OULUN AMMATTIKORKEAKOULU

Niklas Poussu

STRAVA EXPLORE PROOF OF CONCEPT FOR POLAR BEAT ANDROID APPLI-CATION

STRAVA EXPLORE PROOF OF CONCEPT FOR POLAR BEAT ANDROID APPLICATION

Niklas Poussu
Bachelor’s Thesis
Spring 2020
Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Information Technology, Option of Device and Product Design

Author: Niklas Poussu
Title of thesis: Strava Explore Proof of Concept for Polar Beat Android Application
Supervisors: Kari Jyrkkä, Mika Mustonen
Term and year when the thesis was submitted: Spring 2020 Number of pages: 40

The subject of this bachelor’s thesis was to create a proof of concept feature to visualize nearby Strava Live Segments in a Google Map within the Polar Electro’s Beat Android application. Polar Electro Ltd. is a Finnish company that designs and manufactures sports training computers and various other heart rate measurement devices.

The proof of concept feature was implemented as an extension to the Polar Beat Android applications’ exercise feature where the user can track various exercise related metrics in real-time, such as the user’s heart rate, current speed and elapsed time.

Polar Beat is a multisport fitness Android application developed by Polar Electro Ltd that is designed to be used for athletes to train with a live heart rate and to track their training sessions. Polar Beat gives the athlete training analysis and progress reports.

Strava Live Segments is a feature developed by Strava for cyclists and runners to compete on a time-trial with on user-defined stretches of road or terrain, which Strava calls Live Segments. Strava Live Segments encourages competitive and casual athletes alike to beat their records and reach a higher level of fitness.

The implementation consisted of building a new feature capable of fetching data for Strava Live Segments from the Strava REST API and transforming the data in order to visualize nearby Strava Live Segments in a Google Map. The new feature was built using the MVVM software architectural pattern and the Retrofit HTTP client was setup to enable HTTP communications and authentication to the Strava API.

The result was a proof of concept feature capable of visualizing nearby Strava Live Segments in a Google Map and showing details about a segment by selecting a segment in the Google Map.

Keywords: strava live segments, mobile applications, mvvm, proof of concept, software development

TERMS AND ABBREVIATIONS

API	Application Programming Interface, interface for exchanging data via a network
CR	Course Record, marks the fastest time in Strava Live Segment leaderboards
Gradle	Software plugin for compiling, building and packaging Android applications
KOM	King of the Mountain a male holding a record time in a Strava Live Segment
MVVM	Model-View-ViewModel, software architectural pattern
QOM	Queen of the Mountain, female holding a record time in a Strava Live Segment
REST	Representational state transfer, HTTP-protocol based architectural model for developing APIs
Strava Live Segments	Service provided by Strava for cyclists to compete on cycling segments
UI	User interface, the platform the user controls an application or device
Unit tests	Software tests to validate correct behaviour of a piece / unit of code
URI	Uniform Resource Identifier, string of characters that identifies a resource over a network

TABLE OF CONTENTS

1 INTRODUCTION

This Bachelor's Thesis was commissioned by Polar Electro which, amongst other things, manufactures heart rate monitors, activity trackers and various exercise and health related software. One Polar Electro's software component is the Polar Beat sports and fitness tracking Android application, which is extended in this thesis.

The aim of this thesis was to implement the Strava's Live Segments "Explore" feature in the Polar Beat Android application. The Explore feature enables the user search geographically nearby cycling or running routes in a map. Strava offers a paid feature called "Live Segments" on their platform, which enables the users to train on these routes in a competitive manner. The aim in this thesis was to bring the functionality to locate Live Segments while doing an outdoor exercise in Polar Beat.

The implementation consists of extending the Polar Beat's exercise tracking feature by adding the possibility to locate nearby or favourite Strava Live Segments on a Google Map while doing an exercise in Beat. The user can check important details about a segment, such as segment distance, grade and route. This is convenient since the user can locate nearby segments on real-time, without stopping the exercise.

2 PLATFORM AND ARCHITECTURE

2.1 Polar Beat

Polar Beat is a free sports and fitness tracker application developed by Polar Electro Oy. It is available for Android and iOS-devices and is designed to be used for users to train with a live heart rate and to track their training sessions. Beat enables the user to track various training metrics during exercise in real time, such as heart rate, location, speed and calorie expenditure during exercise. Beat is compatible with third-party heart rate sensors in addition to Polar devices (14.)

Polar Beat serves as the base for the new feature introduced in this thesis project. The main feature in Beat is its exercise tracking feature which enables the user to track various exercise related metrics such as the distance travelled during the exercise and the heart rate in real-time. The picture in Figure 1 shows a snapshot of a user using the exercise feature. Location tracking is based on GPS, and it is shown on the map as a route travelled with a red line.

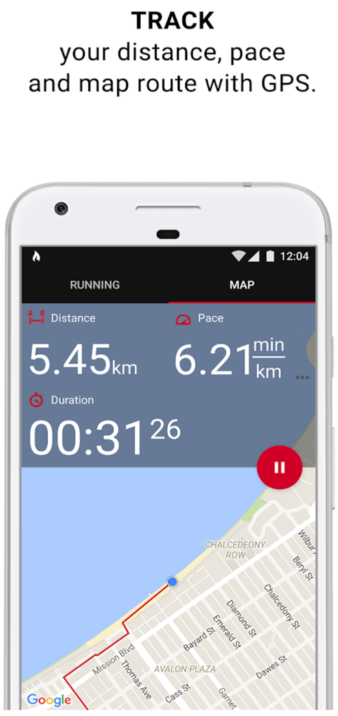


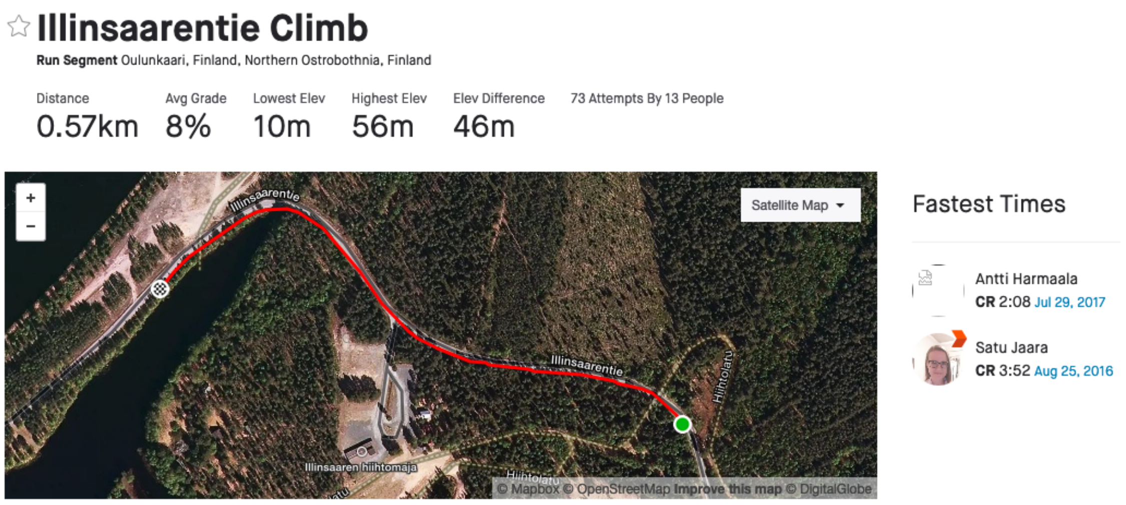
FIGURE 1. Example of a user running the exercise feature in Polar Beat

2.2 Strava

2.2.1 Strava Live Segments

Strava Live Segments is a feature developed by Strava for cyclists and runners to compete on a time-trial with on user-defined stretches of road or terrain, which Strava calls Live Segments. A user using the Live Segments feature can see in real time one's current time in comparison to one's personal record or the segments record holder. This motivates the user to exercise and beat personal records in a competitive way.

Figure 2 shows a snapshot from Strava's website, showcasing one Strava Live Segment details. The overview shows the user some important information about the segment, such as distance, grade (e.g. average change of altitude, positive grade meaning altitude gain, negative altitude lowering) and elevation statistics. The Segment record (CR) is also shown for both men and women.



☆ Illinsaarentie Climb

Run Segment Oulunkaari, Finland, Northern Ostrobothnia, Finland

Distance	Avg Grade	Lowest Elev	Highest Elev	Elev Difference	73 Attempts By 13 People
0.57km	8%	10m	56m	46m	

Fastest Times

Antti Harmaala
CR 2:08 Jul 29, 2017

Satu Jaara
CR 3:52 Aug 25, 2016

FIGURE 2. Strava Live Segment details in Strava's website

2.2.2 Execution of a Strava Live Segment

The execution of the segment can be tracked using different devices such as sport watches and cycling computers between different vendors, such as the Polar Vantage V. Segments can also be run with the Strava mobile application (figure 3). The basic principle of completing a segment is the same and goes as follows:

1) The device tracks the user's location and notifies when a segment start is approached.
2) When the segment starting line is crossed, time starts
3) When the segment is started the user is notified and the time-trial begins. The user can see in real-time their current time in comparison to their personal record and possibly a segment record.
4) When the user crosses the finish line, the segment is finished and time tracking is stopped.
5) Final performance results are shown for the user.

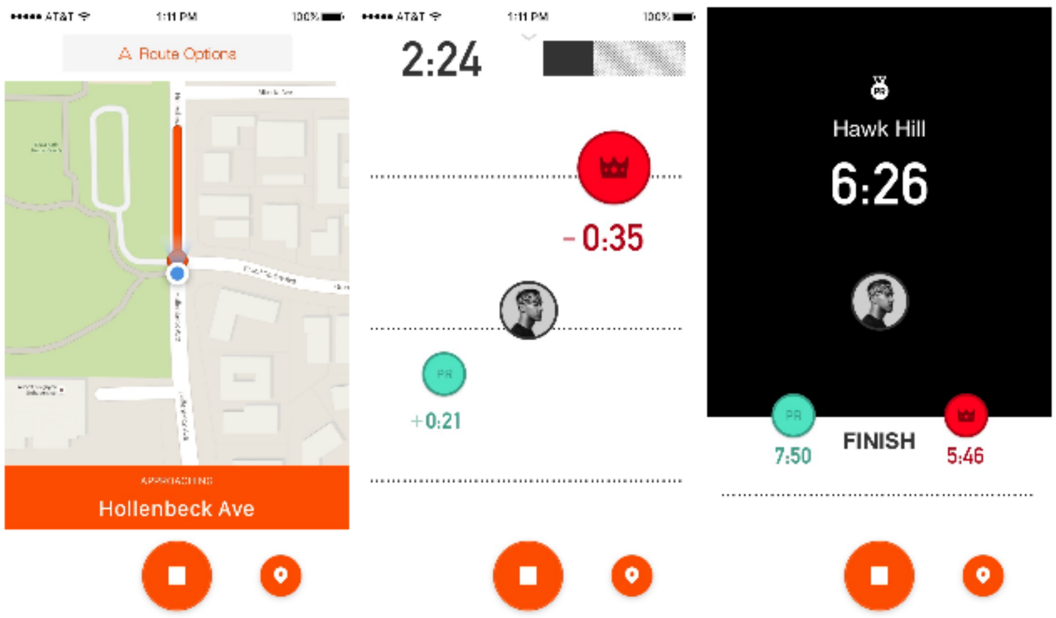


FIGURE 2. Segment execution in Strava Android app

2.3 System architecture

The system consists of a heart rate sensor (Polar H10), a mobile device running the Polar Beat Android application, the Strava REST API and the Polar web service. The Polar H10 tracks the user's heart rate and sends the heart rate data via Bluetooth to the mobile device running Polar Beat.

The mobile device acts as a central hub in the system. It runs the Polar Beat application and is responsible for communicating to the heart rate sensor, the Strava REST API and finally the Polar web service. The mobile device is also responsible for tracking the GPS location. The Strava REST API provides endpoints for fetching data from the Strava ecosystem, in this case data containing information about the user's Strava Live Segments.

Finally, in the system the Polar Flow web service is responsible for storing the user's training sessions. The Flow web service also handles submitting the Strava Live Segment results to Strava when the user synchronizes the mobile device with the Flow web service after a training session is finished. The full system architecture is described in figure 4.

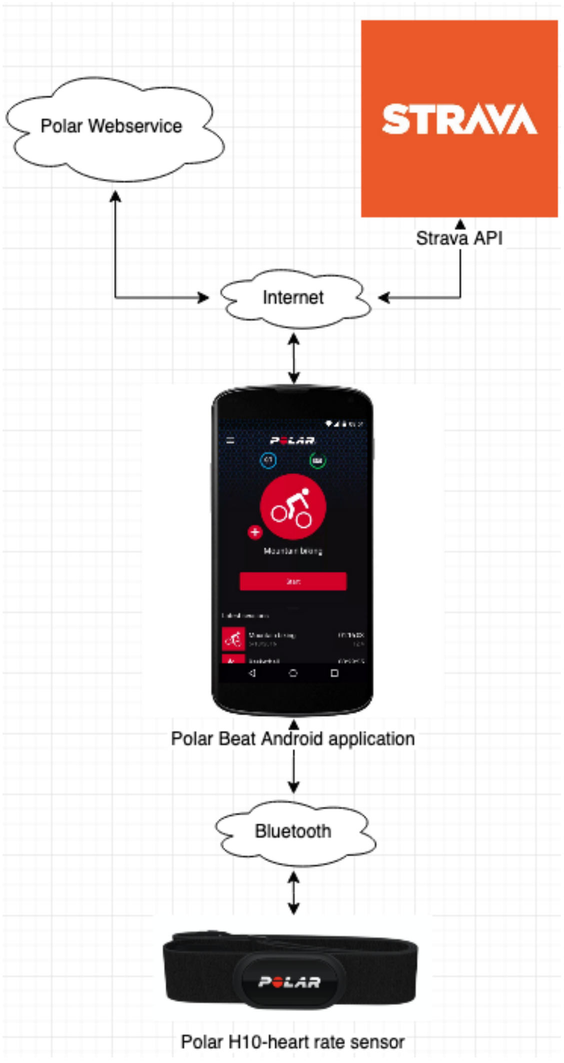


FIGURE 3. Architectural overview of the different components in the system

2.4 Programming language, tools and methodologies used

The new feature to Polar Beat was developed using the Android Studio IDE and Java and Kotlin programming languages. Postman was used to test REST queries and inspect responses. Postman was also used to aid developing Retrofit interfaces responsible for executing REST queries and to create data models matching the REST responses in Kotlin. RxJava was used for asynchronous computation.

2.4.1 Android Studio

Android Studio is the official integrated development environment (IDE) for Android development developed by JetBrains, based on the IntelliJ IDEA. Android Studio provides some features on top of the IntelliJ IDEA platform, specifically catered for Android development, such as the Gradle-build system, support for Android emulation and extensive testing tools and frameworks. (5.)

2.4.2 Programming language

The programming languages used in this project are Java and Kotlin, which both are JVM-based programming languages (Java Virtual Machine) that compile into Java bytecode. The Java-programming language is a general-purpose object-oriented programming language first introduced by Sun Microsystems in 1995. (2.) It is one of the three officially supported programming languages for Android development, the others being Kotlin and C++. The Kotlin programming language replaced Java as the main programming language for Android development in 2019. (1.) Kotlin is fully interoperable with Java, for example Kotlin code can use Java methods and vice versa.

Kotlin is a programming language heavily inspired by Java and developed by JetBrains. Kotlin brings some modern programming language features to Android such as lambda expressions and extensions functions. Kotlin's type system is null-safe by default which reduces the chances of the dreaded null reference exception, or in Java, a NullPointerException, the source for many errors. (3.)

For the reasons above the main programming language for this project is the Kotlin programming language. In Polar Beat there is a lot of previous code written in Java, in these scenarios Java is used to extend the existing functionalities.

2.4.3 MVVM

MVVM (Model-View-ViewModel) is a software architectural design pattern. MVVM heavily utilizes the Observer pattern, and it enables a clear separation of concerns between different logical units of a software. (8.) This leads to many benefits such as greater testability and code reuse. There are three core components in the MVVM pattern, the Model, the ViewModel and the View. (7.)

In MVVM the View is responsible for rendering UI components and handling user actions. The View receives any needed data from one or multiple ViewModels by observing the properties or functions in the ViewModel.

The ViewModel is a source of data for the View. It prepares data from a Model, and possibly processes it, ready for the View to consume. The ViewModel does not have a reference to the View, it is simply a source of data for the View and can be unit tested in isolation. The ViewModel contains observable data fields, which the View can subscribe to receive updates whenever the ViewModel emits data.

Finally, the Model acts as a source of data for the ViewModel in the same way as the ViewModel acts for the View. The Model may contain references to entities, such as a database, a local storage or a remote API.

2.4.4 Postman

Postman is an API Development software provided by Postman Inc. Its API Client tool provides functionalities to create and execute complex REST queries within the software and to inspect responses. Postman provides access to APIs with many different authentication protocols, such OAuth and other JWT-based authentication protocols. (6.)

2.4.5 Retrofit

Retrofit is a type-safe HTTP client for Android. Retrofit can be used to develop a Java/Kotlin interface, which is used for defining the different attributes and parameters of a HTTP request. Retrofit creates an implementation of the interface and offers functions to execute HTTP requests either synchronously or asynchronously. (12.)

2.4.6 RxJava

RxJava is a Java VM implementation of Reactive Extensions which is a library for composing asynchronous and event-based programs by using observable sequences. RxJava extends the Observer pattern and provides observable data sources and operators for composing logical sequences together. RxJava provides abstractions and handles for threading and thread-safety. (9.)

The efficient usage of threading and concurrency is important for any application with intensive operations or importance with performance, but it is especially important in Android. Android applications run on a single thread on default, called the main thread or the UI thread. Operations, such user inputs, system callbacks and drawing user interfaces are responsibilities of the UI thread. Intensive and long-running operations, such as querying a database can last for seconds. If the work is done in the UI thread, the application is not responsive. This leads to a bad user experience. Therefore, intensive operations should always be executed on a separate thread. (10.)

There are many different options for multithreading in Android, such as AsyncTasks, Java Threads and lately introduced Coroutines, but RxJava was chosen for this project for its inherent functional reactive nature.

3 IMPLEMENTATION

3.1 Planning the feature

The implementation began on planning the feature and where changes needed to be made in the software. The Google Map in the Exercise-view lies in the ExerciseMapFragment class. Fragment classes in Android represent a portion of user interface. (11.) Any View-related code regarding drawing in the Google Map, therefore, should be used here. The business logic should be encapsulated in ViewModels / Models.

The aim is to draw markers on the map representing a start of a segment, and to draw the route on the map. By clicking a marker, the user should be able to check relevant details of a segment, such as the distance. Additionally, there should be a way to hide the segments on-demand. These are all new UI components that need to be developed. Figure 5 contains the user interface of the Exercise-view, which serves as a starting point for new development.

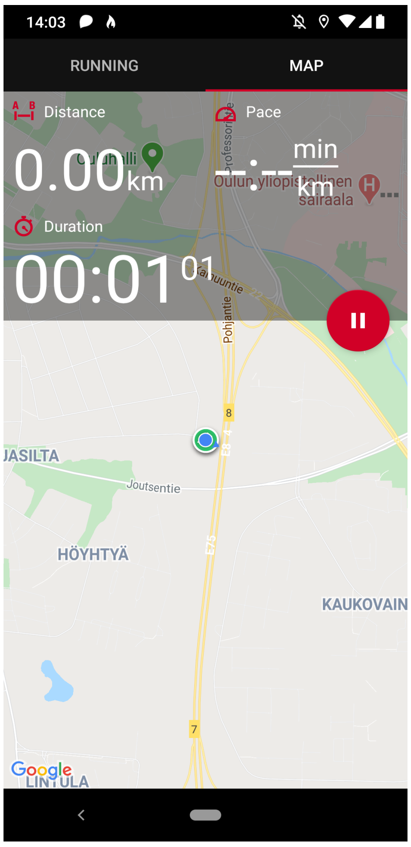


FIGURE 4. User interface of Exercise-view before modifications

3.2 Software architecture breakdown

Figure 6 contains an architectural overview of the new feature built in this Thesis using MVVM. Inside the system boundaries orange colour signifies an existing class that is modified. Green colours signify new classes. The arrows represent references to other entities and the direction represents a dependency. For example, StravaViewModel has dependencies to StravaService and LocationRepository.

All dependencies are injected using a manual dependency injection. Dependency injection (DI) is a technique where references to other classes are provided externally, in the case of manual dependency injection, references are created “by hand”, in Java by using the “new” keyword and then stored in some global context. A common way of dependency injection is a constructor injection, where a dependency is provided when an object is constructed. (17.) The constructor injection was utilized in this project.

BeatApp-class is an Application class and contains a reference to BeatModule-class. Application classes in Android are the entry points for the app and maintain a global application state. Modules that should be globally accessible can be stored in such a class. Here the BeatModule-class is stored in BeatApp-class, which holds references to all references to all Model-classes stated in Figure 6. BeatModule utilizes the Singleton pattern (35.) The ViewModels and the Views are created on-demand. This ensures that the created object instances are shared and can be used globally. For example, there could be many parts of the application needing data from the Strava API, therefore, utilizing the StravaService class in multiple parts of the application.

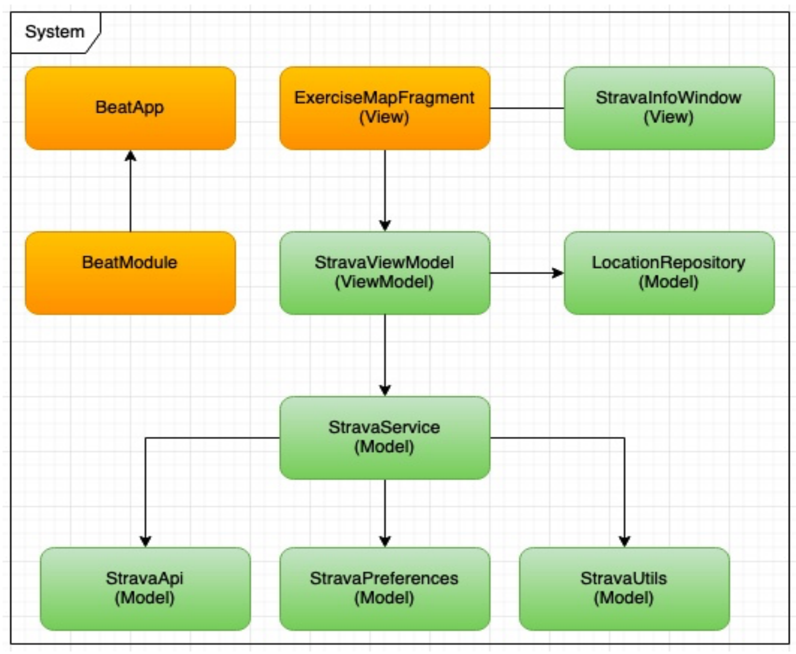

System
BeatApp
ExerciseMapFragment (View)
StravaInfoWindow (View)
BeatModule
StravaViewModel (ViewModel)
LocationRepository (Model)
StravaService (Model)
StravaApi (Model)
StravaPreferences (Model)
StravaUtils (Model)

FIGURE 5. MVVM-based architectural diagram of the new feature

In Figure 7 there are four components depicted. Retrofit is a REST client which uses OkHttp-library internally for HTTP requests. BeatHttpClient is a custom implementation of OkHttp. LocationRepository wraps a FusedLocationProvider that provides the GPS location of the device used and StravaService encapsulates all functionalities regarding communication to Strava over the network.

Encapsulation in object-oriented programming refers to hiding details or restricting access to a certain object's components. The encapsulated object may provide methods for interaction, whilst hiding the details behind it. (21.) StravaService is a class based on this principle, its role in this project is ultimately providing the data containing route and segment data for Strava Live Segments.

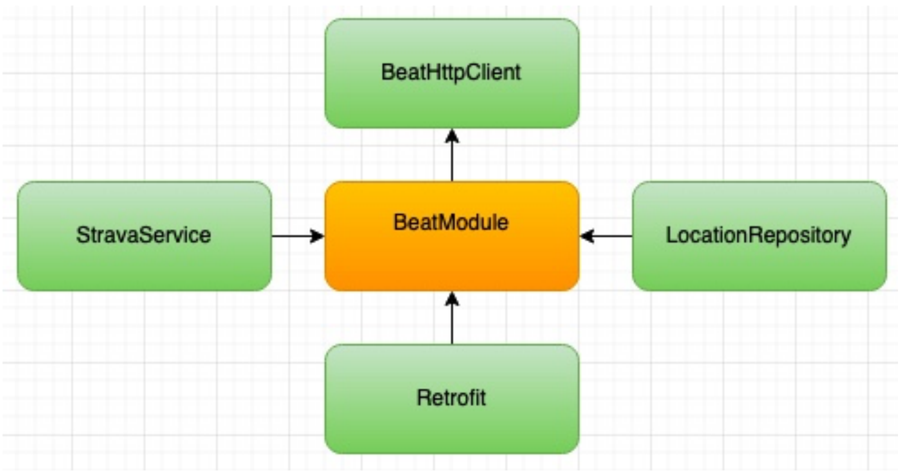
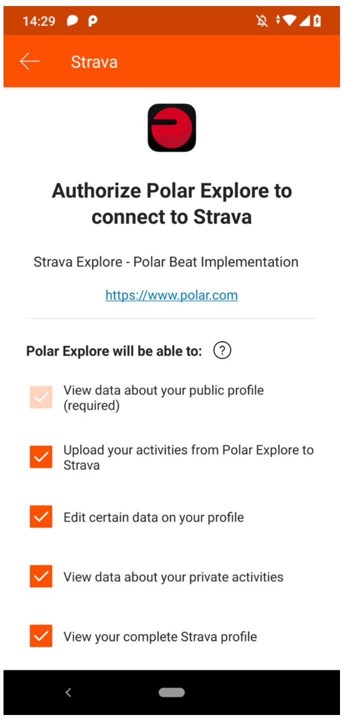


FIGURE 6. Architectural overview of the BeatModule-singleton and its references

3.3 Authentication to Strava API

3.3.1 Authentication format

In order to get an access token, the user needs to grant access using a 3rd-party method. This is typically a web page or some specific application. The user can select which permissions are granted. Figure 8 contains a screenshot of the OAuth authentication process using the Strava Android application to grant access to Strava's API.



Authorize Polar Explore to connect to Strava

Strava Explore - Polar Beat Implementation

https://www.polar.com

Polar Explore will be able to:

- ✓ View data about your public profile (required)
- ✓ Upload your activities from Polar Explore to Strava
- ✓ Edit certain data on your profile
- ✓ View data about your private activities
- ✓ View your complete Strava profile

FIGURE 7. Example of the OAuth-process in Strava Android app

After the user grants the permissions, an authentication token is returned. The authentication token can be used only once and is used to fetch an access token from an authorization server in addition to a refresh token. Access tokens are typically short-lived, the lifetime of the token depends on the service. In the case of Strava the access tokens are valid for 6 hours. Figure 9 contains a sequence diagram describing the OAuth authentication flow.

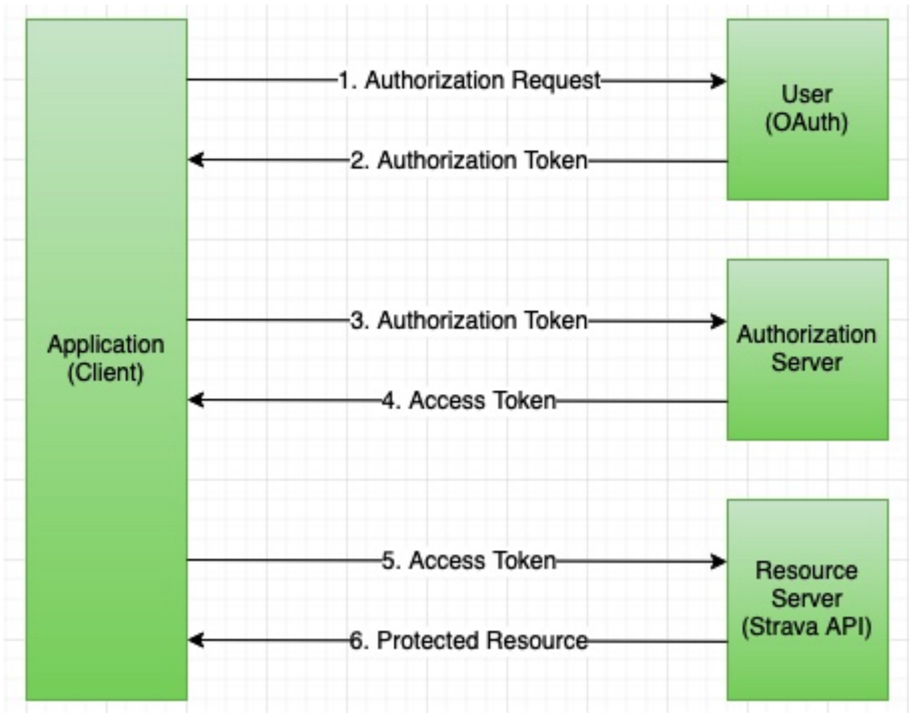
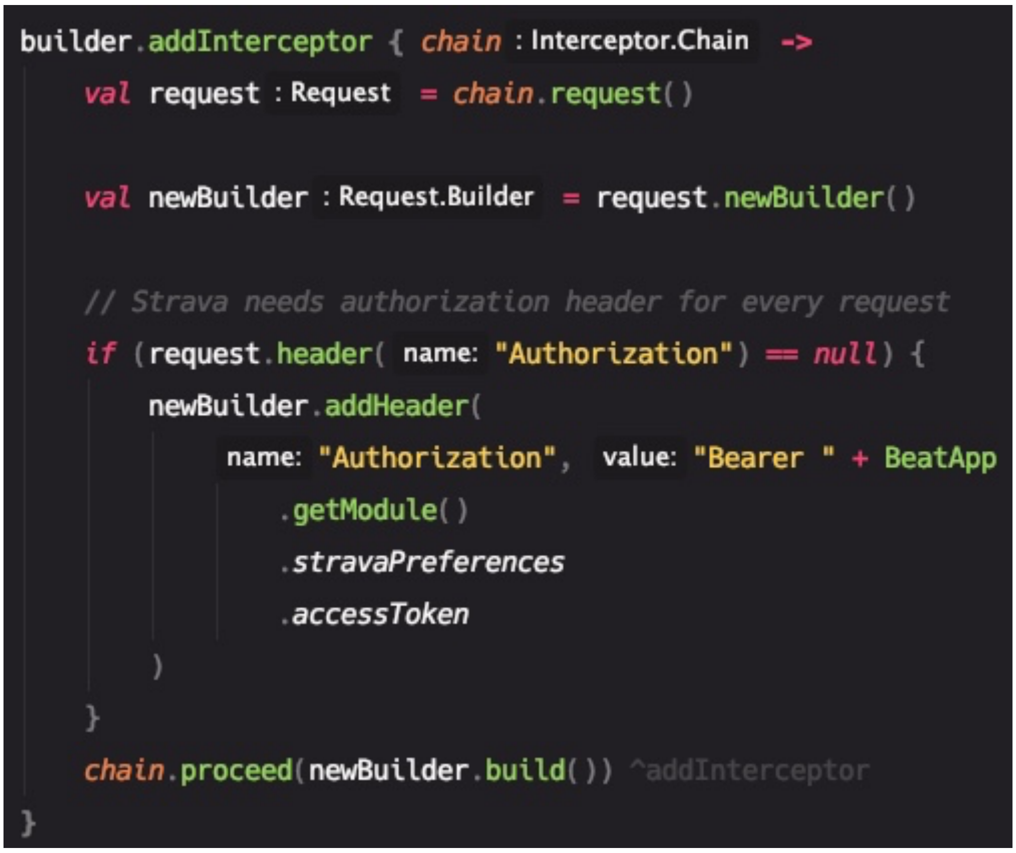


FIGURE 8. OAuth-authentication sequence diagram

After the access token has expired, a new access token can be fetched from the authorization server using a refresh token, essentially renewing the access token in behalf of the user. A refresh token is a token specifically for the purpose of renewing access tokens. Alternatively, the access token can be renewed by the user starting the OAuth process from the start if the refresh token has been invalidated by the resource server for example. Because the tokens are the keys and identity for the user, tokens should be kept in a secure storage.

3.3.2 Authorizing HTTP requests

Strava uses OAuth2 for authentication to their API. (13.) Every request to the Strava REST API needs an access token. An access token is a key to make requests on behalf of a user. In Figure 10 BeatHttpClient, which is a custom implementation of OkHttpClient, is configured with an HTTP request interceptor to intercept HTTP requests and add an authorization header to the request. The access token needed for the requests is stored in SharedPreferences and fetched from there in order to append the request (figure 10).

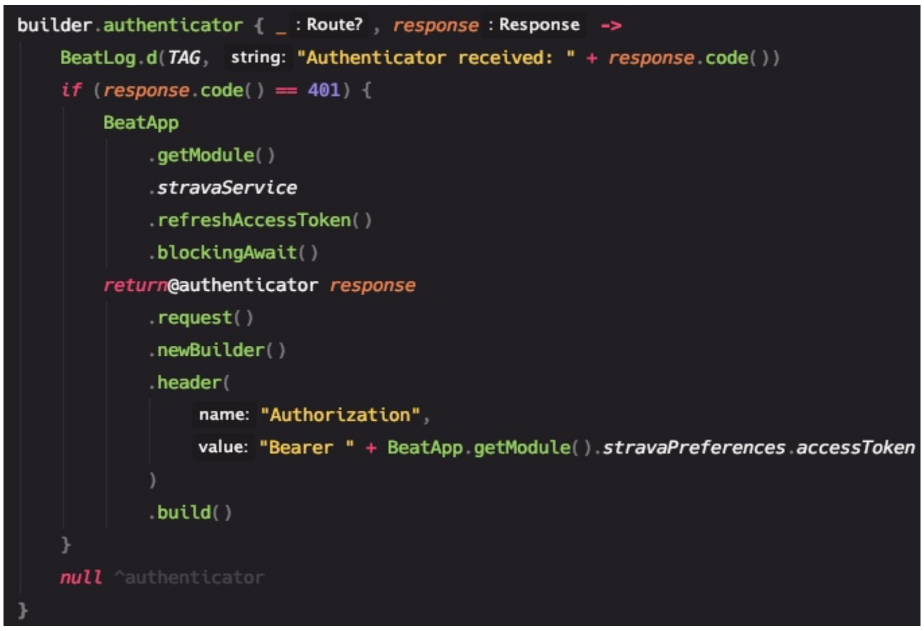


```
builder.addInterceptor { chain : Interceptor.Chain ->
    val request : Request = chain.request()

    val newBuilder : Request.Builder = request.newBuilder()

    // Strava needs authorization header for every request
    if (request.header( name: "Authorization") == null) {
        newBuilder.addHeader(
            name: "Authorization",  value: "Bearer " + BeatApp
                .getModule()
                .stravaPreferences
                .accessToken
        )
    }
    chain.proceed(newBuilder.build()) ^addInterceptor
}
```

FIGURE 9. HTTP-request interception implementation

3.3.3 Automatic token renewal

OkHttpClient can be configured to use an authenticator. An authenticator catches HTTP responses in the client side and provides a handle for getting data from the response. In case the HTTP client receives an HTTP response containing the HTTP code 401, StravaService renews the access token. After that, the same request, which was previously denied due to the HTTP 401 code, is resent with the renewed access token. HTTP codes are status codes for HTTP responses. The code 401 stands for “unauthorized”, meaning that the request lacks valid authorization credentials. (26.) Figure 11 contains the business logic for executing a re-authentication in case a client error code (HTTP 401) is received.



```
builder.authenticator { _ : Route? , response : Response  ->
    BeatLog.d(TAG,  string: "Authenticator received: " + response.code())
    if (response.code() == 401) {
        BeatApp
            .getModule()
            .stravaService
            .refreshAccessToken()
            .blockingAwait()
        return@authenticator response
            .request()
            .newBuilder()
            .header(
                name: "Authorization",
                value: "Bearer " + BeatApp.getModule().stravaPreferences.accessToken
            )
            .build()
    }
    null ^authenticator
}
```

FIGURE 10. Automatic re-authentication configuration implementation with OkHttp

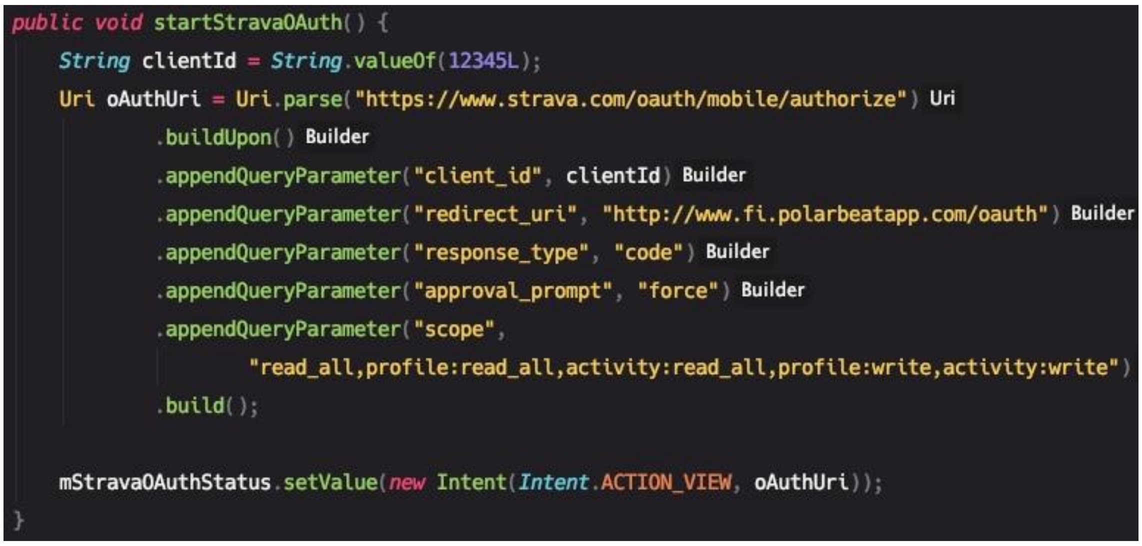
3.3.4 Initiating the OAuth2-authentication process in Beat

The authentication process starts by preparing an implicit Intent. An Intent describes an object that can be used to request an action from another component. (15.) In Figure 12 a URI is initiated with an address to Strava's OAuth authorization endpoint using the parse method.

The buildUpon method returns an object utilizing the Builder-design pattern. The Builder-design pattern essentially provides a way to construct complex objects piece by piece by providing methods for mutating an object. (36.)

The appendQueryParameter method as the name suggests, appends query parameters to the oAuthUri-object being built. In this example the URI is provided with various parameters, the most important ones being the client ID, redirect URI, response type and scope. The client ID as a unique ID in Strava's ecosystem, this acts as an identity for a user. The redirect URI is an URI that after the user has successfully granted permissions, the OAuth-process will be redirected to. This is very important since the redirected URI contains the authorization token needed for further authentication.

Finally, the scope parameter determines the access level to the API. In this example the user is requested to give consent to read or write to profile and activity scopes. mStravaOAuthStatus is an observable member in a ViewModel class. By setting this variable, observers will receive the Intent object built in this method. In this scenario the observer is an Activity (View) class which ultimately launches the request.



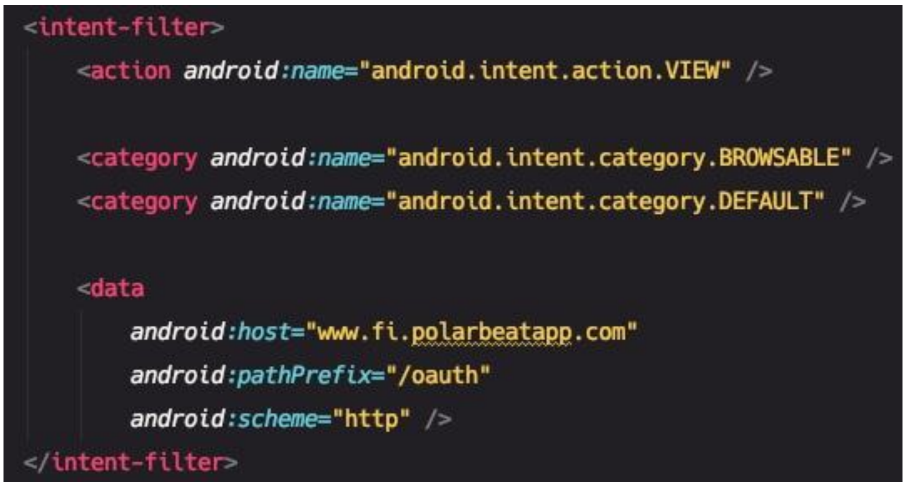
```
public void startStravaOAuth() {
    String clientId = String.valueOf(12345L);
    Uri oAuthUri = Uri.parse("https://www.strava.com/oauth/mobile/authorize") Uri
            .buildUpon() Builder
            .appendQueryParameter("client_id", clientId) Builder
            .appendQueryParameter("redirect_uri", "http://www.fi.polarbeatapp.com/oauth") Builder
            .appendQueryParameter("response_type", "code") Builder
            .appendQueryParameter("approval_prompt", "force") Builder
            .appendQueryParameter("scope",
                    "read_all,profile:read_all,activity:read_all,profile:write,activity:write")
            .build();

    mStravaOAuthStatus.setValue(new Intent(Intent.ACTION_VIEW, oAuthUri));
}
```

FIGURE 11. Constructing Intent to initiate the OAuth-process

3.3.5 Capturing primitive data by deep linking

When a web URI Intent is invoked, the Android system tries to find an app that can handle the Intent. In the authentication redirecting scenario the Polar Beat app is configured to listen to a specific type of Intent. This is called a deep link. (16.) This is achieved by setting up an Intent filter in the AndroidManifest.xml file in the Polar Beat app project (figure 13). The manifest file, amongst other things, declares any components of the app such as Activities and Services, and permissions.

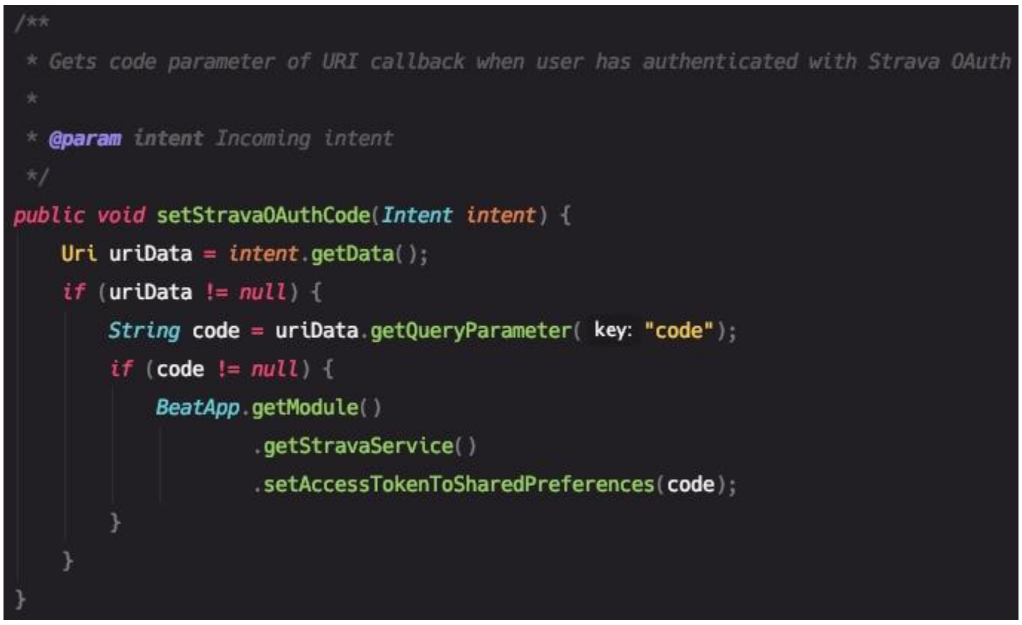
Whenever the an action type of “VIEW” is invoked with the data containing the URI “http://www.fi.polarbeatapp.com/oauth”, the Beat app will be launched with a specific Activity-class and a lifecycle method onNewIntent() in an Activity will be called. Lifecycle methods are essentially callbacks that happen on specific circumstances, in this case when a new Intent arrives. (19.)



```
<intent-filter>
    <action android:name="android.intent.action.VIEW" />

    <category android:name="android.intent.category.BROWSABLE" />
    <category android:name="android.intent.category.DEFAULT" />

    <data
        android:host="www.fi.polarbeatapp.com"
        android:pathPrefix="/oauth"
        android:scheme="http" />
</intent-filter>
```

FIGURE 12. Setting an Intent filter for Polar Beat to listen for the result of the OAuth-process

After the app has received the data in the Intent, the data will be parsed and saved into the local persistent storage by a Model class namely StravaService, as demonstrated on Figure 14.



```
/**
 * Gets code parameter of URI callback when user has authenticated with Strava OAuth
 *
 * @param intent Incoming intent
 */
public void setStravaOAuthCode(Intent intent) {
    Uri uriData = intent.getData();
    if (uriData != null) {
        String code = uriData.getQueryParameter( key: "code");
        if (code != null) {
            BeatApp.getModule()
                    .getStravaService()
                    .setAccessTokenToSharedPreferences(code);
        }
    }
}
```

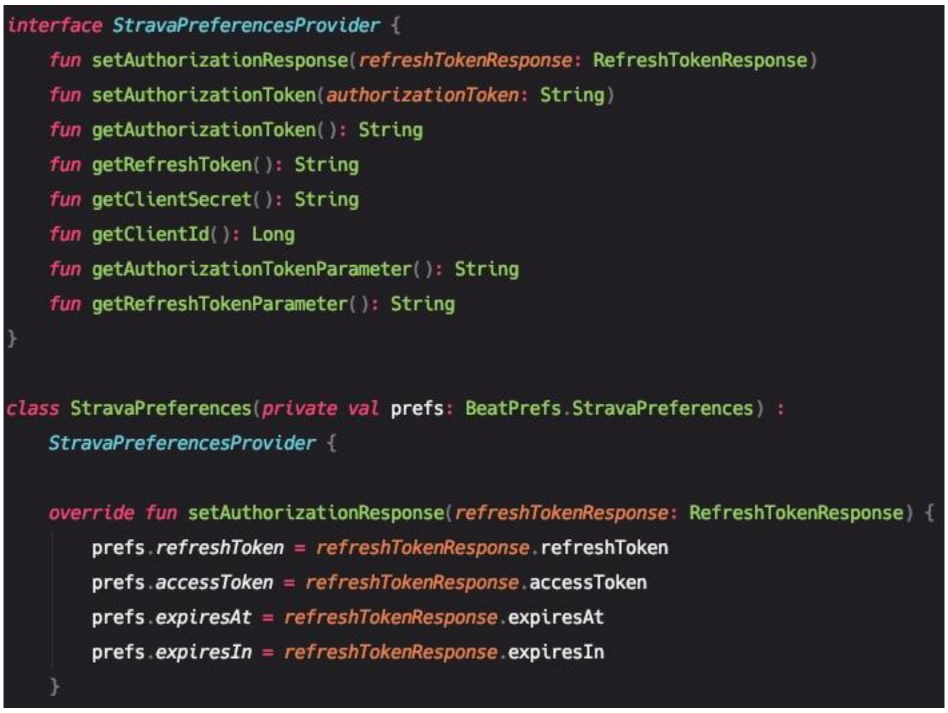
FIGURE 13. Saving the authorization token in local persistent storage

3.3.6 Utilizing local persistent storage

The authorization token is saved to SharedPreferences, which is a persistent local storage in an Android device. SharedPreferences provides a way to store primitive values in key-value pairs and is suitable for this case. For production use, a more secure way should be implemented since

SharedPreferences can be breached relatively easily if a malicious user has access to a device's local storage.

Figure 15 shows an implementation for saving various tokens and details into SharedPreferences. StravaPreferences utilizes the Wrapper-design pattern to encapsulate the functionality of accessing SharedPreferences. This is useful since StravaPreferences implements the interface StravaPreferencesProvider, StravaPreferences can be mocked in unit tests.



```kotlin
interface StravaPreferencesProvider {
    fun setAuthorizationResponse(refreshTokenResponse: RefreshTokenResponse)
    fun setAuthorizationToken(authorizationToken: String)
    fun getAuthorizationToken(): String
    fun getRefreshToken(): String
    fun getClientSecret(): String
    fun getClientId(): Long
    fun getAuthorizationTokenParameter(): String
    fun getRefreshTokenParameter(): String
}

class StravaPreferences(private val prefs: BeatPrefs.StravaPreferences) :
    StravaPreferencesProvider {

    override fun setAuthorizationResponse(refreshTokenResponse: RefreshTokenResponse) {
        prefs.refreshToken = refreshTokenResponse.refreshToken
        prefs.accessToken = refreshTokenResponse.accessToken
        prefs.expiresAt = refreshTokenResponse.expiresAt
        prefs.expiresIn = refreshTokenResponse.expiresIn
    }
```

FIGURE 14. SharedPreferences-wrapper class implementation

3.3.7 Acquiring access credentials

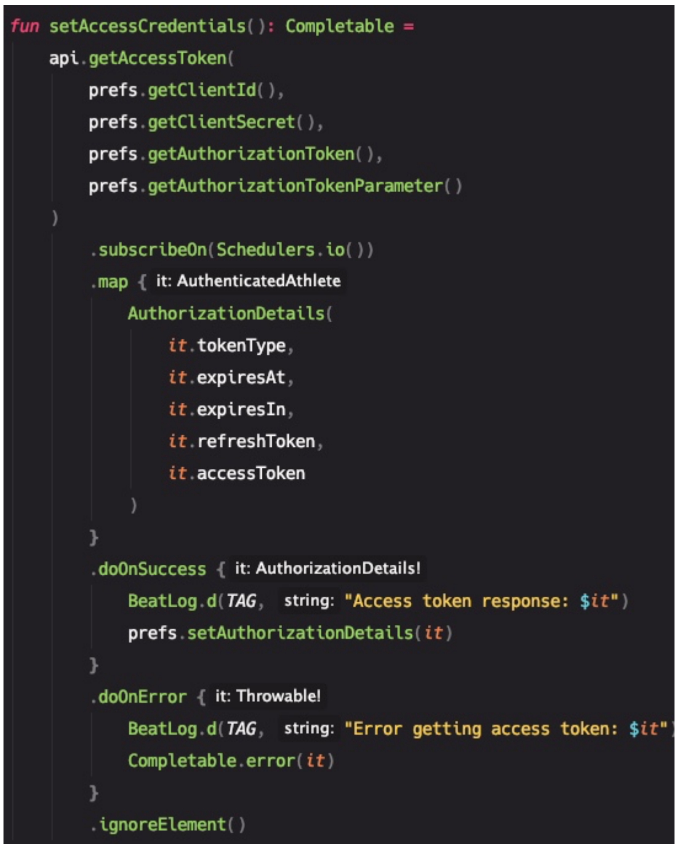
The StravaService-class contains the setAccessCredentials() function in Figure 16. setAccessCredentials() contains chained methods utilizing RxJava observable classes, the Single or Completable, and initiates the process of saving access credentials to a local persistent storage. Single implements the Reactive Pattern for a single value response and only emits a non-null value or an error. (22.) Completable is similar, except for only emitting a success or error state with no return value.

RxJava contains operators for transforming observable streams, in this case the Map and IgnoreElement operators are used. The Map operator transforms an item emitted by an observable stream by applying a function to it and emitting the result downstream. (23.) The IgnoreElement operator ignores a value emitted by a stream terminating the stream by emitting either a success value or an error. (24.)

In Figure 16 the object “api” represents the StravaAPI class which executes the request to the Strava REST API. “prefs” represents the StravaPreferences class which fetches the parameters from local persistent storage that are needed for executing the request for receiving the access token.

The execution in setAccessCredentials() goes as follows:

1) The api.getAccessToken() function is called with required parameters, such as the authorization token and user ID.
2) RxJava-operator subscribeOn() changes the thread where the function is executed from main thread to a background thread from the I/O thread pool.
3) The api.getAccessToken() returns a response containing an object AuthenticatedAthlete which contains the access / refresh token.
4) The response is deconstructed and transformed into another type of object, the AuthorizationDetails-object by the map()-operator.
5) Stream is transformed into a Completable
    a) Success value is emitted to observers if no errors occur in the stream and prefs.setAuthorizationDetails() is called, authorization details are saved into SharedPreferences.
    b) Error value is emitted to observers if an error happens in the stream.



```
fun setAccessCredentials(): Completable =
    api.getAccessToken(
        prefs.getClientId(),
        prefs.getClientSecret(),
        prefs.getAuthorizationToken(),
        prefs.getAuthorizationTokenParameter()
    )
        .subscribeOn(Schedulers.io())
        .map { it: AuthenticatedAthlete
            AuthorizationDetails(
                it.tokenType,
                it.expiresAt,
                it.expiresIn,
                it.refreshToken,
                it.accessToken
            )
        }
        .doOnSuccess { it: AuthorizationDetails!
            BeatLog.d(TAG,  string: "Access token response: $it")
            prefs.setAuthorizationDetails(it)
        }
        .doOnError { it: Throwable!
            BeatLog.d(TAG,  string: "Error getting access token: $it"
            Completable.error(it)
        }
        .ignoreElement()
```

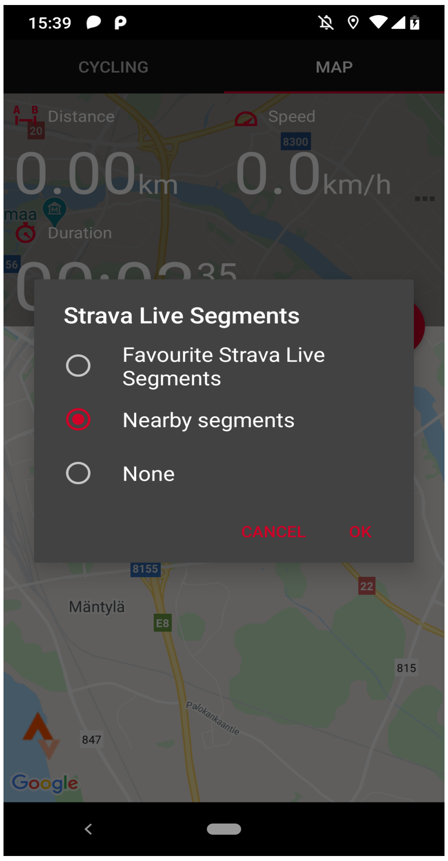
FIGURE 15. Saving access credentials to SharedPreferences utilizing functional reactive programming

3.4 Fetching segments from the Strava API

This chapter describes the steps to achieve the desired result in detail from drawing the Strava Live Segments to a Google Map in Beat.

3.4.1 View-implementation

The user initiates the process by selecting the type of segments to be shown on the map. By clicking the Strava icon on the lower left corner on the map, a Dialog window is opened with the view in Figure 17. The user can here proceed by selecting one of three choices, favourite Strava Live Segments, nearby segments or none.


15:39

CYCLING | MAP

Distance 0.00km

Speed 0.0km/h

Duration

Strava Live Segments

○ Favourite Strava Live Segments

◉ Nearby segments

○ None

CANCEL OK

FIGURE 16. Strava Live Segments-type selector UI

By selecting favourite Strava Live Segments the app will fetch and draw the segments that the user has favourited on the Strava ecosystem. By selecting nearby segments, the app will fetch ten nearby most popular segments from the Strava ecosystem. Nearby in this case signifies a square shaped area where the user stands in the middle. For testing purposes, the area has been configured to have a radius of approximately two kilometres, ideally the radius would scale in proportion with the zoom level set in the Google Map. After fetching the segments, the result is a map with segment routes and starting point markers drawn in orange. Figure 18 contains end result of the View-implementation, containing nearby Strava Live Segments in a Google Map.

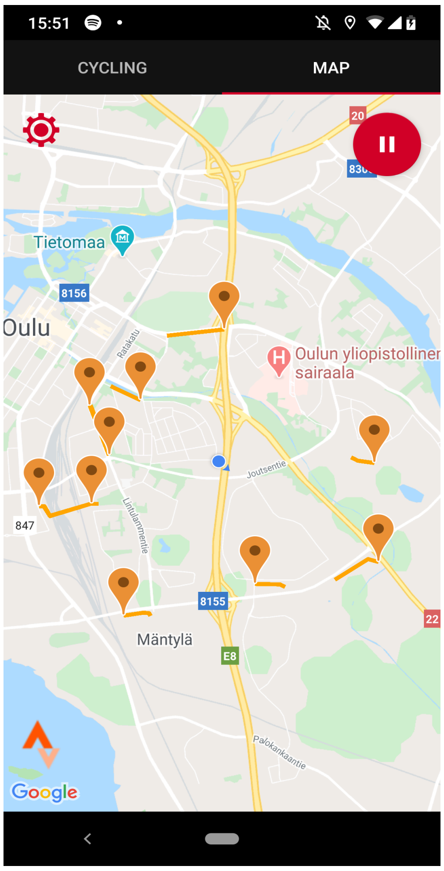
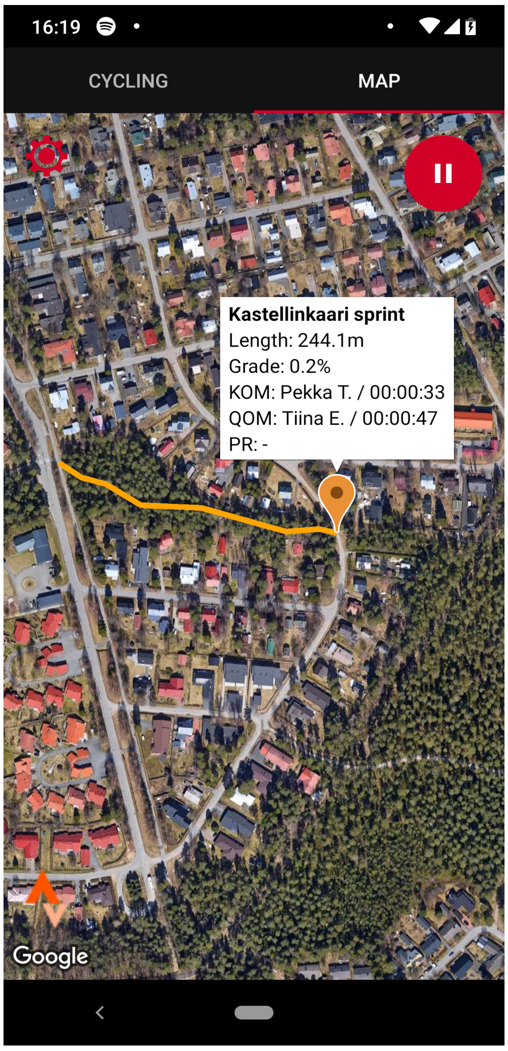


FIGURE 17. Exercise-view with nearby Strava Live Segments drawn in the Google Map

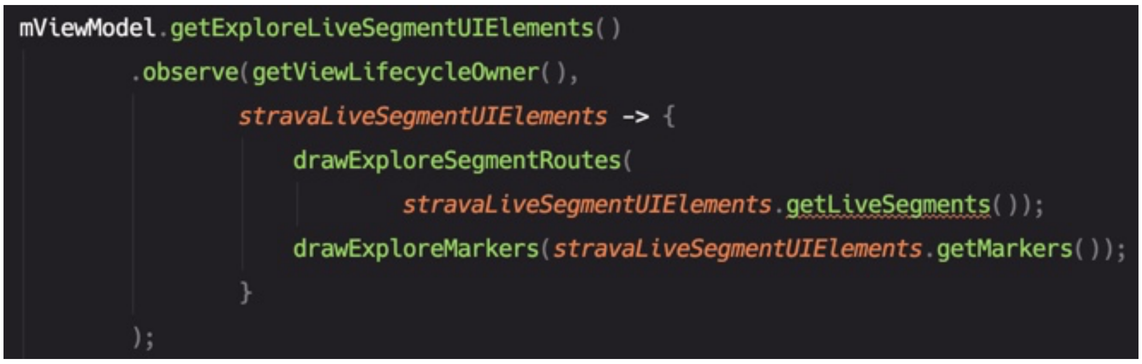
By clicking, an info window pops up and shows details about a segment. Figure 19 shows a screenshot of the app showinga a segment with an opened info window. The name of the segment is written with bold characters, details are below. A grade stands for the change in altitude, positive values mean a gain in average altitude during the segment. KOM/QOM are segment record times. KOM for men and QOM for women. In this scenario the user has not completed the segment in Figure 19 and, therefore, a personal record (PR) time does not exist and is marked with a line.



Kastellinkaari sprint
Length: 244.1m
Grade: 0.2%
KOM: Pekka T. / 00:00:33
QOM: Tiina E. / 00:00:47
PR: -

FIGURE 18. Strava Live Segment-info window containing details about a segment

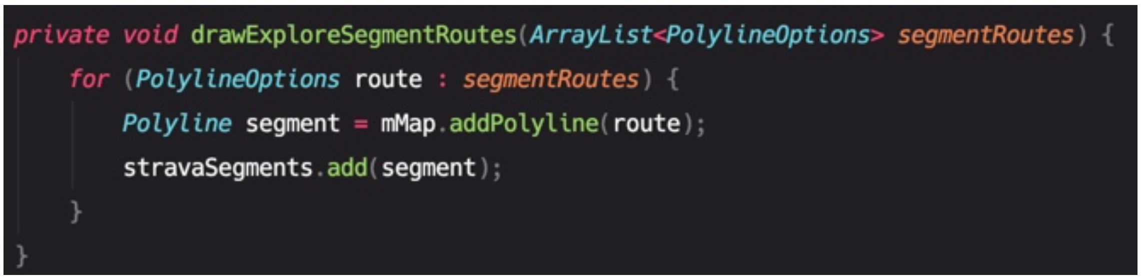
ExerciseMapFragment observes the StravaViewModel for the data regarding the markers. When the StravaViewModel emits data, ExerciseMapFragment will receive that data and perform operations to draw the routes and markers to the map. Figure 20 contains a code snippet where the ExerciseMapFragment subscribes to the segment data and calls functions that draw the routes and markers to the Google Map.



```
mViewModel.getExploreLiveSegmentUIElements()
        .observe(getViewLifecycleOwner(),
                stravaLiveSegmentUIElements -> {
                    drawExploreSegmentRoutes(
                            stravaLiveSegmentUIElements.getLiveSegments());
                    drawExploreMarkers(stravaLiveSegmentUIElements.getMarkers());
                }
        );
```

FIGURE 19. Event driven solution utilizing the Observer-pattern for drawing Strava Live Segments to a Google Map

Routes and markers are drawn to the map by calling methods using a reference to the Google Map. In Figure 21 drawExploreSegmentRoutes() loops through a collection (ArrayList) and iteratively calls mMap.addPolyline(), which adds polylines that represent the routes to the map. The reference to the route is stored to another collection, stravaSegments, which is required for keeping a list of references that contain all the Strava segments. This list is used in case the segments need to be redrawn or deleted from the map. Markers are drawn to the map in a similar manner.

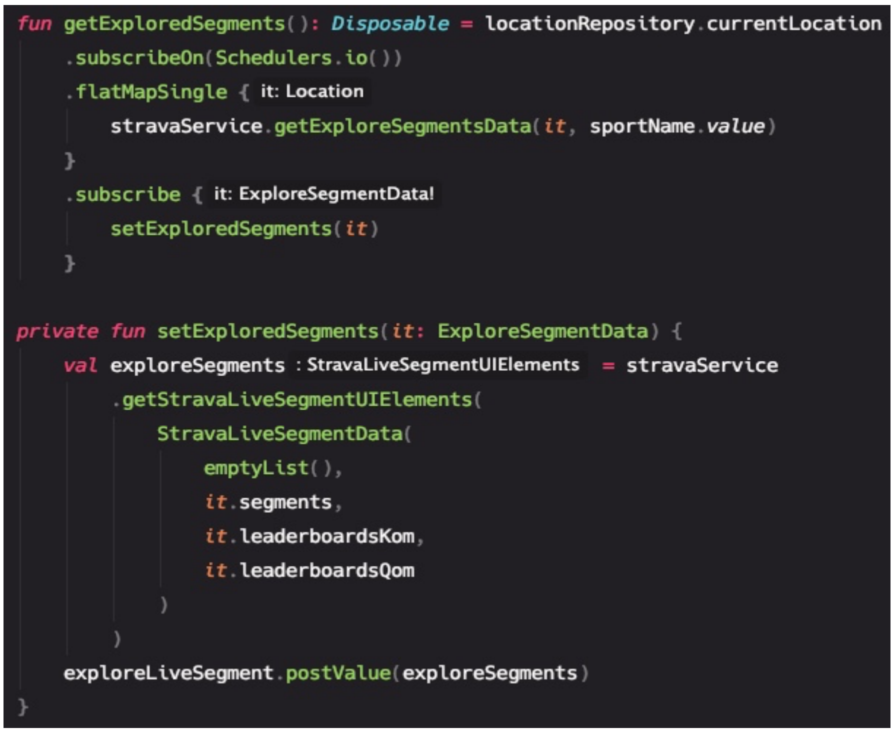


```
private void drawExploreSegmentRoutes(ArrayList<PolylineOptions> segmentRoutes) {
    for (PolylineOptions route : segmentRoutes) {
        Polyline segment = mMap.addPolyline(route);
        stravaSegments.add(segment);
    }
}
```

FIGURE 20. Adding routes (Polylines) to a Google Map

3.4.2 ViewModel-implementation

Figure 22 contains a code snippet from StravaViewModel class. getExploredSegments() initiates an execution flow which ends in emitting segment route / marker data to an observable LiveData field. “exploreLiveSegment”. First LocationRepository is observed for the user’s current location. After receiving the location, StravaService is queried with the location data in addition to the current sport chosen on the Exercise feature in Beat. Finally using the segment data, StravaService is queried to return the UI elements needed for the Google Map using the segment data. The result is emitted to the “exploreLiveSegment” LiveData-observable.

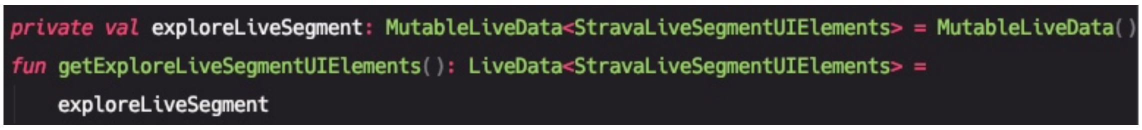


```kotlin
fun getExploredSegments(): Disposable = locationRepository.currentLocation
    .subscribeOn(Schedulers.io())
    .flatMapSingle { it: Location
        stravaService.getExploreSegmentsData(it, sportName.value)
    }
    .subscribe { it: ExploreSegmentData!
        setExploredSegments(it)
    }

private fun setExploredSegments(it: ExploreSegmentData) {
    val exploreSegments : StravaLiveSegmentUIElements  = stravaService
        .getStravaLiveSegmentUIElements(
            StravaLiveSegmentData(
                emptyList(),
                it.segments,
                it.leaderboardsKom,
                it.leaderboardsQom
            )
        )
    exploreLiveSegment.postValue(exploreSegments)
}
```

FIGURE 21. Transforming and emitting data to observers in ViewModel

Figure 23 contains the accessor the exploreLiveSegment() and the exploreLiveSegment-LiveData-observable. exploreLiveSegments is a private scoped member variable of StravaViewModel. In Kotlin private members are visible / accessible only inside the class it is declared. A getter-function is provided for subscribers to observe the exploreLiveSegments-variable.

LiveData-class is immutable meaning that it cannot be mutated, or in other words, the values the object contains cannot be changed after construction. Immutability is encouraged since it provides amongst other things, thread-safety. (18.) Objects can be accessed by multiple threads and their value is guaranteed to be the same for all parties since the value cannot be changed.



```kotlin
private val exploreLiveSegment: MutableLiveData<StravaLiveSegmentUIElements> = MutableLiveData()
fun getExploreLiveSegmentUIElements(): LiveData<StravaLiveSegmentUIElements> =
    exploreLiveSegment
```

FIGURE 22. Observable accessor for observers to access Strava Live Segments data

3.4.3 Model-implementation

Last in the chain of execution is the Model-layer, which is occupied by a class StravaService. The main responsibilities of StravaService are executing HTTP calls to Strava API and parsing the responses to create the data classes that hold data for the Strava Live Segments that ultimately are drawn into the UI. (25.)

Strava API contains an endpoint to return a fixed number of most popular segments in an area. The area is determined by two latitude and longitude points (southwest latitude/longitude, northeast latitude/longitude) describing a rectangular point boundary for the search (figure 24).

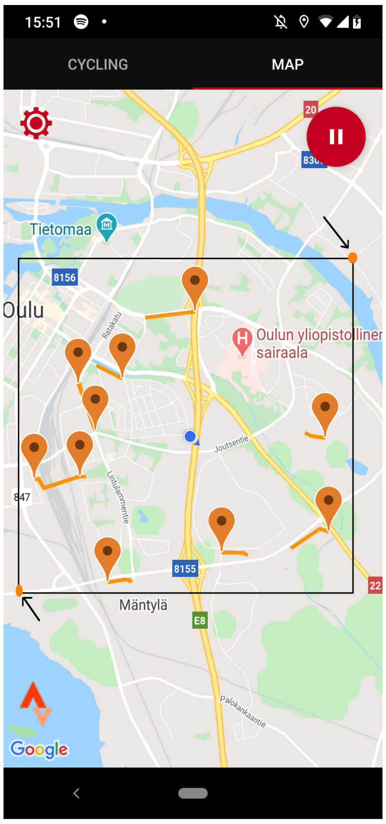
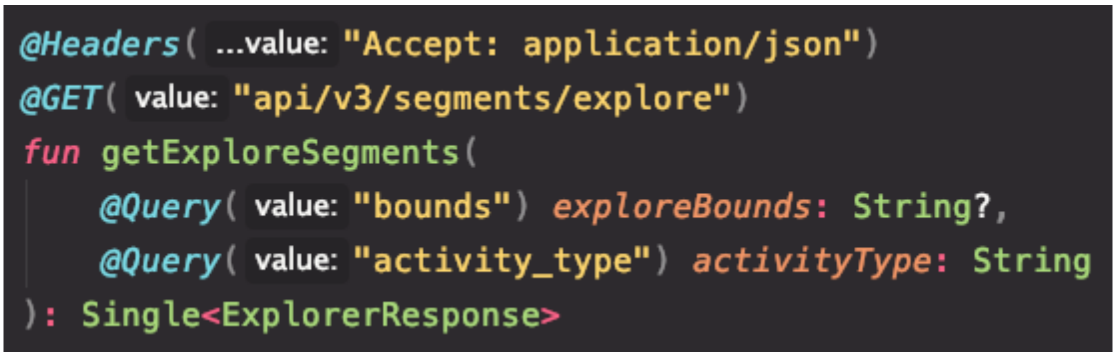


FIGURE 23. Area where Strava Live Segments are fetched

The segments are fetched from Strava API using an implementation of a Retrofit interface. This interface is a Java/Kotlin interface that contain various methods. The function getExploreSegments() in Figure 25 shows a function with Android Annotations. Android Annotations is a library that autogenerates code on compile-time. (27.) In getExploreSegments() various Retrofit annotations are used.

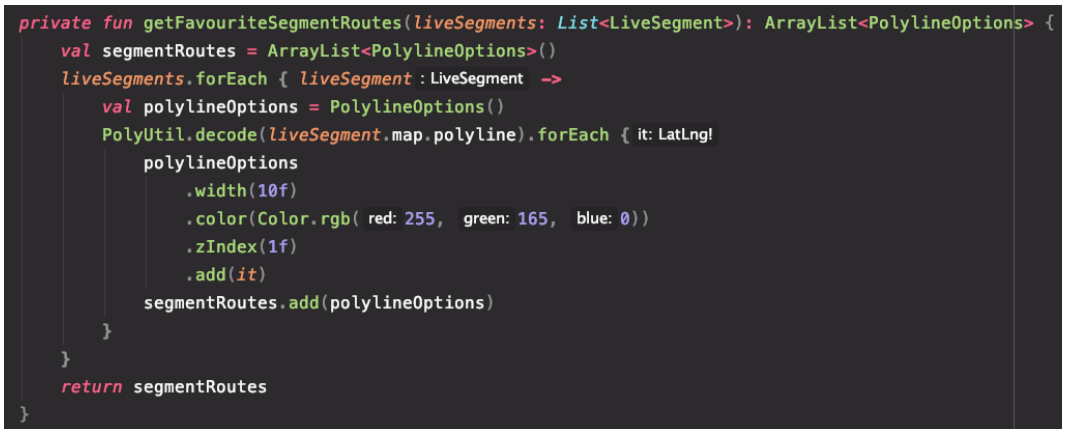
@Headers-annotation marks the methods return type, in this case the return type is JSON. @GET-annotation marks the endpoint where the HTTP request is sent in addition to the request type. Here the request type is an HTTP GET-request. An HTTP GET request represents a representation of a specified network resource. (28.) @Query-header marks that the variable annotated should be added as a query parameter in the request. The Retrofit-library generates an implementation in compile-time of the interface where the method in Figure 25 is situated.



```
@Headers( ...value: "Accept: application/json")
@GET( value: "api/v3/segments/explore")
fun getExploreSegments(
    @Query( value: "bounds") exploreBounds: String?,
    @Query( value: "activity_type") activityType: String
): Single<ExplorerResponse>
```

FIGURE 24. Retrofit-interface function implementation for fetching a list of Strava Live Segments

After the segments are fetched using getExploreSegments(), data for routes and markers is fetched. PolylineOptions-object represents a sequence of location points. In Figure 26 the getFavouriteSegmentRoutes() function loops through iteratively every Strava Live Segment and decodes a PolyLine from the encoded PolyLine data representing a route from Strava API. All Polylines are added to an ArrayList which is returned in the end. This ArrayList contains all the routes for the segments. Constructing the markers are done in a similar fashion by iterating through a collection of items. In the end StravaService returns the segments and needed data to StravaViewModel for further use.



```
private fun getFavouriteSegmentRoutes(liveSegments: List<LiveSegment>): ArrayList<PolylineOptions> {
    val segmentRoutes = ArrayList<PolylineOptions>()
    liveSegments.forEach { liveSegment : LiveSegment ->
        val polylineOptions = PolylineOptions()
        PolyUtil.decode(liveSegment.map.polyline).forEach { it: LatLng!
            polylineOptions
                .width(10f)
                .color(Color.rgb( red: 255,  green: 165,  blue: 0))
                .zIndex(1f)
                .add(it)
            segmentRoutes.add(polylineOptions)
        }
    }
    return segmentRoutes
}
```

FIGURE 25. Example of constructing a route (Polyline) using the location data fetched from Strava API

4 SOFTWARE TESTING

4.1 Software testing techniques

Testing any software is a very important step of the software development lifecycle. By testing the code produced, bugs and undesired functionality or software regression can be caught in the early stages of development. (30.) There are different types of testing in software development which can be divided into two main categories, manual testing and automation testing. Manual testing is typically conducted by a human who tests software by interacting with it manually “by hand”. Automating testing is typically carried out by writing test scripts or using a testing framework to automatically test software without human intervention.

4.2 Automation testing approach

Automation testing in this project was carried out by a Test Last Development approach (TLD) for the new components by building characterization tests on the ViewModel / Model-layers. (29.) Characterization tests are tests that test the correctness of existing behaviour. (31.) In other words, characterization tests can be written on code that was not previously covered by automated tests, such as unit tests. (33.)

A popular choice for automated testing is the Test-Driven Development (TDD) approach. Shortly, in TDD automated tests are written before the executed code using a cyclical approach. In TDD development begins by writing a test case, which fails since the production code does not exist. After this, production code is written and a test case should pass. When a test case passes, the code should be refactored or cleaned up. After refactoring, the cycle begins again until a feature is complete. (37.)

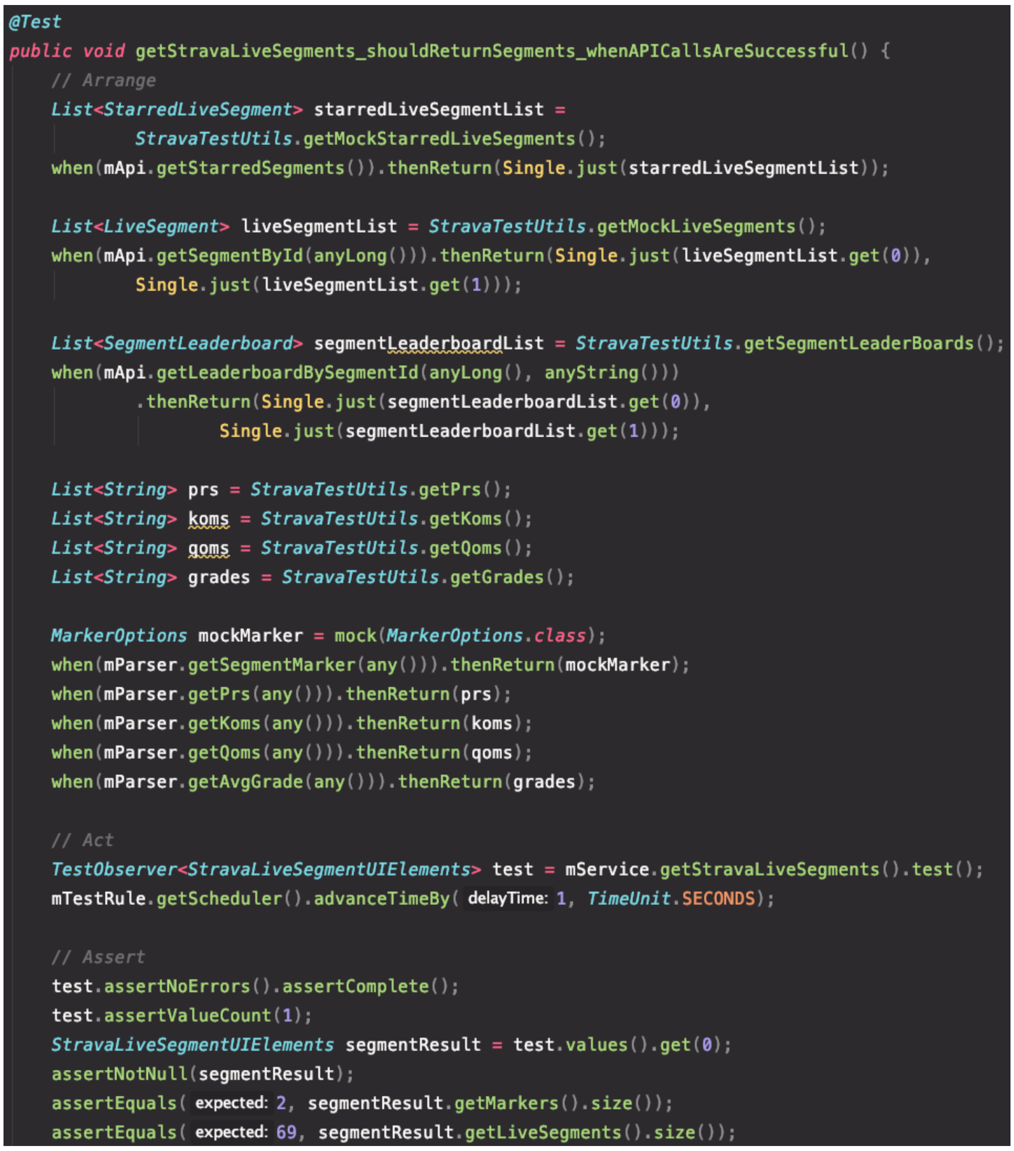
Another choice for automated testing is the more traditional Test Last Development (TLD) approach. In TLD production code is written before any automated test, this leads to fast development speed, since test code is written only after the production code is finished. In many cases TDD produces more test cases, which helps in bug fixing, further development and maintenance since the test coverage has more granularity by testing more scenarios in greater detail. In larger projects this is more beneficial, but for this project TLD was chosen as the followed approach due to the easier and faster development speed. (32.)

4.3 Automation testing the new features

Automated tests in this project were developed using the JUnit test framework. All the tests developed were unit tests. Unit tests are automated tests where individual units of code are tested to validate the functionality. Unit tests can validate individual functions or a single module of code. In this project the ViewModel and Model modules were unit tested.

Figure 27 describes a unit test of a “happy-case” in a test class testing StravaService, where all actions produce the desired behaviour. The unit test is divided into three sections, the “Arrange” section is for setting up or arranging the test scenario, the “Act” section is for executing the action to initiate the execution flow that is tested. “Assert” section is for validating the output of the test. If any assertion fails, the test scenario fails.

External dependencies are replaced by mocked objects or test doubles. Mock objects are simulated objects that mimic the behaviour of real objects. (34.) By using mock objects, a controlled test scenario can be guaranteed since the preconditions for a test scenario are not changing. Test doubles are real objects but setup with a specific state to create a desired precondition.



```java
@Test
public void getStravaLiveSegments_shouldReturnSegments_whenAPICallsAreSuccessful() {
    // Arrange
    List<StarredLiveSegment> starredLiveSegmentList =
            StravaTestUtils.getMockStarredLiveSegments();
    when(mApi.getStarredSegments()).thenReturn(Single.just(starredLiveSegmentList));

    List<LiveSegment> liveSegmentList = StravaTestUtils.getMockLiveSegments();
    when(mApi.getSegmentById(anyLong())).thenReturn(Single.just(liveSegmentList.get(0)),
            Single.just(liveSegmentList.get(1)));

    List<SegmentLeaderboard> segmentLeaderboardList = StravaTestUtils.getSegmentLeaderBoards();
    when(mApi.getLeaderboardBySegmentId(anyLong(), anyString()))
            .thenReturn(Single.just(segmentLeaderboardList.get(0)),
                    Single.just(segmentLeaderboardList.get(1)));

    List<String> prs = StravaTestUtils.getPrs();
    List<String> koms = StravaTestUtils.getKoms();
    List<String> qoms = StravaTestUtils.getQoms();
    List<String> grades = StravaTestUtils.getGrades();

    MarkerOptions mockMarker = mock(MarkerOptions.class);
    when(mParser.getSegmentMarker(any())).thenReturn(mockMarker);
    when(mParser.getPrs(any())).thenReturn(prs);
    when(mParser.getKoms(any())).thenReturn(koms);
    when(mParser.getQoms(any())).thenReturn(qoms);
    when(mParser.getAvgGrade(any())).thenReturn(grades);

    // Act
    TestObserver<StravaLiveSegmentUIElements> test = mService.getStravaLiveSegments().test();
    mTestRule.getScheduler().advanceTimeBy( delayTime: 1, TimeUnit.SECONDS);

    // Assert
    test.assertNoErrors().assertComplete();
    test.assertValueCount(1);
    StravaLiveSegmentUIElements segmentResult = test.values().get(0);
    assertNotNull(segmentResult);
    assertEquals( expected: 2, segmentResult.getMarkers().size());
    assertEquals( expected: 69, segmentResult.getLiveSegments().size());
```

FIGURE 26. A test case validating a correct response when all external dependencies are provided successfully

5 RESULTS AND CONCLUSION

The aim of this thesis was to create a proof of concept demo by bringing the Strava Explore feature to Polar Beat. This was achieved by adding the Strava Explore component to be part of the Exercise-feature in Beat. The new feature brought the ability for the user to search nearby Strava Live Segments and visualize the segment routes in the Exercise-feature on a Google Map.

The project served as a great exercise for building a new feature in a commercial application using clean software architecture. The MVVM-software architectural pattern was utilized in addition to functional reactive programming concepts using RxJava which proved to be beneficial in preparing the new components to be reusable and highly testable. Unit tests were built to cover the new components (ViewModels and Models) which strengthened my skills with software testing. Ideally also the View-layer in regards of the new components should be covered by tests, but this was not implemented due to the need of larger refactoring in order to enable a proper test coverage.

The most difficult part in the project was setting up the Retrofit HTTP client and authentication using Retrofit. This required some studying about HTTP communications in general and authentication, particularly OAuth2, which is required to be setup for communicating with Strava. After the initial slowdowns, the rest of the work went smoothly.

The results of the project were demonstrated and discussed in Polar Electro Oy. The project met a warm reception and could be implemented in future releases after fixing some hard-coded elements and UX improvements in the app such as a hard-coded Strava ID. For the production version the Strava ID should be individual for the Polar user resulting in the correct segments fetched for the user. The app endured some performance issues when rendering large numbers of segments in the map simultaneously, which could be a topic of improvement.

REFERENCES

1. Kotlin as the preferred programming language for Android . TechCrunch. Cited 20.10.2019 https://techcrunch.com/2019/05/07/kotlin-is-now-googles-preferred-language-for-android-app-development/
2. The Java-programming language. Wikipedia. Cited 20.10.2019, https://en.wikipedia.org/wiki/Java_(programming_language)
3. Kotlin compared to Java. JetBrains. 2019. Cited 20.10.2019 https://kotlinlang.org/docs/reference/comparison-to-java.html
4. Polar H10. Polar Electro Oy. 2019. Cited 15.12.2019. https://www.polar.com/sites/default/files/product3/1500x1500/h10_heart_rate_sensor_1500x1500.jpg
5. Meet Android Studio. Google. 2020. Cited 09.02.2020. https://developer.android.com/studio/intro
6. Postman API Client. Postman. 2020. Cited 09.02.2020. https://www.postman.com/product/api-client
7. MVVM-software architectural design pattern. Microsoft. 2020. Cited 09.02.2020. https://docs.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/mvvm
8. Observer pattern. Wikipedia. 2020. Cited 09.02.2019. https://en.wikipedia.org/wiki/Observer_pattern
9. RxJava. Github. 2020. Cited 09.02.2020. https://github.com/ReactiveX/RxJava
10. Processes and threads overview. Google. 2020. Cited 09.02.2020. https://developer.android.com/guide/components/processes-and-threads
11. Fragments. Google. 2020. Cited 09.02.2020 https://developer.android.com/guide/components/fragments
12. Retrofit. Square. 2019. Cited 15.12.2019. https://square.github.io/retrofit/
13. OAuth2. Strava. 2020. Cited 09.02.2020 https://developers.strava.com/docs/authentication/
14. Polar Beat. Polar. 2020. Cited 05.02.2020. https://play.google.com/store/apps/details?id=fi.polar.beat&hl=en
15. Intent and Intent filters. Google. 2020. Cited 09.02.2020. https://developer.android.com/guide/components/intents-filters#ExampleExplicit

16. Create Deep Links to App Content. Google. 2020. Cited 09.02.2020. https://developer.android.com/training/app-links/deep-linking
17. Dependency injection. Google. 2020. Cited 10.02.2020. https://developer.android.com/training/dependency-injection
18. LiveData Overview. Google. 2020. Cited 10.02.2020. https://developer.android.com/topic/libraries/architecture/livedata
19. Handling Lifecycles with Lifecycle-Aware Components. Google. 2020. Cited 11.02.2020. https://developer.android.com/topic/libraries/architecture/lifecycle
20. Encapsulation. Wikipedia. 2020. Cited 12.02.2020. https://en.wikipedia.org/wiki/Encapsulation_(computer_programming)
21. Single. ReactiveX. 2020. Cited 12.02.2020. http://reactivex.io/documentation/single.html
22. Map. ReactiveX. 2020. Cited 12.02.2020. http://reactivex.io/documentation/operators/map.html
23. IgnoreElements. ReactiveX. 2020. Cited 12.02.2020. http://reactivex.io/documentation/operators/ignoreelements.html
24. Data classes. Kotlin. 2020. Cited 13.02.2020. https://kotlinlang.org/docs/reference/data-classes.html
25. 401 Unauthorized. MDN web docs. 2020. Cited 14.02.2020. https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/401
26. Android Annotations. GitHub. 2020. Cited 15.02.2020. https://github.com/androidannotations/androidannotations/wiki
27. HTTP GET. MDN web docs. 2020. Cited 15.02.2020. https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/GET
28. What is Automation Testing. Software Testing Help. 2020. Cited 16.02.2020. https://www.softwaretestinghelp.com/automation-testing-tutorial-1/
29. Software regression. Wikipedia. 2020. Cited. 16.02.2020. https://en.wikipedia.org/wiki/Software_regression
30. Characterization tests. Michael Feathers. 2020. Cited. 16.02.2020. https://michaelfeathers.silvrback.com/characterization-testing
31. Test Last Approach. Compilehorrors. 2020. Cited. 16.02.2020. http://compilehorrors.com/test-driven-development-tdd-vs-test-last-development-tld-a-comparative-study/
32. Unit testing. Software Testing Fundamentals. 2020. Cited. 16.02.2020. http://softwaretestingfundamentals.com/unit-testing/

33. Mock object. Wikipedia. 2020. Cited 16.02.2020. https://en.wikipedia.org/wiki/Mock_object
34. Singleton pattern. Wikipedia 2020. Cited. 01.03.2020. https://en.wikipedia.org/wiki/Singleton_pattern
35. Builder pattern. Refactoring Guru 2020. Cited. 01.03.2020. https://refactoring.guru/design-patterns/builder
36. Test Driven Development. Agiledata. 2020. Cited 01.03.2020. http://agiledata.org/essays/tdd.html