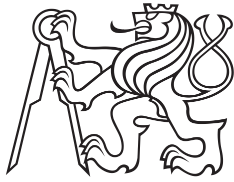
Master's Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Developing Sport Activity Processing Application with Motivational Features

Bc. Tomáš Zahálka

Supervisor: Ing. Zdeněk Rybola, Ph.D.
Field of study: Open Informatics
Subfield: Software Engineering
May 2019

ZADÁNÍ DIPLOMOVÉ PRÁCE

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Zahálka** Jméno: **Tomáš** Osobní číslo: **420918**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Studijní obor: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Developing Sport Activity Processing Application with Motivational Features

Název diplomové práce anglicky:

Developing Sport Activity Processing Application with Motivational Features

Pokyny pro vypracování:

The aim of this thesis is to create a system for synchronizing and analyzing data from an external sport activity tracking app. The system will consist of a set of services hosted on a server and an Android client application.

1. Specify features of the system, focusing on achievements in different sports and activities
2. Review existing solutions and technologies
3. Design the mobile app and define the requirements for the backend interface
4. Design the backend part of the system
5. Implement the whole system based on the design
7. Document the implementation of the system
8. Test the implemented system appropriately, including UI, API and performance

Seznam doporučené literatury:

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Zdeněk Rybola, Ph.D., katedra softwarového inženýrství FIT

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **07.02.2019** Termín odevzdání diplomové práce: **24.05.2019**

Platnost zadání diplomové práce: **20.09.2020**

Ing. Zdeněk Rybola, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.

Datum převzetí zadání	Podpis studenta

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Acknowledgements

I would like to thank my supervisor Ing. Zdeněk Rybola, Ph.D. for a great deal of help during the process of writing this thesis. His approach, deep knowledge of the subject and experience were priceless.

Special thanks goes to my family and friends for their encouragement and for supporting me throughout my studies.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on May 22, 2019

Abstract

This master's thesis deals with designing and creating an application for synchronizing and processing sport activity data that motivates users to lead an active life. The thesis analyzes the current state of the art and proposes a new solution.

The final product includes a client for Android and a server part implemented in the Spring Boot framework. The application contains a concept of achievements that are unlocked by the users based on their activity recordings. These recordings are synchronized from the fitness-tracking application Strava and stored on the server. The app implements a system that allows to easily define achievements with various conditions required for unlocking. The Android app uses modern technologies and supports data caching. The server part is scalable as it consists of multiple services, which can be deployed independently.

Keywords: sport, activity, motivation, processing, Android, Spring, Strava, microservices, synchronization

Supervisor: Ing. Zdeněk Rybola, Ph.D.
Department of Software Engineering,
Faculty of Information Technology,
Czech Technical University in Prague,
Thákurova 9,
Prague 6

Abstrakt

Tato diplomová práce se zabývá návrhem a vytvořením aplikace pro synchronizaci a zpracování dat ze sportovních aktivit, která motivuje uživatele ke zdravému životnímu stylu. Práce analyzuje současný stav a navrhuje nové řešení.

Finální produkt obsahuje klientskou aplikaci pro Android a serverovou část implementovanou ve frameworku Spring Boot. Aplikace obsahuje koncept ocenění, které se dají uživateli odemknout na základě záznamů jejich aktivit. Tyto záznamy jsou synchronizovány z fitness aplikace Strava a ukládány na serveru. Aplikace implementuje systém, který umožňuje snadno definovat ocenění s různými podmínkami pro jejich odemčení. Android aplikace používá moderní technologie a podporuje cachování dat. Serverová část je škálovatelná díky tomu, že se skládá z více služeb, které lze nezávisle nasazovat.

Klíčová slova: sport, aktivita, motivace, zpracování, Android, Spring, Strava, mikroslužby, synchronizace

Překlad názvu: Vývoj aplikace pro zpracování sportovních aktivit s motivačními prvky

Contents

1 Introduction 1
1.1 Motivation . . . 1
1.2 Application Description . . . 1
1.3 Goal of the Thesis . . . 2
2 Analysis 3
2.1 Existing Applications . . . 3
2.1.1 Strava . . . 3
2.1.2 Strava Apps . . . 6
2.1.3 Garmin Connect . . . 9
2.1.4 Summary . . . 10
2.2 Requirements . . . 11
2.2.1 Functional Requirements . . . 11
2.2.2 Non-functional Requirements 14
2.3 Domain Model . . . 15
2.4 Use Cases . . . 18
3 Design 23
3.1 Overview . . . 23
3.2 Android App . . . 23
3.2.1 User Interface . . . 25
3.2.2 Authentication . . . 29
3.2.3 API . . . 29
3.2.4 Entities . . . 31
3.3 Backend . . . 31
3.3.1 Service Architecture . . . 32
3.3.2 Services . . . 33
3.3.3 External Systems . . . 35
3.3.4 Entities . . . 36
3.3.5 Authentication . . . 39
4 Implementation 41
4.1 Android App . . . 41
4.1.1 Structure . . . 42
4.1.2 Libraries . . . 42
4.1.3 User Interface . . . 42
4.1.4 ViewModel . . . 43
4.1.5 Models . . . 43
4.1.6 Local Storage . . . 43
4.1.7 Caching . . . 44
4.1.8 Authentication . . . 44
4.2 Backend . . . 44
4.2.1 Structure . . . 45
4.2.2 Libraries . . . 45
4.2.3 Authentication . . . 46
4.2.4 Activity Synchronization . . . 47
4.2.5 Achievements . . . 47
4.2.6 Communication between Services . . . 48

4.2.7 Deployment 49

5 Testing **51**

5.1 Android App Testing 51

5.1.1 End-to-end Tests 51

5.2 Backend Testing 53

5.2.1 API Tests 53

5.2.2 Unlocking Achievements 53

5.2.3 Integration Tests 54

5.3 Manual Testing 54

5.4 Performance 55

6 Conclusion **57**

6.1 Implemented Application 58

6.2 Future Work 59

Bibliography **61**

A Acronyms **67**

B Contents of enclosed SD Card **69**

C User Manual **71**

Figures

2.1 Strava Challenges—screenshot [1]. 5

2.2 Strava Matched Runs feature—performance trend [2]. 6

2.3 VeloViewer—Your Summary screenshot [3]. 8

2.4 Garmin Connect—earned Badges screenshot [4]. 10

2.5 Domain model. 16

2.6 Actors in the system. 18

2.7 Mobile application use cases. . . . 19

2.8 Backend use cases. 20

3.1 System overview. 24

3.2 Mobile app architecture. 25

3.3 Data loading decision tree [5]. . . 26

3.4 Dashboard screen layout. 27

3.5 Achievements screen layout. 28

3.6 3-layer architecture of a backend service. 33

3.7 Backend services. 34

3.8 Sequence diagram—activity synchronization. 37

3.9 Database model—Authentication service. 38

3.10 Database model—Activity service. 38

4.1 Backend project structure. 46

4.2 Production deployment. 50

5.1 Latency for different number of service instances. 56

6.1 Mobile application—screenshot. 58

Tables

4.1 Libraries in the Android project. 43

4.2 *Spring Cloud* libraries. 46

5.1 Performance testing results. 56

Chapter 1

Introduction

The first part explores the main drivers that stimulated the creation of this thesis, summarizes the functionality of the application and states the goal of the thesis.

1.1 Motivation

For today's athlete, there any numerous options for tracking their performance, including various apps and devices. However, in my opinion, there are still ways to go in terms of increasing the motivation for a regular user to stay being active. There are definitely many great apps with features that do incentivize active lifestyle. Despite this fact, there is a large room for new ideas. The one area in this space that I found not fully developed is the concept of giving athletes awards for their performance. As I mentioned, recording the performance data is done well by various means, so we do have a vast amount of data available to further analyze. That is why I focused only on analyzing the data, and thought about a way it could be used to reward users for their healthy lifestyle.

1.2 Application Description

This section describes the expected functionality and the main features of the app from the user's point of view. The name chosen for the app is **Sportivator**, which is also used throughout the thesis.

The main goal of this app is to provide users of sport activity tracking

apps with some insights into their data and in turn to give them incentive to be active and stay fit. For this thesis, Strava was chosen as the external tracking app because of two main reasons:

1. It offers a great Application Programming Interface (API) [6].
2. It has a large active user base.

Strava even host a curated list of apps from developers that utilize the API [7], which makes them more easily discoverable.

Users are rewarded for their activity in the form of achievements. Achievement is a key concept for Sportivator and represents some form of a milestone a user can reach. Milestones might take different forms, examples include being active some predefined amount of time, or achieving some minimum average speed for activities in a day. These achievements are highlighted for different time intervals: week, month, year and all-time. The goal for the user is to unlock all achievements in the long term, but they might attempt to unlock the easier ones regularly. The regularity of the process is important as it can help create a habit for the user to unlock some achievements all the time and serve as a warning if they are not able to do so.

Sportivator runs on Android mobile devices, and is easy to use as it provides a simple user interface. Users log in to the app using their Strava account. They can ask the app to synchronize their Strava activities, and once it is completed, they are presented with the unlocked achievements.

1.3 Goal of the Thesis

The goal of the thesis is to create an application that satisfies the functionality stated in the previous section. The process consists of several steps.

The first step is to look into solutions with similar functionality on the market, then define the requirements and features of the app. This is followed by designing the structure and components of the system while using best practices that apply to this domain. Based on the design, specific technologies are selected and the application is implemented. The final step is testing the functionality and measuring the performance of the app.

Chapter 2

Analysis

The Analysis chapter looks at several existing applications with functionality similar to the one proposed and implemented as part of this thesis, then lists all the requirements for the app.

2.1 Existing Applications

I started my analysis as long-time user of a few sport-tracking apps. There were features that I missed and had to look for elsewhere. Here, some more detailed descriptions of the apps and features that I found, are provided. Lastly, based on the analyzed apps, I state the criteria for implementing Sportivator.

2.1.1 Strava

It has already been mentioned that Strava is the data provider of choice. In this regard, it is certainly a great option at the time of writing this thesis.

The website says: *"The #1 app for runners and cyclists."*[8]

So, the main focus of Strava is on these two segments of sports, although other sports are available. The primary use of the app is for tracking activities with location, health and other data. The most interesting data for my analysis are distance, moving time and elevation gain. These data are presented in the detail of each activity recording. There are also charts for additional analysis of athlete's performance.

Strava is a huge network of athletes. Everyone can connect with their friends and see their activities. They can give those activities *kudos* and comment on them.

In addition to that, Strava already contains some functionality with motivational features.

Segments

"Your GPS-based activities can be broken down into smaller sections called segments. Segments designate specific features or portions of a route – such as a climb, a tricky stretch of trail or part of an open-water swim. Each time you complete a segment, your time is recorded so you can compare it to previous efforts and those of your friends and other athletes."[9]

This feature has a big social aspect as the user can compare to others in leaderboards. There is a concept of achievements included in Segments too. *"When you upload an activity to Strava, you may be awarded 'achievements' for outstanding performances on the segments you traverse."*[9]

These achievements are:

Crown King/Queen of the Mountain (KOM/QOM) or Course Record (CR).

Trophy Overall Top 10 Placement.

Medal Personal Top 3 Placement.

Strava Challenges

"Strava offers several different types of challenges as a fun way to motivate people to push themselves and accomplish a specific goal. Challenges can last a day, several days or even an entire month. They can be a single activity or cumulative. We currently host running and cycling challenges that are typically centered around hitting a certain distance or elevation. Sometimes we pair with partners to bring you challenges that offer a prize for completing."[10]

In summary, this feature is a way to reward the user for a sustained effort over a stretch of time. The user needs to join a Challenge manually in order to be considered for its completion. Figure 2.1 depicts a list of Challenges as it is available in the Strava mobile app.

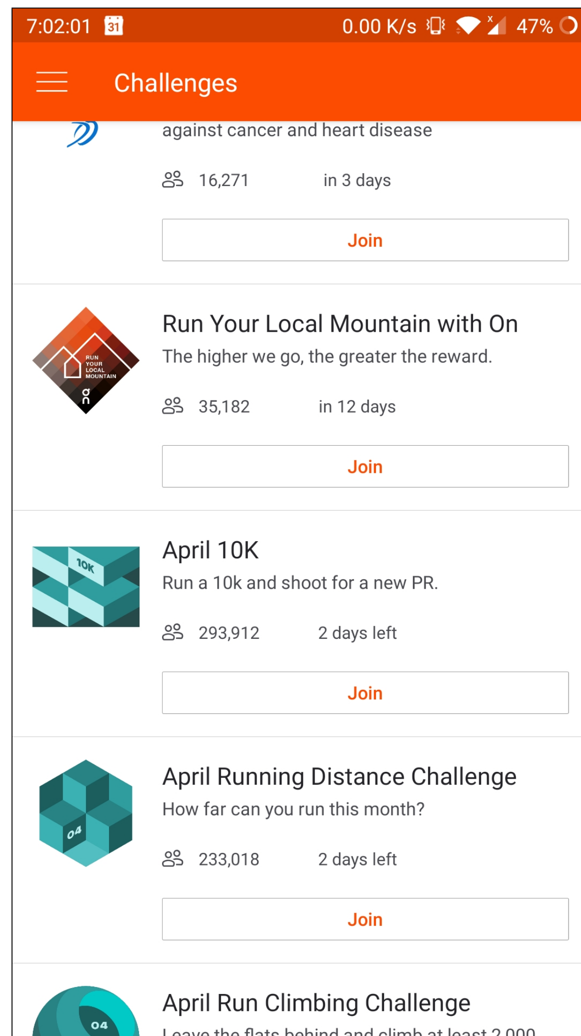


Figure 2.1: Strava Challenges—screenshot [1].

Matched Runs

As the name implies, this feature works only for workouts with running as the sport type, and it is a great way for the user to look into their own performance on their favorite route.

"Matched Runs uses an algorithm to automatically identify when you have completed runs on a route that has been run before. Run the same route multiple times and Strava will group all the efforts together in a single chart to show you a performance trend over time. In order to match similar runs, the algorithm identifies the starting and ending point of the route, the direction in which the route was run, and the distance completed."[11]

An example of a performance trend as presented in the Strava mobile app

is shown in figure 2.2.

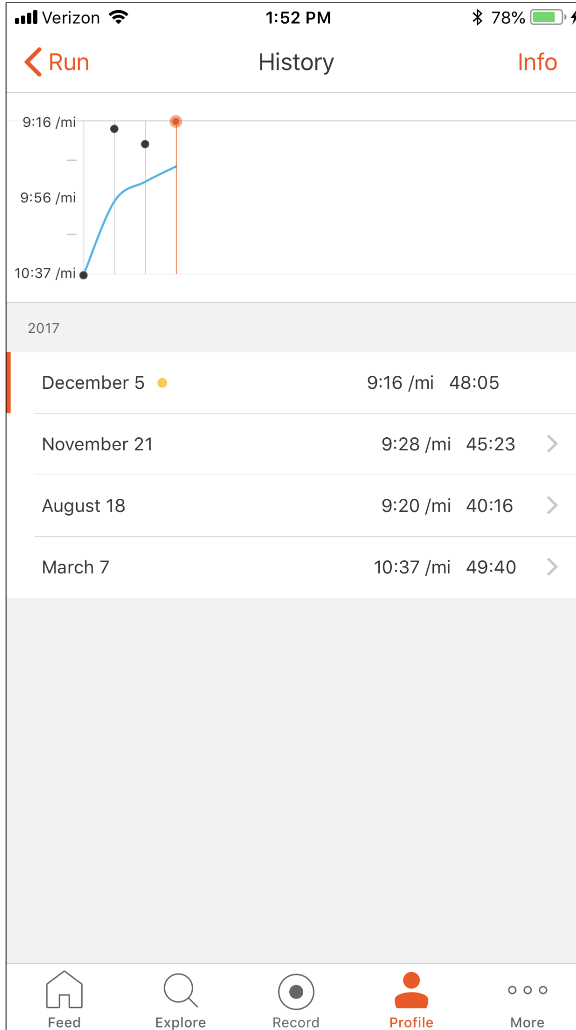


Figure 2.2: Strava Matched Runs feature—performance trend [2].

2.1.2 Strava Apps

The Strava Apps section on the Strava website is a database of many useful third-party apps organized into several categories [7]. For my analysis, the most notable categories were *Performance Analysis*, *Training* and *Social Motivation*. All these apps use Strava as their data source, but some of them also support other fitness services.

VeloViewer

This is a very complex app with many features that is used even by professional cycling teams. As such, it is focused more on detailed analysis and exact numbers rather than "attractive" awards.

"VeloViewer provides new insights, engaging visualizations, motivational goals and in-depth analysis to your Strava data."[12]

Nevertheless, it does contain three forms of *Awards* for different activity parameters (distance, elevation, time). Each *Award* has a number of stars that indicate how many times (number of activities) the user met a certain condition for up to three stars. For example, the user might have gained 500 m in elevation in an activity, this gives them one star for the 500 m *Elevation Reward*.

The app can generate an *Infographic* for every year that summarizes various statistics in a single image. It also provides other visualizations of the data. The app is not available in a mobile version but offers a web interface. Example of the *Summary* page is shown in figure 2.3.

rubiTrack

Another service targeted at detailed analysis of workouts is called rubiTrack [13].

It boasts with a very appealing user interface. There are detailed charts and lists with various data from the recordings. An interesting feature is the ability to compare activity data (like distance and pace) between different time periods. *"It quickly answers questions like this: 'What was my total distance in the last 5 weeks, and how much more was it compared to the previous 5 weeks?.' "*[14]

To mark milestones in their training, users can add events to their calendar. The calendar can display activities and their data with multiple levels of detail.

Activities can be downloaded not just from Strava, but also from other devices and services. Its availability is limited to macOS and iOS.

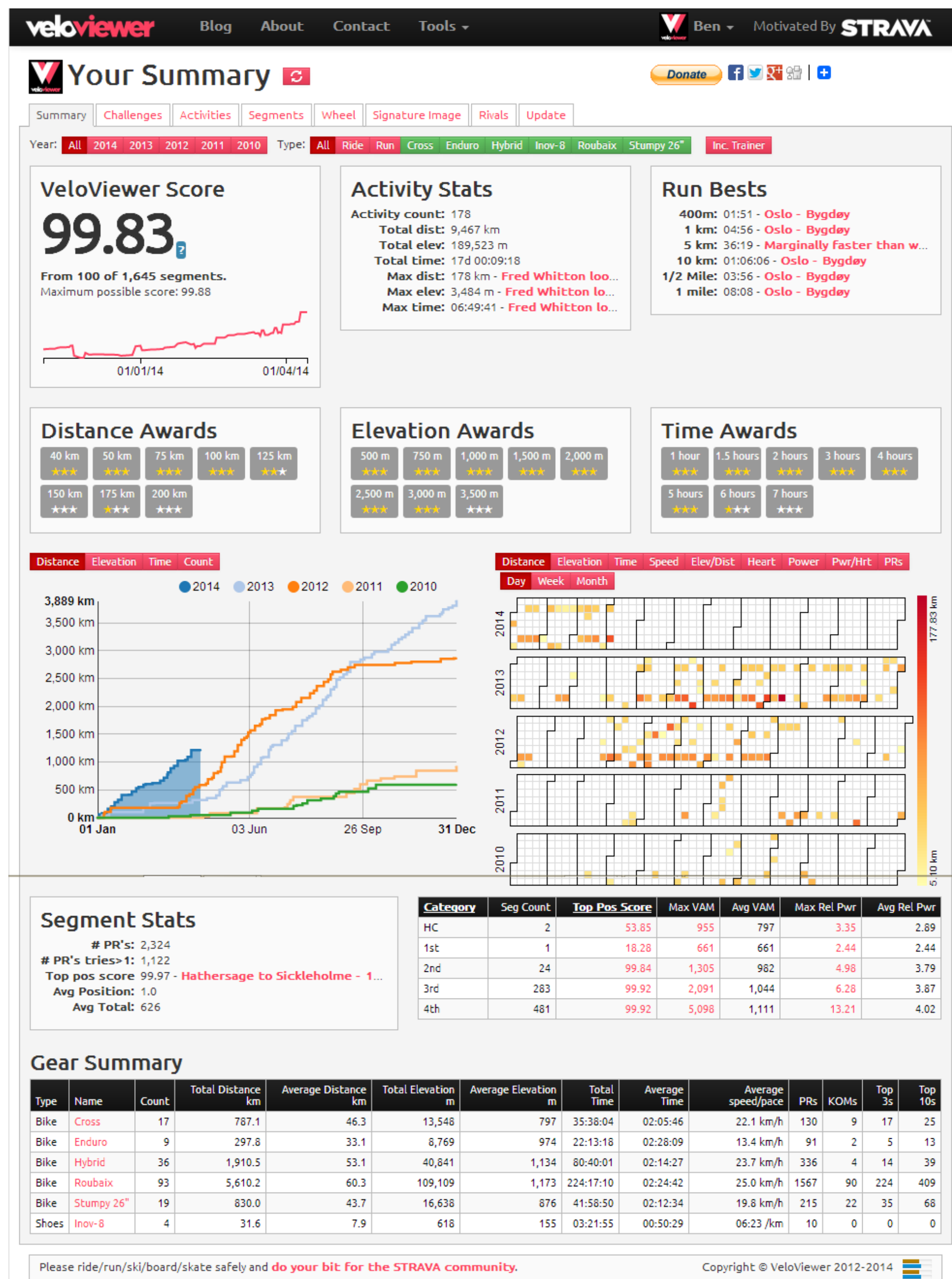


Figure 2.3: VeloViewer—Your Summary screenshot [3].

WeFitter

WeFitter is a bit special as it is not directly targeted at end users but rather businesses for customized implementation. The website says: *"Turn your business into an engaged and healthy community."*[15]

Their system takes data from more that 20 fitness applications and makes

them available under a single API. The idea is to lower healthcare costs for the company *"by encouraging employees to lead a healthy lifestyle."*[16]

Employees get customizable rewards through a gamification system to drive engagement. There are also leaderboards in order to allow the employees to compare themselves with others, and to increase their motivation. Multiple well-known companies are listed between clients that implemented WeFitter into their company [17].

A similar program targeted at cities exists as well. A dedicated website shows a leaderboard with cities that have joined the program along with some statistics. [18]

2.1.3 Garmin Connect

Garmin Connect is the official app for users with Garmin devices. It is available as a mobile app or through a web interface. Both versions contain customizable *Dashboards* with different data (e.g. steps, calories). With a suitable device, it can track activities, or the user can import activities manually.

"On mobile or web, Garmin Connect is the tool for tracking, analyzing and sharing health and fitness activities from your Garmin device."[19]

The most notable feature for my analysis are *Badges*.

"Garmin Connect offers a variety of different badges that you can earn on a daily bases from getting a good night's sleep (Well Rested badge), to running that 100 mile race that you knew would be a piece of cake (Insanity badge)."[20]

Badges can be filtered by several categories. The user can display only earned *Badges* as well as all available *Badges*. Each one has the following attributes:

- image,
- name,
- description,
- number of points.

For each *Badge*, related ones are shown as well. If a *Badge* was previously unlocked, the user can see the date of when that occurred. Each user also has

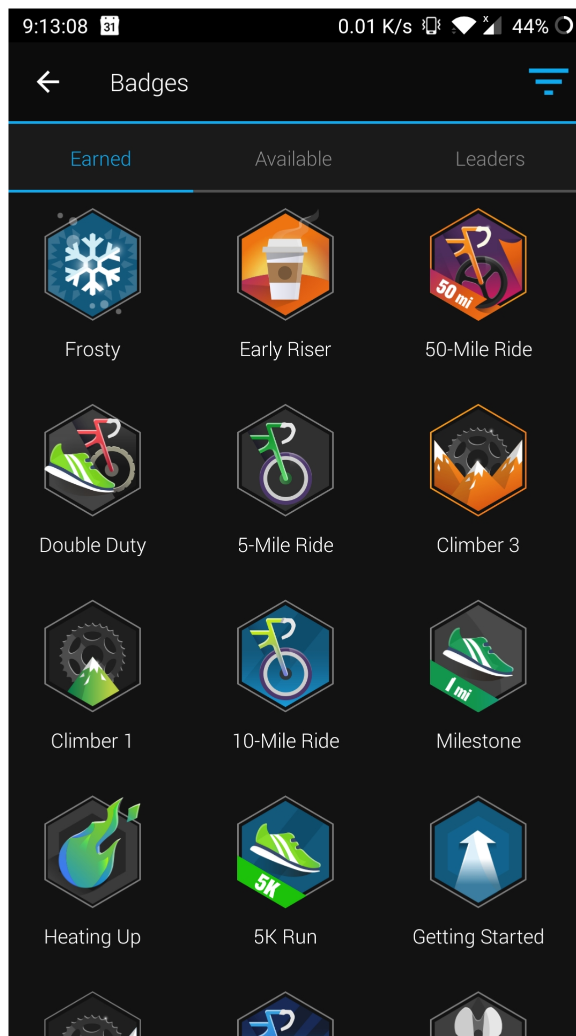


Figure 2.4: Garmin Connect—earned Badges screenshot [4].

a *Level*, which is determined by the sum of all points from earned *Badges*. The *Badge* images are colorful with various effects, and the overall presentation is very attractive. Figure 2.4 shows the screen with earned *Badges*.

2.1.4 Summary

The analysis of the market gave me an inspiration for my own app and showed missing areas that can be addressed. *Badges* in the Garmin Connect app represent a functionality that most closely resembles the functionality of the app proposed in this thesis. However, the main limitation for me is the fact that a *Badge* cannot be earned multiple times for different time periods. Another limitation is that a *Badge* cannot combine multiple conditions (e.g. consider both distance and elevation).

From all of the above, following are the basic criteria I laid down for implementing the app.

1. Provide a clear user interface.
2. Focus on periodic achievements.
3. Consider multiple activity attributes for a single achievement.

2.2 Requirements

Following are the requirements divided into functional and non-functional categories.

2.2.1 Functional Requirements

These are the requirements describing the functionality the application must provide.

FR-01 Strava authentication

The user can authenticate with Sportivator by using their Strava account. The system stores user's tokens for accessing the Strava API. The user authorizes access to their data through a prompt either on the Strava website, or through the official Strava app if they have it installed on their device.

FR-02 Request activity synchronization

The client can request an import of new activities from Strava. The request is accepted by the system only if any of these conditions are satisfied:

1. No request has been made yet.
2. A defined time duration has passed since the last request.

Each request syncs activities with a start time in a certain time range. If there is no previous sync request, a new sync is performed for up to last

40 activities. If there is a previously completed request, the new sync is performed for a range that starts at the end of the previous range and ends at the current time.

FR-03 Number of synchronized activities

The user is able to see the number of currently synchronized activities.

FR-04 Current activity synchronization state

The client can see the state of the last issued activity synchronization request. The possible states are:

- **Never**—no request has been issued yet,
- **Started**—synchronization is in progress,
- **Completed** (and the time of completion)—synchronization was completed,
- **Error**—the request ended with error.

FR-05 Notify user of sync completion

The user is notified once the previously requested activity synchronization is completed.

FR-06 Achievements check

The system checks for new achievements that can be unlocked. This is performed automatically each time after any new activities are synchronized.

FR-07 Unlocked achievements

The user is presented with both a number and a list of currently unlocked achievements for each time interval:

- week—unlocked since the start of the current week,

- month—unlocked since the start of the current month,
- year—unlocked since the start of the current year,
- all-time—unlocked anytime.

The intervals are combined with a filter for a type of sport, one of:

- run,
- ride (cycling).

FR-08 Achievement unlock detail

Details of an unlocked achievement can be displayed, including:

- name of the unlocked achievement,
- description of the achievement,
- the time period for which it was unlocked,
- the date when the achievement was last unlocked.

FR-09 Achievements

Achievements available for the user to unlock are listed below along with the names, conditions and other parameters.

Run 5 km. Run at least 5 kilometers in a single day. Available weekly and monthly.

Fast Pace. Achieve a running pace of 4:30 minutes per kilometer and cover at least 3 kilometers in a day. Available for all time periods.

Hill Runner. Gain at least 50 meters in elevation and achieve a pace of 4:45 minutes per kilometer in a day. Available for all time periods.

30-minute Runs. Run for 30 minutes on at least 2 days in a time period. Available weekly and monthly.

Tough Week. Run 20 kilometers in total with a duration of at least 90 minutes in a week. Available weekly.

Fast Ride (30 km). Ride at least 30 kilometers with an average speed of 25 kilometers per hour in a single day. Available for all time periods.

Ride 100 km. Cover 100 kilometers in a single day (ride). Available for all time periods.

Climber. Gain at least 300 meters in elevation and achieve an average speed of 20 kilometers per hour in a day. Available for all time periods.

Active Rider - Week. Ride for 2 hours and 50 kilometers in total per week. Available weekly.

Active Rider - Month. Ride for 1 hour on at least 10 days in a month. Available monthly.

2.2.2 Non-functional Requirements

These requirements are the *"constraints that must be imposed on the design of the system...."*[21]

I define several quality attributes and conditions under which the system runs.

NFR-01 Android version

The mobile app can run on Android version 5.0 (API level 21) and higher.

NFR-02 Simple usage

Any function in the mobile app should be accessible by less than 3 clicks.

NFR-03 Caching

Data from the backend should be cached in the mobile app and refreshed only after a certain expiration time (if refresh is not invoked by the user manually).

NFR-04 OAuth 2.0

The user can authenticate with the app using the OAuth 2.0 protocol [22].

NFR-05 Asynchronous processing

More performance-intensive tasks can be deferred and later processed asynchronously when resources become available.

NFR-06 Horizontal scalability

The backend can be scaled by increasing the number of nodes.

2.3 Domain Model

The domain model represents identified entities of the system and the relationships between them. The diagram is depicted in figure 2.5.

User. The **User** entity holds authentication data for a single user in the system. These data include the access and refresh *tokens*. It typically also has **External Authentication Data**.

External Authentication Data. This entity holds the necessary authentication data for communication with an external service identified by the *external provider* attribute. In the case of Sportivator, this would only be Strava, but the system allows adding another external provider if needed.

Athlete. **Athlete** does not require any authentication data, and its only connection to the **User** is through the *id*. It can have multiple **Activities** and **Achievement Unlocks**.

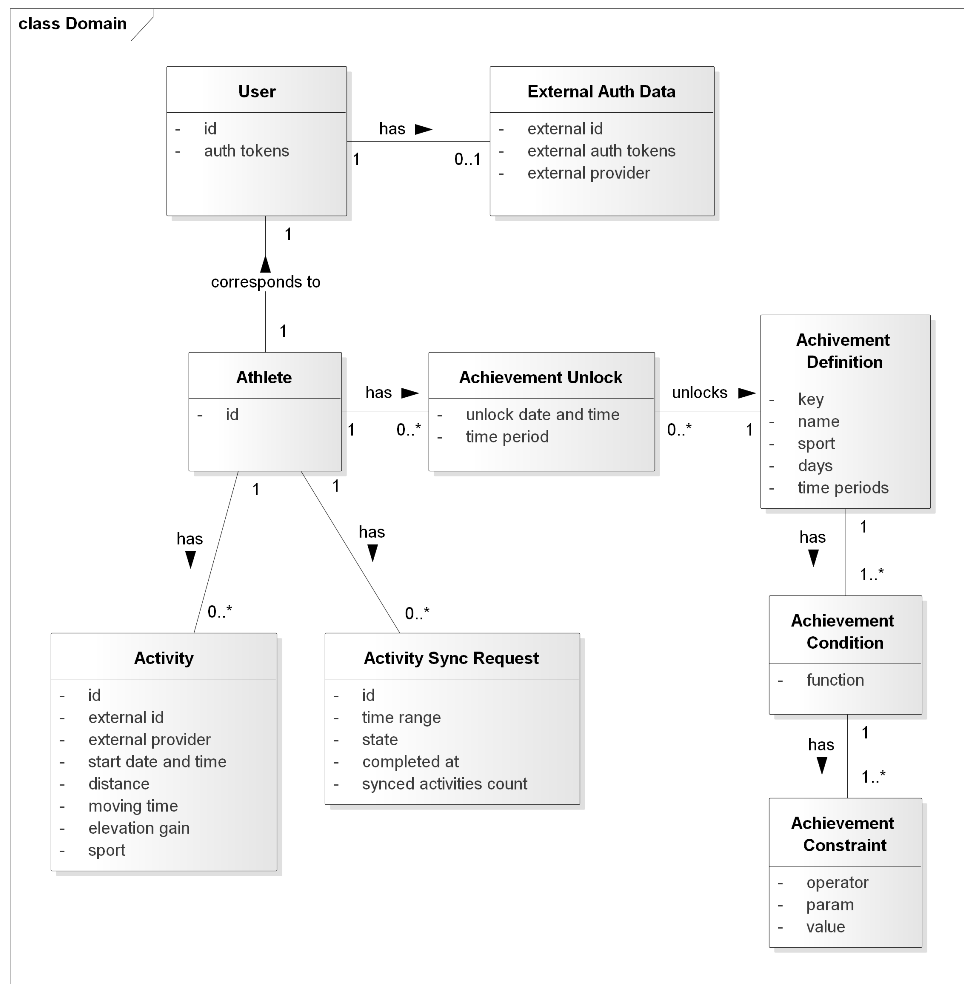


Figure 2.5: Domain model.

Activity. This entity stores the attributes of a single sport activity and belongs to a single **Athlete**. The attributes hold values in metric units as the Strava API returns data in the metric system. All of the attributes listed below are stored in the system for further analysis:

- *distance* in meters,
- *moving time* in seconds,
- *elevation gain* in meters.

The entity has the *external id* and the *external provider* attributes, which together identify it uniquely across all activities.

Activity Sync Request. Records a single request for the synchronization of activities issued by the user. Each request synchronizes activities for a

certain *time range*. It can be in several *states*.

Achievement Definition. This entity defines an achievement that can be unlocked by the user. It is uniquely identified by a *key*. It has a user-friendly *name*, applies to a *sport* and can be unlocked only for the defined *time periods*. These sport types are defined:

- **Run** (running),
- **Ride** (cycling).

The attribute *days* is optional, and if defined, the **Achievement Conditions** must be satisfied for the specified minimum amount of *days* (**Activities** grouped by days) in the *time period*. Otherwise, the conditions can be satisfied by looking at all **Activities** in the *time period* (grouped by *time period*).

Achievement Condition. The condition of an **Achievement** has a single attribute *function*, which defines the aggregation function performed on an attribute of the **Activities**. Typically, we are interested in a sum, e.g. the sum of *distances* of **Activities** in a *time period*.

Achievement Constraint. This entity defines the constraint on the result of the *function* from the **Achievement Condition**. The *param* defines the attribute of an **Activity** we are interested in, e.g. the *moving time*, and represents the first operand. The second operand is a *value*, e.g. *1000*. The *operator* is the operation performed on those two operands, e.g. *greater than or equal to*. In this example, the constraint would be: *moving time is greater than or equal to 1000*. There can also be two *params* not defined in the **Activity** entity:

- *pace*—minutes per kilometer (typical for running),
- *average speed*—kilometers per hour (typical for cycling).

These represent special cases and must be used along with the division aggregation *function* in the **Achievement Condition**, since computing the values requires looking at both *distance* and *time* attributes and dividing them.

Achievement Unlock. The **Achievement Unlock** entity is related to a single **Athlete** and unlocks a single **Achievement Definition**. The unlock occurs for a *time period* and at a certain *date and time*.

2.4 Use Cases

Use case is a *"way in which a system can be used, described as a step-by-step sequence of actions, along with the system's response and certain other information."*[21]

Actor represents a *"role that a user or some other system plays when interacting with... system."*[21]

Figure 2.6 shows the actors identified in the system. Figure 2.7 shows the use case diagram in the context of the mobile app, and figure 2.8 the use cases for the backend. Following are the descriptions of the use cases.

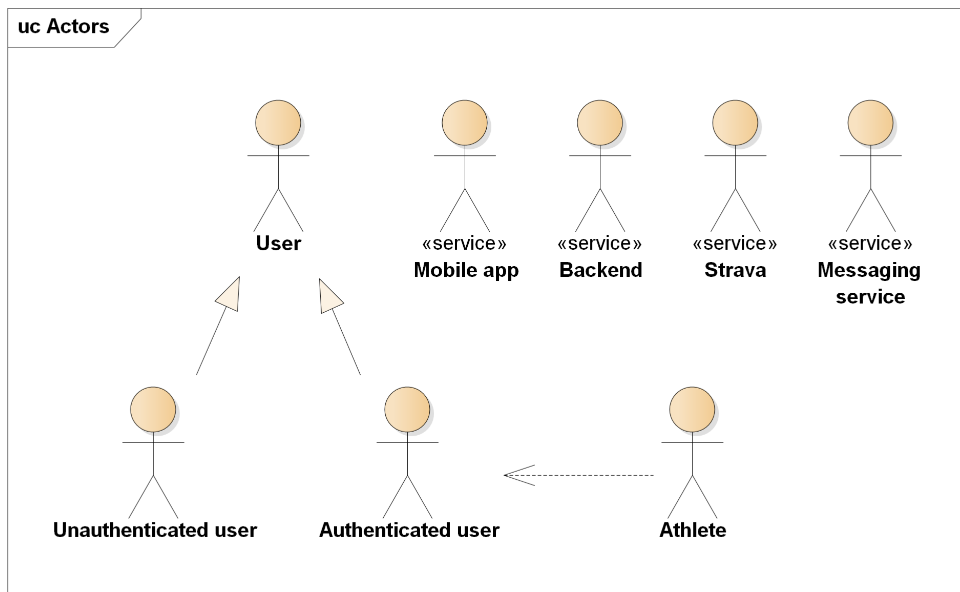


Figure 2.6: Actors in the system.

UC-01 Authenticate through Strava (mobile app)

When the app is first launched, or the user is not currently authenticated in the app, a screen prompting to authenticate with Strava is displayed. The user clicks on a button, is redirected to the Strava app or website and authorizes access to their data. This takes them back to the app, the authentication process completes and opens the main screen in the app.

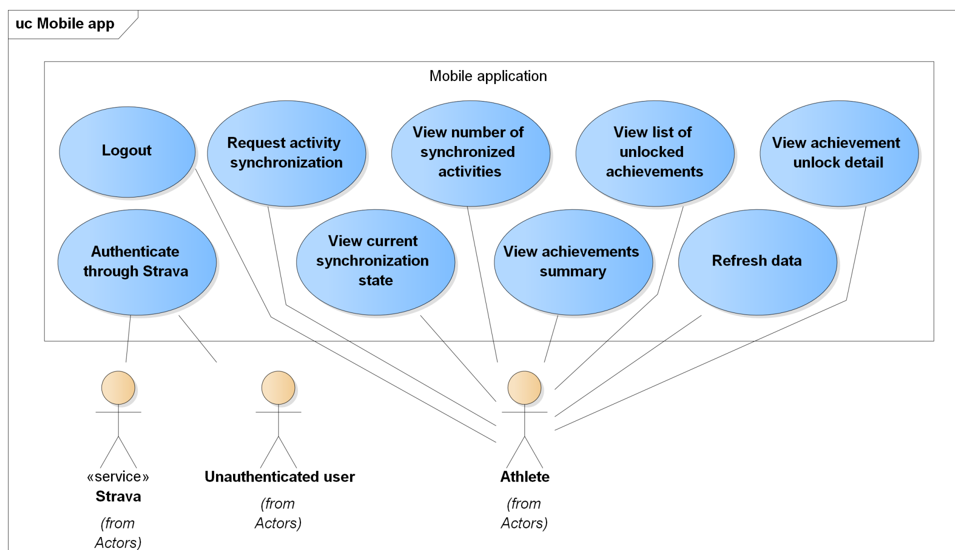


Figure 2.7: Mobile application use cases.

UC-02 Logout

The user can logout in the app by navigating to the settings screen and clicking on the logout option. This takes them to the initial authentication screen.

UC-03 Request activity synchronization (mobile app)

To request the synchronization of activities, the user opens the app and clicks a button on the main screen. The app sends the request and displays a new state.

UC-04 View current synchronization state

When the user opens the app, the current state of the latest request for synchronization of activities is displayed on the main screen. If it was completed previously, the state also displays the date and time of completion. The last state can be refreshed by the user, and it is refreshed automatically after the app is notified by the messaging service.

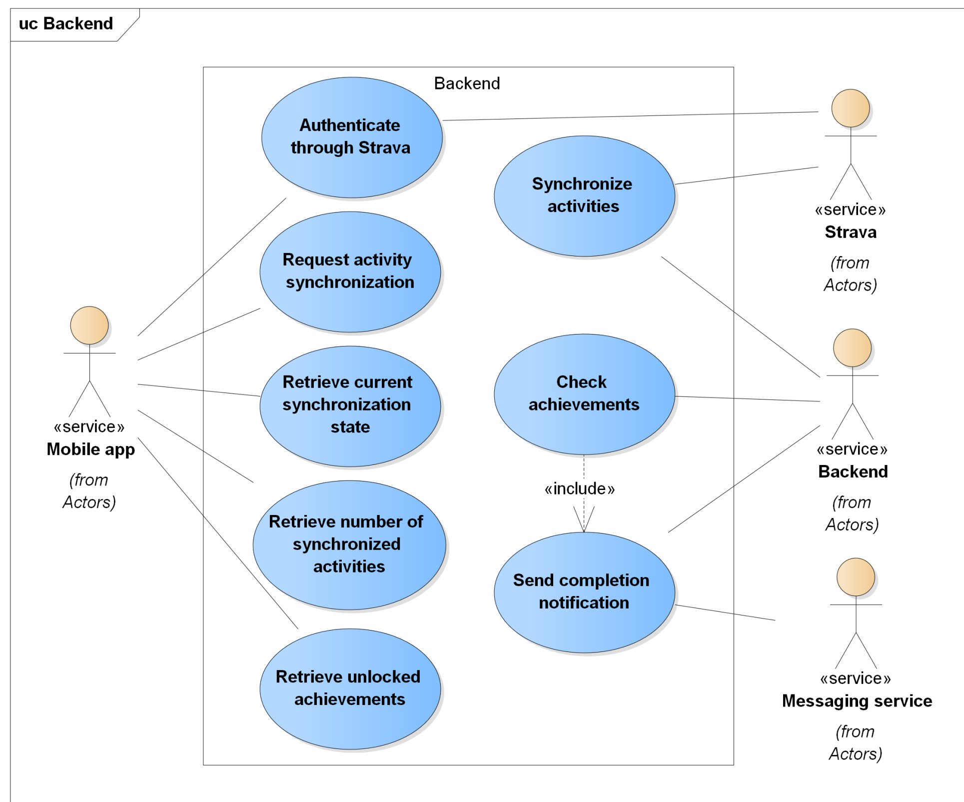


Figure 2.8: Backend use cases.

UC-05 View number of synchronized activities

The user can see the number of currently synchronized activities on the main screen. This number is requested automatically by the app or refreshed upon a user's decision to do so.

UC-06 View achievements summary

The number of achievements for each time period is displayed for the user on the main screen. It can be refreshed by a button.

UC-07 View list of unlocked achievements

To display a detailed list of unlocked achievements, the user has to click a button to open a separate screen. The list there can be further filtered by a time period and sport. Default filters are selected at first when the user navigates to the list.

UC-08 View achievement unlock detail

The user can display the details of an unlock of a single achievement by opening the app, navigating to the screen with the list of unlocked achievements, and clicking on the desired item representing the unlocked achievement. This opens a dedicated screen displaying the details.

UC-09 Refresh data

The user can manually refresh the state of the latest activity synchronization request, the number of synchronized activities and the achievements (including the summary) by pressing a button on the main screen. The refreshed data are then displayed.

UC-10 Authenticate through Strava (backend)

Authentication requires input from the user, which is processed by the mobile app and forwarded to the backend. The backend exchanges authentication information with Strava, and looks whether the user's account was previously created. If no previous account is found, a new account is created. Then the backend issues new authentication tokens and returns them to the mobile app. This completes the authentication process.

UC-11 Request activity synchronization (backend)

The mobile app sends a request for activity synchronization to the backend through the API. The backend then puts this request to the message queue and returns a new state to the app.

UC-12 Synchronize activities

Activities are synchronized from Strava asynchronously once a request is retrieved from the message queue. Then a request to check for new achievements (that can be unlocked for the user) is added to the queue.

UC-13 Check achievements

Checking for new achievements is performed asynchronously as well. The backend processes the imported activities and determines which achievements can be unlocked. The mobile app is then notified of its completion. To send a message with the completion notification, an external messaging service is contacted. The service sends the message to the mobile device.

UC-14 Retrieve current synchronization state

The mobile app contacts the backend to retrieve the current synchronization state for the user and displays it.

UC-15 Retrieve number of synchronized activities

The mobile app asks the backend for the number of currently synchronized activities on the user's behalf. Once returned, the number is displayed.

UC-16 Retrieve unlocked achievements

The unlocked achievements are requested from the backend by the mobile app to provide data for both the achievements summary and the list of unlocked achievements.

Chapter 3

Design

In this chapter, the design choices for implementing the system are discussed.

3.1 Overview

The whole system consists of three major parts:

- mobile application for Android,
- backend running on a cluster,
- database.

Figure 3.1 depicts this situation.

3.2 Android App

Over the years, multiple architectures have been proposed for developing Android apps and this area is still evolving. With that said, the Android team has been active recently by introducing new libraries and creating guidelines with best practices and recommended architectures [23]. One of the recommended architectural patterns is called *MVVM (Model-View-ViewModel)*, which is also employed here. The components are shown in figure 3.2 and described below.

Figure 3.1: System overview.

View. The *View* displays the *model* (data) for the user on the screen. It also handles user's input, which is then forwarded to the *ViewModel*. It subscribes to the *ViewModel* to receive updates of the *model* and other properties (e.g progress update).

ViewModel. The *ViewModel* provides functionality to the *View*, however, it does not contain any reference to it. It communicates with the *Repository*.

Repository. The *Repository* abstracts the underlying data sources. There is a local and a remote source, these are combined together.

Local Storage. This component abstracts access to the database.

Remote. The *Remote* communicates with the backend through the API.

With that in mind and considering the requirements in the analysis part, there are several points that I outlined for the design of the mobile app.

1. It is offline-first. In other words, data should be loaded from the local storage first before being fetched from the backend (caching). And only after it is determined that the local data are expired, refresh should be invoked. This also minimizes the communication with, and in turn the performance demands on, the backend.
2. It uses the principle of *Single Source of Truth (SSOT)*. It means that data always come from a single source, for example a database, and there are no duplicates. So even when data are refreshed, they must be saved

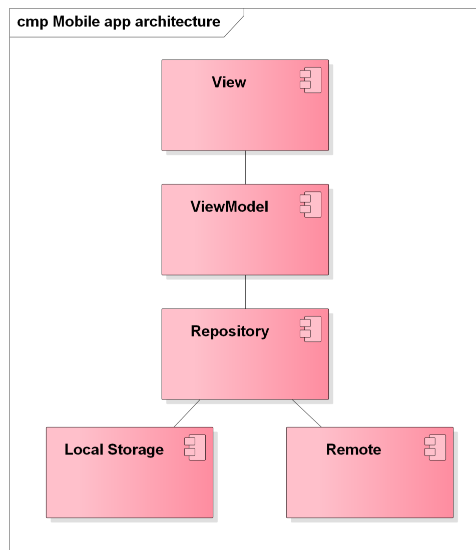


Figure 3.2: Mobile app architecture.

in the database, and this change is reflected in the UI by propagating the change through the system.

3. Reactive programming is employed here to support the propagation of data updates. *"Reactive programming is a general programming term that is focused on reacting to changes, such as data values or events."*[24]

Diagram in figure 3.3, created by the Android team, shows the decision tree for loading data from the local storage and fetching new data from the network.

3.2.1 User Interface

The mobile app does not require an overly complex user interface, which is also very clean. I designed an interface consisting of a bottom navigation bar and a main content area. The bottom navigation bar allows switching between two main screens: dashboard and a list of achievements. There is also a screen dedicated to displaying details about an unlocked achievement. The dashboard layout is depicted in figure 3.4, and the achievements screen design is shown in figure 3.5. Following is a more detailed description of the UI components.

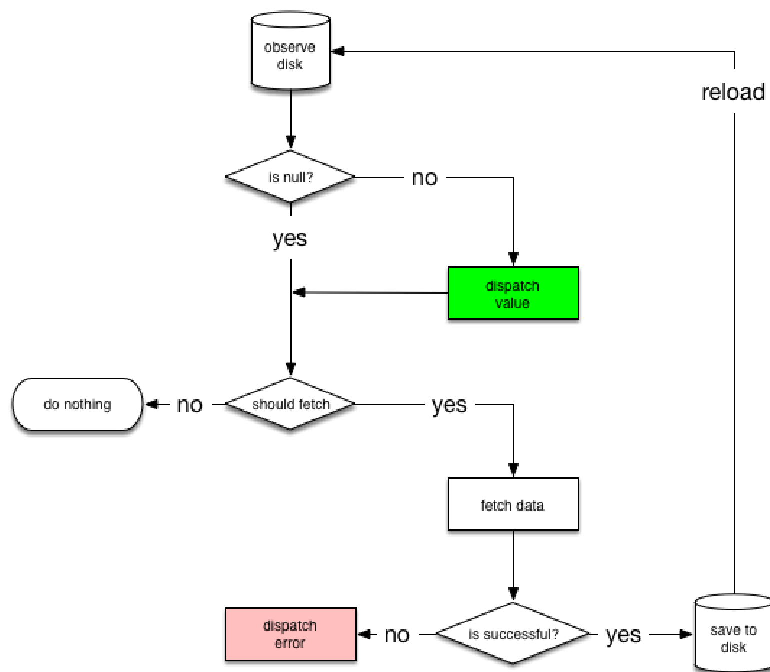


Figure 3.3: Data loading decision tree [5].

Dashboard Screen

Synced Activities Information. This is an area with basic information about synchronized activities. The key elements are:

- a label with a number of synchronized activities,
- a label with date and time information about when the last synchronization was completed,
- a button for requesting activity synchronization,
- a button for refreshing data.

Achievements Summary. Here, the user is presented with a summary of their achievements. This part of the screen shows the number of unlocked achievements for each time period.

Achievements Screen

The screen with achievements displays a list of unlocked achievements, and hosts a filter on top for different time periods:

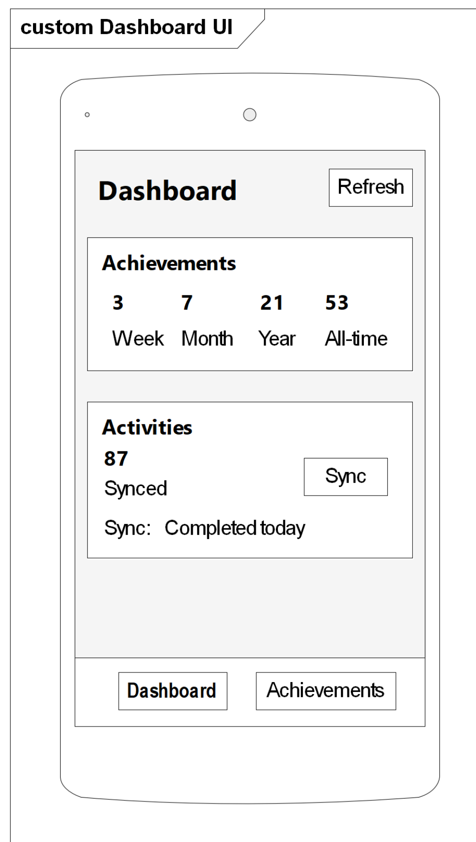


Figure 3.4: Dashboard screen layout.

- week,
- month,
- year,
- all-time.

There is a filter for sports as well.

The list contains items with each one representing an achievement with the following elements:

- name,
- description.

custom Achievements UI

Week Month Year All-time

Run Ride

Achievement 1

Achievement 2

Achievement 3

Achievement 4

Dashboard Achievements

Figure 3.5: Achievements screen layout.

Achievement Unlock Details Screen

This screen displays some details regarding a single unlock of an achievement. Located there are these elements:

- achievement name,
- achievement description,
- time period of the achievement unlock,
- the date when the achievement was unlocked for the time period.

Login Screen

This screen hosts a button that starts the Strava login flow.

3.2.2 Authentication

The user is authenticated through the OAuth 2.0 protocol [22]. Following are the steps to complete the authentication from the mobile app's point of view.

1. The flow starts by contacting the backend, which responds with the redirection to the Strava authorization page including a parameter with a URI to redirect to after the user grants the permission.
2. If the user has the official Strava app installed on their device, it should open and prompt the user to authorize access to their data for this app. Otherwise, the prompt opens in a browser.
3. After the user gives the required permission, Strava redirects them back to the app with a code required to finish the authentication.
4. The code is sent as part of the redirection URI previously received in step 1.
5. The backend exchanges this code for a Strava access token, retrieves user data, issues a new token and returns it to the mobile client.
6. The mobile client saves this token for future requests.

3.2.3 API

Communication of the client with the backend is performed through HTTP with requests and responses in the JSON format [25]. Based on the analysis, following is the specification of the API required by the mobile client. The descriptions include HTTP methods, paths, parameters and responses. By default, paths that require authentication return an *401 Unauthorized* response status if the authentication fails.

GET /activities/achievements

Returns the achievement definitions. Requires authorization.

GET /activities/achievements/unlocked/{timePeriod}

Returns a list of unlocked achievements (unlocks) for a time period specified by the *timePeriod* parameter in the path. Unlocks are returned for the authenticated user (athlete). If the supplied time period is not found, the response status is *404 Not Found*.

GET /activities/data

The response contains the date the and time of the last successfully completed activity synchronization and the number of currently synchronized activities for the authenticated athlete.

GET /activities/sync/status

Gets the status of the latest request for synchronization of activities for the authenticated athlete. This includes the current state, the date and time of completion (if applicable), the number of activities that were synchronized for this request (if completed), and the date and time of the next available synchronization. If there was no previous request found, it is reflected in the attributes.

POST /activities/sync

Requests a synchronization of activities for the authenticated athlete. The response is the same as for the synchronization status request. There are no parameters. If a new request cannot be issued yet, a *400 Bad Request* response status is returned.

GET /auth/oauth2/authorize/strava

This path is used for starting the OAuth authentication flow. The response status is *302 Found* (redirection) here with the *Location* header pointing to the Strava authorization endpoint. The request is not required to be authenticated.

GET /auth/oauth2/redirect

This is the final path to redirect to after a successful authorization on the Strava endpoint. It returns the authentication tokens and expiration information in the response. If the authorization could not be completed, the response status is *401 Unauthorized.*

POST /auth/token/refresh

Issues a new access token based on the supplied refresh token. The response includes the new access token, a new refresh token and the expiration date and time of the new refresh token. If the provided refresh token from the client is invalid or expired, the response status is *401 Unauthorized.*

POST /auth/messaging/register

Registers a unique token from the client. The token is used on the backend to send asynchronous messages through an external messaging service to the client. The request contains the token and must be authenticated.

3.2.4 Entities

The mobile client identifies several entities (persisted in the database and local storage).

- **Achievement**—represents a definition of an achievement.
- **AchievementUnlock**—an unlock of an achievement by the user for a *time period*.
- **ActivityData**—holds information about the number of synchronized activities and the date and time of the last successful synchronization.
- **ActivitySyncStatus**—status of a single request for an activity sync, it can be in several states.

3.3 Backend

The backend is designed using the principle of the *microservices* architectural style.

"Microservices are an approach to distributed systems that promote the use of finely grained services with their own lifecycles, which collaborate together. Because microservices are primarily modeled around business domains, they avoid the problems of traditional tiered architectures."[26]

So, the backend is a collection of largely independent services. Following are the main benefits of this approach. Each service should focus on a single business area, therefore it should be possible to manage it independently of other services. This also supports loose coupling of the system. Services might even use completely different technologies. They can be deployed quickly and independently, and typically have their own data sources as well. It offers a good scalability by replicating those services which have the highest load.

There are obviously some challenges with applying this architecture. One of them could be a greater complexity of the whole system. Versioning and

managing contracts between services can introduce issues too. If, for example, there are services dependent on the one being upgraded, the communication might break due of incompatible interfaces. Logging of user requests and determining the real cause of an error in the system is challenging as the request might travel across multiple services [27].

Another option in terms of the architecture of the backend would be to design a homogeneous system containing all of the functionality. To put it in contrast with the independent microservices approach, all of them would become a part of a single application. This style would mostly solve the drawbacks of independent microservices, but it might be more difficult to keep adding more features to it as the size and complexity grows (if we are not careful and do not make clear boundaries between components). Making a small change would require a redeployment of the whole application as opposed to deploying a new version of a single service. Also, it is much less scalable compared to the *microservices* style. However, it should be noted that a performance issues scenario might not even occur for many apps. A key metric to look at is the expected load of the application in time. Microservices are an obvious choice for companies like *YouTube* and *Netflix*. In contrast, a small online shop does not benefit from this approach as it only slows down the development process.

The bottom line is that the right choice of architecture is largely dependent on the application we are building. Even though the product created as part of this thesis is not very complex, one of its objectives is to show that the *microservices* architecture can be successfully applied here and that it keeps the door open for additional functionality in the future.

3.3.1 Service Architecture

Each service in the system follows a layered architecture with three basic layers (shown in figure 3.6):

1. **Controller**—provides the REST API and handles client requests,
2. **Service**—contains the business logic (a term used for a component in a microservice),
3. **Repository**—abstracts access to the database.

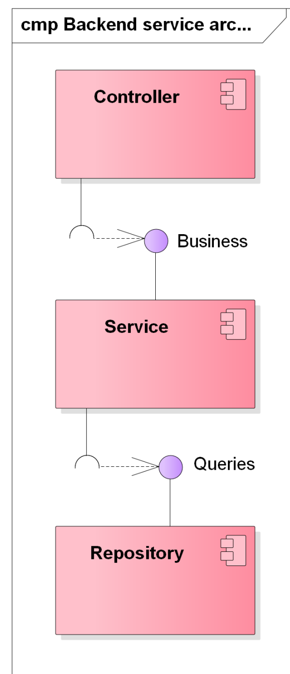


Figure 3.6: 3-layer architecture of a backend service.

3.3.2 Services

The backend consists of multiple independent services (microservices) as depicted in figure 3.7.

Gateway

The **Gateway** service is the only front-facing component (visible for the client) of the backend. It exposes a single interface for all the services behind it. Requests are forwarded to the respective services based on the URLs. The **Gateway** also determines the paths that are available only to authenticated requests. It ensures that requests to those paths have a valid authentication information before forwarding them to the corresponding services (otherwise returns an error response immediately).

Authentication Service

This service is responsible for:

- registering users,

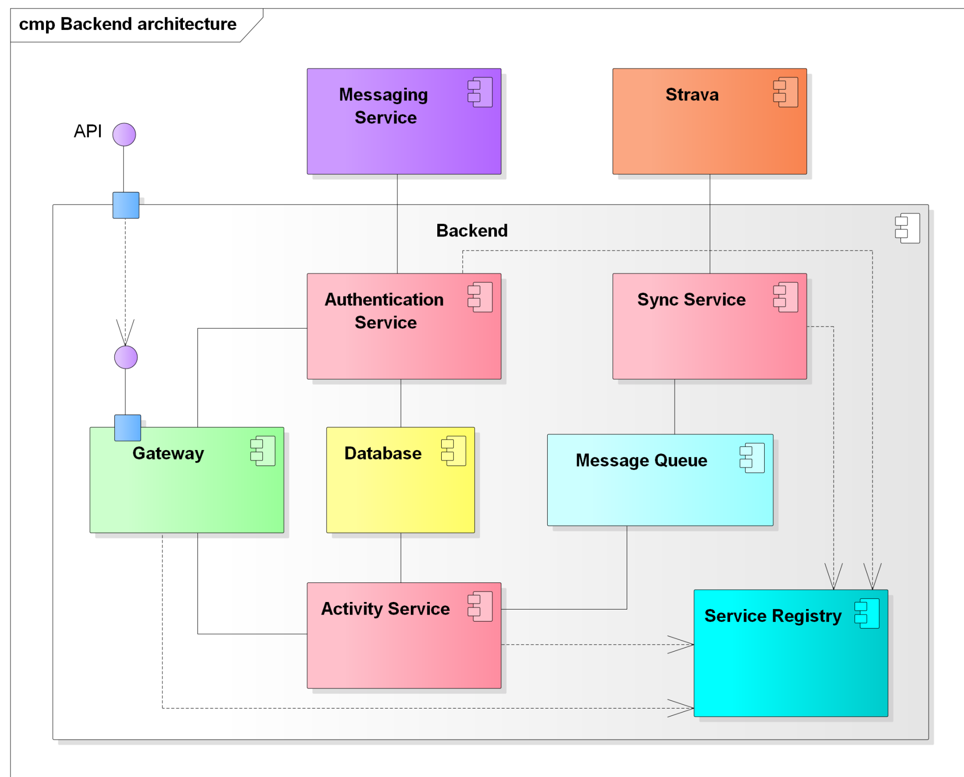


Figure 3.7: Backend services.

- handling the login flow,
- issuing tokens,
- providing tokens for accessing external services,
- sending notifications to the client through an external service.

Activity Service

This service provides all sport-related information and handles the corresponding requests. It subscribes to the **Message queue** to be able to asynchronously communicate with the **Sync service**. This includes:

- persisting and providing information about synchronized activities,
- logic for checking new achievements to unlock,
- persisting and providing a list of achievements,
- requesting activity synchronization.

Sync Service

The **Sync service** subscribes to the **Message queue** and waits for requests for activity synchronization. Then, it connects to Strava, performs the synchronization, and publishes a response with the retrieved activities to the **Message queue**.

Service Registry

The registry provides responses for service queries. It allows other services to register with it and maps the service names to specific locations.

Message Queue

The **Message queue** allows the services to communicate asynchronously by subscribing to it and enqueuing messages to named queues. A message is persisted until a successful confirmation is received from the service handling the message.

Database Service

The **Database service** provides access to relational databases for other services. Each database is accessed by a single service.

3.3.3 External Systems

The backend works with two external systems, the **Messaging service** and **Strava**, both depicted in figure 3.7.

The **Messaging service** is utilized to send asynchronous messages to the mobile app. Specifically, a message is sent to the app once a check for new achievements is completed. A natural choice for Android is the **Firebase Cloud Messaging** solution. It is *"a cross-platform messaging solution that lets you reliably deliver messages at no cost."*[28] The app receives messages through a service running in the background.

Strava is used for authentication and synchronization of sport activities.

Sequence diagram in figure 3.8 shows an approximation of the communication happening between different services when the client asks to synchronize activities. It also illustrates notification of the client after the check for new achievements to unlock is completed.

3.3.4 Entities

The backend works with database entities based on the domain analysis (see figure 2.5). These entities are part of the database model in the **Authentication service** (simplified diagram is shown in figure 3.9):

- **AuthUser**—a single user with *id*,
- **UserExternalAuthData**—authentication data for Strava account (or other external service).

Database model for the **Activity service** contains the following entities (depicted in simplified form in figure 3.10):

- **Activity**—a sport activity with attributes,
- **AchievementUnlock**—an unlock of an achievement by a user,
- **ActivitySyncRequest**—represents a single request for activity synchronization.

Achievement definitions are not stored in a database, instead they are loaded from a file and mapped to the entities listed below:

- **AchievementDefinition**—defines an achievement with conditions,
- **AchievementCondition**—defines the aggregation function over activity attributes and the constraints,
- **AchievementConstraint**—contains the comparison operator, an attribute to look at and a value to compare it with.

The system does not work with an **Athlete** object but uses an attribute with the name *athleteId* in appropriate places (same as the id of the **AuthUser** entity).

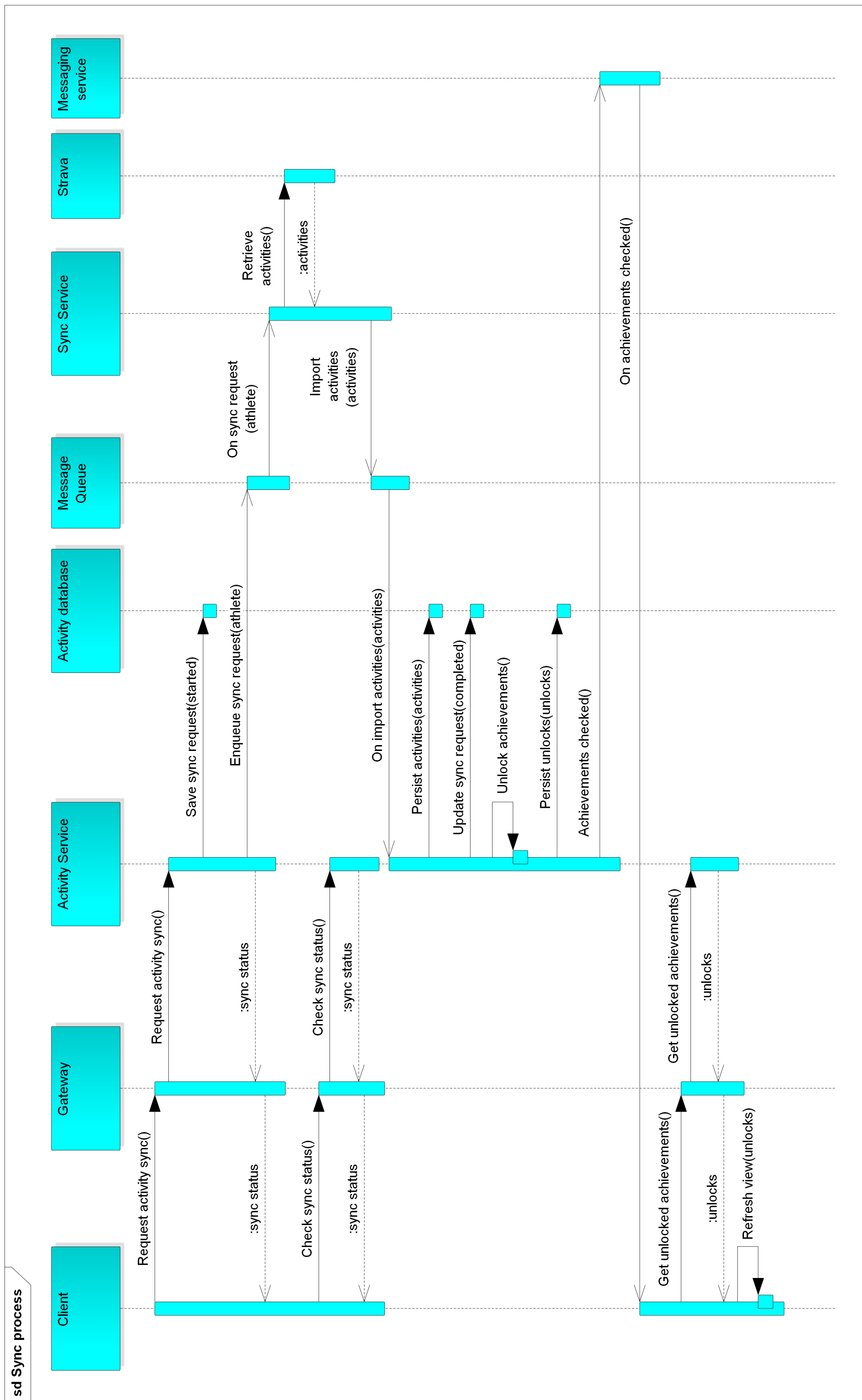


Figure 3.8: Sequence diagram—activity synchronization.

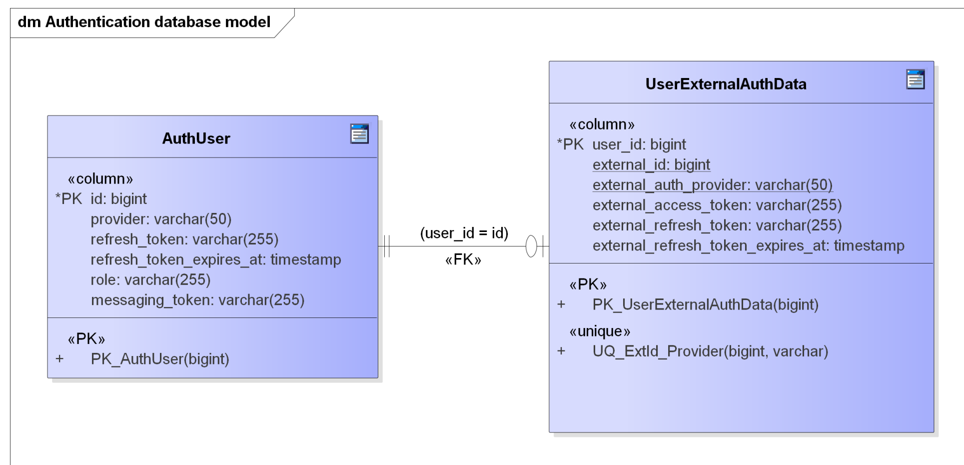


Figure 3.9: Database model—Authentication service.

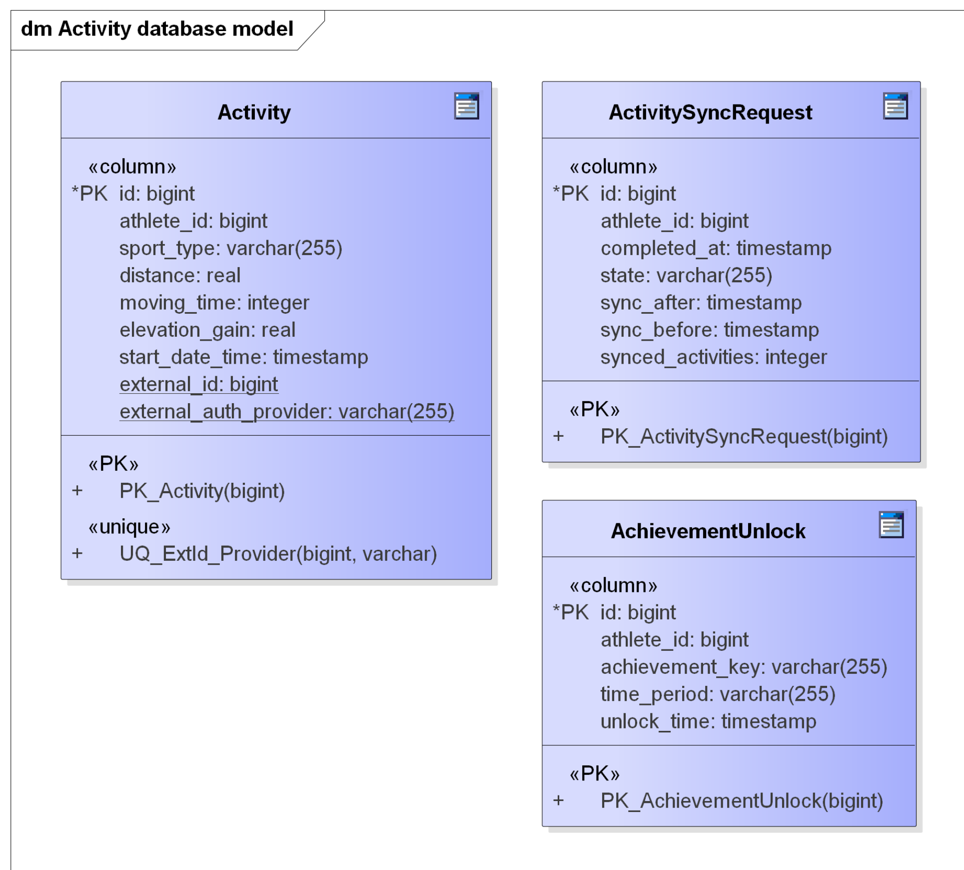


Figure 3.10: Database model—Activity service.

3.3.5 Authentication

As described above, all requests to the backend go through the **Gateway**. Most requests must be authenticated, and that is the responsibility of the **Authentication service**. However, to reduce load on this service and to improve latency, the system should be able to determine if the request is authenticated in the **Gateway** itself. This is best done by using a so-called *stateless* authentication, meaning all authentication information is sent with each request. That is why I decided to use *JSON Web Tokens (JWT)*. *"JSON Web Tokens are an open, industry standard RFC 7519 method for representing claims securely between two parties."*[29] These tokens allow to securely encode authentication information (signed by a secret code and containing an expiration date), and most importantly enable a *stateless* design of the authentication system. Thus, the requests can be authenticated in the **Gateway** just by looking at the included *JWT* and verifying the signature and the expiration date. If the verification fails, the client is forced to go through the OAuth login flow.

Since the backend needs to periodically retrieve data from Strava, it is required to store the tokens for accessing the Strava API. When a synchronization is due, the **Sync service** asks the **Authentication service** for an access token for the Strava API. However, this token does expire after some time: *"Access tokens expire six hours after they are created, so they must be refreshed in order for an application to maintain access to a user's resources. Applications use the refresh token from initial authentication to obtain new access tokens."*[30] Therefore, the authentication service must also be able to store the refresh token and request a new access token if necessary.

Chapter 4

Implementation

This chapter goes into detail regarding the implementation of both parts of the application. The source code is available on the enclosed SD card.

4.1 Android App

For implementing the Android app, the official **Android SDK** is employed [31]. The main arguments for its use are that it allows to access the newest APIs, provides the most flexibility and is easy to use with **Android Studio**, the official tool for developing Android apps from the Android team [31]. In addition, there are several third-party libraries included in the project as well. An alternative option would be to utilize some multi-platform framework, which could simplify releasing the app for other platforms (iOS). But since I have spent a lot of time working with the **Android SDK**, this was a clear choice.

I also decided to use Kotlin as a development language in all of the app's components, as it is the recommended choice for Android development at the time of writing this thesis [32]. It is a modern language with many useful features and it is fully interoperable with Java, which allows to use existing Android libraries [33].

The minimum SDK version for this project is 21 (*Lollipop*, 5.0) and the target SDK is 28 (*Pie*, 9).

4.1.1 Structure

The Android project is split into three main modules. This avoids unnecessary recompilation of modules that did not change and enables parallel compilation.

Domain

This module is a pure Java library—although it contains Kotlin code thanks to the interoperability—without any dependencies on the Android components. It defines interfaces that are implemented in the **data** module. There are also models representing the data displayed in the **mobile app** module. In terms of the architecture depicted in figure 3.2, this module only covers the contracts of the *Repository* component.

Data

The **data** module provides implementation for the interfaces in the **domain** module. It is an Android library, but does not contain any UI elements. The covered components are *Repository*, *Local Storage* and *Remote*.

Mobile App

This module defines the layouts, and implements the functionality of the screens discussed in the Design chapter. It covers the *View* and the *ViewModel* architecture components.

4.1.2 Libraries

There are multiple useful libraries included in the project. Table 4.1 lists some of them.

4.1.3 User Interface

Every screen from the design phase is mapped to a *Fragment* [42]. A logical set of *Fragments* is displayed inside an *Activity* [43]. Both *Fragments* and *Activities* require *ViewModels*.

Library name	Description
Android Navigation	Navigation between app's components [34].
Android Room	Abstraction over database access [35].
Retrofit	HTTP client [36].
RxJava	Java VM implementation of Reactive Extensions [37].
RxKotlin	RxJava bindings for Kotlin [38].
Dagger	Dependency injection framework [39].
ThreeTenBP	Time library, replacement for Java APIs [40].
Timber	Logger for Android [41].

Table 4.1: Libraries in the Android project.

4.1.4 ViewModel

The Android framework provides an implementation of a *ViewModel*, which is aware of the lifecycle of the UI components [44]. As such, it solves common issues with handling configuration changes. All of the *ViewModels* are injected into *Fragments* and *Activities* by the means of dependency injection provided by the *Dagger* library [39].

4.1.5 Models

Models represent data displayed in *Views*. All persisted entities are mapped to their corresponding models (see figure 3.2.4). In addition to those, there are other notable models.

- **AuthData**—holds authentication data, including the tokens.
- **AuthEvent**—information about an authentication event in the app.
- **UnlockWithAchievement**—an achievement unlock joined together with an achievement definition.

4.1.6 Local Storage

There are two local persistence solutions used by the app. The first one—database—is used for persisting **Achievements** and **AchievementUnlocks**. The second one is a key-value persistent storage abstracted by the *Shared-Preferences* interface from the Android components [45].

4.1.7 Caching

Even though today's mobile network infrastructure might suggest otherwise, it is vital for a modern mobile app to support caching of data that it downloads from its remote sources. Connection might be flaky in some places, and caching also helps reducing performance demands on the backend. The strategy I implemented works with constants, which specify the duration after which data are considered to be "expired" and required to be fetched from the remote source again. The user can always refresh the data manually, and the constants can be fine-tuned in the future.

4.1.8 Authentication

The authentication flow of the app requires redirection to Strava and then to the backend. I wanted the mobile client to handle authentication without opening the browser. The user can authorize access for this app directly in the Strava app if it is installed on the device [30]. The app must send an *Intent* with a mobile-specific URI to open the Strava app [46]. After the user's confirmation, the Strava app also sends an *Intent*, which is handled by the mobile client by registering a special *Intent Filter* [47]. Finally, the result is forwarded through the API to the backend to complete the authentication.

Every OAuth request is stored in a cookie, therefore, the client must be able to manage cookies. This is implemented by using an *Interceptor* for the authentication API [48].

Included in the final response of the login flow are the *access* and *request* tokens. Because the *access* token is set to expire in 6 hours, it needs to be refreshed with the help of the *refresh* token after this period. The logic for refreshing the *access* token is implemented in an *Authenticator* [49].

The *refresh* token also has a limited validity for 30 days, even though a new *refresh* token is issued after each successful refresh request. If it comes to the case when both tokens are invalid, the app opens the login screen and prompts the user to reauthenticate with Strava.

4.2 Backend

The framework used for all services in the backend is *Spring Boot*, which is based on the *Spring* platform. *"The Spring Framework provides a comprehensive programming and configuration model for modern Java-based enterprise applications - on any kind of deployment platform."*[50]

It is a framework built on top of the *Java Enterprise Edition (Java EE)* platform consisting of different modules. The core technologies include dependency injection, resource handling, validation and others. The key part of *Spring* is *Spring Web MVC*, which serves the same purpose as other *MVC (Model–View–Controller)* frameworks. *"Spring Web MVC is the original web framework built on the Servlet API and has been included in the Spring Framework from the very beginning.... it is more commonly known as 'Spring MVC'."*[51]

Spring Boot then makes it much easier to get started with *Spring*. *"Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can 'just run'. We take an opinionated view of the Spring platform and third-party libraries so you can get started with minimum fuss. Most Spring Boot applications need very little Spring configuration."*[52] It has the ability to embed a servlet container (like *Apache Tomcat*) directly, so it can be deployed as a standard JAR file [53]. This is useful, as an example, to create a standalone container which can be deployed on a cluster.

Another important part of *Spring* is *Spring Cloud*, which provides many components useful for the *microservices* architecture. *"Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems (e.g. configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus, one-time tokens, global locks, leadership election, distributed sessions, cluster state)."*[54]

4.2.1 Structure

The backend uses *Maven* as a project management tool [55]. Each service in the backend is implemented as a single module while sharing the same parent with other modules. The parent defines dependencies required by all services. Also, some functionality is required by some, but not all, services. For that reason, there are also multiple modules not bound to a specific service that can be shared. Figure 4.1 depicts this structure.

4.2.2 Libraries

Apart from the basic *Spring Boot* libraries, the backend uses dependencies from the *Spring Cloud* family (see Table 4.2 for examples).

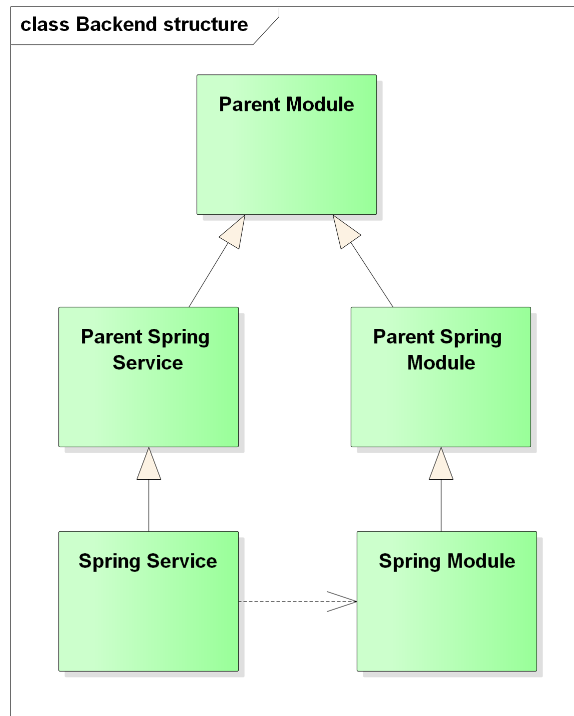


Figure 4.1: Backend project structure.

Library name	Description
Ribbon	Client-side load balancer. [56].
Sleuth	Distributed tracing [57].
Zuul	HTTP client [58].
OpenFeign	Declarative REST Client [59].

Table 4.2: *Spring Cloud* libraries.

4.2.3 Authentication

The backend uses OAuth 2.0 for authentication. The mechanism is implemented in the *Spring Security* package [60]. The **Authentication service** uses this package in the client mode. This requires some basic configuration including these parameters:

- *clientId*—the identifier by which the OAuth provider identifies the client (provided by Strava),
- *clientSecret*—a secret required by the OAuth provider,
- *scope*—the scope of data which the client can access.

On Strava, there are scopes specific to activity data, the one required by

the backend is *activity:read_all*—allows access to all activities on the athlete's profile, including privacy zone data [30].

4.2.4 Activity Synchronization

The synchronization of activities is not handled immediately after a new request from the client is received on the **Activity service**. Instead, a new instance of the **ActivitySyncRequest** is created and put to the **Message queue**. The **Sync service** is subscribed to the queue, and when a new request arrives, it performs the synchronization and puts a new response to the queue along with the result and retrieved activities. **Activity service** then dequeues this response and persists the activities to the database. This way, the **Activity service** is isolated from Strava.

4.2.5 Achievements

I wanted the system for managing achievements to be flexible and allow having the definitions separated from the application code. The entities are designed in a way that supports creating various achievements with multiple conditions. It is easy to define achievements in a file, parse the file and build the entities. I decided to describe the achievements in a JSON file, which can be supplied from a local disk or a separate configuration service if needed. JSON is a lightweight format which is easy to parse and read [25]. Below is an example of a JSON-defined achievement for cycling as the sport type, week time period, with the sum of the distances of all activities being at least 30 kilometers.

```
{
  "key": "ride_dist_30_km_tp",
  "sport": "Ride",
  "timePeriods": ["Week"],
  "conditions": [
    {
      "function": "Sum",
      "constraints": [
        {
          "op": "Ge",
          "param": "distance",
          "value": 30000.0
        }
      ]
    }
  ]
}
```

Since all the necessary activity data for unlocking achievements are stored in the database, a straightforward way to perform the check is to run a series of database queries. The queries must be constructed dynamically from the entities, and that is a task for the *AchievementChecker* class. The queries follow the basic idea below (pseudo-SQL code):

```
SELECT [expression in GROUP BY] FROM Activity
WHERE [specific athlete, sport, start and end dates]
GROUP BY [start date or ‘1’]
HAVING [aggregation function(attribute) operator value]
```

The result of the query must contain at least one row in the case of achievements without a minimum number of *days* specified. Otherwise, there must be at least *days* rows returned.

Activities are, if the *days* attribute is defined, grouped by a *start date* to run the aggregate function on all activities in a single day (since the constraint in the *HAVING* clause must be satisfied considering all activities in that day as a group, usually taking a sum of their attributes). If the *days* attribute is not defined, the activities are grouped by ‘1’ which effectively makes every activity a single group. The result of the aggregate function must then satisfy the comparison in the *HAVING* clause.

4.2.6 Communication between Services

Since clients can only access the **Gateway** from outside the cluster, there can be internal paths available only for internal service-to-service communication without the requirement for authentication. That is exactly what I implemented by adding an *ignored pattern* in the *Zuul* configuration for paths containing the text “protected” [61]. The paths below are part of the internal API.

GET /protected/external/token/{athleteId}. This command is implemented in the **Authentication service** and called from the **Sync service** to request Strava token for athlete with the specified id.

POST /protected/messaging/notify. Requests are being performed on this path to notify an athlete (specified in the body of the request) of an achievements check completion. Also implemented in the **Authentication service**, the path is used by the **Activity service**.

The actual implementation of communication between services is enabled by *Feign*, which makes it easy to define the communication by creating interfaces describing the paths, methods and parameters [59].

4.2.7 Deployment

All services in the system can be deployed independently. A robust way to achieve that is to build a container for each service and deploy it on a cluster. *"A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another."*[62]

I worked with three deployment scenarios while creating the system. To quickly test new features during development, I prepared configuration for each service in my development tool, which was *IntelliJ IDEA* [63]. I did not use containerization here. *Eureka*, part of the *Spring Cloud Netflix* family, served as a local **Service registry** [64].

The other two deployments were based on *Docker* containers inside a cluster. *Kubernetes* was chosen as the cluster technology. *"Kubernetes (K8s) is an open-source system for automating deployment, scaling, and management of containerized applications."*[65]

Since *Kubernetes* supports service discovery out-of-the-box, there was no need to have a dedicated **Service registry** running.

In order to have a working cluster on my local machine, I used *Minikube*, which enables running a local *Kubernetes* cluster [66]. A *PostgreSQL* database was based on a *Docker* image running alongside other services in the cluster [67].

The production deployment of the backend was done on *Google Cloud*, specifically, the *Kubernetes Engine*. *"Kubernetes Engine is a managed, production-ready environment for deploying containerized applications."*[68]

Database solution for production was based on *Cloud SQL* (*PostgreSQL* option), also part of *Google Cloud*, which runs outside the *Kubernetes Engine* [69]. The containers connected to the database through the *Cloud SQL Proxy Docker* image [70]. **Message queue** is based on *ActiveMQ* [71]. The configuration pulls a *Docker* image from the *Docker Hub* registry [72].

Deployment configuration can be found inside the *kubernetes* directory in the *parent-spring-service* module for both *Minikube* and *Google Cloud*.

Figure 4.2 shows the production deployment with the mobile app, backend

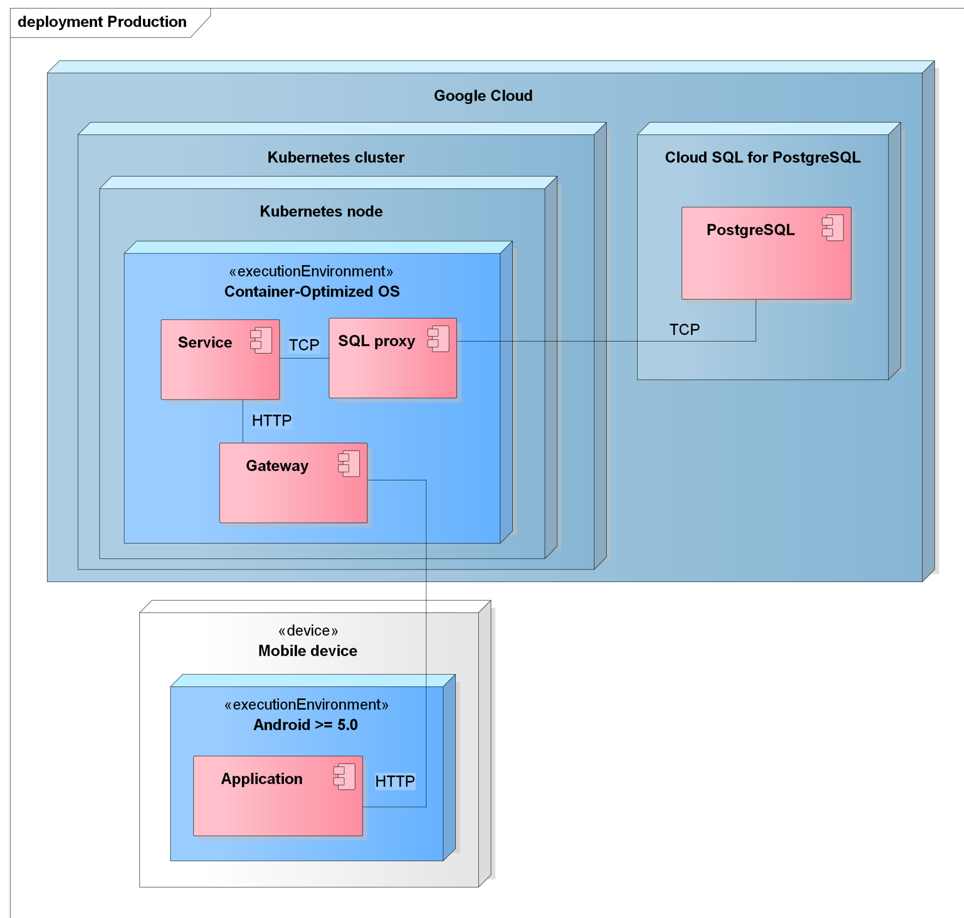


Figure 4.2: Production deployment.

and database in a single diagram. There is the **Gateway** as the front-facing component of the cluster. Other services are abstracted under a single *Service* component in the diagram. The *SQL proxy* then shows the communication with the *Cloud SQL* solution.

Chapter 5

Testing

This chapter lists and describes the test scenarios for the system and their expected results. All tests passed successfully.

5.1 Android App Testing

The Android app uses dependency injection, so it is possible to provide implementation of components specifically designed for testing. As an example, to make the tests not dependent on the backend and its API, mock implementations of the *Remote* components were created. They simulate network behavior and return identical responses to the ones coming from the real backend. However, the tests can initialize the responses with mock data. These mock components and other testing utilities are in a separate module *testutils*.

5.1.1 End-to-end Tests

This series of tests validates a proper behavior of all modules in the app from the user's point of view. It tests the screens, simulates input and validates output in the UI. Mock *Remotes* and database are used. The technology providing user interface testing capability for Android is called *Espresso* [73]. Tests can run on real or virtual Android devices. The tests described below can be found in the *app* module in package *androidTest.java.cz.cvut.masters.screen*.

Dashboard—First Run. In this scenario, mock *Remotes* are set up with initial data as if a new account was created for the user. The dashboard screen is launched. All text elements are expected to have their initial values.

Settings Button. The dashboard screen is opened and the settings button is clicked. This should open the settings screen.

Navigate to Achievements Screen. The dashboard screen is opened and the achievements button in the bottom navigation bar is clicked. Unlocked achievements screen should now be displayed.

Dashboard—Refresh. Mock *Remotes* are set up with initial data. All UI elements should contain initial values. Then, the *Remotes* are updated with some artificial data. The refresh button is clicked. The UI elements are expected to contain the updated values.

Dashboard—Sync Success. Mock *Remotes* are set up with initial data. Assertions on the initial values are performed, this also includes checking that the synchronization state is *Never*. The *Remote* for achievements is updated with a *Started* state and then the sync button is clicked. The text with the synchronization state should now translate to *Started*. Finally, the *Remote* state is changed to *Completed*. Refreshing the state should reflect that.

Dashboard—Sync Error. Initial assertions are performed normally. The *Remote* state is set to *Started* and sync button click is performed. The synchronization state should translate to *Started* now. Then, the state in the *Remote* is changed to *Error*, and the refresh button is clicked. The text with synchronization state should inform about the error.

Achievements Screen—No Achievements. The test prepares initial data in the achievements *Remote* first. Then it navigates to the achievements screen and checks that "no achievements" message is displayed.

Achievements Screen—Filtering. There are multiple test scenarios to check all combinations of filters for time periods and sports. First, achievement unlocks for all those combinations are added to the achievements *Remote*. Each test then clicks on one time period filter and one sport filter. The expected result is that only activities satisfying the filter conditions are displayed in the list.

Achievement Unlock Detail Screen. The *Remote* is set up with two unlocks of achievements. In the scenario, a click is performed on each of those achievement unlocks in the list. This should open the screen with details and display correct values in corresponding UI elements.

Dashboard and Achievements Screen. This test scenario combines multiple actions and navigates between the dashboard and the screen with unlocked achievements. Mock data are prepared such that invoking activity sync by clicking the button on the dashboard and then clicking on the refresh button should update the number of unlocked achievements. Also, the new achievements should be displayed on the achievements screen.

5.2 Backend Testing

Since each microservice is in a separate module, I created a few test modules containing functionality that can be reused in multiple microservices.

5.2.1 API Tests

Test scenarios for the API cover the *Controllers* in microservices. These tests work with a mocked *Service* layer and verify HTTP responses including the status and JSON data. The *Spring Security* package provides functionality for running tests with a mock user to fulfill authentication requirements of the API [74]. To make the tests efficient, a special annotation *WebMvcTest* was added, which instructs Spring to load only the *Controller* layer [75]. The tests are located in corresponding microservices in package *test.java.cz.cvut.masters.service.{serviceName}.controller*.

5.2.2 Unlocking Achievements

Tests that validate logic for unlocking achievements cover the **achievements** module. Because the *AchievementChecker* class builds database queries and works with persisted entities, the test class is annotated with *DataJpaTest*, which enables injecting a special *EntityManager* for testing (utilized for persisting test data) [76]. The tests also work with a *Clock* object providing a fixed system time, so they can be run independently of the local time in the testing environment [77].

There are two test classes for testing both achievements that are defined for a minimum amount of days (day-based) and achievements considering activities across a time period (time period-based). Every achievement is a combination of multiple attributes, conditions and constraints. The tests cover a subset of all possible combinations. Both classes, *DaysAchievementsTest* and *TimePeriodAchievementsTest*, are available in the *activity-service* module in package *test.java.cz.cvut.masters.service.activity.achievement*.

5.2.3 Integration Tests

The tests in the *AchievementServiceIntegrationTest* class use the real achievements *Service* layer as well as the unlocked achievements *Repository*. The interface for communicating with the **Authentication service** is mocked, and responses always return *200 OK* status. To verify correct behavior, the scenarios insert some achievements and activities with various attributes and check that the *Service* returns correct data. Both tests and configuration can be found in the test package of the *activity-service* module under *cz.cvut.masters.activity.integration*.

5.3 Manual Testing

In order to test authentication through Strava, I prepared a few scenarios that must be performed on a mobile device by a tester manually while using a real Strava account.

Strava login success—app. This scenario requires that the official Strava app with version 75.0 or later is installed on the device. First, the user launches the app and clicks on the Strava login button. The Strava app opens and prompts the user to authorize this app to connect to Strava. The user clicks on the *Authorize* button. To successfully complete the test, the login flow must finish and display the dashboard.

Strava login success—web. In this scenario, it is required that the Strava app is not available on the device. The scenario is identical to the preceding one, except that the Strava authorization screen opens in a browser.

Strava login unauthorized—app. Again, the Strava app on the device is required for this test. The user goes through the usual flow and is redirected to the Strava authorization screen. Here, they decide not to authorize this app for connection to Strava and click the *Cancel* button. The expected result of this action is that the user is redirected back to the login screen and an error message is displayed. The login can be repeated again.

Strava login unauthorized—web. The Strava app must not be installed on the mobile device. This scenario is similar to the one above, except that the *Cancel* button is clicked in a browser.

5.4 Performance

To test the performance of the system under load, and to demonstrate the scalability of the *microservices* architecture, I prepared a special testing scenario. The backend was deployed on a local machine in the *Minikube Kubernetes* cluster. The machine had the following configuration:

- CPU: Intel Core i7-6700HQ (2.60 GHz),
- RAM: 16 GB,
- Disk: NVMe Samsung MZVPV512,
- Operating system: Windows 10 Pro.

The *Minikube* virtual machine had available 4 GB of RAM and 2 virtual processors. It ran on the Hyper-V hypervisor for optimal performance [78].

The deployment followed the description provided in 4.2.7. The only difference from an usual testing environment were the resource limits for the **Activity service**. I placed a limit of 0.25 CPU for its container to simulate limited computational resources and to then allow scaling up the number of *replicas* in order to increase performance (horizontal scaling) [79].

The testing scenario involved performing HTTP GET requests on the */activities/data* path by a number of users. This path is served by the **Activity service**. The scenario was implemented in *Apache JMeter*, which is a *"100% pure Java application designed to load test functional behavior and measure performance."*[80]

Since the target API path requires authentication, the scenario starts by sending a login request. Just for testing purposes, I implemented password-based authentication. This authentication option returns the same response as the OAuth authentication flow (including an access token) after a successful login. So, the scenario first performs the login with a test user account and extracts the access token. This request is only performed once, and subsequent requests include the token. The scenario in question was set up with these basic parameters:

- number of threads (users): 50,
- loop count: 20.

So, in total, there were 1000 request sent to the backend. As previously described, requests are received on the **Gateway**, then forwarded to the

Activity service. Since there can be multiple *replicas* of a single container, *Kubernetes* defines a special object called *Service*, which abstracts access to the underlying *Pods* [81]. *Pod* contains the specific containers [82]. For this backend, it means a single *replica* per *Pod*. *Kubernetes* automatically load balances requests between different *Pods* behind a single *Service*. In short, this means that the incoming requests are distributed across all the *replicas* of the **Activity service**.

The test scenario with the parameters described above was performed on the cluster with 1, 2, and 4 *replicas* of the **Activity service**. Several warm-up runs were performed first. For each of the requests, the analyzed metric was latency, which measures the time *"from just before sending the request to just after the first response has been received."*[83] The results are summarized in table 5.1. The graph in figure 5.1 shows the cumulative average of the latency over time for different number of *replicas*.

Number of replicas	Average latency
1	967 ms
2	591 ms
4	318 ms

Table 5.1: Performance testing results.

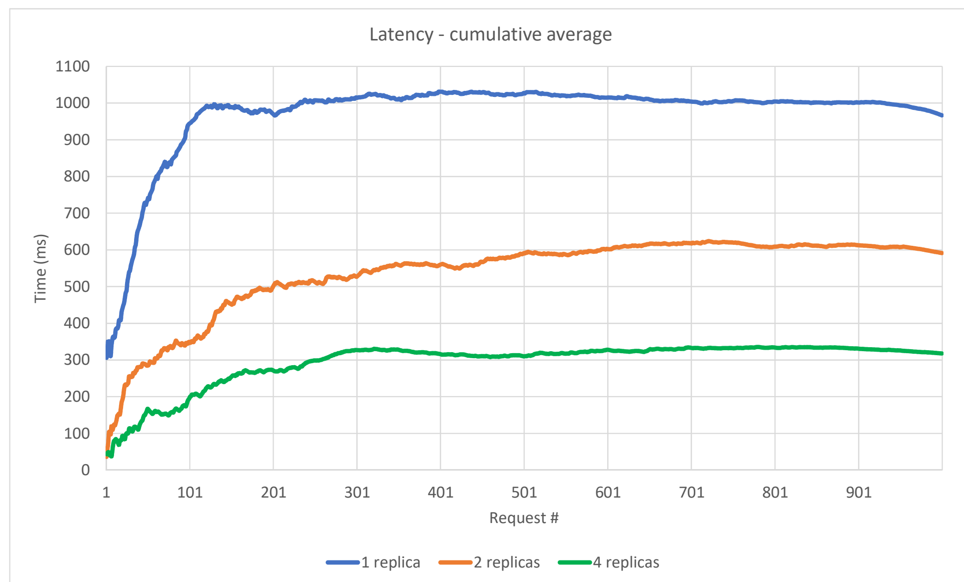


Figure 5.1: Latency for different number of service instances.

The results show that the system can be efficiently scaled up by increasing the number of instances of the microservices.

Chapter 6

Conclusion

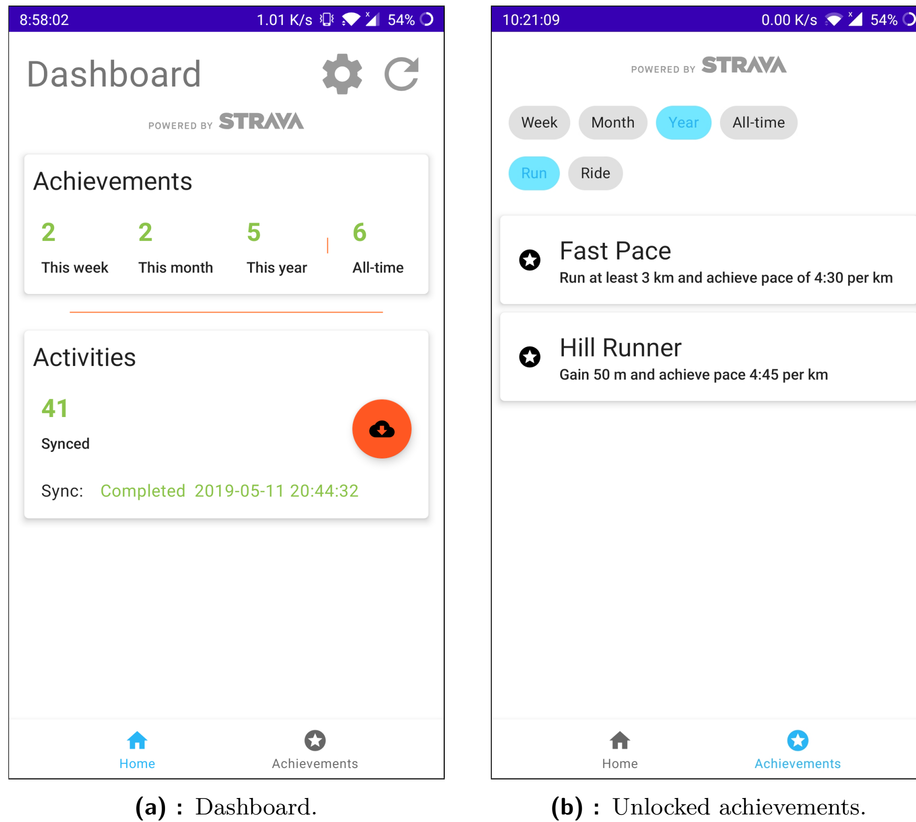
In the first chapter, I stated the goal of creating an application that provides functionality envisioned in 1.2. I started by analyzing similar apps on the market. Based on that, I placed some criteria on my solution. I defined the requirements and identified entities in the domain. The Design chapter showed three major parts of the system, namely the mobile app, backend and database. I discussed the architecture choices and various components for the mobile app and the backend. Then, I created the app and provided some implementation details. The mobile app uses modern technologies and utilizes caching of data. The backend consists of independent services. I worked with multiple deployment environments, those were detailed as well. Finally, the system was tested in a number of scenarios, and I also demonstrated its scalability.

All three main requirements defined in 2.1.4 have been addressed.

Provide a clear user interface. As evident both from the Design chapter and the implemented mobile app, the final user interface does satisfy this requirement.

Focus on periodic achievements. Achievements can be unlocked in four different time periods. There are weekly, monthly, annual and all-time achievements available.

Consider multiple activity attributes for a single achievement. The system for defining achievements is flexible and allows combining multiple attributes of a sport activity. See *FR-09* with a list of achievements that were defined for this thesis.



(a) : Dashboard.

(b) : Unlocked achievements.

Figure 6.1: Mobile application—screenshot.

6.1 Implemented Application

Sportivator runs on Android, and it can be distributed through the *Google Play* store [84]. To make the sign-up process simple, users are prompted to log in with their Strava account. Once they grant access to their data, they are presented with the main screen of the app (dashboard). Through a dedicated button, users can request synchronization of their Strava activities. After those activities are imported and processed, Sportivator checks for achievements that can be unlocked by the imported activities. The number and the complete list of unlocked achievements for each time period are displayed. When new activities are recorded on Strava, users can repeat the process by requesting a new activity sync. Figure 6.1 shows a screenshot of the main screen of the mobile app.

6.2 Future Work

While the created application is a fully-functional product, there are ways in which it could be further improved. It should be possible, thanks to the design considerations, to add another sport activity data providers to the app.

The system works with metric units, but support for imperial system is something that should be included. This goes hand in hand with different time zones, and the fact that week might start on different days in different countries. The app should probably recognize the location of the user in the world, and based on that, adjust when the time periods for unlocking achievements begin. It requires additional planning for situations like traveling between time zones. This might also include support for user personalization. Sportivator can be easily translated to other languages as all text is kept outside the application code.

To make the achievements more appealing, additional information could be displayed to the user. As an example, the number of times an achievement was unlocked might be included in its detail. It might be useful to show the complete list of achievements so the user knows what to aim for. If it can be determined which activities contributed to unlocking an achievement, they could be listed in the unlock detail.

When the athlete updates one of their past activities, the app does not currently reflect that. Also when there are activities imported to Strava with a start date before the date of the last successful synchronization, they are not imported. Therefore, in such cases, there could be a strategy implemented for resynchronization. The list of achievements for this thesis was created mostly to showcase the capabilities of the system. The door is open for defining additional ones.

Bibliography

[1] Strava Training: Track Running, Cycling & Swimming – Apps on Google Play. [online], [Accessed: 2019-04-28]. Available from: https://play.google.com/store/apps/details?id=com.strava

[2] Matched Runs – Strava Support. [online], [Accessed: 2019-04-28]. Available from: https://support.strava.com/hc/article_attachments/360005933332/IMG_0903.PNG

[3] Welcome to VeloViewer! [online], [Accessed: 2019-04-28]. Available from: https://cf.veloviewer.com/img/veloviewer-strava-summary-homepage.png

[4] Garmin Connect™– Apps on Google Play. [online], [Accessed: 2019-04-28]. Available from: https://play.google.com/store/apps/details?id=com.garmin.android.apps.connectmobile

[5] Guide to app architecture | Android Developers. [online], [Accessed: 2019-05-01]. Available from: https://developer.android.com/topic/libraries/architecture/images/network-bound-resource.png

[6] Strava Developers. [online], [Accessed: 2019-04-23]. Available from: https://developers.strava.com/

[7] Strava Apps – There's one for every athlete. [online], [Accessed: 2019-04-23]. Available from: https://www.strava.com/apps

[8] Strava. [online], [Accessed: 2019-05-13]. Available from: https://www.strava.com/

[9] What's a segment? – Strava Support. [online], [Accessed: 2019-04-28]. Available from: https://support.strava.com/hc/en-us/articles/216917137-What-s-a-segment-

[10] Strava Challenges – Strava Support. [online], [Accessed: 2019-04-28]. Available from: https://support.strava.com/hc/en-us/articles/216919177-Strava-Challenges

[11] Matched Runs – Strava Support. [online], [Accessed: 2019-04-28]. Available from: https://support.strava.com/hc/en-us/articles/216918597-Matched-Runs

[12] About | VeloViewer. [online], [Accessed: 2019-04-28]. Available from: https://blog.veloviewer.com/about/

[13] rubiTrack. [online], [Accessed: 2019-05-18]. Available from: https://www.rubitrack.com/index.html

[14] rubiTrack. [online], [Accessed: 2019-05-18]. Available from: https://www.rubitrack.com/features.html

[15] WeFitter. [online], [Accessed: 2019-05-18]. Available from: https://wefitter.com/

[16] Corporate Wellness – WeFitter. [online], [Accessed: 2019-05-18]. Available from: https://wefitter.com/solutions/corporate-wellness/

[17] WeFitter Success Cases. [online], [Accessed: 2019-05-18]. Available from: https://wefitter.com/customer-success/

[18] Wellness Cities. [online], [Accessed: 2019-05-18]. Available from: http://wellnesscities.eu/

[19] Garmin Connect. [online], [Accessed: 2019-04-28]. Available from: https://connect.garmin.com/

[20] Garmin Connect Badge Achievements | Garmin Support. [online], [Accessed: 2019-04-28]. Available from: https://support.garmin.com/en-US/?faq=6pECo6UIFn7ergw8kNmfu9

[21] Lethbridge, T.; Laganière, R. *Object-oriented software engineering*. McGraw-Hill, second edition, 2005, ISBN 9780077109080.

[22] OAuth 2.0 – OAuth. [online], [Accessed: 2019-04-15]. Available from: https://oauth.net/2/

[23] Guide to app architecture | Android Developers. [online], [Accessed: 2019-04-14]. Available from: https://developer.android.com/jetpack/docs/guide

[24] Nurkiewicz, T.; Christensen, B. *Reactive programming with RxJava*. O'Reilly Media, first edition, 2016, ISBN 9781491931653.

[25] JSON. [online], [Accessed: 2019-05-02]. Available from: https://www.json.org/

[26] Newman, S. *Building Microservices*. O'Reilly Media, first edition, 2015, ISBN 9781491950357.

[27] Microservices architecture style – Azure Application Architecture Guide. [online], [Accessed: 2019-01-10]. Available from: `https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices`

[28] Firebase Cloud Messaging. [online], [Accessed: 2019-05-13]. Available from: `https://firebase.google.com/docs/cloud-messaging`

[29] JSON Web Tokens – jwt.io. [online], [Accessed: 2019-04-15]. Available from: `https://jwt.io/`

[30] Strava Developers. [online], [Accessed: 2019-04-15]. Available from: `https://developers.strava.com/docs/authentication/`

[31] Download Android Studio and SDK tools. [online], [Accessed: 2019-05-01]. Available from: `https://developer.android.com/studio`

[32] Kotlin and Android. [online], [Accessed: 2019-04-04]. Available from: `https://developer.android.com/kotlin`

[33] Kotlin for Android – Kotlin Programming Language. [online], [Accessed: 2019-04-04]. Available from: `https://kotlinlang.org/docs/reference/android-overview.html`

[34] Navigation | Android Developers. [online], [Accessed: 2019-05-02]. Available from: `https://developer.android.com/jetpack/androidx/releases/navigation`

[35] Room Persistence Library | Android Developers. [online], [Accessed: 2019-05-02]. Available from: `https://developer.android.com/topic/libraries/architecture/room`

[36] Retrofit. [online], [Accessed: 2019-05-02]. Available from: `https://square.github.io/retrofit/`

[37] ReactiveX/RxJava. [online], [Accessed: 2019-05-02]. Available from: `https://github.com/ReactiveX/RxJava`

[38] ReactiveX/RxKotlin. [online], [Accessed: 2019-05-02]. Available from: `https://github.com/ReactiveX/RxKotlin`

[39] Dagger. [online], [Accessed: 2019-05-02]. Available from: `https://google.github.io/dagger/`

[40] JakeWharton/ThreeTenABP. [online], [Accessed: 2019-05-02]. Available from: `https://github.com/JakeWharton/ThreeTenABP`

[41] JakeWharton/timber. [online], [Accessed: 2019-05-02]. Available from: `https://github.com/JakeWharton/timber`

[42] Fragment | Android Developers. [online], [Accessed: 2019-05-02]. Available from: https://developer.android.com/jetpack/androidx/releases/fragment

[43] Activity | Android Developers. [online], [Accessed: 2019-05-02]. Available from: https://developer.android.com/reference/android/app/Activity

[44] ViewModel Overview | Android Developers. [online], [Accessed: 2019-05-02]. Available from: https://developer.android.com/topic/libraries/architecture/viewmodel

[45] SharedPreferences | Android Developers. [online], [Accessed: 2019-05-02]. Available from: https://developer.android.com/reference/android/content/SharedPreferences

[46] Intent | Android Developers. [online], [Accessed: 2019-05-02]. Available from: https://developer.android.com/reference/android/content/Intent

[47] Intents and Intent Filters | Android Developers. [online], [Accessed: 2019-05-02]. Available from: https://developer.android.com/guide/components/intents-filters

[48] Interceptors. [online], [Accessed: 2019-05-02]. Available from: https://github.com/square/okhttp/wiki/Interceptors

[49] Authenticator. [online], [Accessed: 2019-05-18]. Available from: https://square.github.io/okhttp/3.x/okhttp/okhttp3/Authenticator.html

[50] Spring Framework. [online], [Accessed: 2019-04-04]. Available from: https://spring.io/projects/spring-framework

[51] Web on Servlet Stack. [online], [Accessed: 2019-04-04]. Available from: https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html

[52] Spring Boot. [online], [Accessed: 2019-04-04]. Available from: https://spring.io/projects/spring-boot

[53] Apache Tomcat. [online], [Accessed: 2019-05-02]. Available from: http://tomcat.apache.org/

[54] Spring Cloud. [online], [Accessed: 2019-04-04]. Available from: https://spring.io/projects/spring-cloud

[55] Maven – Welcome to Apache Maven. [online], [Accessed: 2019-05-02]. Available from: https://maven.apache.org/

[56] Client Side Load Balancer: Ribbon. [online], [Accessed: 2019-05-02]. Available from: https://cloud.spring.io/spring-cloud-netflix/multi/multi_spring-cloud-ribbon.html

[57] Spring Cloud Sleuth. [online], [Accessed: 2019-05-02]. Available from: https://spring.io/projects/spring-cloud-sleuth

[58] Netflix/zuul. [online], [Accessed: 2019-05-02]. Available from: https://github.com/Netflix/zuul

[59] Spring Cloud OpenFeign. [online], [Accessed: 2019-05-02]. Available from: https://spring.io/projects/spring-cloud-openfeign

[60] Spring Security. [online], [Accessed: 2019-05-02]. Available from: https://spring.io/projects/spring-security

[61] Router and Filter: Zuul. [online], [Accessed: 2019-05-18]. Available from: https://cloud.spring.io/spring-cloud-netflix/multi/multi__router_and_filter_zuul.html

[62] Docker. [online], [Accessed: 2019-05-16]. Available from: https://www.docker.com/resources/what-container

[63] IntelliJ IDEA. [online], [Accessed: 2019-05-16]. Available from: https://www.jetbrains.com/idea/

[64] Spring Cloud Netflix. [online], [Accessed: 2019-05-16]. Available from: https://spring.io/projects/spring-cloud-netflix

[65] Kubernetes. [online], [Accessed: 2019-05-16]. Available from: https://kubernetes.io/

[66] Running Kubernetes Locally via Minikube. [online], [Accessed: 2019-05-16]. Available from: https://kubernetes.io/docs/setup/minikube/

[67] postgres – Docker Hub. [online], [Accessed: 2019-05-16]. Available from: https://hub.docker.com/_/postgres

[68] Google Kubernetes Engine. [online], [Accessed: 2019-05-16]. Available from: https://cloud.google.com/kubernetes-engine/

[69] Cloud SQL. [online], [Accessed: 2019-05-16]. Available from: https://cloud.google.com/sql/

[70] Connecting from Google Kubernetes Engine. [online], [Accessed: 2019-05-16]. Available from: https://cloud.google.com/sql/docs/mysql/connect-kubernetes-engine

[71] ActiveMQ. [online], [Accessed: 2019-05-16]. Available from: https://activemq.apache.org/

[72] rmohr/activemq – Docker Hub. [online], [Accessed: 2019-05-16]. Available from: https://hub.docker.com/r/rmohr/activemq/

[73] Espresso. [online], [Accessed: 2019-05-14]. Available from: https://developer.android.com/training/testing/espresso/index.html

[74] Spring Security – Testing. [online], [Accessed: 2019-05-15]. Available from: https://docs.spring.io/spring-security/site/docs/current/reference/html/test.html

[75] Testing the Web Layer. [online], [Accessed: 2019-05-15]. Available from: https://spring.io/guides/gs/testing-web/

[76] Spring Boot – Testing. [online], [Accessed: 2019-05-18]. Available from: https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-testing.html

[77] Clock (Java Platform SE 8). [online], [Accessed: 2019-05-18]. Available from: https://docs.oracle.com/javase/8/docs/api/java/time/Clock.html

[78] Introduction to Hyper-V on Windows 10. [online], [Accessed: 2019-05-17]. Available from: https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/about/

[79] Managing Compute Resources for Containers – Kubernetes. [online], [Accessed: 2019-05-17]. Available from: https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/

[80] Apache JMeter. [online], [Accessed: 2019-05-17]. Available from: https://jmeter.apache.org/

[81] Services – Kubernetes. [online], [Accessed: 2019-05-17]. Available from: https://kubernetes.io/docs/concepts/services-networking/service/

[82] Pods – Kubernetes. [online], [Accessed: 2019-05-17]. Available from: https://kubernetes.io/docs/concepts/workloads/pods/pod/

[83] Apache JMeter. [online], [Accessed: 2019-05-17]. Available from: https://jmeter.apache.org/usermanual/glossary.html

[84] Google Play. [online], [Accessed: 2019-05-17]. Available from: https://play.google.com/store

Acronyms

API Application Programming Interface

HTTP Hypertext Transfer Protocol

JAR Java ARchive

JSON JavaScript Object Notation

REST Representational State Transfer

SDK Software Development Kit

UI User interface

URI Uniform Resource Identifier

URL Uniform Resource Locator

VM Virtual machine

Appendix B

Contents of enclosed SD Card

```
readme.txt ............................ file with additional information
user_manual.pdf ....................... user manual for the app in PDF
src ................................... directory with source code
    backend ........................... source code of the backend
    mobile ............................ source code of the Android app
thesis ................................ directory with LaTeX-related files and the output
    img ............................... directory with images for the thesis
    _masters.tex ...................... the main LaTeX source code of the thesis
    masters_en.tex .................... setup LaTeX source code of the thesis
    masters_en.pdf .................... the thesis in PDF
    bibliography.bib .................. the bibliography file
```

Appendix C

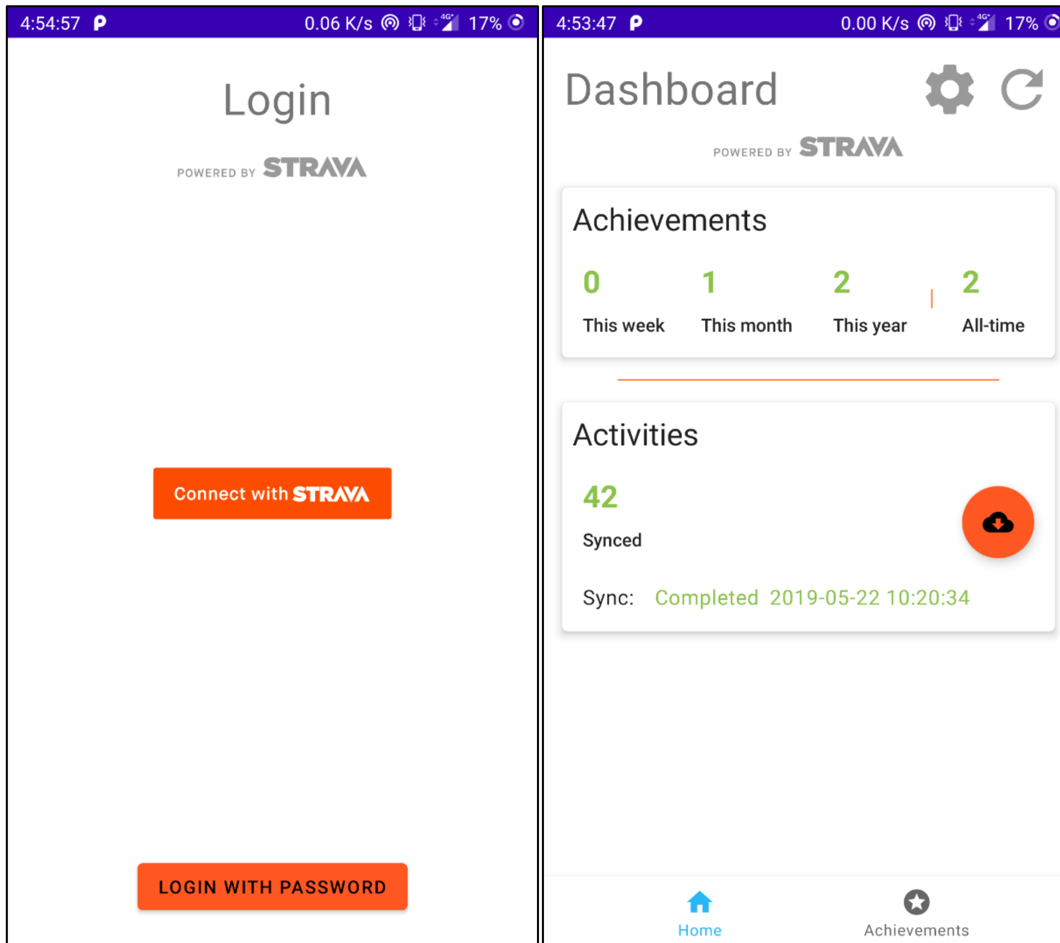
User Manual

Sportivator – User Manual

User Interface

Login Screen

The login screen is displayed on the first start of the application or whenever authentication is required. It hosts a button for authenticating through your Strava account.



Login screen and dashboard.

Bottom Navigation

The bottom navigation bar contains two clickable elements:

1. Home: navigates to the dashboard.
2. Achievements: navigates to the screen with unlocked achievements.

Dashboard

The dashboard contains two main areas with data:

1. **Achievements summary**: contains the current numbers of unlocked achievements for different time periods:
    a. current week,
    b. current month,
    c. current year,
    d. all-time.

2. **Activity-related data**:
   This section displays the following data:
   a. The current **number of synchronized activities**.
   b. The **state of the last synchronization request**:
      i. Never – never requested,
      ii. In progress – synchronization is in progress,
      iii. Completed,
      iv. Error – request finished with error.
   c. If the state of the last sync request is Completed, the section also displays the date and time of the completion of the request.

There are several buttons:

1. Synchronize activities button: orange circular button in the activity-related section.
2. Refresh button: button in the top-right corner for refreshing data.
3. Preference button: with the cog symbol, opens the preferences screen.

Achievements Screen

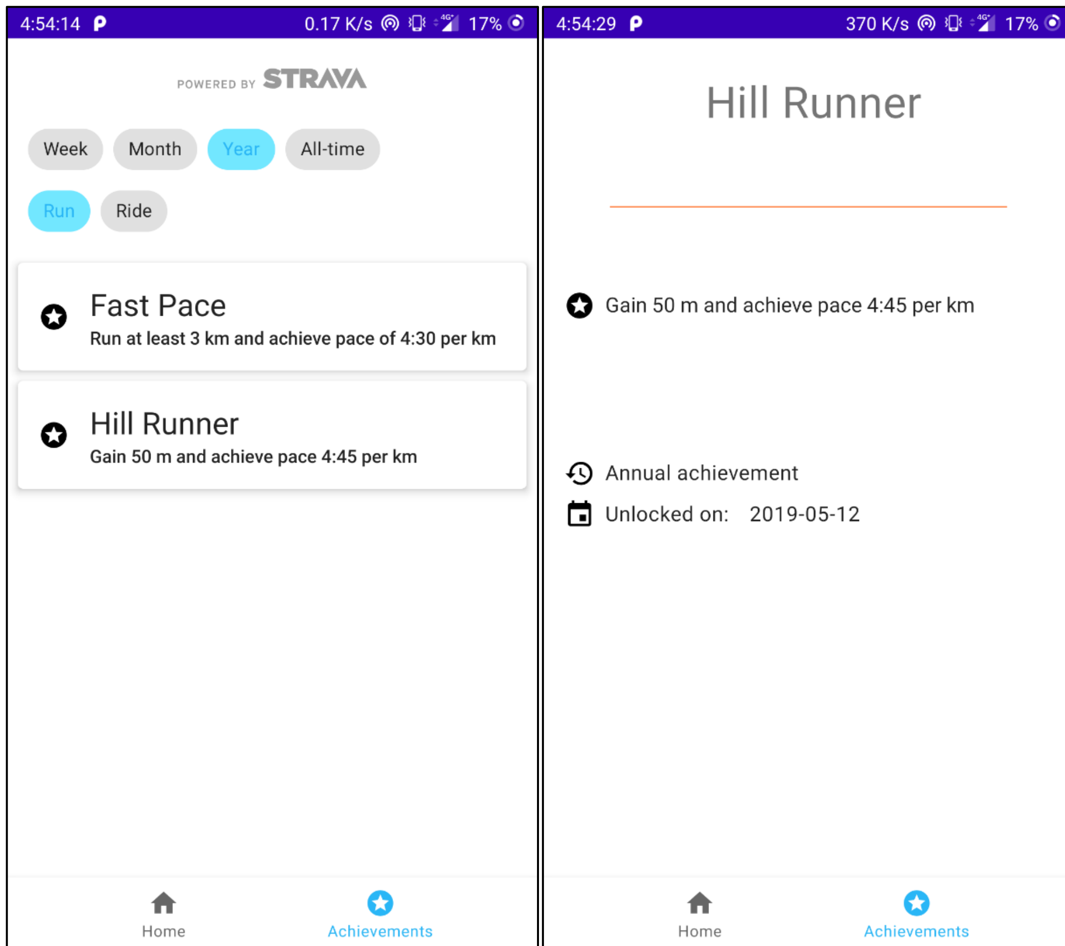
This screen hosts a **list of unlocked achievements**. The unlocks are filtered by the buttons in the top area of the screen. Each row of buttons represents a single type of filtering. The first row switches between unlocks in these **time periods**:

a. current week,
b. current month,
c. current year,
d. all-time.

The second row adds another filter combined with the previous one based on the type of **sport**:

a. run (running),
b. ride (cycling).

The active filters are highlighted, only a single button in each row can be active at a time. If there are no achievements available for the current combination of filters, a corresponding message is displayed instead of the list.



Unlocked achievements and achievement unlock detail screens.

Achievement Unlock Detail Screen

The screen with **details about a single unlock of an achievement** displays the following information:

a. name of the unlocked achievement,
b. its description,
c. the time period for which it was unlocked,
d. the date of the last unlock of the achievement for this time period.

Usage

Authenticating

To successfully authenticate, follow these steps:

1. Click on the **Connect with Strava** button on the login screen to open a Strava authorization screen.
2. If you have the official Strava app (version 75.0 or higher) installed on the device, the screen opens in the Strava app. Otherwise, the Strava Authorization screen opens in a browser. Either way, if prompted to login to Strava with your account, please do so.
3. Click on the Authorize button. This should follow by navigating back to Sportivator, finishing the login process and opening the main screen of the app – dashboard.

Synchronizing Activities

To synchronize new activities from Strava, click on the orange circular **sync button** on the dashboard. The current sync status gets updated. Once the sync is completed, the app automatically refreshes the data, including the unlocks of achievements.

Refreshing Data

Data can be also refreshed manually, if needed, by clicking on the **refresh button** in the top right corner of the dashboard.

Logout

If you want to logout from the app, click on the **preference button** (with the cog symbol) on the dashboard. When the preferences screen opens, click on the **Logout option**. This logs you out of the app and opens the login screen.

Display Unlocked Achievements

Navigate to the screen with unlocks by clicking on the navigation element with the text Achievements in the bottom navigation bar. Optionally, filter the unlocks by the filters described previously.

Display Achievement Unlock Detail

To display the detail of a single unlock of an achievement, click on one in the list with unlocks. This opens the screen with details for the selected unlock.