OST
Eastern Switzerland
University of Applied Sciences

Bachelorarbeit

# Fitness Gamification

Department of Computer Science

OST - University of Applied Sciences

Campus Rapperswil-Jona

| | |
|---|---|
| **Author(s)** | Joel Suter & Lucas von Niederhäusern |

| | |
|---|---|
| **Advisor** | Frank Koch |
| **Project Partner** | Michael Güntensperger |
| **External Co-Examiner** | Hansjörg Huser |
| **Internal Co-Examiner** | Mitra Purandare |

# Abstract

**Initial Situation** In today's world, gamification is used and implemented in more and more aspects of daily life. The purpose of gamification is to motivate users to do certain things. Another growing market is physical exercise, especially with young people. An increasing amount of people are trying to do more physical exercise. Everyone agrees that physical exercise is the key to a healthy life, but the main problem is that people often need more motivation. So we see massive potential in solutions that address this problem. A potential solution to motivate people to do more sports is a platform that gamifies physical exercise and makes it a fun, competitive game that can be played between friends and other users. The need for physical exercise and motivation to do it will draw people to platforms and solutions like this.

**Objective** With this project, we aim to build a platform mobile app of this type. The main goal is to create an initial version of such a platform. That means we want to create an app that allows users to challenge their friends and other users to do physical exercises. We will implement various gamification aspects, such as allowing users to achieve milestones, track their data, and display that to other users. We plan to use a "turn-based" approach for the challenges, allowing users to play with users in an "async" fashion. That means participants can be online at different times.

**Conclusion** In this project, we created a prototype for a mobile app that gamifies exercise. We created this mobile App using React Native and Expo in our Frontend and Firebase as our Backend. Our App allows users to create an account using their phone number and challenge anyone in their contact list or a random user. The user can select the challenge type and duration one has to perform in the challenge. Challenges then get completed in a turn-based fashion. A user performs a challenge, and it gets sent to his opponent. The opponent can then perform, and it will get sent back. This goes back and forth until one user cannot beat the other user's score. We will track various statistics for the user, which will be displayed in his profile and displayed to other users to generate a more competitive and gamified environment. We implemented one challenge type for this initial version, "Pull-ups." In the future, this should be extended to allow the user to perform various challenges and can generally be expanded upon due to the modularity we considered while implementing it.

Figure 1: Firebase

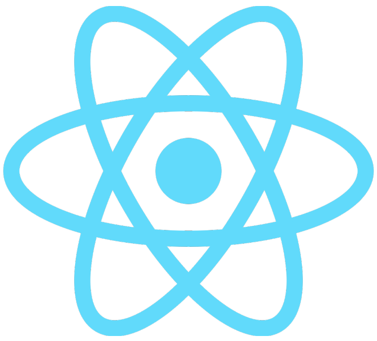


Figure 2: React Native

# Management Summary

Promoting fitness and health while making it fun is a rare combination. Our project aimed to develop a mobile application that achieves this by incorporating gamification elements to motivate users to exercise and enjoy a sense of gratification. The application we created is simple and user-friendly. It allows you to Challenge your friends, family, or even random individuals to participate in various physical activities.

In our context, a Challenge involves daring your chosen Opponent to partake in physical activity. The chosen Opponent will then receive a Notification on his phone that he got challenged. He then has 12 hours to perform the chosen activity as best as he can. Once he is done, the challenge will get sent back to the Creator, and he will be able to try to beat the score. This process will repeat until someone is not able to outperform his Opponent. If a user cannot beat the score, he will lose the Challenge, and his Opponent will be crowned as the winner. To detect repetitions (score) of the exercise accurately, our application requires you to place your mobile device in your pocket or another suitable location.

As you engage in activities within the app, you can earn Badges (Milestones) and increase your statistics that will be showcased on your Banner. The Banner is a part of the profile and will be displayed to all your Opponents before they perform on a Challenge. This way the users can see their Opponent's experience before competing. This incentivizes each user to maximize his statistics and win as many Challenges as possible so that other users see their achievements. It creates a more competitive environment and encourages users to participate in physical exercise. Additionally, you can personalize your Banner with your profile picture, making it unique to you. The app also provides a comprehensive overview of your key statistics and performance metrics and identifies your favorite Opponent.

Furthermore, the app incorporates a global ranking feature that showcases the top 100 players based on their performance. You will be included in this prestigious ranking if your performance is exceptional.

Currently, our application supports a "pull-ups Challenge", but we have designed it with modularity in mind, allowing for efficient integration of additional Challenges in the future. Looking ahead, the plan is to explore the utilization of machine learning or AI to enhance the detection of repetitions in various physical activities, ensuring more efficient and reliable tracking while implementing safeguards against cheating.

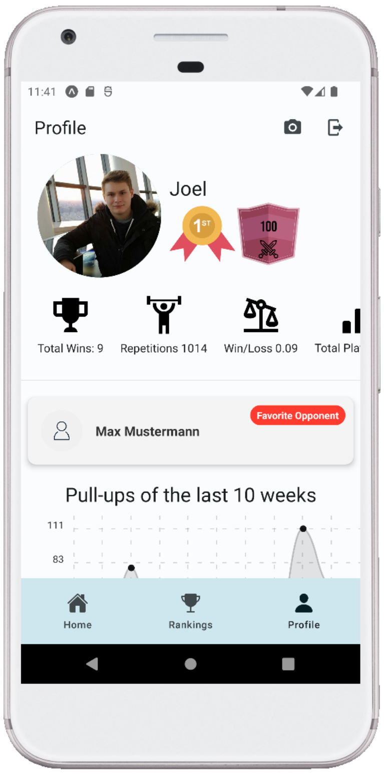


Figure 3: User Profile

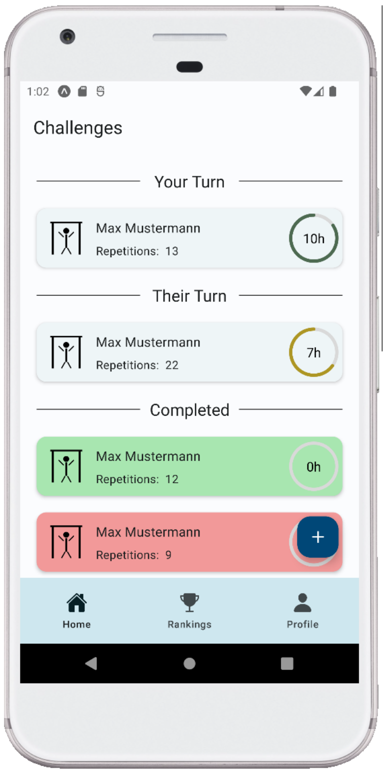
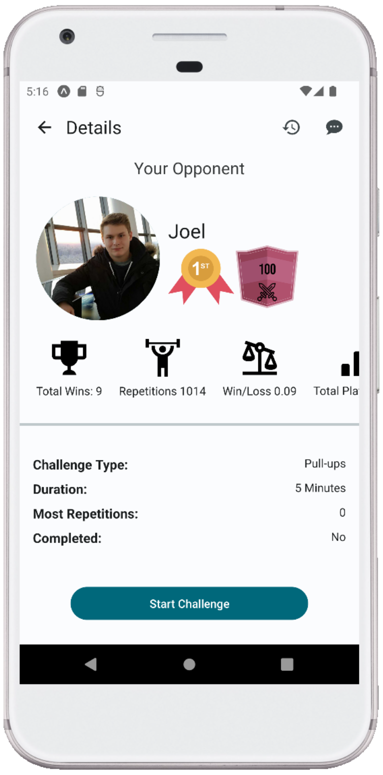


Figure 4: Home Screen



| | |
|---|---|
| **Challenge Type:** | Pull-ups |
| **Duration:** | 5 Minutes |
| **Most Repetitions:** | 0 |
| **Completed:** | No |

Figure 5: Challenge Details Screen with Opponents Banner

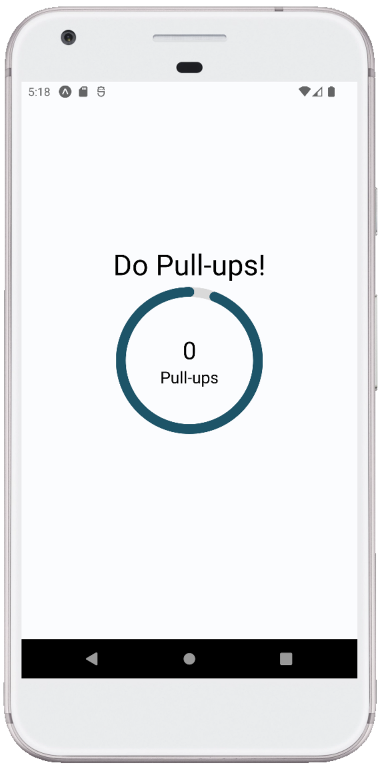


Figure 6: Challenge Perform Screen

## Glossary

**Agile** Agile is a project management and product development approach that prioritizes flexibility, collaboration, and customer satisfaction. The methodology is adaptive rather than predictive, with changes and iterations being a normal part of the development process.. 17

**Apple Push Notification Service** Apple Push Notification service (APNs) is a service provided by Apple that enables developers to send push notifications to iOS, macOS, watchOS, and tvOS devices. It allows developers to send lightweight, timely, and relevant notifications to users even when the app is not running in the foreground. APNs acts as a mediator between the server and the device, ensuring secure and efficient delivery of notifications. To implement push notifications in an iOS app, developers need to integrate the APNs API and configure the necessary certificates and permissions in the Apple Developer Portal.. 34, 75

**backend** The part of a computer system or application that is not directly accessed by the user, typically responsible for storing and manipulating data. 20, 22, 25, 27, 30, 32–34, 37–39, 44, 49, 50, 52, 56

**Badges** In the context of this paper the term "Badges" is used to mean a "Badge" a user earns after completing a specific milestone. For example a user earns a badge for winning his first challenge. A badge could be compared to a medal someone earns after achieving some kind of goal.. 2, 4, 18, 29, 36, 41, 72, 80–83, 91, 94, 95, 98

**Banner** In the context of this project "Banner" is the upper half of the users profile. It displays the avatar, Badges and Tracked Data. The Banner can be viewed by other users in the challenge details. It's used to show off the achievements and experience of a user to his opponents.. 2, 18, 19, 41, 42, 69–72, 80, 84, 94, 95, 98, 102

**Blob** A term commonly used in computer science and data storage to refer to a binary large object. It represents a collection of binary data, typically stored as a single entity in a database or file system. Blobs can include various types of data, such as images, audio files, or serialized objects.. 71

**Challenge** In the context of this paper, a challenge is a sports game played between two of our users. A user can challenge another user to a certain sport activity. The challenged user then has the opportunity to perform the sport activity to his best abilities. It then gets sent back to the creator which then has to perform on it. It goes back and fourth until someone can't beat the other users score.. 2, 18–20, 29, 34, 40, 42, 58–62, 65, 66, 69, 72, 76–78, 80, 81, 90, 91, 94–96

**Collection** In the context of Firestore, a "Collection" is a grouping of documents that share a common path. It is a fundamental concept in Firestore's data model. Collections can contain zero or more documents, and each document within a collection is uniquely identified by its document ID. Collections provide a way to organize and query related data in Firestore. They are often used to represent entities or categories within an application, such as users, products, or posts. Firestore allows for hierarchical structuring of collections to create subcollections within a collection.. 29, 32, 35, 49, 50, 54, 57, 59, 76, 79, 80, 83

**Component** In the context of React Native, a "Component" is a reusable building block used to create user interfaces. It represents a self-contained piece of UI functionality that can be composed and reused across different parts of an application. Components in React Native are typically written in JavaScript or TypeScript and encapsulate a combination of visual elements, logic, and state management. They help promote code reusability, maintainability, and a modular approach to UI development.. 38, 39, 51, 72, 87

**Creator** In the context of this project the creator is always refered to as the person that created a challenge.. 2, 18, 29, 59

**Document** In the context of Firestore, a "Document" is a unit of data storage and retrieval. It represents a set of key-value pairs, where each key is a field name and each value is the corresponding field value. Documents are stored within collections and are uniquely identified by their document ID. Firestore uses a flexible, schema-less data model, allowing documents within a collection to have different structures. Documents can contain nested fields, arrays, and other complex data types. They serve as the main building blocks for organizing and retrieving data in Firestore.. 28, 29, 32, 49–51, 57, 66, 71, 76–80, 82, 83, 87

**Document Reference** In the context of Firestore, a "Document Reference" is an object that points to a specific document within a Firestore database. It represents a handle to a document and provides access to read, write, and listen to changes in the referenced document. A Document Reference consists of the collection path and the unique identifier (document ID) of the document it references. It allows developers to perform operations on a specific document, such as retrieving its data, updating fields, or deleting the document. Document References are commonly used for navigating and interacting with documents in Firestore.. 28, 59

**Expo** Expo is a free and open-source platform for building native mobile applications using JavaScript and React Native. It provides a set of tools, libraries, and services that simplify the development process, allowing developers to create high-quality cross-platform apps with ease. Expo offers features like easy app setup, hot reloading, and access to device APIs, making it a popular choice for rapid mobile app development.. 25, 34, 38, 44, 45, 48, 56, 65, 71, 75, 76, 92, 95, 107

**Firebase** Google Firebase is a comprehensive mobile and web application development platform that provides a wide range of backend services, including authentication, real-time database, cloud messaging, and more. It offers developers an easy way to build high-quality applications with built-in scalability, security, and offline support.. 25–27, 30, 32, 34, 36–39, 44–46, 48, 49, 53, 55–57, 85–91, 95, 96, 98, 107

**Firebase Admin SDK** The Firebase Admin SDK is a set of server-side libraries provided by Google Firebase that enables developers to build server applications that interact with Firebase services. It allows for programmatic access to Firebase functionality, including managing user authentication, reading and writing data in the Realtime Database, and interacting with Cloud Firestore and Cloud Storage. The Firebase Admin SDK supports multiple programming languages, such as Node.js, Java, Python, and Go, and provides powerful tools for server-side integration with Firebase services.. 33, 46, 85

**Firebase CLI Tools** The Firebase CLI Tools, also known as the Firebase Command-Line Interface (CLI), is a set of command-line tools provided by Google Firebase. It offers a convenient way to interact with Firebase services from the command line or terminal. With the Firebase CLI, developers can perform various tasks such as deploying web applications, managing Firebase projects, configuring security rules, and accessing Firebase features and services.

It simplifies common development tasks and enables automation and scripting capabilities for Firebase projects. The Firebase CLI is an essential tool for Firebase developers to streamline their workflow and manage Firebase resources efficiently.. 33, 44

**Firebase Cloud Functions** Firebase Cloud Functions is a serverless compute platform provided by Google Firebase. It allows developers to write and deploy server-side code in JavaScript or TypeScript that runs in response to events in Firebase services, such as changes to the database, user authentication, or file uploads. Firebase Cloud Functions enables developers to extend their Firebase applications with custom server-side logic without the need to manage infrastructure.. 26, 27, 33, 37, 44, 46, 52, 54, 57, 67, 76, 78, 80, 85, 95, 96

**Firebase Cloud Messaging** Firebase Cloud Messaging (FCM) is a cross-platform messaging solution that allows developers to reliably deliver messages and notifications to Android, iOS, and web applications. It provides a simple and reliable way to send messages to devices and handle downstream messages from them. FCM handles all aspects of message delivery, including message routing, security, and scaling, allowing developers to focus on the application logic. FCM supports various types of messages, including notification messages, data messages, and topic-based messaging.. 34, 75

**Firebase JavaScript SDK** The Firebase JavaScript SDK is a library provided by Google Firebase that allows developers to easily integrate Firebase services into web applications. It provides a set of client-side APIs and tools that enable real-time data synchronization, user authentication, cloud storage, and other Firebase features. The Firebase JavaScript SDK supports JavaScript and TypeScript and simplifies the process of interacting with Firebase services in web development projects.. 26–28, 32, 36, 48, 49, 53

**Firebase Storage** Firebase Storage is a cloud-based storage service provided by Google Firebase. It offers developers a secure and scalable solution for storing and serving user-generated content, such as images, videos, and other files, in their applications. Firebase Storage integrates seamlessly with other Firebase services, allowing developers to easily manage and access files from their web and mobile applications. It provides features like file uploads, downloads, and security rules to control access to stored content.. 28, 36, 71, 80, 83

**Firestore** Firestore is a flexible and scalable NoSQL document database provided by Google Firebase. It is designed to store, sync, and query data for client- and server-side applications. Firestore offers real-time data synchronization, offline capabilities, and powerful querying with automatic scaling, making it suitable for building modern, cloud-based applications.. 27–29, 32, 34–36, 38, 46, 48–53, 57, 59, 70–72, 80, 83, 86, 87, 95, 96, 99

**Floating Action Button** A Floating Action Button (FAB) is a circular button commonly used in user interfaces to promote a primary action. It is a prominent and elevated button that "floats" above the content, usually positioned at the bottom right corner of the screen. The FAB provides a quick and easily accessible way for users to perform the most important action within an application. It is often used to trigger actions such as creating a new item, composing a message, or initiating a call to action. The appearance and behavior of the FAB can vary depending on the platform and design guidelines.. 58, 62

**Jest** Jest is a popular JavaScript testing framework developed by Facebook. It is widely used for testing JavaScript applications, including frontend applications built with frameworks like React, Vue, and Angular, as well as backend Node.js applications. Jest provides a comprehensive set of features for writing and running tests, including test runners, assertion libraries, mocking capabilities, and code coverage analysis. It offers a simple and intuitive syntax for writing test cases and supports powerful features such as snapshot testing and parallel test execution. Jest is known for its fast and efficient test execution, making it a preferred choice for many JavaScript developers.. 89

**Jira Cloud** Jira Cloud is a cloud-based project management and issue tracking solution provided by Atlassian. It offers a comprehensive set of features for teams to plan, track, and manage their projects, tasks, and workflows. Jira Cloud provides a web-based interface accessible from any browser, eliminating the need for self-hosted infrastructure. It offers features such as customizable issue tracking, agile project management, real-time collaboration, reporting and analytics, and integration capabilities with other popular tools. Jira Cloud is a scalable and flexible solution that caters to teams of all sizes and industries, enabling effective project management and streamlined workflows.. 17

**Method Swizzling** A technique used in object-oriented programming to dynamically change the behavior of methods in a class at runtime. It involves swapping the implementation of one method with another, usually within a class hierarchy. This technique is commonly used in languages like Objective-C and Swift, where dynamic runtime features are supported. Method swizzling can be employed for various purposes, such as adding new functionality, modifying existing behavior, or intercepting method calls for debugging or profiling. However, method swizzling should be used judiciously, as it can introduce complexity and make code harder to understand and maintain. Improper use of method swizzling can lead to unexpected behavior and difficult-to-debug issues.. 75

**Multiplatform** In the context of mobile development, "Multiplatform" refers to the ability to develop and deploy applications that can run on multiple platforms, such as iOS and Android, using a single codebase. It typically involves utilizing frameworks or technologies that allow developers to write shared code in a common programming language, while still maintaining a native look and feel on each platform. Multiplatform development can help reduce development time, effort, and costs by enabling code reuse across different platforms.. 15, 24, 71

**NoSQL** NoSQL, short for "Not only SQL," is a term that refers to a type of database management system that diverges from the traditional relational database model. Unlike relational databases, NoSQL databases are designed to handle large volumes of unstructured or semi-structured data, providing high scalability, performance, and flexibility. NoSQL databases use various data models, such as key-value, document, columnar, and graph, to store and retrieve data. They are often used in modern applications requiring agility, horizontal scalability, and the ability to handle rapidly changing data schemas.. 25, 29, 35

**Opponent** In the context of this paper, a opponent is refered to as the other player in a challenge, from the perspective of the currently logged in user. It is also refered to as the person that received the challenge, but did not create it.. 2, 18, 29, 41, 50, 58–60, 62, 66, 69, 72, 77, 94

**Profile** In the context of this project "Profile" is the dashboard of the user where the user can see various statistics and his avatar. 29, 36, 57, 68–72, 91

**React Native** React Native is an open-source mobile application framework developed by Facebook, Inc. It is used to develop applications for Android, iOS, Web and UWP by enabling developers to use React along with native platform capabilities.. 24, 25

**Real-time Updates** Real-time Updates refer to the capability of Firebase to provide instantaneous and synchronized data updates across connected devices or clients. Firebase offers real-time updates by utilizing a real-time database, where changes made to the data are immediately propagated to all connected clients. This enables real-time collaboration, instant messaging, live data synchronization, and other real-time features in Firebase-powered applications. Real-time Updates in Firebase are achieved through web sockets or other efficient data synchronization mechanisms, ensuring that all connected clients have the latest data in real-time.. 35, 36, 49, 50, 69–71, 78, 88–90

**Realtime Database** Firebase Realtime Database is a NoSQL cloud-hosted database provided by Google Firebase. It allows developers to store and synchronize data in real time between clients and servers. The database uses a JSON-based structure and provides real-time data synchronization, offline support, and powerful querying capabilities. Firebase Realtime Database simplifies the development of collaborative and real-time applications by handling data synchronization and conflict resolution automatically.. 34, 35

**Single Page Application** In the context of React Native, a "Single Page Application" (SPA) is a web application that dynamically updates its content without requiring a full page reload. In a React Native SPA, the user interface is built using a single HTML page, and navigation between different views or screens is managed through JavaScript, typically using a library like React Navigation. SPAs provide a seamless and responsive user experience by loading data and updating the UI asynchronously, resulting in faster and more interactive applications.. 39

**System Context** The environment in which a system operates. This encompasses the boundaries of the system, interactions with users and other systems, and external conditions impacting system operations. Understanding the system context is vital in system design and analysis, as it helps to clarify the operational conditions and requirements for the system.. 15

**Tracked Data** In the context of this paper the term "Tracked Data" is used to mean all the data we track of a user, while he plays challenges on our Application. "Tracked Data" contains information like, challenges played, challenges won and the favorite opponent of a user. This data is then displayed or used for further calculations. 4, 69, 72, 76, 77, 81, 89, 91

## Acronyms

**API** Application Programming Interface. 15, 16, 25

**DoD** Definition of Done. 23, 113, 114

**DoR** Definition of Ready. 113

**NFR** Non-Functional Requirements. 19, 21, 23

# Contents

**1 Starting Position** **12**

**2 Conceptual Formulation** **13**

**3 Framework Conditions** **14**

**4 System Context** **15**

**5 Requirements** **17**
5.1 Functional Requirements . . . 17
5.1.1 Use Case Description . . . 17
5.1.2 Evolution of Functional Requirements . . . 19
5.2 Non-Functional Requirements . . . 21
5.2.1 Disregarded and Untested NFRs . . . 23
5.2.2 Verification . . . 23
5.3 Optional Requirements . . . 23

**6 Design and Architecture** **24**
6.1 Container Diagram . . . 24
6.1.1 Client . . . 24
6.1.2 Backend . . . 25
6.1.3 External Dependencies . . . 25
6.2 Component Diagram . . . 26
6.2.1 Backend Access . . . 27
6.3 Domain Model . . . 28
6.3.1 Types of References . . . 28
6.3.2 Description . . . 29
6.4 Backend - Firebase . . . 30
6.4.1 Authentication . . . 31
6.4.2 Functions . . . 32
6.4.3 Firebase Cloud Messaging And Notifications . . . 34
6.4.4 Database - Firestore . . . 34
6.4.5 Firebase Storage . . . 36
6.4.6 Scaling . . . 36
6.4.7 Pricing . . . 37
6.5 App Architecture . . . 38
6.5.1 Multiplatform - Expo . . . 38
6.5.2 Structure . . . 38
6.5.3 UI Design . . . 39
6.5.4 Coding Style Guide . . . 43
6.6 Deployment . . . 44
6.6.1 Firebase Functions . . . 44
6.6.2 Mobile App . . . 44

6.7 Security . . . 45
6.7.1 Firestore Access . . . 46
6.7.2 Storage Access . . . 46
6.7.3 Input Validation . . . 47

**7 Implementation 48**
7.1 Firebase Initialization . . . 48
7.1.1 Setting Up Services . . . 48
7.1.2 Frontend Configuration . . . 48
7.1.3 Firebase SDK . . . 49
7.2 Frontend Firestore Access . . . 49
7.2.1 Read Operations and Real-time Updates . . . 49
7.3 Firebase Functions . . . 52
7.3.1 Implemented Functions . . . 54
7.3.2 Firebase Admin SDK . . . 54
7.4 Authentication . . . 54
7.4.1 Setup . . . 54
7.4.2 Implementation . . . 55
7.4.3 Firebase Phone Auth Limitations . . . 57
7.5 Challenge Creation . . . 58
7.5.1 Opponent Selection . . . 59
7.5.2 Challenging random user . . . 61
7.6 Home Screen . . . 61
7.7 Challenge Details . . . 63
7.7.1 Challenge History . . . 63
7.8 Perform a challenge . . . 64
7.8.1 Implement new Challenge . . . 67
7.8.2 User did not Perform on Challenge . . . 68
7.8.3 Shortcomings . . . 68
7.9 Profile . . . 68
7.9.1 10 Week History . . . 69
7.9.2 Profile Picture . . . 70
7.9.3 Banner . . . 72
7.9.4 Component Structure . . . 72
7.10 Chat . . . 72
7.11 Rankings . . . 73
7.11.1 Extending the Rankings . . . 74
7.12 Push Notifications . . . 75
7.13 Tracked Data . . . 76
7.13.1 Challenge completed . . . 76
7.13.2 User Performed on Challenge . . . 78
7.13.3 Track new Data . . . 78
7.14 Badges - Milestones . . . 80
7.14.1 Sequence . . . 81
7.14.2 Add new Badge . . . 83
7.14.3 Badge Copyrights . . . 84
7.15 Localization . . . 84
7.16 Technical Test Dokumentation . . . 85
7.16.1 Firebase Cloud Functions Testing . . . 85
7.16.2 Frontend Testing . . . 87

7.16.3 Testing concessions . . . . . 90
7.16.4 Manual Test Protocol . . . . . 90
7.17 Deployment . . . . . 92
7.17.1 App Store . . . . . 92
7.17.2 Google Play Store . . . . . 92

**8 Result** **94**

**9 Conclusion** **98**
9.1 Needs to be Adressed . . . . . 98
9.2 Future Vision . . . . . 99

**10 Project and Time Management** **101**
10.1 Project Plan . . . . . 101
10.1.1 Deviations . . . . . 102
10.2 Project organization . . . . . 103
10.2.1 Issue management . . . . . 103
10.2.2 Time tracking . . . . . 103
10.2.3 Division of Tasks . . . . . 104
10.3 Risk Management . . . . . 104
10.3.1 Realized risks . . . . . 106
10.4 Test Concept . . . . . 108
10.4.1 Additional . . . . . 109
10.4.2 System . . . . . 109
10.4.3 Integration . . . . . 109
10.4.4 Component . . . . . 109
10.4.5 Unit . . . . . 110
10.4.6 Deviations . . . . . 110
10.4.7 End User Usability Tests . . . . . 110
10.5 Definition of Ready / Definition of Done . . . . . 113

**11 Meeting Minutes** **119**

**12 Personal Reports** **135**

**13 Screenshots** **136**

**14 Usability Test Protocols** **141**

**15 Task** **149**

# 1 Starting Position

As gymnasium, sports, video game, and IT enthusiasts, we are interested in projects combining our passions. We have worked on a previous project with the same partner and advisor and achieved great results, and we are more than thankful to be able to work with them again for our bachelor thesis.

A mobile application that combines sports with challenges and games in a digital aspect is something we think is missing in the industry and a great opportunity to encourage all kinds of people to compete against each other in friendly competitions.

This project aims to build a mobile application with which a player can challenge other players to play in friendly competitions. The competitions consist of exercise, time duration, and players. The goal is to beat the score of your opponent. Each challenge has to be performed in a given time frame and will be sent back to the original challenger, who has to beat the score again. As long as no user can beat the other user's score, the challenge will be sent back and forth until one user cannot beat the other user's score. He will lose if a player cannot beat the score or exceeds the given time.

# 2 Conceptual Formulation

To motivate people to exercise more sports, we want to implement a mobile application where friends and groups of people can challenge each other. A player creates a challenge, including the exercise, duration, and opponent, to determine who can do more repetitions of a given exercise in a specific duration. The opponent can do the challenge and return the challenge with his score to his original challenger. This cycle is repeated until someone cannot defeat his opponent's score or the timer runs out to perform the challenge. On the dashboard, the player can see his statistics regarding how often he wins or loses, his history for a given challenge type, and an overview with whom he has played challenges the most.

# 3 Framework Conditions

This project is part of the bachelor thesis which is required for the eligibility of the bachelor. The planned time budget for this project is 360 hours per person and equals to 12 ECTS.

# 4 System Context

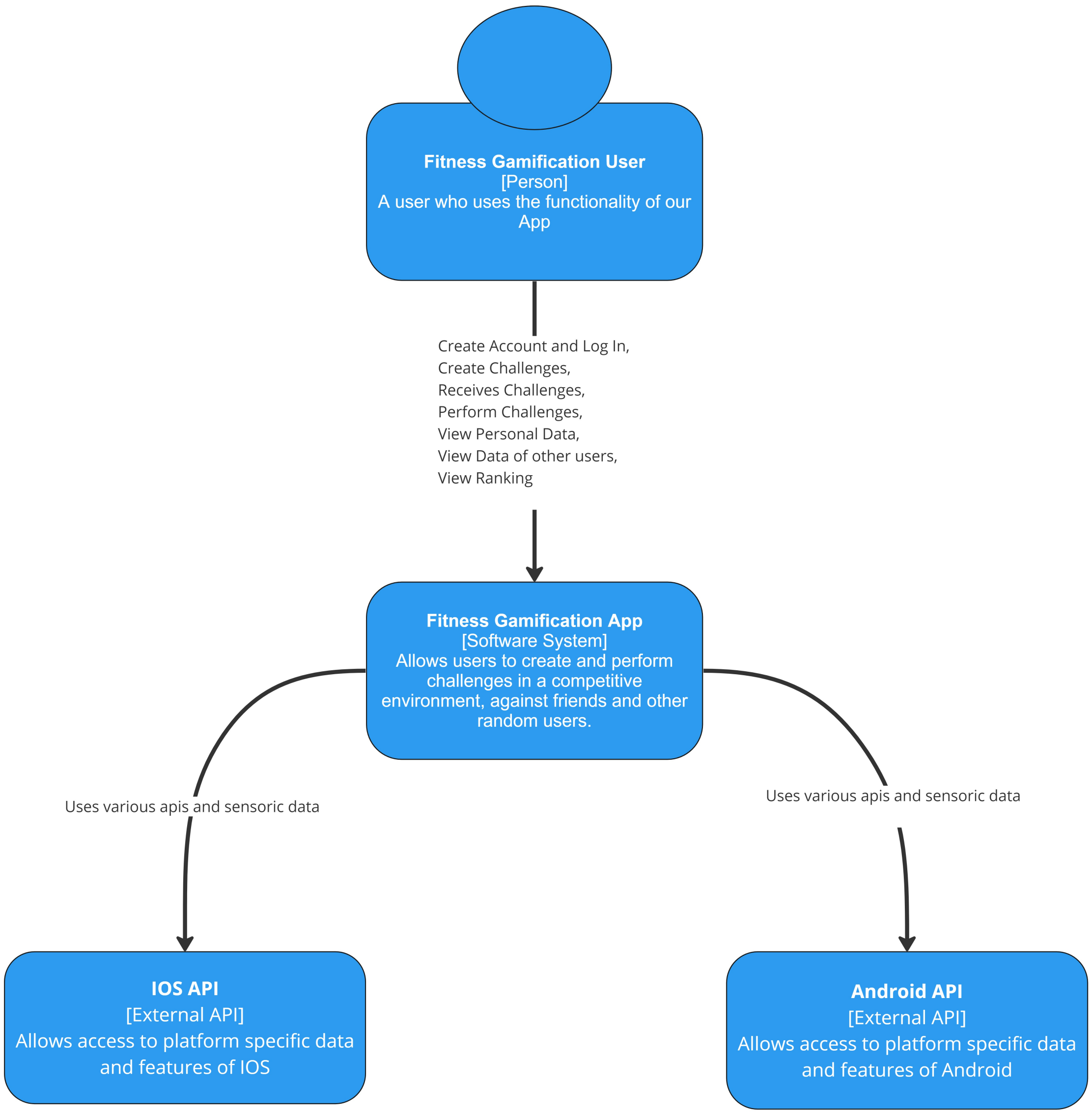

**Fitness Gamification User**
[Person]
A user who uses the functionality of our App
Create Account and Log In,
Create Challenges,
Receives Challenges,
Perform Challenges,
View Personal Data,
View Data of other users,
View Ranking
**Fitness Gamification App**
[Software System]
Allows users to create and perform challenges in a competitive environment, against friends and other random users.
Uses various apis and sensoric data
Uses various apis and sensoric data
**IOS API**
[External API]
Allows access to platform specific data and features of IOS
**Android API**
[External API]
Allows access to platform specific data and features of Android

Figure 4.1: System Context

The above graphic 4.1 shows the System Context of our application. The person "Fitness Gamification User" represents our software system user. We only have one type of user since there is currently no plan to implement users with different functionality or elevated privileges. Our Software System, "Fitness Gamification App," contains all the code and infrastructure we control, manage, and develop. The System will run as an app on both Android and IOS, allowing for a Multiplatform approach. We have two external Systems, specifically, two external Application Programming Interfaces (APIs) that we use. These are platform-specific, and which API is used depends on the app's platform. These APIs are primarily used to create a cross-platform app that runs on both these platforms, and the core API we will use from them is the sensor API, but there are other APIs we use such as contacts.

# 5 Requirements

We use Agile Methodologies to keep track of our functional requirements. These are tracked as User-Stories or issues in Jira Cloud. We prioritize the User-Stories and issues in our Jira Cloud-Backlog.

## 5.1 Functional Requirements

This section will show all our project's Functional Requirements as Use Cases. It shows the latest version of the functional requirements. Any changes made to functional requirements in the duration of the project will be covered in 5.1.2

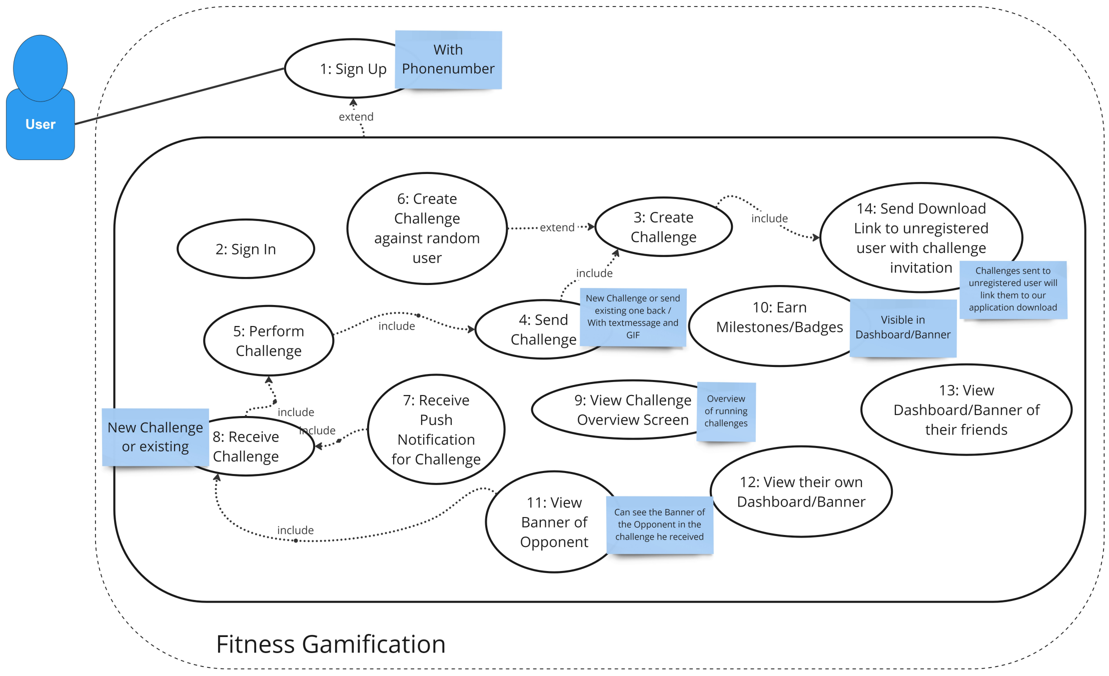

User
1: Sign Up
With Phonenumber
extend
2: Sign In
6: Create Challenge against random user
extend
3: Create Challenge
include
14: Send Download Link to unregistered user with challenge invitation
Challenges sent to unregistered user will link them to our application download
include
New Challenge or send existing one back / With textmessage and GIF
5: Perform Challenge
include
4: Send Challenge
10: Earn Milestones/Badges
Visible in Dashboard/Banner
include
include
New Challenge or existing
8: Receive Challenge
7: Receive Push Notification for Challenge
9: View Challenge Overview Screen
Overview of running challenges
13: View Dashboard/Banner of their friends
12: View their own Dashboard/Banner
include
11: View Banner of Opponent
Can see the Banner of the Opponent in the challenge he received
Fitness Gamification

Figure 5.1: Use Case Diagram Final

Our Application only contains a single type of user. This user is anyone that downloads our Application or receives a link to download the Application. All use cases extend the "1: Sign Up" use case since, to use the functionality of our App, a user needs to have an account.

### 5.1.1 Use Case Description

| Nr | Use Case | Description |
|---|---|---|

| | | |
|---|---|---|
| 1 | Sign Up | The user expresses the wish to be registered on the platform. Registration will contain their Name and their phone number. |
| 2 | Sign In | The user wants to be able to sign into our application after he creates an account. |
| 3 | Create Challenge | The user wishes to create a "Challenge" and send it to another user. While creating the Challenge, the user can select the type of challenge/exercise, the time to complete the exercise and choose a single user. After creating the Challenge, the Opponent will be prompted to perform the exercise in the given period. After completing it, it will be returned to the Challenge's Creator. |
| 4 | Send Challenge | The user wants to be able to send a Challenge to a user. Sending a Challenge can be either one he just created and completed or one he received and responded/participated in. When receiving the Challenge, the user will be able to perform his best for the given exercise and send it back to the user he received the Challenge from. |
| 5 | Perform Challenge | The user wants to be able to perform a Challenge. That means he wants to be able to participate in the given exercise type and that our app will record his result and, when completed, sent it to the other user |
| 6 | Create Challenge against random user | The user wants to be able to create and perform a Challenge against a random user that also uses the App. |
| 7 | Receive Push Notification for Challenge | A user wants to be notified when he is challenged to a new Challenge or if it is his turn to attempt it. The user also wants to be notified if he won the Challenge if no one could beat his score. |
| 8 | Receive Challenge | Users can receive a Challenge from a user. There are two distinct cases when receiving the Challenge in this use case. The user receives a Challenge that was either newly created or the previous attempt of the user was topped, and he will be prompted to attempt to beat the current high score of the Challenge. In the other case, the user receives the Challenge where the other participants were unable to best his attempt/score, and he will be declared a winner and unable to respond/attempt to best the score again. |
| 9 | View Challenge Overview Screen | The user wants to be able to view a screen (home page) where all his running challenges are listed and where he can open specific challenges and view their details. The user will see if it is his turn to perform the Challenge or if he has to wait for the other user to complete his turn |
| 10 | Earn Milestones/Badges | The user wishes to earn Badges for completing certain milestones and achievements, which he can then view in his Banner on his Dashboard and show to other users. |
| 11 | View Banner of Opponent | The user wishes to see the Banner of his Opponent, where he can see the earned Badges and some tracker of statistics of his Opponent so that he can guess his Opponent's skills. |
| 12 | View their own Dashboard/Banner | The user wants to be able to view a Dashboard of his most important information, including the Banner that will be displayed to his opponents. |

| | | |
|---|---|---|
| 13 | View Dashboard/Banner of their friends | The user wants to be able to view the Dashboard of his friends. |
| 14 | Send Download Link to unregistered User | A user wants to be able to challenge users that are currently not registered with the application. In that case, the challenged user will receive a download link to the store page of our application. |

Table 5.1: Use Case Description

### 5.1.2 Evolution of Functional Requirements

The functional requirements represent the requirements provided in the initial task formulation. Some requirements were removed, changed, or added during the project's first few weeks. The changes done to them and the reason for that will be listed here. The Use Cases from the original task formulation looked like this:

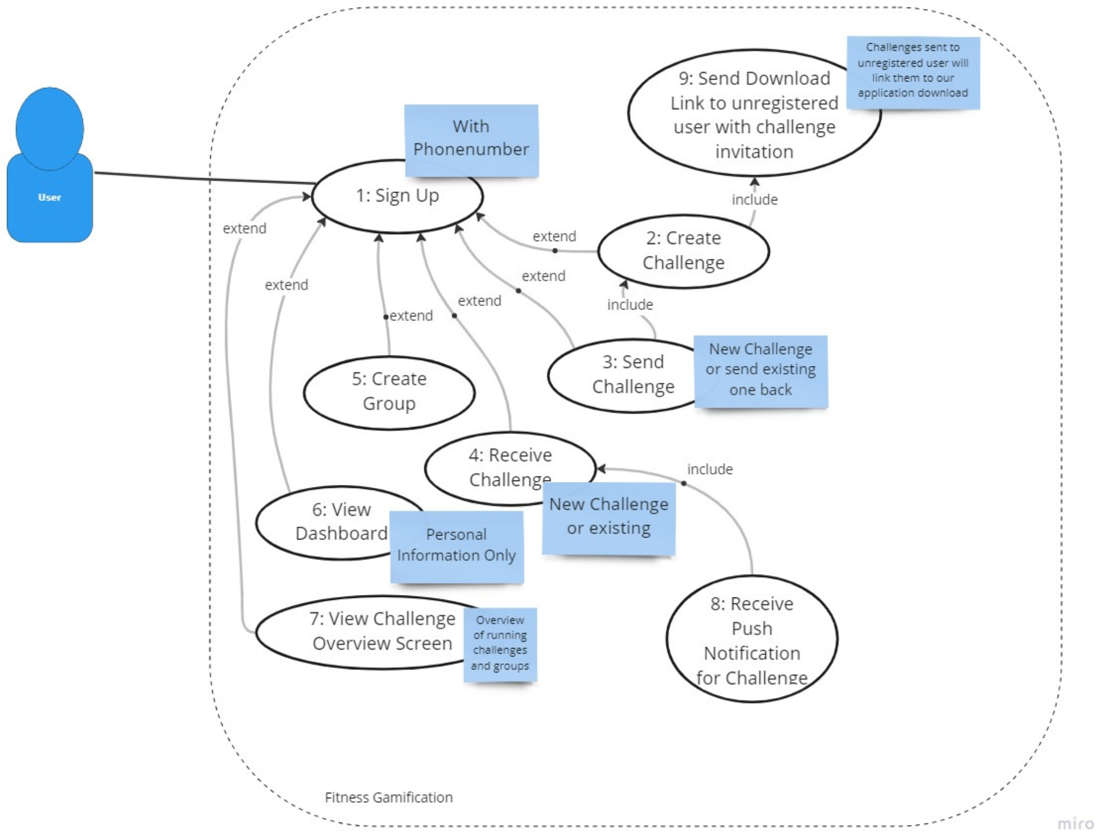

User
1: Sign Up
With Phonenumber
2: Create Challenge
3: Send Challenge
New Challenge or send existing one back
4: Receive Challenge
New Challenge or existing
5: Create Group
6: View Dashboard
Personal Information Only
7: View Challenge Overview Screen
Overview of running challenges and groups
8: Receive Push Notification for Challenge
9: Send Download Link to unregistered user with challenge invitation
Challenges sent to unregistered user will link them to our application download
extend
include
Fitness Gamification
miro

Figure 5.2: Use Cases Version 1

In the first three weeks, we worked on finalizing the requirements, Non-Functional Requirements (NFR)s, and setting up infrastructure. When we discussed the functional requirement in our meetings with our professor and customer, we realized that some of the functional requirements were more complex than we thought in the beginning and that many edge cases would arise. The functional requirements that caused us to overthink if we should implement them had to do with groups. In short, a user could create a group, manage that group, send a Challenge to all group

members, and view the ranking inside a group. When trying to specify these requirements more, we noticed that there would be some edge cases and questions regarding the flow and management of the group. A lot of our time would have to go into finding solutions for these edge cases and also the group's management with invitations, removing members, etc. That means a lot of the work would go into the backend away from the UI, and the UI would suffer for that reason. We were not as interested in the group management personally but more in the Challenges, achievements, and being able to focus more on the look and feel of our application.
The specific issues that arose are:

- How do we handle participants in a group that do not participate in the Challenge (aka Bums)?
- How is the flow of the Challenge inside of a group? When is a winner selected? Is there a fixed amount of turns?
- How will placement in a group Challenge be counted in the global ranking?

After discussing these issues we made two proposals to our customers. One proposal where we implement other functionalities instead of groups. The other proposal was how to address the "problems" that arose with the groups. with our recommendation being implementing other use cases and disregarding the group functionality. We will show the mind map 5.3 with alternative functionalities that we used to specify our new functional requirements. This suggesting was accepted and resulted in the functionality and use cases listed in the section 5.1

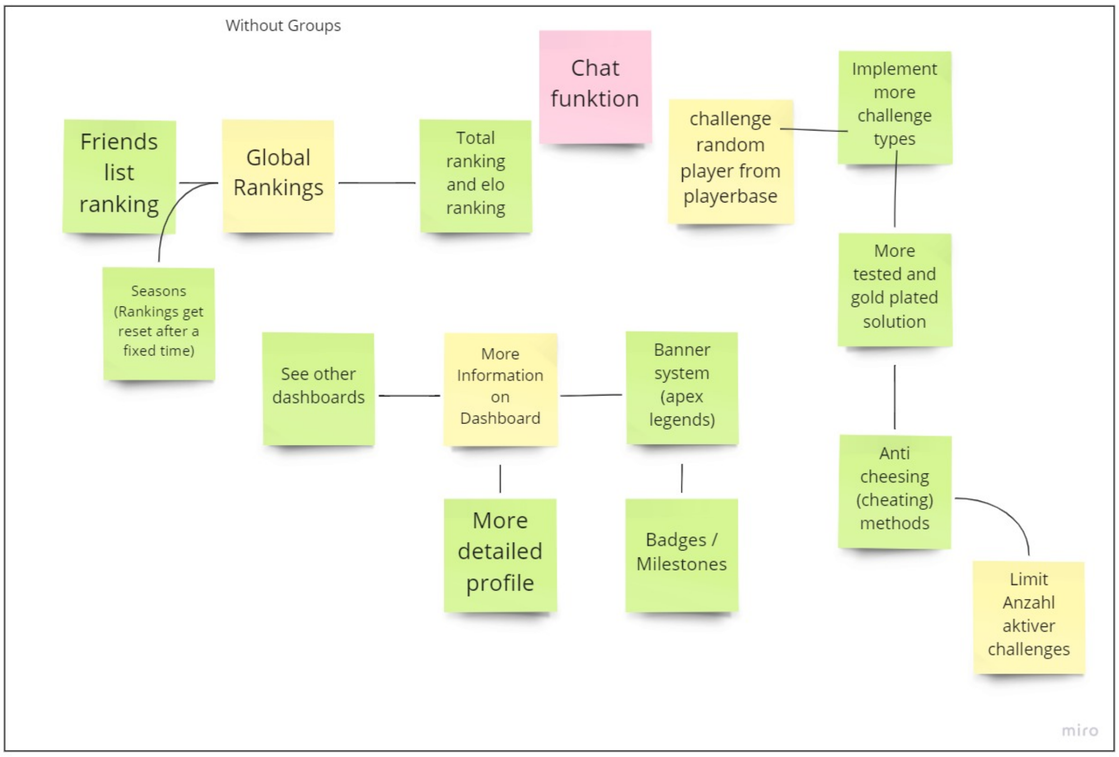

Without Groups
Chat funktion
Implement more challenge types
challenge random player from playerbase
Friends list ranking
Global Rankings
Total ranking and elo ranking
Seasons (Rankings get reset after a fixed time)
More tested and gold plated solution
See other dashboards
More Information on Dashboard
Banner system (apex legends)
Anti cheesing (cheating) methods
More detailed profile
Badges / Milestones
Limit Anzahl aktiver challenges
miro

Figure 5.3: Mind Map Alternatives

## 5.2 Non-Functional Requirements

We express our Non-functional requirements as defined in FURPS+. While this project progresses, this table will be updated with new NFR's according to the knowledge of the domain and its technologies. We will give each NFR a priority of must, should, can. Where must is any NFR that has to be implemented and taken into account and tested. Should is a NFR that as the name suggests should be considered and implmented but in some cases can be disregarded and won't be tested as heavily as the should requirements. And can are all the truly optional NFR's. All NFR's listed here are a must unless explicitly stated otherwise. Below is the List of NFR's in our project:

| Type | Description | Acceptance Criteria | Priority |
|---|---|---|---|
| Functionality | The App must run on both Android and IOS | Manual Testing and development on both platforms | Must |
| Functionality | The Backend is deployed and frontend must be published in the App Store and Play Store | Manual Testing and development on both platforms | Must |
| Functionality | All exceptions are handled and logged. | All Exceptions occurring in the tests are handled and logged. Furthermore, all possible exceptions should be tested. | Must |
| Reliability | Errors do not cause Systemfailures, but generate an error message and re- set the system to a previous state | Automated Testing / Manual Deployment Testing | Must |
| Performance | backend is able to handle 50 requests per second | Tested with Performance Testing Framework | Can |
| Performance | App content should not take longer then 150ms to load | Performance Testing Framework | Should |
| Performance | The Database should be able to support 10000 challenges and 2000 user | Testing at scale (manually) | Can |
| Supportability | The Business logic must be built in a modular way so that it can be easily extended | Business Reviews | Must |
| Supportability | Backend-API must be tested with an API-Testing tool | Api Testing tool | Must |
| Usability | At least three test users out of 4 should rate the Application with a grade of 8 from 10 (layout, responsiveness, colour, content) | Usability Tests | Should |
| Usability | Push notifications should be sent ongoing | Unit and Manual Testing | Must |
| Others (Security) | Data entered in input fields must be validated before it is passed to the backend | SQL Injection tests | Must |
| Others (Security) | User passwords are not save in plain text | Review Database records | Must |
| Others (Security) | Data should only be visible to users that should have access to it | Manually tested | Must |
| Others | Implemented functionality (backend, Database, Frontend) must be deployed | CI/CD | Must |

Table 5.3: Non-Functional Requirements

### 5.2.1 Disregarded and Untested NFRs

In the course of the project we decided to disregard some of the NFR, for various reasons which are mainly driven by 10.3.1. The NFRs that were not implemented can be viewed in 8.

### 5.2.2 Verification

If possible the NFRs will be checked automatically via automated tests. If that is not a possibility we will check each NFR after each sprint, and if not all are met the Sprint does not meet the DoD and has to be fixed as soon as possible.

## 5.3 Optional Requirements

This section describes the optional and additional requirements that were either known at the beginning of the project or were ideas and inputs that came up during the project's duration. They are not part of the functional requirements and have low priority.

| Type | Description |
|---|---|
| Functional | Implement additional challenge types |
| Functional | User is able to delete his account and data associated with his account |
| Functional | Authentication possible with Google / Facebook / Apple-Account |
| Functional | User is able to capture challenge types manually that are not implemented by our application |
| Functional | User is able to view a timeline of all his challenges |
| Functional | User is able to send a video message with the challenge |
| Non-Functional | Implementation of anti cheating measures |

Table 5.5: Optional Requirements

# 6 Design and Architecture

## 6.1 Container Diagram

The container diagram 6.1 shows what technologies we chose to implement our application and how the code is split up, and how the different components interact with eachother. It should be viewed in the context of our system context 4.1

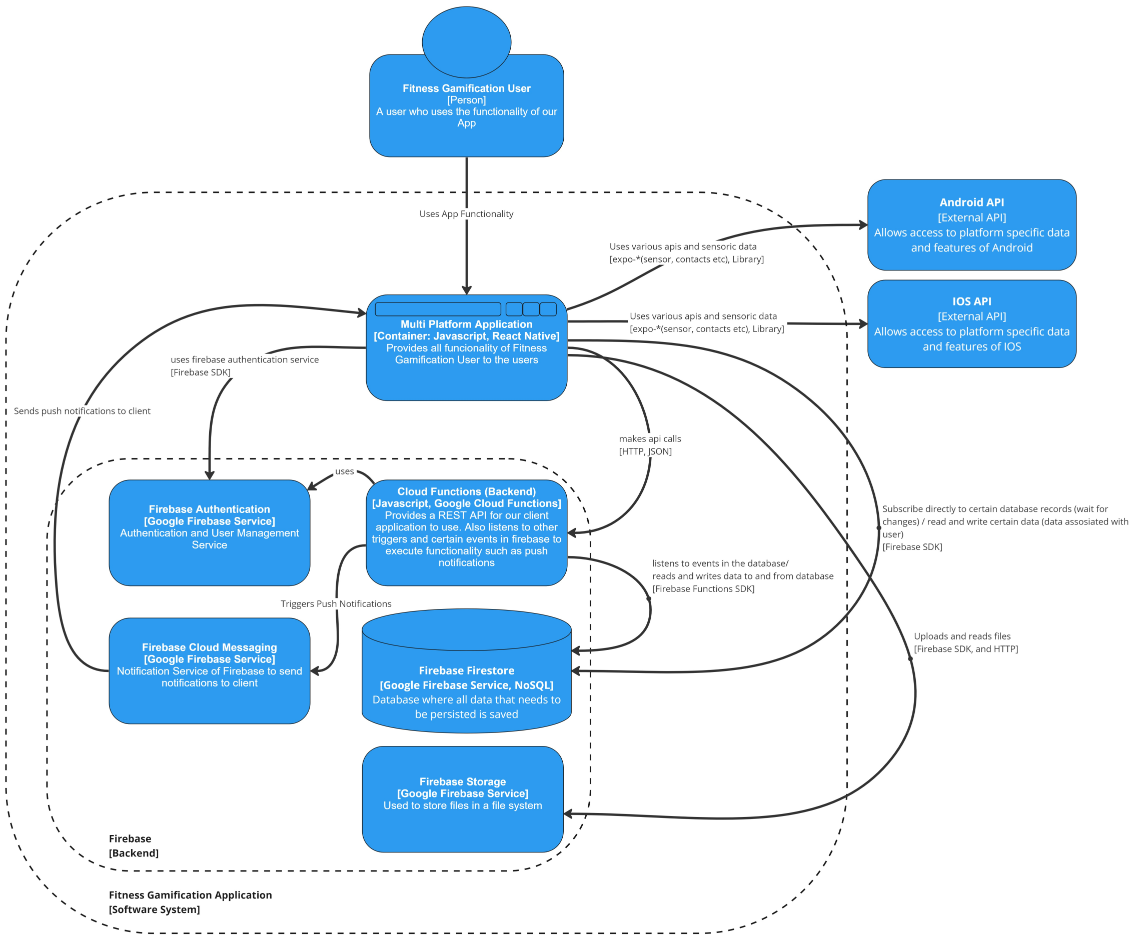

Fitness Gamification User
[Person]
A user who uses the functionality of our App
Uses App Functionality
Android API
[External API]
Allows access to platform specific data and features of Android
Uses various apis and sensoric data
[expo-*(sensor, contacts etc), Library]
IOS API
[External API]
Allows access to platform specific data and features of IOS
Uses various apis and sensoric data
[expo-*(sensor, contacts etc), Library]
Multi Platform Application
[Container: Javascript, React Native]
Provides all funcionality of Fitness Gamification User to the users
uses firebase authentication service
[Firebase SDK]
Sends push notifications to client
makes api calls
[HTTP, JSON]
Subscribe directly to certain database records (wait for changes) / read and write certain data (data assosiated with user)
[Firebase SDK]
uses
Firebase Authentication
[Google Firebase Service]
Authentication and User Management Service
Cloud Functions (Backend)
[Javascript, Google Cloud Functions]
Provides a REST API for our client application to use. Also listens to other triggers and certain events in firebase to execute functionality such as push notifications
listens to events in the database/
reads and writes data to and from database
[Firebase Functions SDK]
Triggers Push Notifications
Firebase Cloud Messaging
[Google Firebase Service]
Notification Service of Firebase to send notifications to client
Firebase Firestore
[Google Firebase Service, NoSQL]
Database where all data that needs to be persisted is saved
Uploads and reads files
[Firebase SDK, and HTTP]
Firebase Storage
[Google Firebase Service]
Used to store files in a file system
Firebase
[Backend]
Fitness Gamification Application
[Software System]

Figure 6.1: Container Diagram

### 6.1.1 Client

Our mobile client is implemented using React Native a Multiplatform language that allows us to share one codebase for several platforms; in our case, these would only be Android and IOS. React Native could be separated into different containers since its architecture is very complex, but that is not our goal to show the internal architecture of React Native. We are not writing separate code for Android or IOS, so there is no need to split it into two containers for each platform. We only have very few lines of platform-specific code, and this is because we use the Expo toolset on top of React Native that abstracts the platform-specific API calls and allows us to access android and ios API in the same way. More on Expo in chapter 6.5.1

### 6.1.2 Backend

As the backend we chose Firebase, a so-called backend as a service provided by Google. For a detailed explanation of what Firebase is and why we chose it, refer to 6.4. The container diagram shows the various Firebase services we use in our application. The services we use are:

- Firebase Authentication: Firebase service that implements authentication logic, and which we use to authenticate our users. 6.4.1.
- Firebase Firestore: Firebase NoSQL Database service. The client will subscribe, read and write data directly from the client. 6.4.4
- Firebase Cloud Messaging: Service that is used to send push notifications to our client. 6.4.3
- Google Cloud Functions for Firebase: Code that cannot be run in the client and code that gets triggered by listening to certain events. These are on-demand functions, which means a "instance" gets spun up every time a function is triggered, and after that, it gets torn down. Contains mostly computation heavy code and calculations and general operations that don't belong in the client code. 6.4.2
- Firebase Storage: Service that allows us to store files on the google cloud. 6.4.5

### 6.1.3 External Dependencies

We use two "external resources". These are various apis from the android and IOS platforms. The APIs we use vary from contacts to sensors, notifications, etc. We always use a Expo abstraction library of the API which allows us to not write separate code for the two platforms.

## 6.2 Component Diagram

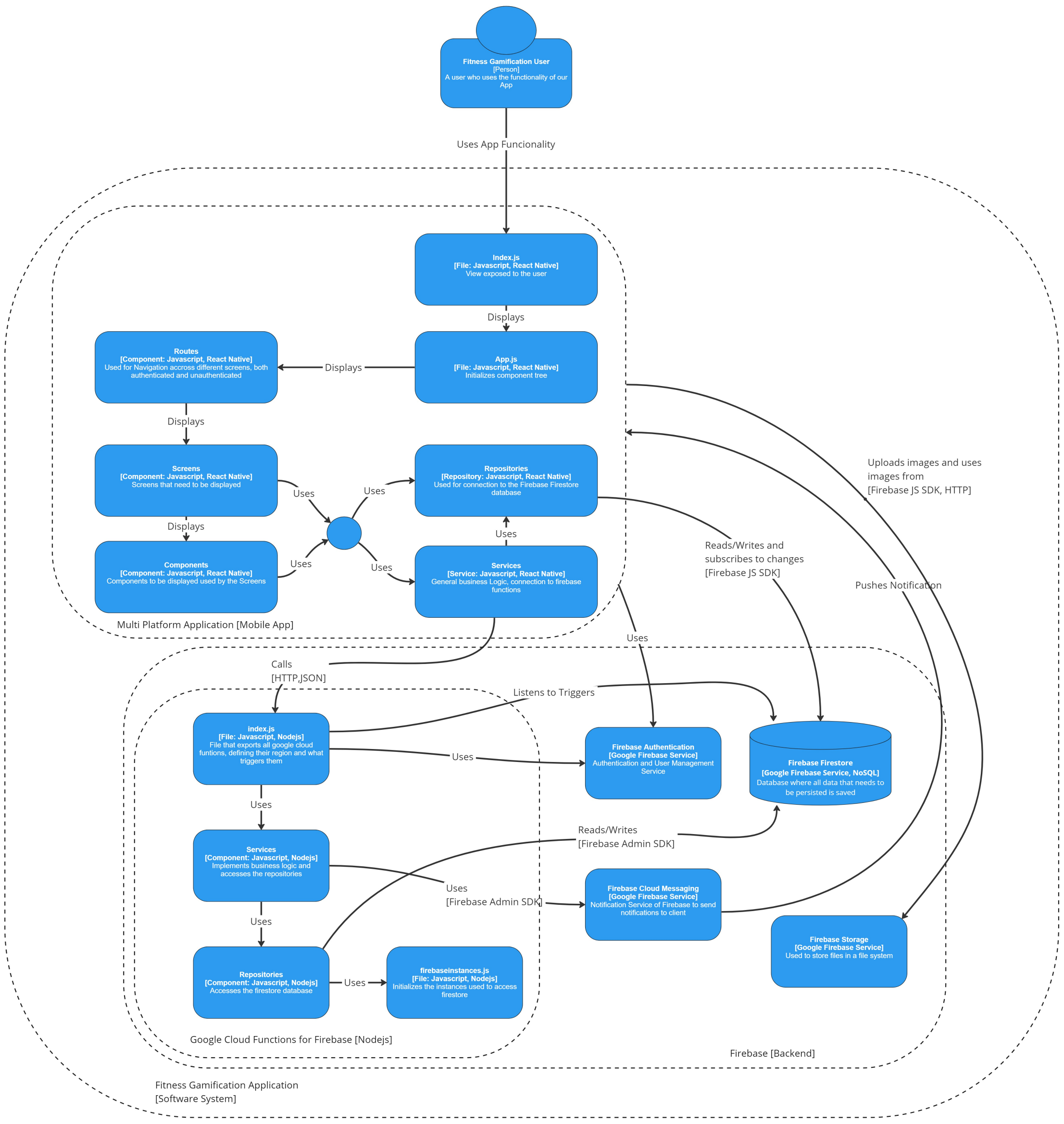

Fitness Gamification User
[Person]
A user who uses the functionality of our App
Uses App Funcionality
Index.js
[File: Javascript, React Native]
View exposed to the user
Displays
App.js
[File: Javascript, React Native]
Initializes component tree
Displays
Routes
[Component: Javascript, React Native]
Used for Navigation accross different screens, both authenticated and unauthenticated
Displays
Screens
[Component: Javascript, React Native]
Screens that need to be displayed
Displays
Components
[Component: Javascript, React Native]
Components to be displayed used by the Screens
Uses
Uses
Uses
Uses
Repositories
[Repository: Javascript, React Native]
Used for connection to the Firebase Firestore database
Uses
Services
[Service: Javascript, React Native]
General business Logic, connection to firebase functions
Multi Platform Application [Mobile App]
Uploads images and uses images from
[Firebase JS SDK, HTTP]
Reads/Writes and subscribes to changes
[Firebase JS SDK]
Pushes Notification
Uses
Calls
[HTTP,JSON]
Listens to Triggers
index.js
[File: Javascript, Nodejs]
File that exports all google cloud funtions, defining their region and what triggers them
Uses
Firebase Authentication
[Google Firebase Service]
Authentication and User Management Service
Firebase Firestore
[Google Firebase Service, NoSQL]
Database where all data that needs to be persisted is saved
Uses
Services
[Component: Javascript, Nodejs]
Implements business logic and accesses the repositories
Reads/Writes
[Firebase Admin SDK]
Uses
[Firebase Admin SDK]
Firebase Cloud Messaging
[Google Firebase Service]
Notification Service of Firebase to send notifications to client
Firebase Storage
[Google Firebase Service]
Used to store files in a file system
Uses
Repositories
[Component: Javascript, Nodejs]
Accesses the firestore database
Uses
firebaseinstances.js
[File: Javascript, Nodejs]
Initializes the instances used to access firestore
Google Cloud Functions for Firebase [Nodejs]
Firebase [Backend]
Fitness Gamification Application
[Software System]

Figure 6.2: Component Diagram

Figure 6.2 represents the structure of our frontend code, our Firebase Cloud Functions code, and represents how we use the various Firebase services. It is a generalization and there might be deviation in the actual implementation and code. We tried to only represent the key files, components and services in our application, to give a feel how our interal interactions of our application work. Generally communication of our frontend with Firebase is done with the Firebase JavaScript SDK. For more specific details refer to the specific design and implementation chapters you want more information on.

### 6.2.1 Backend Access

Since the design decision of how we access our Firebase backend services from the frontend influences the whole design, from backend to frontend, it fits in this section most appropriately.

In short, we access most of the backend services from the frontend directly using an SDK, unlike a lot of other projects that expose and abstract them through an API. The main reason is that Firebase is designed to be used in that way. The Firebase JavaScript SDK, which is used in the frontend allows you to access them directly. It could be argued that it still is an API, but an API that is exposed through an SDK instead of a REST API. The SDK uses HTTPS to communicate with the backend, and does this in a secure way that is encrypted.

It also allows us to do small workloads that use small resources directly on the user's system, which will not hinder the user's performance, but could accumulate and shoot up costs if there is a large user base and all these have to be run on our servers.

This decision might also raise some valid security concerns, which will be addressed in more detail in section 6.7. In short, security is not the primary concern of this project, and we do not have actual requirements regarding that. That does not mean we will not consider security, but it is not our focus. For that reason and the reason that we do not collect sensitive data from the user, it is of low concern. Still, Firebase is considered very secure, and unwanted manipulation of data, or injections, are generally checked by the SDK we use. In short, the benefit of accessing the backend through the SDK directly from the frontend, especially for the database, outweighs the valid drawbacks in security that might arise from that decision. For our in-depth security decisions and more on that, refer to 6.7.

#### Authentication

We use the Firebase JavaScript SDK to authenticate the user and use the Firebase authentication service from the frontend. The Firebase JavaScript SDK then manages the token and authentication for us, which we can access and use in our logic.

#### Database

In most backend-frontend projects, the backend directly accesses the database and exposes it through an API, such as a REST API, and the frontend requests this API. We do this completely differently, and the main reason for that is that we use Firebase as our backend. Firebase and it's database Firestore allows us to leverage the advantages of real-time updates. Data can be directly accessed from the frontend using the Firebase JavaScript SDK. It allows us to then subscribe to the data we requested. If implemented correctly, Firestore will now push updates to subscribed data if this data changes, and the UI can then be updated to display the new data.

For us to directly access the data from the frontend looks pretty unorthodox. However, from our research, this is very common, and the suggested way to access the database is to use all features from Firebase to their full extent. Of course, we have exceptions and do not do all the database access directly in the frontend. We also have Firebase Cloud Functions as a Node.js backend, and work that is heavy on performance or security sensitive will be done in the Firebase Cloud Functions. The frontend generally only does basic read and writes, and some small workloads that could accumulate on our backend systems but do not affect the client. For implementation details on how we access the database, refer to 7.2

### Storage

We also use the Firebase JavaScript SDK to access Firebase Storage and allow the user to upload directly to them. This mainly concerns the glsprofile pictures of the user. Of course, we control what a user can upload and what he can read from the storage. Restrictions are listed in 6.4.5.

## 6.3 Domain Model

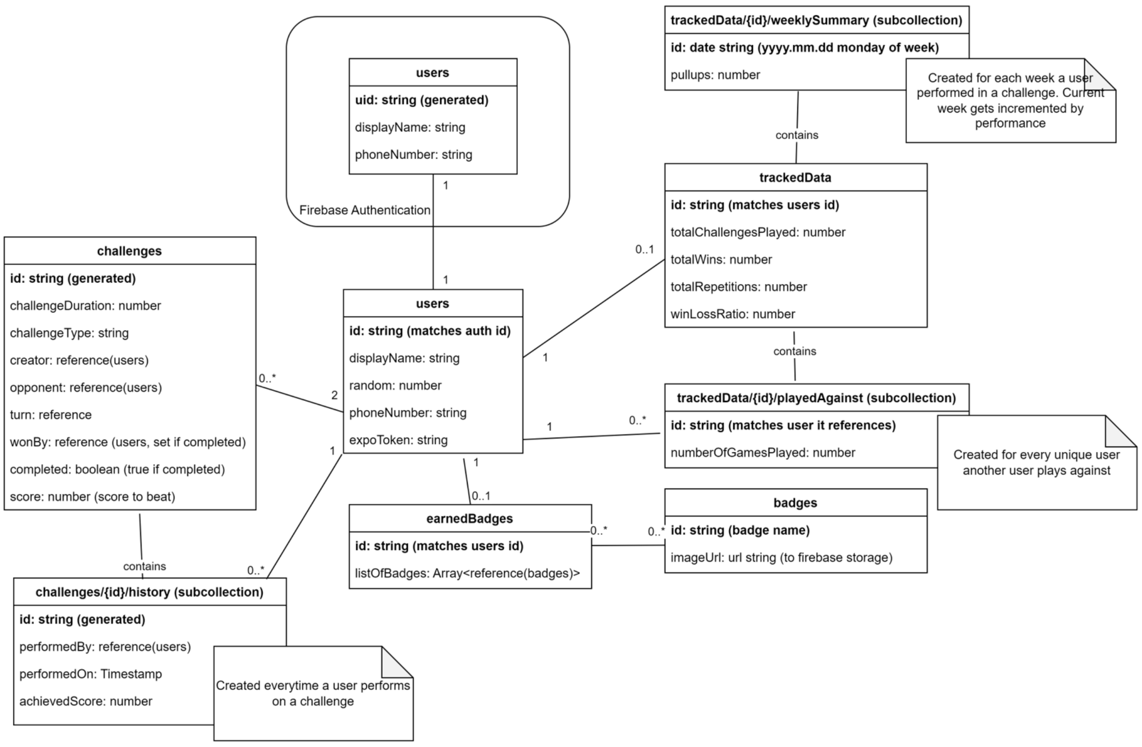

users
uid: string (generated)
displayName: string
phoneNumber: string
Firebase Authentication
1
1
users
id: string (matches auth id)
displayName: string
random: number
phoneNumber: string
expoToken: string
challenges
id: string (generated)
challengeDuration: number
challengeType: string
creator: reference(users)
opponent: reference(users)
turn: reference
wonBy: reference (users, set if completed)
completed: boolean (true if completed)
score: number (score to beat)
0..*
2
1
contains
0..*
challenges/{id}/history (subcollection)
id: string (generated)
performedBy: reference(users)
performedOn: Timestamp
achievedScore: number
Created everytime a user performs on a challenge
trackedData/{id}/weeklySummary (subcollection)
id: date string (yyyy.mm.dd monday of week)
pullups: number
Created for each week a user performed in a challenge. Current week gets incremented by performance
contains
trackedData
id: string (matches users id)
totalChallengesPlayed: number
totalWins: number
totalRepetitions: number
winLossRatio: number
0..1
1
contains
trackedData/{id}/playedAgainst (subcollection)
id: string (matches user it references)
numberOfGamesPlayed: number
Created for every unique user another user plays against
1
0..*
1
0..1
earnedBadges
id: string (matches users id)
listOfBadges: Array<reference(badges)>
0..*
0..*
badges
id: string (badge name)
imageUrl: url string (to firebase storage)

Figure 6.3: Domain Model

The graphic 6.3 serves both as a representation of our domain but also as a database model.

### 6.3.1 Types of References

We use two types to reference another Document from a Document. We will explain these quickly here since to understand the diagram, one has to understand the types of references:

- **Firestore reference type:** Firestore provides a type called Document Reference which can be used to reference another Document. Strictly seen, this type is a string that points to another Document. For example, the "creator" field in "challenges" could have a value like "users/12dafn2123avnneran384". The neat part about that is even though this is a string when loading a Document from Firestore with the Firebase JavaScript SDK, this field will directly be resolved as a DocumentReference. With this type we can directly resolve to the document it points to and read the data from it.
- **Id of Document matches the id of the Document it points to:** We use this type of reference for 1 to 1, or 1 to 0..1 relationship. For example, a user will only ever have at maximum one

trackedData Document. A very efficient way to model this relationship is to use the users' id as the id of the trackedData Document. Because the user will have a maximum of one Document in trackedData, there is no problem with duplicated ids in the same Collection. Also, NoSQL has no problem with duplicated ids as long as they are not in the same Collection, since the "unique key" of a Document is as follows: "/documentName/documentId". The main benefit for us to model such relationships in that way, is performance and cleaner code. Suppose, for example, we want to retrieve the trackedData of a user. In that case, we can directly reference the Document, since we know the Collection name "trackedData" and the id "the users id", and therefore the absolute path of the Document. The need to query a whole Collection to find the Document that belongs to a user is not needed anymore. Therefore, we get more performance, no need for complex queries, and less code.

### 6.3.2 Description

At the center of it all is the **user**. Since we want to save other data for the user and reference the user in other Documents, we have to mirror the user in our Firestore database. The user object in Firebase authentication cannot be extended. Furthermore, it cannot be referenced from inside Firestore. That is the reason we need to mirror it in our database (forum post that lead us to this decision). The Document id will match the uid of the user in the Firebase Authentication Service . The user contains only their phone number, display name, a expoToken used for sending notifications and a random number used for challenges against random users. There is no need for a password since the authentication is done via a one-time password sent by sms to the user's phone.

The most central part of the domain next to the user is the **challenges**. In this Collection, all active and completed Challenges are stored. The Challenge holds all the most essential information, such as the duration each user has to complete the Challenge, which Challenge type it is, the Creator, and the challenged Opponent of the Challenge. As seen in the model 6.3, the association with the user is not the typical association one sees in a domain model. A Challenge must have precisely two users, not more or less. A Challenge also contains a subcollection named **history**, where everytime a user performs on a Challenge, a Document is saved. This subcollection is a history of all performances in a Challenge. The path to this can be seen in the diagram. The benefit of having this saved in a subcollection instead of an array is that not all history is loaded when we load the Challenge. It helps with the performance.

**trackedData** and **earnedBadges** is the persisted data that will be shown in the Profile of a user and their Banner. We have on one side trackedData. This is data like the win/loss ratio, a user's total wins, Etc. The Documents in this Collection can be extended to track more data in the future. The data in this Collection is used to determine if a user earned a badge or to evaluate a user's rankings. **badges** are all the Badges we will make available for the user to earn. It contains the url to the image/icon as a string. In **earnedBadges**, we will track the Badges the users obtained.

**weeklySummary** is a subcollection of trackedData and tracks the performances per week of a user. The id is always the Monday of that week as a date, which is also a field in the document. There the total performances for each challengeType for that week are tracked.

**playedAgainst** is a subcollection of trackedData and tracks all the users a user has played against. The id of the Documents in this collection are the id's of the users a user has played against. These Documents track the number of times a user has played against a specific Opponent.

## 6.4 Backend - Firebase

In the Task formulation 15, it is stated that we use Node.js as our Backend technology. This requirement changed during the project elaboration phase. Our Industry Partner suggested Firebase as a possible backend technology. Since we thought it looked exciting and would match our project well, we used Firebase instead of Node.js.

For the backend we chose Firebase. Firebase allows for a certain abstraction of the backend. It is a Development-Plattform for mobile and web applications, and can be called a "Backend as a Service". It allows developers to manage and create a backend for their application more easily. Their tools cover many services developers normally build themselves, such as analytics, authentication, databases, configuration, file storage, push messaging, etc. These services get abstracted by Firebase and allow the developers to focus on functionality and UI instead of using time and resources to build yet another authentication service. With Firebase developers can "click" these services into existence and then use them without the overhead of writing them themselves.

We chose this for the exact reasons mentioned above. It allows us to focus on functionality and UI and use existing, tested and performative solutions.

This section will cover our Firebase setup, including the database and all the code that runs on Firebase and how it is triggered. It will not cover how we specifically use Firebase in our client applications. This will come in the following section 6.5 and 6.2.1. Below is a screenshot of the Firebase console for our project:

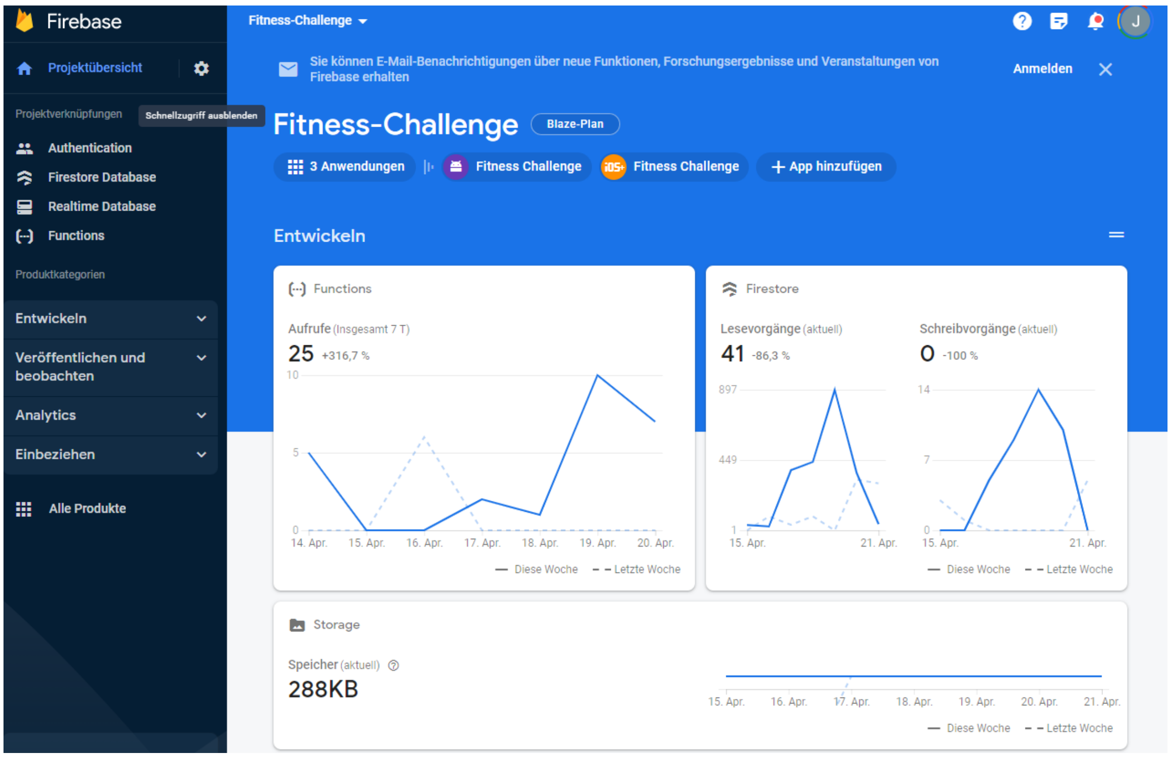


Figure 6.4: Firebase Console

### 6.4.1 Authentication

We used the Firebase Authentication Service for the authentication. We activated the phone number authentication method, so users can log in with their phone number and deactivated the default authentication method of email/password. This authentication method was agreed upon as the only authentication method the users can use. Initially, we also wanted to use Google authentication. However, after some discussion with our Industry Partner, we chose not to implement it since it would make the whole management of the users, choosing an opponent, Etc, that much harder.

We activated specific phone numbers for testing, so no sms must be sent when running the application locally.

Below 6.5 is a screenshot of all possible authentication methods:

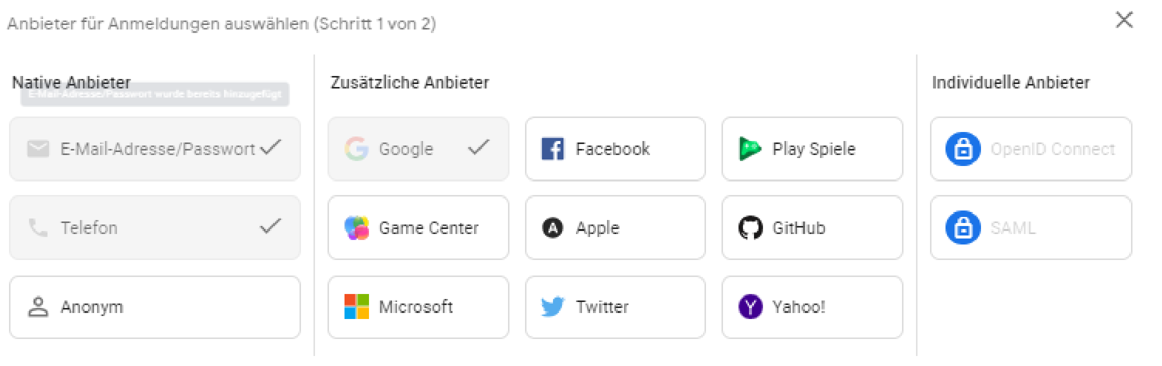


Figure 6.5: Possible Authentication Methods

#### Monitoring

Firebase Authentication allows for easy monitoring of active users and the usage of the phone verification instances. Depending on which authentication methods are enabled, this monitoring screen will look different and track more metrics.

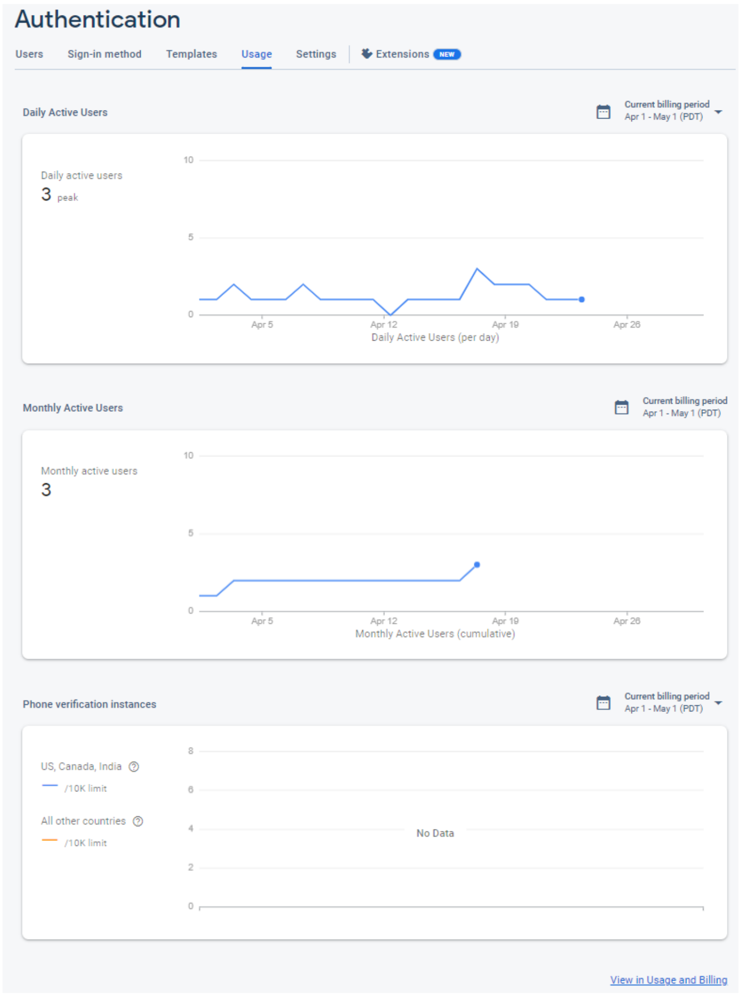


Figure 6.6: Authentication Monitoring

### 6.4.2 Functions

Most of the functionality is implemented in the client app, but some code must be run in the backend. To create custom backend code for Firebase, we use Google Cloud Functions for Firebase. It is a Nodejs backend, but different from typical Nodejs backends; it is not "running" as long as no code is running. Each function created exists on its own and is spun up in its instance when it is triggered and then torn down again when it finishes. That means we have a completely stateless backend. That way, the costs are very low, and no resources are consumed as long as no one uses them. It also automatically scales since each call creates an instance of a function. We use three types of triggers that trigger the functions:

- **HTTPS callable Function**: A function triggered through an HTTP request. It differs from a standard Rest Endpoint that the function can only be triggered through an application that uses our Firebase configuration, which is internal. That means it is not publicly available and cannot easily be triggered from an unknown source. We call these functions through the Firebase JavaScript SDK "firebase/functions" library. It uses the HTTP protocol to send the request.
- **Firestore Triggers**: Firebase listens to changes in specific Collections and Documents we specify and run the function if the event matches the trigger. This is mainly used to send

notifications to the client if something specific is called.

- **Scheduled Triggers**: Gets triggered/run in a predefined interval.

The functions are written with Nodejs and either javascript or typescript. We chose javascript since our frontend is implemented with javascript. We changed the default configuration of Firebase Cloud Functions so that we can use the same javascript syntax that we use in our frontend. We also use almost the same Linting to have the same coding style. This allows for easier development between frontend and backend.

### Structure

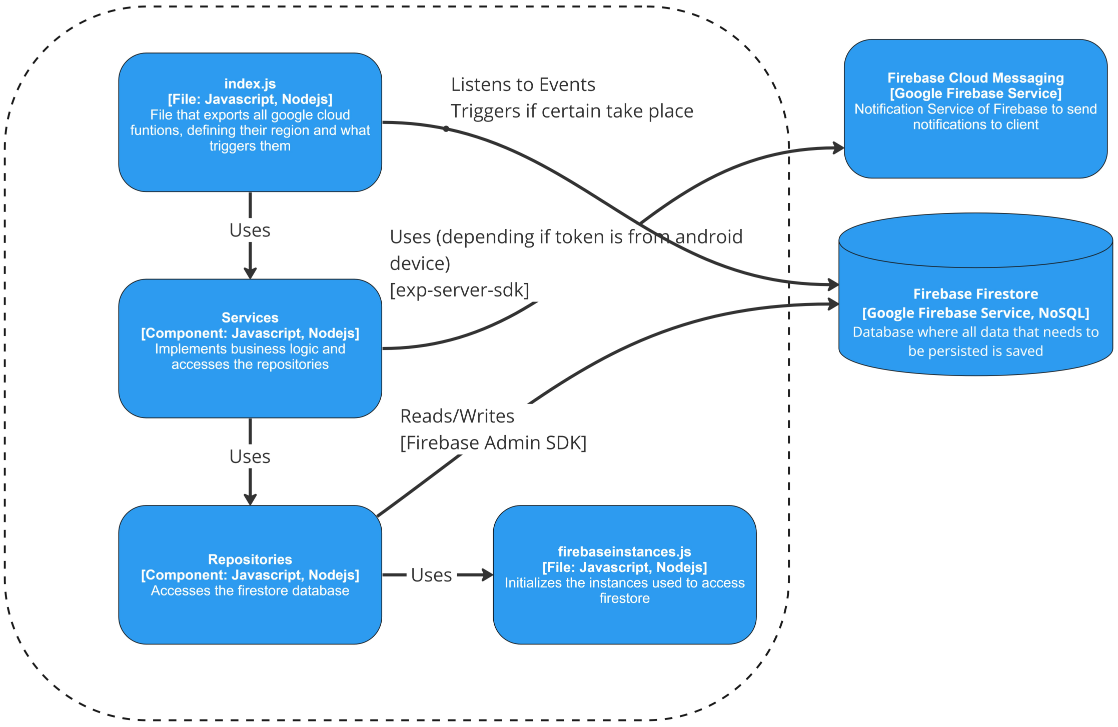

index.js
[File: Javascript, Nodejs]
File that exports all google cloud funtions, defining their region and what triggers them
Listens to Events
Triggers if certain take place
Firebase Cloud Messaging
[Google Firebase Service]
Notification Service of Firebase to send notifications to client
Uses
Uses (depending if token is from android device)
[exp-server-sdk]
Firebase Firestore
[Google Firebase Service, NoSQL]
Database where all data that needs to be persisted is saved
Services
[Component: Javascript, Nodejs]
Implements business logic and accesses the repositories
Reads/Writes
[Firebase Admin SDK]
Uses
Repositories
[Component: Javascript, Nodejs]
Accesses the firestore database
Uses
firebaseinstances.js
[File: Javascript, Nodejs]
Initializes the instances used to access firestore

Figure 6.7: Firebase Functions Structure

The above 6.7 structure is an excerpt from the component diagram 6.2. It represents the structure of our Firebase Cloud Functions code. There is a mandatory index.js, where you must export all your Firebase Cloud Functions. This file calls the services it needs, which do some business logic and call the repositories to read and write data. The file "firebaseinstances.js" correctly initializes the Firebase Admin SDK.

For development details for the Firebase Cloud Functions refer to 7.3.

### Deployment

We chose to deploy the functions currently locally from the CLI using the Firebase CLI Tools. To deploy the functions to the backend, go into the functions directory and run the following command:

```
firebase deploy --only functions
```

### 6.4.3 Firebase Cloud Messaging And Notifications

Firebase Cloud Messaging is another component of the Firebase economy which allow us to send notifications to devices. It is versatile because it allows you to send notifications to single devices, groups of devices, or devices that are subscribed to a specific topic.

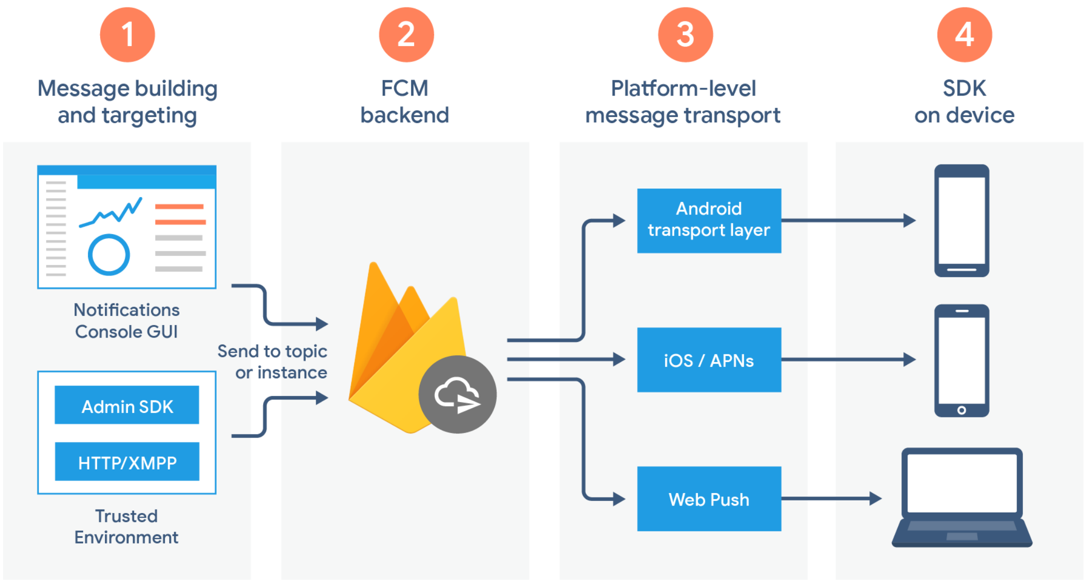

1
Message building and targeting
2
FCM backend
3
Platform-level message transport
4
SDK on device
Notifications Console GUI
Send to topic or instance
Admin SDK
HTTP/XMPP
Trusted Environment
Android transport layer
iOS / APNs
Web Push

Figure 6.8: FCM Architectural Overview

To send notifications, we will implement triggers using the expo-server-sdk. The backend, depending on the destination device, will use either Firebase Cloud Messaging for Android or Apple Push Notification Service for IOS to deliver the notification. We do not differentiate between the platform in our code since we chose to use the Expo abstraction, the expo-server-sdk package. This package does this for us; for details on that, refer to expo-push-notifications.

We chose to send Notifications for the following events:

- User receives Notification if he got challenged.
- User receives Notification when it's his turn to perform in a Challenge.

### 6.4.4 Database - Firestore

The decision for which kind of data persistence we use to persist our data was between two kinds of databases, Firebase provides.

- Firestore
- Realtime Database

Google provides a documentation which helps to choose which database to use. Realtime Database was created before the Firestore; therefore, Firestore is newer and not as much "battle-tested."

Firestore builds on top of the Realtime Database, generally providing everything the Realtime Database can provide and more. In a lot of articles and blog posts, also mentioned in this one, it is recommended for newer applications to use the new Firestore database, since there is no functionality that the real-time database provides that the Firestore cannot. However, there are still a few reasons to choose the real-time database over the Firestore. According to the previous mentioned blog post by Todd Kerpelman [Ker17] the following applies:

- **Detecting presence**: Native support to detect if a user has gone online or offline. Also possible with Firestore but less elegant.
- **Better latency**: Slightly better latency than Firestore, but not by much.
- **Pricing model**: Due to their different pricing model, applications that perform large numbers of reads and writes per second can be significantly cheaper using the Realtime Database.

Any of these reasons above did not apply enough to our application to justify using the older Realtime Database. Detecting presence is also possible with Firestore; slightly higher latency is not a problem since we do not deal with bulk data or require the fastest possible data access, and we do not perform enough reads and writes that would impact the price significantly. The most significant factor that spoke for Firestore was its data model, the NoSQL model. Compared to the Realtime Database where data is written into a single json tree, the NoSQL model with its Collections is much more readable, and we have much more experience with it. This structure allows for better expansion of the application in the future and better maintainability.

Key features of the Firestore are:

- **NoSQL Datamodel**: Easy to read and maintain data structure that is familiar to most developers.
- **Expressive Querying**: Vast filter and sorting possibilities.
- **Real-time Updates**: The client can subscribe to persisted data and gets updated if the data changes in the Firestore database. (Key Feature for our Application)
- **Offline Support**: Caches data that the app actively uses. The app can still be used, and when the device comes back online, the data is synchronized with the Firestore data.
- **Designed to Scale**: Automatically scales are replicated across different regions (depending on the chosen option). It is designed to handle workloads from the world's biggest apps.

These reasons are why we chose and are satisfied with our decision to use Firestore.

Reference 7.2 on how we implemented database access in our application, and 6.3 to see how we modeled our data structure.

### Redundancy and Transaction

We chose to host our Firestore as a multi-regional database. This means it is replicated across multiple regions. We chose the Europe multi-region, which uses a Belgium, Netherlands, and Finland data center. The various regions are listed here. This allows for better availability and durability.

Each operation in Firestore is atomar. That means if it fails, nothing happens. Firestore provides transactions that one can use. We did not use them since no write to the database depends on another write to succeed. Therefore we do not use any transactions in our operations since those operations are transactional and do not depend on others.

**Monitoring**

The usage of the database can be easily monitored through the Firebase console.

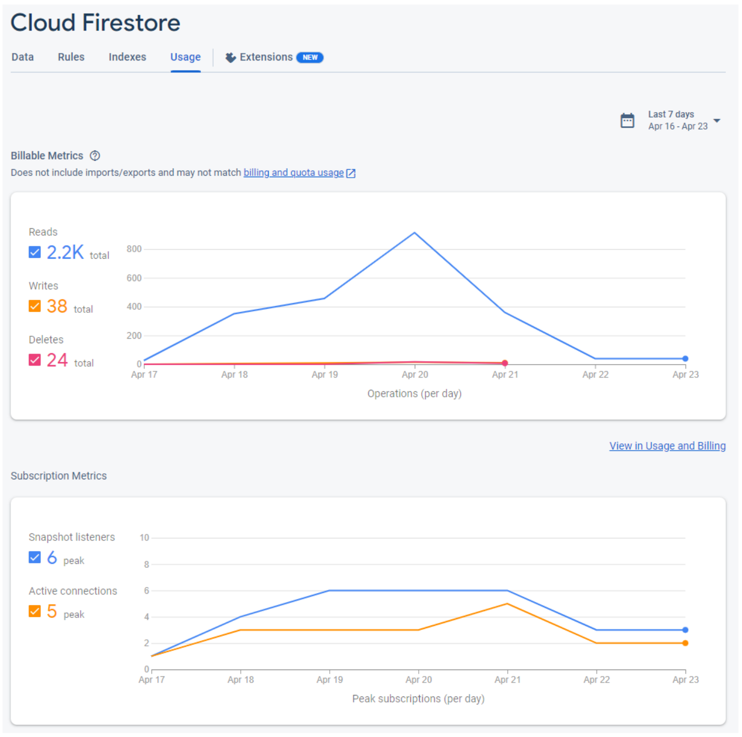


Figure 6.9: Firestore Monitoring

In a specified period, it is possible to examine the number of read, write, and delete operations that have been performed. The lower graph also shows how many subscriptions to data in the Firestore from a client are active. Those subscriptions are there so that the user gets Real-time Updates if the data he subscribed changed.

## 6.4.5 Firebase Storage

We also use the Firebase Storage service to mainly store pictures, such as the Profile pictures the users upload. Firebase Storage can be directly accessed by the Firebase JavaScript SDK, which allows downloading and uploading files to and from the storage. Like in the other services, Firebase Storage configures security rules to strictly control who can read and write from it. There is also already a setup monitoring dashboard to view the reads and writes to storage.

What is stored:

- **badges/{badgeName}:** Images of the Badges a user can earn.
- **profile/{userUid}:** glsprofile picture of the users (if they uploaded one).

## 6.4.6 Scaling

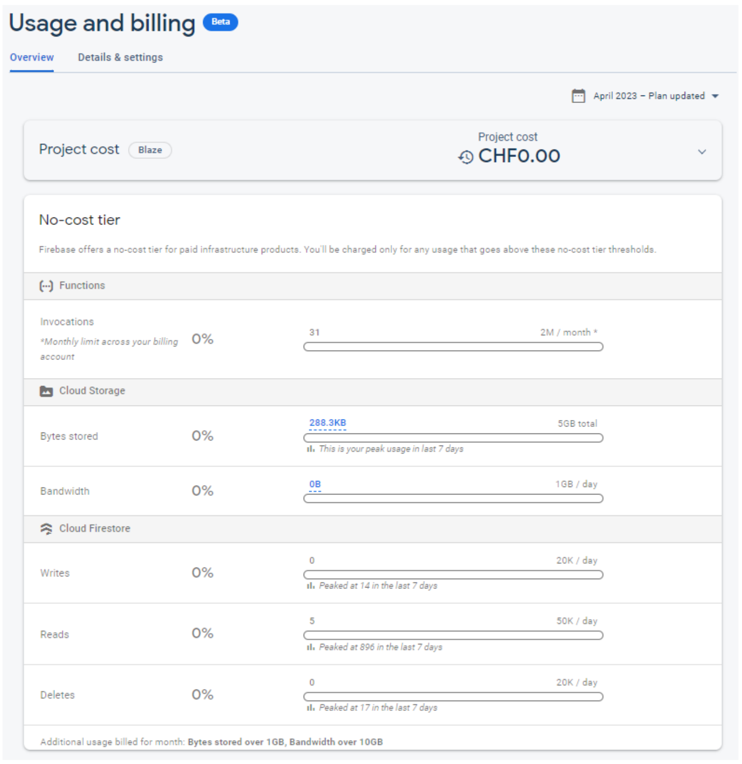
Firebase automatically scales and bills you as you go. On the developers ' end, there is little to do to scale up the application, Firebase does that automatically depending on data, traffic, and user volume.

### 6.4.7 Pricing

We began using the "Spark Plan" when we created the Firebase project. This plan is entirely free, but some features are unavailable, and most features are limited to monthly usage volume.

Since we also need to use the Firebase Cloud Functions to implement certain features such as push notifications, and this feature is not possible to use under the "Spark Plan," we changed it to the "Blaze plan." This plan is generally also free until the usage volume that the "Spark Plan" provides, but instead of blocking further usage, it will continue to work and start billing our account. The specifics of the plan we use and costs that could arise are detailed here.

Detailed billing and cost of the current and past billing period can be viewed in the Firebase console. It details the usage of the different Firebase services, the cost that arose, and how much percentage of the free resources were used up. In the screenshot 6.10 below, in the time around 25.04.23, you can see that we are far from where costs would arise.



Project cost Blaze

Project cost CHF0.00

No-cost tier

Firebase offers a no-cost tier for paid infrastructure products. You'll be charged only for any usage that goes above these no-cost tier thresholds.

| | | | |
|---|---|---|---|
| **Functions** | | | |
| Invocations **Monthly limit across your billing account* | 0% | 31 | 2M / month * |
| **Cloud Storage** | | | |
| Bytes stored | 0% | 288.3KB *This is your peak usage in last 7 days* | 5GB total |
| Bandwidth | 0% | 0B | 1GB / day |
| **Cloud Firestore** | | | |
| Writes | 0% | 0 *Peaked at 14 in the last 7 days* | 20K / day |
| Reads | 0% | 5 *Peaked at 896 in the last 7 days* | 50K / day |
| Deletes | 0% | 0 *Peaked at 17 in the last 7 days* | 20K / day |

Additional usage billed for month: **Bytes stored over 1GB, Bandwidth over 10GB**

Figure 6.10: Usage and Billing for Firebase

Generally for "small apps" (compared to apps like twitter, reddit etc) Firebase is very cost competitive, especially if you consider that the initial setup cost of the backend is much smaller. The maintainability and monitoring of your backend are much easier and come with less setup. For a massive application with an enormous user base, the costs compared to self-hosted or other services are considerably higher, especially since the setup cost is more negligible than the hosting cost. In a small app, the setup and development cost is much higher than the hosting costs.

## 6.5 App Architecture

We use React Native to develop our App. React Native is an open-source software Framework. It is used to develop applications on various platforms such as Android, IOS, Windows, Web, and more. It allows developers to develop apps using the React Framework and native platform capabilities—developers program in JavaScript, which will be converted to native code. The cross-platform and seamless integration on different operating systems and the ability to use the operating-specific APIs and capabilities made React Native a perfect choice. It has a ton of documentation and allows us to easily integrate our Firebase backend to our App. We will focus on two platforms specifically:

- Android
- IOS

That means we will implement a mobile app.

### 6.5.1 Multiplatform - Expo

The need for platform-specific code in our app can be summed up to a couple of lines of code, where differentiating between platforms was absolutely mandatory. That can be explained by our choice to use Expo. Expo is a set of tools and services that can be used to allow developers to develop one codebase that can be used on multiple platforms. We do not directly access platform-specific APIs but use an abstracted API from Expo to access APIs like the file system across multiple platforms using the same codebase.

### 6.5.2 Structure

This section will describe our app's general folder structure and the contents of these different folders.

- **assets:** Contains icons and pictures.
- **components:** Contains our own written Components that are used by the screens.
- **constants:** Contains constants such as translations, navigational strings, configs etc.
- **models:** Models the data we want to use from Firestore, and defines converters, that convert from and to Firestore.
- **repositories:** Access to the backend data. Different from other frontends, we directly access certain backend data 7.2. These files subscribe to specific datasets, which allows the backend to push realtime updates 7.2.1 if these datasets change.
- **routes:** Contains everything needed for navigation across different screens.
- **screens:** Contains files which each represent a different screen in our app.
- **services:** Contains service files that do data manipulation, communication with firbase functions, and other general work and business logic.

For a diagram view on how this structure communicates, refer to 6.2.

Generally all files are considered equal, but there are a few key files worth mentioning:

- app/index.js: This is the root Component of our application. Once the Javascript is loaded, this Component will be activated and the remaining Component tree will be initialized.
- app/App.js: This is the only child of our root Components. It has as its child Component the routes/main/MainNavigator.js. React Native is a Single Page Application, that means App.js will always be displayed to our user, just with different child Components.
- app/routes/main/MainNavigator.js: This Component is the main navigational Component.
- app/firebase.js: This file contains the configuration and initialization of Firebase that allows our app to communicate and connect with our Firebase backend.

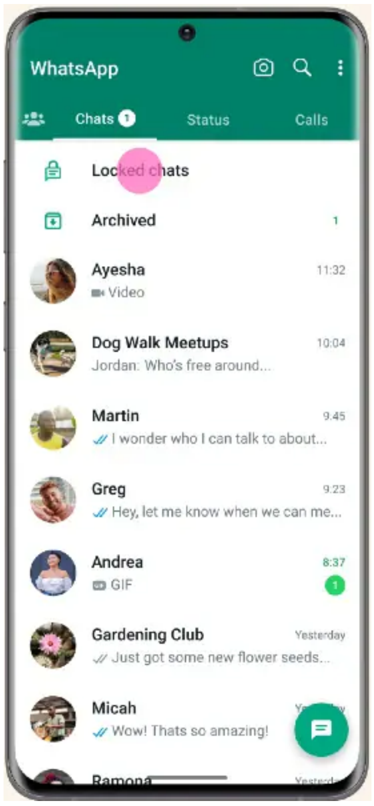
### 6.5.3 UI Design

This section will show the initial draft at the beginning of the project. Designing a UI is an evolutionary process; therefore, the final product may differ from our mockups. We will still include the initial state of the draft at the beginning of the project to give a feeling of our design's evolution over this project. The Design of the UI might change with reviews from our customers and with the beginning of the actual implementation and with the consideration that it's a prototype. It serves as an initial guide for our App.

#### Inspirations

We were inspired and guided by several different applications while designing the product and refining the use cases with our customers. The three applications are:

**Whats App Mobile**



Locked chats
Archived 1
Ayesha 11:32
Video
Dog Walk Meetups 10:04
Jordan: Who's free around...
Martin 9:45
I wonder who I can talk to about...
Greg 9:23
Hey, let me know when we can me...
Andrea 8:37
GIF 1
Gardening Club Yesterday
Just got some new flower seeds...
Micah
Wow! Thats so amazing!
Ramona Yesterday

Figure 6.11: Whats App Inspiration

The widely used messaging app WhatsApp served as a general inspiration and guide on how to structure our app. It influenced the placement of the navigation elements, hover buttons, and the look and feel of the app. We choose this as a guide since everyone with a smartphone uses it, and structuring our app this way allows users to easily use and understand the structure of our app since they are already familiar with it.

**Polytopia**

When designing the Challenge screen, where users see their Challenges, we immediately thought of Polytopia. Polytopia is a turn-based strategy game that can have turn times of up to one week. Since our Challenges are also turn-based, in the sense that one performs a Challenge and then waits, sometimes several hours, for the other users to perform the Challenge, we thought a similar approach to the one Polytopia has would make perfect sense.

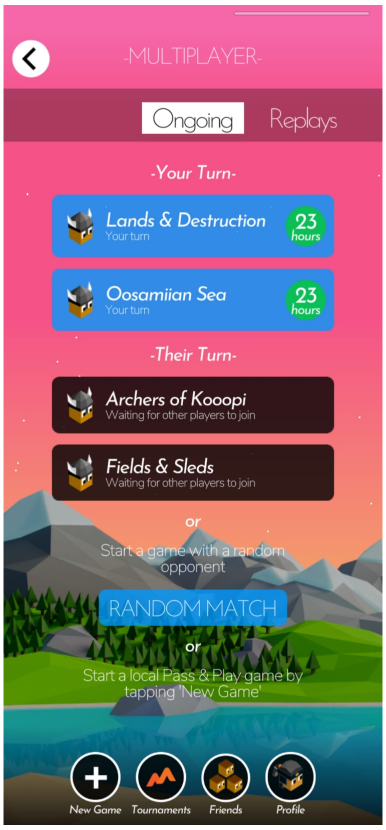


Figure 6.12: Polytopia Inspiration

As one can see in the screenshot 6.12, several games are listed. The screen is divided into two sections. The "Your Turn" section shows the games where the Player is on their turn. The "Their Turn" section shows all the games where the User waits for the Opponent to complete their turn.

**Apex Legends**

In the elaboration phase of the project, where we discussed the specifics of the functionality of our App, we thought implementing some way for the user to "show off" his statistics and a system where a user can earn rewards would make much sense. After all, we are trying to gamify fitness exercises. For this reason we chose to display a Banner of a user that his opponents can see. A game that we know does that excellently is Apex Legends. It is a highly competitive shooter with a high skill ceiling and encourages users to improve. One way it does that is by allowing users to show their achievements with Badges or trackers of specific statistics for other users to see.

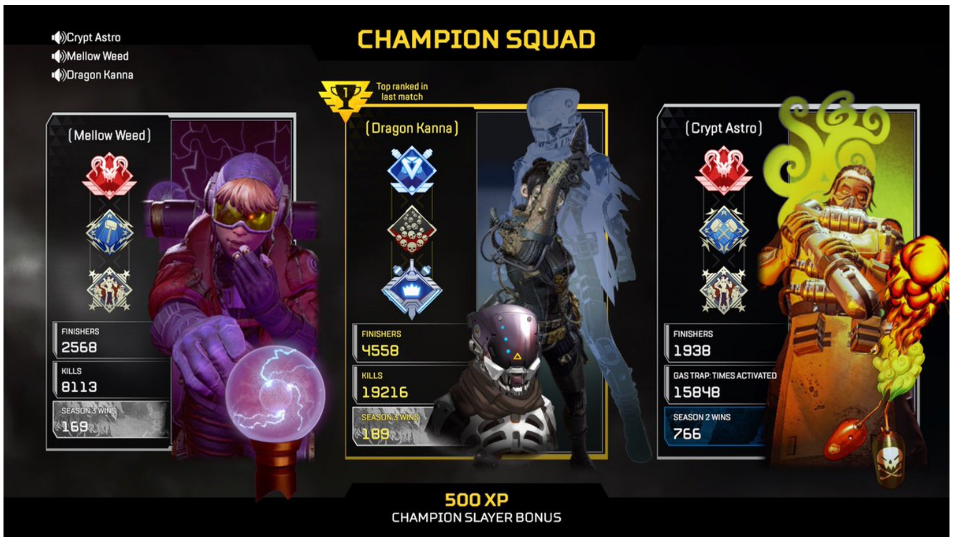


Figure 6.13: Apex Legends Banner

As seen in the screenshot 6.13, players can show off their achievements in their player Banner for other people to see. In the game, the Banner is customizable, but we decided against that feature to focus on the application's core, which is the performing of Challenges.

### UI Mockup

In this section, we will list the initial draft of our UI mockup and, where needed, give some descriptions and elaborations about our reasoning for this initial draft. These mockups where created in the first 2 weeks of the project. As seen, the design was heavily influenced by our inspirations.

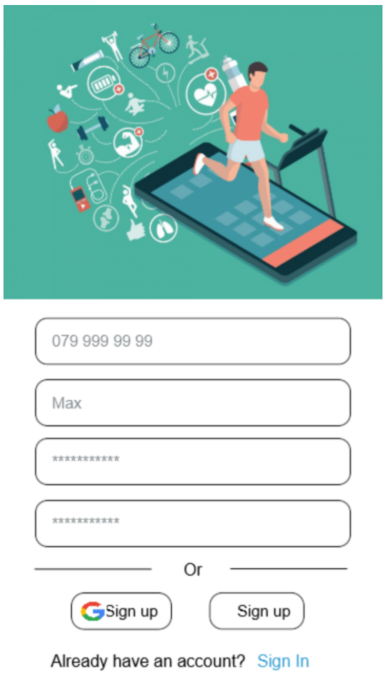


Figure 6.14: Register Screen Mockup

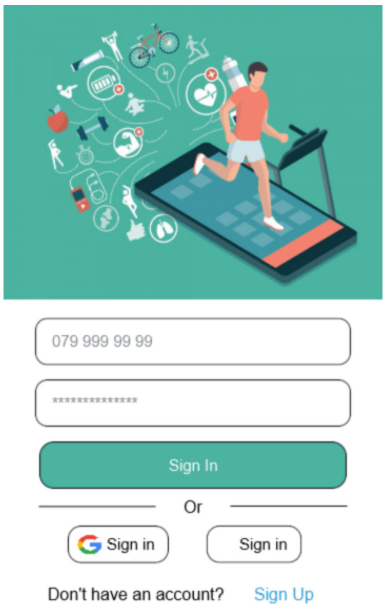


Figure 6.15: Login Screen Mockup

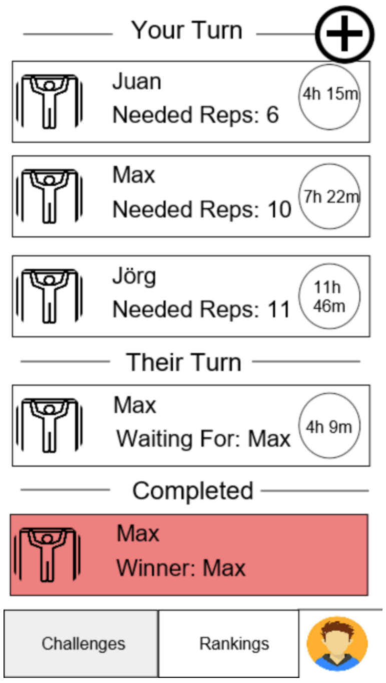


Figure 6.16: Challenge Screen Mockup

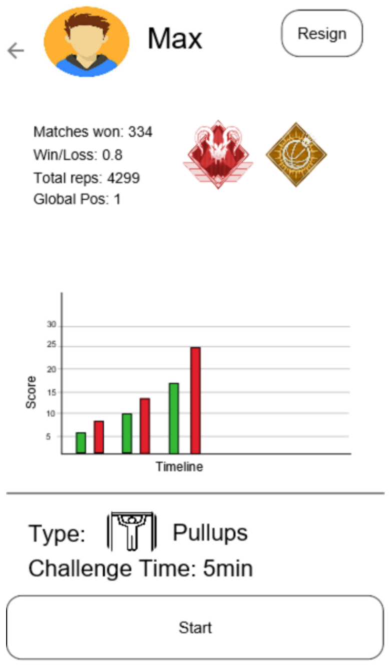


Figure 6.17: Challenge Details Screen Mockup

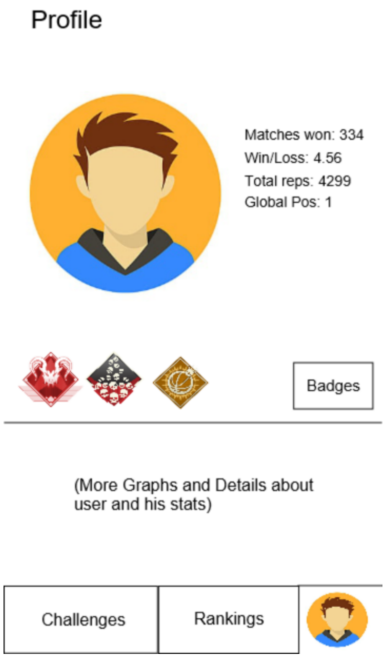
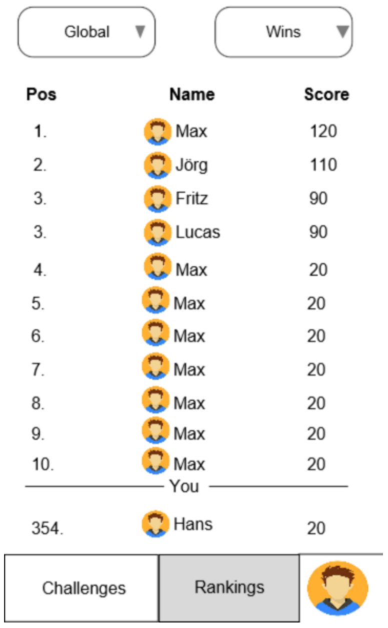


Figure 6.18: Profile/Dashboard Screen with Banner Mockup



| Pos | Name | Score |
|---|---|---|
| 1. | Max | 120 |
| 2. | Jörg | 110 |
| 3. | Fritz | 90 |
| 3. | Lucas | 90 |
| 4. | Max | 20 |
| 5. | Max | 20 |
| 6. | Max | 20 |
| 7. | Max | 20 |
| 8. | Max | 20 |
| 9. | Max | 20 |
| 10. | Max | 20 |
| | You | |
| 354. | Hans | 20 |

Figure 6.19: Rankings Screen Mockup

### 6.5.4 Coding Style Guide

To guarantee that our code has a coherent coding style that makes our code more readable and maintainable, we use an eslint for automatic coding rule enforcement and styling. Since there is already much research on how to style your code and a set of best practices exists and is agreed upon by most developers, it did not make sense to define all our styling guidelines on our own, so we chose two styling packages we use in our linter to help us format the code:

- Plugin react Recommended
- Airbnb code style guidelines

We chose these two since the first is recommended for all react applications, and the second one, Airbnb, is probably the most commonly used pre-prepared set of linter rules. In the link above for Airbnb, any developer developing this application can read up on the styles and rules we use for our coding. We use the default rules enforced by our linter and adhere to the suggestions by Airbnb that are not enforceable by rules. Any deviations in the linting rules can be read in the rules section of our ".eslintrc.json" file in the root of our client application

## 6.6 Deployment

### 6.6.1 Firebase Functions

Deploying our backend code, which is our Firebase Cloud Functions, we can just run "firebase deploy –only functions" or "npm run deployfunctions" in the "firebase" directory of our code. Remember that the Firebase CLI Tools has to be installed, logged in, and connected to the correct Firebase project. A guide on how to do that can be seen here.

The rest of our Firebase services do not need any deployment, and can be directly modified in the Firebase console.

### 6.6.2 Mobile App

One of our requirements was to deploy our application to both the Google Play Store for our Android distribution and also to the IOS App Store. We create two different builds for Android and ios, which are then published in the respective store. The file responsible for building the application is the "app/eas.json" file. We specify two different builds: A preview build, which is only used for internal testing, and a production build, which builds the application that should be published.

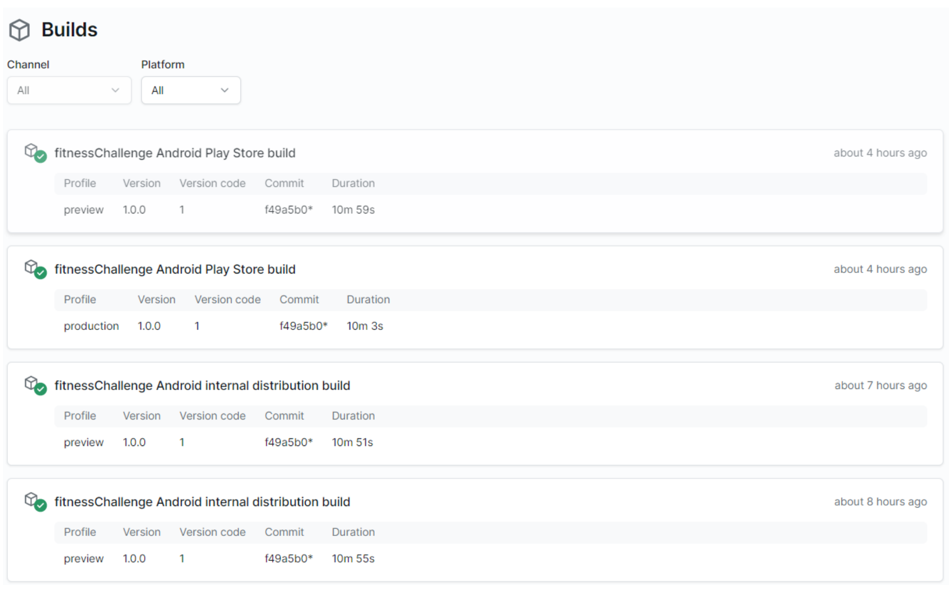
#### Building the App

To build the application, we use the EAS cli. Noteworthy is that the one running the command is logged in to his Expo account and that his account has access to the Expo project.

One can then run the following command in the project root:

```
eas build --platform <all/android/ios> --profile <preview/production>
```

The application is then built in the Expo project. The progress can be tracked there.



Builds

Channel: All
Platform: All

fitnessChallenge Android Play Store build — about 4 hours ago

| Profile | Version | Version code | Commit | Duration |
|---|---|---|---|---|
| preview | 1.0.0 | 1 | f49a5b0* | 10m 59s |

fitnessChallenge Android Play Store build — about 4 hours ago

| Profile | Version | Version code | Commit | Duration |
|---|---|---|---|---|
| production | 1.0.0 | 1 | f49a5b0* | 10m 3s |

fitnessChallenge Android internal distribution build — about 7 hours ago

| Profile | Version | Version code | Commit | Duration |
|---|---|---|---|---|
| preview | 1.0.0 | 1 | f49a5b0* | 10m 51s |

fitnessChallenge Android internal distribution build — about 8 hours ago

| Profile | Version | Version code | Commit | Duration |
|---|---|---|---|---|
| preview | 1.0.0 | 1 | f49a5b0* | 10m 55s |

Figure 6.20: Expo Build

Once the build is done, we can open the build, download the generated file, and install it on our devices for testing.

### Publishing

Both the ios and Google credentials used for publishing to these app stores were provided to us by our Industry Partner. We saved these in our Expo project, which then allows us to deploy the latest production build from our CLI with the following command:

```
eas submit --platform android
eas submit --platform ios
```

Note that to deploy to ios, the field "appleId" in the "app/eas.json" file has to be set to the correct Apple ID. We purposely do not track them anywhere for security reasons.

## 6.7 Security

Security was not the biggest factor in our project, and there were a few reasons for that:

- It's a prototype and not intended to be launched in full production in the duration of the project. The goal as explained by our Industry Partner was to create a prototype which can then be expanded upon.
- The nature of the application does require the biggest security considerations (no financial information, health information). It's no banking or health software where breaches or down time would cause huge damages.
- Firebase is generally very secure in itself.
- As explained in 10.3.1, Joel was unavailable for a considerable amount of time. We discussed this with our Industry Partner, and the decision was to focus on features and not on security.

For those reasons, we chose to not implement the full security measures, that Firebase provides. Still, we made sure that our Firebase Cloud Functions can only be accessed by a authenticated user, and that our database can generally only be accessed by an authenticated user.

For further development, security should of course be addressed more. This page explains very well which kind of security measures should and can be implemented in a Firebase project.

### 6.7.1 Firestore Access

Firestore allows for very detailed rules, on who can access which records, under specific conditions. In this documentation is everything a developer needs to understand on how to write and read Firestore rules.

For development purposes, every record is available to everyone (every registered client). We changed these rules in the progress of our development, to expose only as much data to certain clients to make the app functional, and keep everything else hidden from the clients and inaccessible, to prevent unwanted manipulation.

The rules we chose to implement are (refer to domain model 6.3 to see all collections). :

- Global rule: Every access has to be authenticated.
- challenges: Only allow read or write if user is part of challenge.
- challenges/history: Only allow read or write if user is part of the parent challenge.
- trackedData: Only all reads, but no writes (data is only updated by Firebase Cloud Functions, 7.13)
- trackedData/playedAgainst: Allow all reads, but no writes ("")
- trackedData/weeklySummary: Allow all reads, but no writes ("")
- users: Allow all reads, but only writes from the user it belongs to.
- badges: Allow all reads, no writes (is only changed manually in the console 7.14.2)
- earnedBadges: Allow all reads, no writes (data is only updated by Firebase Cloud Functions, 7.14)
- messages: Allow reads and writes only for user that message belongs to.

The reasoning for those rules will become evident in this paper, when one sees what and when a user should be able to see, or what we write from the client.

Keep in mind that these rules do not apply to our Firebase Cloud Functions backend. The Firebase Admin SDK "ignores" these rules.

### 6.7.2 Storage Access

Similar to the Firestore security rules 6.7.1, we can define rules for storage access to. The rules we defined are:

- Global rule: Allow authenticated read access, but no writes.
- badges/: Allow authenticated reads, but no writes.
- profile/: Allow authenticated reads, but only writes to files that match the users id.

### 6.7.3 Input Validation

As mentioned earlier, we had a big step back when Joel injured himself. Moreover, to still implement all functionalities, we had to make some compromises 10.3.1. One of them was the decision not to implement frontend data validation. The reason for that was that we were running out of time. Of course, this should be addressed and listed in 9.1. However, even without Frontend data validation, Firebase does not allow any "code" input or any input that could create any security risk. So in our opinion this is not a big security risk, and can therefore be ignored in the context of a prototype.

# 7 Implementation

This chapter will describe the most important implementation details and implementation aspects of our project.

## 7.1 Firebase Initialization

The Firebase project was created by our Industry Partner, and then access was provided to us.

### 7.1.1 Setting Up Services

We needed to use several Firebase services for our Application. These can be easily initialized by selecting the services from the Firebase console. This is all we needed to do for Firestore and storage. We recommend setting up the database first since we had to choose our final region, which will be used for all the other services. We will go into a little more detail for the initialization of firebase functions in section 7.3, and authentication in section 7.4.1.

### 7.1.2 Frontend Configuration

The next step was registering an application in our Firebase project. Since we use React Native with Expo 6.5.1 and do not have separate code bases, we have to create a web app in Firebase as required by Expo. When creating a web app, the Firebase dashboard will display a quick how-to to get our app running, which we will expand upon to show how we initialized it in our case.

1. Install the Firebase JavaScript SDK. We chose the latest version, 9.21.0, which is a preview version. We choose a preview version because it contains more functionality, such as more complex queries.
2. Copy the Firebase configuration from the web app we created in Firebase and paste it into a variable in the frontend. This configuration should be recopied if new services are used from Firebase. We configured the whole Firebase connection in the app/firebase.js file:

   ```
   export const firebaseConfig = {
     apiKey: "someKey",
     authDomain: "domain",
     projectId: "projectId",
     storageBucket: "url",
     messagingSenderId: "id",
     appId: "id",
   };
   ```

3. Use this configuration to initialze the app:

   ```
   const firebaseApp = initializeApp(firebaseConfig);
   ```

4. Initialize the Firebase services we use:

   ```
   initializeAuth(firebaseApp, {
     // Keeps users logged in on app close
     persistence: getReactNativePersistence(AsyncStorage),
   });
   export const functions = getFunctions(firebaseApp, "europe-west1");
   export const db = getFirestore(firebaseApp);
   export const storage = getStorage(firebaseApp);
   ```

   We have to specify the region for the firebase functions, since they cannot run multi region like the other services do.

After these configurations, we are able to use Firebase and all the services we specified from it.

### 7.1.3 Firebase SDK

We use the Firebase JS SDK to connect and use our Firebase backend functionality from the frontend. The Firebase JavaScript SDK can be found on npm. We currently use the "9.21.0" version of the Firebase SDK. At the beginning of the project, this was a preview version and was not considered stable. We chose to use the preview version since, before this version; you could not use "or" in query conditions, which made it hard for us to query the data we needed. It is since then the new stable version.

## 7.2 Frontend Firestore Access

As explained in our design decision 6.2.1 we access most of the data directly from the frontend using the Firebase JavaScript SDK API, specifically "firebase/firestore". In this section, we will show how we access the data in our frontend and how we can receive Real-time Updates from Firestore.

### 7.2.1 Read Operations and Real-time Updates

We have two ways to retrieve data from a Document or multiple Documents.

#### Referencing the Document directly

For some read operations, we know the absolute path of the Document we want to retrieve. This is often the case when there is a 1 to 1 or 1 to 0..1 relationship between the "user" Collection and another Collection as explained in 6.3.1. Since we know the Collection name, and the id of the Document (same as the userUid) we want to retrieve, we therefore know the absolute path of the Document. We can then directly reference the Document as seen in the example below, and also add a converter which becomes relevant later 7.2.1 (app/repositories/UserRepository.js):

```
import { doc } from "firebase/firestore";
...
export function getDocReferenceForUid(uid) {
  return doc(db, userCollectionName, uid).withConverter(userConverter);
}
```

```
...
```

The function above returns a DocumentReference using the doc() function. We pass the Firestore instance we initialized 7.1.1, the Collection path (mostly collection name), and the path to the Document (document id). We do not know if the Document exists yet, and have not retrieved the data yet. No request was sent to the backend. To retrieve the data, we can do this (example code):

```
import { getDoc } from "firebase/firestore";
...
export async function retrieveUserData(uid) {
    const userDocRef = getDocReferenceForUid(uid);
    const userDocSnap = await getDoc(userDocRef);
    if (docSnap.exists()) {
        return docSnap.data();
    }
    return undefined
}
...
```

We first generate the reference to the Document we want to retrieve. After that, we use getDoc() to retrieve a DocumentSnapshot from the reference. We now have access to the actual Document that is stored. To check if the Document exists, we use the exists() call. We can just call .data() on the snapshot to read the data. A more in-depth overview to retrieve a Document can be viewed here.

### Querying a Collection

As in all databases, we can also run complex queries to retrieve specific Documents that meet a particular condition. Here is an example of a query that retrieves all the challenges a user is a part of, either as the glscreator or Opponent (app/repositories/ChallengeRepository.js (outdated)):

```
export function getQueryResultForAllChalengesBelongingToUser(uid) {
  const userDocReference = getDocReferenceForUid(uid);
  const q = query(collection(db, challengeCollectionName)
    .withConverter(challengeConverter), or(
    where(creatorField, "==", userDocReference),
    where(opponentField, "==", userDocReference),
  ));
  return q;
  //use getDocs(q) to retrieve a querySnapshot that contains all documents.
}
```

Firestore allows using all the standard query operators and conditions. A list of them and a more in-depth overview of queries can be viewed here (make sure to view for Web Version 9).

### Real-Time Updates

We are able to receive Real-time Updates, on queries or Document references, if we do the following (app/repositories/TrackedDataRepository.js):

```
//For document reference
```

```
export function getTrackedDataForUser(userUid, trackedDataCallback) {
  const docRef = doc(db, trackedDataCollectionName, userUid)
    .withConverter(trackedDataConverter);
  const unsubscribe = onSnapshot(docRef, (docSnap) => {
    if (docSnap.exists()) {
      const newData = docSnap.data();
      trackedDataCallback(newData);
    }
  });
  return () => {
    Log.info("Unsubscribed from tracked data updates");
    unsubscribe();
  };
}
```

As seen above, "onSnapshot" returns a function. This function can then be used to unsubscribe from the updates.

To see how we can subscribe to a whole query, go to "app/services/challenge/ChallengeService.js" function "getChallengesForUserToDisplay".

After we run a query or get the reference for a Document, we can then pass this reference or query to the onSnapshot function, and also pass a callback function. This callback function will run every time the provided Document or query gets updated in Firestore. We usually want our Components to rerender if new data gets pushed. We must run a state update in the callback function (app/components/trackedDataList/TrackedDataList.js) to do that.

```
const [trackedData, setTrackedData] = useState(new TrackedData());

  useEffect(() => {
    const unsubscribe = getTrackedDataForUser(userUID, setTrackedData);
    return unsubscribe;
  }, []);
```

As seen, we provide our update function with a callback that updates the state of our Components. This way, the Components will update with the new data. It is also essential that it is run in a useEffect with no dependencies (seen by the []). Such a useEffect function is called an "onMount" function, which will only get run once when the component is mounted. If we subscribed outside of a useEffect we would subscribe to the data again each time the component rerenders (for example, if a state changes). It is also essential to return the unsubscribe function inside of the useEffect. Everything returned in this useEffect is used to clean up if the component unMounts. So when the component unMounts we want to unsubscribe from any new updates.

### Converters

We can add a converter to a Document reference, which could look like this:

```
export const trackedDataConverter = {
  toFirestore: (trackedData) => ({
    totalWins: trackedData.totalWins,
    totalChallengesPlayed: trackedData.totalChallengesPlayed,
    winLossRatio: trackedData.winLossRatio,
    totalRepetitions: data.totalRepetitions,
  }),
  fromFirestore: (snapshot, options) => {
    // Todo: Try to use return of promise.then() (see functions index.js)
    const data = snapshot.data(options);
    return new TrackedData(
      data.totalWins ?? 0,
      data.totalChallengesPlayed ?? 0,
      data.winLossRatio ?? 0,
      data.totalRepetitions ?? 0,
    );
  },
};
```

This allows us to use a format we can control in our code and add logic to the conversion, like reformatting timestamps, default values if a document does not have all values set, and more.

## 7.3 Firebase Functions

### Setup

There is a good guide on how to setup the firebase functions, which we followed. When running "firebase init function" to initialize the functions, the cli will prompt you to choose some options. An important option we selected was JavaScript as our functions language. We chose JavaScript over TypeScript since using the same language in the backend as we use in the frontend makes sense. After the initialization of the functions, we have a Node.js project, which can be used as a normal Node.js project. It will run differently internally. We listed these differences in 6.4.2. We changed two significant things after the initialization that differed from the default functions setup because we wanted the syntax of the functions project to be as similar to the frontend as possible. That way, there is no need to code in two different "dialects" of javascript. Those are:

- We added the line "type: "module"" in the package.json file. The reason for that is being able to use export/import instead of module.exports/require("x").
- We added the airbnb eslint package to guide our coding style, since we use that in the frontend too.
- Copied other eslint rules from the frontend to the functions project.

### Implementation

For a overview of the structure we implemented for the Firebase Cloud Functions refer to 6.7.

This section will cover high-level implementation details for our Firebase functions. We have several different functions that are being used. Each function has to be defined in the same way in the index.js file at the root of our functions project. Here are two examples of functions (firebase/functions/index.js); the first is an httpOnCall function (HTTP Callable Function 6.4.2) , the second is a Firestore trigger 6.4.2:

```
import functions from "firebase-functions";
...
export const completeChallenge = functions
  .region("europe-west1")//
  .https
  .onCall((data, context) => {
    ...
  }
...
export const newPerformanceAddedToHistory = functions
  .region("europe-west1")
  .firestore
  .document("challenges/{challengeId}/history/{historyId}")
  .onCreate(async (snap, context) => {
    ...
  });
```

We have to use the functions library, seen imported above, to initialize a function. We can can define a region where this function should run. The default is "us-central1". Unfortunately, multi-region support, as we have in Firestore is yet to be a functionality. So, for now, we have the requirement that our functions run on "europe-west1". After defining the region, we can specify the type of trigger and the event the function is triggered. Functions defined with a Firestore trigger are background functions and are not actively called, but passively through an event in our Firestore database. The HTTP onCall functions, on the other hand, are called from our application in the following way, where we have to pass the instance of our firebase functions which we initialized in the firebase file as seen in 7.1.2, and pass the exact name of the function as a string. (app/services/cloudfunctions/cloudFunctionsService.js):

```
import { httpsCallable } from "firebase/functions";
...
export default function runJobAfterChallengeCompletion(challengeUid) {
  const challengeCompletionCall = httpsCallable(functions, "completeChallenge");
  challengeCompletionCall({ challengeUid }).then(() => {
    ...
  }).catch((err) => {
    ...
  });
}
```

This is different from a regular http REST API that we do not provide a path but the exact name of the function we want to call. The requested path is generated by the Firebase JavaScript SDK through the function name and the configuration we initialized the SDK with 7.1.1. To call this function, the application must be able to initialize Firebase with our internal configuration. So this function is not publically available and at less risk for DDos attacks than a standard REST API might be.

Very important to keep in mind is that all promises must be completed as soon as the Firebase function returns. In regular Node.js backends that stay up and running, this is not a problem, and work can be done even if the request to the api has already returned. This is not the case with Firebase functions. If a firebase functions returns/completes, the instance is torn down. Each promise or async function that is still running will be abruptly terminated. This can lead to incomplete tasks and confusing errors.

### 7.3.1 Implemented Functions

The following list 7.1 represents all the implemented functions in Firebase, their triggers and what their implementations does.

| Function | Trigger | Description |
|---|---|---|
| completeChallenge | Manually by the frontend when a challenge is completed | Updates relevant statistics for both parties and checks for any badges that got earned |
| newPerformance AddedToHistory | Automatically when a new history document is created | Updates the total repetitions of the user that performed and also his total repetitions of the current week. Sends a push notification to the opponent to inform him that it is their turn now. |
| checkForAccount WithAssociated Number | Gets called manually by the authentication process | Will check for account associated with the phone number in our user database |
| sendChallenge Notification | Automatically when a challenge document is created | Sends the opponent a push notification to inform them about the challenge |
| scheduled ExpirationChecker | Automatically every 10 minutes | Checks for any expired challenges in the database and updates them accordingly |

Table 7.1: Implemented Cloud Functions

### 7.3.2 Firebase Admin SDK

We do not use the same Firebase SDK as we use for the frontend development. In the Firebase Cloud Functions, we use the Firebase Admin SDK. Since the functions are entirely under our control and run on "our" servers, it is best to use this SDK. With this SDK, we can read and write to any Collection storage path and "ignore" the security rules set on the users. It is the recommended SDK by Google to be used in the Firebase Cloud Functions.

## 7.4 Authentication

This section will explore how we setup and implemented authentication and user management using Firebase Authentication.

### 7.4.1 Setup

Before we can use authentication in our application, we have to set up the Firebase Authentication service 6.4.1. After setting up the service, we enabled authentication by phone number using a one-time password verification sent by sms. To enable easier testing, we also configured some testing numbers with predefined one-time passwords that we can enter without needing an sms. We then initialized the authentication in our frontend as explained in 7.1.2. In the code segment there, a line is visible where we use getReactNativePersistence. Adding this line allows us to keep the user logged in even if he closes the app.

These testing numbers should be removed when the application goes into production.

### 7.4.2 Implementation

To explain the authentication implementation, we will go through an example and register a new user. Logging in as an existing user will use the same one-time password functionality from Firebase; therefore, we will only focus on registering.

A user who is not logged in will see the Login Screen (app/screens/authentication/LoginScreen.js). The user can then navigate to the Register Screen 7.1 (app/screens/authentication/Register-Screen.js) using the link.

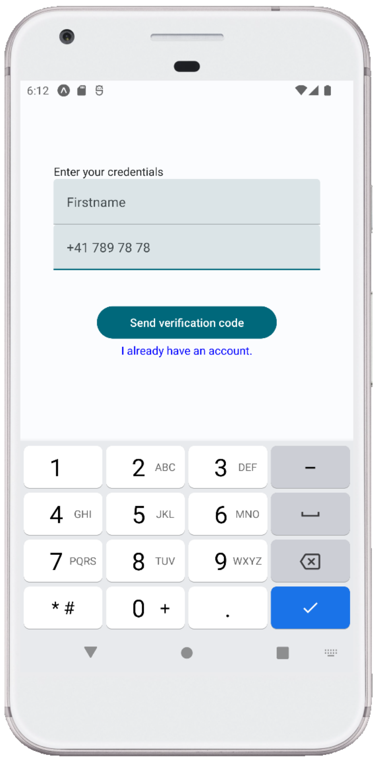


Figure 7.1: Register Phone Authentication

The user then has to enter his first name and a valid phone number. When he presses the button below "Send verification code", the following code is run:

```
...
(app/screens/authentication/RegisterScreen.js)
//expo-firebase-recaptcha
<FirebaseRecaptchaVerifierModal
    ref={recaptchaVerifier}
    firebaseConfig={firebaseConfig}
/>
...

(app/services/auth/phoneAuthService.js)
...
export function sendPhoneAuthVerificationCode(
  phoneNumber,
  recaptchaVerifier,
  verificationIDCallback,
) {
  const phoneProvider = new PhoneAuthProvider(getAuth());

  return phoneProvider.verifyPhoneNumber(
    phoneNumber,
    recaptchaVerifier.current,
  ).then((verificationI) => {
    verificationIDCallback(verificationI);
  }).catch((err) => {
    Log.error("Was not able to verify phone number", err);
  });
}
...
```

The entered phone number is passed, and a reference to a reCAPTCHA (using Expos reCAPTCHA). reCAPTCHA has to be used to avoid spam attacks to our backend. This reference is then used and further passed in the function call "verifyPhoneNumber (api reference not up to date)" of the PhoneAuthProvider. It will then open the reCAPTCHA which will most of the time prompt the user to check a checkbox and then solve a puzzle. After the user completes the reCAPTCHA, "verify-PhoneNumber" will return a verificationId, and the Firebase backend will then send a sms to the provided phone number. The user can then enter the sms in the following textfield and confirm his input.

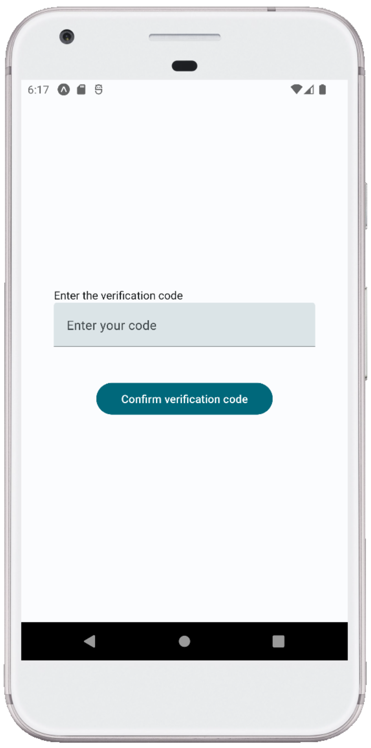


Figure 7.2: Register OTP Challenge

When the user inputs his one time password, the following code section will get run:

```
export async function phoneAuthVerifyVerificationCodeForRegister(
  verificationId,
  verificationCode,
  name,
) {
  const credential = PhoneAuthProvider.credential(
    verificationId,
    verificationCode,
  );
  await signInWithCredential(getAuth(), credential).then((user) => {
    initializeUser(user.user, name);
  }).catch((err) => {
    Log.error("Was not able to log in user", err);
  });
}
```

We pass the "verificationId" from before. This id has to be passed with the "verificationCode" when generating the credentials for this login. That way, even if someone knows the phone number and can read the one-time password, he cannot use it on any device other than the one where the OTP challenge was initialized. Finally, we sign in the user with the generated credentials, and if the user is successfully logged in, we will initialize the user's default Profile configuration. We set the displayName of the user as the name he provided and set a default Profile picture for the user.

Furthermore, as seen in the domain model 6.3, we mirror the user in our Firestore database. So when the user registers, we will create a default Document for the user in our Firestore "users" Collection.

### 7.4.3 Firebase Phone Auth Limitations

Something that we overlooked while doing our research about phone authentication using Firebase was that there is no separate process/code flow for registering and Login. The same functions and logic have to be used in both cases. Since we do not want to allow a user to use Login for registering or vice versa (because they have to choose a display name), we had to develop a workaround.

We wrote a Firebase Cloud Functions 7.3.1 that gets called if a user tries to log in or register. The function returns a true or false, depending on if the account the user tries to log in or register with exists. If, for example, the account already exists, we do not allow the user to use our registration form, and vice versa.

Here is an example for registering:

```
...
checkForAccountWithAssociatedPhoneNumber(phoneNumber).then((result) => {
                if (!result.data.exists) {
                  sendPhoneAuthVerificationCode(
                    phoneNumber,
                    recaptchaVerifier,
                    setVerificationID,
                  );
                } else {
                  setDoesAccountExist(true);
                }
              });
...
```

### Development Hurdles

This issue took us longer to resolve than expected since firebase authentication for React Native and Expo was not documented well, and we had problems finding an existing example. Therefore, we overshot our estimation quite a bit, as stated in 10.3.1.

## 7.5 Challenge Creation

A user can create a Challenge from the home screen by pressing the Floating Action Button with the "+" symbol. A button of this kind is used in various applications to create a new post in a forum, a new chat in a chatting app, and more. Also, WhatsApp uses a similar Floating Action Button, and since we use it as inspiration, we chose to implement a similar button. The Floating Action Button can be seen in 7.7. Once the user presses the button, a new screen opens with a form the user has to input:

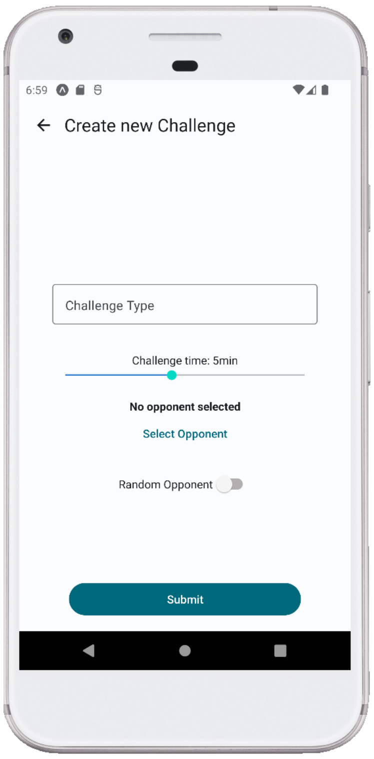


Figure 7.3: Create Challenge Form

The user can then input three different values. He can also switch the slider to create a challenge against a random user, which will be covered in 7.5.2. The user can choose the challenge type from a dropdown (currently only one value), select the time each competitor has to perform in the Challenge (1min-10min), and select an Opponent, which will be discussed in the following chapter 7.5.1. The user is required to select a Challenge type and choose an Opponent; the default value of the Challenge duration is already selected (5 min). If the user has not entered the needed values, a red text will be displayed below the input field he needs to enter a value.

Once the user creates a Challenge, the Challenge will be saved into the "challenges" Firestore Collection. It gets saved with the following values:

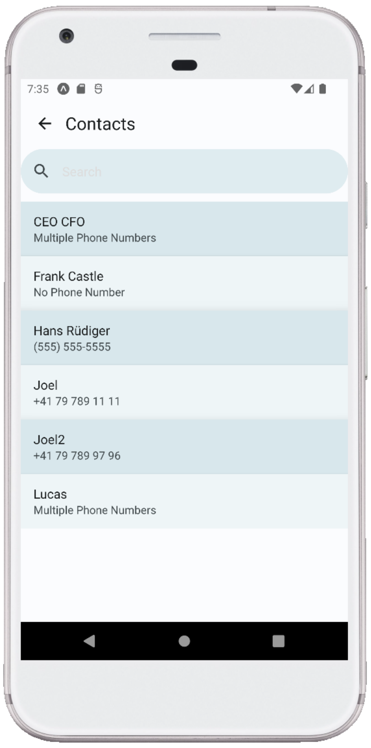
| **Field Name** | **Saved Value** |
|---|---|
| challengeDuration | The duration selected from the slider (number) |
| challengeType | The type of the Challenge selected from the Dropdown (string) |
| completed | Marks if Challenge is completed, initialized with false (boolean) |
| Creator | The user that filled out the form and created the challenge (Document Reference) |
| Opponent | The Opponent the user selected, or a random opponent 7.5.2 (Document Reference) |
| score | The current high score of the Challenge, initialized with 0 (number) |
| turn | Which user has to perform next on the Challenge. Initialized with the opponent field value (Document Reference) |
| lastPerformed | The date when the last performance on challenge happened. Is initialized with the server timestamp of Firestore |
| turn | Which user has to perform next on the Challenge. Initialized with the opponent field value (Document Reference) |

Table 7.2: Challenge Document after Creation

### 7.5.1 Opponent Selection

Since we use phone authentication 6.4.1, we do not have to manage a friends list but can directly access the user's contacts and use that as a friends list. When creating a Challenge the user has to select an Opponent, which opens a screen where all his contacts from his phone are displayed. The user can then search the list for the contact name he wants to choose. There are three different cases we had to consider.

- **Contact has only one phone number saved**: Is displayed directly.
- **Contact has no phone number**: We display a text that says Contact has no number.
- **Contact has multiple numbers saved**: We display that the Contact has multiple phone numbers.


7:35

Contacts

Search

CEO CFO
Multiple Phone Numbers

Frank Castle
No Phone Number

Hans Rüdiger
(555) 555-5555

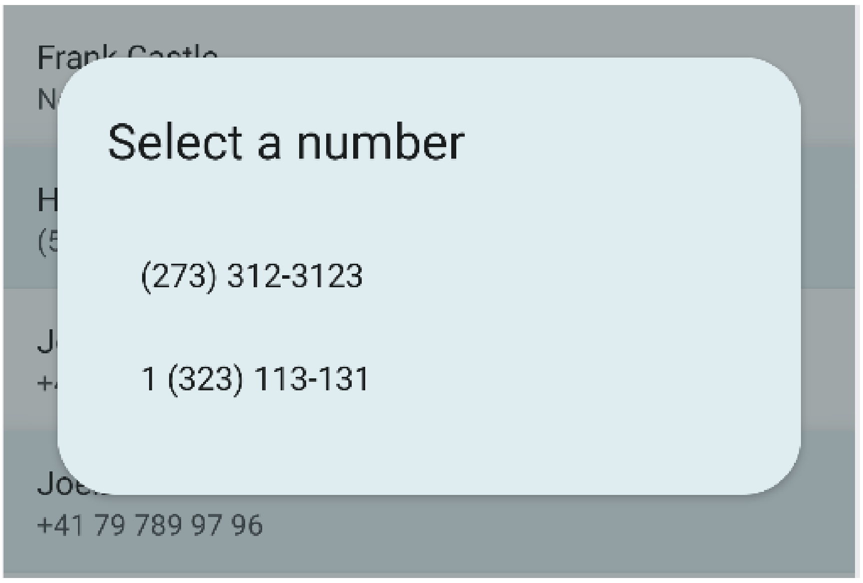
Joel
+41 79 789 11 11

Joel2
+41 79 789 97 96

Lucas
Multiple Phone Numbers

Figure 7.4: Opponent Selection Screen

If the user selects a contact with only one number, this number is used to search the Opponent in our system. If the selected contact has multiple numbers, the following dialog 7.5 opens up, where the user has to decide which phone number he wants to Challenge:


Select a number

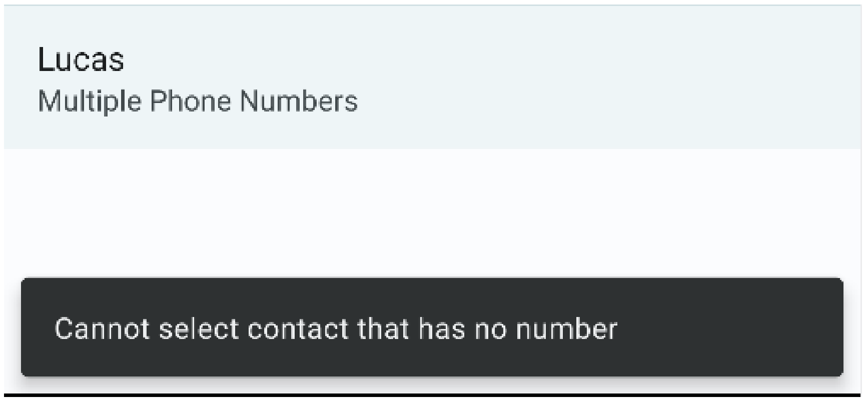
(273) 312-3123

1 (323) 113-131

+41 79 789 97 96

Figure 7.5: Choose Number Dialog

A user cannot choose a contact with no number saved. If he tries to select one, we display a hint, as a snackbar 7.6 at the bottom of the screen, that he has to choose a contact with a number:



Lucas
Multiple Phone Numbers

Cannot select contact that has no number

Figure 7.6: Contact not allowed Snackbar

### 7.5.2 Challenging random user

We considered multiple approaches to implement the functional requirement to challenge a random user from our database. The first approach was to retrieve the whole user collection and calculate its size. We used this size to generate a random number between 1 and said size and pick the user that way. We quickly realized that this approach would significantly impact performance once our user collection grows, so we deviated to the approach of saving a randomly generated number for each user between 1 and 100’000’000. On our backend, we will also generate a random number between 1 and 100’000’000 and retrieve the user document closest to that generated number. We also covered the edge case of challenging oneself in this approach, which could happen very frequently if the user collection is small.

## 7.6 Home Screen

The Home Screen 7.7 is the screen displayed to the user once he logged in. The home screen displays all the Challenges of the user.

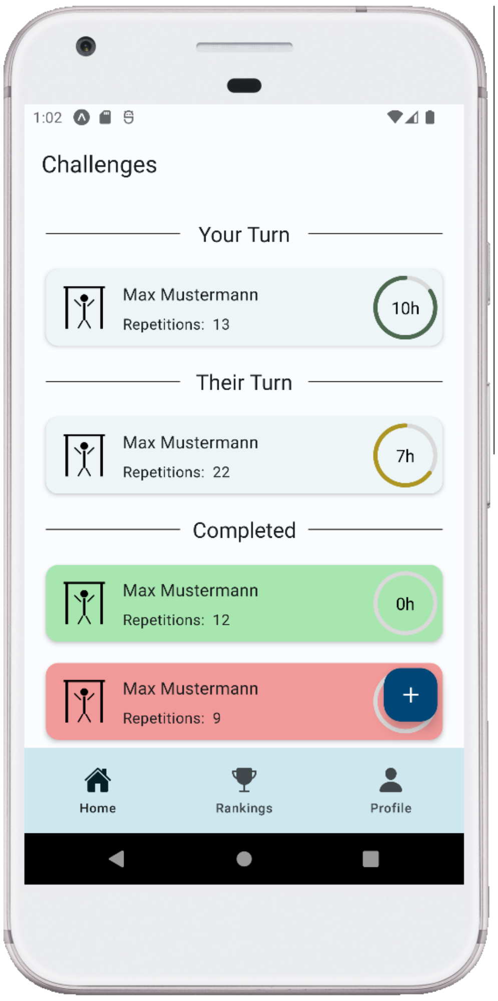


Figure 7.7: Home Screen

It is divided into three different sections, which was inspired by the game "Polytopia" 6.12:

- **Your turn (Du bist dran):** Under this section it displays all the Challenges which are the current users turn. He is able to perform 7.8 on these Challenges.
- **Their turn (Spielt):** These are all the Challenges which the current user waits for his Opponent to perform his turn.
- **Completed (Beendet):** All the Challenges that are already completed. Marked green if the user won, red if the user lost.

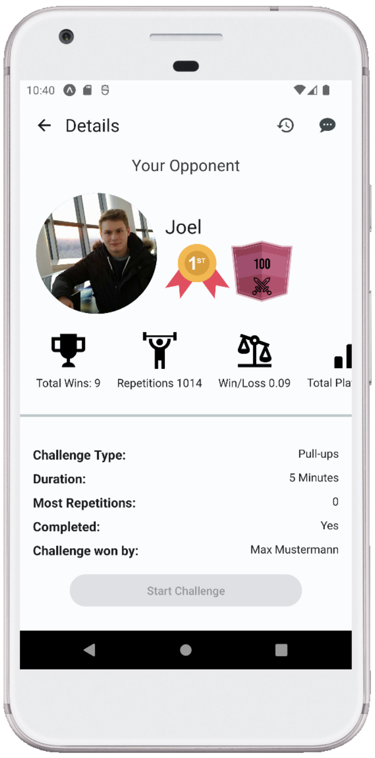
The user can press on each of these Challenges to view the Challenge details 7.7 and also the history 7.7.1.

On the bottom right it also displays a Floating Action Button, which is the "main action" of our application, and allows a user to create a new Challenge 7.5.

The challenges are displayed in so called Card components from our react-native-paper library. We modified the component so it displays relevant information about the challenge and also made it pressable. To display the challenges, we first fetch all the challenges related to the current user and sort them by their completion status and turn status. Afterwards we map those challenges each into their own card component.

## 7.7 Challenge Details

When a user clicks a challenge card from the home screen 7.7, they will be directed to our challenge details screen 7.8 where we present important information about the challenge.



| | |
|---|---|
| **Challenge Type:** | Pull-ups |
| **Duration:** | 5 Minutes |
| **Most Repetitions:** | 0 |
| **Completed:** | Yes |
| **Challenge won by:** | Max Mustermann |

Figure 7.8: Challenge Details Development

The screen is divided into two sections.

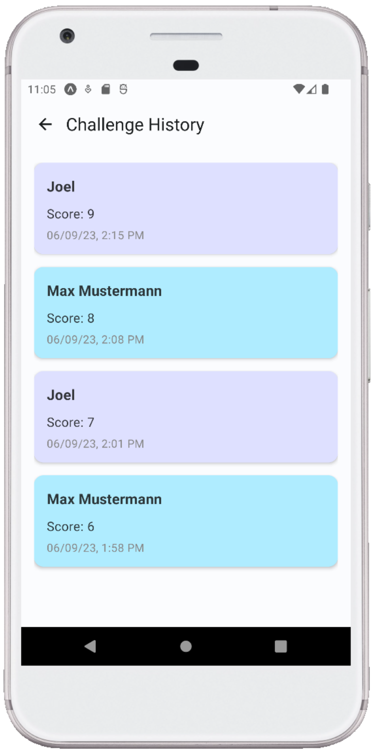
The upper half of the screen showcases the opponent's banner (refer to section 7.9.3).

The lower half of the screen contains a collection of details about the challenge, including the challenge type, duration, current score, whether the challenge has been completed or not, and the winner of the challenge.

From this screen, the user has several options. They can navigate to the chat screen 7.10, the challenge history screen 7.7.1, or directly initiate the challenge 7.8.

### 7.7.1 Challenge History

When the user chooses to access the history screen from the challenge details screen 7.8, they will be presented with a simplified view of cards.


11:05

Challenge History

Joel
Score: 9
06/09/23, 2:15 PM

Max Mustermann
Score: 8
06/09/23, 2:08 PM

Joel
Score: 7
06/09/23, 2:01 PM

Max Mustermann
Score: 6
06/09/23, 1:58 PM

Figure 7.9: Challenge History Screen

Each card represents a user's performance on a challenge (details on that 7.8). The color scheme of the cards will vary depending on the performer.

It is important to note that the history screen is limited to the current challenge, meaning only the performances related to the ongoing challenge will be displayed here.

## 7.8 Perform a challenge

The challenge will commence immediately when the user initiates a challenge by pressing the corresponding button on the challenge details screen (refer to section 7.7).

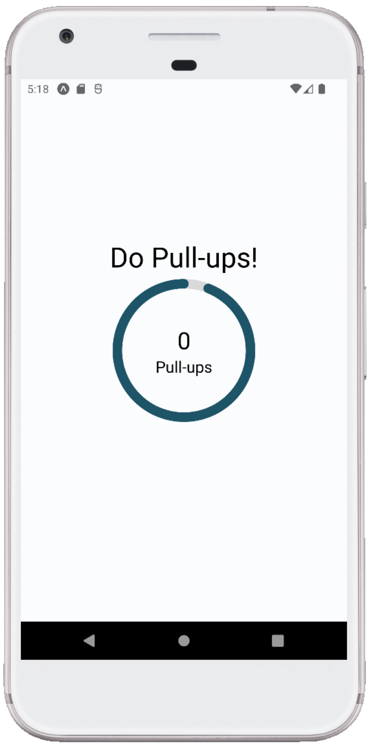


Figure 7.10: Challenge Perform Screen

A large circular countdown timer will appear in the center of the screen, indicating the remaining time for the challenge. Concurrently, the currently detected repetitions will be displayed at the center of the countdown timer.

Our requirement was 5.1 to implement one challenge type. We chose to implement a "pull-ups" Challenge. We use the acceleration of the y-axis to detect repetitions and, for that, the user's score. As with all platform-specific API access, we used Expo 6.5.1, specifically expo-sensors. We access the "Accelorometer" sensor to detect the pull-ups. As with everything else, we considered modularity and extendability highly in this requirement. The code that implements the screen 7.10, is the file "app/screens/challenge/challengePerform/ChallengePerform.js". If the screen opens it initializes the challenge, as follows:

```
...
    useEffect(() => {
        const unsubscribe = initialize(challenge.challengeType, setScore);
        return () => unsubscribe();
    }, []);
...
```

It calls the function "initialize" from the file "app/services/challenge/PerformChallengeService.js" with the requested challenge type and a callback to set the score (state update). Even though we prevent the user from going back (he has to wait until the timer runs down), we make sure to "unsubscribe" from challenge detection if the screen unmounts for some reason (more on umounting in 7.2.1).

```
...
    export function initialize(challengeType, setScoreCallback) {
      if (challengeType === "pullups") {
        return initializePullUpPerform(setScoreCallback);
      }
      return null;
    }
...
```

First, we check which challenge type is requested, and depending on Challenge, we initialize the detection differently. For pull-ups, we do the following initialization:

```
...
    function initializePullUpPerform(setScoreCallback) {
      const accelerationThreshold = 1.5;
      const handleAccelerometerDataForPullUp = ({ x, y, z }) => {
        const accelerationMagnitudeY = Math.abs(y);

        if (accelerationMagnitudeY >= accelerationThreshold) {
          setScoreCallback((prevCount) => prevCount + 1);
          Vibration.vibrate();
          Log.debug("Pull-up detected with acceleration in y: ",
            accelerationMagnitudeY);
          Log.debug("Current accelerations in x, y, z: ", x, y, z);
        }
      };

      Accelerometer.addListener(handleAccelerometerDataForPullUp);
      Accelerometer.setUpdateInterval(100);
      return () => {
        Accelerometer.removeAllListeners();
      };
    }
...
```

In this function, we provide a listener to listen to the Acceloremeter and set a updateInterval, on how many times we check if a pull-up was performed. We also return a correct tear-down function that removes all the listeners from the sensor.

Once the countdown completes, the following is run:

```
...
const handleCountdownComplete = () => {
    tearDownChallenge(challenge.challengeType);
    updateChallengeWithPerformance(challenge.key, score);
    setChallengePerformanceCompleted(true);
};
...
```

It tears down the repetitions detection and updates the Challenge with the performed score, and adds the performance to the history 7.7.1) (which will trigger newPerformanceAddedToHistory 7.3.1). A push notification will be sent to his Opponent, who tells him that he has performed on the challenge and it is now his turn to perform 7.12. While updating the challenge ("app/repositories/ChallengeRepository" -> "updateChallengeWithPerformance") two cases can apply:

- **Challenge is lost:** The challenge is considered lost if the user cannot beat his opponent's score. The challenge Document is updated accordingly and marked as won and completed.

Furthermore, we also run the "completeChallenge" Firebase Cloud Functions task (what it does list in 7.3.1)

- **User was able to beat the score:** If this occurs, the challenge will be sent back to the other user in the challenge, which now has to perform on the challenge and try to beat the achieved score.

### 7.8.1 Implement new Challenge

We mentioned earlier that modularity was also highly considered while implementing the "perform on a challenge" requirement. This section will explain in great detail how new challenges could be implemented.

First, add a new challenge type to the Dropdown in challenge creation 7.3. Note down the value you used as the new challenge type.

Generally, mirror all the translations as we did for the pull-ups. Do that in the "translations.js" file 7.15.

Now navigate to the "app/screens/challenge/challengePerform/ChallengePerform.js" file. Make sure you add a new set of strings as we did for the pullups in the variable "challengeStrings".

Now it is time to add performing logic. Navigate to "app/services/challenge/PerformChallengeService.js." Implement a function called "initialiaze<NewChallengeType>." Ensure it has the same parameter signature as "initializePullUpPerform" and returns a teardown function. In this function, implement your challenge to perform logic (for example, repetition detection).

Write a teardown function "tearDown<NewChallengeType>Challenge" in the same file. Make sure you tear down the perform logic correctly.

Modify the "initialize" function as follows:

```
export function initialize(challengeType, setScoreCallback) {
  if (challengeType === "pullups") {
    return initializePullUpPerform(setScoreCallback);
  }
  //Your new challenge perform logic
  if (challengeType === <NewChallengeType>) {
    return initialize<NewChallengeType>(setScoreCallback);
  }
  return null;
}
```

Also modify the "tearDownChallenge" as follows:

```
export function tearDownChallenge(challengeType) {
  if (challengeType === "pullups") {
    tearDownPullUpChallenge();
  }
  //Your new challenge tear down logic
  if(challengeType === <NewChallengeType>) {
    tearDown<NewChallengeType>Challenge();
  }
}
```

Everything should be functional, and users can perform your new challenge type.

**Optionally**: If you want the 10-week history 7.9.1 to display your new challenge type, you would have to refactor it as explained in 7.9.1.

### 7.8.2 User did not Perform on Challenge

Each challenge remains active for 12 hours until it expires. If the user currently required to perform the challenge does not do so within these 12 hours, the challenge will automatically be marked as complete, and the opponent will be declared the winner.

We have implemented a scheduled cloud function in our Firebase backend to address any expired challenges. This function is triggered every 10 minutes (can be lowered for production), check for any challenges that have reached their expiration time, and completes those. For a high-level overview what this function generally does, refer to 7.3.1.

### 7.8.3 Shortcomings

While we can detect repetitions, "hardcoding" detection like this is not the optimal way to detect repetitions. A repetition may not be detected, or for one repetition, two are detected. Unfortunately, there is no way around it with how we currently implement it. The best way to improve it would be machine learning. However, since this is a prototype and adding machine learning would completely explode our scope, we decided not to do it. However, we recommend considering it for further development 9.1.

## 7.9 Profile

Each user has their own Profile, which can be accessed through the main navigation at the bottom of the screen. The final Profile looks like this:

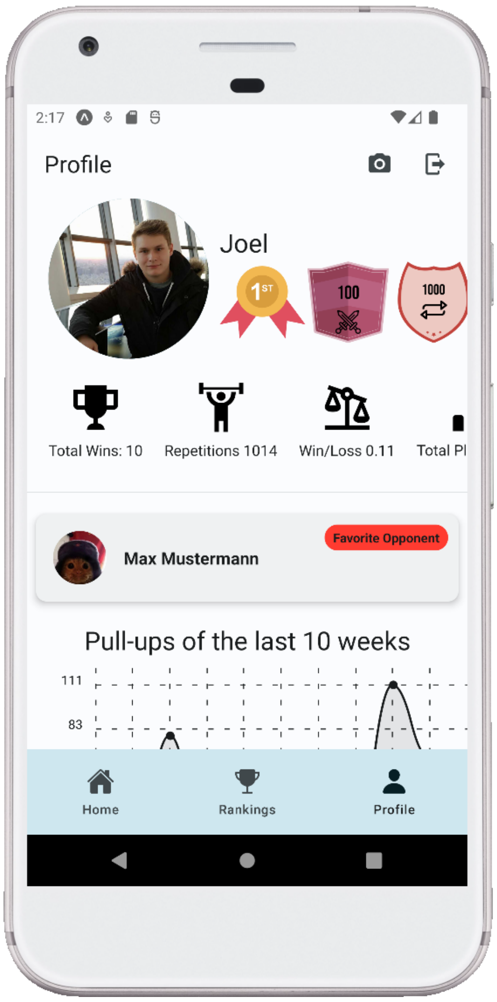


Figure 7.11: Profile Screen

The user sees his most important information in this Profile. It is divided into two sections. The section above the divider is called the Banner, which we will expand upon in 7.9.3. The section below the divider is only visible to the user it belongs to. It displays two things, the Opponent the user played the most against and the pull-ups per week for the last ten weeks. How the data is tracked can be viewed in chapter Tracked Data 7.13. If more Challenge types are implemented in the future, the graph could easily be extended with a dropdown to change the Challenge type it displays. This section of the Profile is vertically scrollable since only some content can be displayed simultaneously.

The Profile contains two interactive actions, both of which are located in the top right. The user can log out, which will clear his session, cancel all Real-time Updates and return him to the login screen. The second is changing the profile picture which we will expand more on in 7.9.2.

### 7.9.1 10 Week History

How we track this data can be viewed in 7.13. The complete graph looks as follows:

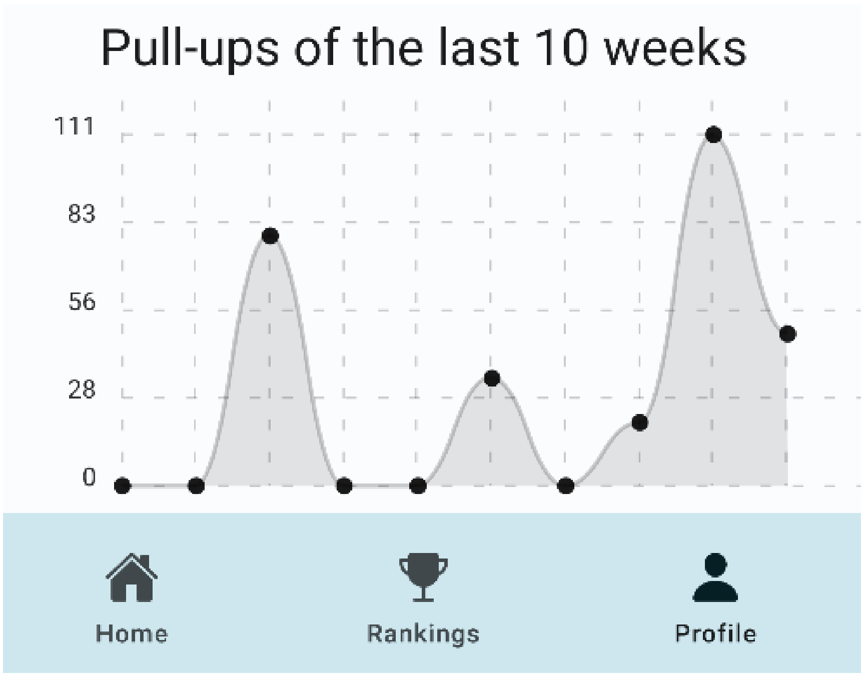


Figure 7.12: 10 Week history graph

The graph fetches data for the last ten weeks from Firestore for the user. This data can have gaps or miss entirely. So before we display the graph, we interpolate data for each week we want to display where no data is available. We interpolate data with 0 pull-ups. We also subscribe to Real-time Updates of the last ten weeks for the user. If a new weekly summary is pushed, for example, a new week begins, and the graph will update accordingly. The code regarding this logic can be viewed in "app/repositories/TrackedDataRepository.js -> getTotalRepetitionsByWeek".

#### Extend with new Challenge Type

This is probably the only component we did not write fully extendible. At the moment, it is only able to show the pull-ups. One only would have to refactor the "app/repositories/Tracked-DataRepository.js -> getTotalRepetitionsByWeek" function.

The reason for that is that time was running out, and we had to focus on more critical aspects of the project.

### 7.9.2 Profile Picture

We display a Profile picture in the Banner 7.9.3 of the user, which he can change. When the user first creates his account, the user is initialized with some values 7.4.1, one of them a default glsprofile picture. For more information about the initialization, refer to 6.4.1—the default Profile picture points to a picture inside the application itself. The user can change his glsprofile picture, which is displayed in the Banner 7.9.3. We chose to use the expo-image-picker to allow the user to choose a new picture.

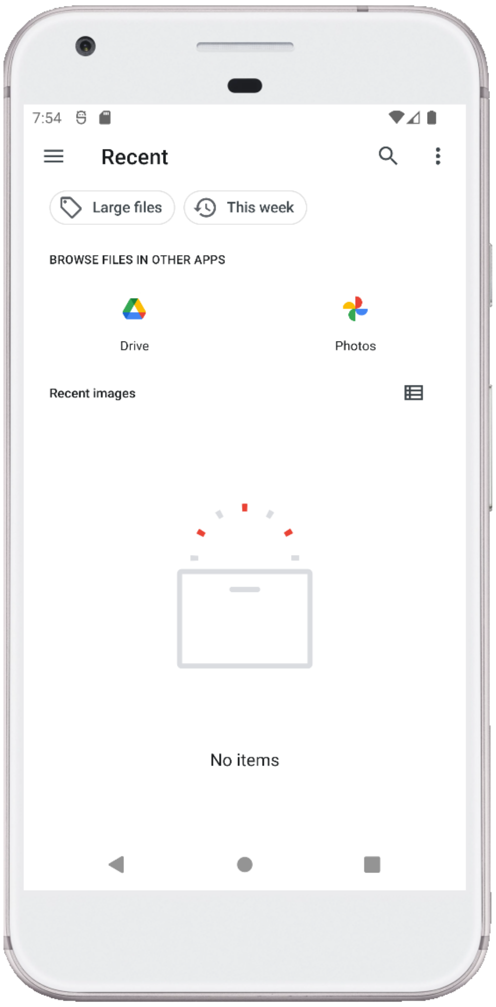


Figure 7.13: Profile Screen

Since it is a Expo package, it is Multiplatform 6.5.1 and works for both our targeted platforms. It allows the user to select an image from almost all familiar sources. It also works with other apps, such as an additional file manager, galleries, onedrive, google drive, and much more apps the user has installed.

Everything to do with selecting a new Profile picture is coded in the "app/services/profilePicture-Service.js." We allow the user to crop his picture to the size he wants. He can only choose a single picture, but we still check if more than one picture is selected and do not proceed with the process. The selected image is then converted to a Blob. We upload this Blob to Firebase Storage 6.4.5 with the UID of the user as the file name. After it is uploaded, we retrieve the "download url," a URL pointing to the picture in our storage. Those images are publicly readable. We save that URL in the Firestore 6.4.4 Document of the user that just uploaded the picture. The field in the Document is called "photoURL". Since we subscribe to the users Document when displaying the Banner 7.9.3, every user that currently is viewing the Profile picture of that user, including the user that changed it, Real-time Updates 7.2.1 are pushed, and the image gets updated in realtime. If the image cannot be fetched for some reason, we fall back to the default Profile picture.

### 7.9.3 Banner

The Banner 7.14 is the upper portion of the user's Profile.

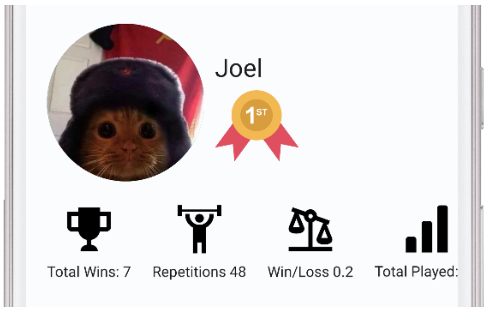


Figure 7.14: Banner

This part of the Profile is not just visible in the Profile of the user but also in the Challenge details 7.7. When a user opens the Challenge details, he sees the banner of his Opponent. So the banner is to show off to other users and motivate the user to get the best results possible. It makes the whole App more competitive since other users can see the statistics. The inspiration to implement a Banner was "Apex Legends," which we discussed in 6.13. The banner displays four kinds of information:

- **Profile Picture:** The image the user selected 7.9.2
- **Display Name:** The display name the user entered while registering 6.4.1
- **Badge List:** The milestones a user has earned. Details how these are calculated in chapter 7.14. It is horizontally scrollable if the user has many Badges.
- **Tracked Data List:** Some metrics we track for the user. Details how those are tracked are calculated in chapter 7.13. Is horizontally scrollable if the Tracked Data can not be fully displayed or if more would be tracked in the future.

### 7.9.4 Component Structure

Here we will show the rough structure of the Profile screen, and how it is split into it's own Components. "app/screens/main/profile/ProfileScreen.js" structure:

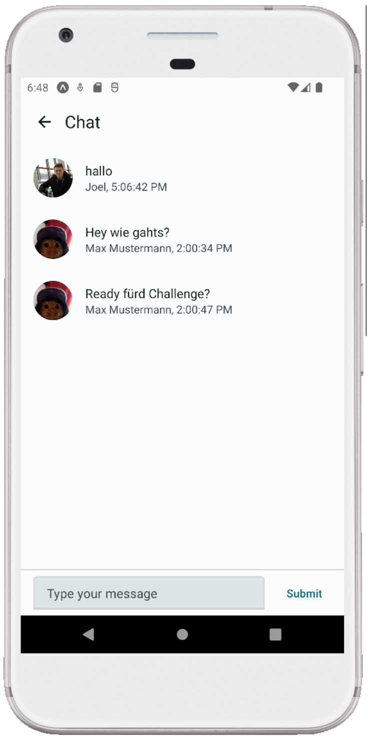
- **Banner** (app/components/banner/Banner.js)
  - **Horizontal Badges List** (app/components/badges/HorizontalBadgeList.js)
  - **Tracked Data List** (app/components/trackedDataList/TrackedDataList.js)
- **Favorite Opponent** (app/components/favoriteOpponent/FavoriteOpponent.js)
- **History Line Chart** (app/components/historyLineChart/HistoryLineChart.js)

## 7.10 Chat

When users navigate to the chat screen (through the challenge details screen 7.8), they encounter a straightforward chat application. The screen comprises an input field for entering messages and a chat history section. The messages are stored in our Firestore database containing the message, receiver, sender and a timestamp. Depending on the user and their opponent, we will filter out the corresponding message and sort them by their timestamp.

As long as the screen does not get unmounted, the user is subscribed to the messages and will get real-time updates on the chat screen.

We also included the profile picture, a formatted timestamp and the display name at each message to identify who sent the message and at what time as depicted in the figure 7.15.


6:48

Chat

hallo
Joel, 5:06:42 PM

Hey wie gahts?
Max Mustermann, 2:00:34 PM

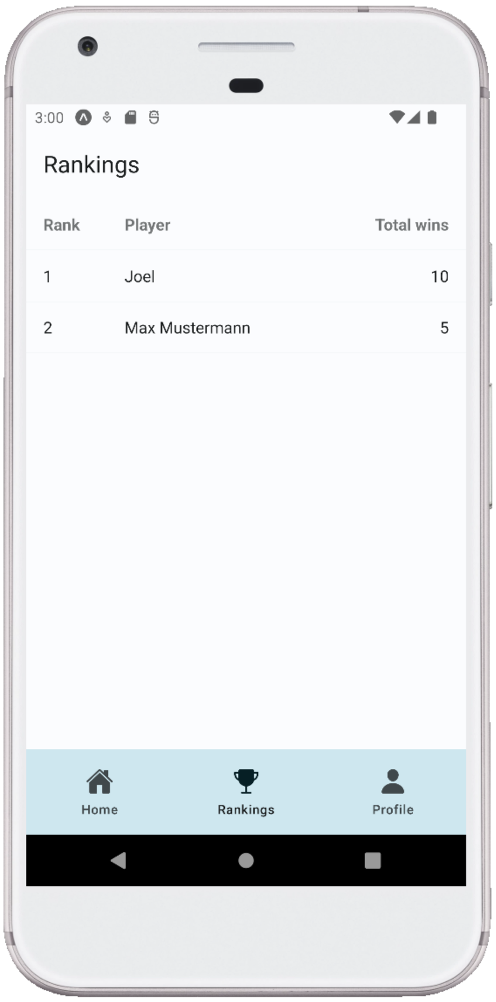
Ready fürd Challenge?
Max Mustermann, 2:00:47 PM

Figure 7.15: Chat Screen

Unlike the challenge history screen, the chat history will persist across multiple challenges. However, only the latest 100 messages will be displayed in the chat history for performance reasons. This can be addressed in the future.

## 7.11 Rankings

The rankings screen presents the top 100 players who have achieved the highest number of total wins. The screen features a ranking table with columns for Rank, Player, and total wins. The function that fetches the rankings is "app/repositories/TrackedDataRepository.js -> getTrackedDataOrderedByTotalWins".


3:00

Rankings

| Rank | Player | Total wins |
|---|---|---|
| 1 | Joel | 10 |
| 2 | Max Mustermann | 5 |

Figure 7.16: Rankings Screen

The table is sorted based on the total wins (tracked by 7.13), allowing us to calculate the rank for each user. Any changes in the ranking table are immediately reflected in real time as we have subscribed to the database for updates. This ensures that the rankings screen dynamically updates whenever there are changes to the players' win counts.

Due to time limitations partly driven by a realized risk 10.3.1 we were not able to implement a friends ranking list as we do not have a friends list feature.

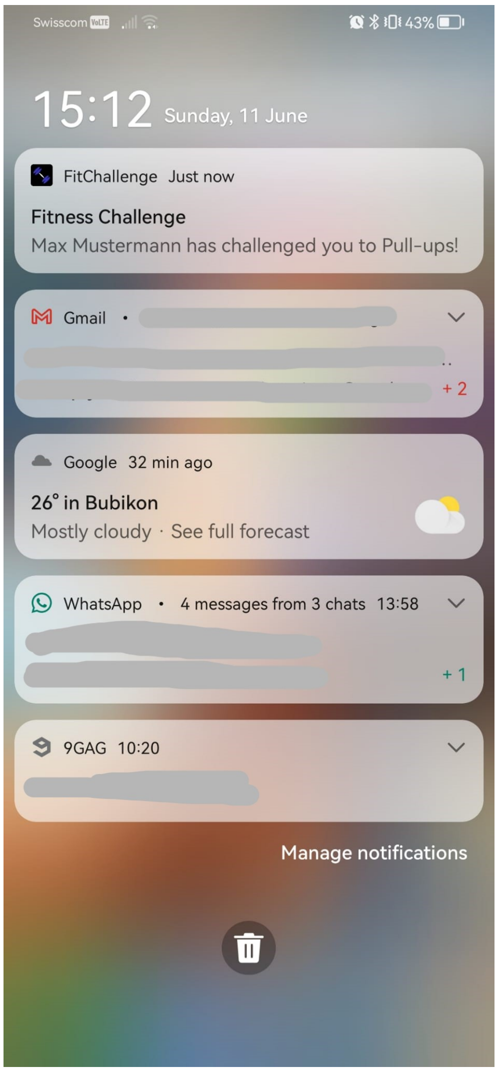
### 7.11.1 Extending the Rankings

Should the rankings be extended to display a ranking based on different statistics, one could add something like a dropdown to allow the user to select what ranking he wants to use. This value should then be used to retrieve a different set of data in the following code section of the RankingsScreen (app/screens/main/ranking/RankingsScreen.js):

```
useEffect(() => {
    //Add rankings selection logic here
    const unsubscribe = getTrackedDataOrderedByTotalWins(setPlayers);
    Log.info("Updated rankings");
    return () => {
            Log.info("Unsubscribed from tracked data");
            unsubscribe();
        };
    }, []);
```

## 7.12 Push Notifications

This section explains how push notifications are implemented for both iOS and Android platforms. A user can receive notifications for several events, outlined in 6.4.3. Here is an example of a notification a user received because he got challenged:



Swisscom 43%

15:12 Sunday, 11 June

FitChallenge Just now

Fitness Challenge

Max Mustermann has challenged you to Pull-ups!

Gmail • [redacted]

[redacted] + 2

Google 32 min ago

26° in Bubikon

Mostly cloudy · See full forecast

WhatsApp • 4 messages from 3 chats 13:58

[redacted] + 1

9GAG 10:20

[redacted]

Manage notifications

Figure 7.17: Android Notification Example

As explained in 6.4.3, for Android devices, push notifications are sent using Firebase Cloud Messaging, which generates a unique key for each user and handles the notification delivery process.

In the case of iOS devices, push notifications are sent through the Apple Push Notification Service. Although Firebase Cloud Messaging offers Method Swizzling to send notifications to iOS devices, for this project, we decided to utilize the push notification service provided by Expo.

This service can be viewed in detail here. As explained in 6.5.1, Expo allows us to use a single codebase for both platforms. Using this SDK, Expo uses the correct notification service for the targeted platform 6.4.3.

When a user logs in or opens the app, we generate an expo-token, which we save in the user's Document. This token will be updated every time a user changes his device. We do this by utilizing the package expo-notification. We first request Notification permission, and if permission is granted, we generate an expo-token ("app/services/notification/Notification.js" -> "registerForPushNotificationsAsync"). We can then use this token to send Notifications to Android and iOS devices.

We then listen to specific events in the Firebase Cloud Functions 7.3.1 and push notifications to the correct user.

## 7.13 Tracked Data

This section will describe what kind of data we track for each user in our application. The data we track can have multiple purposes and be used by other functional requirements. The data we decided to track finally, and were discussed with our stakeholders, are the following (listed with their name in the database, "trackedData" collection 6.3):

- **Total Challenges played (totalChallengesPlayed):** The number of all Challenges a user played. Incremented after Challenge is completed.
- **Number of Challenges won (totalWins):** The number of Challenges a user won. Incremented after Challenge is completed and if a user is the winner of that Challenge
- **Win to Loss ratio (winLossRatio):** The ratio of wins to losses of Challenges. Calculated with the totalChallengesPlayed and totalWins. Wins divided by losses.
- **Total number of repetitions (totalRepetitions):** A total of all the repetitions a user has done across all his Challenges, active or completed. It gets incremented each time a user performs on a Challenge.
- **Weekly Summary (/weeklySummary subcolllection):** A summary of the repetitions performed for a user for each week for each challenge type. Used for 7.9.1.
- **Played Against (/playedAgainst subcollection):** A document is created in this subcollection for each unique user a user has played against. Counts the total games played against a user. Used in 7.9 to calculate the favorite opponent.

We collect the Tracked Data of the users in the Collection "trackedData." A user may only have at maximum one Document. A user may not have any trackedData Document if he never played in a Challenge.

### 7.13.1 Challenge completed

One event that triggers an updating of the Tracked Data is if a Challenge is completed. The metrics updated are:

- totalChallengesPlayed
- totalWins

- winLossRatio
- playedAgainst/id of user played against -> numberOfGamesPlayed

We use http onCall trigger after a Challenge is completed. If a user cannot beat the score of his Opponent, he lost the Challenge, and the Challenge is completed. We use an HTTP onCall trigger since no firebase trigger can listen to a certain field. With a onWrite trigger, the "completeChallenge" task would be invoked every time something is written into the Challenge Document. That means unnecessary invocations and costs that could be avoided. Keep in mind that this call does update not only the Tracked Data of a user, but also other tasks that need to be done if a Challenge is completed (outlined in 7.3.1). To see how these tracked data are displayed view 7.13.

### Sequence

Suppose the user could not beat the score, which is checked in the frontend. In that case, the frontend will complete the Challenge and send a httpsCallable request to our firebase functions 7.3.1, specifically the "completeChallenge" function. After this request is sent, the following sequence of code is run:

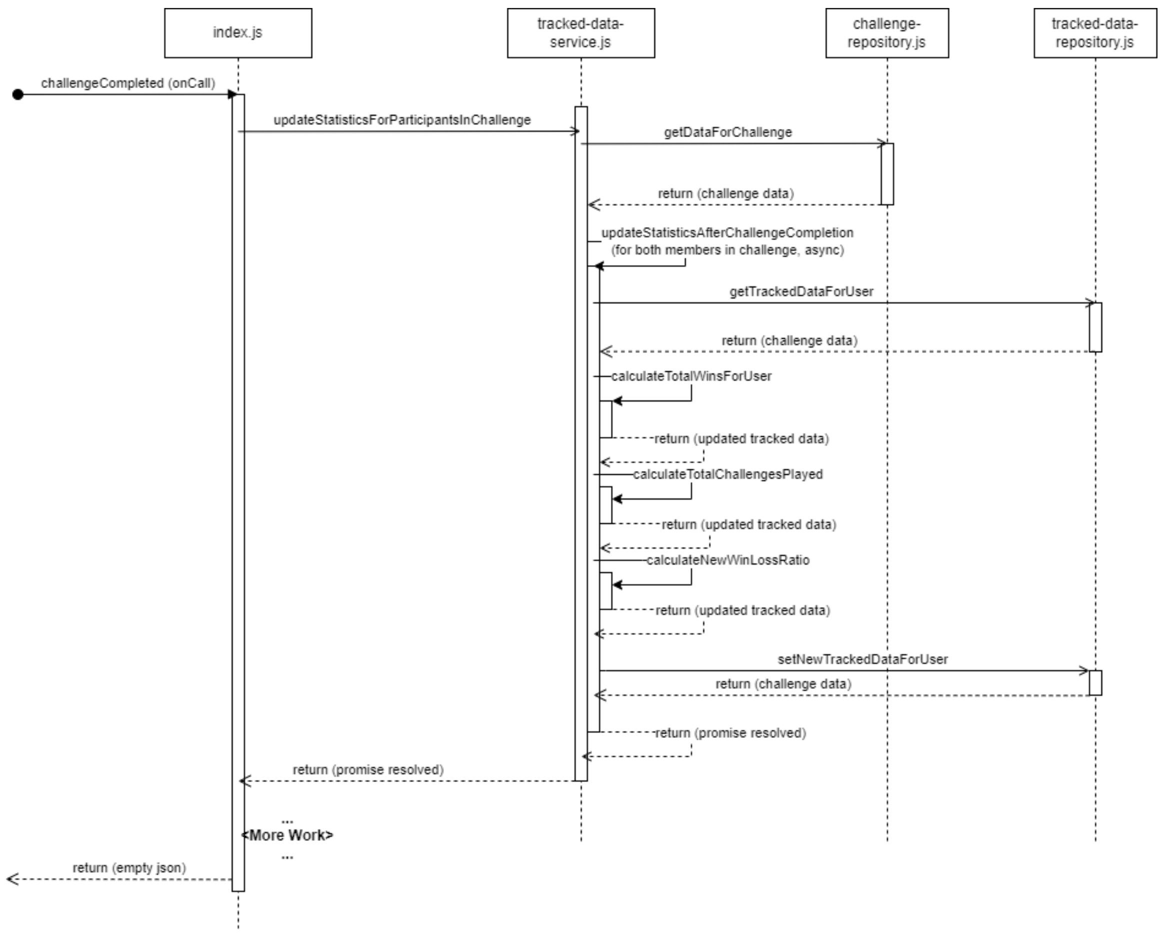

index.js
tracked-data-service.js
challenge-repository.js
tracked-data-repository.js
challengeCompleted (onCall)
updateStatisticsForParticipantsInChallenge
getDataForChallenge
return (challenge data)
updateStatisticsAfterChallengeCompletion (for both members in challenge, async)
getTrackedDataForUser
return (challenge data)
calculateTotalWinsForUser
return (updated tracked data)
calculateTotalChallengesPlayed
return (updated tracked data)
calculateNewWinLossRatio
return (updated tracked data)
setNewTrackedDataForUser
return (challenge data)
return (promise resolved)
return (promise resolved)
...
<More Work>
...
return (empty json)

Figure 7.18: Tracked Data Update After Challenge Completion

Keep in mind that there is more work done in the "completeChallenge" function, which is not displayed in the above graphic 7.18 and can be viewed in 7.1.

Even though a firebase function should only return a result to the caller when all promises are resolved as described in 7.3, the frontend calls the function but does not wait for its completion. The updates this function does will automatically be displayed in the frontend, through subscription and Real-time Updates we do on specific datasets. So there is no need to respond to the function completion

The function "updateStatisticsForChallengeCompletion" is run two times for both users in the Challenge. They run parallel to each other.

### 7.13.2 User Performed on Challenge

When a user performs on a challenge, two metrics are updated:

- totalRepetitions: Gets incremented by the amount performed.
- weeklySummary/<date of current monday> -> pullups: The amount of pull-ups the user performed, will be added to the summary of the current week.

The Firebase Cloud Functions that gets triggered when a user performs is called "newPerformanceAddedToHistory" 7.3.1. It gets triggered whenever a new Document is created in the "history" subcollection in a Challenge document 6.3.

Since we currently only have one challenge type, only pull-ups are counted towards the weekly summary. But as with everything, we programmed it to allow it to track different challenge types. So this code does not have to be altered in any way to track different challenge types since we use the challenge type as the field name in the weekly summary document 6.3 (how to add a new challenge is explained in 7.8.1). The weekly summaries are displayed in the 10-week summary visible in the profile 7.12. As the document's id, we use the date of the Monday of the week it references (format: yyyy.mm.dd). This value is also saved in the field "startDateOfWeek," which is used for ordering.

As in the "challengeCompletion" function 7.13.1, this function is not only designed to track metrics but do general work that needs to be done after performing on the Challenge, such as sending notification to the other user 7.12. Complete outline what this function does is explained in: 7.3.1.

### 7.13.3 Track new Data

The data tracking was intentionally designed in a way that can be easily extended in the future. We only track data after a Challenge is completed or when a user performed on a Challenge.

#### Track more Data after challenge completion

The function that needs to be extended if we want to track more data for a user after a Challenge is completed is the following (firebase/functions/services/datatracking/tracked-data-service.js):

```
async function updateStatisticsAfterChallengeCompletion(
  userUid,
  challengeSnapShotData,
) {
  let trackedDataToBeUpdated = await getTrackedDataForUser(userUid);

  // Devtipp: add new method call that calculates new metrics
  // for the user here (similar structure)
  trackedDataToBeUpdated = calculateNewTotalWinsForUser(
    userUid,
    challengeSnapShotData,
    trackedDataToBeUpdated,
  );

  trackedDataToBeUpdated = calculateTotalChallengesPlayed(trackedDataToBeUpdated);

  trackedDataToBeUpdated = calculateNewWinToLossRatio(trackedDataToBeUpdated);

  await incrementAmountPlayedAgainstUser(userUid, challengeSnapShotData);

  await setNewTrackedDataForUser(userUid, {...trackedDataToBeUpdated});
}
```

There are a few things that must be kept in mind while extending this function to collect more data for a user (**as long as it updates the trackedData document directly**):

- The function must accept a parameter of type "TrackedData".
- The parameter passed to the function must be the "trackedDataToBeUpdated" variable, as seen above.
- The function must overwrite or add a new field to this Document.
- The function must return the updated Document.
- The function return must override the "trackedDataToBeUpdated" variable as seen in the code above.
- If the function depends on other metrics, it has to be placed in the correct sequence.
- If the function is async, saving the Document should only happen if the call is completed.

When a new field is added to the "trackedData" Collection of a user, the converter "services/converters/TrackedData.js" has to be extended in the following way:

```
export const trackedDataConverter = {
  toFirestore: (trackedData) => ({
    ...
  }),
  fromFirestore: (snapshot, options) => {
    // Todo: Try to use return of promise.then() (see functions index.js)
    const data = snapshot.data(options);
    return new TrackedData(
      data.totalWins ?? 0,
      data.totalChallengesPlayed ?? 0,
      data.winLossRatio ?? 0,
      data.totalRepetitions ?? 0,
      //Like this:
      data.someNewData ??
      "some default value, if user doesnt have this field or document"
    );
  },
};
```

If we want to update data that is not a field that directly lies in the "trackedData" document, add a new line as seen with the "await incrementAmountPlayedAgainstUser(userUid, challengeSnapShotData)" call.

#### Track or check more badges after User Performed on Challenge

If more metrics should be updated after a user performs, one can add a line directly in the firebase functions function "newPerformanceAddedToHistory". Make sure to await for the updating to complete, and that they are in the correct order. Also, update the converter as described in the section above 7.13.3.

## 7.14 Badges - Milestones

Badges are pictures/icons a user can earn for achieving certain milestones. The Badges will be displayed in the Banner of the user 7.14, which is displayed in the profile 7.13 or visible for other users if they open a Challenge 7.8 they are a part in.

Currently the user can earn the following Badges:

- **First Win:** This badge is earned when a user wins his first Challenge.
- **One Hundred Games Played:** This badge is earned when a user completes one hundred challenges.
- **One Thousand Repetitions Done:** This badge is earned when a user did one thousand repetitions of any challenge type.

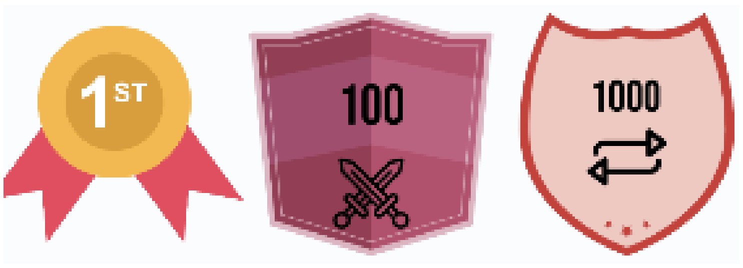


Figure 7.19: Earnable Badges

To fulfill this requirement, we use Firebase Storage, Firestore, and Firebase Cloud Functions. We save the image of the badge in Firebase Storage under "badges/" folder. Each badge in our system has a unique Document in the "badge" Collection with the badge type as it's id. We also save the URL that points to Firebase Storage image in the Document, which belongs to this badge. Each user also has a unique Document in the "earnedBadges" Collection, where we save an array of references that point to the Badges a user has earned. This Document is created when the user completes his first Challenge through firebase cloud function 7.3.1. Below 7.20 is an excerpt of our domain model 6.3 that shows how this is represented in our Firestore:

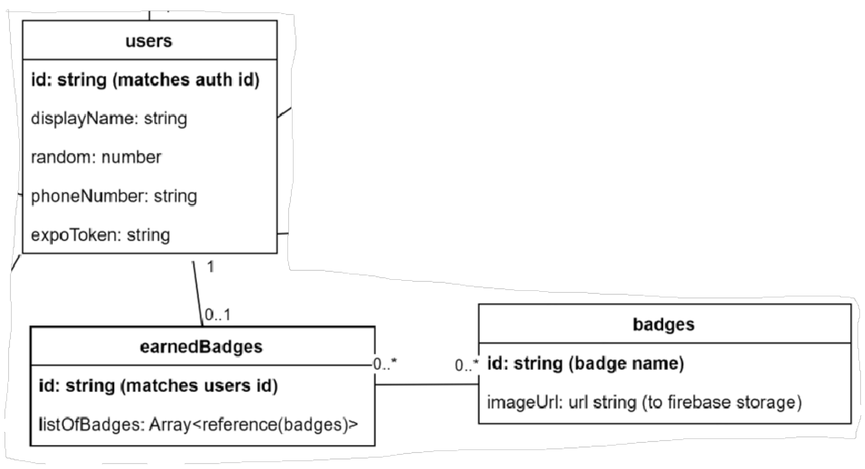

users
id: string (matches auth id)
displayName: string
random: number
phoneNumber: string
expoToken: string
1
0..1
earnedBadges
id: string (matches users id)
listOfBadges: Array<reference(badges)>
0..*
0..*
badges
id: string (badge name)
imageUrl: url string (to firebase storage)

Figure 7.20: Domain Model Badge Excerpt

### 7.14.1 Sequence

Below is a diagram that displays the sequence that is done to check badge conditions every time a Challenge is completed.

Checking conditions if a user earned a badge is done after a Challenge is completed. The same firebase function that updates the Tracked Data also updates the Badges 7.3.1. The function is called "completeChallenge." The Badges must be checked after the Tracked Data is updated. The reason is that the Badges depend on the Tracked Data and should only be checked if those are up to date.

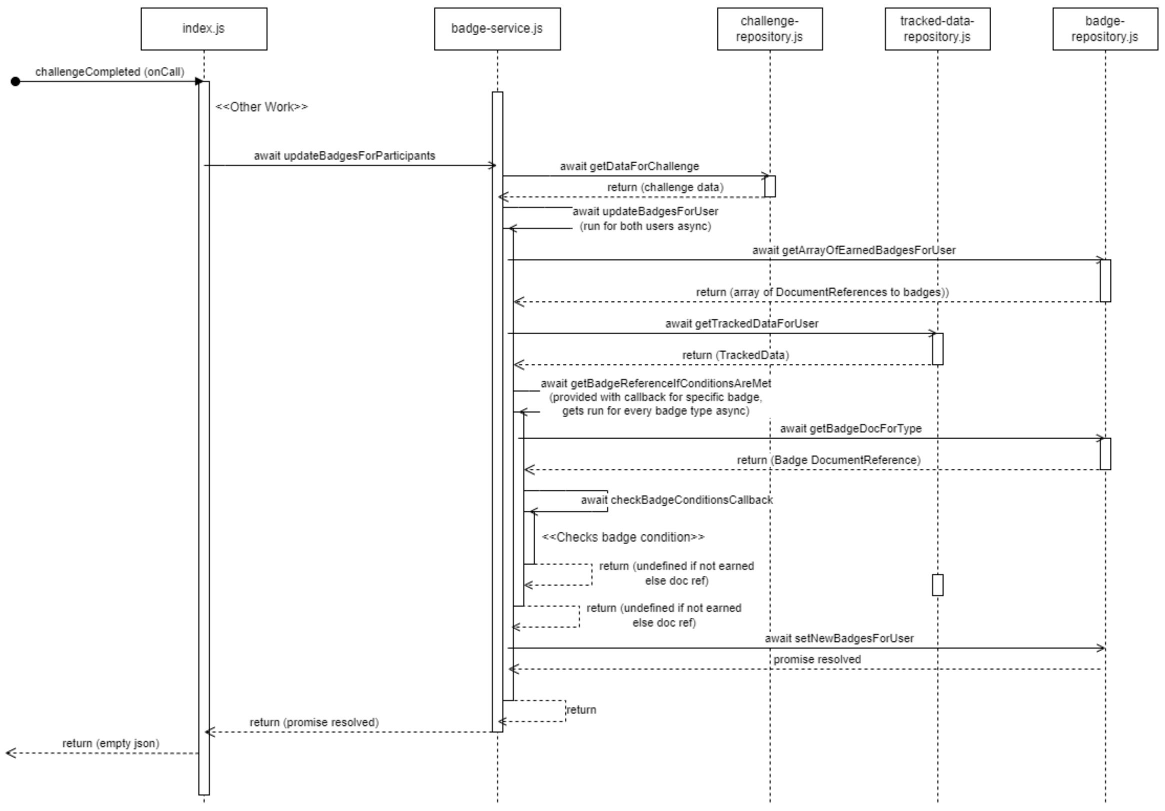

index.js
badge-service.js
challenge-repository.js
tracked-data-repository.js
badge-repository.js
challengeCompleted (onCall)
<<Other Work>>
await updateBadgesForParticipants
await getDataForChallenge
return (challenge data)
await updateBadgesForUser (run for both users async)
await getArrayOfEarnedBadgesForUser
return (array of DocumentReferences to badges))
await getTrackedDataForUser
return (TrackedData)
await getBadgeReferenceIfConditionsAreMet (provided with callback for specific badge, gets run for every badge type async)
await getBadgeDocForType
return (Badge DocumentReference)
await checkBadgeConditionsCallback
<<Checks badge condition>>
return (undefined if not earned else doc ref)
return (undefined if not earned else doc ref)
await setNewBadgesForUser
promise resolved
return
return (promise resolved)
return (empty json)

Figure 7.21: Update Badges Sequence

The diagram is a rough overview of how the code flows when updating the Badges for a user. It is an abstraction, and for an in-depth understanding, the code should be referenced, and the diagram helps to understand that code.

The code was designed in a way that there is no code duplication and to optimize performance. The checking of the Badges for both users is done async to each other. We also tried to keep read and write access to the database to a minimum to improve performance. Only Badges that a user did not earn are checked.

We will go into more depth about one part of the code that needs some explenation since the design of the code might not be so clear (firebase/functions/services/badges/badge-service.js).

```
async function getBadgeReferenceIfConditionsAreMet(
  userUid,
  badgeType,
  listOfBadges,
  trackedDataOfUser,
  checkBadgeConditionsCallback,
) {
  try {
    let earnedBadgeReference;
    const badgeDocument = await getBadgeDocForType(badgeType);
    if (!listOfBadges.some((earnedBadge) => badgeDocument.id === earnedBadge.id)) {
      functions.logger.info("User has not earned badge yet.
        Check conditions if user has earned badge");
      earnedBadgeReference = await checkBadgeConditionsCallback(
        userUid,
        badgeDocument,
        trackedDataOfUser,
      );
    } else {
      functions.logger.info("User already earned this badge.
        Skip condition check for badgeType: ", badgeType);
    }
    return earnedBadgeReference;
  } catch (err) {
    functions.logger.error(‘Error occurred while checking if badge conditions were
    met for badge: ${firstWinBadgeTypeName}.
    Skip updating of this badge due to error:‘, err);
    return undefined;
  }
}
```

We decided to implement a method like this above to minimize code duplication. Some stuff needs to be done for every badge that is checked, for example, retrieving the badge Document and checking if the user already earned the badge. After that, we callback a function that does the specific check if the user meets the condition to earn this badge. The function then returns undefined or the badge Document if the user earned it. For a further deep dive into the code, refer to section 7.14.2.

### 7.14.2 Add new Badge

This section will explain the process of adding a new badge and provide a more in-depth understanding of how the code works.

The first step that needs to be done is to add a new image that will be used for the Badges to the Firebase Storage under "badges/". Open the image and copy the URL in the browser. After that navigate to firebase Firestore and add a new Document to the "badges" Collection, with the following field:

- imageUrl: url to the image that you copied before

Make sure to give as the id of the badge, it's type (ex: firstWin).

Now navigate to the "badge-service.js" file. Write a new async function with the same signature as:

```
async function getFirstWinBadgeReferenceIfConditionsAreMet(
  userUid,
  badgeDocument,
  trackedDataOfUser,
) {
    ...
    //returns undefined if condition is not met
    //or returns badgeDocument if condition was met
    ...
}
```

Inside of this function check if the user has met the conditions to earn this badge. Return undefined if not and return the provided "badgeDocument" parameter if condition was met.

Go to the top of the "badge-service.js" file and add a line as described in the comment:

```
const firstWinBadgeTypeName = "firstWin";
//Add string for new badgeType
//needs to match string used as the id of the badge document.
```

Now navigate to the "updateBadgesForUser" function

```
export async function updateBadgesForUser(userUid) {
  const listOfBadges = await getArrayOfEarnedBadgesForUser(userUid);
  const trackedDataForUser = await getTrackedDataForUser(userUid);
  const badgeArrayResult = await Promise.all(
    [
      getBadgeReferenceIfConditionsAreMet(
        userUid,
        firstWinBadgeTypeName,
        listOfBadges,
        trackedDataForUser,
        getFirstWinBadgeReferenceIfConditionsAreMet,
      ),
      // Add new "getBadgeReferenceIfConditionsAreMet" call
    ],
  );
  badgeArrayResult.forEach((badgeDocument) => {
    if (badgeDocument) {
      functions.logger.info("Add new earned badge to bageList for user: ",
        userUid, "with badge: ", badgeDocument.id);
      listOfBadges.push(badgeDocument);
    }
  });
  await setNewBadgesForUser(userUid, listOfBadges);
}
```

Add a new getBadgeReferenceIfConditoinsAreMet call in the list where the comment is in the above code. Change the second parameter to the string you defined at the top of the file, and change the latest parameter with the function you just have written that checks the condition.

Congratulations now you have a new Badge that a user can earn, which will be displayed in the Banner 7.14.

### 7.14.3 Badge Copyrights

The first win badge was used from Freepik Company, which is free to use, but has to be referenced here.

The other two badges were created using Badge Generator, which from our understanding is completely free to use and has no copyright as explained here.

## 7.15 Localization

We provide language support for both German and English, as outlined by our requirements 5.1. To support multiple languages in our app we use i18n-js. We can then define language strings as follows (app/constants/i18n/translation.js):

```
const translations = {
  en: {
    // Screen titles
    challenges: "Challenges",
    rankings: "Rankings",
    profile: "Profile",
    home: "Home",
    ....
    },
   de: {
    // Screen titles
    challenges: "Herausforderungen",
    rankings: "Rangliste",
    profile: "Profil",
    home: "Start",
    ...
    }
}
...
const i18n = new I18n(translations);
```

```
i18n.locale = Localization.locale;
...
export default i18n;
```

We can then use it as follows, which will then display the language that was set by "Localization.locale":

```
    <Tab.Screen
        name={i18n.t("challenges")}
        component={HomeScreen}
        options={{
          tabBarLabel: i18n.t("home"),
          tabBarIcon: ({ color, size }) => (
            <MaterialCommunityIcons name="home" color={color} size={size} />
          ),
        }}
      />
```

We use the device language of the device that accesses our app. If the language is not present in our translations, we use English.

## 7.16 Technical Test Dokumentation

To get an overview what kind of tests we used, refer to 10.4.

For both frontend and backend we decided to only use unit tests as technical tests as explained in our test concept 10.4. We also created a manual test protocol 7.16.4.

### 7.16.1 Firebase Cloud Functions Testing

#### Unit Tests

There were three main ways how we could unit test our Firebase Cloud Functions.

- Create a clone project, which the unit tests connect to.
- Mock all firebase SDK function calls
- Use the local emulators for testing

Mocking all Firebase Admin SDK function calls was ruled out quickly. It would make testing our repository files almost unnecessary since we would have to mock pretty much the whole function, and it would make the tests very static and not test the actual functionality. We also did not want to create a whole new Firebase project. We must constantly clone our configurations to the Firebase test project. Moreover, as far as we have researched and seen in forums, there is no inbuilt feature to mirror a Firebase project. Therefore, we would have to manually export and import the Firebase configurations.

The most elegant solution for us was to use the local firebase emulator suite. The same rules apply to the local emulators since we define our security rules locally and push them to the Firebase project. In combination with the emulators, we use jest to write our unit tests.

To allow our unit tests to connect to the local emulators, we had to change a few things on how we connect to the Firebase services. The file where we connect to our Firebase services and also provide the Firestore instance for database operations is "firebase/functions/firebaseinstances.js". Which was updated to look like this:

```
...
export function initializeFunctionApp() {
  if (!initialized) {
    if (process.env.TEST !== "true") {
      //Production
      initializeApp();
      firestoreInstance = getFirestore();
      initialized = true;
      testEnvInitialized = false;
    } else {
      //Testing
      if (!admin.apps.length) {
        initializeApp({
          credential: admin.credential.applicationDefault(),
          projectId: "your-project-id",
        });
        getFirestore().settings({host: "localhost:8080", ssl: false});
        firestoreInstance = getFirestore();
        initialized = true;
        testEnvInitialized = true;
      }
    }
  }
}

export function getFirestoreInstance() {
  if (initialized && firestoreInstance != undefined) {
    return firestoreInstance;
  }
  throw new Error("Trying to use firestore but
    app has not been initialized yet");
}

//Used only in testing
export async function deleteFunctionsApp() {
  if (initialized) {
    await deleteApp(getApp());
    initialized = false;
  }
}
...
```

As seen, we only initialize the app if it has not already been initialized; otherwise, it throws an error. As seen above, the app gets initialized differently, depending on the value set in the process.env.TEST env variable. If it is set, the app gets initialized in test mode, using the local Firestore emulator instead of trying to connect to the production services running on the server. We also created a function that deletes the app "deleteFunctionsApp()". This is only used in testing. The above file was also unit tested to ensure it behaves as expected.

We have defined a "firebase/functions/test/tests.setup.js" file, which does some work, before and after each test, which is required for every test case.

- Before every test: Defines the process environment "process.env.TEST = "true"" and calls "initializeFunctionApp()" from the firebaseinstances.js. I am setting up a clean environment for each test.
- After every test: Calls "deleteFunctionsApp()" to tear down the Firebase app, to get a clean environment. Each test file also deletes all the Firestore data after each test.

We decided not to use a predefined set of test data that gets loaded on startup but to initialize an empty database. The reason for that is that we want to test as broadly as possible. So each test sets up its test data. Here is an example from "badges-repository.unit.spec.js":

```
...
const expectedBadgeRef = db.collection(badgesCollectionName)
    .doc("firstBadge");
await db.collection(badgesCollectionName).doc(badgeId).set(firstBadge);

const retrievedBadge = await getBadgeDocForType(badgeId);

expect(retrievedBadge).toEqual(expectedBadgeRef);
...
```

This test setups a Document, which then should be retrieved by the tested method. For more or different examples, please refer to the code.

**How to run the tests**

This is a short guide on how to run the tests:

1. **Go to firebase project**: Navigate to the "firebase/functions" folder.
2. **Install the** firebase cli tools: Run "npm install -g firebase-tools" (one time setup)
3. **Login to your firebase account:** Run "firebase login". (one time setup)
4. **Make sure the cli uses the correct project:** Run "firebase use <firebase project id>" (one time setup)
5. **Make sure all node packages are installed:** Go to the functions folder and run "npm install"
6. **Start the local emulators:** Run "firebase emulators:start", and wait for them to be fully started up.
7. **Run the tests:** Run "npm test" to run all unit tests or run single test cases using the IDE.

### 7.16.2 Frontend Testing

**Test Scope**

We decided to only automatically unit test the services and repositories and not test the screens and Components in our frontend code. This has two reasons, which both play into each other. First, almost all business logic is being done in the services and repositories. The screens and Components almost always call these functions and do not implement significant business logic themselves. Therefore, focusing on and covering the services and repositories entirely made sense. Also, since the UI changes a lot in the project process, adjusting the tests every time would be very time-consuming.

Moreover, this brings us two the second reason for this decision, time. As elaborated in 10.3.1 Joel Suter has become unavailable for some time, and we needed to make some concessions, which are elaborated in more detail there. Therefore, there may not be the test coverage we first wanted to achieve.

So what we tested in these tests specifically was the following:

- Database access: Do the write and read operations behave as expected?
- Business logic: Do the services behave as expected?
- Real-time Updates: Do we correctly subscribe to the data?

**How it was tested**

We also use the jest testing framework to unit test our frontend. We do not use the prodduction environment but the local Firebase emulators. Similar to how we test the functions 7.16.1, we have to ensure that our tests connect to our local emulators and not try to connect to the production environment. We achieve this by adding the following conditions to the "app/firebase.js" file, which is responsible for the connection to Firebase:

```
...
if (process.env.NODE_ENV === "test") {
  console.log("Running in test environment");
  firebaseApp = initializeApp({
    apiKey: "testKEYVWpWwW8ftsPqQLu1D6CppZuP5B_nevbU",
    authDomain: "",
    databaseURL: "",
    projectId: "test-id",
    storageBucket: "",
    messagingSenderId: "",
    appId: "",
  });
  initializeAuth(firebaseApp, {
    // Keeps users logged in on app close
    persistence: getReactNativePersistence(AsyncStorage),
  });
} else {
  console.log("Running in prod environment");
  firebaseApp = initializeApp(firebaseConfig);
  initializeAuth(firebaseApp, {
    // Keeps users logged in on app close
    persistence: getReactNativePersistence(AsyncStorage),
  });
}

export const functions = getFunctions(firebaseApp, "europe-west1");

export const storage = getStorage(firebaseApp);
export const db = getFirestore(firebaseApp);

if (process.env.NODE_ENV === "test") {
  console.log("Connecting to emulator database");
  connectFirestoreEmulator(db, "127.0.0.1", 8080);
  connectFunctionsEmulator(functions, "127.0.0.1", 5001);
}
...
```

We check the process variable "NODE_ENV" which Jest will always set to "test" when running any or all tests. With this variable we setup the Firebase services differently. We also have no predefined set of test data, but prepare test data in each test individually, and after the test delete all the test data again.

We will quickly highlight how we tested Real-time Updates, since these are more exotic kinds of unit tests, and might be usefull to know how to test these for further development. The following function subscribes to the Tracked Data of a user (app/repositories/TrackedDataRepository.js):

```
export function getTrackedDataForUser(userUid, trackedDataCallback) {
  const docRef = doc(db, trackedDataCollectionName, userUid)
    .withConverter(trackedDataConverter);
  const unsubscribe = onSnapshot(docRef, (docSnap) => {
    if (docSnap.exists()) {
      const newData = docSnap.data();
      trackedDataCallback(newData);
    }
  });
  return () => {
    unsubscribe();
  };
}
```

How the Real-time Updates work is explained in 7.2.1. What we want to test here is if the data changes, that means is our callback "trackedDataCallback" run? We can achieve that by writing a test like this:

```
...
const unsubscribe = await getTrackedDataForUser(testUserUid, mockCallback);

// timeout is needed for callback.
await new Promise((resolve) => {
  setTimeout(async () => {
    // update doc
    const updatedTrackedData = new TrackedData(15, 10, 0.5, 100);
    const updatedTrackedDataDoc =
        doc(db, trackedDataCollectionName, testUserUid);
    await setDoc(updatedTrackedDataDoc, { ...updatedTrackedData });

    expect(mockCallback)
      .toHaveBeenCalledTimes(2);
    expect(mockCallback)
      .toHaveBeenCalledWith(trackedData);
    expect(mockCallback)
```

```
      .toHaveBeenCalledWith(updatedTrackedData);
    mockCallback.mockReset();
    unsubscribe();
    resolve();
  }, 1000);
});
...
```

In this test, we expect that our callback is run when we subscribe (get data as it is) and then run again when the subscribed data is updated (update doc). We expect first to run with the initial data (trackedData) and then run with the updated data (updatedTrackedData). Since we cannot control when the initial callback is run, we have to wait a bit with a timeout before doing any assertions. After we have done the assertions, we unsubscribe from the Real-time Updates to prevent unnecessary resources from being used.

**How to run the tests** Here is a quick guide on how to run the tests:

1. **Navigate to the Firebase project:** Go to the "firebase/functions" folder in the firebase project.
2. **Install the** firebase cli tools: Run "npm install -g firebase-tools". (one time setup)
3. **Make sure the cli uses the correct project:** Run "firebase use <firebase project id>" (one time setup)
4. **Make sure all node packages are installed:** Go to the functions folder and run "npm install"
5. **Start the local emulators:** Run "firebase emulators:start", and wait for them to be fully started up.
6. **Setup frontend:** Go to the "app" folder and run "npm install"
7. **Run the tests:** Run "npm test" to run all unit tests or run single test cases using the IDE

### 7.16.3 Testing concessions

Due to significant changes in the scope and focus as elaborated in 10.3.1, we did not adjust our tests to accommodate the last changes we did in the code. Focus was on finishing up certain features and not testing. That means the current tests do not cover the latest changes. Of course, this should be addressed if development should continue, as mentioned in 9.1.

### 7.16.4 Manual Test Protocol

We also decided to set up a manual test protocol that gets run after each sprint or when desired. At first, it was more informal, but we formalized it here. These tests will be used as integration tests. This Test protocol represents the final version of the test protocol. It had evolved throughout the project and was first created when a user could perform on a Challenge. We decided not to create a mirror Firebase project as explained in 7.16.1. The results of every time this protocol was run are not listed since it would clutter this document and were often run more informally.

#### Prerequisites

1. Navigate to the "firebase" folder of the project and run "firebase emulators:start –import=exported-dev-data"

2. Wait for the emulators to start and open the emulator suite UI through the link that is printed in the console.
3. Navigate to the "app" folder. Run "npx expo start" and scan the qr code.
4. Have atleast one contact in your phone that also is registered in the app.

**Protocol**

| Nr. | Action | Expected Result |
|---|---|---|
| 1 | Register as a new user | A new user was created in Firebase. The user sees "Home Screen." Chosen display name and default avatar is displayed in the glsprofile dashboard. No earned Badges and Tracked Data is 0 everywhere |
| 2 | Challenge another user (you have access to). Logout and login as the other user | See the Challenge you created under "Your Turn" |
| 3 | Perform on Challenge (score should be beatable) | Challenge should now be displayed in "Their turn". Also, check glsprofile and check if "total repetitions" was incremented by the score you achieved |
| 4 | Open the Challenge and view the Challenge details | Able to see the banner of the other user with his data. See what score he achieved. Able to see the Challenge details. |
| 5 | Logout, Login as first user, perform on Challenge and make sure you loose | Challenge should now be displayed as "Completed". Go to profile and check if the Tracked Data and Badges was updated as expected |
| 6 | View Rankings | Check if rankings were updated accordingly |
| 7 | Change Profile picture | Able to select picture and picture was updated in Profile. |
| 8 | Challenge the user again, press on the Challenge and write them a text. Log out and log in as that user. | The user received the message and can view it |
| 9 | Challenge a random user | Does a random user get challenged? |
| 10 | View a history of a challenge | Is the history as you expect it to be? |
| 11 | Go to the dashboard and view the milestones | User won the "first win" badge/milestone |
| 12 | Challenge a user with a number that does not have a account yet and you have access to | A sms with the download code should have been sent |

Table 7.3: Manual Test Protocol

**Edge test cases**

Everytime you run the test protocol, also try to do random inputs and see if anything breaks.

## 7.17 Deployment

The deployment and publishing for our application required a lot more permissions and effort than expected. Before publishing any builds on their respective stores we had to create app builds on the Expo pipelines 6.20.

As we are using the free tier of Expo we were put on a waiting queue and had to wait for priority users before we could build the application. These waiting times were sometimes up to 50 minutes and additionally the build can take up to 8-10 minutes as well. The whole workflow was very slow and if a build failed on the Expo pipeline we had to issue a new build with new version codes which required a lot of time.

After creating the builds for iOS and Android we would have to request a review for both app stores.

We have divided this section into two subsections explaining the process for both iOS and Android.

### 7.17.1 App Store

To create iOS builds, several steps were required. First, we need an Apple Developer Account, which our industry partner gave us access to. Secondly, we have to create a distribution certificate necessary for publishing the application on the store, and lastly, we need a key and share it with the "App Store Connect" website. Our industry partner had to give us full admin rights to his developer accounts to acquire all these components.

After uploading the build on the App Store Connect website, we had to fill in several forms, which are listed below:

- Copyright
- License Agreement
- Age Rating
- Category
- Pricing and Availability
- Screenshots
- App Privacy and Privacy Policy

After filling out the forms, we could send the build for review. At first, our review was rejected, but an additional review finally approved our app for publication in the iOS App Store. For further references see: App Store

### 7.17.2 Google Play Store

Publishing the application on Android required a service account connected to the publishing portal. Due to permission issues, only the account owner can create one; that is why we had our industry partner connect our service account to the portal before we could proceed with the publication.

To enable the automatic workflow of Expo we must manually upload a .aab file to the portal. After filling out the respective forms as listed in 7.17.1, we could send the build for review.

Due to time delays during this process (refer to section 10.3.1 and 8), the review process was not finished before we had to hand in the project at the time of writing this section (15.06.2023).

# 8 Result

In this project, we created a prototype for a mobile application that runs on Android and IOS that motivates people to do more physical exercises by "gamifying" them. This was achieved by creating a competitive platform allowing users to play physical exercise Challenges against other users.

The user's name and phone number are required to register for our service. He can select a profile image that will be seen by other users after signing up. The user can Challenge any of his contacts or a random user, who will receive a push notice that he has been challenged, to compete in a physical activity which is currently limited to "pull-ups." He is able to participate in this Challenges and will attempt to outperform his rival. The other player will then aim to surpass this score once more. This will continue until one user is unable to surpass the score, at which point his rival is crowned the winner. We added features that encouraged the user to be competitive, such as recording statistics, earning milestones as Badges, and displaying those in a Banner that is displayed to his opponent, in order to add more gamification elements. This gives him encouragement to compete in more exercises and give his opponent a sample of what he would be up against. A global ranking system that shows the top 100 participants was also put into place.

Screenshots of the developed app can be viewed in 13.

The following table 8.1 contains the completed functional requirements as stated in the task formulation 15 with deviations if there are any, or further elaboration of the result that fullfills this requirement.

| **Requirement** | **Result** |
|---|---|
| Registration of new accounts using the phone number | As stated |
| Login of existing accounts | As stated |
| Creating Challenges | As stated |
| Implement atleast one Challenge utilizing handy sensors. | Created a pull-up challenge using Accelerometer data from the device. |
| Access to the contact list of the User. | Access contact list to allow the User to choose an opponent of his contacts. |
| Challenges can be sent to a person | Able to challenge Opponent from the contact list or a random user, which then receives a Notification |
| A screen providing an overview of all active Challenges including additional information like current repetitions | Implemented by the Home Screen 13.3 |
| If a user is not registered with our app, the User can be prompted to download the application with a shared link | User can choose to share the link through several different communication services. |

| | |
|---|---|
| A dashboard containing interesting information for the User (for example, a view for earned Badges) | Implemented using the Profile and the Banner associated with the user 13.4 |
| Sending push notifications to User who gets challenged | As stated. We even send Notifications for different events mentioned in 6.4.3 |
| Provision of a chat for participant communication. | Able to chat with users that are part of the same challenge. |
| Sending Challenges back and forth, including text messages and GIFs, in a mutual manner | Partially achieved. User is able to send challenges back and forth. |
| Creation of ranking lists at both the friend level and global level | Partially achieved. Created a global ranking system that displays top 100 users with the most wins. |
| User can challenge a random user | As stated |

Table 8.1: Implemented Functional Requirements

The following table will list all the non-functional requirements that were implemented and achieved from the task formulation 15 with a description how they were achieved and or checked.

| **NFR** | **Result** |
|---|---|
| The app works on Android and IOS | Achieved using Expo 6.5.1, development and testing on both platforms |
| Push notifications are continuously sent | Using Firebase Cloud Functions that are triggered by certain events and send Notifications to the targeted user. |
| Errors should not cause system failures but instead, display an error message and revert the system to its previous state | Firebase implements such measures, and due to the nature of Firebase Cloud Functions, a crash in one function does not cause a full system failure. |
| Every error should be logged in the system | Every sensitive data access is checked for error. Firebase Cloud Functions does also employ a global error logger, that logs any errors that might not be handled by us. |
| Access to the contact list of the User. | Able to access contacts on both platforms using Expo |
| Data filled in input fields should be validated before being processed by the system. SQL injection tests on the input fields should not reveal any vulnerabilities | Partially achieved. Firebase has server-side input validation which is very secure, but we did not implement frontend validation or injection tests, as stated in 6.7 for reasons stated in 10.3.1 |
| User-passwords are not persisted in plain text | We do not have any passwords we save from the user. Authentication is done with an OTP. |
| When a user logs into the app, only their data or the data they have access to will be displayed to them | We only give access to data that the user has a right and need to see. Implemented using Firestore security rules 6.7 |

| | |
|---|---|
| The business logic in the backend should be modularly structured, allowing for easy expansion and extension | Both frontend and backend were designed to allow for easy extension and expansion as stated in the implementation chapters. |
| The backend should be able to handle 50 requests per second | Achieved due to the nature of Firebase scaling 6.4.6. |
| The database should be able to manage 10’000 Challenges and 2’000 users | This is also achieved by the nature of Firestore and the scaling of Firebase 6.4.6. |

Table 8.2: Implemented NFRs

## Not Achieved

This section will address the functional and non-functional requirements that were not or only partially met. The reasons for not implementing them boil down to time constraints caused by occurred risks 10.3.1. The backlog was initially prioritized with our stakeholders, and as soon as we noticed the time constraints, the open tasks were again prioritized with our stakeholders. So everything we could not achieve was accepted by them, and had lower priorities than others.

### Functional

#### Sending Challenges back and forth, including text messages and GIFs, in a mutual manner

This requirement was partially completed. The user can send challenges back and forth and compete in them. The user cannot send a text message or GIF after performing a challenge or when he invites another user to a challenge.

We do, however, have a chat screen for participants to communicate.

#### Creation of ranking lists at both the friend level and global level

This functional requirement was also only implemented to a certain degree. Our application has a global ranking displaying the top 100 players with the highest wins. We do not, however, have a friends ranking list.

### Non-functional

#### The backend API should be tested using API testing tools

As mentioned in the realized risks 10.3.1 and in the testing chapter 7.16, we took focus away from testing due to the priority being on implementing the functional requirements and our application being a prototype. We employed no backend API testing tools for our Firebase Cloud Functions.

#### The implemented functionalities (database, backend) should be deployed, and the app should be published on Google and Apple App Stores

Database and Backend are both deployed using Firebase. App Store has approved our review, and the application can be viewed on the App Store. See chapter 7.17.1. The frontend of our mobile app must still pass the review to be published in the Play Store. For this reasons, we do not consider this NFR as achieved. Reasons for not achieving it are stated in 10.3.1.

**The loading time of the app's content should not exceed 150ms**

We did not use frontend testing to test if the app's content loads in the given time.

# 9 Conclusion

In this project, we created a basis for an extendible service that motivates users to be more physically active by adding gamification elements to physical exercise. This is done by implementing a service that allows users to challenge their friends or random users to physical exercises. We added a lot of gamification elements, such as performance statistics, achievements (Badges), and a Banner where they can show their experience. Furthermore, we implemented a global ranking that users will strive to climb.

We are thrilled with the result and had a great time developing such a unique idea. Both of us are very interested in sports, especially the gym, and thought that the idea to gamify some aspects is a genius idea with a lot of potential. We would like to see it further developed and fleshed out. We employed state-of-the-art technologies, and the experience we gained from them and starting a project from scratch to a complete prototype, employing gamification techniques, will be invaluable to us and our careers.

We achieved almost everything we set out to achieve. Unfortunately, a significant accident Joel Suter had 10.3.1 set us back more than we would like and did not allow us to achieve everything as formulated in the task 8.

## 9.1 Needs to be Adressed

Even though we are delighted with what we have achieved, some things should be addressed if the development of this service continues before implementation of any new features should be considered. These are our recommendations, and are informed based on our knowledge of the application. We will list these here, in a rough order of priority:

- **Testing**: Since we had to make some compromises, we did not test this application to the extent required from an app that, for example, should go into production. So before any more work is done, the application testing should be extended (more unit tests and fix existing, api testing, frontend testing, more usability tests, firebase test project, ...)
- **Security:** Security concerns, as mentioned in this document, should be addressed and evaluated. It might be advisable to get an expert in the field that can provide detailed feedback. Should security be expanded more? Are there any concerns regarding phone authentication, frontend data access, input validation, and security rules of Firebase.
- **Improve repetition detection of exercises:** As mentioned in 7.8.3, detecting repetitions of pull-ups using a "hardcoded" solution of handy sensors is very hard to perfect. Improving the detection should be addressed before a rollout to production. In our opinion, the best solution to improve it is to use machine learning or external trackers such as Fitbit.
- **Not achieved NFRs and Functional Requirements:** As mentioned in 8, we did not achieve everything that we had planned. Before developing new ideas, these requirements should be revisited and reevaluated if they should be implemented next.

- **Evaluate Modularity**: We have implemented everything focusing on modularity. Where refactoring is needed, we mentioned it in the implementation chapters. Those should be refactored, and the rest of the code should be checked for modularity.
- **Reevaluate Datastructures of Firestore**: We are satisfied with the data structures we used for the implementation, but it might be advisable to recheck them and see if anything can be optimized to improve performance and readability of the code.
- **Performance**: Performance should be evaluated using more extensive datasets and automated tests and improved if needed.

## 9.2 Future Vision

While developing the application, we noticed ourselves thinking about new features and improvements to current features that could be considered for future development. Since we think this application has a lot of potential we will list those recommendations here. They are not sorted in any kind of priority. In our opinion the list of additional requirements listed in the task formulation 15 would also be very good additions and should be counted towards this list.

- **More Badges/Milestones:** Add more earnable badges and milestones.
- **Seasons:** Add a seasonal approach, similar to many modern games. Seasonal badges, rankings etc.
- **Frame for Profile picture:** Allow the user to earn a special frame in which his profile picture will be placed (flames, thorns, Etc).
- **ELO System:** Create a new or additional rankings system that is based on ELO (similar to chess), that ranks players on their "recent" performance and not the total of the games played.
- **Description:** Add description or explanations for certain elements (badges). A "tutorial" could also be considered the first time a user opens the app.
- **More challenges:** Implement more challenge types for different physical exercises or even other health related metrics.
- **More authentication methods:** Implement more authentication methods instead of just phone numbers.
- **Switching to friends list:** Switching from the contact list to a managed friends list should be considered to make integrating different authentication methods easier.
- **Teams:** Enable creating teams; users can join. This could allow for new features such as Team vs. Team challenges, rankings, banners, and more.
- **Synced challenges:** Implement challenges that are not async as currently but done in sync. Good for when both users are in the same space (party game)
- **Group challenges:** Allow challenges with more than two users. Enables features such as internal group rankings and more.
- **Customizable Banner:** Allow the user to customize what he wants to show in his banner. Which trackers and or badges he wants to show?
- **Banners in Rankings:** Allow users to see the banners of users in the rankings table.

- **Better leaderboards:** Add more detailed leaderboards. It could be extended to display leaders of a given challenge.
- **Events:** Events could be implemented, for example a daily push-up event, where users can participate and earn rewards for placements.
- **Connection to health api's or external trackers:** Implement connection to external trackers such as Garmin. This would benefit from better accuracy in detecting certain metrics, such as steps and altimeter data, and allows for many more challenges without having to implement the detection algorithm.

# 10 Project and Time Management

## 10.1 Project Plan

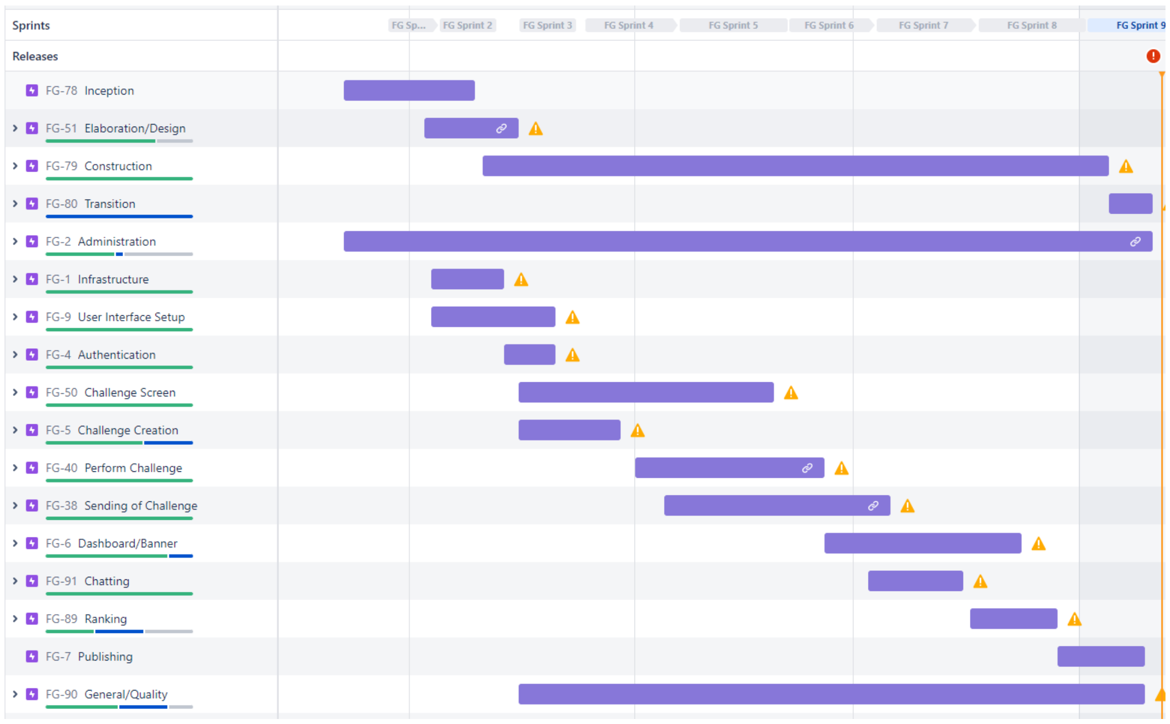


Figure 10.1: Project Plan

The figure 10.1 shows our initial project plan at the beginning of the project. In the end, it deviated due to some factors that were unknown at the beginning of the project and some risks that were realised 10.3.1.

We will quickly explain what the specific milestones mean. We will begin with the milestones specific to RUP 10.2, since we used a mix of Scrum and RUP. We used milestones like these, to guide the whole project plan and all other milestones. They are a very high level overview of the project plan. We use them as a general check to see if we are still on path, even though we might deviate from the other milestones and their deadlines.

- **Inception**: Summarizing and finalizing all Non-functional requirements and the given task.
- **Elaboration/Design**: Designing the application. Work like mock-ups and technical decisions. It has cross-over with the inception phase since designing an application might influence the requirements.
- **Construction**: The central part of the project, Contains all the implementation work. It also overlaps the design since beginning with the Construction might impact the design.

- **Transition**: This is the last phase of the project. It happens when all the major construction work is finished. It contains work like publishing to the app stores, finishing the documentation, and preparing Transition to Industry Partner.

The following milestones are a more detailed representation of the milestones above, where we split our tasks and features into different epics. All our work can be applied to those epics.

- **Administration**: This was work on the documentation, meetings, and general work that could not directly be associated with an issue.
- **Infrastructure**: Everything to do with Infrastructure, such as Firebase
- **User Interface Setup**: The initialization of the User interface, Mockups, technology decisions, and setting up a connection to backend services.
- **Authentication**: Work regarding the authentication of users into our application.
- **Challenge Screen (Home Screen)**: Creating and implementing the landing page/home screen.
- **Challenge Creation**: Implementing logic that allows a user to create a challenge.
- **Perform Challenge**: Allowing users to perform/participate in a challenge.
- **Sending of Challenge**: Sending of the Challenge between users (including completing a challenge).
- **Dashboard/Banner**: Work regarding the profile and also the Banner of a user.
- **Chatting**: Implementing a chatting function that allows users to communicate with each other.
- **Ranking**: Implementing a ranking system.
- **Publishing**: Publishing the Application to the IOS App Store and Android Play Store
- **General/Quality**: General work regarding quality assurance, such as testing, small bug fixes, usability tests, and more.

### 10.1.1 Deviations

As in all projects, we ended up deviating from the project plan in some aspects. They will be listed here

- **Authentication**: Authentication took longer than expected due to insufficient documentation mentioned in 10.3.1.
- **Publishing**: Due to some communication errors and insufficient permissions. This milestone was partially achieved, and was stretched to a few days before our deadline.
- **Elaboration/Design**: Since we significantly reworked the project scope, the design phase took longer than expected.
- **Infrastructure**: In the project's progress, we often returned to the Infrastructure and had to make minor changes.
- **Dashboard and Banner**: First, we decided to cut some tasks from this milestone because of a risk that occurred 10.3.1. However, at the end of the project and after a discussion with our Industry Partner, we decided to implement these features too.

- **General**: As we progressed in the project's development, some issues for milestones we considered to be completed arose after the completion of the milestones. These were often minor tasks that mainly addressed edge cases that only became evident later. Also, because of an injury Joel suffered 10.3.1, a lot of the milestones took longer.

## 10.2 Project organization

We have organized ourselves using a combination of RUP and Scrum, which was recommended to us by our Industry Partner. As we have learned from our SA, working on a project that starts from scratch takes much work to predict precisely. RUP has the advantage that we can create a very high-level long-term plan, which helps a lot to guide the project. However, we still use Scrum to create a more detailed project plan and use our project's general Scrum iteration process. We have minor deviations which are listed here:

- Our sprints last two weeks except for the first sprint, which we decided on for one week as it only consisted of administrative and setup issues.
- Our review meetings will be held every week. Inconsistencies can occur due to advisors and clients who have different work schedules.
- We scratched the Daily SCRUM Meeting due to overhead and effectiveness for a team of only two people. The dailies were held more informally.

### Roles

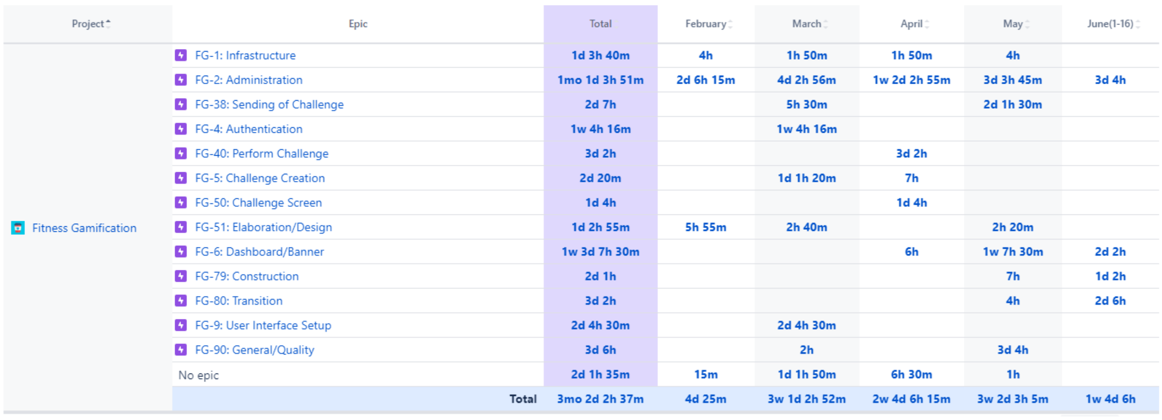
Usually, SCRUM-based projects divide their participants into roles. We decided to leave the role distribution out because we are only a team of two developers. Our team members divide their work depending on who has time and more experience in a given issue. As both team members bring similar experiences in the field, there is no need to define specific roles for individuals. The other team member will review any code or major decisions someone makes.

### 10.2.1 Issue management

For issue management, we decided to use Jira Cloud Solution. Both of the team members have experience with this Issue Tracking Tool. We have divided our requirements into epics which will be further divided into more minor issues. The only release we captured is the final submission, so it is easier to estimate which epics must be done in a time frame.

### 10.2.2 Time tracking

The following screen capture 10.2 will show the time spent in the whole duration of the project, categorized by each epic and month. We will not list the time spent per person since both members worked in sync. However, due to an injury 10.3.1, Joel Suter could not work for some time, which can be seen in the dip in productivity in April.



| Project | Epic | Total | February | March | April | May | June(1-16) |
|---|---|---|---|---|---|---|---|
| Fitness Gamification | FG-1: Infrastructure | 1d 3h 40m | 4h | 1h 50m | 1h 50m | 4h | |
| | FG-2: Administration | 1mo 1d 3h 51m | 2d 6h 15m | 4d 2h 56m | 1w 2d 2h 55m | 3d 3h 45m | 3d 4h |
| | FG-38: Sending of Challenge | 2d 7h | | 5h 30m | | 2d 1h 30m | |
| | FG-4: Authentication | 1w 4h 16m | | 1w 4h 16m | | | |
| | FG-40: Perform Challenge | 3d 2h | | | 3d 2h | | |
| | FG-5: Challenge Creation | 2d 20m | | 1d 1h 20m | 7h | | |
| | FG-50: Challenge Screen | 1d 4h | | | 1d 4h | | |
| | FG-51: Elaboration/Design | 1d 2h 55m | 5h 55m | 2h 40m | | 2h 20m | |
| | FG-6: Dashboard/Banner | 1w 3d 7h 30m | | | 6h | 1w 7h 30m | 2d 2h |
| | FG-79: Construction | 2d 1h | | | | 7h | 1d 2h |
| | FG-80: Transition | 3d 2h | | | | 4h | 2d 6h |
| | FG-9: User Interface Setup | 2d 4h 30m | | 2d 4h 30m | | | |
| | FG-90: General/Quality | 3d 6h | | 2h | | 3d 4h | |
| | No epic | 2d 1h 35m | 15m | 1d 1h 50m | 6h 30m | 1h | |
| | Total | 3mo 2d 2h 37m | 4d 25m | 3w 1d 2h 52m | 2w 4d 6h 15m | 3w 2d 3h 5m | 1w 4d 6h |

Figure 10.2: Time Tracking Report

### 10.2.3 Division of Tasks

We mainly worked together on most tasks, but still, there were some divisions on how the overlying requirements were divided between each other. The rough divisions of tasks will be listed below. Some tasks both worked on equally, and that will be mentioned too. The following table 10.1 will outline who had the main lead in each task.

| **Task** | **Lead** |
|---|---|
| Firebase Setup | Both |
| Frontend Setup | Lucas v. Niederhäusern |
| Requirement Elaboration | Joel Suter |
| Authentication | Joel Suter |
| Frontend Design | Lucas v. Niederhäusern |
| Firestore | Joel Suter |
| Cloud Functions | Both |
| Performing on Challenge | Lucas v. Niederhäusern |
| Creation of Challenge | Both |
| Sending of Challenge | Both |
| Profile and Banner | Joel Suter |
| Home Screen | Both |
| Ranking | Lucas v. Niederhäusern |
| User Statistics | Joel Suter |
| Chat Functionality | Lucas v. Niederhäusern |
| Push Notifications | Lucas v. Niederhäusern |
| Quality Assurance | Joel Suter |
| Localization | Lucas v. Niederhäusern |
| Documentation | Both |

Table 10.1: Division of Tasks

## 10.3 Risk Management

This section will describe the risks to our project with their respective probability and severity. If they occur, they are listed in 10.2 The section will be constantly updated for new risks that might become evident later in the project. Any countermeasures and their effectiveness will be documented as well.

| Risk ID | Risk | Countermeasure | Severity | Probability |
|---|---|---|---|---|
| 1 | Team member is unavailable | Communicate through channels to distribute tasks, and try to keep as few dependencies as possible. | Very High | Very Likely |
| 2 | Scope Creep | Any changes to scopes must be estimated again, and the plan must be adjusted accordingly. | High | Very Likely |
| 3 | Technical inexperience | Assign issues to the person best suited. Eliminate any inexperience as early as possible. | High | Likely |
| 4 | Infrastructure breakdown | Create backups and ensure high availability of the infrastructure. | Very High | Unlikely |
| 5 | Poor project planning | Review the project plan after every sprint. | High | Likely |
| 6 | Technical impossibility | Research and analyze requirements regarding their technical needs and possibilities. | Very High | Improbable |
| 7 | No available libraries/resources for project needs (especially sensor libraries) | Analyse possible research libraries and their abilities as early as possible so that project can evolve around those findings. | Very High | Unlikely |
| 8 | Equipment failure | Keep care of equipment and be able to switch to a new system with no more than a day expended. | Very High | Likely |
| 9 | Changing specifications | Try to fix app specifications and project requirements at the beginning of the project and minimally change them during the duration. | Moderate | Unlikely |

Table 10.2: Project Risks

This graph helps us decide if we implement the suggested countermeasures from the table above 10.2—anything below the red line we accept and do not try to mitigate.

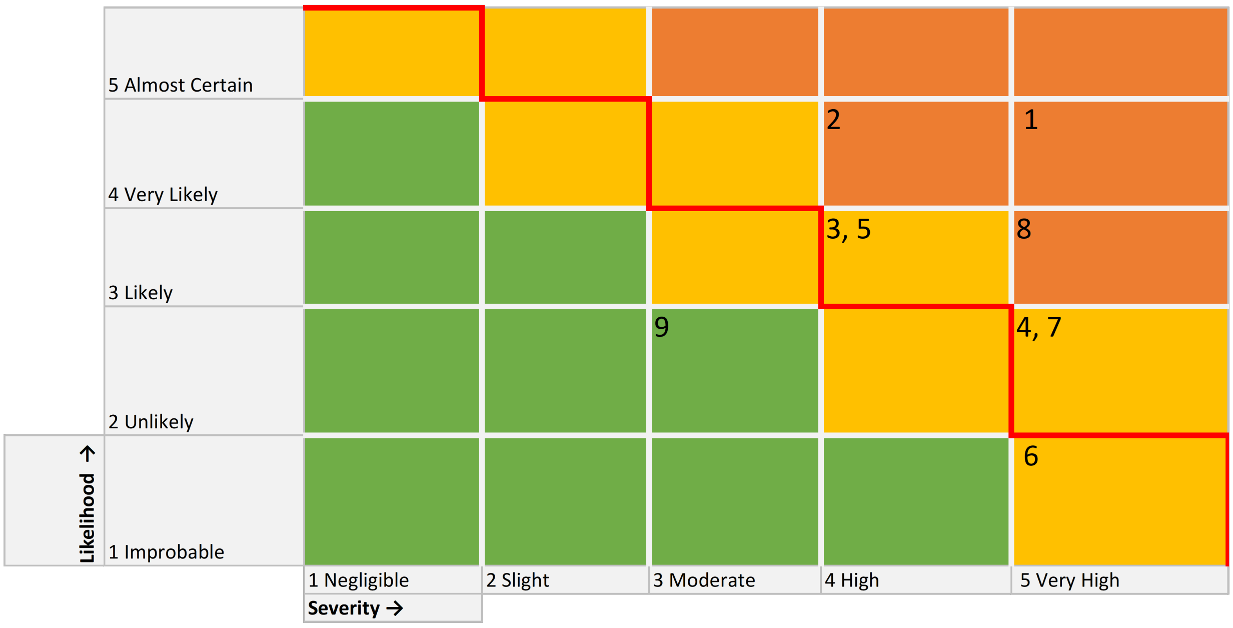


Figure 10.3: Risk acceptance graph

### 10.3.1 Realized risks

**Team member is unavailable**

This risk was realized and realized in a big way. In the middle of the project, Joel Suter had an accident while doing sports, where he injured his knee, which needed pretty quick surgery. Because of that injury plus surgery, Joel Suter was unavailable for at least 2-3 weeks of the project: One and a half weeks right after the injury and surgery, and the other week accumulated because of doctor and rehabilitation appointments. So the impact, as expected if a team member was unavailable, was very severe. Because of that, a few features and quality aspects could not be fulfilled entirely as planned. We estimated the total **lost time about 45+ hours**. Because of that lost time, we had to decide with our Industry Partner which features, NFR, or quality aspects to take away some focus. The focus should be on functionality, modularity, and design. Focus was taken away from some quality aspects such as testing, testing some non-functional requirements, security and additional features. Those decisions guided what we focused on. What we were not able to achieve can be viewed both in the results 8 and conclusion 9. It is hard to pinpoint exactly what was not achieved because of this risk.

**Actual Severity**: Extremely Severe

**Countermeasure success**: It was successful up to a point. However, there is only so much one can do when someone is unavailable for almost three weeks. We ensured that Lucas v. Niederhäusern had no dependencies on Joel Suter before the surgery and that he could continue working. Joel Suter also ensured he was available for calls or text messages if any arose.

**No available resources for project needs**

An issue that cost time was insufficient or no documentation/examples for some Firebase features. The problem was that we only discovered some of these issues when we began implementing them. When we did the research, it seemed very obvious. The critical point here was for Firebase phone authentication. When we researched, we found many examples of it. However, when we started implementing the functionality, we realized it was more challenging to implement with the Expo toolset, and almost no examples were available online. That cost us about two full days of work. Another example is that some of the Firebase documentation needed to be updated since a new version of the SDK was just released.

**Actual Severity:** High

**Countermeasure success:** Partially successful. We ensured at the beginning that we were not trying to do anything impossible. However, as always, the problems make them visible while implementing. And at that point, we weren't able to change course since the cost of that would be higher than trying to work around it.

**Technical Inexperience**

We generally knew all the project technologies except for Firebase. We were familiar with it and somewhat with its capabilities. However, this is our first time working with the Firebase JS SDK. We took this risk because we were highly interested in the technology. We were held back at a few points while developing since we needed to determine how certain things could be implemented. However, the time it saved us in aspects like real-time updates, automatic scaling, etc., made up for it. Moreover, the decision to choose Firebase was accepted by our industry partner.

**Actual Severity:** Low

**Countermeasure success:** Successful

**Technical Impossibility**

This risk occurred while implementing phone authentication for Firebase, highlighted in 7.4.3. In short, firebase phone authentication does not differentiate between login and register. Even though we did research before, we did not notice that the process for both is the same. This caused the issue that we need a different login and registration process. The Problem was that a user that has not registered yet could log in (and register that way), and a registered user could use the register and log in that way. We had to implement a workaround 7.4.3, which cost us a little time.

**Actual Severity:** Minor

**Countermeasure success:** Partially successful

**Poor project planning / Technical inexperience**

During the final phase of this project, we encountered significant time constraints when it came to publishing. Because we lacked experience in publishing apps on official stores, we sought guidance from our industry partner and agreed to initiate the publishing process at least one and a half weeks prior to the deadline.

Upon obtaining the necessary credentials for developer accounts, we encountered obstacles due to permission restrictions on our accounts. After exchanging numerous emails, encountering communication errors, and facing limited availability, we could only submit our applications for publishing three days before the deadline.

Due to permission issues, our industry partner had to upgrade our account permissions multiple times. Ultimately, we required full admin access to both the iOS and Android developer portals before we could submit our applications to the stores. Certain aspects of the submission had to be handled manually by our industry partner due to their exclusive account ownership rights.

Apple has approved our submission of the application and it can be viewed on the App Store. See chapter 7.17.1.

As of the writing of this document (15.6.2023), we have not received a response from Google regarding our submission, and we cannot guarantee the availability of our application on the Google Play Store.

This risk can be attributed to both inadequate project planning, a lack of technical experience and communication errors.

Insufficient research and a lack of prior experience caused us to underestimate the level of effort required for the publishing process on official stores.

**Actual Severity:** High

**Countermeasure success:** Not successful

## 10.4 Test Concept

Generally, we will conform to the testing Pyramid and design our test concept accordingly, as it is considered a good guideline for software testing. Since we are developing a prototype, testing was not the most significant focus, and we will not test the application to the extent that might be expected from an application that runs in production. The test concept should be reworked if the application should be deployed into production.

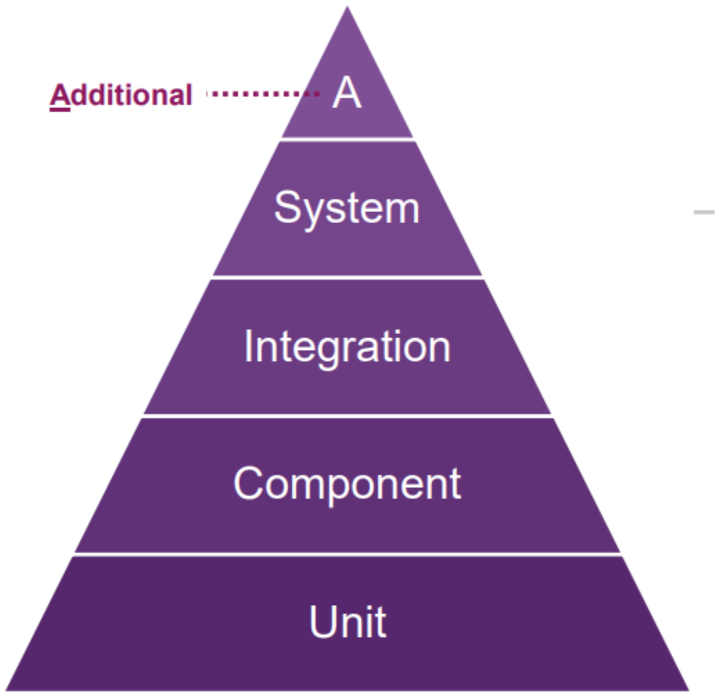


Figure 10.4: Testing Pyramid

### 10.4.1 Additional

As of now, we do not see the need for additional test techniques since the scope of our project is overseeable. We instead concentrate on the following testing techniques.

### 10.4.2 System

We will test the system manually using a test protocol 7.16.4. This protocol will evolve during the duration of the project. If one of the tests fails in the protocol, the Sprint cannot be finished; it does not meet the definition of done. The final iteration of the protocol will be listed in the technical test documentation.

### 10.4.3 Integration

We cover these with our unit tests and manual testing. So we decided against Integration testing.

### 10.4.4 Component

Since we have small modules/components, testing them separately makes little sense. For that reason, we will not test the modules separately but will cover these tests throughout the unit and manual testing.

### 10.4.5 Unit

We will implement most of our testing through unit tests 7.16. With these, we want to ensure that all the code units run as intended.

### 10.4.6 Deviations

We could not implement the initially planned test concept fully due to some realized risks 10.3.1 and the shift in focus because of that. Concessions in unit testing were highlighted in 7.16.3.

### 10.4.7 End User Usability Tests

This section will go into the usability test protocol, the criticism we received, and the end rating they give the application.

Generally, in a usability test, one would do a pre-interview, where we ask general questions about what kind of apps similar to ours he uses. Since we do not do market research, we only ask demographic questions and questions related to our functionality, design, and general look and feel.

We interview people aged 18-30, who do sports or maybe are interested in sports but are not motivated, do sports with friends, and like to be competitive with friends.

**Pre-interview**

- How old are you?
- What gender are you?
- What do you do professionally?
- What is your relationship to sports?

**General Knowledge Goal**

What do we want to find out with this test generally?

- What is the general impression of our design?
- Where do the test subjects get lost? Which screens/elements cause the user to get lost?
- Does the user understand the intention of the displayed elements (buttons, titles, etc)?
- Does the user understand the flow of the application?
- Does the user have specific feedback regarding the design?
- Does the user have ideas for new functionality?
- Did the user discover any edge cases or ideas we did not consider?

**Prerequisites**

**For Test subject:** Create a contact in your phone with the phone number: "redacted (account we setup for testing)." Do you have access to your phone? Download the app and open it.

**Scenarios**

The test scenarios we give the user. We generally give him a goal and make notes where the user had problems and if he achieved the goal.

| Nr. | Name | Scenario | Knowledge Goal |
|---|---|---|---|
| 1 | Creating Account | You want to create an account in the app | Does the user understand how to create an account? |
| 2. | Creating Challenge | You want to create a pull-up challenge against the user with the phone number "redacted" | Does the user know how to create a challenge, and does he understand the create challenge form? |
| 3. | Send message to opponent | Taunt your opponent. Send them a message. | Does the user know how to contact his opponent through the app? |
| 4. | Perform on challenge | **For Tester:** Challenge the user. **For test subject:** Perform a challenge and beat the score | Does the user understand when it is his turn? Does the user understand how to perform? Does the user know if he won? |
| 5. | Check challenge details | Tell us the name of the opponent you performed against. Tell us what his previous score was. What is his profile picture? How many repetitions has the user done in his entire career? | Does the user understand the concept of the banner? Does the user understand the challenge details screen? |
| 6. | View Rankings | Tell us which user has the most repetitions in his career | Does the user find the rankings and understand the information displayed? |
| 7. | Viewing Profile | Find your profile and tell us how many repetitions you have. Did you earn any milestones? If yes, tell us how many | Can the user find his profile? Does he understand the tracker concept? Does he understand the concept of the badges? |
| 8. | Change profile picture | Change your profile picture to an image you like | Is the user able to change his profile picture? |
| 9. | View Challenge History | Tell me the last result of an opponent of a challenge of your choosing | Does the user find the history for a challenge? Does he understand the screen? |
| 10. | Logout | Logout of the app | Does the user understand how he can logout? |
| 11. | Login | Log back into the app | Is he able to log back into the app |

Table 10.3: Usability Test Protocol

**Post-Interview**

What is the opinion of the user? (open questions)

- What are your general Impressions?
- What did you like about our app (generally and design)?
- What bothered you (generally and design)?
- Did you get confused at any point on how to proceed?
- Are there specific elements that you were unsure what their purpose was? Tell us which.

- Would you use the app? Tell us why yes/not.
- Give the app a rating from 0-10. (layout, responsiveness, colour, content)
- Do you have any general feedback, positive or negative?
- Do you have any suggestions for additional features? What would you be interested in?

**Consolidated Test results**

Here we will list the specific and general feedback we received consolidated from the section 14. We will also list the individual scores the user gave our app. Of course for a more accurate user feedback the testing scope should be extended to more users.

**Scenario Success**

Problems that arised in muliple users are for the following scenarios:

- **1:** Some users got hasty and tried to login before registering. Did not take time to read the text.
- **3:** The keyboard on IOS covers the input while typing (keyboard avoiding view does not work).
- **9:** It was not always clear how to access the history of a challenge.

**Consolidated Post-Interview Feedback**

- **What are your general Impressions?** Simple, very cool idea, maybe it is too easy to cheat in a challenge.
- **What did you like about our app (generally and design)?** Not too much information, simple, clean.
- **What bothered you (generally and design)?** Some buttons were not responsive (too small). The app is not optimized for Dark Mode. Else nothing really.
- **Did you get confused at any point on how to proceed?** Mostly no (else see scenario success)
- **Are there specific elements that you were unsure what their purpose was? Tell us which.** Meaning of the various badges. Descriptions and toutorial would be nice.
- **Would you use the app? Tell us why yes/not.** For additional motivation. But before anyone would use it it needs anti cheat and better recognition of repetitions.
- **Give the app a rating from 0-10. (layout, responsiveness, colour, content)**
  - **Layout:** 9, 10, 8, 8
  - **Responsiveness:** 10, 6, 8, 10
  - **Colour:** 4, 5, 10, 8
  - **Content:** 6, 7, 7, 6.5
- **Do you have any general feedback, positive or negative?** Empty home screen does not look so nice. Colour needs to be improved, and anti cheat is needed. Generally but very satisfied.
- **Do you have any suggestions for additional features? What would you be interested in?**

  - Tracker support
  - Better repetition detection
  - More challenge types
  - Challenges that can be done everywhere, such as push-ups
  - Description for elements such as badges
  - Tutorial

## 10.5 Definition of Ready / Definition of Done

This section will discuss and list the different DoDs and DoRs. These lists will serve as checklists either before we do something (DoR) or when we want to check if something is finished (DoD). In addition, it will ensure that quality standards are met with a clearly defined process.

### Definition of Ready

If any of the criteria in the checklist are violated, the concerned item is not ready to be worked. It will have to be corrected until it meets them. It is ready only if all criteria are met.

- **User Stories**
  - Acceptance Criteria defined
  - Story definition accepted by Team
  - Story well-defined
  - No ambiguities
  - Story estimated
  - Story in current Sprint
  - Story assigned
- **General Ticket**
  - Acceptance Criteria defined
  - Ticket in current sprint
  - Ticket estimated
- **Sprint**
  - Tickets with highest priority in sprint
  - Sprint backlog prioritized
  - No hidden work, everything in tickets
  - Sprint planned according to capacity
  - All Stories meet definition of Ready

## Definition of Done

If any of the criteria in the checklist are violated, the concerned item is considered unfinished. It will not be allowed to be moved to done or regarded as finished. It is done only if all are met.

- **User Stories**
  - Assumptions of User Story met (justified if not)
  - Project builds without errors
  - Tests written according to test concept
  - Story tested against acceptance criteria
  - Configuration, Architecture or Build changes documented
  - Code Review passed
  - Code written according to defined guidelines
  - NFR stay unviolated
- **General Ticket**
  - Assumptions of Ticket met (justified if not)
  - Changes to Documentation reviewed
  - Acceptance Criteria met
  - No Contradictions in Documentation
  - Decisions accepted by Team
- **Sprint**
  - DoD of each Ticket included in Sprint is met
  - Sprint scope completed
  - All tests pass
  - Backlog refined and updated
  - Application deployed
  - Upcoming Sprint planned

# Bibliography

[Ker17] Todd Kerpelman. “Cloud Firestore vs the Realtime Database: Which one do I use?” In: *Firebase Blog* (2017). URL: https://firebase.blog/posts/2017/10/cloud-firestore-for-rtdb-developers/#the-tldr-just-tell-me-what-to-use.

# List of Figures

1 Firebase . . . . . 1
2 React Native . . . . . 1
3 User Profile . . . . . 3
4 Home Screen . . . . . 3
5 Challenge Details Screen with Opponents Banner . . . . . 3
6 Challenge Perform Screen . . . . . 3

4.1 System Context . . . . . 15

5.1 Use Case Diagram Final . . . . . 17
5.2 Use Cases Version 1 . . . . . 19
5.3 Mind Map Alternatives . . . . . 20

6.1 Container Diagram . . . . . 24
6.2 Component Diagram . . . . . 26
6.3 Domain Model . . . . . 28
6.4 Firebase Console . . . . . 30
6.5 Possible Authentication Methods . . . . . 31
6.6 Authentication Monitoring . . . . . 32
6.7 Firebase Functions Structure . . . . . 33
6.8 FCM Architectural Overview . . . . . 34
6.9 Firestore Monitoring . . . . . 36
6.10 Usage and Billing for Firebase . . . . . 37
6.11 Whats App Inspiration . . . . . 40
6.12 Polytopia Inspiration . . . . . 41
6.13 Apex Legends Banner . . . . . 42
6.14 Register Screen Mockup . . . . . 42
6.15 Login Screen Mockup . . . . . 42
6.16 Challenge Screen Mockup . . . . . 43
6.17 Challenge Details Screen Mockup . . . . . 43
6.18 Profile/Dashboard Screen with Banner Mockup . . . . . 43
6.19 Rankings Screen Mockup . . . . . 43
6.20 Expo Build . . . . . 45

7.1 Register Phone Authentication . . . . . 55
7.2 Register OTP Challenge . . . . . 56
7.3 Create Challenge Form . . . . . 58
7.4 Opponent Selection Screen . . . . . 60
7.5 Choose Number Dialog . . . . . 60
7.6 Contact not allowed Snackbar . . . . . 61
7.7 Home Screen . . . . . 62
7.8 Challenge Details Development . . . . . 63
7.9 Challenge History Screen . . . . . 64

7.10 Challenge Perform Screen . . . 65
7.11 Profile Screen . . . 69
7.12 10 Week history graph . . . 70
7.13 Profile Screen . . . 71
7.14 Banner . . . 72
7.15 Chat Screen . . . 73
7.16 Rankings Screen . . . 74
7.17 Android Notification Example . . . 75
7.18 Tracked Data Update After Challenge Completion . . . 77
7.19 Earnable Badges . . . 80
7.20 Domain Model Badge Excerpt . . . 81
7.21 Update Badges Sequence . . . 81

10.1 Project Plan . . . 101
10.2 Time Tracking Report . . . 104
10.3 Risk acceptance graph . . . 106
10.4 Testing Pyramid . . . 109

13.1 Login Screenshot . . . 136
13.2 Register Screenshot . . . 136
13.3 Home Screen Screenshot . . . 137
13.4 User Profile Screenshot . . . 137
13.5 Challenge Details Screenshot . . . 137
13.6 Challenge History Screenshot . . . 137
13.7 Challenge Perform Screenshot . . . 138
13.8 Rankings Screenshot . . . 138
13.9 Create Challenge Screenshot . . . 138
13.10 Opponent Selection Screenshot . . . 138
13.11 Select Phone Number Screenshot . . . 139
13.12 No Number in Contact Screenshot . . . 139
13.13 User Does Not Exist Screenshot . . . 139
13.14 Share App Screenshot . . . 139
13.15 Change Profile Picture Screenshot . . . 140
13.16 Chat Screenshot . . . 140

# List of Tables

5.1 Use Case Description . . . 19
5.3 Non-Functional Requirements . . . 22
5.5 Optional Requirements . . . 23

7.1 Implemented Cloud Functions . . . 54
7.2 Challenge Document after Creation . . . 59
7.3 Manual Test Protocol . . . 91

8.1 Implemented Functional Requirements . . . 95

8.2 Implemented NFRs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 96

10.1 Division of Tasks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104
10.2 Project Risks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105
10.3 Usability Test Protocol . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 111

14.1 Scenario Notes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 141
14.2 Scenario Notes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 143
14.3 Scenario Notes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 145
14.4 Scenario Notes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 147

# 11 Meeting Minutes

## Protocol 18:00 23.02.2023

### Present

Joel Suter, Lucas von Niederhäusern, Frank Koch, Michael Güntensperger

### Review

- None

### Topics / Questions

- Requirement, if user does not have the application he is prompted to download it from the respective app store, can only be tested late into the project
- How does the publication of the application work?
- Should we use Kanban or SCRUM for project management
- Shall we use the repository from our client again?
- Can we do a double deployment for test / production or should we do testing directly onto the production deployment?
- Which NFR's are required or must / should / can?
- Does step counter count as sensor for a challenge?
- Does a group only exist for one challenge or for multiple?
- Functional requirement "Dashboard should contain interesting information about the user" is very broad

### Decisions

- Access to contacts required to send challenges to friends, users not searchable with our application
- Don't use steps for challenges instead use sensors to detect pushups or pullups (difference in height)
- Be able to send challenges back and forth
- Have a challenge ID in the URL of the invitations
- Challenge starts immediately, opponent does not have to accept

- There are winners, losers and bums (people who did not participate in the challenge)
- Change distribution of points depending on how many players there are
- Upload demo version to stores for review from Apple and Google
- We will get access for publishing our application from Michael
- Group is persistent over multiple challenges
- Players can be removed from groups
- Statistic stay the same even if players are no longer in the group
- Dashboard with statistics about yourself (no graphs required, simple numbers)
- Use Scrum with less overhead -> **Document this decision**
- Set up new environments to set clear boundaries to our previous project
- Document any usage of previous work
- Challenge result should be enriched with a ranking table
- Zwischenpräsentation 18. April 2023
- Bachelorprüfung -> Let Frank know asap when we get our agenda for the finals

### Open Tasks / Topics for next week

- Create priority list of requirements before next meeting
- Have a look at the technologies used for the projects for questions
- Read "Leitfaden Bachelorarbeit v1.1"

## Protocol 17:00 02.03.2023

### Present

Joel Suter, Lucas von Niederhäusern, Frank Koch, Michael Güntensperger

### Review

- Priority list for functional and non-functional requirements
- Present our project plan

### Topics / Questions

- Can we use React Native + Firebase for our application
- How does RUP + Scrum exactly work
- Questions to groups:
- When user creates group does one have to accept the invitation or is he accepted automatically
- Is a group invite only?
- Can every group member manage the group
- Is there an admin?
- How does a challenge in a group work?
- Are all group members challenged simultaneously or one after another?
- If someone does not beat the score is he immediately out?
- Can we use NoSQL as it is the standard for Firebase instead of MySQL?
- Can we only use mobile phone number and remove password completely for the authentication?
- General questions about challenges and challenge screen

### Decisions

- We use React Native + Firebase for our application
- Use SCRUM+ -> Split RUP Phases into Sprints
- Groups
- People are automatically added to groups
- Use Whatsapp as analogy for our project
- Groups are invite only, no search function for groups
- Person who creates group will be admin
- Function to promote other people to admin

- If admin leaves group and no other admin is present, everyone else will be promoted to admin
- One can either challenge a person or a whole group
- If someone joins a group with an active challenge, he will not participate in that challenge
- If someone leaves a group, the group will be deleted from his dashboard
- Create a distinctive separation between group challenges and 1v1 challenges
- Challenge titles are either group name or the opponent name
- There can only be one active challenge in a group at a time
- General feedback to mockup
- Document our inspiration from polytopia for our frontend
- Localisation -> App has to be in german, if possible support german and english
- No global statistics for the moment, have specific statistics for groups and challenges

### Open Tasks / Topics for next Week

- Create a proposal for group solution
- Meeting on wednesday 08.03.2023 15:00

## Protocol 18:00 23.02.2023

### Present

Joel Suter, Lucas von Niederhäusern, Frank Koch, Michael Güntensperger

### Review

- New Features without Groups (our priority list)
- Challenge random players (regional?) from the playerbase
- Banner system (Display badges / milestones of your work)
- Detailed Profile / Dashboard
- See profile / dashboard of other players
- Global ranking
- Friendslist ranking
- Total ranking of all players (elo ranking?)
- How about seasons? (optional)
- Chat function
- Implement more challenge types (..., step counter, not-fitness-related, stairs)
- Implement Anti-cheesing methods (optional)

### Topics / Questions

- Should we document decision about changing the scope?

### Decisions

- None

### Open Tasks / Topics for next week

- None

## Protocol 18:00 16.03.2023

### Present

Joel Suter, Lucas von Niederhäusern, Frank Koch, Michael Güntensperger

### Review

- State of current progress
- Updated mock-ups

### Topics / Questions

- None

### Decisions

- None

### Open Tasks / Topics for next week

- None

## Protocol 17:00 23.03.2023

### Present

Joel Suter, Lucas von Niederhäusern, Frank Koch, Michael Güntensperger

### Review

- Phone Authentification
- Google Authentification

### Topics / Questions

- How are we going to implement the friends list?

### Decisions

- Use phone number for the first phase to create a friends list, people who signed in using a different service can be implemented later

### Open Tasks / Topics for next week

- None

## Protocol 15:00 30.03.2023

### Present

Joel Suter, Lucas von Niederhäusern, Frank Koch, Michael Güntensperger

### Review

- Update to current progress in app

### Topics / Questions

- Firestore usage (Cloud Firestore or Realtime Firebase)

### Decisions

- None

### Open Tasks / Topics for next week

- None

## Protocol 17:00 06.04.2023

### Present

Lucas von Niederhäusern, Frank Koch, Michael Güntensperger

### Review

- Persistance of challenges

### Topics / Questions

- Joel not present due to injury

### Decisions

- None

### Open Tasks / Topics for next week

- None

## Protocol 17:00 13.04.2023

### Present

Lucas von Niederhäusern, Frank Koch, Michael Güntensperger

### Review

- Pull-up Challenge Performance

### Topics / Questions

- Joel not present due to injury

### Decisions

- No meeting next week due to midterm presentation
- Document Joel's absence and update our risk management / occurred risks

### Open Tasks / Topics for next week

- None

## Protocol 15:00 27.04.2023

### Present

Joel Suter, Lucas von Niederhäusern, Frank Koch, Michael Güntensperger

### Review

- Interim presentation
- Update on progress of our application
- Joel's absence

### Topics / Questions

- We might need Apple Developer Account for notifications

### Decisions

- None

### Open Tasks / Topics for next week

- Apple Developer Account

## Protocol 17:00 04.05.2023

### Present

Joel Suter, Lucas von Niederhäusern, Frank Koch, Michael Güntensperger

### Review

- Reviewing progress of badges and notifications
- Firebase pricing

### Topics / Questions

- None

### Decisions

- There will be no meeting next week
- Next Meeting will be on 16.05.2023

### Open Tasks / Topics for next week

- None

## Protocol 18:00 16.05.2023

### Present

Joel Suter, Lucas von Niederhäusern, Frank Koch, Michael Güntensperger

### Review

- Demonstration of notification and chat function
- Reviewing Firebase backend
- Reviewing progress of banners and badges

### Topics / Questions

- Time diffculties due to Joel's medical absence
- Should we persist chat history over multiple sessions?
- Who initiates a challenge?

### Decisions

- Frank will be absent next week
- Implement a limit for challenge requests
- Initiator should perform challenge first not the opponent

### Open Tasks / Topics for next week

- None

## Protocol 17:00 25.05.2023

### Present

Joel Suter, Lucas von Niederhäusern, Michael Güntensperger

### Review

- Review of the current state of the app

### Topics / Questions

- We need access to the developer accounts of Joel and Lucas

### Decisions

- Will be sent to us in following week

### Open Tasks / Topics for next week

- None

## Protocol 17:00 01.06.2023

### Present

Joel Suter, Lucas von Niederhäusern, Frank Koch, Michael Güntensperger

### Review

- Current state of the app
- These features might not be implemented in time:
- Display statistics visually (e.g. graphs)
- We cannot send GIFs in chat
- Additional badges

### Topics / Questions

- How should we implement challenge expiration? (Firebase Scheduled Functions)
- How do we test the speed of the app? (e.g. how long does it take to load a challenge)
- Push notifications are not working on Android (iOS is working)

### Decisions

- We will do a hard cut on the features we will implement next week and focus on publishing the app and documentation
- Use scheduled functions to implement challenge expiration (scheduled functions are triggered by a cron job every 10 minutes)
- We decided on which features we will implement in the next week
- Ranking system
- Useability improvements / testing
- Implementing the challenge expiration
- Challenge random opponent

### Open Tasks / Topics for next week

- None

## Protocol 17:00 08.06.2023

### Present

Joel Suter, Lucas von Niederhäusern, Frank Koch, Michael Güntensperger

### Review

- Update on the features we were able to implement in the last week

### Topics / Questions

- How does the poster have to look like? Do we have templates?
- Review of the documentation structure if possible

### Decisions

- Focus is now on the documentation and publishing the app
- There will be no meeting next week (unless there are any questions)

### Open Tasks / Topics for next week

- None

# 12 Personal Reports

**Lucas von Niederhäusern**

Working with a customer on this project was a significant milestone for me. It gave me a worthwhile and enjoyable experience while putting everything I had learned at the university into practice. I had the chance to use state of the art technologies, and to improve my knowledge as well as consolidate what i have learned.

Using Firebase as our backend was one of the project's more exciting features. It was the first time I used a cloud backend that exclusively uses cloud functions to handle logic. We also developed a user-friendly and cutting-edge interface for the frontend by utilizing the ideas and knowledge I had learned from the web engineering modules we had studied in prior semesters.

Despite experiencing several challenges near the project's conclusion, we are proud of the outcome. Our prototype will act as a strong starting point for the future application development that our client desires. Participating in app programming was a rewarding opportunity, and I can continue in this field professionally after graduation.

**Joel Suter**

In this project, I tested my skills, which I learned from my experience as a DevOps engineer, and all the newly learned skills during my studies. Furthermore, I could not only put these skills into use but also sharpen them and even learn more, such as using Firebase. I certainly gained much valuable experience in the process.

The project was a success in my and my partner's eyes. We were able to achieve what we set out to achieve and were able to create an excellent prototype for a fantastic idea. I envision that this prototype and the idea behind it could have much potential for further development, and we were able to bring much value to it.

Of course, as in most projects, some things did not go as planned. Areas in which I want to improve in the future are planning. The plan we created was a good guideline, but especially in the end; we deviated quite a bit from it. Sadly, I had a severe injury while working on my bachelor project. For that reason, we had to compromise the project's scope. However, even setbacks such as that are very valuable in their own right. It taught me how to handle such risks and communicate with customers if such risks occur. Overall, I am pleased with the result of our project and proud of what we achieved, and it was an enriching project to participate in.

# 13 Screenshots

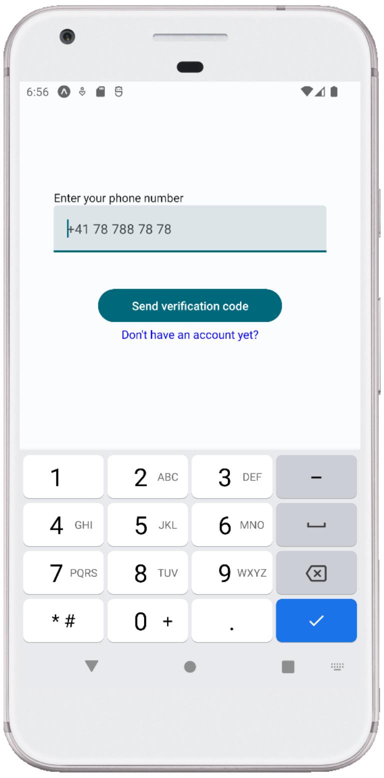


Figure 13.1: Login Screenshot

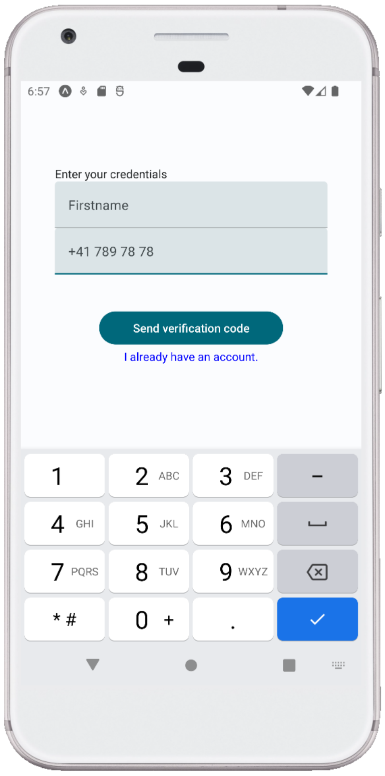


Figure 13.2: Register Screenshot

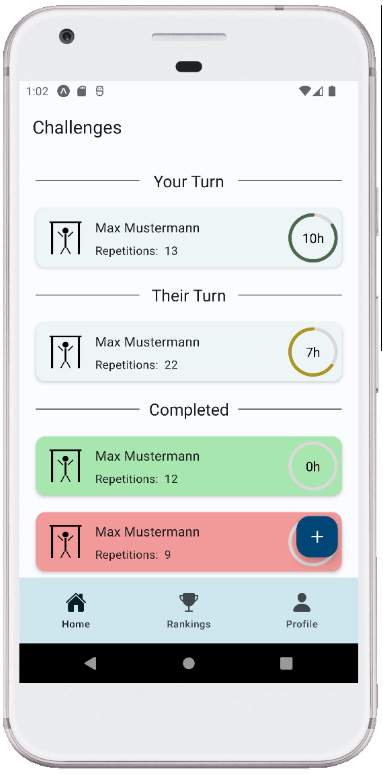


Figure 13.3: Home Screen Screenshot

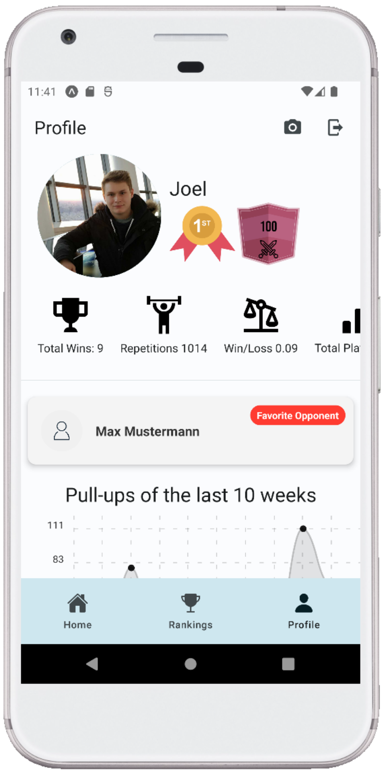
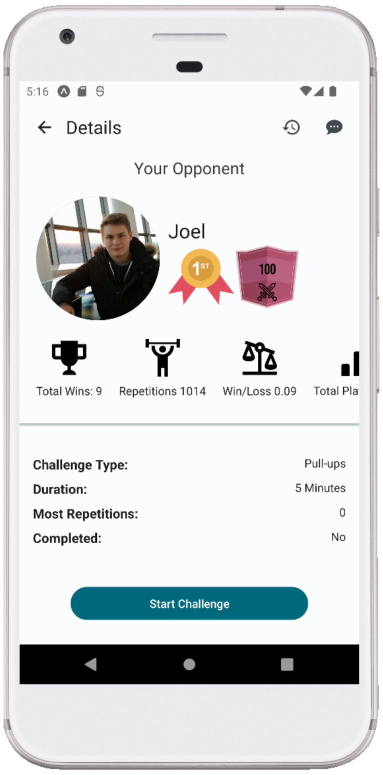
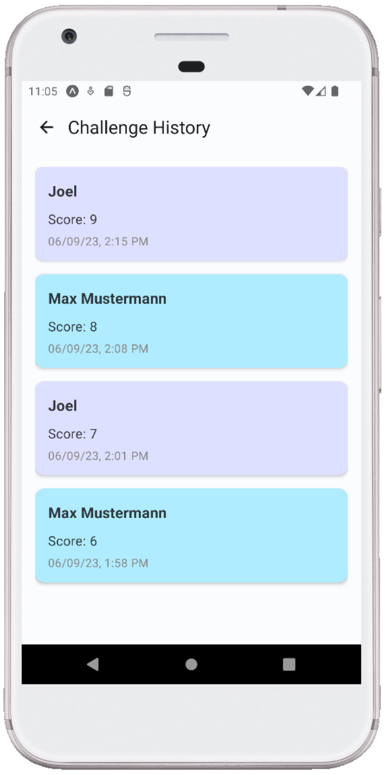


Figure 13.4: User Profile Screenshot



| | |
|---|---|
| Challenge Type: | Pull-ups |
| Duration: | 5 Minutes |
| Most Repetitions: | 0 |
| Completed: | No |

Figure 13.5: Challenge Details Screenshot



Joel
Score: 9
06/09/23, 2:15 PM

Max Mustermann
Score: 8
06/09/23, 2:08 PM

Joel
Score: 7
06/09/23, 2:01 PM

Max Mustermann
Score: 6
06/09/23, 1:58 PM

Figure 13.6: Challenge History Screenshot

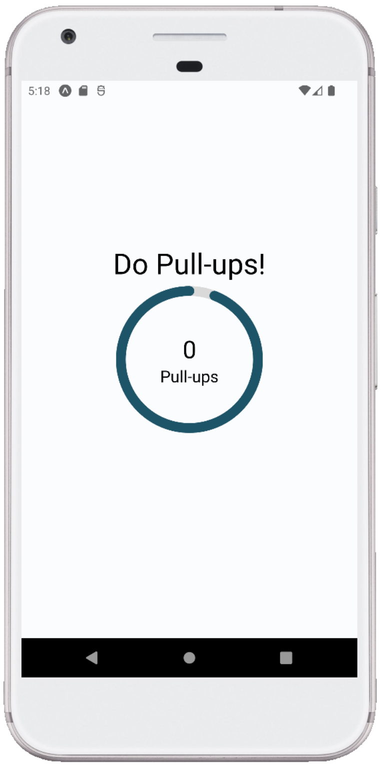
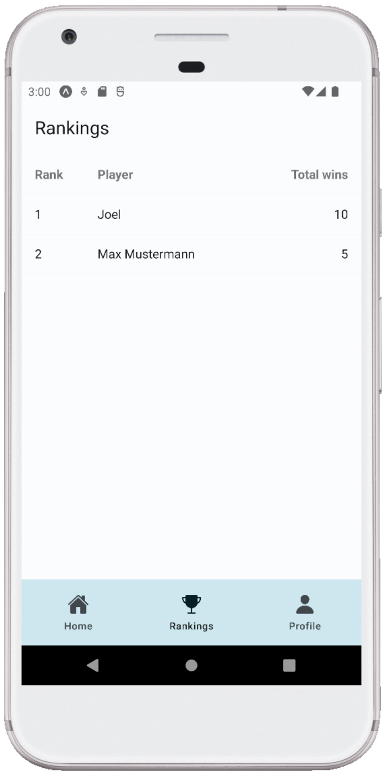


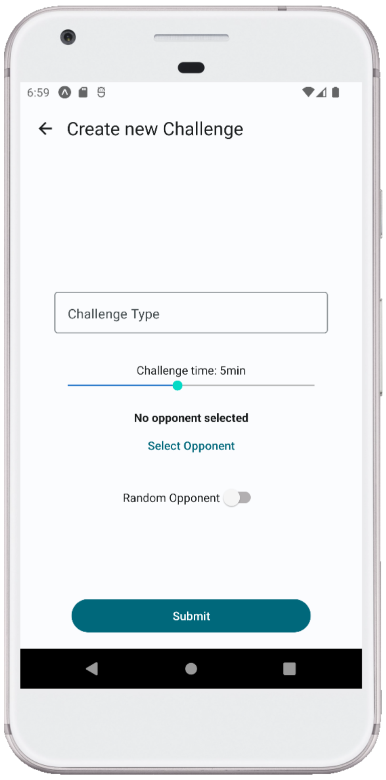
Figure 13.7: Challenge Perform Screenshot


3:00

Rankings

| Rank | Player | Total wins |
|---|---|---|
| 1 | Joel | 10 |
| 2 | Max Mustermann | 5 |

Figure 13.8: Rankings Screenshot


6:59

Create new Challenge

Challenge Type

Challenge time: 5min

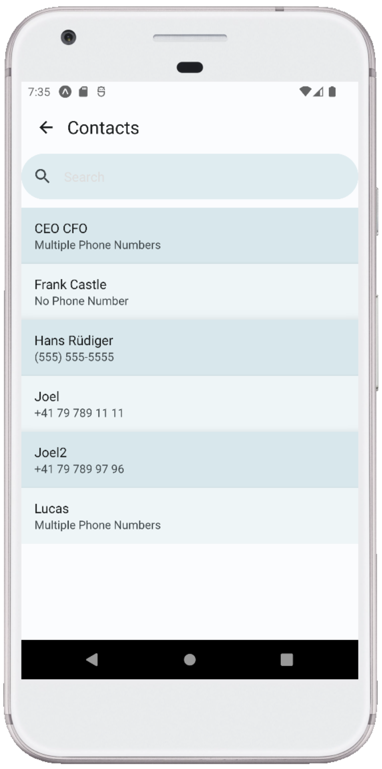
No opponent selected

Select Opponent

Random Opponent

Submit

Figure 13.9: Create Challenge Screenshot


7:35

Contacts

Search

CEO CFO
Multiple Phone Numbers

Frank Castle
No Phone Number

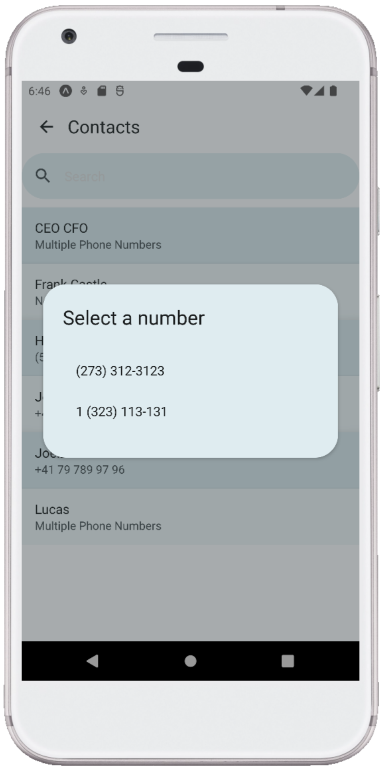
Hans Rüdiger
(555) 555-5555

Joel
+41 79 789 11 11

Joel2
+41 79 789 97 96

Lucas
Multiple Phone Numbers

Figure 13.10: Opponent Selection Screenshot



CEO CFO
Multiple Phone Numbers

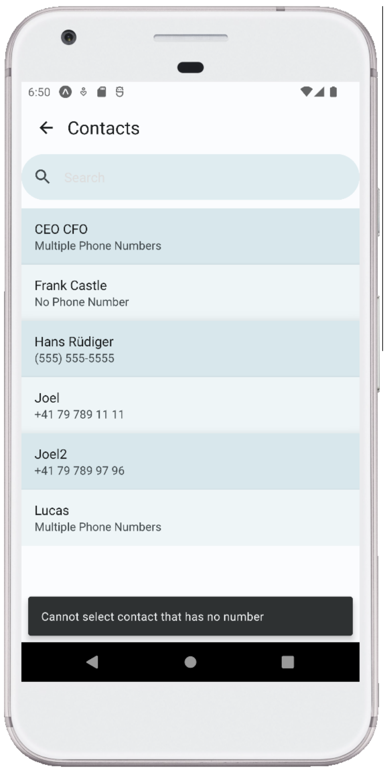
(273) 312-3123

1 (323) 113-131

+41 79 789 97 96

Lucas
Multiple Phone Numbers

Figure 13.11: Select Phone Number Screenshot



CEO CFO
Multiple Phone Numbers

Frank Castle
No Phone Number

Hans Rüdiger
(555) 555-5555

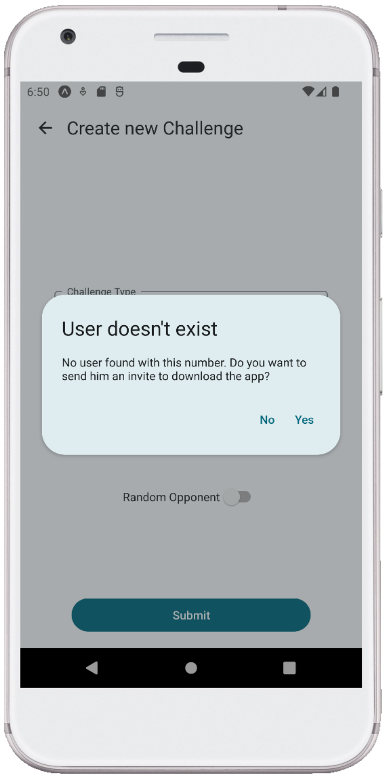
Joel
+41 79 789 11 11

Joel2
+41 79 789 97 96

Lucas
Multiple Phone Numbers

Cannot select contact that has no number

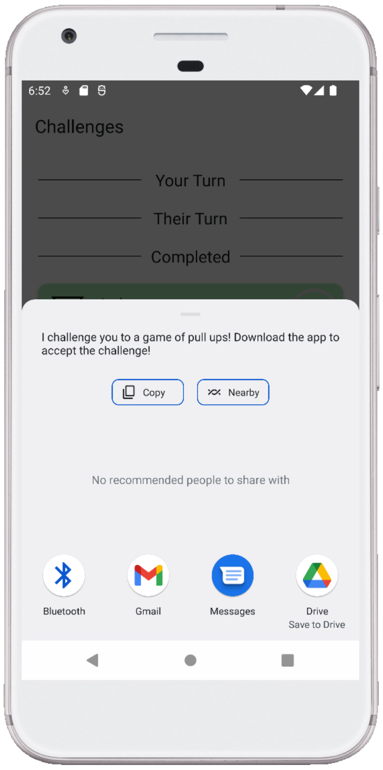
Figure 13.12: No Number in Contact Screenshot



User doesn't exist

No user found with this number. Do you want to send him an invite to download the app?

Figure 13.13: User Does Not Exist Screenshot



I challenge you to a game of pull ups! Download the app to accept the challenge!

No recommended people to share with

Figure 13.14: Share App Screenshot

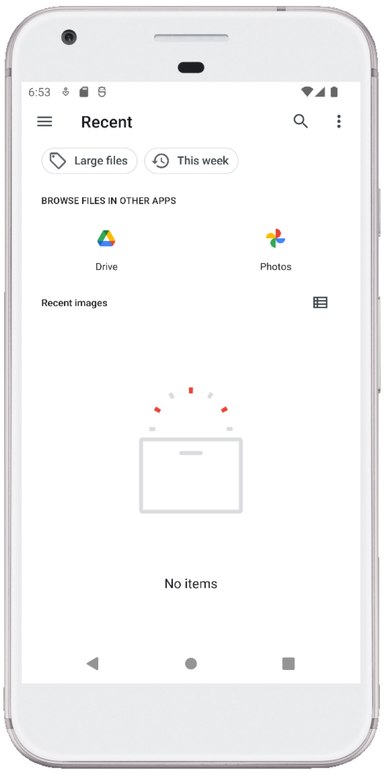
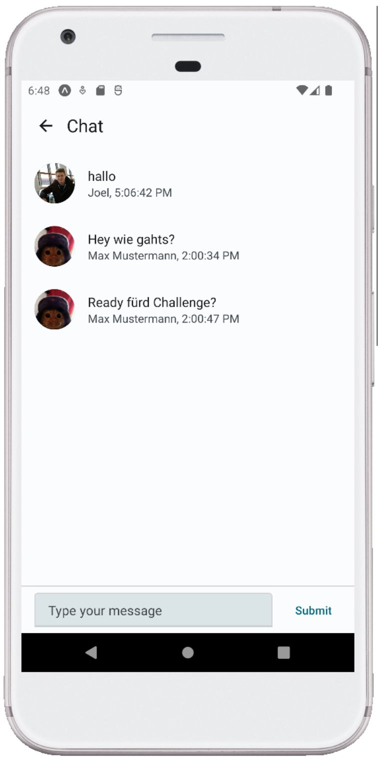


Figure 13.15: Change Profile Picture Screenshot



hallo
Joel, 5:06:42 PM

Hey wie gahts?
Max Mustermann, 2:00:34 PM

Ready fürd Challenge?
Max Mustermann, 2:00:47 PM

Figure 13.16: Chat Screenshot

# 14 Usability Test Protocols

## Usability Test 1 - Android, Dark Mode

### Pre-Interview

- **How old are you?** 25
- **What gender are you?** Male
- **What do you do professionally?** Taxes
- **What is your relationship to sports?** Would like to do more sports, has problems motivating himself.

### Szenario Notes

| Nr. | Notes | Success |
|---|---|---|
| 1 | Tried to login before registering. Thought the login form was for register. He did not fully understand the process and the form. | Partially |
| 2 | He understands how to create a new challenge. | Fully |
| 3 | He did not fully understand how to contact his opponent at first. He didn't know the chat function was inside the challenge details | Partially |
| 4 | Understood when it's his turn and how to perform. | Fully |
| 5 | Understood the banner concept but not the division to challenge details | fully |
| 6 | Easily found | Fully |
| 7 | No issues | Fully |
| 8 | No issues | Fully |
| 9 | Was not able to find the button to open the history of a challenge. Needed help | None |
| 10 | No issues | Fully |
| 11 | No issues | Fully |

Table 14.1: Scenario Notes

### Post-Interview

- **What are your general Impressions?** Cool idea. Would like exercises that can be done in the "alltag".
- **What did you like about our app (generally and design)?** Easy, not too much information, simple, modern, no ads
- **What bothered you (generally and design)?** Dark mode was not optimized. Detection of pullups is not reliable enough, allows for cheating. Better rankings (more rankings). The completed challenges were not in order of their completion.

- **Did you get confused at any point on how to proceed?** Registering, tried to login instead of registering. Challenge details deliniation from banner not clear. The 10 week graph, what the content of it is.
- **Are there specific elements that you were unsure what their purpose was? Tell us which.** The 10 week graph.
- **Would you use the app? Tell us why yes/not.** Would be cool to try out and see if it gives motivation. But needs better tracking first and cheating is possible.
- **Give the app a rating from 0-10. (layout, responsiveness, colour, content)**
  - Layout: 8
  - Responsiveness: 10
  - Colour: 4 (dark mode didn't work everywhere)
  - Content: 6 (because of lack of challenges, and detection)
- **Do you have any general feedback, positive or negative?** Colours need to be improved. Anti cheating. Toutorial would be nice.
- **Do you have any suggestions for additional features? What would you be interested in?** Maybe add tracker support for trackers like fitbit and add more challenges.

## Usability Test 2 - Android, Dark Mode

### Pre-Interview

- **How old are you?** 26
- **What gender are you?** Male
- **What do you do professionally?** Unemployed, student
- **What is your relationship to sports?** Sport enthusiast who does weightlifting regularly

### Scenario Notes

| Nr. | Notes | Success |
|---|---|---|
| 1 | Tried to login first but changed to register screen immediately | Fully |
| 2 | Created challenge very fast, no issues | Fully |
| 3 | No issues | Fully |
| 4 | Waited for his turn and performed the challenge | Fully |
| 5 | Could immediately tell the statistics and their meaning | Fully |
| 6 | Navigated to ranking screen and extracted the information immediately | Fully |
| 7 | User found his profile. Did not understand at first where his badges are | Fully |
| 8 | No issues | Fully |
| 9 | Navigated back to the challenge and opened it, no issues | Fully |
| 10 | Tried to close the app in the background first before logging out | Fully |
| 11 | No issues | Fully |

Table 14.2: Scenario Notes

### Post-Interview

- **What are your general Impressions?** Very impressed. He thinks the whole thing is a great idea
- **What did you like about our app (generally and design)?** Interface and useability are good
- **What bothered you (generally and design)?** Some parts of the app were not readable in dark mode
- **Did you get confused at any point on how to proceed?** No issues
- **Are there specific elements that you were unsure what their purpose was? Tell us which.** What are the meaning of the various badges?
- **Would you use the app? Tell us why yes/not.** He would use the app and challenge his friends
- **Give the app a rating from 0-10. (layout, responsiveness, colour, content)**
  - Layout: 8
  - Responsiveness: 6
  - Colour: 5
  - Content: 7

- **Do you have any general feedback, positive or negative?** He had no feedbacks
- **Do you have any suggestions for additional features? What would you be interested in?** He would like a better detection system for repetitions and more challenge types

## Usability Test 3 - iOS, Light Mode

### Pre-Interview

- **How old are you?** 25
- **What gender are you?** Male
- **What do you do professionally?** Student
- **What is your relationship to sports?** He usually maintains a good balance between sports and life but lacks the motivation lately

### Scenario Notes

| Nr. | Notes | Success |
|---|---|---|
| 1 | Found the register screen immediately and created an account | Fully |
| 2 | Found challenge creation, gave access to his contacts. Had to tap on the submit button multiple times as it was not responsive. | Fully |
| 3 | Found the chat screen. Had issues regarding keyboard covering the input field. Successfully sent the message. | Fully |
| 4 | User waited for his turn in the challenge and performed it accordingly. | Fully |
| 5 | Found all the information we asked. Had trouble recognizing lifetime and challenge score. | Fully |
| 6 | Was able to find the ranking screen and rank leader immediately | Fully |
| 7 | Instantly changed to his profile screen and found the relevant information of the scenario. | Fully |
| 8 | Found the profile picture picker but after selecting the photo the application crashed (2x) | None |
| 9 | Found the challenge history screen of the previous game immediately. | Fully |
| 10 | No issues | Fully |
| 11 | No issues | Fully |

Table 14.3: Scenario Notes

### Post-Interview

- **What are your general Impressions?** Solid app with clear overview
- **What did you like about our app (generally and design)?** No information cluttering, clean interface
- **What bothered you (generally and design)?** Button not responsive, crashing of the application and keyboard covers his text input field
- **Did you get confused at any point on how to proceed?** No issues
- **Are there specific elements that you were unsure what their purpose was? Tell us which.** Nothing he was unsure about
- **Would you use the app? Tell us why yes/not.** Yes, he would like to use the app, might help staying motivated to do sports
- **Give the app a rating from 0-10. (layout, responsiveness, colour, content)**

  - Layout: 10
  - Responsiveness: 8
  - Colour: 10
  - Content: 7
- **Do you have any general feedback, positive or negative?** He likes the idea of the application
- **Do you have any suggestions for additional features? What would you be interested in?** He would like a push up challenge as it is easier to do from home, tournament modes and maybe make something like the App "BeReal" where users get prompted to do something once a day

## Usability Test 4 - IOS, Light Mode

### Pre-Interview

- **How old are you?** 25
- **What gender are you?** Male
- **What do you do professionally?** Facility Managment
- **What is your relationship to sports?** Not actively doing sports.

### Szenario Notes

| Nr. | Notes | Success |
|---|---|---|
| 1 | Understood how to create an account eventhough at first he did not add the prefix | Fully |
| 2 | Knows how to create a challenge and instantly understood the form. | Fully |
| 3 | No issues expect that ios does not display input while inputting. | Fully |
| 4 | Understood when it's his turn. Understands how to perform | Fully |
| 5 | No issues | Fully |
| 6 | Found the rankings instantly | Fully |
| 7 | No issues | Fully |
| 8 | At first did not understand the image picker (which is not under our control) | Partially |
| 9 | Found the history, but clicked around first in the rest of the app. Took a few seconds | Fully |
| 10 | No issues | Partially |
| 11 | Logged in with no issues. Added prefix instantly. | Fully |

Table 14.4: Scenario Notes

### Post-Interview

- **What are your general Impressions?** Very good. Simple. Unfortunately too easy to cheat.
- **What did you like about our app (generally and design)?** Home Screen could be styled a little nicer, maybe a nice background.
- **What bothered you (generally and design)?** Nothing "bothered" really.
- **Did you get confused at any point on how to proceed?** Not really.
- **Are there specific elements that you were unsure what their purpose was? Tell us which.** Not really. Everything speaks for itself. Descriptions for badges or a toutorial would be nice though.
- **Would you use the app? Tell us why yes/not.** If i would be more interested in sports maybe. But it could be a nice motivation being able to play against friends, but it needs better recognition.
- **Give the app a rating from 0-10. (layout, responsiveness, colour, content)**
  - Layout: 9

  - Responsiveness: 10
  - Colour: 8
  - Content: 6.5
- **Do you have any general feedback, positive or negative?** Emtpy Home Screen makes a bad impression, there is no content there. Better styling.
- **Do you have any suggestions for additional features? What would you be interested in?** Descriptions for elements like badges, toutorial, make everything clickable (most wins to see history of wins, etc). Able to click profile or view banner of a opponent in the rankings.

# 15 Task

## Fitness Gamification

### Beteiligte Personen

- Diplomanden: Lucas von Niederhäusern und Joel Suter
- Industriepartner: AdaptIT GmbH, Michael Güntensperger
- Experte: Prof. Hansjörg Huser
- Betreuer: Prof. Frank Koch

### Problembeschrieb

Fitness-Tracker-Daten werden im Bereich Personal Training immer wichtiger. I.d.R. haben Trainer nur die Möglichkeit, anhand separater Plattformen der Fitness-Tracker-Hersteller die Bewegung der Kunden zu tracken und auszuwerten. Dies ist aufwendig, wenn ein Trainer mehrere Kunden mit unterschiedlichen Tra- ckern betreut (z.B. Samsung, Google, Apple, Fitbit, Garmin, ...). Aus diesem Grunde haben 2 Studierende der OST im HS22 eine "Fitness Data Platform" zum Anbinden von Fitness-Trackern und der Harmonisie- rung der multiplen Datenquellen entwickelt. Das Tool beinhaltet die Anbindung an Fitbit und Garmin, die Erstellung eines ersten Dashboards mit Auswertungen zu der Schrittzahl wie auch die Möglichkeit, Dash- boards anderen Benutzer: innen freizugeben. Angestrebt wird eine App, die Sportler mit einem Gamification-Ansatz zusätzlich motiviert, sich gegenseitig weiter zu pushen. Ein Spieler erstellt ein Spiel (Auswahl von Übung, Dauer, Challenger), sobald er dieses startet wird er auf- gefordert, den ausgewählten Übungstyp so viele Male wie möglich in der ausgewählten Dauer durchzufüh- ren. Die App zeigt während dieser Zeit die verbleibende Zeit wie auch die Anzahl Ausführungen. Sobald die Zeit abgelaufen ist, hat der Spieler die Möglichkeit einen Text / GIF zu ergänzen und das Spiel zu senden. Der Empfänger hat einen Durchgang und eine limitierte Zeit das Ergebnis zu toppen. Schafft er dies nicht, wird er zum Verlierer. Auf einem Dashboard wird angezeigt, wie oft ein Spieler gewonnen / verloren hat, wie viele Durchführun- gen einer Übung er im Total erreicht hat (aktuelle Woche und persönlicher Rekord) und eine Übersicht mit welchen Kollegen er am Meisten Spiele gemacht hat.

### Aufgabenstellung

Mit der BA soll die Plattform um verschiedene Gamification-Ansätze erweitert werden. Dazu müssen sich die Träger der Fitness-Tracker zu Gruppen verbinden und miteinander kommunizieren bzw ihre Leistungen miteinander vergleichen können. Folgende Challenges/Games sind angedacht.

#### Technische Umgebung

Für die Umsetzung wird mit Web-Technologien gearbeitet.

- Frontend: Flutter / React Native / ionic
- Backend: Node.js
- Datenbank: Firebase
- Hoster: DigitalOcean (und Hostpoint)

**Funktionale Anforderungen**

- Registration eines neuen Accounts mithilfe der Handynummer
- Login eines bestehenden Users
- Erstellen von Challenges
- Mindestens eine Challenge-Art mithilfe der Handysensoren tracken können
- Zugriff auf Adressbuch des Users
- Challenges können an eine Person gesendet werden
- Gegenseitiges hin und her senden von Challenges inklusive Textnachricht und GIF
- Screen mit einer Übersicht aller laufenden Challenges inklusive Zusatzinformationen wie die Anzahl Wiederholungen.
- Hat ein User die App nicht, kann der Challenge-Link trotzdem gesendet werden, der User wird dann aufgefordert, die App herunterzuladen
- Erstellen eines Dashboards mit den spannendsten Informationen für den Benutzer, z.B. Anzeige der gewonnenen Badges.
- Versand von Push-Nachrichten bei Aufforderung zur Teilnahme an einer Challenge
- Bereitstellung eines Chat für den Austausch von Teilnehmenden.
- User kann zufälligen Spieler herausfordern.
- Erstellen von Rankinglisten auf Ebene Freunde sowie Global.

**Optionale Anforderungen**

- Userregistration mit / Google / Facebook / Apple-Account
- Löschen des Accounts und aller zugehörigen Daten
- Weitere Challenge-Arten implementieren
- Timeline aller Challenges eines Users erstellen
- Eine Challenge soll mit einer Videobotschaft versendet werden können
- Erfassung von eigenen Challenges (z.B. Anzahl Treppenstufen)
- Massnahmen gegen Spielbetrug

**Nicht-Funktionale Anforderungen**

- Das App soll auf Android und iOS laufen
- Pushnachricht werden laufend versendet
- Errors sollen keine Systemfehler erzeugen, aber eine Error Nachricht Zeigen und das System auf den vorherigen Zustand zurücksetzen.
- Jeder Error soll im System geloggt werden
- Daten welche in Eingabefelder abgefüllt werden, sollen zuerst validiert werden, bevor diese durch das System verarbeitet werden. SQL Injection test der Eingabefelder sollte keine Verletzlichkeiten zeigen.
- User-Passwörter werden nicht in plain-text in der Datenbank gespeichert.
- Wenn sich ein User in die App einloggt, werden ihm auch nur seine Daten / auf Daten die er Zugriff haben soll, angezeigt.
- Businesslogik im Backend soll modular aufgebaut werden, so dass sie erweitert werden kann.
- Die Backend-API soll durch API-testing Tools getestet werden.
- Implementierte Funktionalität (Datenbank, Backend) sollen deployed und die App bei Google und Apple im App-Store veröffentlicht werden
- Das Laden des Inhaltes der App soll nicht länger als 150ms dauern
- Drei von vier Test-Usern sollen das UI (Kategorien: layout, responsiveness, colour, content) der Ap- plikation mit iPhone / Android Handy mit einer Note von mindestens 8 von 10 bewerten, wobei 10 das Beste ist.
- Das Backend sollte 50 Requests pro Sekunde handeln können
- Die Datenbank soll bis zu 10'000 Challenges und 2000 Benutzer managen können.

## Zur Durchführung

Mit dem Betreuer finden Besprechungen gemäss Absprache statt. Die Besprechungen sind von den Stu- dierenden mit einer Traktandenliste vorzubereiten und die Ergebnisse in einem Protokoll zu dokumentie- ren, das dem Betreuer per E-Mail zugestellt wird. Für die Durchführung der Arbeit ist ein Projektplan zu erstellen. Dabei ist auf einen kontinuierlichen und sichtbaren Arbeitsfortschritt zu achten. An Meilensteinen gemäss Projektplan sind einzelne Arbeitsresultate in vorläufigen Versionen abzugeben.

## Dokumentation und Abgabe

Siehe Leitfaden Abschnitt 5.5 "Umfang und Form der Abgabe".

## Termine

Siehe veröffentlichte «Termine BA FS23».

## Bewertung

Siehe «Leitfaden für Bachelor- und Studienarbeiten» Abschnitt 6 "Bewertung", insbesondere Kap 6.5.