tabs component creating a blinking effect, leading to a non-modern user experience.

The solution for this was to break the code into smaller and more manageable components and apply logic to only show the components needed for the current tab. Conceptually the way I achieved this solution was by capturing the content that changes into an object and I could use this object as a mapping object reference point to match the current tab with and only populating the content that matches the current tab for this knowledge of the current tab that the user had clicked was also captured. Overall breaking the component into smaller parts and fitting them back together dynamically by tracking user interactions and matching them with the dynamic data to only display the corresponding data was the most optimal solution.

The last complexity was in the `WeeksStyleSvg` component, when I used React's `useState` and `map` functions to create a dynamic SVG element. The problem I encountered was the state and day indexing. The `useState` hook is used to store the current day of the week as shown in Figure 18. The new `Date().getDay()` function returns a number from 0 (Sunday) to 6 (Saturday). However, my `daysOfWeek` array starts with "M" (Monday) at index 0, while `getDay` returns 1 for Monday. This is because `getDay()` follows a conversion of Sunday being 0, Monday being 1, and so on till Saturday which is 6. To align these, I used `index + 1 === currentDay` in my conditions. This effectively shifts the indexing of `daysOfWeek` to match the `getDay()` function. This way when `getDay()` returns 1 (Monday), it highlights "M" in my array.

4.1.3. Lessons Learned

4.1.3.1. User-Centered Design

At the heart of UX design is the user. Every design decision should be made with the user's needs, preferences, and behaviors in mind. This principle is about understanding their goals, empathizing, and designing solutions that meet theirs (Christopher Nguyen, 2023). User-centered design was what I learned when implementing the Basic UX/UI in React because I implemented user-friendly interface components that were both easy to use and consistent. For example, I used the same font sizes and hex values for color throughout elements with relationships and I used the same iconography library, which was React Icons for all my icons, additionally, I used React Bootstrap as boilerplate component integration before additional implementations. This all created a User-Centered Design because of the consistency of the underlying decision-making for consistency.

4.1.3.2. Composition and Reusability

The key feature of React is the composition of components. It is important that you can add functionality to a component without causing rippling changes (legacy reactjs, 2024). I learned Composition from creating my own components and reusing React Bootstraps components when I was adding unique functionality and also making sure the components were accessible.

As the developer, I built tiny components that were then used to build bigger components and eventually leading to the full desired implementation being built, in this case the Basic UX/UI. I learned that components must have their own logic and they function as plugins in a larger system (Filip Grkinic, 2022). A good example of composition and reusability was when I built components to make up the header, showing certain components upon user interactions and later reusing them for smaller screen sizes while ensuring the components were still accessible on the keyboard. I also broke the left sidebar into smaller components to only update content that needed to be updated based on the tab the user interacted with, this created a smoother user experience.

4.1.3.3. Performance

React is used to build the front end of an application, meaning what people see and interact with. The real magic of React lies in the way it handles the state of an application. It has a thing called Virtual DOM, which is a virtual copy of the actual DOM. When a user interacts with the application, the state changes, and React will automatically update the UI, but only the parts that need to be updated thanks to the Virtual DOM (Filip Grkinic, 2022). Therefore I took advantage of this performance by breaking my components into smaller components and only loading the components that were necessary for the user's current view, reducing initial load time and improving the overall performance of the web app. Additionally, I used the best coding standards for React such as using hooks, I have leveraged the power of React functional components and state management. Conditional rendering with ternary operator, and logical OR and logical AND expressions for unique executions made my code more readable and efficient as shown in Figure 15. It allowed me to handle different states of my web app and render the UI based on certain conditions.

4.1.4. In Summary

In this section of my report, I have detailed the process complexity of implementing a basic UX/UI for the Strava web app replica homepage. I have coded accessible and visually consistent header and footer components, managed dropdown menus, left, center, and right sidebars, and ensured responsive design. I also tackled tab functionalities by breaking down components and displaying content based on user interactions. The overall result is a responsive, accessible homepage that provides a smooth user experience. My approach was user-centered, leveraging React features like Virtual DOM and efficient state management for performance optimization. In the next section of the report, I will share the process of activity management with create, read, update, and delete (CRUD) operations on the web app.

4.2. Activity Management

After the creation of basic UX/UI creation, I turned my focus to activity management, specifically having the user be able to create, read, update, and delete (CRUD) their activities. For these operations to be feasible, I would need a database to hold the data of the activities that the user has created and based on the knowledge I gained from creating the UX/UI, I knew I would also need to create components that feed off each other the data of the activity that user has created.

4.2.1. Firebase Database

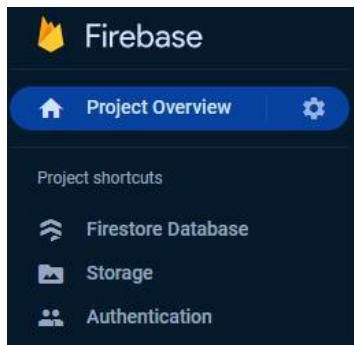


Figure 20: Firebase Project Features. Screenshot by Fortune Ndlovu (2024)

As mentioned in the literature review Firebase is a database, which means it can hold data for apps for iOS, Android, Web Apps, and Unity Flutter builds. In my case, I used it for a web app and Firebase offers services within builds for ways to build applications in general. I will be using Firebase for its Firestore Database, storage, and Authentication Features as shown in Figure 20. Firestore is a flexible, scalable NoSQL cloud Database, used to store and synchronize data. Storage will be used to store photos as it's built on fast and secure Google Cloud infrastructure. Authentication will be used to identify users for Multiple-user integration later on in the development process (Firebase, 2024). As of now, the focus is on the Firestore Database and Storage because we are on the chapter on Activity Management.

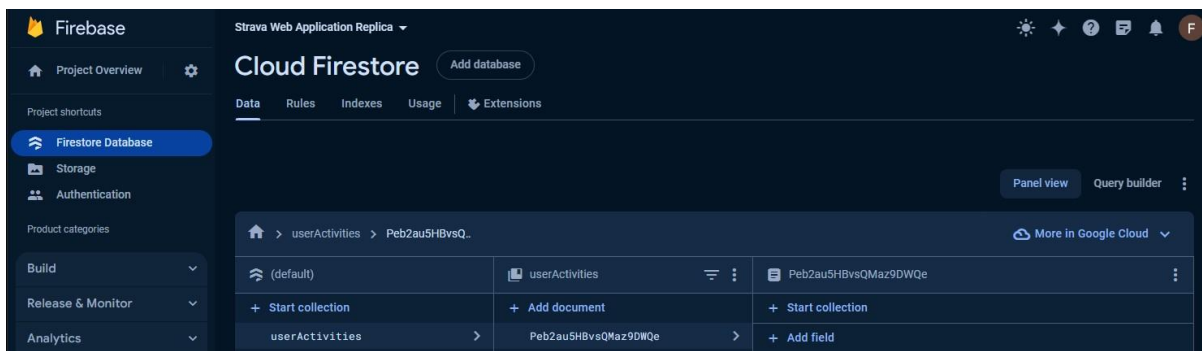


Figure 21: Firestore Dashboard. Screenshot by Fortune Ndlovu (2024)

The design of Firestore is interesting because the design process of the way it holds data is important, Figure 21 shows the dashboard process. There are three columns, and the first column

holds a collection. These are the primary storage units in Firestore. You can have multiple collections, each holding its own unique set of data. Each collection contains documents, which are displayed in the second column of the dashboard. A document represents a unit of data within a collection. The actual data is held within the documents and is displayed in the third column of the dashboard. Additionally, each document can contain complex nested data in addition to simple data types and you can chain these collections and documents together to access deeply nested data. This hierarchical data structure allows for flexible and scalable data organization (Firebase, 2024).

In the Firebase project settings you get information about your project and a guide to how to integrate Firebase into your web app. Figure 22 below shows the Software Development Kit (SDK) setup and configuration. Essentially all that is needed is to npm install firebase copy and paste the code provided to initialize Firebase and begin using the SDKs for the preferred Firebase products.

SDK setup and configuration

npm CDN Config

If you're already using [npm](#) and a module bundler such as [webpack](#) or [Rollup](#), you can run the following command to install the latest SDK ([Learn more](#)):

```
$ npm install firebase
```

Then, initialize Firebase and begin using the SDKs for the products you'd like to use.

```
// Import the functions you need from the SDKs you need
import { initializeApp } from "firebase/app";
import { getAnalytics } from "firebase/analytics";
// TODO: Add SDKs for Firebase products that you want to use
// https://firebase.google.com/docs/web/setup#available-libraries

// Your web app's Firebase configuration
// For Firebase JS SDK v7.20.0 and later, measurementId is optional
const firebaseConfig = {
  apiKey: "AIzaSyACKlxBeZiCfUoFcF0akUU3Mxg7kFOR_Jo",
  authDomain: "strava-web-application-replica.firebaseio.com",
  projectId: "strava-web-application-replica",
  storageBucket: "strava-web-application-replica.appspot.com",
  messagingSenderId: "1061110930450",
  appId: "1:1061110930450:web:2b85608831d9e8a1734ccb",
  measurementId: "G-FBJTPMR2X9"
};

// Initialize Firebase
const app = initializeApp(firebaseConfig);
const analytics = getAnalytics(app);
```

Note: This option uses the [modular JavaScript SDK](#), which provides reduced SDK size.

Figure 22: Firebase Project Settings. Screenshot by Fortune Ndlovu (2024)

```
JS firebase.js 1, M X
src > firebase > JS firebase.js > ...
You, 7 seconds ago | 1 author (You)
1 import { initializeApp } from "firebase/app";
2 import { getFirestore } from "@firebase/firestore";
3 import { getStorage } from "firebase/storage";
4 import { getAuth } from "firebase/auth";
5
6 // TODO: Add SDKs for Firebase products that you want to use
7 // https://firebase.google.com/docs/web/setup#available-libraries
8
9 // Your web app's Firebase configuration
10 // For Firebase JS SDK v7.20.0 and later, measurementId is optional
11 const firebaseConfig = {
12   apiKey: "AIzaSyACk1xBzZiCfUoFcFOakUU3Mxg7kFOR_Jo",
13   authDomain: "strava-web-application-replica.firebaseio.com",
14   projectId: "strava-web-application-replica",
15   storageBucket: "strava-web-application-replica.appspot.com",
16   messagingSenderId: "1061110930450",
17   appId: "1:1061110930450:web:2b85608831d9e8a1734ccb",
18   measurementId: "G-FBJTPMR2X9"
19 };
20
21 const app = initializeApp(firebaseConfig);
22 const db = getFirestore(app);
23 const storage = getStorage(app);
24 const auth = getAuth(app);
25
26 // Get the current user ID after authentication
27 const getCurrentUserId = () => {
28   const user = auth.currentUser;
29   return user ? user.uid : null;
30 };
31
32 export { app, db, storage, auth, getCurrentUserId }
```

Figure 23: Firebase Configuration. Screenshot by Fortune Ndlovu (2024)

Figure 23 shows a component in React that I created with the code from my Firebase project settings, however it is intentionally updated with imports of the products I will be using, the firebaseConfig object stayed the same because that is sensitive data about my Firebase project. I am also initializing the Firebase project app, and using the constant variable that holds this initialization to get the projects from Firebase that I have imported above and export each constant variable independently together. This means now in my web app I can use the app constant variable as a reference to my entire Firebase project, I can use the db constant variable to get reference to my Firestore database for data population and manipulation, I can use the storage constant variable to get reference to the storage product to hold images for my web app and auth constant for adding authentication later. Now that configuration has been completed and does not need to be touched, I could begin putting my Firebase database to work. The User ActivitiesManager section will outline the process of making use of my Firebase database.

4.2.2. UserActivitiesManager Component, Create, Read, Update, Delete

UserActivitiesManager is a component to manage all of the activity operations in one place. This includes functions for creating, reading, updating, and deleting activities created by the user. I created an array to hold all of the userActivities in the web app. I created a createActivity asynchronous function that takes in an argument as an object through its parameter newActivity which is a temporary variable. I use the newActivity object as the data for my document in Firebase and because I am using my db constant variable from Figure 22 which is a reference to my database I get access to methods to help me populate and manipulate the data.

```
const UserActivitiesManager = ({ showForm }) => {
  const [userActivities, setUserActivities] = useState([]);
  const userActivitiesCollection = collection(db, "userActivities");

  const createActivity = async (newActivity) => {
    // Adding the creation date and time to the new activity object
    const currentTimeStamp = serverTimestamp(); // Import serverTimestamp from 'firebase/firestore'
    newActivity.createdAt = currentTimeStamp;

    const docRef = await addDoc(userActivitiesCollection, {
      ...newActivity
    });

    // Creating a new object by spreading the properties of the doc id
    // Ensuring the returned object includes both the original activity data and Id assigned by Firestore
    const createdActivity = { ...newActivity, id: docRef.id };

    return createdActivity;
  };
};
```

Figure 24: createActivity Function. Screenshot by Fortune Ndlovu (2024)

Figure 24, shows the creation of the userActivitiesManager functional component and the createActivity function inside. What the code essentially does is that we are getting a reference to our Firebase collection and assigning it to a constant variable named userActivitiesCollection, it is a global variable because I needed to use it throughout the rest of the overall code. The createActivity is asynchronous and this means while we are executing code in the code block of the asynchronous function will continue executing the other lines of code in the rest of the file. In the createActivity code block, we get the timestamp of the created activity, and we add a document into the userActivitiesCollection, the document we add is an object with new data by using the spread operator we create a new instance of the data for data integrity and garbage collection purposes. What this function is returning is the createdActivity which is an object with the users data and also the document id corresponding to that data. The id was auto generated by Firestore

within the addition of the document instance. It's worth noting that while the `createActivity` function is adding a document representing the users data into the `userActivities` collection, it does not return the data from Firestore, it only adds the data to Firestore upon function execution. However, we do also capture the data that was passed as an argument and also attached the document id which is return from Firestore and we wrap this up in an object that we return. This way we are not fetching the data from Firestore when the data is the same as the data passed as an argument and when we attach the document id we ensure that the correct data has the corresponding unique identifier for data integrity purposes.

```
return (  
  <div>  
    {showForm && <ManualEntryForm onCreateActivity={createActivity} />}  
  )
```

Figure 24: ManualEntryForm Import. Screenshot by Fortune Ndlovu (2024)

The `UserActivitiesManager` component returns a `ManualEntryForm` component and assigns it the `createActivity` function as a prop. The `ManualEntryForm` component is essentially a form that the user fills in information about their activity, shown in Figure 25 and as the user clicks the orange Create button the `onCreateActivity` prop from `ManualEntryForm` is passed to the `createActivity` prop here in `userActivitiesManager` so the data that the user has inputted via form inputs can be uploaded to the Firestore database as a document with the data as an object part of the user activities collection storage unit. This means the two props `onCreateActivity` and `createActivity` are the getaway for transferring the user's data between both components, this is powerful because it allows the code to be more concise and easier to read.

4.2.2.1. Manual Entry Form UX/UI, Create

The screenshot displays the Strava 'Manual Entry' form. At the top, the Strava logo and navigation menu (Dashboard, Training, Explore, Challenges) are visible, along with a 'Start Trial' button. The form is titled 'Manual Entry' and is organized into several sections:

- Device:** A sidebar menu with options for 'File', 'Manual', and 'Mobile'.
- Distance:** A text input field with '0' and a 'Kilometers' dropdown menu.
- Duration:** A time input field showing '0:00:00'.
- Elevation:** A text input field with '0' and a 'Meters' dropdown menu.
- Sport:** A dropdown menu currently set to 'All Sport Types'.
- Date & Time:** A date input field showing 'dd/mm/yyyy' with a calendar icon and a time selection icon.
- Title:** A text input field containing 'Night Run'.
- Description:** A text area with the placeholder text 'How'd it go?'.
- Media:** A large rectangular area with a central icon and the text 'Drag and drop media or click to upload'.

At the bottom of the form, there are two buttons: 'Create' (in orange) and 'Cancel'. Below the form, there are two sections: 'Your Recent Activities' (with the text 'No recent activities found' and 'Upload an activity') and 'Strava Stories' (with the text 'With athlete profiles, training tips and advice, and the latest product updates, Strava Stories is the place to discover the latest content from Strava.'). The footer contains the Strava logo, copyright information (© 2023 Strava), and links for 'About', 'Explore', 'Follow', 'Help', and 'More'.

Figure 25: ManualEntryForm UX/UI. Screenshot by Fortune Ndlovu (2024)

Figure 25 shows the UX/UI design of the form used by the user to input their data about the activity they have done. The form has 11 inputs, for distance, duration, elevation, sport, date and time, title, description, and media. I have divided the inputs into 2 different components in the ManualEntryForm component where one component will be responsible for the activity stats and the other for the activity details. Firstly to get the value of the user from a given input I first need to store that value in a state variable for each input as shown in Figure 26, the beauty about these state variables is that I can use their built in methods to update the current value of the state and in the Figure 26 screen cap we also do bring the onCreateActivity prop into the ManualEntryForm component through restructuring.

```

const ManualEntryForm = ({ onCreateActivity }) => {
  const navigate = useNavigate(); // Hook to navigate between pages
  const [newImages, setNewImages] = useState([]); // Change to an array for multiple images
  const [dragging, setDragging] = useState(false);

  const [newDistance, setNewDistance] = useState(0);
  const [newHour, setNewHour] = useState(0);
  const [newMinute, setNewMinute] = useState(0);
  const [newSecond, setNewSecond] = useState(0);
  const [newElevation, setNewElevation] = useState(0);

  const [newSportSelection, setNewSportSelection] = useState("");
  const [newDateValue, setNewDateValue] = useState("");
  const [newTimeValue, setNewTimeValue] = useState("");
  const [newActivity, setNewActivity] = useState("");
  const [newDescription, setNewDescription] = useState("");

```

Figure 26: ManualEntryForm useState hooks Code . Screenshot by Fortune Ndlovu (2024)

```

<ActivityStats
  distanceValue={newDistance}
  distanceOnChange={(e) => setNewDistance(e.target.value)}
  hourValue={newHour}
  hourOnChange={(e) => setNewHour(e.target.value)}
  minuteValue={newMinute}
  minuteOnChange={(e) => setNewMinute(e.target.value)}
  secondValue={newSecond}
  secondOnChange={(e) => setNewSecond(e.target.value)}
  elevationValue={newElevation}
  elevationOnChange={(e) => setNewElevation(e.target.value)}
/>


---


<ActivityDetails
  sportSelectionValue={newSportSelection}
  sportSelectionOnChange={(value) => setNewSportSelection(value)}
  dateValue={newDateValue}
  dateOnChange={(e) => setNewDateValue(e.target.value)}
  timeValue={newTimeValue}
  timeOnChange={(e) => setNewTimeValue(e.target.value)}
  titleValue={newActivity}
  titleOnChange={(e) => setNewActivity(e.target.value)}
  descriptionValue={newDescription}
  descriptionOnChange={(e) => setNewDescription(e.target.value)}
/>

```

Figure 27: ManualEntryForm ActivityStats and ActivityDetails Components. Screenshot by Fortune Ndlovu (2024)

Figure 27 shows the code I coded of the two components that break up the overall ManualEntryForm component making it more concise and maintainable. The ActivityStats and ActivityDetails components essential have input fields and they assign the value of the input fields via props these props hold the value whatever the user types but moreover, they also pass as a

prop, and using onChange I can get exactly whatever the user is typing in the input. Whatever the user typed in this event it is captured and the evaluated value is passed to the state variable. At this point, the user has finished typing, and the result has been captured and temporarily stored.

```
//Triggering the creation of a new activity doc in Firestore
const createdActivity = await onCreateActivity({
  // Passing an object as an argument containing details needed for the activity
  distance: newDistance,
  hour: newHour,
  minute: newMinute,
  second: newSecond,
  elevation: newElevation,
  sport: newSportSelection,
  date: newDateValue,
  time: newTimeValue,
  name: newActivity,
  description: newDescription,
  imageUrls: imageUrls, // Use the outer-scoped imageUrl
});

// Redirect to the new activity details page using the id from the created activity
const createdActivityID = createdActivity?.id;
if (createdActivityID) {
  navigate(`/home/activity/${createdActivity.id}`);
}
```

Figure 28: ManualEntryForm createdActivity Object. Screenshot by Fortune Ndlovu (2024)

When the user clicks the Create orange button shown in Figure 25, we essentially get all the user inputs that we temporarily stored in the state variables and create an instance object where we pass the values to the newly defined properties. This instance object will be passed to the onCreateActivity method which is a prop from the userActivitiesManager component so the object will be brought to the component to perform the necessary computations to add the object to Firestore and later retrieve the unique identifier of the object that was newly created. If the created activity has been created and has an id, we navigate to a new URL with reference to the id of the newly created activity. This all happens once the user clicks the Create orange button.

4.2.2.2. ActivityDetailsPage for Read and EditActivityForm for Update

Once the user has been dynamically navigated to a new page that is also dynamically generated by the ID of the activity they have created, they can read the information that they have inputted visually and consistently using React Bootstrap, shown in Figure 29. We are able to inject the

user's information in the React Bootstrap components because we fetch the data from Firebase and using dot notation, we can actually plug in the pieces of data in the interface, respectively.

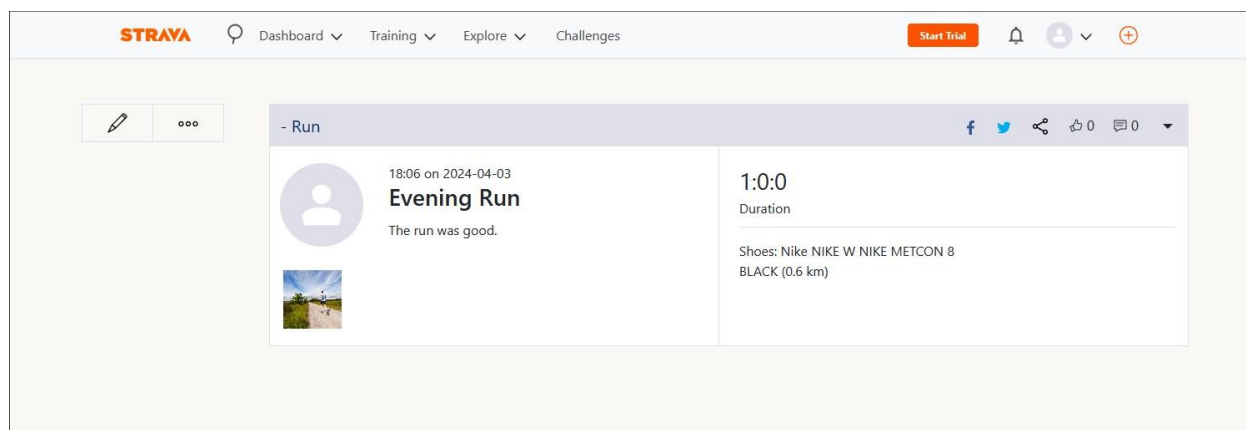


Figure 29: ActivityDetails Component. Screenshot by Fortune Ndlovu (2024)

```
useEffect(() => {
  // Fetching activity details from the Firestore
  const fetchActivityDetails = async () => {
    const activityDocRef = doc(db, "userActivities", activityId);
    const activityDocSnapshot = await getDoc(activityDocRef);
    if (activityDocSnapshot.exists()) {
      setActivityDetails(activityDocSnapshot.data());
    }
  };

  return () => {
    fetchActivityDetails();
  }
});
```

Figure 30: ActivityDetails Component. Screenshot by Fortune Ndlovu (2024)

Figure 30 shows how I was able to fetch the data from Firestore so I could plug it into the interface. Essentially, I got the document reference inside the user activities collection that had the same id as the newly created activity and once I had the document reference meaning I had a path for where the document was, I was able to get the document and its data and assign this document data into a state variable in this component so I could spread the data across the UI.

To edit the data or update the user clicks the pencil icon on the top left of the screen shown in Figure 29 and when they click this button, they are again dynamically navigated into a new component named EditActivityForm component which is dynamically generated.

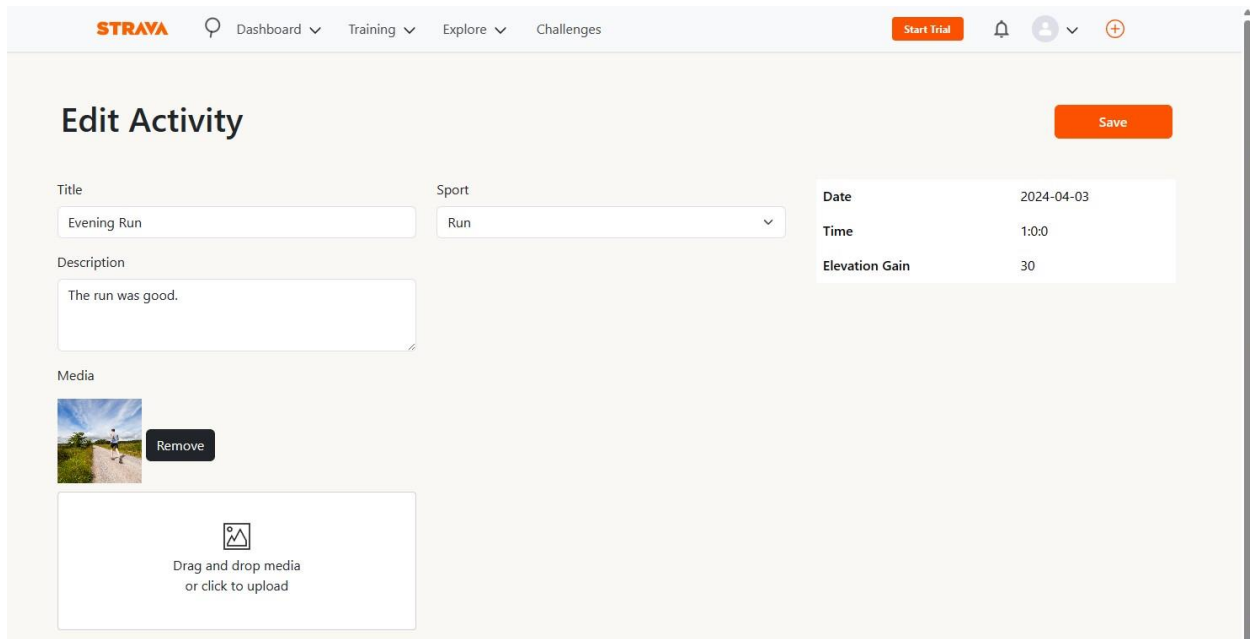


Figure 31: EditActivityForm UX/UI Component. Screenshot by Fortune Ndlovu (2024)

Figure 31 shows what EditActivityForm's UX/UI looks like, the input fields are essentially imported into the component pre-filled with the fetched data from the database, and the user can update the areas they want to update, and when they press save, they are navigated back to the activity details page shown in Figure 29.

The way I edit the data in the EditActivityForm is that I create a new object that will hold the values and for each input, I attach an onChange event handler that triggers any change in the input field, I track the name and value of this event and update the state variable with a shallow copy of the existing editedActivity state to ensure immutability and a dynamic update of the property specified by name with the new data into the overall object all as shown in Figure 32.


```

const EditActivityForm = () => {
  const { activityId } = useParams(); //extract from the URL
  const navigate = useNavigate(); //programmatic navigation
  const [activityDetails, setActivityDetails] = useState(null);
  const [editedActivity, setEditedActivity] = useState({
    name: "",
    description: "",
    sport: "",
    // ... other fields
    imageUrl: [], // Array to store image URLs
  });

  useEffect(() => {
    // Fetch the activity details when the component mounts based on id
    const fetchActivityDetails = async () => {
      // Generating a ref to the Firestore doc for the specified activityId
      const activityDoc = doc(db, "userActivities", activityId);
      // Retrieving the doc snapshot for the specified doc ref
      const activitySnapshot = await getDoc(activityDoc);

      // if the activity doc exists update the state by creating an object by copying properties of an existing object
      if (activitySnapshot.exists()) {
        setActivityDetails({
          ...activitySnapshot.data(),
          id: activitySnapshot.id,
        });
        setEditedActivity({
          ...activitySnapshot.data(),
        });
      }
    };

    fetchActivityDetails();
  }, [activityId]);

  // Handling input changes, connected to input fields through the onChange event
  const handleInputChange = (e) => {
    // Destructuring the properties name and value from the event object
    const { name, value } = e.target; // You, 2 months ago • When the user clicks the "Save Changes" button,...

    // Creating a shallow copy of the existing editedActivity state to ensure immutability
    // Dynamically updating the property specified by name with the new value
    setEditedActivity({
      ...editedActivity,
      [name]: value,
    });
  };
};

```

Figure 32: EditActivityForm Component edit data Code. Screenshot by Fortune Ndlovu (2024)

```

// Handles the saving of the edited activity to Firestore by updating the doc
const handleSaveChanges = async () => {
  const userDoc = doc(db, "userActivities", activityId);
  await updateDoc(userDoc, editedActivity);

  navigate(`/home/activity/${activityId}`);
}; // You, 2 months ago • When the user clicks the "Save Changes" button,...

```

Figure 33: EditActivityForm Component handleSaveChanges Code. Screenshot by Fortune Ndlovu (2024)

Figure 32 shows how I was able to edit the newly created activity, and it was essentially fetching the data from the database for initial visual display in the inputs and then having the ability to store

the new data the user has edited into an object passed to a state variable as a value. This means when the user edits and inputs I pass the value to the appropriate property in the object. Lastly as shown in Fire 32 when the user clicks the save orange button shown in Figure 31, I essentially update the Firestore document with the new data as shown in Figure 33 and navigate back to the ActivityDetailsPage where we fetch the newly created display and display the results as shown in Figure 29.

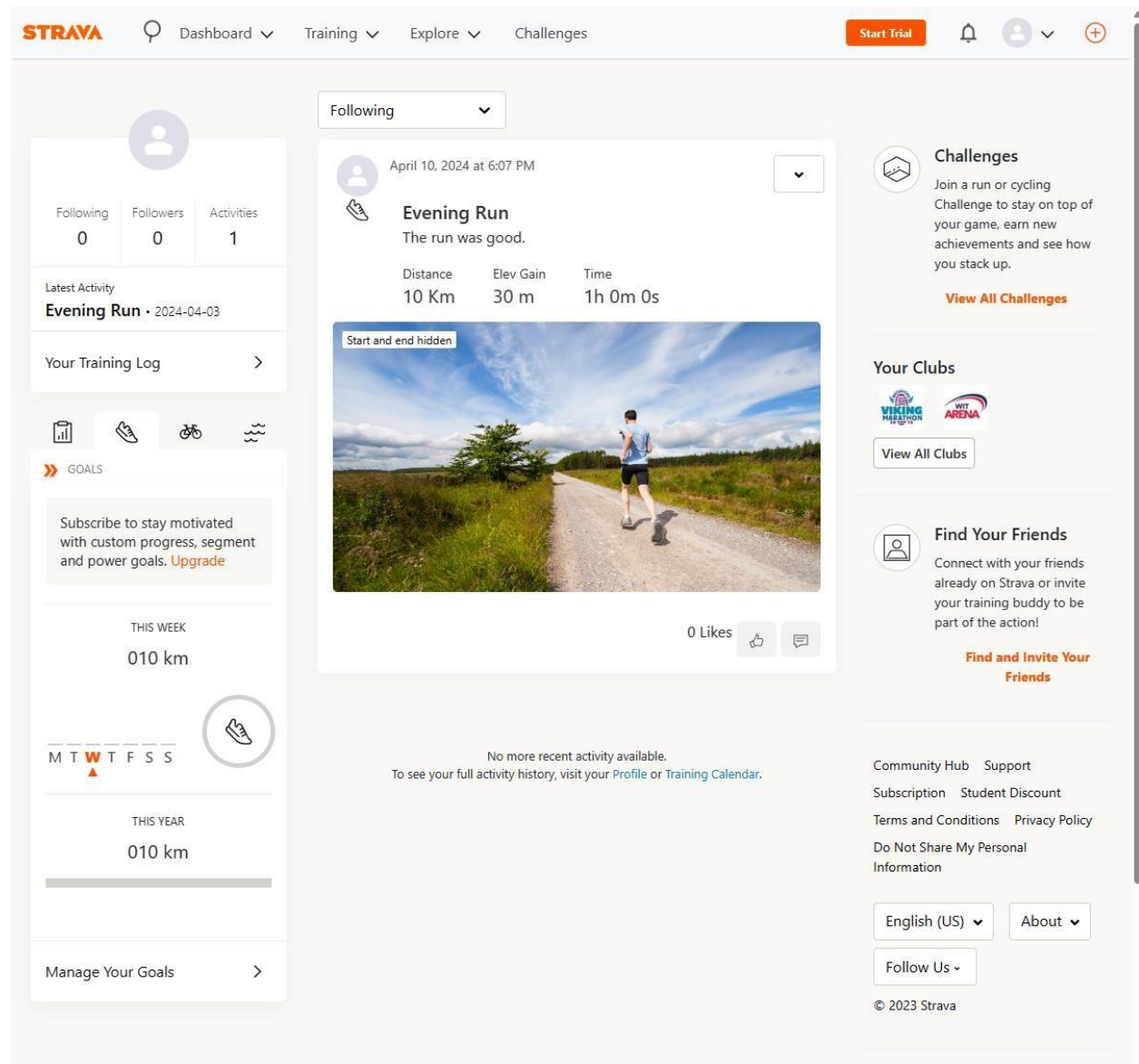


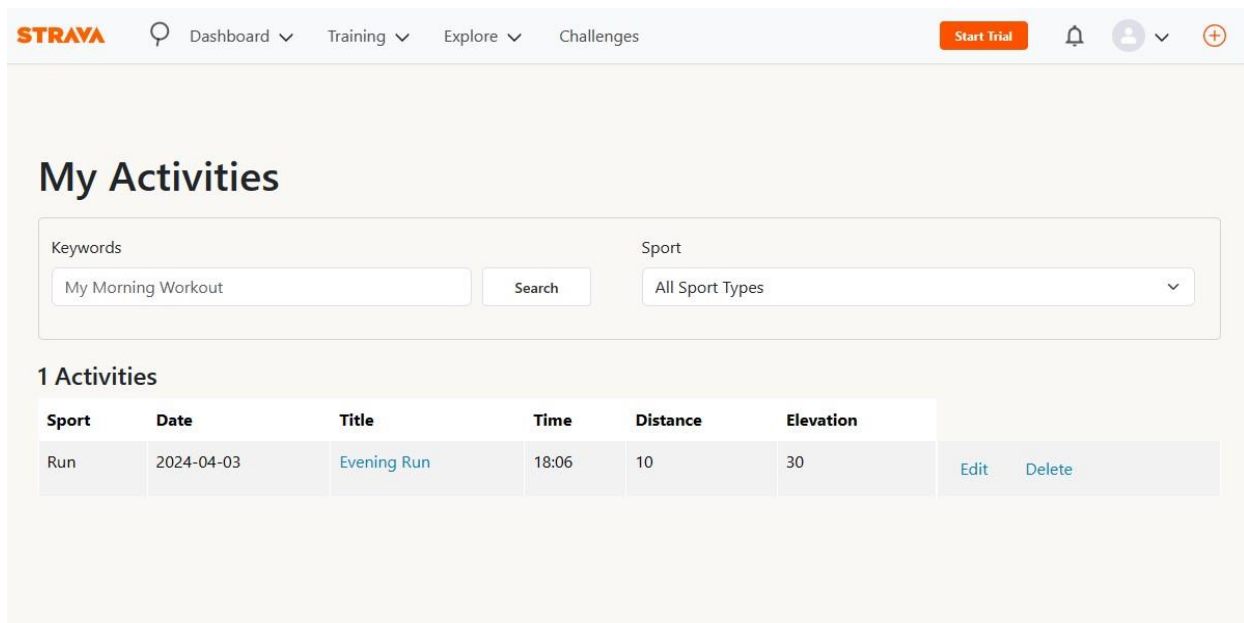
Figure 34: Dashboard UX/UI With Generated Activity. Screenshot by Fortune Ndlovu (2024)

Figure 34 shows the updated UX/UI of the dashboard with the newly created activity data displayed in the form of a card. In this instance I am fetching the data from Firestore and once I

have fetched the data, I update the state variable with the data, and I populate the card with the data using dot notation. The card has data such as the timestamp of when the activity was created, the name, description, distance, elevation, time, and image. What is also interesting now that we have data stored in Firestore I was able to just fetch the data and populate other surrounding components respectively, such as the left component sidebar.

4.2.2.3. MyActivitiesTable for Read, Update and Delete

When the user interacts the header navigation specifically hovering over the training dropdown and clicking the my activities link in the dropdown, they are navigated to a new page about all of their activities, in this instance shown in Figure 35, the user has only created one activity so far and they can create more if they want and they will be incrementally be added to the database and fetched to display on the UI.



The screenshot displays the Strava 'My Activities' page. At the top, there is a navigation bar with the Strava logo, a search icon, and menu items for 'Dashboard', 'Training', 'Explore', and 'Challenges'. A 'Start Trial' button is visible on the right. Below the navigation bar, the page title 'My Activities' is prominently displayed. Underneath the title, there is a search section with a 'Keywords' input field containing 'My Morning Workout', a 'Search' button, and a 'Sport' dropdown menu currently set to 'All Sport Types'. Below the search section, the heading '1 Activities' is shown above a table. The table has columns for 'Sport', 'Date', 'Title', 'Time', 'Distance', and 'Elevation'. The single activity listed is 'Run' on '2024-04-03' with the title 'Evening Run', a time of '18:06', a distance of '10', and an elevation of '30'. To the right of the table row, there are 'Edit' and 'Delete' buttons.

Sport	Date	Title	Time	Distance	Elevation	
Run	2024-04-03	Evening Run	18:06	10	30	Edit Delete

Figure 35: MyActivitiesTable UX/UI. Screenshot by Fortune Ndlovu (2024)

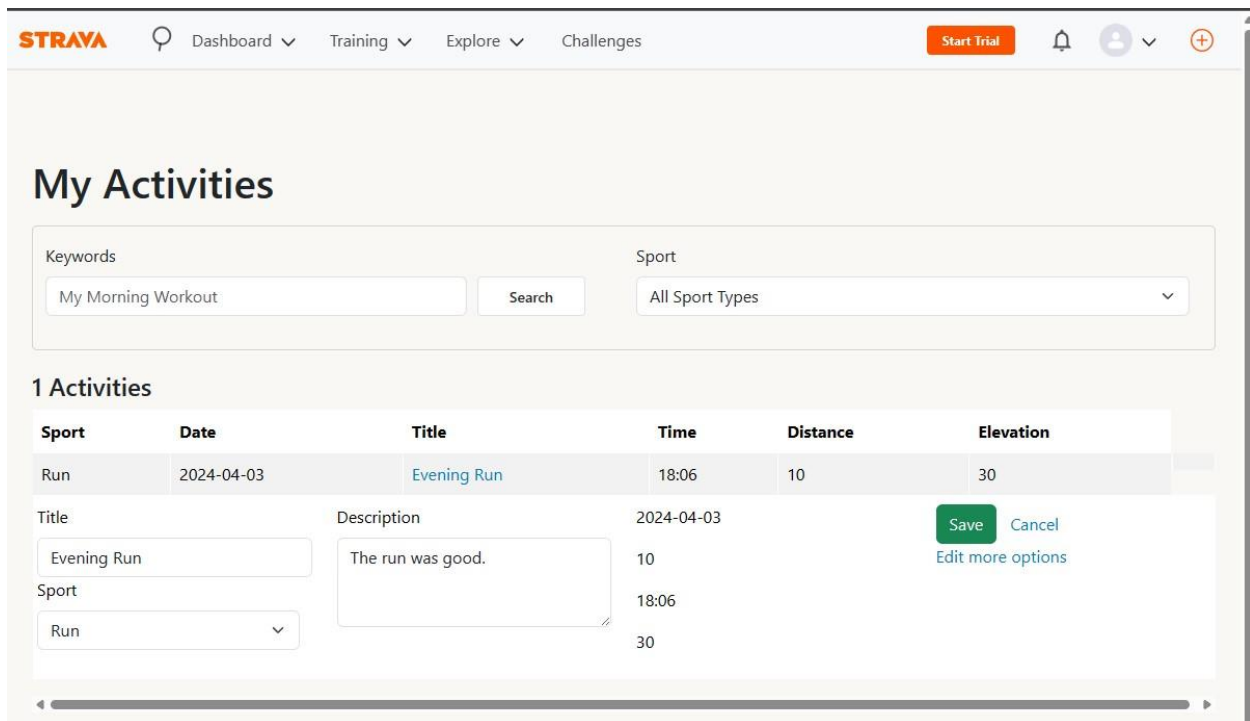


Figure 36: MyActivitiesTable Edit Button clicked UX/UI. Screenshot by Fortune Ndlovu (2024)

Figure 35 shows the UX/UI of the MyActivitiesTable component, its purpose is to display all of the activities that the user has created in the form of a table. Figure 36 shows what happens when the edit button is clicked from the Figure 35 table row also next to the delete button. A component named EditableRow displays, and this component has input fields to update the data of the activity in question. In the EditableRow component rendered after clicking the edit button the user can edit the Title, Description, and Sport type, and click the green save button which will remove the EditTableRow from view and also update the relevant Firestore collection document with the data. Moreover the user can click the cancel link to remove the EditableRow or the Edit more option link to bring them to the component with more input fields to edit.

This flexibility to Read the activities, edit, and delete, brings us back to the User Activities Manager component because I not only created the function to create activities, but I also created functions to edit, and delete activities and even listen for changes to the user activities collection in the database and get real-time updates. The reasoning behind this logic was to have all of the functions in the same component for conciseness, readability, maintaining purposes, and even data integrity because the user's data is handled closely together to reduce errors along the way.

```

const editActivity = async (index, updatedActivity) => {
  // Obtaining the id of doc in Firestore that corresponds to the activity being edited
  const userDoc = doc(db, "userActivities", userActivities[index].id);

  const batch = writeBatch(db); // Initializes a batch to write multiple operations together
  // A writeBatch is a set of operations that are performed atomically(all or none)

  // Check if userActivities[index].imageUrl is defined
  const imageUrl = userActivities[index].imageUrl || null;

  // Update the document with the new activity details, including the image field
  batch.update(userDoc, { ...updatedActivity, imageUrl });

  // Commit the batch to Firestore, this ensures that all the updates in the batch are applied automatically
  await batch.commit();
};

const deleteActivity = async (index) => {
  // Obtaining the id of the doc in the Firestore that corresponds to the activity being deleted
  const userDoc = doc(db, "userActivities", userActivities[index].id);
  const batch = writeBatch(db); // Initialize a WriteBatch

  // Delete the activity
  batch.delete(userDoc);

  // Commit the batch
  await batch.commit();
};

// Listening for changes to the collection and get real-time updates
useEffect(() => {
  const userActivitiesQuery = query(
    userActivitiesCollection,
    where("userId", "=", currentUserId)
  );

  const unsubscribe = onSnapshot(userActivitiesQuery, (snapshot) => {
    const sortedActivities = snapshot.docs
      .map((doc) => ({ ...doc.data(), id: doc.id }))
      .sort((a, b) => new Date(b.date) - new Date(a.date));

    setUserActivities(sortedActivities);
  });

  return () => unsubscribe();
});

```

Figure 37: UserActivitiesManager Functions. Screenshot by Fortune Ndlovu (2024)

```

return (
  <div>
    {showForm && <ManualEntryForm onCreateActivity={createActivity} />}
    {!showForm && (
      <MyActivitiesTable
        activities={userActivities}
        onEditActivity={editActivity}
        onDeleteActivity={deleteActivity}
      />
    )}
  </div>
);
};

export default UserActivitiesManager;

```

Figure 38: MyActivitiesTable component return. Screenshot by Fortune Ndlovu (2024)

Other than the createActivity function the UserActivitiesManager component also manages functions to edit, delete, and listen for changes as shown in Figure 37. The editActivity function is also asynchronous but takes in two arguments. The index is for the index of the image we are editing and the updatedActivity is the object with the data of the updated activity. What happens here is we get reference to our userActivities collection in Firebase and then writeBatch on the database, what this means is we are able to edit a lot of input fields and they will all be wrapped up into a single atomic operation as a batch, this means then all we need to do it update the document in our collection with the operations and commit the batch, rather than sending each operation one by one to our database, writeBatch allows developers to ensure that all the updates in the batch are applied automatically and all as one maintaining data consistency (Firebase, 2024). The second function is deleteActivity also shown in Figure 37, in this function, we also writeBatch on the database and delete the document and commit the batch.

I am also not only returning the ManualEntryForm but also the MyActivitiesTable component and I am also using the functions in the UserActivitiesManager as props such as the activities we are listening for in real-time, editActivity for editing activities, and deleteActivity for deleting activities. The MyActivitiesTable gets passed these functions props and it uses them to perform data manipulations based on user interactions as shown in Figure 36.

The code for the MyActivitiesTable component uses the activities, onEditActivity, and onDeleteActivity props inside additional functions that are called when certain user interactions

take place. Figure 39 below shows the structure of the code that is being returned by the MyActivitiesTable component essentially, conditionally checking the length of the activities in that the user has created if it is greater than 0 which means the user has indeed created activity at least 1, then map over the activities creating a table row for each activity and populate each cell with the relevant data from the activity using dot notation. If the user has not created an activity, then we display a table row with text saying No activities found.

```

return (
  <div className="table-responsive">
    <h4>{activities.length} Activities </h4>
    <Table striped bordered hover>
      <thead>
        <tr>
          <th>Sport</th>
          <th>Date</th>
          <th>Title</th>
          <th>Time</th>
          <th>Distance</th>
          <th>Elevation</th>
        </tr>
      </thead>
      <tbody>
        {activities && activities.length > 0 ? (
          activities.map((activity, index) => [
            <React.Fragment key={index}>
              <tr>
                <td>{activity.sport}</td>
                <td>{activity.date}</td>
                <td><Link to={`/home/activity/${activity.id}`} className="link-of-unique-activity">{activity.name}</Link></td>
                <td>{activity.time}</td>
                <td>{activity.distance}</td>
                <td>{activity.elevation}</td>
                <td className="activities-table-buttons">
                  {editIndex === index ? (
                    <</>
                  ) : (
                    <</>
                    <Button variant="link" className="link-of-unique-activity" onClick={() => handleEditClick(index)}>Edit</Button>
                    <Button variant="link" className="link-of-unique-activity" onClick={() => handleDeleteClick(index)}>Delete</Button>
                  </>
                </td>
              </tr>
            </React.Fragment>
          ) : (
            <tr>
              <td colspan="3">No activities found</td>
            </tr>
          )
        )}
      </tbody>
    </Table>
  </div>
)

```

Figure 39: MyActivitiesTable JSX return. Screenshot by Fortune Ndlovu (2024)

Figure 39 also shows innermost conditions that are true equal to an index of the table row if so, this means the user has clicked the edit button, so we display the EditableRow component shown in Figure 36 above. Else we show the two buttons Edit and Delete. The EditableRow component is essentially a component with input field for typing in text we want to use as the edited text and

also contains buttons such as Save for saving the newly entered data and Cancel for removing the EditTableRow component and the Edit more options link for directing the user to the EditActivityForm component shown back in Figure 31 with more initial edit inputs.

The EditTableRow get passed props that are functions such as editedActivity which is the data, handleSaveClick for saving the edited data, handleCancelClick for removing the EditableRow component, handleInputChange for capturing whatever the user types using the onChange event handler, handleSportChange for the selected select input and all of these functions are shown below in figure 40.

```
const MyActivitiesTable = ({
  activities,
  onEditActivity,
  onDeleteActivity,
}) => {
  const [editIndex, setEditIndex] = useState(null);
  const [editedActivity, setEditedActivity] = useState({});

  const handleEditClick = (index) => {
    setEditIndex(index);
    setEditedActivity({ ...activities[index] });
  };

  const handleSaveClick = (index) => {
    onEditActivity(index, editedActivity);
    setEditIndex(null);
    setEditedActivity({});
  };

  const handleCancelClick = () => {
    setEditIndex(null);
    setEditedActivity({});
  };

  const handleInputChange = (key, value) => {
    setEditedActivity((prev) => ({ ...prev, [key]: value }));
  };

  const handleDeleteClick = (index) => {
    onDeleteActivity(index);
  };

  const handleSportChange = (sport) => {
    setEditedActivity((prev) => ({ ...prev, sport }));
  };
};
```

Figure 40: MyActivitiesTable Event Functions. Screenshot by Fortune Ndlovu (2024)

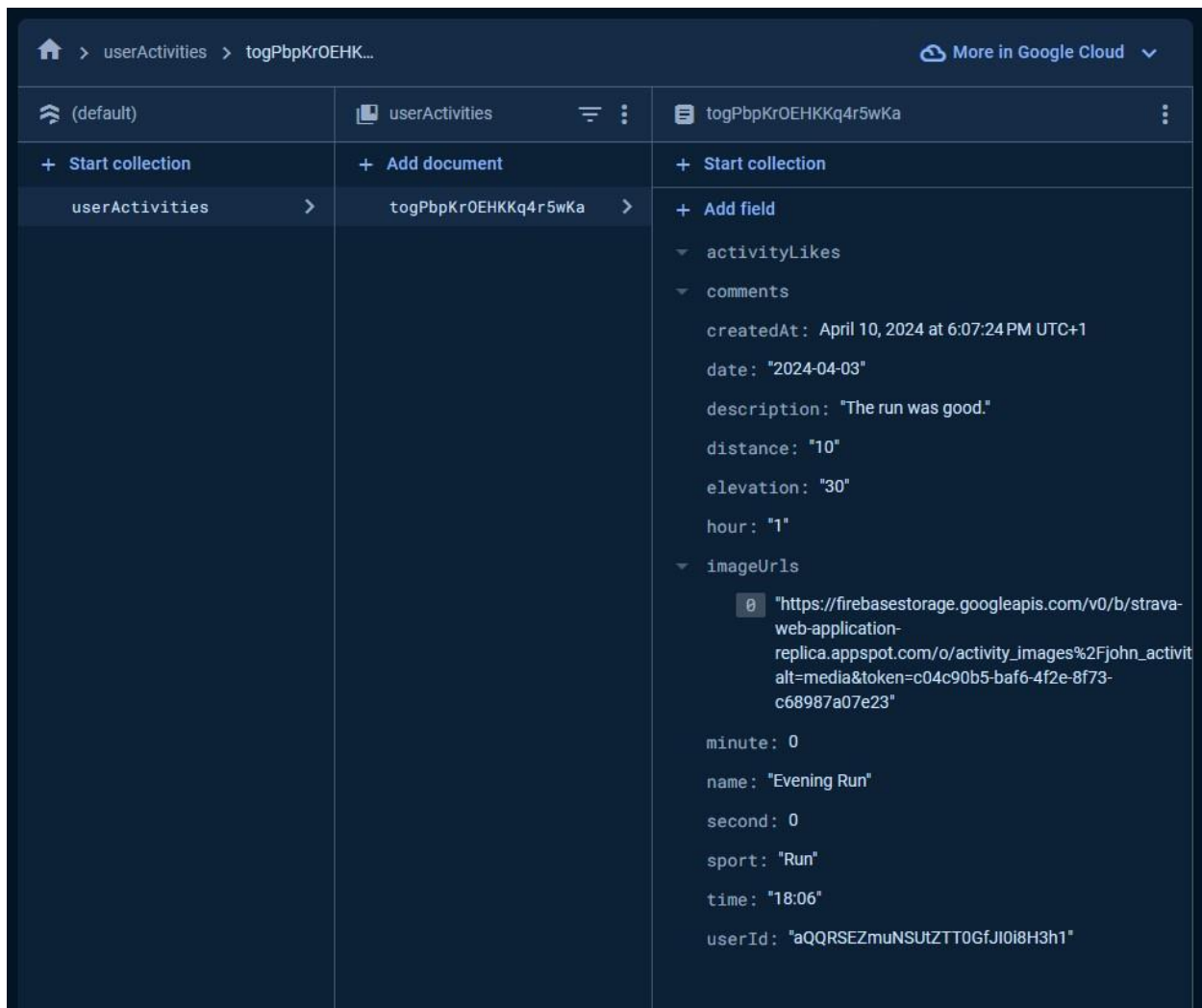


Figure 41: Firestore UserActivities Collection first document. Screenshot by Fortune Ndlovu (2024)

Figure 40 shows all the functions used in the MyActivitiesTable component. Starting at the top I created two state variables editIndex and editedActivity, editIndex is to track if the edit id has been clicked and we are in the correct index that's why for the rest of the methods editIndex is set to null, and it is also for conditional rendering the edit and delete buttons with the EditTableRow components. The editedActivity state variable is used to store the edited data that is finalized and click the Save button is true.

The handleEditClick function is called by the Edit button, it updates the editedActivity state with new data, so we are not editing the older data for immutability and garbage collection reasons. The handleSaveClick function is called when the user has added their edits then we update the editedActivity state with edited data and we pass the data to the onEditActivity prop so it can be updated to the database. The handleCancel function assigns null to the editIndex that means we

haven't clicked the button and therefore we don't want to render the EditTableRow as of now. The handleInputChange function is used to get the value of what is typed. For handleDeleteClick we call the onDeleteActivity to delete the document from the database. The handleSportChange is for changing the value of the selected dropdown by using its previous value and updating it with the new value. Figure 41 shows what happens in Firestore when the data is populated into the document unit of the userActivities collection storage.

4.2.2.4. Image Interactions Implementations

```
// func triggered when a file is selected using the file input
const handleImageChange = async (e) => {
  // converting selected files into an array
  const files = Array.from(e.target.files);

  // Compress each image before uploading
  const compressedImages = await Promise.all(
    files.map(async (file) => await compressImage(file))
  );

  // adding compressed images to the existing images state array
  setNewImages([...newImages, ...compressedImages]);
};
```

Figure 42: handleImageChange function. Screenshot by Fortune Ndlovu (2024)

In the ManualEntryForm component when the user clicks the image input to add an image with their activity a few things happen. Firstly, we call a handleImageChange function on that onClick event. This function is shown in Figure 42 just in case the user has selected more than one file, we convert them into an array and map over the array and compress each image. After all the images are compressed, we update the state with the combined new arrays of both the new Images and the compressed images because this ensures we are not overriding the previous state.

```

const handleCreateActivity = async () => {
  let imageUrls = []; // Use an array for multiple images

  try {
    // Upload multiple images to Firebase Storage
    await Promise.all(
      // Iterating over each compressed image in newImages
      newImages.map(async (file) => {
        const storage = getStorage(app);
        const storageRef = ref(storage, `activity_images/${file.name}`);

        // upload the selected images to Firebase Storage
        await uploadBytes(storageRef, file);

        // Retrieving the download URL and adding it to the imageUrls array
        const imageUrl = await getDownloadURL(storageRef);
        imageUrls.push(imageUrl);
      })
    );
  }
};

```

Figure 43: handleCreateActivity function. Screenshot by Fortune Ndlovu (2024)

At this point, the user has added their images and we have compressed them. Now when the user clicks the create orange button, that represents the creation of the entire activity, a handleCreateActivity function is called as shown in Figure 43. The goal of this function is to upload the images to firebase storage and then retrieve the image URLs, by downloading the URLs and pushing them into an array which is used as the value of the imageUrls property within the overall createdActivity object shown back in Figure 28. This object represents the data added to Firestore and as shown also in Figure 41, the imageUrls property within the document is in fact a URL representation of the image taken from Firebase Storage.

Looking closely at this operation of initially adding images to Firebase storage (Figure 43), we first create an imageUrls array to store all the images and then a try block to allow me to handle errors gracefully for the asynchronous operations that could potentially fail due to network issues and any other environmental factors. Inside the try block I use a Promise.all method to handle multiple asynchronous operations concurrently, this means that it will take an array of promises and returns a single promise that resolves when all the promises in the array have resolved or rejects if any of the promises reject. The mapping of the images creates an array of promises.

The process of each promise representing the uploading a single image, involves getting the storage from Firebase of my web app and creating a reference for the given URL, this means we create a folder called `activity_images` if it has not yet been created and we include the name of the image in question. Moreover, we upload the image to that specified location, using the `uploadBytes` method. Once the image(s) have been uploaded I get the downloaded URL of the given storage reference and push that URL into the `imagesUrls` array which is used as a value for the object passed to the `onCreateActivity` method.

I have shared the process of how I was able to upload the images to Firebase storage then providing the Firebase, Firestore database with the reference downloaded URL as part of the overall creation of the activity in question for data integrity. Next, I want to share how I was able to add and delete the images in the `EditActivityForm` component.

```
// func triggered when the user selects an image file using the file input
const handleFileInputChange = async (e) => {
  // check if the file is selected
  const file = e.target.files[0];
  if (file) {
    try {
      // Compress the image before adding it to the state
      const compressedImage = await compressImage(file);
      // Converting compressed image to a data URL
      const reader = new FileReader();
      // Updating the state with the new image URL
      reader.onloadend = () => {
        handleAddImage(reader.result);
      };
      reader.readAsDataURL(compressedImage);
    } catch (error) {
      console.error("Error compressing image:", error);
    }
  }
};
```

Figure 44: `handleFileInputChange` function. Screenshot by Fortune Ndlovu (2024)

In the case of the user wanting to add another image to their activity, when the image input is clicked the `handleFileInputChange` function is called to capture the event made by the user. The function checks if the user has selected a file and if they have selected a file it compresses the file and converts the compressed image to a data URL to help with the loading of the image as

we as asynchronously allowing the web app read the content of the file, and what we are passing to the `handleAddImage` function as the result being the loaded image.

```
// handles the adding of images by taking the newImageUrl as an argument
const handleAddImage = (newImageUrl) => {
  // Ensuring that the state update is based on the previous state with the new array of image URLs
  setEditedActivity((prev) => ({
    ...prev,
    // Creating a new array with the existing image URLs and appending the new imageUrl to it
    imageUrls: [...prev.imageUrls, newImageUrl],
  }));
};
```

Figure 45: `handleAddImage` function. Screenshot by Fortune Ndlovu (2024)

Figure 45 shows the `handleAddImage` function used to add the images into the `imageUrls` array. The function takes in the argument which was the image and using the state function we are creating a new array of the entire previous state and ensuring the new array has the newly added image.

```
const [editedActivity, setEditedActivity] = useState({
  name: "",
  description: "",
  sport: "",
  // ... other fields
  imageUrls: [], // Array to store image URLs
});
```

Figure 46: `editedActivityState`. Screenshot by Fortune Ndlovu (2024)

Figure 46 shows the state variable that is used to update the relevant document in Firestore. The state variable is assigned an object, and a property of this object is the `imageUrls` with a value being an array that is to hold all the images added by the user.

```

{editedActivity.imageUrls.map((imageUrl, index) => (
  <div key={index}>
    <img
      src={imageUrl}
      alt={`Activity ${index + 1}`}
      width={100}
      height={100}
      loading="lazy"
      style={{
        objectFit: "cover",
        marginRight: "5px",
        marginBottom: "10px",
      }}
    />
    <Button
      type="button"
      variant="dark"
      onClick={() => handleRemoveImage(index)}
    >
      Remove
    </Button>
  </div>
)}}

```

Figure 47: imageUrls mapping. Screenshot by Fortune Ndlovu (2024)

When the user add their image I also display the image they have added on the UI by mapping over the imageUrls array and for each image I am displaying the image with a button associated with that image that means they both share the same index (shown in Figure 47) this is important because the button has an onClick event that when clicked calls at the handleRemoveImage function passing the index and event of where the event took place.

```

// Handling the removal of an image by taking the index of the image to be removed as a parameter
const handleRemoveImage = (index) => {
  // Ensuring that the state update is based on the previous state
  setEditedActivity((prev) => {
    // Creating a copy of the current imageUrls array
    const newImageUrls = [...prev.imageUrls];
    // Removing the image at the specified index
    newImageUrls.splice(index, 1);
    // Returning new object with the updated imageUrls array effectively removing the desired image from the state
    return {
      ...prev,
      imageUrls: newImageUrls,
    };
  });
};

```

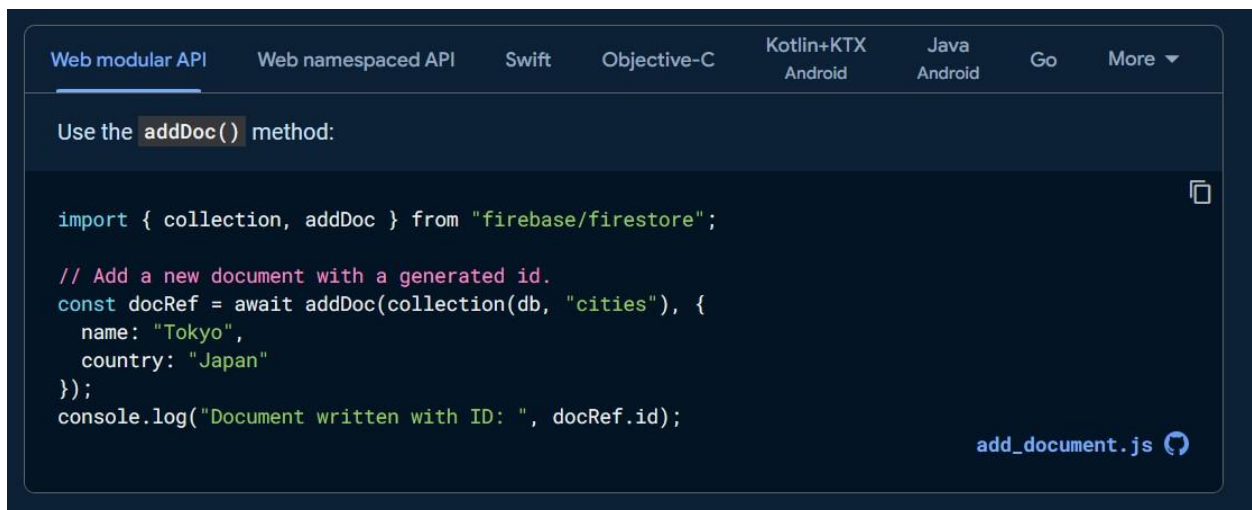
Figure 48: handleRemoveImage. Screenshot by Fortune Ndlovu (2024)

Figure 48 displays the `handleRemoveImage` function that takes in the index of the event that was evoked to remove the image as an argument. It then updates the `editedActivity` state with its `setEditedActivity` state function by creating a copy of the `imageUrls` array and removing the image at the specified index using the `splice` method often used to remove elements from an array and if necessary, inserts new elements in their place. The `handleRemoveImage` is returning a new object with the updated `imageUrls` property that now has the value of `newImageUrls` that is a new array containing images that were not deleted.

At this point, the user has added and removed their images and now when they click the Save button, a method is called named `handleSaveChanges` it is shown back in Figure 33, it essentially updates the document in Firestore with the edited object, then navigates the user back to `ActivityDetails` page where their edited changes are displayed. The next section of my report will share the complexities and solutions I encountered in the CRUD process.

4.2.2.5. Complexities and Solutions

The first complexity I encountered was creating the `createActivity` function in the `UserActivitiesManager` functional component. As shown previously on Figure 24, the `createActivity` function when it is called expects an object as an argument through its parameter and it essentially adds this object that has the user's data into the `userActivities` collection as a document. Therefore, I used the `addDoc` method that Firestore provides for adding documents into collections and it also provides an auto generated id for each document that it creates, as shown in Figure 49.



The screenshot shows the Firestore documentation for the `addDoc()` method. At the top, there are navigation tabs for different languages: Web modular API, Web namespaced API, Swift, Objective-C, Kotlin+KTX Android, Java Android, Go, and More. Below the tabs, it says "Use the `addDoc()` method:". The code snippet is as follows:

```
import { collection, addDoc } from "firebase/firestore";

// Add a new document with a generated id.
const docRef = await addDoc(collection(db, "cities"), {
  name: "Tokyo",
  country: "Japan"
});
console.log("Document written with ID: ", docRef.id);
```

The file name `add_document.js` is visible in the bottom right corner of the code editor.

Figure 49: Firestore Documentation `addDoc` method. Screenshot by Fortune Ndlovu (2024)

Figure 49 shows documentation code snippets from the Firestore section part of Firebase (Firebase, 2024), the code snippet clearly outlines that to add a document into your collection you need to pass an object of the user data into your specified collection name and with reference to the database. The complexity for me was I wanted to have the object be a dynamic argument because the data that will be passed is initially unknown.

The solution for this was, when I created my `createActivity` function include a parameter and apply logic to add the document inside the `userActivities` collection using the `addDoc` method but instead of including an object directly, I included the parameter as a spread operator shown in Figure 24, this was before I add into the database it is a new object each time and this improve immutability and data integrity. Moreover, now that the document has been added into the relevant collection using the `addDoc` method that auto-generated an id for each document, I made the `createActivity`

function return a new object and the id of the new object after it had been added to the database because once it is in Firestore we can use its id to get reference to the document in question. The data comes from the prop within the ManualEntryComponent and it used as an argument essentially passing the data as an object to the createActivity function, while in the Firebase reference they directly attached the object of the users data for demonstration purposes.

The second complexity I encountered was creating functions to edit and delete activities. This was complex because I was having issues with my database in Firebase crashing because there were too many requests being sent to the database and on the Freemium plan the storage capacity could not handle the vast number of requests and provide the necessary responses at rate.

To fix this the solution was to not update my code from the standard way of updating documents using the updateDoc or deleteDoc methods. But instead, I used batches because they decreased the number of requests and responses between the client side and the server by collectively grouping the data and sending them as a unit.

The third complexity was images. When images were being uploaded into my storage service within the Firebase database, they were initially too large and this was slowing my web app, causing my Firebase to overload too quickly due to its limited storage capacity. Large images not only consume significant storage space but also can slow down the server response time negatively impacting the performance of the entire application. It can also lead to bandwidth usage, which can be a concern in environments with limited or metered connections, (Faruq, Y. 2022).

The solution to fixing these large images on the server problem was to essentially compress the images before they are uploaded to Firebase. Compressing images takes up less storage space due to the smaller file size of the image, this means faster uploads improving the user experience for users with slower internet connections.

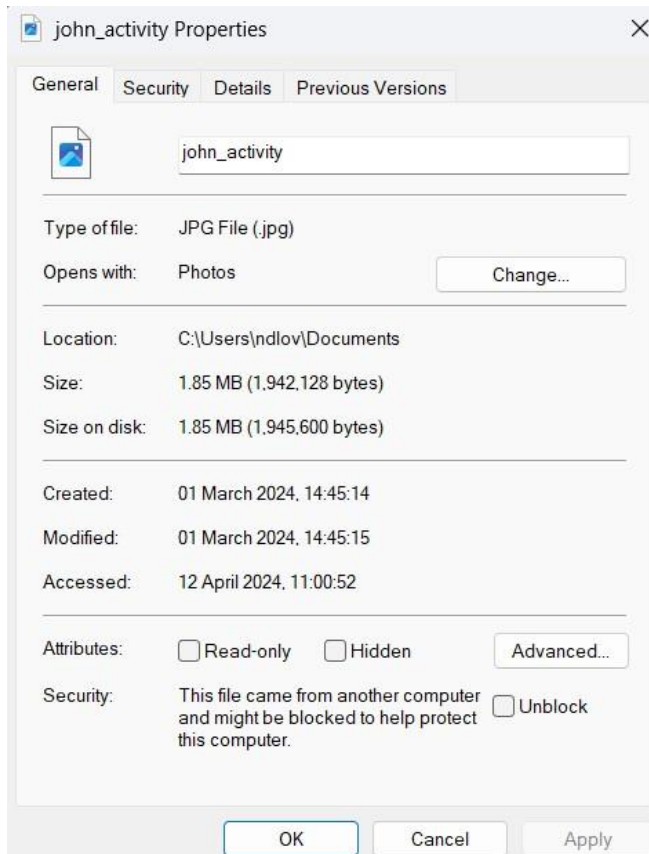


Figure 50: Image Properties File Explore inspection example. Screenshot by Fortune Ndlovu (2024)

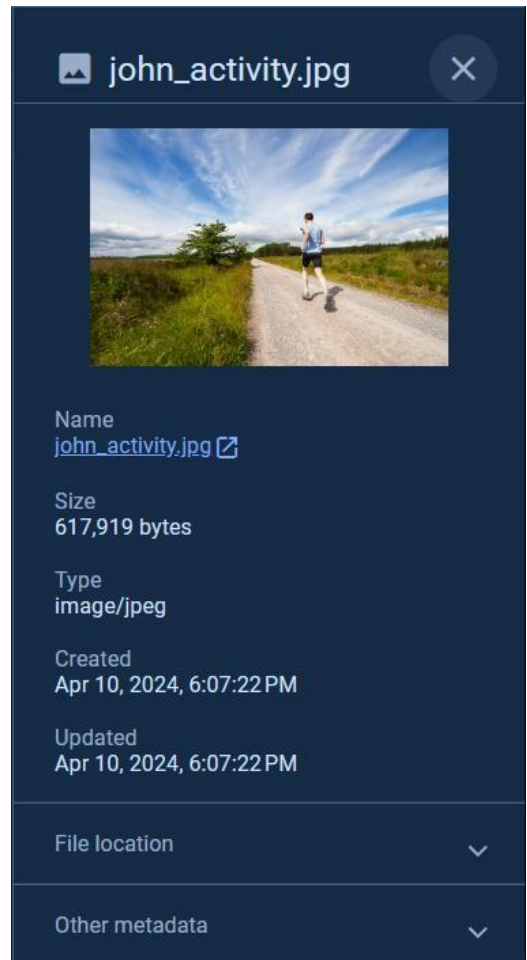


Figure 51: Image Properties Firebase Storage inspection example. Screenshot by Fortune Ndlovu (2024)

Figure 50 shows an example of the size of the original image on my file explorer before it was uploaded to Firestore storage as a compressed file. The original image is 1,942,128 bytes.

Figure 51 shows the example of the image size after it had been uploaded to Firestore Storage with a file size of 617,919 bytes. This means the reduction was 1,324,209 bytes because 1,942,128 bytes (original size) - 617,919 bytes (compressed size) = 1,324,209 bytes. This significant reduction saves storage space and improves web app performance.

```

// Utilizing browser-image-compression library's compression functionality
import imageCompression from "browser-image-compression";

const compressImage = async (file) => {
  try {
    const options = {
      maxSizeMB: 1, // Maximum size of the compressed image
      maxWidthOrHeight: 800, // Maximum width or height of the compressed image
    };

    // Compress the image using the provided options
    const compressedFile = await imageCompression(file, options);

    // return the compressed file
    return compressedFile;
  } catch (error) {
    console.error("Error compressing image:", error);
    throw error;
  }
};

export default compressImage;

```

Figure 52: Browser Image Compression Code. Screenshot by Fortune Ndlovu (2024)

Figure 52 shows the code used to compress the image file before being uploaded to Firestore Storage. I used the browser-image-compression library because its purpose is exactly to compress images before being uploaded to servers (npm, 2023). When this functional component named compressImage is called it will compress the image by using the values defined in the options object and then returns the compressed image, the component also logs out any error to the console and then rethrown to be handled by the handling code, during the compression process.

The options object is interesting because the process of how its properties maxSizeMB and maxWidthOrHeight would affect the compression of an image is a combination of both properties. maxSizeMB sets the maximum for the compressed image, in this case it's set to 1 MB. The browser-image-compression library will attempt to compress to fit within this size limit. If the original image was 1,942,128 bytes (approximately 1.85 MB), the library would compress the image to reduce its size to under 1 MB. maxWidthOrHeight sets the maximum width or height (in pixels) for the compressed image. The library will scale down the image to fit within these dimensions while maintaining the aspect ratio of the original image (npm, 2023). In the next section of my report, I will share the lessons learned throughout the overall experience of activity management.

4.2.3. Lessons Learned

4.2.3.1. CRUD Operations with HTTP Requests

I learned about what was actually happening under the hood when creating CRUD operations. Essentially, I was using HTTP Request with Firestore's REST API which translates these HTTP requests into internal database operations, ensuring data integrity, security and compliance with Firestore's rules and constraints. Each CRUD operation corresponds to specific HTTP methods (POST, GET, PATCH/PUT, DELETE) and follows the principles of RESTful API design for data manipulation.

When I was adding a document into Firestore I was using the `addDoc` method which in Firestore's SDK corresponds to a HTTP POST request. When I called `addDoc`, Firebase internally constructs HTTP POST requests to Firestore's REST API endpoint. This request contains data I wanted to add. Firestore then processes this request, performs validations, generates a unique document ID if needed, and adds the document to the specified location. The response is typically a success status code usually 201 created along with the newly created document's ID or metadata.

When I was getting my documents from Firestore so I can display the data in my UI, I was using Firestore's `getDoc` method which constructs a HTTP GET request to Firestore's REST API endpoint, specifying the document or collection to retrieve. Firestore processes the request, retrieves the requested data and sends it back to the response body. If the data does not exist or there is permission issues Firestore responds accordingly with the appropriate status code usually 404 Not Found or 401 Unauthorized.

When I was updating my document in Firestore I used the `update` method which constructs either a HTTP PATCH or PUT request to Firestore's REST API endpoint. For the HTTP PATCH requests only the specified fields are updated with the new values, maintaining existing data. For the HTTP PUT requests the entire document is replaced with the new data provided. Firestore processes these requests, identifies the document to update, applies the changes and responds with a success status code upon completion usually 200 OK allowing with any updated metadata.

When I was deleting a document, I used the `delete` method and this means Firebase initially constructs a HTTP DELETE request to Firestore's REST API endpoint, specifying the document

to delete. Firestore processes this request, removes the specified document from the collection and responds with a success status code to indicate successful deletion typically 204 No Content indication that the operation was successful.

I truly learned to use asynchronous functions efficiently when making HTTP requests such as CRUD operations because they ensured my web app remained responsive, while handling errors effectively. Asynchronous functions ensured that my code did not block the execution of other tasks while waiting for a response from the server. HTTP requests involve network communication, which is inherently asynchronous, therefore it was logical for me to use async functions to handle the asynchronous operations in a structured manner as it ensures that other parts of my web app can continue running while the HTTP request is in progress. In the next section of my report, I will share another element that I learned which was the coding pattern.

4.2.3.2. The Coding Pattern

After implementing the CRUD operations in React, I realized the process of how I was coding, what I was coding, how I was coding, and why I was coding the code in such a structured way, it was at this moment where it all made sense to me. As it turns out there is a pattern to coding in React at least for my case when implementing the web app. At a basic level there are typically three steps to this pattern. Firstly, we begin by implementing the functional component and having knowledge of its intentions, its purpose essentially what it is meant to return. This is often an event invoked by the user or mathematical evaluation, which then triggers or calls such a function to operate the operations of the intended execution required by the event invoked or mathematical evaluation. Sometimes the functional component can get too large increasing complexity therefore we create sub components and import them into the main component and to insure the synchronization with the operations of the intended execution between all the components, the evaluation invoked by user event trigger or mathematical evaluations are passed via properties.

Secondly, this function typically takes in the argument and performs the logical operations, depending on its intentions, it will have its own functions, with their own temporary variables and conditions applied but the overall function will typically return or update a value to the global use state variable using such a function. So now we have gone from the event taking place to the

execution of the function to operate the event's intentions which then updated the overall functional component with evaluation.

Thirdly, this global state variable can now be used to conditionally check if it contains such a value by applying relevant conditions to it, or it can be used to display the evaluated data it contains as part of the viewport or screen real estate components.

This is the three-step process that I realized I was doing over and over again. I was essentially updating state variables logically and manipulating their data using functions and methods respectively and if my component was too large, I would break it down into smaller components and pass the data and props that way the data was known by the relevant components. Overall, having knowledge of this process allowed me to understand my code and debug more effectively, by working backwards to figure out what was the value meant to be in the relevant state variable and what value did we get instead and where did we go wrong. Working backwards to debug meant I was following the functions and method evaluation operations to figure out where the execution went wrong. Typically, it was the functions that were really important to check because they direct the flow and evaluation of the data, and the variables simply hold the values for later usage.

I learned a lot from creating the CRUD operations in my Strava Replica web app. At this point, the user can create, read, update and delete their activities in a clean and intuitive UX/UI. The next section of my report shares how I was able to take my web app to the next level of application by applying additional complexities involving multiple user integration and interactions within the web app.

4.3. Multiple User Integration

After the user was able to create, read, update and delete (CRUD) activities within the web app, I added a layer of complexity which was having the ability to have multiple users within the web app. I knew having multiple users would mean that each user would sign into the same web app but on different dynamic instances based on their unique identifier (UID). In this section of the report, I will write about how I was able to implement multiple users and have each user be able to operate CRUD operations, follow other user profiles, and interact with their activity uploads.

4.3.1. Multiple User Data Architecture

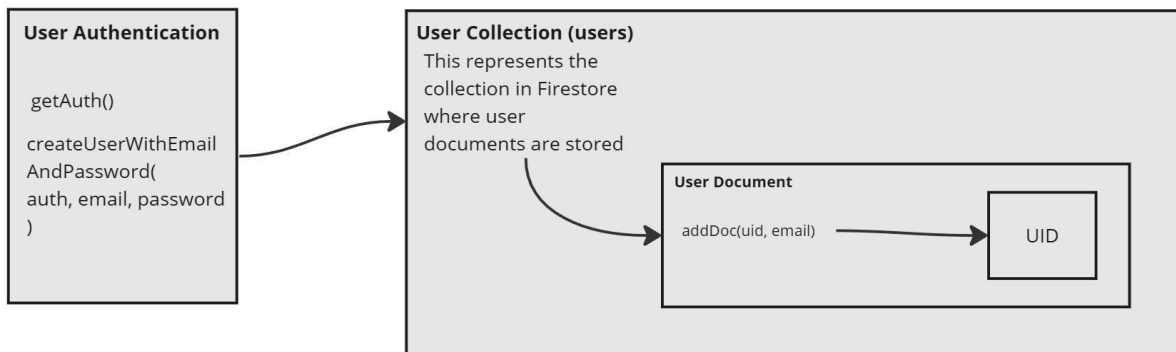


Figure 53: Multiple User Data Architecture. Screenshot by Fortune Ndlovu (2024)

Figure 53 shows the architectural design that provides a top view perspective of what the creation of multiple users looks like conceptually. When the user signs up to the web app with their email and password they are provided a unique identifier (UID) by the authentication instance. A user's collection is also created to hold documents, each document representing each user, that means when the user signs up a document is created for that user with initial data being their email they provided and their UID.

```

import { getAuth, createUserWithEmailAndPassword } from "firebase/auth";
import { db } from "../firebase/firebase";
import { collection, addDoc } from "firebase/firestore";

const authSignUpWithEmailAndPassword = async (email, password) => {
  const auth = getAuth();
  try {
    const userCredential = await createUserWithEmailAndPassword(auth, email, password);
    const user = userCredential.user;

    // Create a user document in Firestore
    await addDoc(collection(db, "users"), {
      uid: user.uid,
      email: user.email,
    });

    return { user, uid: user.uid };
  } catch (error) {
    return { error: error.message };
  }
};

export default authSignUpWithEmailAndPassword;

```

Figure 54: authSignUpWithEmailAndPassword function. Screenshot by Fortune Ndlovu (2024)

Figure 54 shows the function I used to create a new user by having them sign in with their email and password. I was able to call this function taking in their email and password as arguments. Then calling the getAuth instance and then using the auth, email and password as arguments passed to the createUserWithEmailAndPassword function to create the user account. Now when the user was created, their account details are added into the authentication service in my Firebase project therefore, I added a document in the Firestore database with the users UID and email, this document represents the user that has just signed in and the document is in the user's collection, if the users collection is not created, it will be created when we add the first document representing the first ever user to sign up.

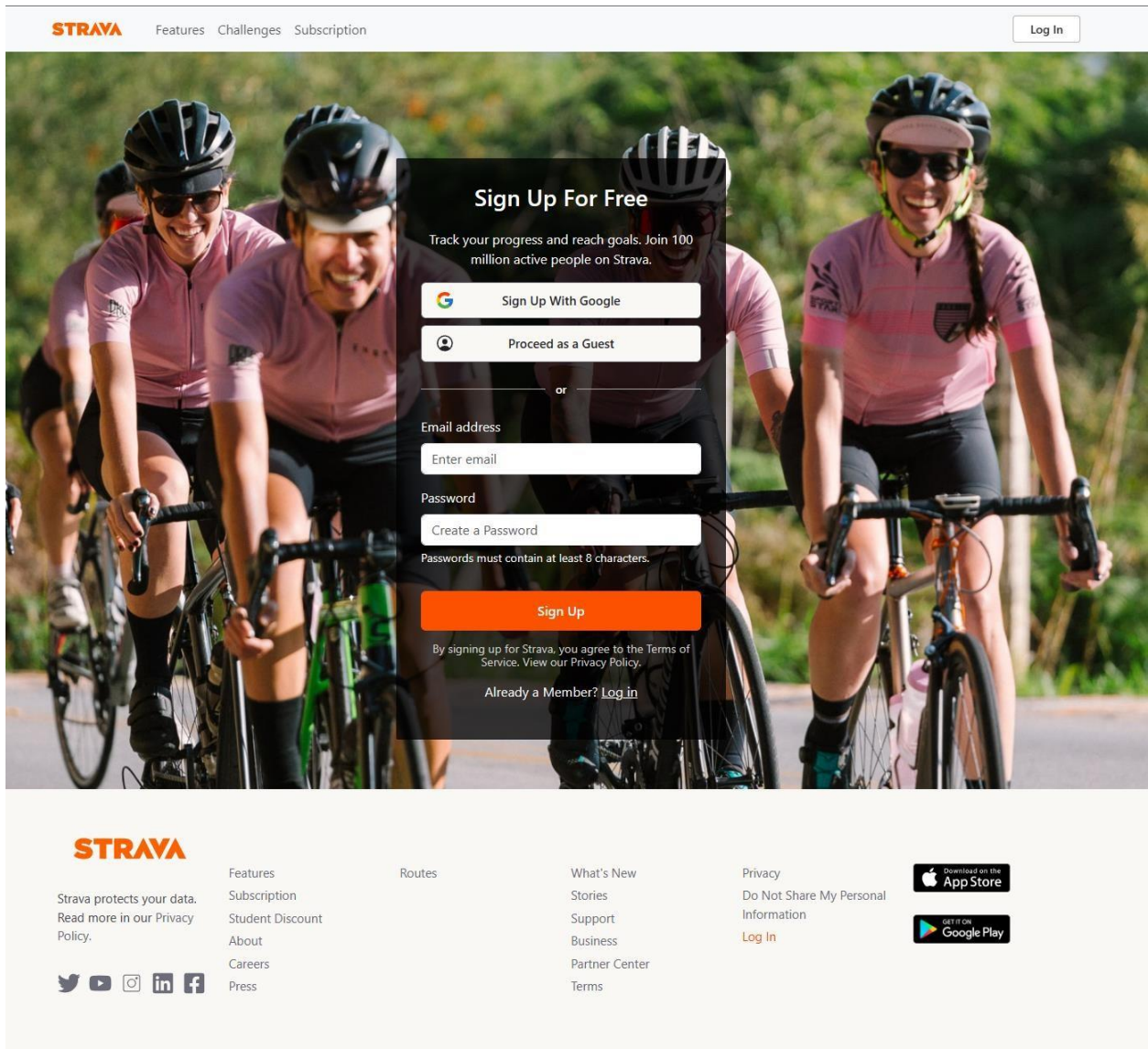


Figure 55: Sign Up Form UI. Screenshot by Fortune Ndlovu (2024)

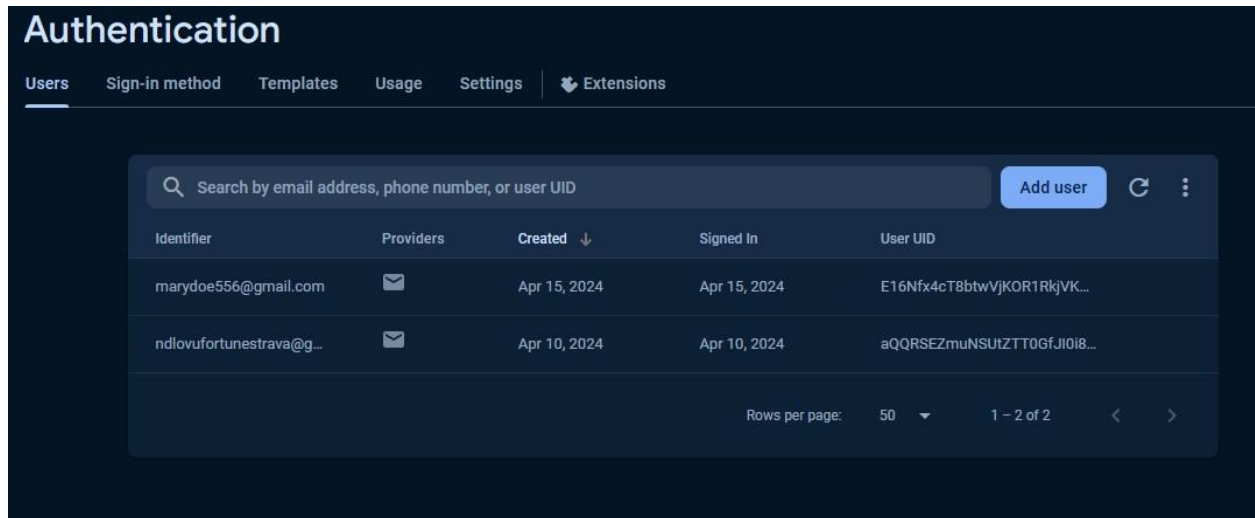


Figure 56: Firebase Authentication Dashboard. Screenshot by Fortune Ndlovu (2024)

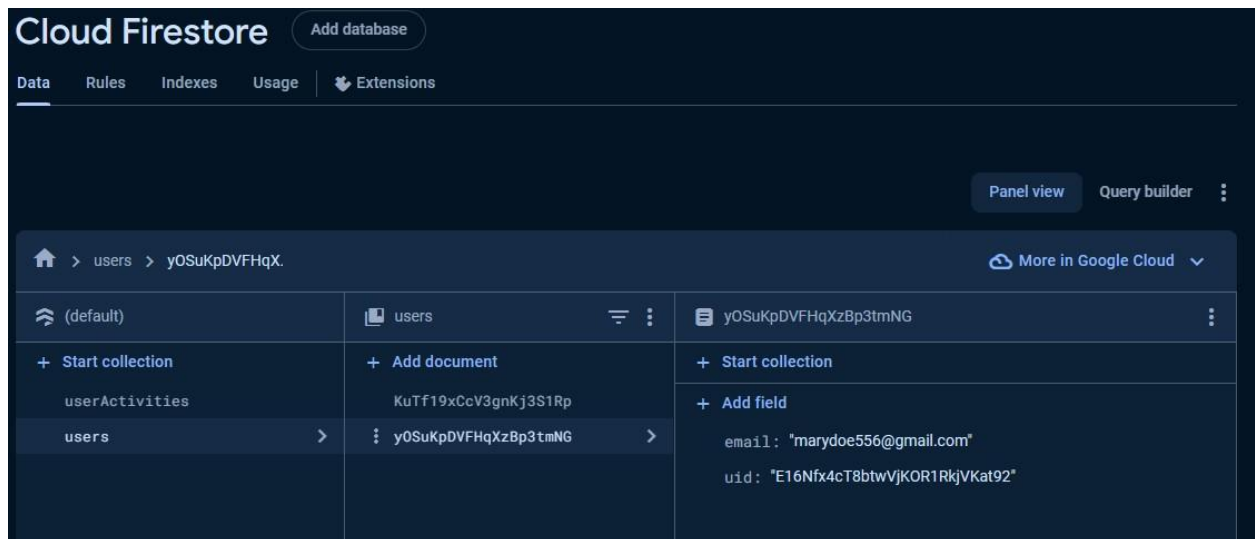


Figure 57: Firebase Cloud Firestore Dashboard. Screenshot by Fortune Ndlovu (2024)

Figure 55 shows the UI that I implemented using the same React iconography, similar font sizes and styling to maintain consistency. When the user types in their email and password these values are captured using the onChange event handler, onChange is used to handle changes to an input field, when there is a change it is sets an event and we capture that event as a value that we use to update the state variable, that we pass to the sign up function for the users account to get created. Once the user signs up they are directed to the home page (viewport similar to Figure 14) to provide an experience that they have successfully signed up. Figure 56 shows the Authentication dashboard in Firebase being updated when a user signs up and also in Figure 57 we notice that our Firestore database also updates with a document of the newly created user with their email and UID. When the user signs up for their account to be created in the

Authentication instance, I just happen to bring the users into the Firestore database to keep the data centralized and easier to manage as data gets more complex to handle. At the next section of the report, I talk about how I implemented the ability for the current signed in user to be able to see their own unique activities.

4.3.2. Current User Activities on the Dashboard

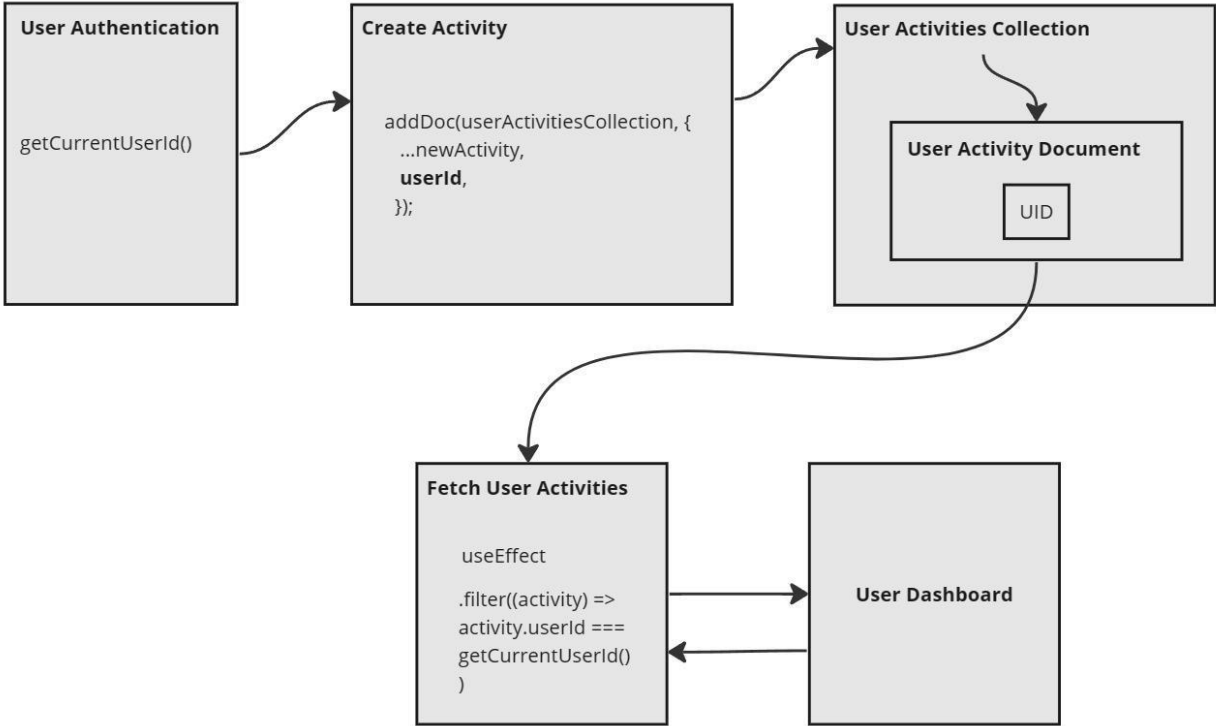


Figure 58: Current User Activities on the Dashboard Authentication. Screenshot by Fortune Ndlovu (2024)

Figure 58 shows my architecture that I used to have only the activities that the currently signed user created display on the dashboard creating a personalized experience. Since we have the ID of the user that is currently logged in from the authentication instance and we also have the user in the user's collection in firebase we can track every user based on the user id. When a user creates an activity before I upload the data of the object to Firestore I attach the user ID of the currently signed user being the user that created the activity, this means every time an activity is created a userId of the user that created it is attached as a property to that object. At this point all the documents representing activities in the userActivities collection have property with the value being the userId of the user that created that activity, next I fetch only the activities that have the

same userId as the currently signed in user's Id because this would mean we are display only the activities that the currently signed in user has created in the dashboard and each time an activity is added to Firestore the dashboard will updates respectively.

```
useEffect(() => {
  // Setting up a snapshot listener to track changes in the collection
  const unsubscribeActivitiesCollection = onSnapshot(
    collection(db, "userActivities"),
    (snapshot) => {
      // Update the state whenever there's a change in the collection
      // Filtering activities to show only those created by the current user
      const userActivities = snapshot.docs
        .map((doc) => ({ ...doc.data(), id: doc.id }))
        .filter((activity) => activity.userId === getCurrentUserId())
        .sort((a, b) => b.createdAt.toMillis() - a.createdAt.toMillis());

      // Clone the userActivities array
      const clonedUserActivities = [...userActivities];

      setActivities(clonedUserActivities);
    }
  );

  // Cleanup the listener when the component unmounts
  return () => unsubscribeActivitiesCollection();
}, []);
```

Figure 59: Tracking Changes in the userActivities Collection. Screenshot by Fortune Ndlovu (2024)

Figure 59 shows the process used to get the activities representing the currently signed in user using the useEffect hook because the code contains effectful code to the entire functional component. I used the onSnapshot function because it listens for changes in the collection and updates the state fast. Essentially within the userActivities collection I am mapping over each document and creating a new instance of the previous object for immutability purposes and filtering only the activities that have a property of userId that is strictly the same as the currently signed in user's id and lastly I happen to sort the activities based on the time the activities were created at this way the most recent activity appears at the top of the UI, lastly I update the state variable with the data of only the currently signed in user. This means every user that is signed in only sees their own data creating a personalized experience.

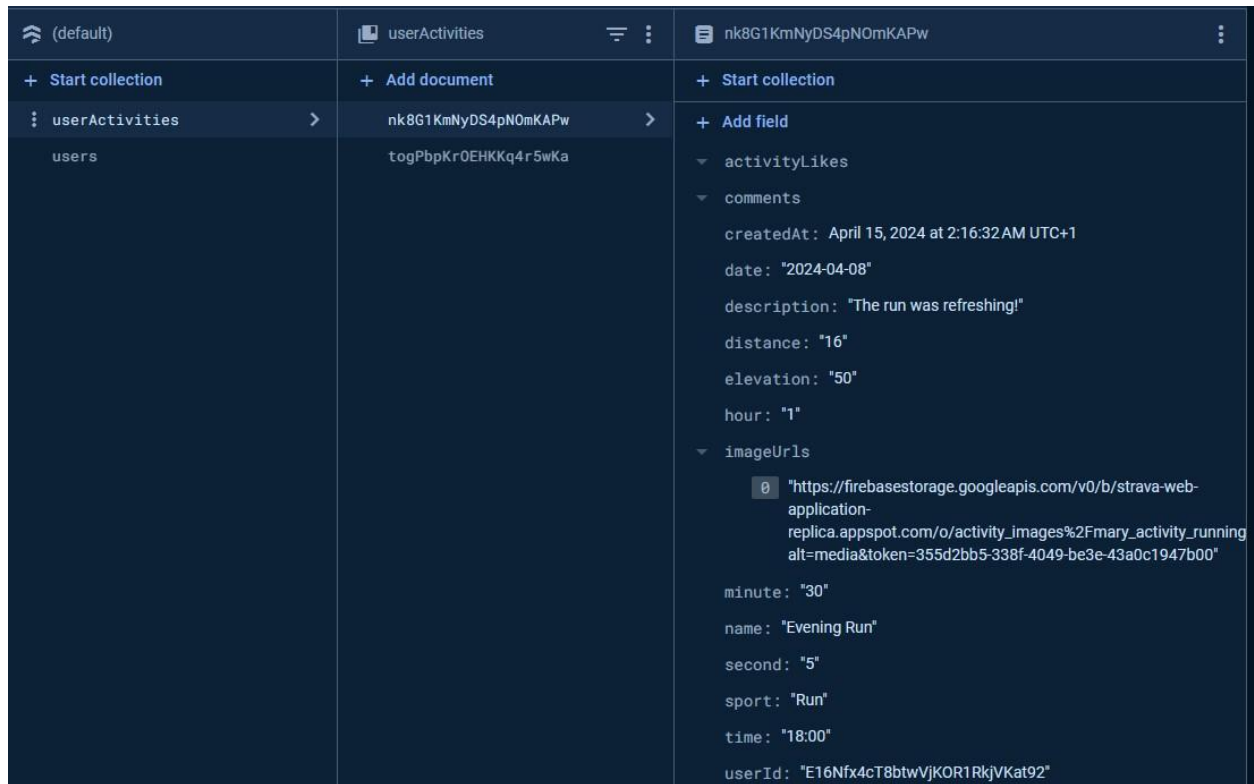


Figure 60: Firestore UserId in Document. Screenshot by Fortune Ndlovu (2024)

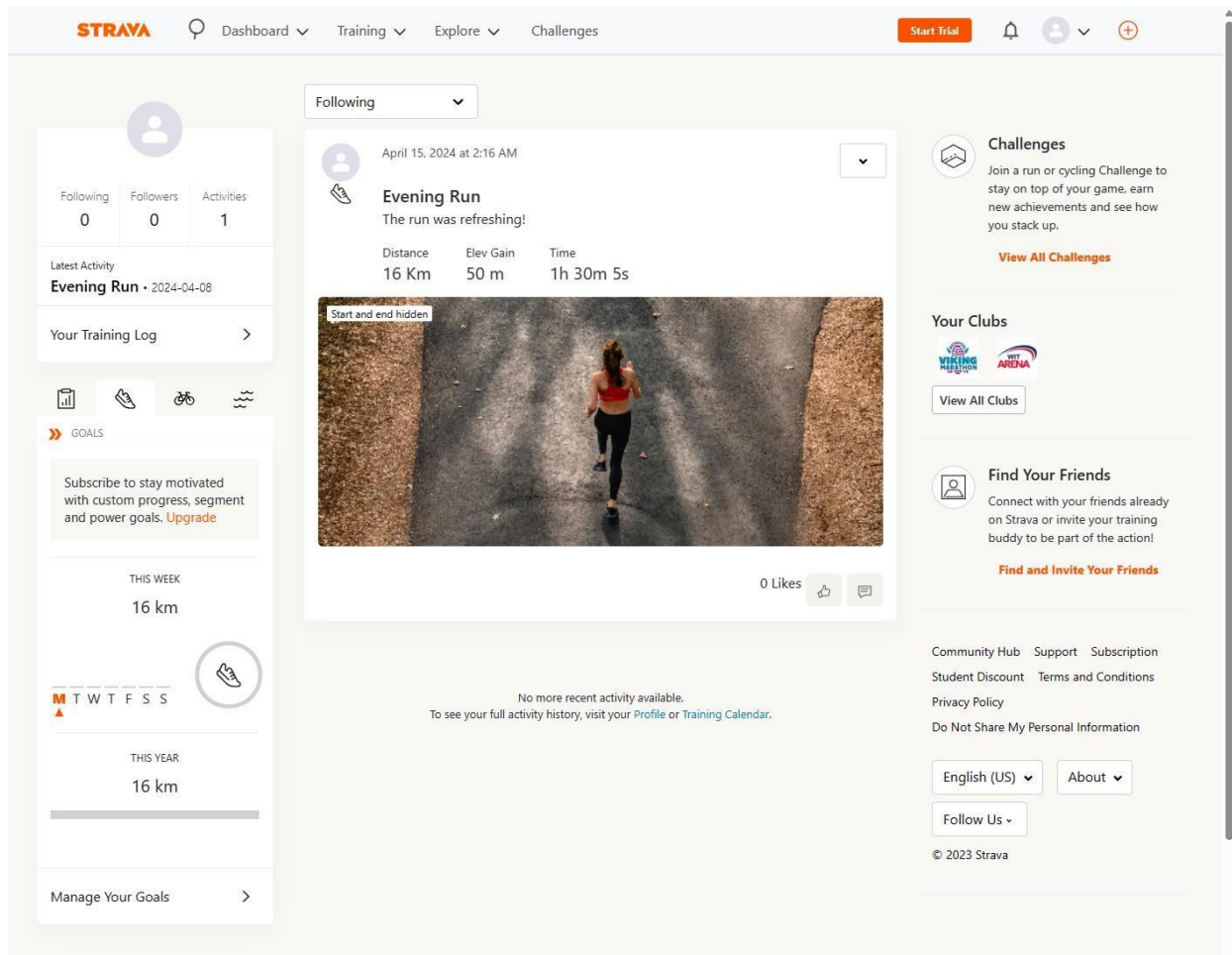


Figure 61: Currently Signed in users dashboard UI. Screenshot by Fortune Ndlovu (2024)

Figure 60 shows the Firestore dashboard with a newly added document meaning a user has created an activity and, in this document, a `userId` has been added with the value being the id of the user that created the activity and Figure 61 shows the user's data displaying respectively in the card UI of the dashboard. Therefore essentially, we have 2 collections in Firebase `users` and `userActivities`. The `users` collection is used to hold all the users that have signed up in the web app. Each user has their own document with their email and unique identifier provided by the authentication instance when their account was created. The `userActivities` collection holds the activities created by users, this means that each document represents an activity and each activity also has a property with the value being the `userId` of the user that created the activity, this makes it organic to display personalized dashboards because for each user that is signed in, I fetched only the activities that have the same `userId` as the currently signed in user, this way only the activities the user creates are displayed on their dashboard. The next section of the report will be about how I implemented the functionality of having the currently signed-in user follow other users.

4.3.3. Following User Profiles

As the currently signed-in user, you can follow other user profiles but only the users that have created an account in the web app because all we are doing essentially is fetching the specified data from the Firestore user's collection.

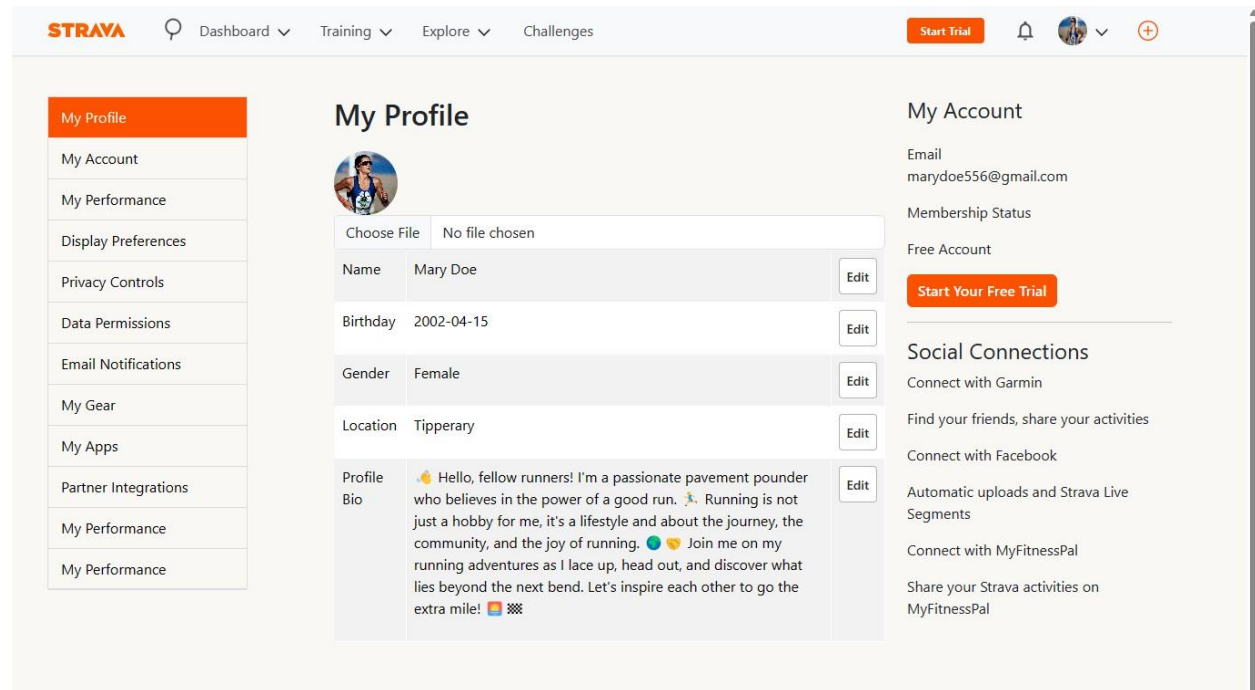


Figure 62: Currently Signed in users Profile UI. Screenshot by Fortune Ndlovu (2024)

Figure 62 displays a form that the currently signed-in user can fill in with their relevant information and this data is updated to the Firestore user's collection inside the current users document, now throughout the web app we can display the currently signed-in user's name and profile image creating a more personalized experience.

```

const handleSearch = async () => {
  try {
    const userQuery = query(collection(db, "users"), where("name", "==", searchQuery));
    const userSnapshot = await getDocs(userQuery);
    if (!userSnapshot.empty) {
      const userDoc = userSnapshot.docs[0];
      const userId = userDoc.data().uid;
      navigate(`/home/search/${userId}`);
    } else {
      console.log('User not found');
    }
  } catch (error) {
    console.error('Error searching for user:', error);
  }
}

```

Figure 63: handleSearch function. Screenshot by Fortune Ndlovu (2024)

Figure 63 shows the function that I wrote to be able to search for other users. This function is located in the SearchBar component because it is where the search bar is and is interacted with by the user when wanting to search for another user. The function essentially queries the user's collection to find a property of where the name is the same as the name that the user is searching for, if it is, we navigate to a dynamic URL of the user that we are searching for.

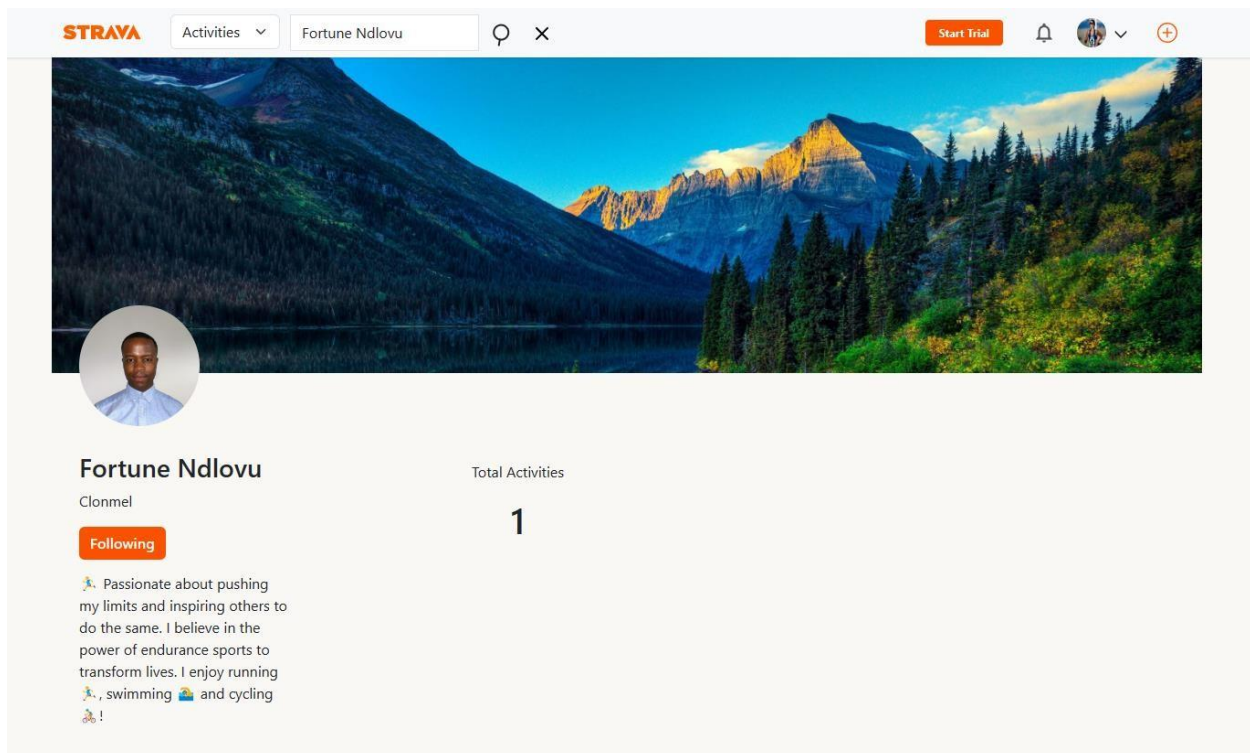


Figure 64: Searched Users Profile Page. Screenshot by Fortune Ndlovu (2024)

Figure 64, shows that as the currently signed-in user in this instance Mary Doe, we have searched for Fortune Ndlovu and because he is signed up in the web app we are navigated to their profile page and are able to follow them by clicking the orange button, and we can also see they only have 1 activity created. When the currently signed-in user follows another user, their userId is added to the user's collection document of the currently signed-in user.

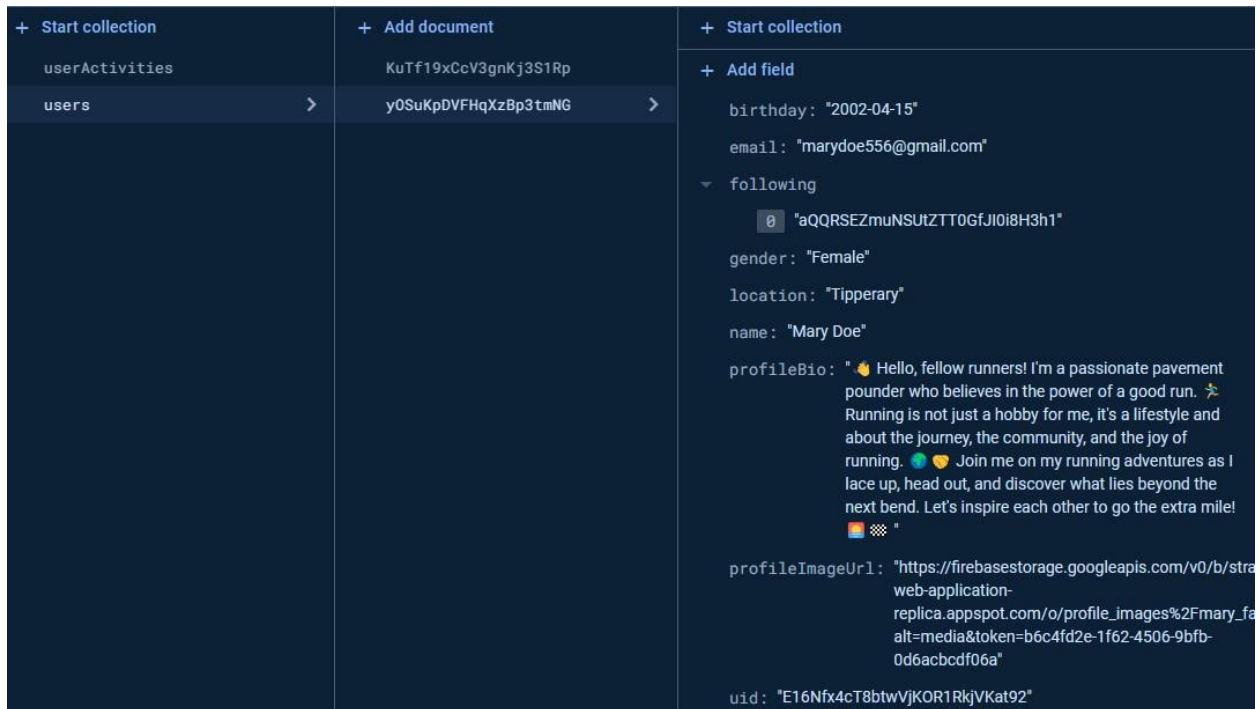


Figure 65: Users collection following array. Screenshot by Fortune Ndlovu (2024)

Figure 65 shows that Mary Doe followed Fortune Ndlovu and Fortune Ndlovu's UID was added into the following array for reference of the user she has followed.

```

if (!currentUserSnapshot.empty) {
  const userDoc = currentUserSnapshot.docs[0].ref; // Get the Docu
  const userData = currentUserSnapshot.docs[0].data();

  const followingArray = userData.following || [];

  if (!followingArray.includes(userId)) {
    // Follow user
    followingArray.push(userId);
    await updateDoc(userDoc, { following: followingArray });
  } else {
    // Unfollow user
    const updatedFollowingArray = followingArray.filter(
      (id) => id !== userId
    );
    await updateDoc(userDoc, {
      following: updatedFollowingArray,
    });
  }
} else {
  console.log("User document not found");
}

```

Figure 66: Following user and Unfollowing User Logic. Screenshot by Fortune Ndlovu (2024)

Figure 66 shows the logic of how I was able to implement the functionality of following and unfollowing users. Essentially creating a following array if it is not already created and checking if it does not include the userId of the user that we want to follow, if it doesn't, we push their userId into our array and update our document with the relevant data. To Unfollow we essentially create a new array by filtering out all the ids except for the id that we want to remove and update our document with the new array. The next section of my report will share how I was able to now also display the activities of the users that the current user follows on their dashboard.

4.3.4. Followers Activities in the Dashboard

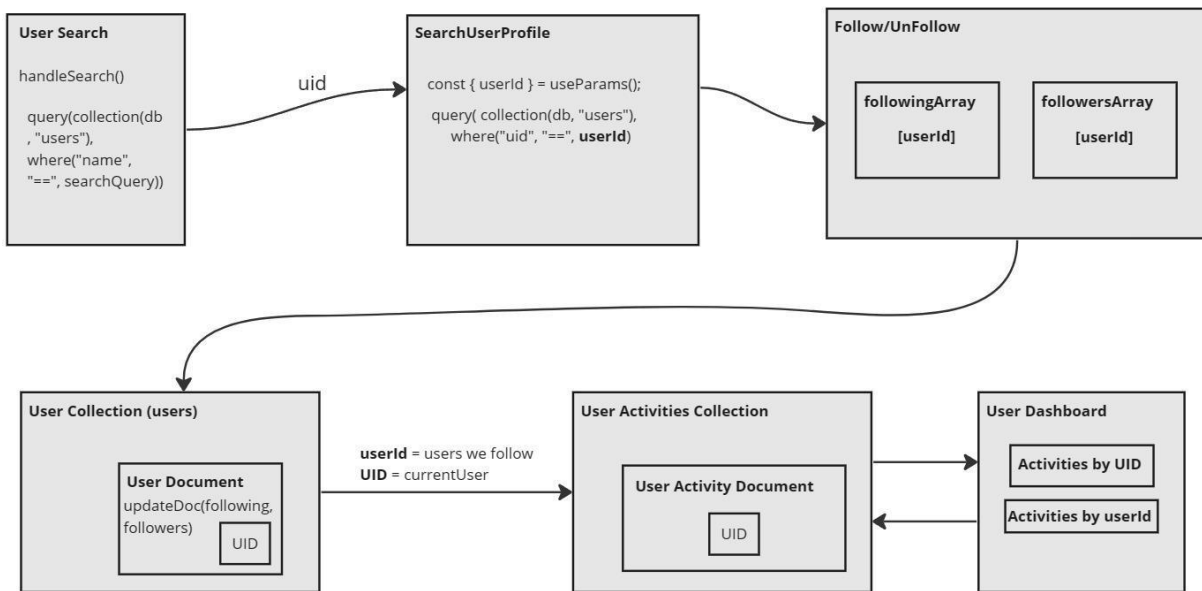


Figure 67: Followers Activities in the Dashboard. Screenshot by Fortune Ndlovu (2024)

Figure 67 shows the architecture I designed to illustrate the flow of data logic to have both the currently signed in user and the users that they follow's activities be displayed in the currently signed in user's dashboard. As of now as the currently signed in user we have searched for a user by name, and we were navigated to their profile page, and we followed them adding their id into our array that holds id of the users we follow. So now that we have the id of the user(s) we follow we can not only fetch the UID of the current user but also the userId of the user we follow in the userActivities collection and display the results in the dashboard.

```

if (!followingSnapshot.empty) {
  const followingData = followingSnapshot.docs[0].data();

  if (followingData.following && followingData.following.length > 0) {
    for (const followingUserId of followingData.following) {
      // Fetch activities of followed users
      const followingActivitiesQuery = query(
        currentUserActivitiesRef,
        where("userId", "==", followingUserId)
      );
      const followingActivitiesSnapshot = await getDocs(
        followingActivitiesQuery
      );
      const followingUserActivities =
        followingActivitiesSnapshot.docs.map((doc) => ({
          id: doc.id,
          ...doc.data(),
        }));

      // Merge followed user activities with current user activities
      currentUserActivities = currentUserActivities.concat(
        followingUserActivities
      );
    }
  }
}

```

Figure 68: Display followed user and current user activities in the dashboard Code. Screenshot by Fortune Ndlovu (2024)

Figure 68 shows the code used to display the users that the current user follows activities in the current user's dashboard. Essentially what the code does is that it queries the userActivities collections for activities that have a property of userId with the same value as the followingUserId which is the user's id that the current user follows. We then update their activities and merge the activities with the activities of the current user.

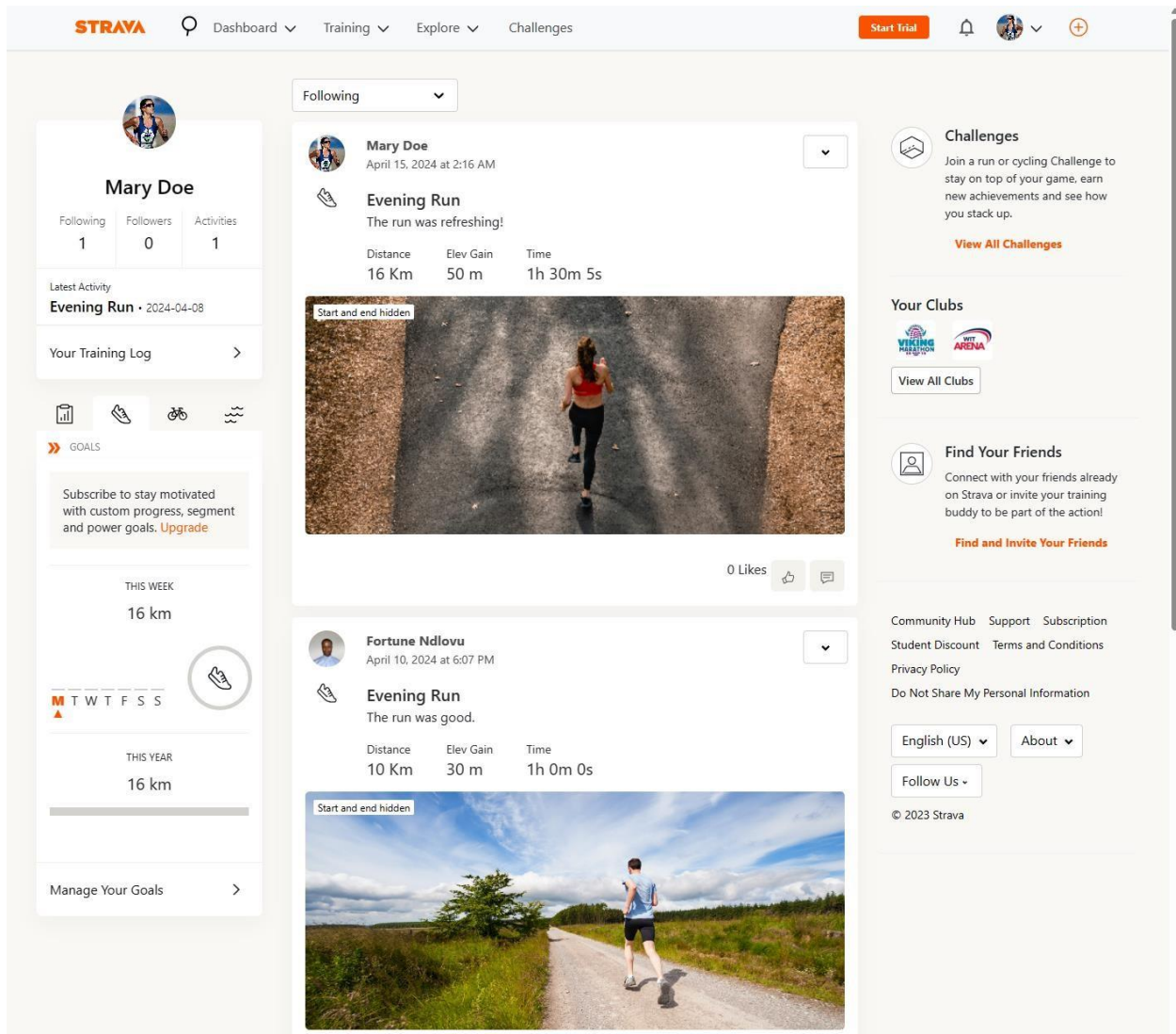


Figure 69: Dashboard containing current user and followed user activities. Screenshot by Fortune Ndlovu (2024)

Figure 69 shows the dashboard of the currently signed-in user in this instance it is Mary Doe, and the dashboard has both the activities of her own and the activities of the users that she follows in this instance Fortune Ndlovu. This is possible because I merged both activities. Once both activities have been merged it was just a matter of displaying them in the UI. I also included the users name and profile image beside the activity they created because each activity has the userId representing the user that created the activity so I was able to fetch the relevant users data and correspond it with the activity the created and display the results. The next section of my report will talk about how I was able to implement the like and commenting interactions on the activities.

4.3.5. Liking and Commenting on Activities

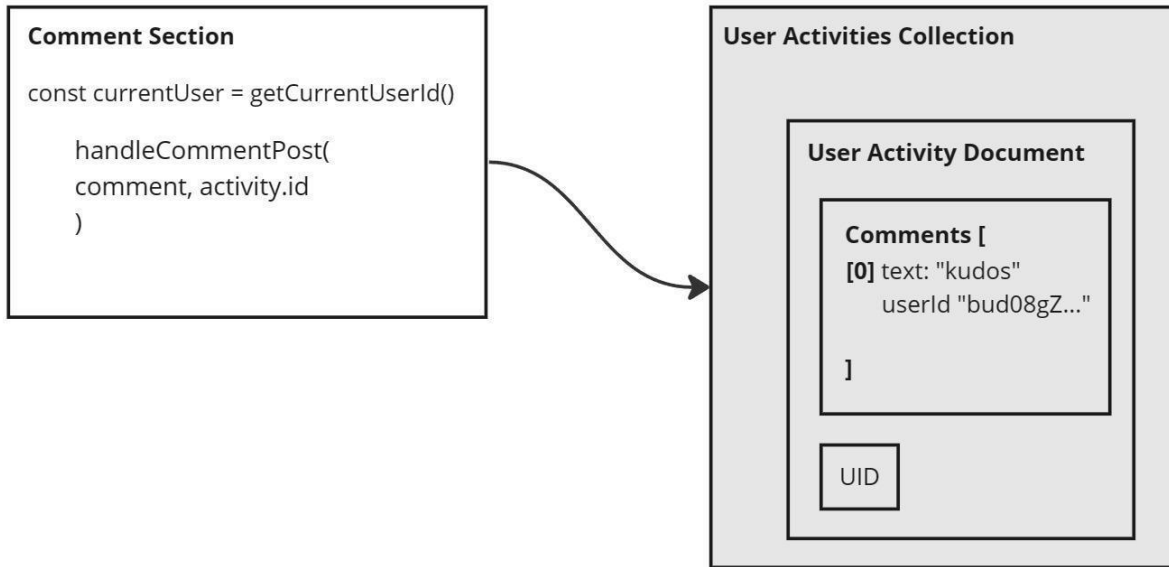


Figure 70: Commenting on activities. Screenshot by Fortune Ndlovu (2024)

Figure 70 shows the design of how users can comment on each other's activities. Essentially the comment section is a small form and when the current user types in their comment and sends it to Firestore, the data is added into the activity's document inside the comments array. The comments array holds the data using objects represented as indexes and for each object it will have the text representing the comment made by the user and the userId representing UID of the user that made the comment.

Before we even post the comment, we get the current user's id because they will be user that made the comment, and when we make the comment, we make sure to add the data into the userActivities collection and in the activity document itself to keep the data centralized. We have access to the activity in question because when displaying the data each activity is attached their unique identifier (id) therefore this means when interacting with the activity we can track exactly which activity is being interacted with and in this instance, we are making a comment. The reason why we also included the userId of the user that created the comment into the comments array in the user activity document of the userActivities collection is because we want to later cross reference and query the userId with the UID in the user's collection to fetch their data.

```

const handleCommentPost = async (comment, activityId) => {
  try {
    const currentUser = getCurrentUserId();
    const userDoc = doc(db, "userActivities", activityId);

    // Get the existing comments for the activity
    const existingComments = (await getDoc(userDoc)).data().comments || [];

    // Construct the new comment object with userId and text
    const newComment = {
      userId: currentUser, // Include userId of the current user
      text: comment, // Comment text
    };

    // Update the comments state with the new comment for the specific activity
    setComments((prevComments) => ({
      ...prevComments,
      [activityId]: [...(prevComments[activityId] || []), newComment],
    }));

    // Update the document with the new comments
    await updateDoc(userDoc, { comments: [...existingComments, newComment] });
  }
}

```

Figure 71: handleCommentPost function. Screenshot by Fortune Ndlovu (2024)

Figure 71 shows the function called when the user clicks the send button to send their comment. This function takes in the comment and the activityId, so we are putting the comment into the correct activity. Within the document of the activity if part of its data does not include a comment array, we create the array and then we create an object to hold the user that created the comment alongside their userId. We then upload the state with the new comment for visualization purposes and then update the document in Firestore with the comments data.

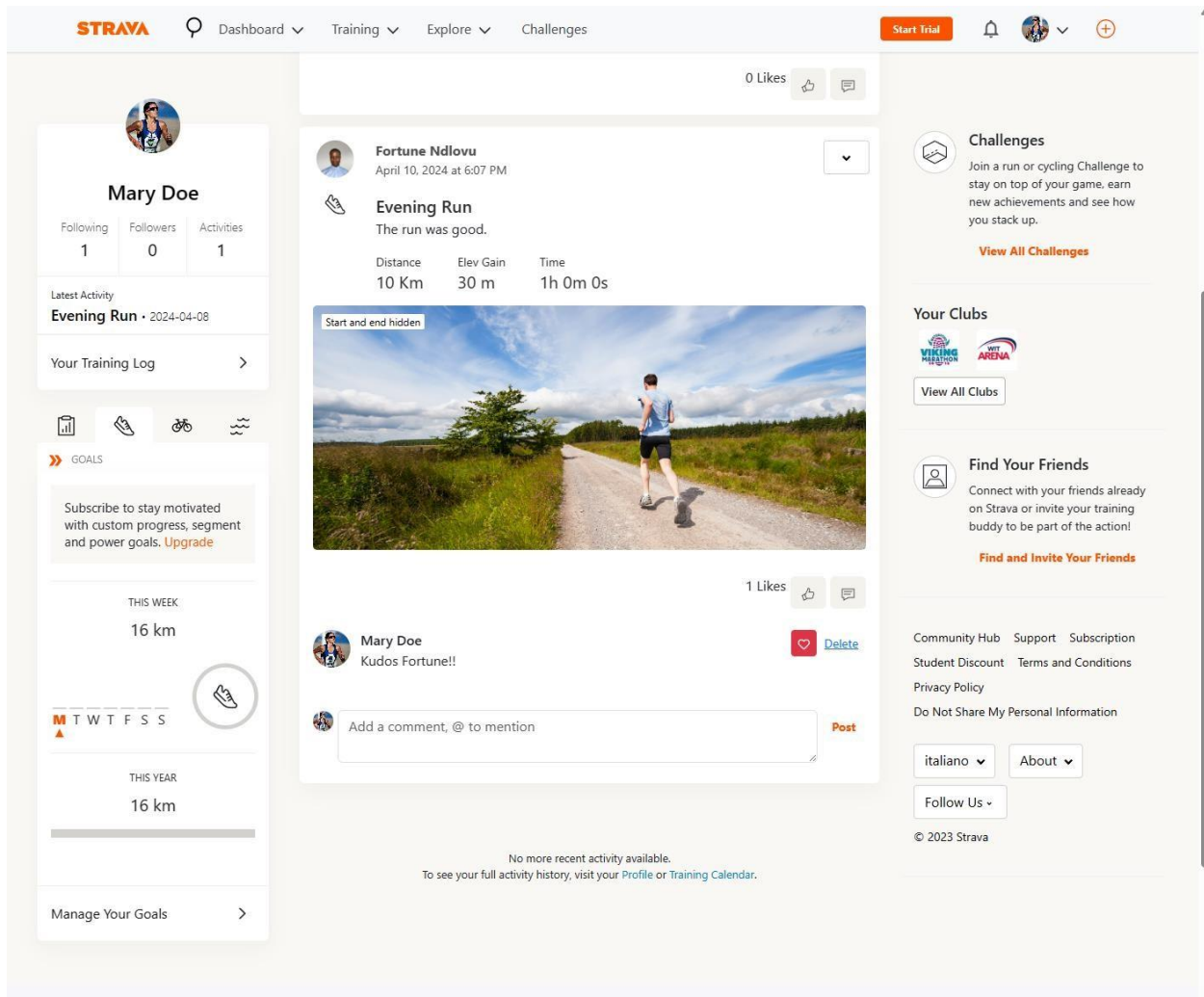


Figure 72: Commenting on Followers Activities. Screenshot by Fortune Ndlovu (2024)

Figure 72 shows that the current user in this instance Mary Doe has commented on a user that she follows Fortune Ndlovu's activity. This works because I constantly listen for updates in my Firestore data and if there is an update or change in this instance data for commenting has been added I then fetch the relevant data and display it in the UI. It is also worth noting that when displaying the comment(s), we also display the like and delete button within the mapping instance this means that the when triggering these events the operations are triggered on the correct comment based on its index in the array.

The screenshot displays the Strava user interface. At the top, the navigation bar includes the Strava logo, a search icon, and menu items for Dashboard, Training, Explore, and Challenges. On the right, there are links for Start Trial, a notification bell, a user profile icon, and a plus sign for additional options.

The main content area is divided into three columns:

- Left Column (Profile):** Shows the profile of Fortune Ndlovu, including a profile picture, name, and statistics: 0 Following, 1 Followers, and 1 Activities. It also lists the latest activity as 'Evening Run' from 2024-04-03 and provides a link to the 'Your Training Log'.
- Middle Column (Activity Post):** Features a post by Fortune Ndlovu titled 'Evening Run' from April 10, 2024, at 6:07 PM. The post includes a photo of a runner on a path and activity details: 10 Km distance, 30 m elevation gain, and a 1h 0m 0s time. Below the photo, there is a comment from Mary Doe saying 'Kudos Fortune!!' and a reply from Fortune Ndlovu saying 'Thank you, Mary!!'. A comment input field at the bottom prompts the user to 'Add a comment, @ to mention'.
- Right Column (Sidebar):** Contains sections for 'Challenges' (with a 'View All Challenges' link), 'Your Clubs' (listing 'VIKING' and 'ARENA' with a 'View All Clubs' link), and 'Find Your Friends' (with a 'Find and Invite Your Friends' link). At the bottom of the sidebar, there are links for 'Community Hub', 'Support', 'Subscription', 'Student Discount', 'Terms and Conditions', 'Privacy Policy', and 'Do Not Share My Personal Information'. Language and region settings are set to 'English (US)', and there is a 'Follow Us' button.

At the bottom of the page, a message states: 'No more recent activity available. To see your full activity history, visit your Profile or Training Calendar.'

Figure 73: Replying to commented Activities. Screenshot by Fortune Ndlovu (2024)

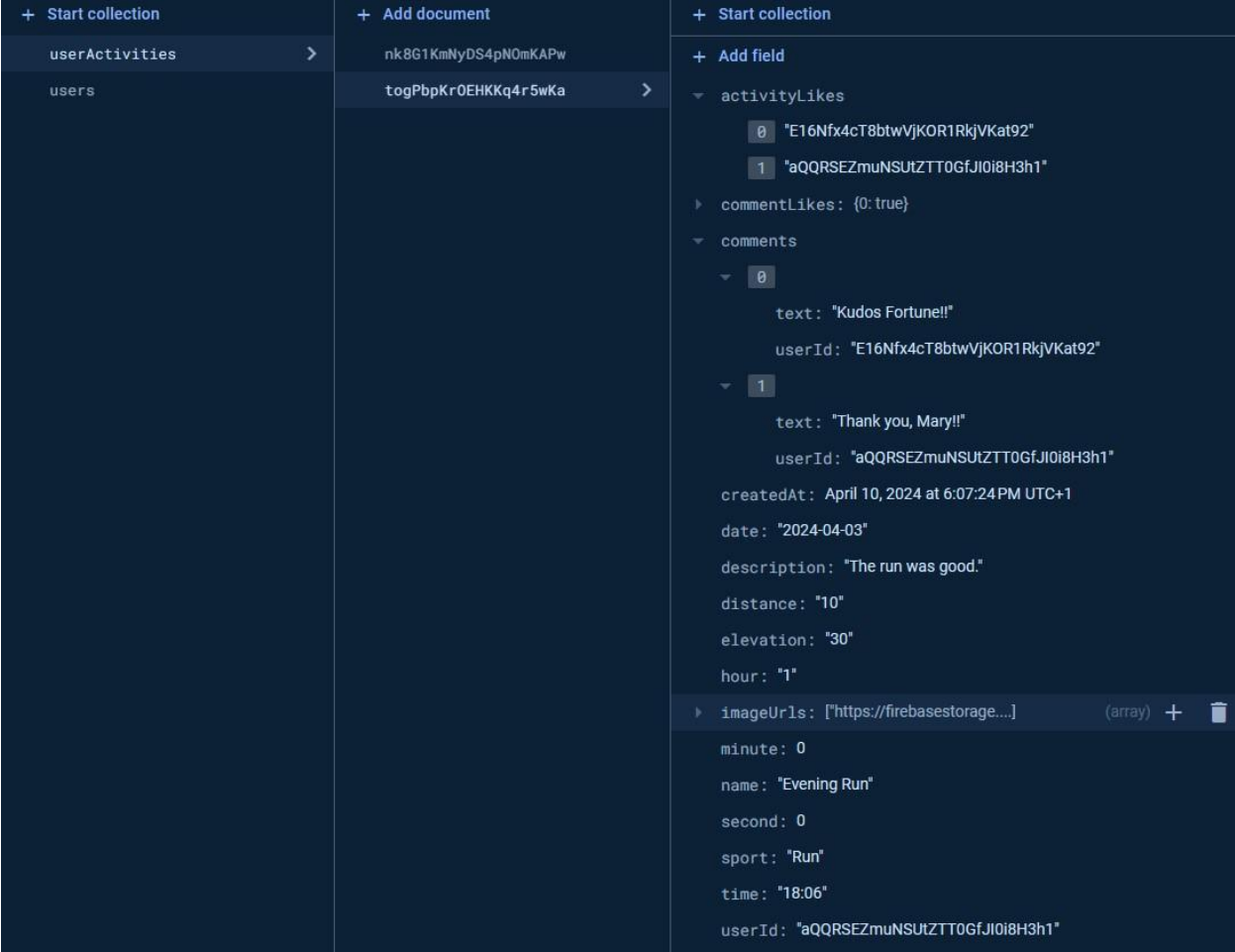


Figure 74: Firestore collection, documents, comment's data update. Screenshot by Fortune Ndlovu (2024)

Figure 73 shows the dashboard UI but in this instance the currently signed user is the user that Mary Doe follows. I signed in to another account to share what the other user sees, and they see only one activity which is the one they created because they do not Follow Mary Doe, but Mary Doe follows them and on the left sidebar, it is visible that they have 1 follower. Figure 73 also shows that the commenting interactions work on both ends of the viewport and Figure 74 shows that the data was in fact successfully added into Firestore userActivities collection within the correct document. The next section of my report will share the challenges and solutions.

4.3.6. Complexities and Solutions

I encountered a lot of complexities when implementing multiple-user integration. Managing multiple users required a robust authentication system to ensure security and data isolation. I implemented user sign up using Firebase Authentication and stored user data in Firestore. This separation of authentication and user data storage was an effective approach because it centralized the data and offered flexibility when query including manipulating the data respectively.

Each user was to have their own personalized dashboard displaying only their activities and relevant data. By associating user IDs with activities in Firestore userActivities collection and querying and filtering activities based on the current users ID, I achieved personalized dashboard functionality. The use of onSnapshot for real-time updates enhanced the UX/UI.

Implementing a system where users can follow/unfollow other users was complex due to the social interaction complexity adding additional queries to Firestore at different instances. I designed a system where users can follow/unfollow others storing these relationships in Firestore users' collection and following array with the ID of the followed user and was able to use that ID for effective data fetching. This system allowed for personalized content based on followed user's activities, enhancing user engagement through complexity.

Enabling users to like and comment on activities required managing these interactions in real time and associating them with specific activities and users. I implemented an implementation of liking and commenting on activities by updating Firestore documents in the correct userActivities collection with comments array and handling real-time updates to reflect these interactions in the UI. Overall, I attached the user's ID to the relevant components which allowed me to easily organize the components based on user interactions. The next section of my report will be about the lessons learned.

4.3.7. Lessons learned

One of my key learnings was understanding the relations between different data entities, such as users, activities, and interactions like comments, and followings. This understanding helped me design an effective data model that could support these relationships efficiently. By structuring Firestore collections and documents based on these relationships such as having separate collections for users and activities and attaching relevant ID for more defined control, it became easier to query and manage data based on specific user interactions. I learned to put my thinking into complex diagrams before coding the implementation of the data related logic because once I could visualize my intentions the implementation became easier.

However, while this approach worked it was not the most efficient approach because if my web app was to scale it would crash easily due to an excessive number of computations between the client, request and server, responses. I was thinking about the most effective solution at the current time due to the lack of time of implementing a more scalable solution, while attaching the user identifiers into the relevant collections and documents is effective there is a more effective approach. The more effective approach is to truly utilize the Firestore ability to be flexible and handle the data for you, this means creating sub collections and documents respectively offering a more scalable and easier to maintain approach. This requires additional planning which I should have taken the time to draw out on paper and execute implementation, I learned to not rush into implementation because it is the easiest route and not to only implement in the structure you already know, take the time to think about more effective solutions and define trade-off analytics.

While my web app works there are a lot of refactors to be made specifically in the multi-user interactions, we are getting a lot of documents from Firestore causing a minor re-render due to the excessive number of utilizing the `getDoc` method, while it is minor it is not professional standard enough we should have created a global instance that gets the documents only once or reduce the method usage another way and also utilizing other unexplored React hooks. The next section of my report will talk about what I learned as a person.

Chapter 5. Personal & Professional Reflection - What I Learned as a Person

Reflecting on my journey as a Full Stack Developer, I've come to realize that the process of building this functional web application has been transformative, both personally and professionally.

From a personal perspective, this journey has been a testament to the power of the human mind. I've learned to harness my cognitive abilities in ways I never thought possible. I've discovered that my brain is not just a repository of information, but a dynamic, logical machine capable of solving complex problems and creating innovative solutions. I've learned to manage my cognitive load, understanding when to delve deep into a problem and when to step back and look at the bigger picture.

I've also come to appreciate the importance of Long-Term Memory (LTM) and Short-Term Memory (STM) in the learning process. Building a web application involves learning and remembering a multitude of concepts, languages, and frameworks. By continually challenging myself, I've been able to strengthen my working memory, enhancing my ability to retain and recall information.

I personally believe that our mind is our greatest asset, and we should take care of our mind. While I worked extremely hard in solving complexities that I had never solved before, I made sure to take care of myself, by eating healthy (fruit and vegetables), sleeping (7 hours minimum) and exercising everyday (running 2.5km minimum). Taking care of my mind interestingly allowed me to think more broadly and focus more effectively within the given complexity because, most of the complexities I had never encountered before therefore knowledge of my previous experiences were crucial (LTM). Moreover, because my mind was fresh and active, it was able to store more information into my STM and limit the number of computations in my working memory for problem solving effectively and with ease.

From a professional perspective, this experience has underscored the importance of continuous learning and improvement in the field of software development. The technology landscape is ever evolving, and to stay relevant, one must be willing to learn and adapt. I've learned that it's okay to not know everything and that every problem I haven't solved before is an opportunity for growth.

Most importantly, I've learned to be one with learning. I've embraced the fact that learning is not a destination but a journey. Every day brings new challenges and new opportunities to learn and grow. This mindset has not only made me a better developer but also a better person.

Furthermore, from a professional perspective, I not only learned to solve complex problems but also, I learned to be more agile, this means I created, adapted and respond to changes quickly through agility. More often I would be handling user data, and I would need to demonstrate critical and creative thinking and for the current solution and later adapt to changing needs which could mean scalability and increased speed through performance.

In everything that I do, especially from a professional perspective, I need to produce efficient and high-standard work that is both error proof and maintainable. I learned the difference between 99% and 100% that 1% is the difference between a Software Developer that just codes to code and the Software Developer that continuously improves so they can solve complex problems that help real users, they give it 100% every time in everything they do to ensure the user experience is simple to use, intuitive and robust.

As a professional, I value producing high quality work because it gives me proof that I am learning from the positive material that I am consuming daily. I strive to continuously improve that is why while this project is my best work yet, now I see a lot of improvements that give me the motivation and discipline to produce even greater work and continue learning from my experiences.

In conclusion, this journey has been enlightening. It has taught me about the power of the human mind, the importance of continuous learning, and the beauty of embracing complexities. It has been a journey of personal growth, professional development, and philosophical enlightenment.

I wish to extend my heartfelt appreciation to my family, lecturers, and friends. Their unwavering support and constructive feedback have been invaluable in navigating the complexities of my journey. They have inspired me to work diligently, fostered my curiosity, and encouraged my explorations and problem-solving endeavours. Their positive influence has been a significant catalyst in my personal and professional growth.

Chapter 7. Conclusion

In this report, I have embarked on a journey to replicate the Strava web application using React and Firestore. The process was challenging, yet rewarding, and provided valuable insights into the complexities of modern web application development.

The exploration began with an in-depth literature review, where I examined the origins and functionalities of Strava, the importance of coding, and the unique features of React. I also delved into the intricacies of replicating Strava in React, addressing the complexities involved in data integration, real-time updates, activity analysis, visualization, user authentication, security, social features, interactions, and handling external APIs and services.

The project's complexities processes allowed me to design and implement basic UX/UI components, manage activities, and integrate multiple users, and multiple user interactions. I encountered various complexities along the way, but I was able to devise effective solutions, demonstrating the flexibility and robustness of my chosen technology stack.

This project contributes to the software development community by showcasing how popular web applications can be replicated using modern technologies like React and Firestore. It serves as a testament to the power of these technologies and their potential to create efficient, scalable, and user-friendly web applications.

This project also contributes to the software development community not only the complexities of integrating accessible UX/UI, activity management, multiple user and multiple user interactions, but also the solutions to these complexities in an easily to digest language. In addition, the lessons learned through each complexity, highlighting that facing complexities head on is a journey not a task and embrace the opportunities.

The process of replicating Strava in React and Firestore was a learning curve that allowed me to deepen my understanding of Full Stack Web Application Development. It was a testament to the power of perseverance, creativity, and technical acumen.

Bibliography and References

ADAM HAYES (2022) 'HyperText Markup Language (HTML): What It Is, How It Works', 24 November. Available at: <https://www.investopedia.com/terms/h/html.asp> (Accessed: 12 May 2024).

Akbulut, S. (2021) 'Data Visualization with React JS and Chart JS'. Available at: <https://medium.com/codex/data-visualization-with-react-js-and-chart-js-cb6ca5d77ff6> (Accessed: 17 October 2023).

APRIL BOHNERT (2023) 'What is React? A Brief Guide to the Front-End Library', 13 November. Available at: <https://www.hackerrank.com/blog/what-is-react-introduction/> (Accessed: 20 February 2024).

Athena Ozanich (2022) 'What Is JavaScript & Why Is It Important?', 3 July. Available at: <https://blog.hubspot.com/website/what-is-javascript> (Accessed: 12 February 2024).

Bhatnagar, S. (2023) 'What is React? Examples, Features, Components, Pros and Cons', 5 September. Available at: <https://www.knowledgehut.com/blog/web-development/what-is-react> (Accessed: 17 October 2023).

Ceci, L. (2023) *Most popular fitness and sport apps worldwide 2023, by revenue*. Available at: <https://www.statista.com/statistics/1239716/top-fitness-and-sport-apps-by-revenue/> (Accessed: 19 November 2023).

Christopher Nguyen (2023) '7 Essential UI/UX Design Fundamentals: A Comprehensive Guide for Designers', 26 June. Available at: <https://uxplaybook.org/articles/7-ux-fundamentals-a-comprehensive-guide> (Accessed: 1 April 2024).

Clive Longbottom and Stephen J. Bigelow (2020) 'infrastructure (IT infrastructure)', November. Available at: <https://www.techtarget.com/searchdatacenter/definition/infrastructure> (Accessed: 15 February 2024).

Create React App (2023) 'Create React App'. Available at: <https://create-react-app.dev/> (Accessed: 18 October 2023).

Domantas G. (2024) 'What Is CSS and How Does It Work?' Available at: <https://www.hostinger.com/tutorials/what-is-css> (Accessed: 12 February 2024).

Doug Stevenson (2018) 'What is Firebase? The complete story, abridged.', 24 September. Available at: <https://medium.com/firebase-developers/what-is-firebase-the-complete-storyabridged-bcc730c5f2c0> (Accessed: 12 February 2024).

Faruq, Y. (2022) 'Handling user authentication with Firebase in your React apps', 10 January. Available at: <https://blog.logrocket.com/user-authentication-firebase-react-apps/> (Accessed: 17 October 2023).

Filip Grkinic (2022) 'A designer's guide to React', 8 November. Available at: <https://uxdesign.cc/a-designers-guide-to-react-d5a8867f214> (Accessed: 1 April 2024).
Firebase (2024) 'Firebase Documentation'. Available at: <https://firebase.google.com/docs/auth?authuser=1&hl=en> (Accessed: 1 April 2024).

Fred Churchville (2021) 'user interface (UI)', September. Available at: <https://www.techtarget.com/searchapparchitecture/definition/user-interface-UI#:~:text=The%20user%20interface%20%28UI%29%20is%20the%20point%20of,user%20interacts%20with%20an%20application%20or%20a%20website.> (Accessed: 26 February 2024).

GeeksForGeeks (2023) 'What are the advantages of React.js?' Available at: <https://www.geeksforgeeks.org/what-are-the-advantages-of-react-js/> (Accessed: 17 October 2023).

Gobo, K. (2020) 'Understanding Firebase Realtime Database using React'. Available at: <https://codesource.io/understanding-firebase-realtime-database-using-react/> (Accessed: 17 October 2023).

HALL, C. (2023) 'What is Strava, how does it work and is it worth paying for?' Available at: <https://www.pocket-lint.com/apps/news/154854-what-is-strava-and-how-does-it-work/> (Accessed: 5 October 2023).

Hansel (2018) 'The History of Coding and Computer Programming'. Available at: <https://www.thecoderschool.com/blog/the-history-of-coding-and-computer-programming/> (Accessed: 6 October 2023).

HANSEN-GILLIS, L. (2020) 'Strava's latest update will probably change the way you use the app', 18 May. Available at: <https://cyclingmagazine.ca/sections/news/strava-subscriber-update/> (Accessed: 18 November 2020).

Ibadehin Mojeed (2022) 'What is the virtual DOM in React?', 16 August. Available at: <https://blog.logrocket.com/virtual-dom-react/> (Accessed: 14 February 2024).

Ikechukwu, V. (2022) 'Learn React Hooks – A Beginner's Guide'. Available at: <https://www.freecodecamp.org/news/the-beginners-guide-to-react-hooks/> (Accessed: 18 October 2023).

interaction-design (2024) 'User Experience (UX) Design'. Available at: <https://www.interactiondesign.org/literature/topics/ux-design#:~:text=User%20experience%20%28UX%29%20design%20is%20the%20process%20design,inclusing%20aspects%20of%20branding%2C%20design%2C%20usability%20and%20function.> (Accessed: 12 February 2024).

InterviewBit (2022) 'Difference Between Coding and Programming'. Available at: <https://www.interviewbit.com/blog/difference-between-coding-and-programming/> (Accessed: 17 October 2023).

Ivy Wigmore (2019) 'unique identifier (UID)', September. Available at: <https://www.techtarget.com/iotagenda/definition/unique-identifier-UID#:~:text=A%20unique%20identifier%20%28UID%29%20is%20a%20numeric%20or,that%20it%20can%20be%20accessed%20and%20interacted%20with.> (Accessed: 20 February 2024).

Janani (2021) 'A Beginner's Guide to DOM (Document Object Model)', 21 December. Available at: <https://www.atatus.com/blog/a-beginners-guide-to-dom/> (Accessed: 12 February 2024).

Jeffery Parker (2023) 'What Is Programming Architecture?', 7 April. Available at: https://www.architecturemaker.com/what-is-programming-architecture/#google_vignette (Accessed: 1 April 2023).

Jessica Scarpath and John Burke (2021) 'URL (Uniform Resource Locator)', September. Available at: <https://www.techtarget.com/searchnetworking/definition/URL> (Accessed: 27 February 2024).

Jest (2023) 'Jest Delightful JavaScript Testing'. Available at: <https://jestjs.io/> (Accessed: 17 October 2023).

Ko, N. (2021) 'What is Coding and What is it Used For? A Beginner's Guide'. Available at: <https://www.zdnet.com/education/learn-it-design-and-coding-skills-for-just-20-with-this-coursepack/> (Accessed: 17 September 2023).

legacy reactjs (2024) 'Design Principles'. Available at: <https://legacy.reactjs.org/docs/designprinciples.html> (Accessed: 1 April 2024).

Lemonaki, D. (2021) 'What is Coding? Computer Coding Definition'. Available at: <https://www.freecodecamp.org/news/what-is-coding/> (Accessed: 6 October 2023).

Ijaz, U. (2023) 'How to use the Fetch API with React?' Available at: <https://rapidapi.com/guides/fetch-api-react> (Accessed: 17 October 2023).

Margaret Rouse (2016) 'Scalable Vector Graphics', 15 November. Available at: <https://www.techopedia.com/definition/5239/scalable-vector-graphicssvg#:~:text=Scalable%20vector%20graphics%20%28SVG%29%20is%20a%20textbased%20graphics,devices.%20In%20addition%2C%20SVG%20supports%20animation%20and%20scripting.> (Accessed: 25 February 2024).

Margaret Rouse (2024) 'Application Programming Interface', 23 January. Available at: <https://www.techopedia.com/definition/24407/application-programming-interface-api> (Accessed: 23 February 2024).

Mark (2023) *How to Use Strava: A Guide to Strava's Popular Features*. Available at: <https://bikexchange.com/strava-popular-features-guide/> (Accessed: 8 November 2023).

Mason (2022) 'Strava Features Complete Overview'. Available at: <https://communityhub.strava.com/t5/athlete-knowledge-base/strava-features-completeoverview/ta-p/275> (Accessed: 17 October 2023).

Merchant, A. (2023) 'Ac(count)ing for Scale'. Available at: <https://medium.com/stravaengineering/ac-count-ing-for-scale-becf9c27b104> (Accessed: 5 October 2023).

Nick Barney (2023) 'authentication', November. Available at: <https://www.techtarget.com/searchsecurity/definition/authentication> (Accessed: 12 February 2024).

Niedringhaus, P. (2020) 'Polling In React Using The UseInterval Custom Hook', 9 October. Available at: <https://blog.openreplay.com/polling-in-react-using-the-useinterval-custom-hook/> (Accessed: 17 October 2023).

npm (2023) 'browser-image-compression'. Available at: <https://www.npmjs.com/package/browser-image-compression> (Accessed: 1 April 2024).

Ozanich, A. (2022) 'What Is JavaScript & Why Is It Important?', 3 July. Available at: <https://blog.hubspot.com/website/what-is-javascript> (Accessed: 18 October 2023).

PARTIDA, D. (2022) 'The Evolution of Coding: From Its Origins to the Future'. Available at: <https://rehack.com/tech-efficiency/the-evolution-of-coding-from-its-origins-to-the-future/> (Accessed: 6 October 2023).

Paul Kirvan (2024) 'Amazon Web Services (AWS)', January. Available at: <https://www.techtarget.com/searchaws/definition/Amazon-Web-Services> (Accessed: 20 February 2024).

React Dev (2023) 'Component'. Available at: <https://react.dev/reference/react/Component> (Accessed: 18 October 2023).

React Router (2023) 'React Router: Declarative Routing for React.js'. Available at: <https://v5.reactrouter.com/web/guides/quick-start> (Accessed: 17 October 2023).

redhat (2019) 'What is an IDE?', 8 January. Available at: <https://www.redhat.com/en/topics/middleware/what-is-ide> (Accessed: 18 January 2024).

Redux (2023) 'Redux A Predictable State Container for JS Apps'. Available at: <https://redux.js.org/> (Accessed: 17 October 2023).

Richroll (2021) 'The Strava Story: Building A Fitness Community Fueled By Emotional Connection'. Available at: <https://www.trustedreviews.com/explainer/what-is-strava-4259841> (Accessed: 5 October 2023).

Robert Sheldon (2023) 'PHP (Hypertext Preprocessor)', April. Available at: <https://www.techtarget.com/whatis/definition/PHP-Hypertext-Preprocessor#:~:text=PHP%20%28Hypertext%20Preprocessor%29%20is%20a%20generalpurpose%20scripting%20language,scripting%20and%2C%20to%20a%20limited%20degree%2C%20desktop%20applications.> (Accessed: 14 February 2024).

Sawh, M. (2020) 'Heart rate monitors for Strava: Compatible devices to track your workouts'. Available at: <https://www.wareable.com/running/heart-rate-monitors-strava> (Accessed: 18 October 2023).

'software development kit (SDK)' (2022), October. Available at: <https://www.techtarget.com/whatis/definition/software-developers-kit-SDK> (Accessed: 21 February 2024).

Strava (2023) 'Strava Privacy Policy'. Available at: <https://www.strava.com/legal/privacy> (Accessed: 5 October 2023).

STRAVA stories (2023) 'Why Athletes Choose Strava'. Available at: <https://blog.strava.com/why-athletes-choose-strava/>.

WebandCrafts (2021) 'Top Features and Benefits of Using React JS for Web Development', 30 August. Available at: <https://webandcrafts.com/blog/react-js-features> (Accessed: 17 October 2023).

Wesley Chai and Kevin Ferguson (2021) 'HTTP (Hypertext Transfer Protocol)', March. Available at: <https://www.techtarget.com/whatis/definition/HTTP-Hypertext-Transfer-Protocol> (Accessed: 14 February 2024).

Appendix 1 - List of Figures

Figure 1: Leading fitness and sport apps worldwide in March 2023, by revenue (Statista 2023)

Figure 2: Strava founders Mark Gainey and Michael Horvath (HANSEN-GILLIS, 2020)

Figure 3: Coding. Photograph by Chris Ried. (Unsplash, 2018)

Figure 4: John von Neumann with the stored-program computer at the Institute for Advanced Study, Princeton, New Jersey, in 1945. Photograph by Getty (The Guardian, 2012)

Figure 5: Example of a React code that uses JSX, hooks, and components to create a simple counter app. Coded by Fortune Ndlovu.

Figure 6: Header Search box open. Screenshot by Fortune Ndlovu (2024)

Figure 7: Header Resize Hamburger display. Screenshot by Fortune Ndlovu (2024)

Figure 8: Header Resize Hamburger dropdown display. Screenshot by Fortune Ndlovu (2024)

Figure 9: Conditional Rendering Logical OR inside Logical AND (&&) Expression. Screenshot by Fortune Ndlovu (2024)

Figure 10: Initially Setting Dropdown Menus to true based on isHamburger. Screenshot by Fortune Ndlovu (2024)

Figure 11: useEffect hook for listening to window resizes. Screenshot by Fortune Ndlovu (2024)

Figure 12: Toggling State to its previous value. Screenshot by Fortune Ndlovu (2024)

Figure 13: Footer UX/UI. Screenshot by Fortune Ndlovu (2024)

Figure 14: Home Page UX/UI. Screenshot by Fortune Ndlovu (2024)

Figure 15: Home Page Left Sidebar UX/UI. Screenshot by Fortune Ndlovu (2024)

Figure 16: Basic Example of the Left Sidebar UX/UI Code. Screenshot by Fortune Ndlovu (2024)

Figure 17: Left Sidebar AthleteStatsTabs Code Breakdown. Screenshot by Fortune Ndlovu (2024)

Figure 18: Left Sidebar WeeksStyleSvg Code Breakdown. Screenshot by Fortune Ndlovu (2024)

Figure 19: Left Sidebar WeeksStyleSvg daysOfWeek Map Code Breakdown. Screenshot by Fortune Ndlovu (2024)

Figure 20: Firebase Project Features. Screenshot by Fortune Ndlovu (2024)

Figure 21: Firestore Dashboard. Screenshot by Fortune Ndlovu (2024)

Figure 22: Firebase Project Settings. Screenshot by Fortune Ndlovu (2024)

Figure 23: Firebase Configuration. Screenshot by Fortune Ndlovu (2024)

Figure 24: createActivity Function. Screenshot by Fortune Ndlovu (2024)

Figure 25: ManualEntryForm UX/UI. Screenshot by Fortune Ndlovu (2024)

Figure 26: ManualEntryForm useState hooks Code. Screenshot by Fortune Ndlovu (2024)

Figure 27: ManualEntryForm ActivityStats and ActivityDetails Components. Screenshot by Fortune Ndlovu (2024)

Figure 28: ManualEntryForm createdActivity Object. Screenshot by Fortune Ndlovu (2024)

Figure 29: ActivityDetails Component. Screenshot by Fortune Ndlovu (2024)

Figure 30: ActivityDetails Component. Screenshot by Fortune Ndlovu (2024)

Figure 31: EditActivityForm UX/UI Component. Screenshot by Fortune Ndlovu (2024)

Figure 32: EditActivityForm Component edit data Code. Screenshot by Fortune Ndlovu (2024)

Figure 33: EditActivityForm Component handleSaveChanges Code. Screenshot by Fortune Ndlovu (2024)

Figure 34: Dashboard UX/UI With Generated Activity. Screenshot by Fortune Ndlovu (2024)

Figure 35: MyActivitiesTable UX/UI. Screenshot by Fortune Ndlovu (2024)

Figure 36: MyActivitiesTable Edit Button clicked UX/UI. Screenshot by Fortune Ndlovu (2024)

Figure 37: UserActivitiesManager Functions. Screenshot by Fortune Ndlovu (2024)

Figure 38: MyActivitiesTable component return. Screenshot by Fortune Ndlovu (2024)

Figure 39: MyActivitiesTable JSX return. Screenshot by Fortune Ndlovu (2024)

Figure 40: MyActivitiesTable Event Functions. Screenshot by Fortune Ndlovu (2024)

Figure 41: Firestore UserActivities Collection first document. Screenshot by Fortune Ndlovu (2024)

Figure 42: handleImageChange function. Screenshot by Fortune Ndlovu (2024)

Figure 43: handleCreateActivity function. Screenshot by Fortune Ndlovu (2024)

Figure 44: handleFileInputChange function. Screenshot by Fortune Ndlovu (2024)

Figure 45: handleAddImage function. Screenshot by Fortune Ndlovu (2024)

Figure 46: editedActivityState. Screenshot by Fortune Ndlovu (2024)

Figure 47: imageUrls mapping. Screenshot by Fortune Ndlovu (2024)

Figure 48: handleRemoveImage. Screenshot by Fortune Ndlovu (2024)

Figure 49: Firestore Documentation addDoc method. Screenshot by Fortune Ndlovu (2024)

Figure 50: Image Properties File Explore inspection example. Screenshot by Fortune Ndlovu (2024)

Figure 51: Image Properties Firebase Storage inspection example. Screenshot by Fortune Ndlovu (2024)

Figure 52: Browser Image Compression Code. Screenshot by Fortune Ndlovu (2024)

Figure 53: Multiple User Data Architecture. Screenshot by Fortune Ndlovu (2024)

Figure 54: authSignUpWithEmailAndPassword function. Screenshot by Fortune Ndlovu (2024)

Figure 55: Sign Up Form UI. Screenshot by Fortune Ndlovu (2024)

Figure 56: Firebase Authentication Dashboard. Screenshot by Fortune Ndlovu (2024)

Figure 57: Firebase Cloud Firestore Dashboard. Screenshot by Fortune Ndlovu (2024)

Figure 58: Current User Activities on the Dashboard Authentication. Screenshot by Fortune Ndlovu (2024)

Figure 59: Tracking Changes in the userActivities Collection. Screenshot by Fortune Ndlovu (2024)

Figure 60: Firestore UserId in Document. Screenshot by Fortune Ndlovu (2024)

Figure 61: Currently Signed in users dashboard UI. Screenshot by Fortune Ndlovu (2024)

Figure 62: Currently Signed in users Profile UI. Screenshot by Fortune Ndlovu (2024)

Figure 63: handleSearch function. Screenshot by Fortune Ndlovu (2024)

Figure 64: Searched Users Profile Page. Screenshot by Fortune Ndlovu (2024)

Figure 65: Users collection following array. Screenshot by Fortune Ndlovu (2024)

Figure 66: Following user and Unfollowing User Logic. Screenshot by Fortune Ndlovu (2024)

Figure 67: Followers Activities in the Dashboard. Screenshot by Fortune Ndlovu (2024)

Figure 68: Display followed user and current user activities in the dashboard Code. Screenshot by Fortune Ndlovu (2024)

Figure 69: Dashboard containing current user and followed user activities. Screenshot by Fortune Ndlovu (2024)

Figure 70: Commenting on activities. Screenshot by Fortune Ndlovu (2024)

Figure 71: handleCommentPost function. Screenshot by Fortune Ndlovu (2024)

Figure 72: Commenting on Followers Activities. Screenshot by Fortune Ndlovu (2024)

Figure 73: Replying to commented Activities. Screenshot by Fortune Ndlovu (2024)

Figure 74: Firestore collection, documents, comment's data update. Screenshot by Fortune Ndlovu (2024)

Appendix 2 - List of Tables

Table 1. Mark (2023) *How to Use Strava: A Guide to Strava's Popular Features*. Available at: <https://bikexchange.com/strava-popular-features-guide/> (Accessed: 8 November 2023).

Table 2. Siu, K. (2022) 'Why coding is important for the future'. Available at: <https://teachyourkidscode.com/how-important-is-coding-for-the-future/> (Accessed: 15 October 2023).

Table 3. Programiz PRO Resources (2021) 'Coding vs. Programming'. Available at: <https://programiz.pro/resources/coding-vs-programming/> (Accessed: 15 October 2023). React Dev (2023) 'Component'. Available at: <https://react.dev/reference/react/Component> (Accessed: 18 October 2023).

Table 4. Nyakundi, H. (2021) 'What is the Difference Between Coding and Programming?' Available at: <https://www.freecodecamp.org/news/difference-between-coding-and-programming/> (Accessed: 15 October 2023).

Appendix 3 - Definitions

Architecture

Architecture refers to the overall structure and organisation of a software system. It's an essential aspect of software development because it determines the structure and organisation of the codebase. It plays a crucial role in the maintainability, scalability, and overall success of a project (Jeffery Parker, 2023).

Application Programming Interface (API)

An Application Programming Interface (API) is like a bridge that allows different software applications to talk to each other. It's a set of rules and tools that developers use to build software that can interact with other services easily. For instance, when you track a package on an e-commerce site, it's the API that retrieves the tracking information from the postal services system and displays it on the website for you. APIs are versatile and can be used with various programming languages, often using a RESTFUL architecture which relies on standard web protocols like HTTP. They are crucial for creating interconnected systems where different applications share data services, making our digital experiences smoother and more integrated (Margaret Rouse, 2024).

Amazon Web Services (AWS)

Amazon Web Services (AWS) is a vast and comprehensive cloud platform offered by Amazon. It's essentially a collection of cloud services that provide users with resources like computing power, storage, and content delivery capabilities. AWS is notable for its variety of services that cater to diverse needs such as infrastructure management, application development, and data analytics (Paul Kirvan, 2024).

Authentication (Auth)

Authentication is a process used to verify that someone or something is actually who or what they claim to be in the digital world. Think of it as showing your ID card before entering a secure building. Just like an ID card confirms your identity, authentication checks digital credentials to confirm users' identity. Often part of a multi-step process, which might include passwords, biometrics, or secure tokens, adding layers of protection against unauthorized access (Nick Barney, 2023).

Cascading Style Sheet (CSS)

Cascading Style Sheet (CSS) is a language used to add style to web documents. It allows developers to apply styles such as colors, fonts and layouts to HTML elements, making web pages visually appealing. It simplifies maintenance and improves user experiences by separating structure from style (Domantas G., 2024).

Document Object Model (DOM)

The Document Object Model (DOM) is a programming interface that allows web documents to be treated as a tree structure where each node is an object representing a part of the document. It enables dynamic access and manipulation of the document's content structure and styles using various programming languages, primarily JavaScript (Janani, 2021).

Firebase

Firebase is a comprehensive suite of tools provided by Google that enables developers to build, improve, and grow their mobile and web applications efficiently. It offers a variety of services that are typically challenging to create from scratch such as authentication, database, and storage solutions. These services are cloud hosted meaning they are managed by Google, which allows developers to focus more on infrastructure (Doug Stevenson, 2018).

Hyper Text Markup Language (HTML)

Hyper Text Markup Language (HTML) is the standard language used to create and design documents on the World Wide Web. HTML uses markup to annotate text images, and other content for displaying in a web browser. It consists of a series of elements or tags that define the structure and presentation of content (ADAM HAYES, 2022).

HyperText Transfer Protocol (HTTP)

HyperText Transfer Protocol, is essentially the system that enables the fetching and displaying of web pages on the internet. It's a set of rules for transferring various types of data on the web like text, images and videos. When we use a browser it sends requests to a server using HTTP, and the server responds with request of the page (Wesley Chai and Kevin Ferguson, 2021).

Integrated Development Environment (IDE)

An Integrated Development Environment (IDE) is a comprehensive software suite that combines essential tools for coding into a single user-friendly interface, enabling developers to write, test, and debug their code more efficiently. It integrates tools like text editors, compilers, and debuggers onto one platform (redhat, 2019).

Infrastructure

Infrastructure in the context of IT, refers to the collective physical and virtual components that form the backbone for operating and managing IT environments. It's the underlying framework that supports systems and organizations, it includes servers, storage, networking devices, and software that enables data flow, storage, and processing (Clive Longbottom and Stephen J. Bigelow, 2020).

JavaScript (JS)

JavaScript is a high-level interpreted programming language that is primarily used to add interactivity and complex features to websites. JavaScript is mainly used for client-side scripting where scripts are downloaded from the web server along with the HTML code and run directly on the user's browser. This allows for faster response times and a smoother user experience. While JavaScript started as a browser-based language it has grown to be used in many other environments such as server-side programming with Node.js, mobile app development with React Native, and even game development (Athena Ozanich, 2022).

HyperText Processor (PHP)

HyperText Preprocessor (PHP) is a widely used open-source scripting language that's especially suited for web development. It can be embedded into HTML, which makes it particularly convenient for creating dynamic web pages. PHP is designed for a variety of programming tasks even outside of the web context. Primarily used for developing server-side applications and dynamic web content. It can easily be integrated with various databases, and it offers a rich set of functionalities including handling forms, accessing cookies and databases, and managing sessions. PHP scripts are executed on the server and the result is sent to the client as plain HTML making it a powerful tool for creating interactive and dynamic websites (Robert Sheldon, 2023).

Portable Network Graphics (PNG)

Portable Network Graphics (PNG) is an image format that's used for compressing images without losing any quality. It's great for images with sharp edges and solid colors, like logos or text because it keeps the details crisp and clear. Unlike JPEG which can lose some image quality when its compressed PNG keeps, PNG keeps everything looking just as it did before it was compressed (Portable Network Graphics).

React

React is an open-source JavaScript library used for building user interfaces, especially for singlepage applications. It allows developers to create web applications that can update and display data dynamically without needing to reload the page. React is known for its component-based architecture which enables developers to build reusable UI components that manage their own state, leading to more efficient and easier-to-maintain code. Additionally, React uses a virtual DOM to optimize rendering performance making it a popular choice for complex applications that require fast and interactive user experiences (APRIL BOHNERT, 2023).

Software Development Kits (SDK)

A Software Development Kit (SDK) is like a toolbox that provides developers with all the necessary tools, components, and instructions they need to build software applications efficiently. It typically includes libraries that are pre-written code that developers can use to perform common tasks. SDKs also have APIs as sets of rules that allows different software programs to communicate with each other ('software development kit (SDK)', 2022)

Scalable Vector Graphics (SVG)

Scalable Vector Graphics (SVG) are text-based graphics that use vector shapes and text descriptions to render images. They are lightweight, scalable, and can include animations and interactivity. SVG is an open standard developed by the W3C and is widely supported by modern web browsers. SVGs can resize without looking quality (Margaret Rouse, 2016).

User Interface (UI)

User Interface (UI) is the part of a software application or device that people use to interact with it. It's what people see and touch, like buttons, text, images and sliders on a screen. A good UI makes it easy and pleasant for users to complete tasks, while a bad one can make even simple tasks frustrating (Fred Churchville, 2021).

User Identifier (UID)

User Identifier (UID) is like a special label or code that is given to someone or something to tell it apart from others in a system. It's a bit like having a unique name or number that no one else has so that you can be easily found and recognized. UIDs can be used for all sorts of things, like keeping track of users on a website and sorting out parts in a factory. They can also be chosen by a person or created automatically by a computer in short UIDs are super important for organizing and keeping track of everything in a neat and secure way (Ivy Wigmore, 2019).

Uniform Resource Locator (URL)

A Uniform Resource Locator (UR) is like an address for a specific resource on the internet. Just like you would need an address to find a house or a store in the real world you need a URL to find a webpage or other resources on the internet. A URL typically consists of several parts firstly a Protocol, this is the method used to access the resource. It's usually http or https, but there are others like ftp for file transfers. Secondly Domain Name, this is the name of the server where the resource is located, is usually something like www.example.com. Thirdly Path this is the specific location of the resource on the server (Jessica Scarpath and John Burke, 2021).

User Experience (UX)

User Experience (UX) is the practice of creating products that offer meaningful and relevant experiences to users. It encompasses the entire process of product interaction from branding and design to usability and function. UX design aims to address user needs and pain points ensuring the product is not only functional but also enjoyable to use (interaction-design, 2024).

Virtual Document Object Model (VDOM)

The Virtual Document Object Model (VDOM) in React is a lightweight abstract copy of the actual DOM in the browser. It's a concept that allows React to know when to re-render and when to ignore certain pieces of the DOM. When a component's state changes a new VDOM representation of the user interface is created. The new VDOM is then compared with the previous VDOM snapshot. Through a process called diffing, React figures out which parts of the actual DOM need to change. React then updates only those parts in the actual DOM, rather than re-rendering the entire DOM, this process makes React highly efficient (Ibadehin Mojeed, 2022).