University for the Common Good

Kreasport: Learning modern programming practices for web and mobile development

Project work report

by

Christopher CARONI & Adel TIHIANINE

UNIVERSITÉ LILLE 1
IUT A
DUT INFORMATIQUE 2016/2017

GCU TUTOR: EDDIE GRAY
IUT A TUTOR: PATRICK LEBÈGUE

Acknowledgments

First, I’d like to thank Professor Eddie Gray for welcoming us to Glasgow Caledonian University and his supervision. To GCU for accepting us and giving us the opportunity to carry out this project.

Thank you to all our professors at the computer science department of IUT A, Université Lille 1 who helped us to reach this point.

Thank you to all the staff at the Relations Internationales department at the IUT for organizing the exchange.

Abstract

The last decade has seen the emergence and rise of mobile computing devices. Smartphones, as we now call them, are an ever-present part of our lives. We do everything and anything with them: we use them for business, education, entertainment and even while playing sports.

This project's main goal was to gain a maximum amount of knowledge about modern programming principles for Android by designing and creating a useful and functional application. The application presented here is designed for orienteering races. The users download races from a server and then can choose to run any one of them with the GPS in their smartphone.

This report aims to document the work done and the process behind it. It details the development cycle, our design choices, the various problems we've faced and how we responded to them.

Résumé

Les dix dernières années ont vu l'émergence et le succès de l'informatique mobile. Les « smartphones », tels que nous les appelons, occupent aujourd'hui une place important dans nos vies. Nous les utilisons pour faire tout et n'importe quoi : nous les utilisons dans notre travail, pour apprendre, pour nous distraire et même pour enregistrer nos activités sportives.

L'objectif principal de ce projet fut d'apprendre un maximum sur les principes modernes de programmation pour Android en concevant et créant une application utile et en mesure de fonctionner correctement. L'application présentée ici est conçu pour réaliser des courses d'orientation. Les utilisateurs peuvent télécharger des parcours depuis un serveur en ligne et ensuite y prendre part grâce au GPS intégré dans leur smartphone.

Ce rapport a pour objectif de documenter le travail réalisé et la procédure employée. Il détaille le cycle de développement, les choix de conception, les problèmes rencontrés et comment nous y avons répondu.

Introduction

This project was chosen and designed by Christopher Caroni with Adel Tihianine joining halfway through. We are both studying a DUT Informatique at the IUT A of Lille 1 University of Science and Technology. Our course requires you to complete either an internship or a project work in one of Lille 1's partner universities and we chose to take this opportunity to travel. So, we chose Glasgow Caledonian University in Scotland for our placement.

Our tutor, Professor Eddie Gray, let us choose our subjects for the project and I decided to pursue the short Android programming lessons we had at the IUT and create a real application. I had previously tried to create an application for orienteering races but the code was very messy and didn't work that well. This time I wanted to do it seriously with better coding practices and modern APIs[1]. Since you can't really use the app by itself without downloading then races from somewhere, I also decided that I should create a simple server to host the data.

But building the application was still the main focus, even if it was just a means to an end: that of learning how to program an Android app with modern APIs and programming principles by creating a real application. The main problems I wanted to be able to know how to respond to by the end of the project were the following:

- How to securely authenticate the user and his data in the app?
- How request resources from a server with secure authentication.
- How to automatically request the user's position with the smartphone's GPS in a battery-friendly way?
- How to store data on Android with minimal boilerplate code while still allowing for easy querying and updating?
- How to automate the updating of data presented in the UI while separating the UI and back-end logic?

This report will show the result of my work done throughout the project. Firstly, I'll explain the design choices leading up to the creation of the app and the server. Then I'll walk you through the development process and finally, I will present an assessment of the project and what I've learned.

Table of Contents

I. Project presentation

A) Subject choice

After arriving at GCU, Prof. Gray informed us of what our work was to be while there. We were to choose a project to work on and complete in our three months. The subject was free of choice, though of course still pertaining to our field of study. We had about a week to come up with ideas before choosing one to do as our project. From the beginning, I was pretty set on working on Android development and a server to support it. The other students who were on the exchange with me, Adel & Cyriaque, were more interested in information security and so we decided to go our separate ways. Later, Adel realized that his project was too ambitious and asked if he could join my project.

As for me, my poor attempt at creating an application for orienteering in the past made me want to finish the job. I realized how bad it was and I wanted to be able to learn how to properly create and Android app. Most importantly, I wanted to learn the tools and methods that real Android programmers use.

B) Setting the project goals

1) For the app

The basic goals of the app were to present an interface with the following features:

- Activity A:
  - Sign in/ sign up screen.
- Activity B:
  - Present a map able to display the user's position and respond to his actions.
  - Display available races on the map.
  - Progressively reveal the checkpoints of the ongoing race.
- Activity C:
  - Allow downloading of a private race by entering a key.
  - Allow downloading of public races.
- Activity D:
  - Allow downloading of an area of the map for offline racing.

As for the backend, since I wanted to employ modern Android programming principles and APIs, I would need to learn and use the following:

- Volley or Retrofit for network requests.
- A modern data storage API.
- A modern architectural pattern to separate the UI and data logic capable of respond to events.
- An API for manipulating a map that allowed offline access.

2) For the server

The server only needed a couple of front end functions:

- A sign in/sign up page.
- A page to manage and create races.

But these two entailed quite a few backend requirements:

- Use a modern framework to minimize boilerplate code and provide a working server as fast as possible.
- Host or connect to a database for:
  - Storing user data and/or login info
  - Storing the races and user records
- Provide a REST[2] API for CRUD (Create Read Update Delete) operations on the races
- Secure access to the APIs via modern authentication techniques either by implementing my own or using a professional grade solution.
- Find hosting for the server for a static online presence.

C) Conclusion

As you can see, many of the requirements are just to use modern software. This is because the main goal of the project was to learn as much as possible about real world software development and put it into practice. The programming experience was of as much importance as having a functional prototype by the end of the project.

II. Laying the foundation

Before jumping directly into the programming phase of all the features I had planned, I decided to create a foundation for both the Android and server projects. This way, I could program the two side by side and easily cross over if I needed to modify anything that required both to be in sync.

A) Setting up the work environment

1) The language

Starting out, it immediately appeared that I was going to be programming in Java. The reasons were quite simple:

- Android is based on Java and most of its APIs are designed to be called from Java.
- My previous knowledge in servers and handling web development was either in Java or in languages with Java-like syntax.
- My only in-depth knowledge of a programming language that I had enjoyed working with was in Java.
- I wanted to focus on learning about new software tools and methods rather than learn a new language.

2) The software tools

Considering that I would be working with Java on both the server and app sides, it was quite evident which IDEs I would be using.

To program the server, I had the choice between either between Eclipse and IntelliJ but I knew right away that I would be using IntelliJ. IntelliJ presents an all-around better experience for programming. It has smart code completion, automatic refactoring suggestions and facilitates debugging. Besides that, its UI is more user-friendly and easy to navigate. In theory, this promised higher productivity.

For the Android project, I chose Android Studio because it is the de facto IDE for programming in Java for Android. Plus, I already had experience with it and it is based of IntelliJ which would allow for a seamless experience when transitioning between the server and android projects.

B) Choosing the software and frameworks

1) For the server

Setting up the base for a server proved to be quite challenging. I started by trying to implement software that I had learned during my coursework but after a frustrating beginning, I had to switch and find something else.

First attempt:

For project management, I had already worked with Maven and since it provided a simple way to manage and build projects, I decided to pursue with it.

When it came to building the server, I had previously worked with Jetty, a relatively simple solution for creating a server, and Jersey, which is a Java framework for REST services.

For data storage, I was quite familiar with SQL DMBSs[3], PostgreSQL specifically which is an ORM (Object Relational Mapping) solution. Once again since I didn't want to waste too much time learning software itself, I decided to continue with it.

The idea was to create a server with Jetty and Jersey, and then build and host it via Docker. Docker is a container platform that allow your project to be built on a single configuration for fast deployment and configuration.

Figure 1: The initial software stack for my server

The problem:

Unfortunately, from the first couple of weeks, I ran into quite a few problems while trying to work with this stack.

- First, since Docker creates an entire OS as a container, it required quite a lot of startup time and took a lot of space. Therefore, the initial upload of the server would take over a couple of hours on my internet connection. On top of that, Docker runs in a VM (Virtual Machine) which required disabling some CPU configurations on my PC. This caused quite a problem since the Android development required these configurations to be enabled. It would prevent me from easily switching between the two or even changing a server configuration and then running it to check it worked properly with the application.
- Secondly, the Jetty + Jersey combo required quite a bit of xml configuration which was getting quite complicated and frustrating to understand.
- And finally, working with a SQL database was quite limiting because for every single change in my data, I would also need to modify the structure in the database. Besides, it just was not quite adapted for my needs since I had many 1 to 1 relationships for a single object in my data and referencing these relations in a SQL database is quite redundant and end up requiring more boilerplate and ultimately useless data.

Second attempt:

- Server framework:

After working with all these annoyances and setbacks for a week or two, I decided that it just wasn't worth the time overall and needed to look elsewhere.

After a little bit of research, I found an alternative to the Jetty + Jersey combo with Spring Boot. Spring Boot builds upon the Spring framework, most notable with dependency injection and aims to make it easy to create stand-alone, production-grade Spring based Applications that you can "just run". For example, all you need to create a server with Spring Boot in Java are the following lines:

```java
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Figure 2: Creating a server with the Spring framework

As you can see, we basically only need to add $@SpringBootApplication$ for and then call $SpringApplication.run()$. Then, when you run the code with Maven using $mvn\ spring\text{-}boot{:}run$, the framework automatically scans, find the annotated class and treats it accordingly.

But that was only the beginning of the good news. With Spring, you can use Spring Data and Spring Security. These two provide all the core logic for data storage, authentication, and authorization. With Spring, I only need add my configuration and most of the heavy lifting is done by the framework, effectively accomplishing the first back end objective for the server.

- Data:

Since I could now use Spring Data to automate most of my database configuration, I also decided to look for another framework for the data itself. I was interested in switching to a NoSQL DBMS because I could easily take advantage of nested classes without having to specifically code references to every object and I wouldn't even have to setup tables for the data. I found MongoDB to be an easy framework to use with plenty of documentation. MongoDB stores data in flexible, JSON-like documents. The document model maps to the objects in the application code, making the data simple to work with. Furthermore, Spring Data has direct compatibility with MongoDB with special classes adapted to using MongoDB. This would further simplify the development process for storing and accessing the data.

I also found a DBaaS[4] called mLab supporting MongoDB with a free pricing tier that fit my needs. Using a DBaaS meant that I wouldn't have to worry about the database administration or maintenance. I also wouldn't have to find out how to host the database myself which was a great advantage.

- Authentication

Since I was overhauling my whole server stack, I decided to look around and see if there was any IDaaS[5] that could handle my needs, and I ended up with Auth0. They seemed to be one of the major providers for identity management and they also had a free pricing tier that had everything I needed. They could immediately solve my whole identity stack by providing the authentication verification, a hosted user database, automatic permissions management, and APIs for presenting a login interface with both Spring Security and Android.

- Hosting:

All that was left to find was something to replace the Docker container and hosting. I turned towards the Google Cloud Platform and Amazon Web Services but after a couple of days of testing both out, I realized that they were aimed towards more professional projects where you need more control over your hosting solution.

Then I found Heroku. Heroku is a PaaS[6], that takes care of your infrastructure for you. Its main features include instant deployment via Git, a large repository of addons to extend your app's features and processes scaling for your app.

It was exactly what I needed. I didn't worry about how to package my server. All I needed to do is use Git like usual, but push it to my Heroku remote. Heroku then scans my app and automatically recognizes my Maven project and runs it. With one single command, I could deploy my server online.

Figure 3: My new stack for the server: Spring Security, Spring Boot, Spring Data, Auth0, and Heroku

As far as I was concerned, migrating to Spring had incidentally solved my three main problems with my server. I now had instant deployment, fast and simple configuration, and direct data access.

2) For the app

Once I had a barebones server with all the frameworks set up, I started working on setting down a foundation for the Android side of things.

- Network

One of the first major choices I needed to make with the app was to choose a HTTP client API for accessing data from my server. The main competitors in this area were Volley and Retrofit. Both are open source, widely used and have a lot of documentation. I ended up going with Retrofit for two main reasons:

- Retrofit can automatically parse a request into a Java Object thus removing the need to code the conversion from the JSON data manually.
- Retrofit is designed to make it easy to consume REST services which is exactly what the application will by using. You basically need only 3 Java classes to use it:
  - A Model class: whatever your JSON data will be mapped to
  - An Interface as an API to define the possible HTTP operations (GET, POST, DELETE…)

    - Another class to use *Retrofit.Builder* to create a client built to your interface operations. This can just be done in the class where you will do the request.

- Data storage

Simply storing data on the server and then downloading it on the app as needed is not an efficient solution for two main reasons:

- It puts an unnecessary load on the server
- The data downloaded is tied to the lifecycle of the activity and will need to be redownload when it gets destroyed.

So, we need an actual permanent storage solution. I already had experience using relational databases on Android but I realized I would have the same inconvenience as I had with the first builds of the server even if the amount of data I would be using would be significantly less. And so, once again I turned towards a NoSQL type of system. After some research, I only found Realm to correctly fit my needs. The Realm framework uses a container, called a *Realm*, to store objects. Here are the main reasons I chose it:

- Realms store native objects: The objects you store in a Realm are the objects you work with in the rest of your code. This means I wouldn't have to spend time designing a schema for the data.
- Realms are zero-copy: data is not copied in and out of the database to be accessed; you're working with the objects directly. This reduces boilerplate code by removing the need to have to convert the objects from ones designed for the DB to ones ready to be manipulated in the app and vice versa.

- Architectural pattern

I had already used the MVC[7] pattern in previous projects but I felt like there was a more appropriate solution in the context of an android app. Indeed, a new architecture called MVVM[8] is gaining steam in the Android community, and for good reason: data binding has recently gained official support in Android. Data binding reduces boilerplate code and having to manually update the Views.

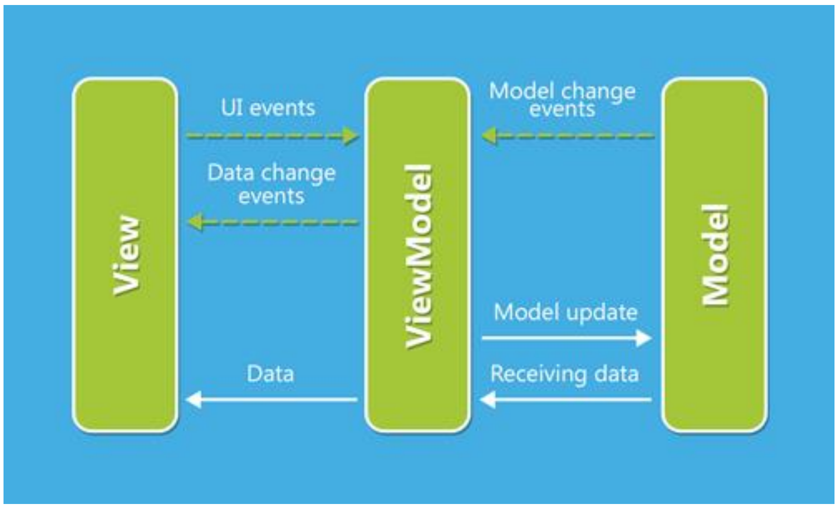


Figure 4: The MVVM pattern

The MVVM architecture (on Android) is centered around data binding which allows for reactive programming. The ViewModel is an abstraction of the View and formats the data for the View. With data binding, the View can directly bind itself to data in the ViewModel and when the ViewModel's data is updated, the View is also directly updated. With two-way data binding, the opposite can also happen: the user can interact with data in the View, and the data in the ViewModel is updated accordingly.

The main area I would be using this pattern is for the activity that would display the races or the current race and its checkpoints. This is because this activity the only part of the app where I would need present actual data from the database. If I ever decided to create a new activity just to present a list of the available races, I could also use the MVVM pattern but it wouldn't strictly be necessary. Besides, I mainly wanted to use the MVVM pattern to learn about *reactive* programming which means interacting and modifying the data rather than just viewing it.

- Maps

Once again, I found myself having to choose between an API that I already had experience with and going for something new. I had previously played around with the Google Maps API and knew it to be easy to implement with plenty of features.

Unfortunately, their map licensing was missing one important feature that was essential to my vision of the app: downloading the map for offline use. Therefore, even if I wanted to use the Google API, I had to look elsewhere for a map provider. After some research, I realized that most map providers charged for access and by number of requests. Even if they had a free pricing tier, it was too limiting to use in real circumstances. The only one who didn't charge for the type access I needed was OpenStreetMap.

Then, I found out about Osmdroid, an open source API that supported OpenStreetMap that aimed to completely replace Google's mapping solution. Since it was built around OpenStreetMap, I could just easily completely switch to it instead of having to learn how to provide the Google API with the maps from OpenStreetMap.

C) Conclusion

As you can see, I spent a lot of time just setting up and choosing what software I would be using. Though this requires a lot of initial planning, once it is done, you can focus purely on the programming part of things.

Of course, I wish I hadn't spent so much time trying to make things work with Jetty and Docker, but that allowed to understand what use cases they are more suited to. I also learned that I should have done the same amount of research on the first stack as I had done on the second before committing to it. It was my shortsightedness that cost me precious time.

But in the end, I'd say it turned out all right, since my server is functional, with all the features that I initially planned for. Besides, switching to Auth0 and a NoSQL database allowed me to take that knowledge and apply it to the Android side of the project.

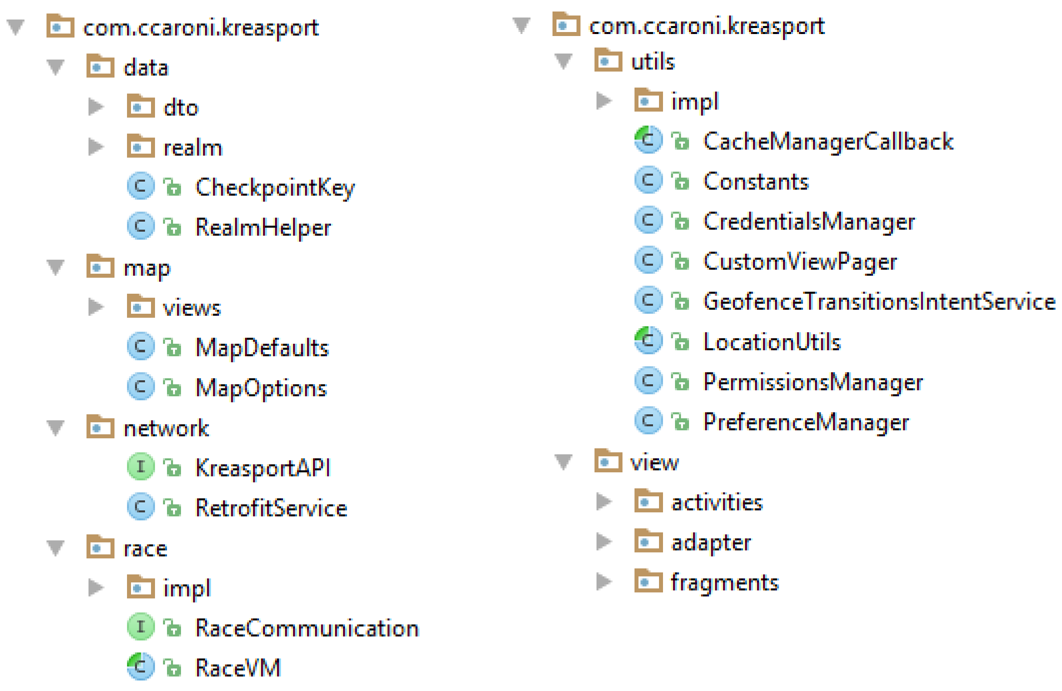
III. Development

In this section, I'll present how the project is built and explain its different components. The project is split into an Android and server subprojects. This is because they both serve a different purpose and are targeted to different platforms.

A) Android project architecture

When building the Android app, I decided to separate the Java classes by their features: if they belonged to one big set that interacted with each other, I put them in one package. Otherwise, if they could function independently, I put them in a "utility" package or another package that was common to them such as "view".

Below is an overview of the project structure with only the main sub-packages expanded. I'll go through the sub packages one by one to explain their place in the project.



```
▼ com.ccaroni.kreasport
    ▼ data
        ▶ dto
        ▶ realm
          CheckpointKey
          RealmHelper
    ▼ map
        ▶ views
          MapDefaults
          MapOptions
    ▼ network
          KreasportAPI
          RetrofitService
    ▼ race
        ▶ impl
          RaceCommunication
          RaceVM

▼ com.ccaroni.kreasport
    ▼ utils
        ▶ impl
          CacheManagerCallback
          Constants
          CredentialsManager
          CustomViewPager
          GeofenceTransitionsIntentService
          LocationUtils
          PermissionsManager
          PreferenceManager
    ▼ view
        ▶ activities
        ▶ adapter
        ▶ fragments
```

Figure 5: Android project structure

1) Data package

This package contains the POJOs[9] that compose the data that the app manipulates. They are separated into DTO[10] and Realm sub packages.

Separating the data into these two sub packages allows to restrict the data flow between the app and the server by only serving DTO objects to the server. It also helps instantly recognize in what context we are dealing with the data: a transfer with the server or in-app processing.

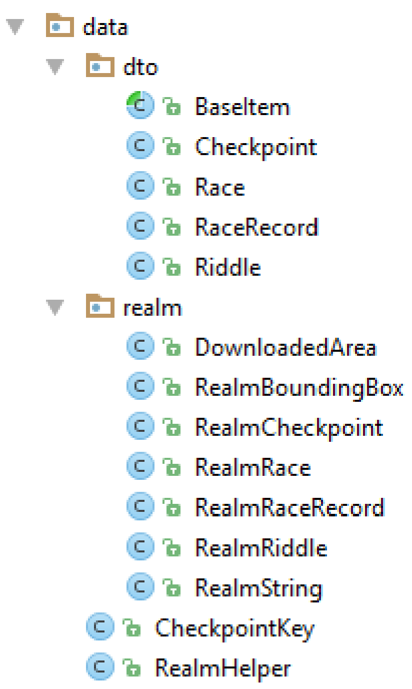

```
data
  dto
    BaseItem
    Checkpoint
    Race
    RaceRecord
    Riddle
  realm
    DownloadedArea
    RealmBoundingBox
    RealmCheckpoint
    RealmRace
    RealmRaceRecord
    RealmRiddle
    RealmString
  CheckpointKey
  RealmHelper
```

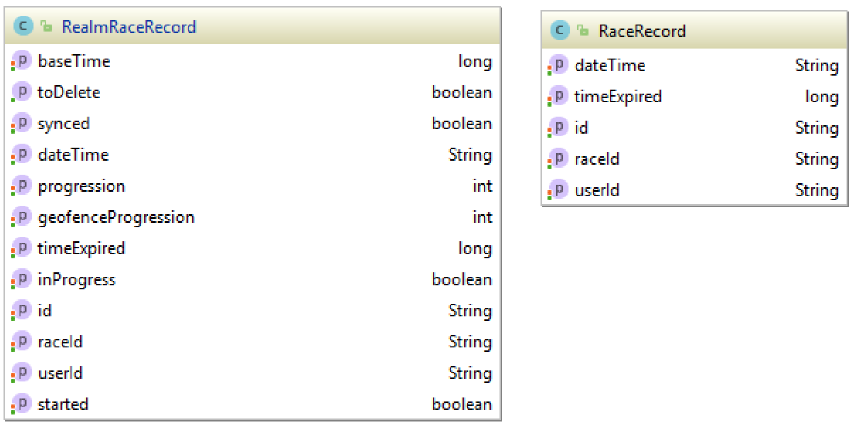
Figure 6:Data package overview

- DTO sub package

The DTO package represents plain Java classes that are just used to encapsulate data such as a race ID, a title, a date… They are just simple classes that don't depend on any framework as they are destined for data transfer between the app and the server.

- Realm sub package

The Realm sub package contains versions of the DTO classes adapted to be used with Realm. They encapsulate the same data as the DTO classes but add some more fields that are necessary only to the application context. They represent the actual objects that are to be manipulated in throughout the app. For example, *RealmRaceRecord* is a version of the DTO class *RaceRecord* but with attributes that are specific to the app and don't need to be transferred to the server.



RealmRaceRecord	
baseTime	long
toDelete	boolean
synced	boolean
dateTime	String
progression	int
geofenceProgression	int
timeExpired	long
inProgress	boolean
id	String
raceId	String
userId	String
started	boolean

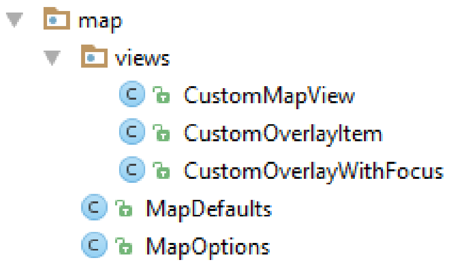
RaceRecord	
dateTime	String
timeExpired	long
id	String
raceId	String
userId	String

Figure 7:The difference between RealmRaceRecord and RaceRecord

As you can see in the figure above, the Realm version contains many more fields. For example, we need to store the user's race progression to the database so that we can access it later to verify he isn't cheating or to know what the next checkpoint he needs to target is. This corresponds to the integers *progression* and *geofenceProgression*. Another example is the boolean *synced*. These fields do not need to be stored on the server so it only makes sense to make two separate objects since they are for different purposes.

2) Map package

This package groups the classes that are specific to the mapping features of the app and don't depend on any specific implementation in this project. This allows for future extension or even use beyond this project.



```
map
  views
    CustomMapView
    CustomOverlayItem
    CustomOverlayWithFocus
  MapDefaults
  MapOptions
```

Figure 8: Map package overview

- #### Views sub package

The "views" sub package contains custom classes extending the features already presented by the Osmdroid API. More specifically, they represent the actual visual objects behind the Android views. By using custom classes, I can continue using the Osmdroid feature set but add our own improvements. For example, in the *CustomOverlayItem* class, I added attributes for race and checkpoint IDs. This way, when the user taps on an item on the map, I can access these attributes and have a reference to which race or checkpoint it corresponds to.

- #### MapDefaults & MapOptions

This package contains two POJOs that facilitate the use of the *CustomMapView* class. They allow for easy toggling of options within a *CustomMapView* object.

Here is an example of how they are used in the project:

```
// enable some base options for the map
MapOptions mMapOptions = new MapOptions()
        .setEnableLocationOverlay(true)
        .setEnableMultiTouchControls(true)
        .setEnableScaleOverlay(true);

// set default coordinates to Lille
MapDefaults mMapDefaults = new MapDefaults(new GeoPoint(50.633621,
3.0651845), 9);

// initialize the map
mMapView = new CustomMapView(this, mMapOptions, mMapDefaults);
```

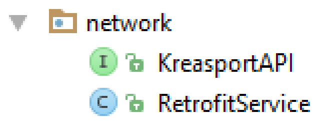
Figure 9: Initializing a map object with our custom implementation

As you can see, the *MapOptions* class allows for easy chaining of options and the *MapDefaults* class sets a default location for the map. Then we create a *CustomMapView* object, my custom implementation of the Osmdroid *MapView* class, with these two.

The constructor automatically applies the relevant options from MapOptions and MapDefaults. This allows us to separate the coding of the options from the map instantiation as well as their automatic application without having to manually code it every single time. This massively reduces the amount of code if we need to display a map in different areas of the application because with the original Osmdroid code, you'd have to manually enable each feature for each new *MapView* instantiation and the code is pretty verbose.

3) Network package

This package contains the necessary classes to create a Retrofit client and an API specific to our server. As you can see below, with Retrofit, we only need two Java classes for configuration and the third class just implements the network request as it sees fit.



network
KreasportAPI
RetrofitService

Figure 10: Network package overview

- #### RetrofitClient

This class serves to create a *Retrofit* object and then an implementation of the *KreasportAPI* class. With Retrofit, creating an instance of your API is child's play:

```
Retrofit retrofit = new Retrofit.Builder()
        .baseUrl(baseUrl)
        .addConverterFactory(GsonConverterFactory.create())
        .build();

KreasportAPI kreasportAPI = retrofit.create(KreasportAPI.class);
```

Figure 11:Creating a Retrofit API

- #### KreasportAPI

This class defines the API for communicating with the server. With Retrofit, defining an HTTP request is as simple as defining a java method annotated with the HTTP request type and endpoint. Below are some requests used in the project:

```
@POST("/records")
Call<Void> uploadRaceRecord(@Body RaceRecord raceRecord);

@DELETE("/records/{id}")
Call<Void> deleteRaceRecord(@Path("id") String recordId);

@GET("/records/{id}")
Call<RaceRecord> getRaceRecord(@Path("id") String id);
```

Figure 12: Defining a Retrofit API

Using the API is even simpler:

```
KreasportAPI kreasportAPI = RetrofitService.getKreasportAPI(true,
accessToken);
Response<List<Race>> races = kreasportApi.getPublicRaces().execute();
```

Figure 13:Using a Retrofit API

In the example above, in just two lines we create an instance of our API, send a request, and get the result. All the networking itself and the conversion from the JSON in the body of the response to a *List<Race>* object is done automatically.

4) Race package

The *race* package contains the classes necessary to manipulate the state of a race and the recording associated with it.

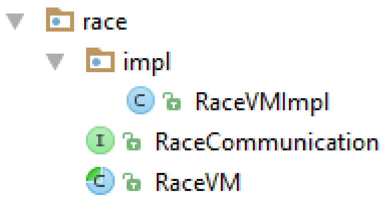

```
▼ race
    ▼ impl
          RaceVMImpl
      RaceCommunication
      RaceVM
```

Figure 14: Race package overview

- #### RaceVM

This is an abstract class that contains base attributes and methods needed to act as a ViewModel. It serves as a contract to expose the methods needed by the Model and the View. These methods are abstract so that this class only holds methods which the Model and the View need to get necessary data. The real code that reacts to events in this class is decided by the implementing class.

- #### RaceVMImpl

This class extends the *RaceVM* class and decides the specific actions to execute following a user interaction or *Model* update. Here are some of the methods it implements:

```
/**
 * Call to stop the current race. Used by passiveBottomSheet.
 */
public abstract void onStartClicked();

/**
 * Call to stop the current race. Used by activeBottomSheet.
 */
public abstract void onStopClicked();
}

/**
 * Call from the activity to confirm that user wants to stop in response to
{@link RaceCommunication#confirmStopRace()}
 */
public abstract void onConfirmStop();
```

Figure 15: RaceVMImpl overrides

5) Utils package

This package contains one-off classes that only serve one purpose and that can fit in multiple situations without depending on other classes.

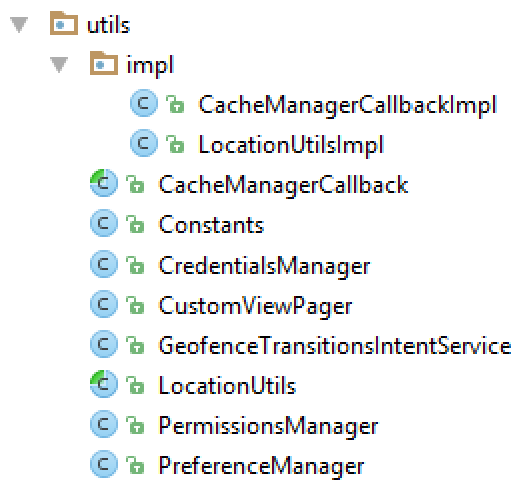

```
utils
    impl
        CacheManagerCallbackImpl
        LocationUtilsImpl
    CacheManagerCallback
    Constants
    CredentialsManager
    CustomViewPager
    GeofenceTransitionsIntentService
    LocationUtils
    PermissionsManager
    PreferenceManager
```

Figure 16: Utils package overview

- Managers

The classes named *AbcManager* are exactly that: they manage a specific feature with public methods to access them. For example, *PermissionsManager* serves as the contact point for verifying and asking for permissions within this app.

Using such *Managers* helps reduce redundant code by putting it one class that can be reused multiple times from different situations.

- Impl sub package

This holds the classes that implement specific methods from their interface or abstract class. Using such a pattern of interfaces with separate implementations allows to expose base methods that can be used from other classes without having to refactor them if we wish to extend or change how we want these methods to be coded.

For example, the abstract *LocationUtils* class only holds one method that automatically notifies its attached class when it receives a location and three abstract methods to access location. This abstraction means we can easily switch between ways to get the device's location without having to change what the main class calls to access the location.

Whether we want to use the Google API or just the base Android one, all we need to do is change the code in the implementation of *getLastKnownLocation()* or *startLocationUpdates()*. Since main class accessing the location doesn't have any knowledge of the logic behind them, it doesn't have to be refactored itself.

Here are the three abstract methods of the *LocationUtils* class:

```
/**
 * Just a simple call to {@link
LocationCommunicationInterface#onLocationChanged(Location)} to notify the
class that this is attached to.
 *
 * @param location the new location
 */
@Override
public void onLocationChanged(Location location) {
    mLocationReceiver.onLocationChanged(location);
}

public abstract Location getLastKnownLocation();

/**
 * Stops calling for location updates.
 * Depending on what implementation decided on, the location updates may
only need to be called on a certain point in time in the lifecycle of the
{@link Context}.
 */
public abstract void stopLocationUpdates();

/**
 * Starts calling whatever implementation decided on to give location
updates.
 * Depending on what implementation decided on, the location updates may
only need to be called on a certain point in time in the lifecycle of the
{@link Context}.
 */
public abstract void startLocationUpdates();
```

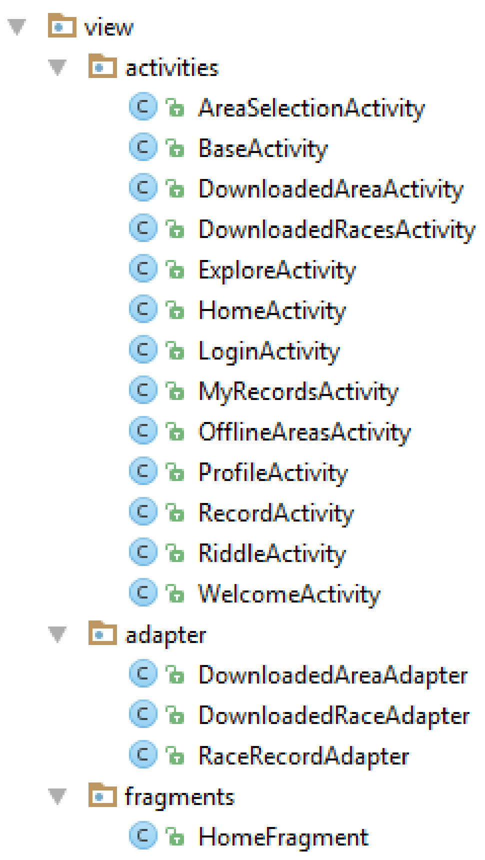
Figure 17:LocationUtils abstraction

6) Views package

This package holds all the Android Activities, Fragments and front end logic that make the application. Since the activities and fragments are just simple class that initialize the view itself and set up navigation throughout the app, I will not go into detail about how they work. They all are, except for *ExploreActivity* which works in conjunction with *RaceVM* with the MVVM pattern, just simple classes that follow the base Android standard.

As such, you'd think that overall, the views package wouldn't have taken much time. But since I wanted to use the MVVM pattern with two-way databinding as much as possible for *ExploreActivity*, building the view and coordinating its appearance with the viewmodel took more time than you'd expect. Since the view's appearance and layout depended on data held in the viewmodel, I had to properly program the updating of the data for all the different situations the user could be in.

Another thing that I can't show with the code, is the time it took to learn how to make basic activities with proper Android styles and themes. This may not seem an essential part of the project but it was part of my goal to learn how to create a modern app.



```
view
  activities
    AreaSelectionActivity
    BaseActivity
    DownloadedAreaActivity
    DownloadedRacesActivity
    ExploreActivity
    HomeActivity
    LoginActivity
    MyRecordsActivity
    OfflineAreasActivity
    ProfileActivity
    RecordActivity
    RiddleActivity
    WelcomeActivity
  adapter
    DownloadedAreaAdapter
    DownloadedRaceAdapter
    RaceRecordAdapter
  fragments
    HomeFragment
```

Figure 18: Views package overview

B) Server project architecture

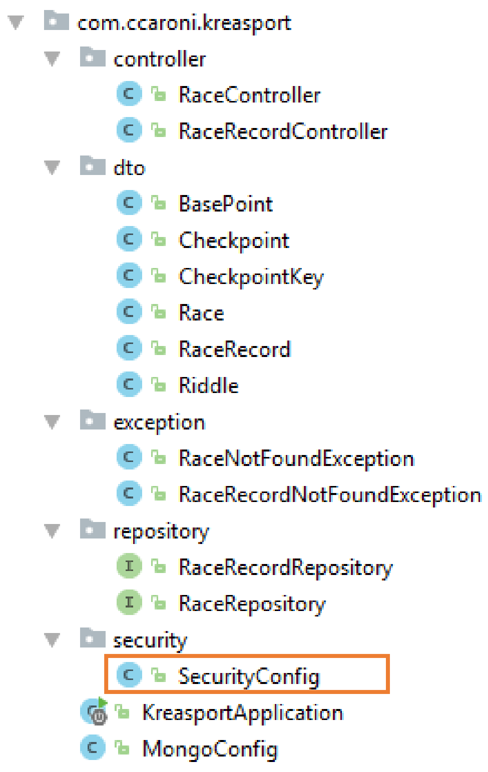
Building a server for both a backend and a front-end presence proved to be quite difficult with the goals I had in mind. I needed a REST backend using my Auth0 authentication service as well a webpage front-end using the same Auth0 service. The problem was that the Auth0 API for Android stores the user's credentials with JWTs[11] but the Auth0 APIs for webpages and Spring Security used the browser's session to store a cookie once the user was authenticated.

That meant that I would have to configure Spring security to accept both methods of authentication. After several tutorials on how to do this, I finally understood the concept and tried to apply it to my case. But after many hours of trying, I realized that the Maven project just could not accept both configurations at once. As the end of the project was nearing, I decided that I should just split the server into two to separate both features. Consequently, I would have one server for a web interface to manage the races and another server to manage requests from the Android app.

Since both must interact with the MongoDB database and use Spring Security, the front-end server is basically an extension of the backend server with a modified security configuration. So, I'll present the backend server and then show what the front-end one adds on.

1) Backend server

As I said earlier, the backend server presents all the necessary code for the Android app to send requests to. The only class specific to this server is the *SecurityConfig* even though the front-end version contains a security configuration class as well. The only difference is that they use different Maven dependencies and therefore need a different build.



```
com.ccaroni.kreasport
    controller
        RaceController
        RaceRecordController
    dto
        BasePoint
        Checkpoint
        CheckpointKey
        Race
        RaceRecord
        Riddle
    exception
        RaceNotFoundException
        RaceRecordNotFoundException
    repository
        RaceRecordRepository
        RaceRepository
    security
        SecurityConfig
    KreasportApplication
    MongoConfig
```

Figure 20: Backend project overview

i. Controller package

With Spring MVC, requests are dispatched to handlers which are based on the *@Controller* and *@RequestMapping* annotations. These annotations are added to classes or methods, thus giving them the role of a controller, which then decide how to respond to requests.

This package contains the controllers which give CRUD access to the *Race* and *RaceRecord* objects in the database. Since the authentication is automatically handled beforehand, this package focuses solely on treating and responding to HTTP requests.

Below is an example of how a controller is used in the *RaceController* class. As you can see, the class is annotated with *@RestController* which extends the features of *@Controller* by adding *@ReponseBody*. This indicates that the method return value should be bound to the response body. For example, the *createRace()* method has a return type of *ResponseEntity<?>*

which means that the content of the response will be a serialized ResponseEntity<?>. In this case, the location of where the race is stored.

```java
@RestController
@RequestMapping("/races")
public class RaceController {

    private final RaceRepository raceRepository;

    @Autowired
    public RaceController(RaceRepository raceRepository) {
        this.raceRepository = raceRepository;
    }

    @RequestMapping(method = POST)
    public ResponseEntity<?> createRace(@RequestBody Race race) {
        raceRepository.save(race);
        URI location = ServletUriComponentsBuilder
                .fromCurrentRequest().path("{id}")
                .buildAndExpand(race.getId())
                .toUri();
        return ResponseEntity.created(location).build();
    }
}
```

Figure 21:Spring controller as used in the server

ii. Dto package

This package groups the POJOs that map our data. With the Spring Data framework for MongoDB, we can use these POJOs to map to the data in the database with minimal configuration. In the example below, we have a POJO, *RaceRecord*, with an example of what you need to store an object with Spring Data and MongoDB:

```java
@Document(collection = "raceRecords")
public class RaceRecord {

    @Id
    private String id;
}
```

Figure 22:Excerpt from RaceRecord showing all that is necessary to use Spring Data & MongoDB

The *@Document* annotation indicates that one instance of this object should map to one document in the database, inside the *raceRecords* collection. The *@Id* annotation indicates that the field *id* should act as an ID for the document. In MongoDB, a document ID is any object that can serve as a unique reference to this object. In our case, we use a String for simple manipulation and access. To ensure uniqueness, we simple instantiate it with a random value on object creation.

But unbelievably, these two annotations are not even necessary. Spring Data can automatically recognize that it should save an object as a document if the object is called with the Spring Data *save* method. The *@Id* annotation can be removed as well, but for a different reason that I'll explain in the repository section below.

iii. Exception package

This package contains two simple classes that correspond to Java Exceptions but with the Spring *@ResponseStatus* annotation. This enables us to respond to a request with the desired HTTP status quickly and effortlessly from anywhere in the server. Here is an example of it in use:

```
@RequestMapping(path = "/{id}", method = GET)
public Race getRaceById(@PathVariable("id") String id) {
    validateRace(id);
    return raceRepository.findById(id).get();
}

private void validateRace(String id) {
    if (!raceRepository.existsById(id))
        throw new RaceNotFoundException(id);
}
```

Figure 23: Exception handling in practice

In this example, the *getRaceById()* method acts as a controller and is supposed to return a Race. Before getting the race from the database, we verify if it even exists with *validateRace()* so as not to respond with empty content. In this method, if we find that the desired race does not exist, we throw a *RaceNotFoundException*. Here is the code for it:

```
@ResponseStatus(value = HttpStatus.NOT_FOUND, reason = "Could not find
race")
public class RaceNotFoundException extends RuntimeException {

    public RaceNotFoundException(String id) {
        super("Could not find race with id:" + id);
    }
}
```

Figure 24:Defining Exceptions with @ReponseStatus

After we throw the exception, the Spring framework detects the *@ResponseStatus* and ends the http request by sending the http response code and text specified in the annotation.

Using the *@ReponseStatus* annotation with Java Exceptions allows us to reduce duplicate code since we can just throw the exception whenever we happen on the error instead of manually coding the response.

iv. Repository package

The repository package contains the classes that allow us to communicate with the database. But with Spring Data, if we only need direct, simple CRUD operations on the data mapped in the database, we don't have to code anything. Here is the full file of *RaceRepository*:

```
@Repository
public interface RaceRepository extends MongoRepository<Race, String> {}
```

Figure 25: Full code for RaceRepository

Since we extend the generic interface *MongoRepository* with our dto object *Race* as a type parameter, we have access to over ten predefined CRUD methods on our *Race* objects in the database. As shown in the controller and exception sections beforehand, we can use the *save()* and *exists()* methods without having to code them ourselves.

In the dto section above, I mentioned that we don't even need to use the *@Id* annotation to indicate which field should serve as the document ID. This is because the *MongoRepository* already contains methods that operate on a field called *id* such as *findById()* or *deleteById()*. Since these methods are mapped to operate on the field named *id* in the database and the *@Id* annotation simply tells MongoDB to save the annotated field with the name *id* in the database, we can just name our field *id* and end up with the same result. This is because by default MongoDB saves the fields in the documents as they are named in your code.

v. Other

The other classes that haven't been covered by the previous section are *SecurityConfig*, *MongoConfig*, and *KreasportApplication*. The last one is a simple class with a *main()* method to run the server.

• MongoConfig

MongoConfig as you can guess is just a class that configures Spring Data to automatically connect our mLab MongoDB server. And Spring Data, this is quite trivial:

```
@Configuration
@EnableMongoRepositories(basePackages = "com.ccaroni.kreasport.repository")
public class MongoConfig extends AbstractMongoConfiguration {

    @Override
    protected String getDatabaseName() {
        return "kreasport-mongodb";
    }

    @Override
    public MongoClient mongoClient() {
        String MONGO_URI = System.getenv("MONGO_MLAB");
        MongoClientURI uri = new MongoClientURI(MONGO_URI);
        return new MongoClient(uri);
    }
}
```

Figure 26:Full code for MongoConfig

Annotating this class with *@Configuration* allows the Spring framework to load this class on server startup. Extending the *AbstractMongoConfiguration* means that Spring will use this class as the configuration to automatically connect to the MongoDB database.

As for connecting to the database itself, all we need to do is load an environment variable that has the credentials and address of the mLab server and use it to create a *MongoClient*. I used an environment variable so that the credentials and the address of the server aren't exposed in the code.

- SecurityConfig

With Auth0 and Spring Security, securing access to the resources, or the server for that matter, becomes child's play. Here is an example of what a basic configuration for the server could look like:

```java
@EnableWebSecurity
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Value(value = "${auth0.audience}")
    private String apiAudience;
    @Value(value = "${auth0.issuer}")
    private String issuer;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        JwtWebSecurityConfigurer
                .forRS256(apiAudience, issuer)
                .configure(http)
                .authorizeRequests()
                .antMatchers(HttpMethod.GET, "/login").permitAll()
                .antMatchers(HttpMethod.GET, "/races/**").hasAuthority("read:races")
                .antMatchers(HttpMethod.POST, "/races/**").hasAuthority("create:races")
                .antMatchers(HttpMethod.PUT, "/races/**").hasAuthority("update:races")
                .antMatchers(HttpMethod.DELETE, "/races/**").hasAuthority("delete:races")
                .anyRequest().authenticated();
    }
}
```

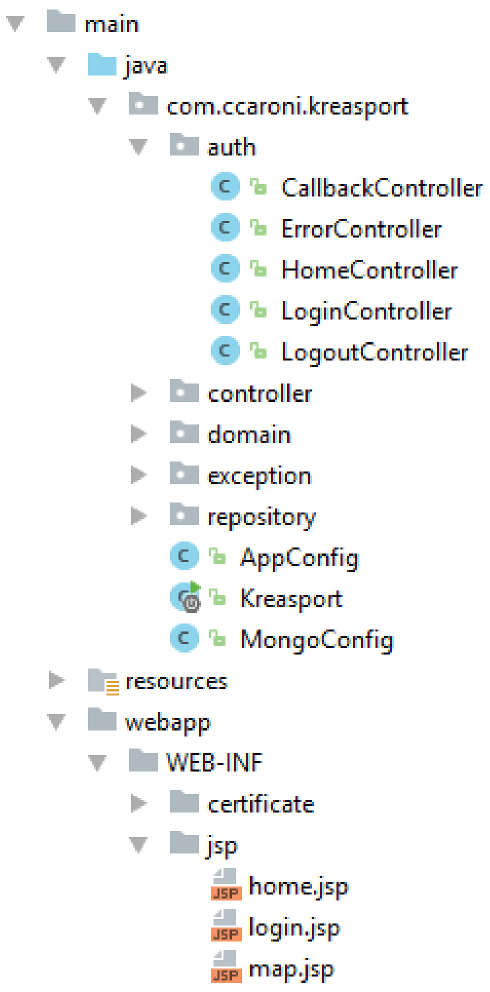
Figure 27: Possible security configuration for the server with Auth0

@EnableWebSecurity tells the Spring framework that this class will sever as configuration for the security. Extending *WebSecurityConfigurerAdapter* gives us access to the *configure()* method which allows us to customize access to the server, in this case, with JWTs and URLs.

In the implementation of this method above, we tell Spring to secure the server with JWTs, based on the RS256 algorithm, against an issuer and an API. This means it verifies the JWT's signature against the issuer. The issuer and the API in the example above are automatically loaded from a text configuration file. Next, we use the *antMatchers()* method to allow access to the */races/*** URLs based on types of HTTP requests. For example, we only allow POST requests if the JWT in the request has the *create:races* scope.

2) Front-end server

As I mentioned before, the front-end server basically builds upon the backend server. This means that the only changes it makes are a different security configuration using a different Maven dependency and jsp webpages with their associated controllers. The JSPs are contained in a *webapp* folder which Spring automatically adds to its classpath.



```
▼ main
    ▼ java
        ▼ com.ccaroni.kreasport
            ▼ auth
                    CallbackController
                    ErrorController
                    HomeController
                    LoginController
                    LogoutController
            ▶ controller
            ▶ domain
            ▶ exception
            ▶ repository
                AppConfig
                Kreasport
                MongoConfig
    ▶ resources
    ▼ webapp
        ▼ WEB-INF
            ▶ certificate
            ▼ jsp
                    home.jsp
                    login.jsp
                    map.jsp
```

Figure 28: Front-end overview

Above is an overview of the front-end project. As you can see the *SecurityConfig* class is replaced by *AppConfig* and its associated controllers in the *auth* sub package to serve the login and home JSPs. These controllers also allow us to intercept the requests to these pages and do some processing.

i. Security

First off, the *AppConfig* extends the Auth0 security class *Auth0Config* which itself extends *WebSecurityConfigurer*. This means we can automatically secure access to the server by the URLs by using the same Auth0 API as for the backend server.

Since the user is automatically authenticated thanks to *AppConfig*, the controllers in *auth* only need to respond to requests by serving the appropriate pages and some additional processing for the http session. For example, *LogoutController* receives requests on the */logout* URL, invalidates the http session which effectively logs out the user, and then redirects to */login*. *LoginController* simply receives requests on */login* and serves up the login page with the Auth0 API in response.

ii. Webpages

- Login

The login page is just a façade for the Auth0 API. It sets up an Auth0 widget configured to authenticate against our own Auth0 API.

```
<script type="text/javascript">
    $(function () {
        var lock = new Auth0Lock('${clientId}', '${clientDomain}', {
            auth: {
                redirectUrl: '${fn:replace(pageContext.request.requestURL,
pageContext.request.requestURI, '')}${loginCallback}',
                responseType: 'code',
                params: {
                    state: '${state}',
                    scope: 'openid roles user_id name nickname email
picture'
                }
            }
        });
        setTimeout(function () {
            lock.show();
        }, 1500);
    });
</script>
```

Figure 29:Login code

As you can see, we simply initialize an *Auth0Lock* object with our configuration and display it. We tell it to authenticate against our Auth0 API at the *clientDomain* address with the id *clientId*. The widget takes care of securely sending the request to our Auth0 API which redirects back to our server with the authentication result.

- Map

This is the webpage that Adel worked on. It allows authorized users to create races and upload them to the database. Here we use the Google Maps API to display the map since we don't need any special features like the Android app.

Unfortunately, Adel hasn't completely finished the page. For example, you must enter exactly five checkpoints to save a race to the database. Also, all the checkpoint objects are hard coded and all have the same ID. This means that there is no way to differentiate between checkpoints. So even if you can save the race to the server, if you download it on the app, the app won't be able to get a specific checkpoint and display it.

C) Conclusion

In the development phase, I would switch between the server and the Android projects whenever I needed features to be present on both platforms for simultaneous testing. But since the only things that really needed to be coordinated were the DTO objects, most of the development time was spent on the Android project. Moreover, since the server was mostly for backend functions, using the Spring framework meant that once I had the foundation, I already had most of what I needed. Additionally, I decided to spend more time on the Android project because, after all, it was the main objective. I may not have had as big problems on the Android project as I had on the server, but it did take a long time to learn the modern programming principles and Android APIs to create a good enough looking application.

IV. Conclusion

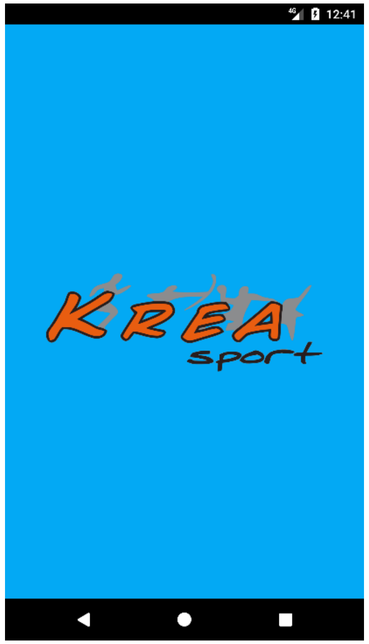
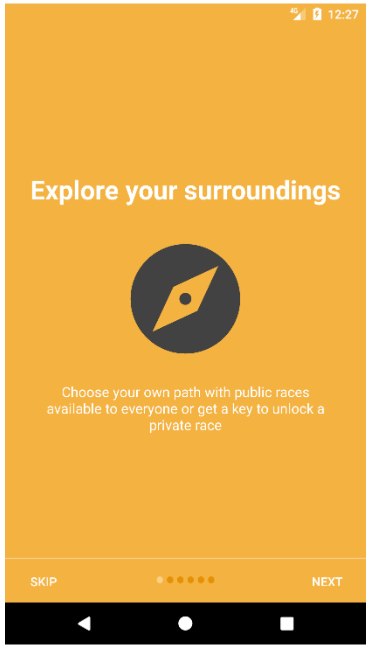
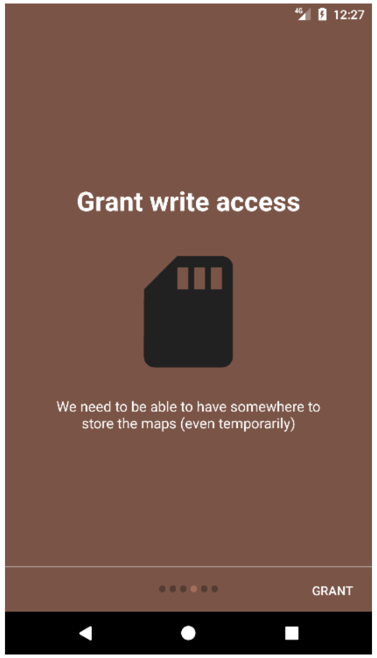
Looking back at my work, I can say I am quite pleased with what I did. This project has been a great learning experience. Besides learning about modern programming principles and how to apply them in practice, I also learned how to analyze my problems and search for solutions. Having a fully functional server with all the features I wanted proved to be quite a challenge but allowed me to discover Spring's extensive framework, NoSQL databases with MongoDB, and Auth0 for professional, secure authentication. Since the programming world moves so fast, I feel that learning how to use these modern frameworks constitute valuable knowledge.

Besides, I can truthfully say that all the front facing features that I originally planned for, besides the webpage to create races that Adel made, are working. And more importantly, I reached all my backend targets whether they were learning MVVM for Android, learning a modern framework to create servers or learning how to securely authenticate users on multiple platforms.

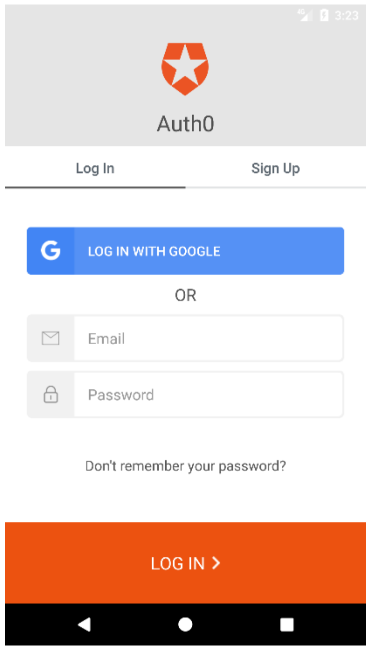
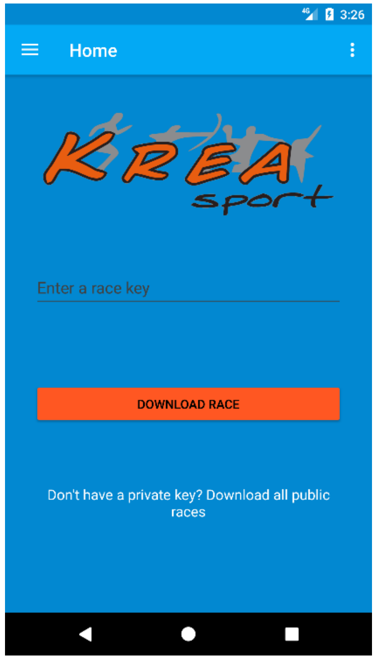
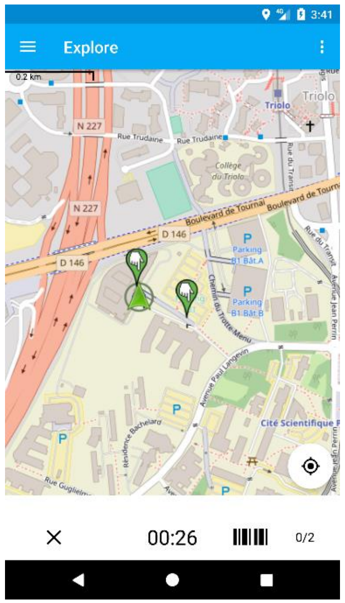
Overall, I am quite satisfied with the result of this project. If I wanted to add features to the app for the user to fully manage his records, or save his GPS tracks, I have a solid base with clean code that I can build on. I feel that this is more important than just churning out features with a tangled project structure.

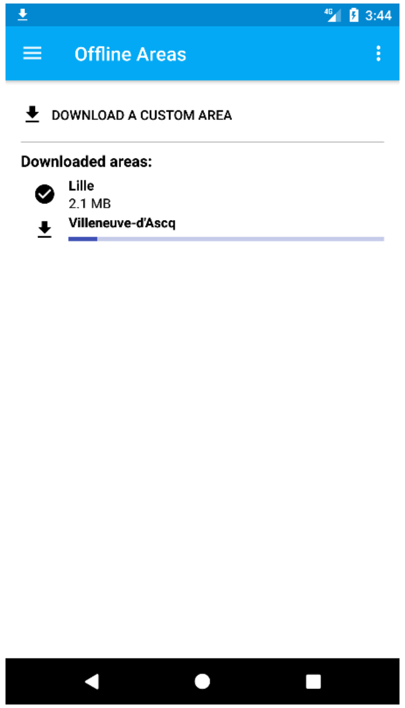
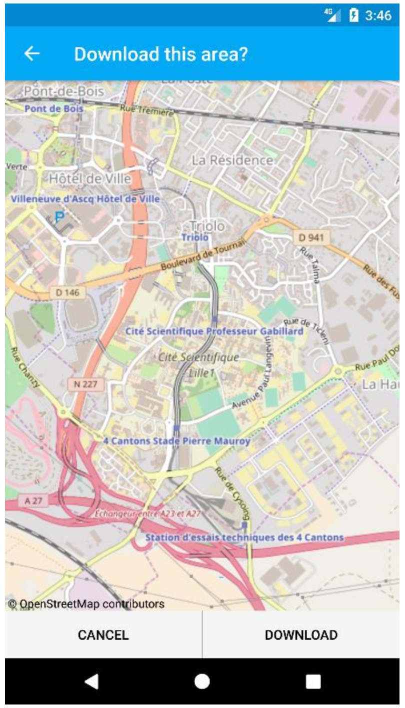
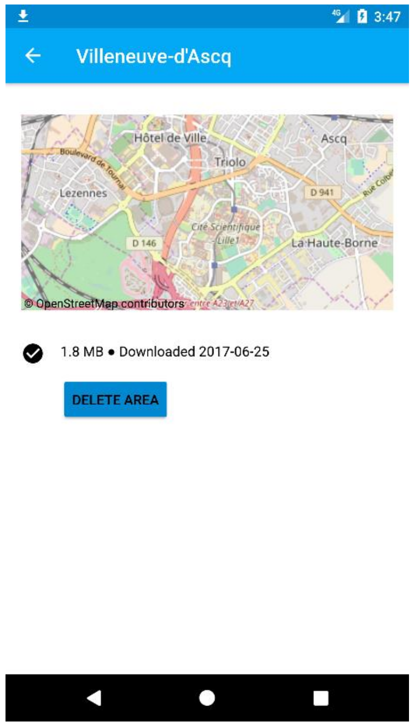
Appendices

How to use the Android app: introduction

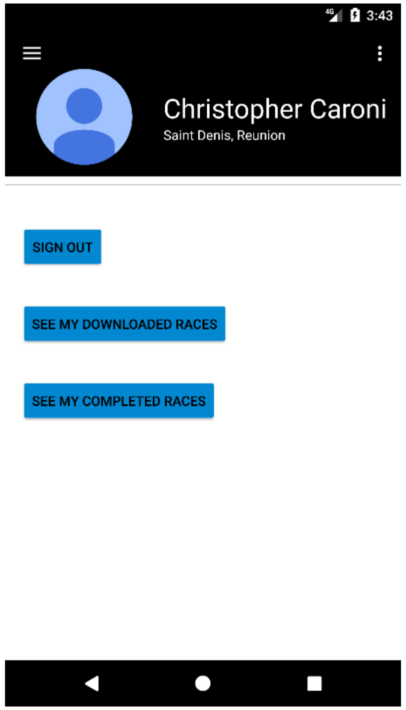
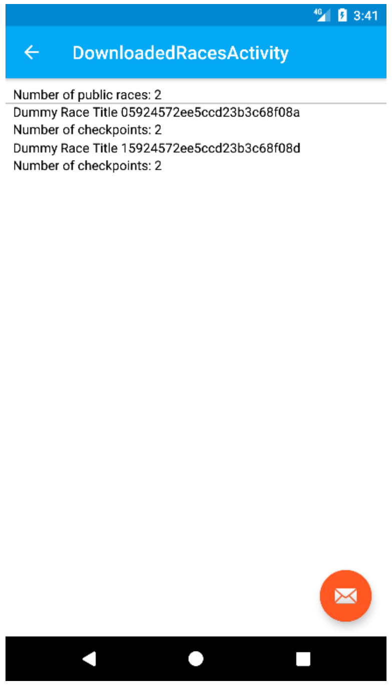
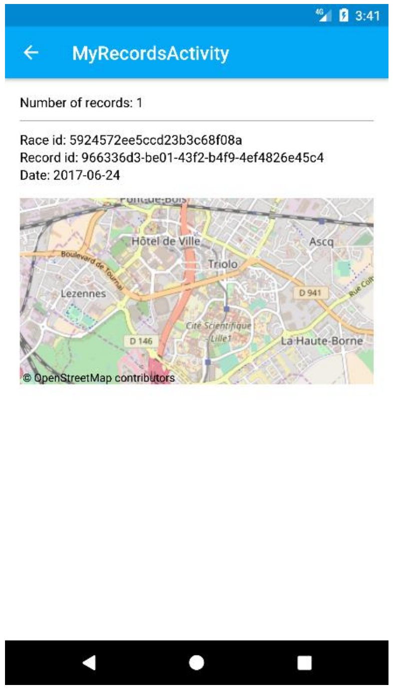
Splash screen:	Welcome screen:	Welcome screen, continued:
 When the app is launched, it displays this screen until the next activity is loaded. This is part of the Material guidelines to help provide a unique experience for the app and allows for branding.	 Explore your surroundings Choose your own path with public races available to everyone or get a key to unlock a private race The first time the user launches the app, he is presented with an introduction of the app that explains what the app is for.	 Grant write access We need to be able to have somewhere to store the maps (even temporarily) After the introduction screens, the user is asked to grant permissions to the app with an explanation of why the app needs them. We also present this screen every time the app is launched if it detects that the permissions aren't already granted.

How to use the Android app: main activities

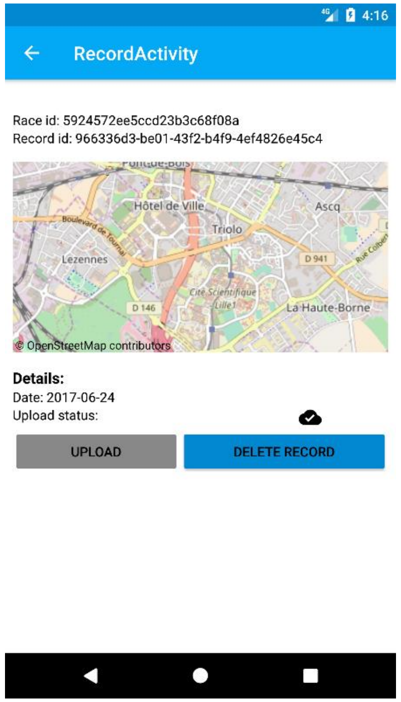
Login Screen:	Main Activity:	Explore:
 Auth0 Log In Sign Up LOG IN WITH GOOGLE OR Email Password Don't remember your password? LOG IN >	 Home KREA sport Enter a race key DOWNLOAD RACE Don't have a private key? Download all public races	
When the user launches the app, if his credentials aren't already stored or are expired, he must first authenticate himself to use the app.	This is the "home" screen. Here the user can download a specific race if he has the ID, or he can download all public races.	This activity displays the map and is where the user can engage in a race. In this example, a race is ongoing with the timer at the bottom. The "X" button stops the race. When no race is ongoing, the map displays all the races instead of the checkpoints for the single race. The bottom can slide upwards and display a description of the currently selected race or checkpoint.

Offline areas:	Offline area selection:	Managing an offline area:
 3:44 Offline Areas DOWNLOAD A CUSTOM AREA Downloaded areas: Lille 2.1 MB Villeneuve-d'Ascq	 3:46 Download this area? © OpenStreetMap contributors CANCEL DOWNLOAD	 3:47 Villeneuve-d'Ascq © OpenStreetMap contributors 1.8 MB • Downloaded 2017-06-25 DELETE AREA
This activity displays all the areas that the user has downloaded, with the ongoing downloads as well, as you can see in the example. The button at the top leads to the next screen:	On this screen, the user simply pans and zooms to the desired area to download. The download button at the bottom starts the download for the area shown on the map.	From the "offline areas" screen, the user can click on an area to manage it which leads to this screen. Here the user can view the area this download covers, the date it was downloaded, and its size on the SD card. He can also delete the downloaded area if doesn't need it anymore.

How to use the Android app: secondary activities

User profile:	Downloaded races:	My race records:
 3:43 Christopher Caroni Saint Denis, Reunion SIGN OUT SEE MY DOWNLOADED RACES SEE MY COMPLETED RACES	 3:41 DownloadedRacesActivity Number of public races: 2 Dummy Race Title 05924572ee5ccd23b3c68f08a Number of checkpoints: 2 Dummy Race Title 15924572ee5ccd23b3c68f08d Number of checkpoints: 2	 3:41 MyRecordsActivity Number of records: 1 Race id: 5924572ee5ccd23b3c68f08a Record id: 966336d3-be01-43f2-b4f9-4ef4826e45c4 Date: 2017-06-24 Hôtel de Ville Ascq Triolo Lezennes D 941 Cité Scientifique D 146 La Haute-Borne © OpenStreetMap contributors
Here, the user can log out from the app or view his downloaded races or records. The image at the top is loaded from the internet if he logged in using a Google account. The name and location are also displayed if they are available.	This activity displays all the races that the user has downloaded. I did not originally plan this feature, which is why it looks very basic, but quickly added it while working on the next activity because it required the same core code.	Here, the user can view and manage the races he has completed. Clicking anywhere on one record in the list (there is only one item in the example) leads to the next activity.

Managing a record:



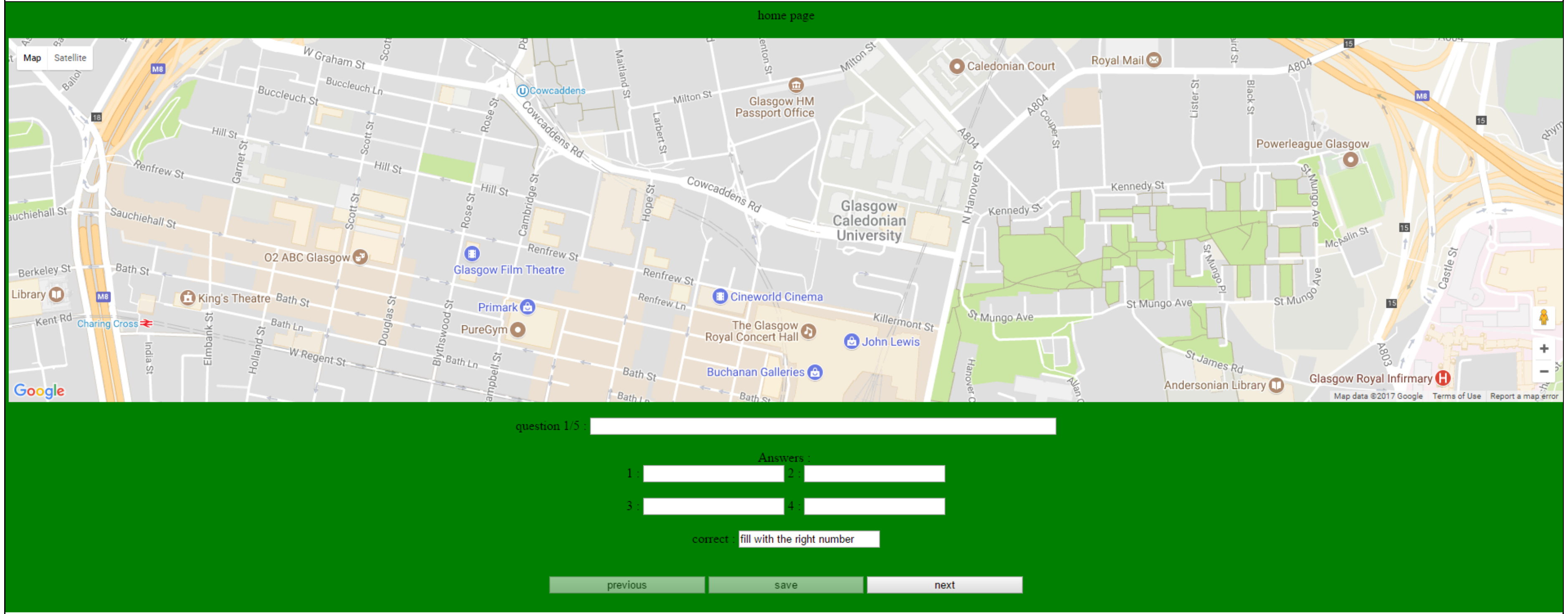
Race id: 5924572ee5ccd23b3c68f08a
Record id: 966336d3-be01-43f2-b4f9-4ef4826e45c4

Details:
Date: 2017-06-24
Upload status:

This activity allows the user to view the basic information about the recording and a map of where the race was.
He can manually upload it to the server or delete it from the device and server.

How to use the webpage:

Creating a race


home page

question 1/5 :

Answers :

1 : 2 :

3 : 4 :

correct : fill with the right number

This is the page that Adel worked on.
On this page, the administrator can create races and then save them to the database.
You click on the map to set a checkpoint and then enter in the question and answers for that checkpoint. Once you have all the checkpoints you want, you click the save button which sends a request to the server to save the race created.

Bibliography

Android Open Source Project. (n.d.). *Creating and Monitoring Geofences*. Retrieved from Android Developers: https://developer.android.com/training/location/geofencing.html

Android Open Source Project. (n.d.). *The Activity Lifecycle*. Retrieved from Android Developers: https://developer.android.com/guide/components/activities/activity-lifecycle.html

Auth0. (n.d.). *JavaScript Login*. Retrieved from Auth0: https://auth0.com/docs/quickstart/spa/vanillajs

Auth0. (n.d.). *Spring Security Java API Getting Started*. Retrieved from Auth0: https://auth0.com/docs/quickstart/backend/java-spring-security

Osmdroid. (n.d.). *Osmdroid wiki*. Retrieved from Github: https://github.com/osmdroid/osmdroid/wiki

Pelgrims, K. (2017, March 27). *Data Binding in the Real World*. Retrieved from Realm: https://news.realm.io/news/droidkaigi-kevin-pelgrims-data-real-world-data-binding/

Pivotal. (n.d.). *Accessing Data with MongoDB*. Retrieved from Spring: https://spring.io/guides/gs/accessing-data-mongodb/

Pivotal. (n.d.). *Accessing MongoDB Data with REST*. Retrieved from Spring: https://spring.io/guides/gs/accessing-mongodb-data-rest/

Pivotal. (n.d.). *Building a RESTful Web Service*. Retrieved from Spring: https://spring.io/guides/gs/rest-service/

Realm. (n.d.). *Realm Java*. Retrieved from Realm: https://realm.io/docs/java/latest/

Vogel, L., Scholz, S., & Weiser, D. (2017, June 02). *Using Retrofit 2.x as REST client - Tutorial*. Retrieved from Vogella: http://www.vogella.com/tutorials/Retrofit/article.html

Glossary

[1] API: Application Programming Interface, a set of subroutine definitions, protocols, and tools for building application software.
[2] REST: Representational State Transfer, a set of design principles for making network communication more scalable and flexible.
[3] DMBS: Database Management System, a system software for creating and managing databases.
[4] DBaaS: Database as a Service, a cloud computing service model that provides users with some form of access to a database without the need for setting up physical hardware, installing software or configuring for performance
[5] IDaaS: Identity as a Service, an authentication infrastructure that is built, hosted, and managed by a third-party service provider.
[6] PaaS: Platform as a Service, a cloud computing model that delivers applications over the Internet.
[7] MVC: Model View Controller, a software pattern.
[8] MVVM: Model View ViewModel, a software pattern
[9] POJO: Plain Old Java Object, it refers to a Java object that isn't bogged down by framework extensions.
[10] DTO: Data Transfer Object, an object that carries data between processes.
[11] JWT: JSON Web Token, a means of representing claims to be transferred between two parties.