Kotlin Practical Course Report

Movies App

Kirill Gringauz
Lisa Kropivianska
Carlos Nechwatal
Nugzari Mchedlishvili
Otar Kalandadze
Tsotne Mikadze
Temo Pkhakadze

Technical University of Munich, 2021

Introduction

Right from the beginning we had two goals in mind. Obviously, we wanted to pass the practical course, but we also wanted to create something for us and our friends to use outside of the course. Such an approach has defined our functional requirements and the way of working from day zero.

The idea of the app is quite simple: when a group of friends gathers for a movie night and can't agree on a movie everybody would want to watch, Swovie *(which is the name we've given to the project)* comes to the rescue. Our app tries to solve a real problem by providing a Tinder-like interface with movie cards that people swipe left and right to pick a movie.

The first thing we did was to talk through the use cases of the app and users' journey through it. Our main goal was to reduce the time users spend on the app: intuitive and quick setup, unobtrusive and quick way of picking the movie, quickly leading to watching the movie itself. The less time a user spends using an app, the more time a user has to enjoy the movie. Most of our later decisions were based on this principle.

Since time was of the essence, we wanted to setup internal work structure and processes for communication and task-tracking right from the beginning. With the task at hand, we needed to be as efficient as possible during this 10-day hackathon. It took us some time in the beginning of the course to get acquainted with each other and the project, break down the workload into smaller tasks and assign those based on each team member's knowledge and strong suits. Lisa took over the UI/UX part of the app as the only one of us with practical experience in design and later was the one to implement most of it; Carlos had created the screens and UI elements, navigation between them, animations, and has written code to enable interactions with the UI; Kirill was responsible for making the backend logic, including data exchange with The Movies DB's API and Firebase Realtime Database; our Georgian colleagues — Nugzari, Otar, Tsotne, and Temo — have helped with the UI implementation. Obviously, we still worked as a team and helped each other when problems arose.

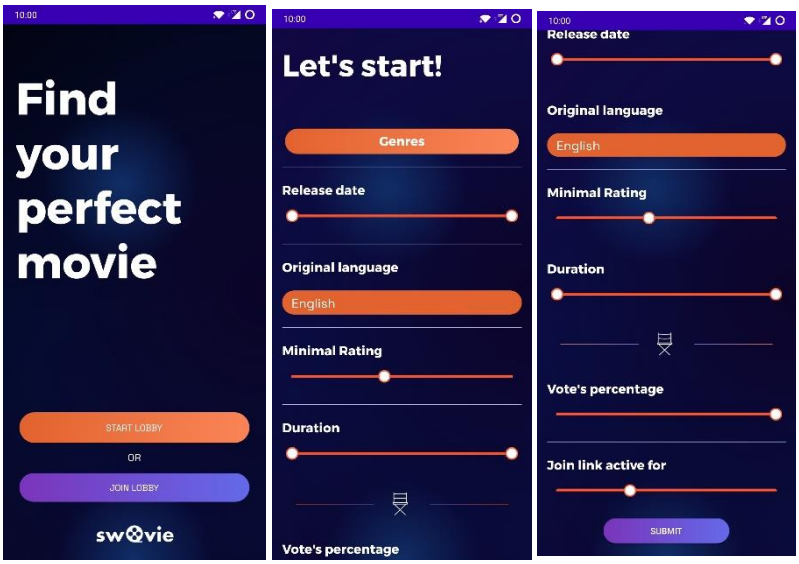
Use cases and Functionality

As mentioned above, our app solves a problem of choosing a movie that would satisfy all watching parties.

The basic use case, as we have imagined it, is that a group of people gathers in a room — or in the current COVID-19 reality in a voice chat — and decides to watch a movie. It is very seldom that everyone agrees on the same movie quickly, and the chances of quickly finding a common ground decrease in inverse proportion to the size of the group.

The layout of the app was designed in a way that enables the user to go through their journey intuitively, supporting our app's functionality through coherent and distinctive design.

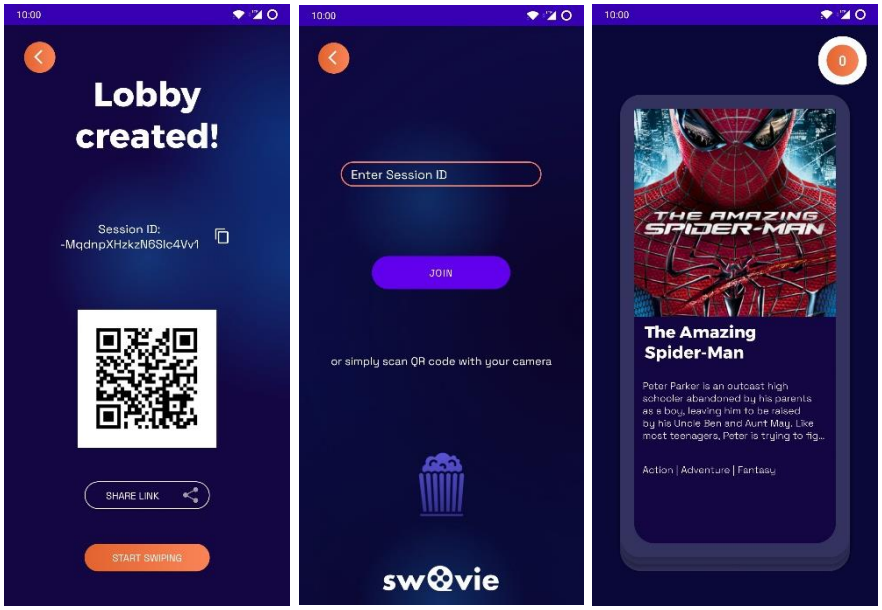
The first screen, which the user sees when opening an app, provides two options: ***starting*** or ***joining a session***.



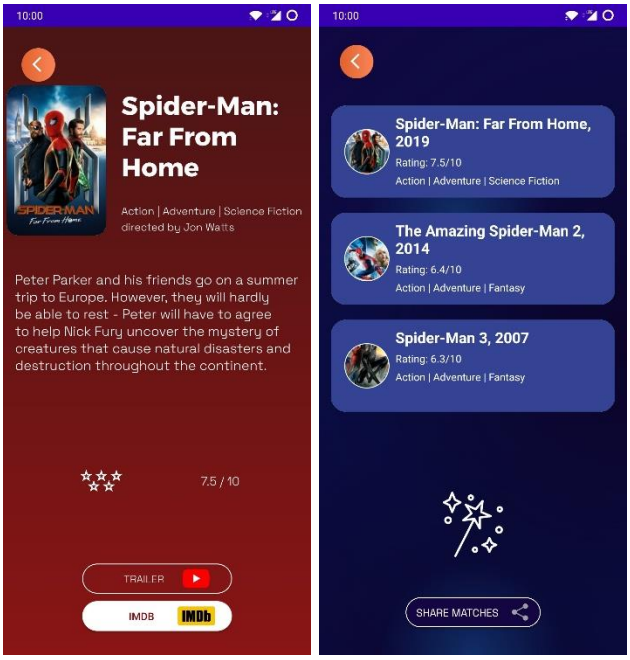
When choosing the first option, the user is taken to the settings screen, where the session gets tailored to the needs of the user. The settings menu allows to ***filter movies*** by several filters: release date, original language, minimal rating, and movies' duration.

Additionally, it enables another functionality – the user can ***adjust the session*** by deciding on a percentage of votes needed to add a movie to the list of matches, as well as adjusting the time when a join link is active, allowing anyone who's got a code or a link to join the session during that time. Moreover, after the user's preferences are submitted, the share screen appears. It includes two functionalities: a possibility to ***let others join the session*** either by QR code, or by a copyable link shown on the screen and, finally, a start button that ***invokes the start of the swiping session***.

In case the user decides to choose the "join session" option on the start screen, they will be firstly directed to a "join session" view with a text field to enter a session ID and a "join" button, which will directly take the user to the swiping session screen.



As mentioned before, the user gets to the start of swiping by starting a new session or by joining an existing one. The swiping view has a Tinder-like functionality — the main screen's part is occupied by a movie card, consisting of a poster and a brief description of the movie's name, plot, and genres. A user can ***decide whether he likes the film or not*** by swiping in the left/right direction. Sometimes, though, the information shown on this screen does not suffice to get a decent impression of the film. That's why additionally, when tapping on the card, the view changes to contain ***more detailed information*** about the movie, including full description, a rating, and links to the trailer and IMDB description. From this view the user can easily go back to the swiping session by tapping the back button.



Another important functionality we have thought through – the user can ***end the session at any time***, by clicking a "Back" button on their Android device. This will generate

a pop-up, where they can choose to exit the swiping session thus ending it or to go back to swiping.

Further, the round counter on the top right of the screen ***shows the real-time overall number of matches*** for the current session. Tapping the counter leads the user to the list of matches.

After the maximum number of matches is reached, the ***list view of all matches*** automatically appears on the screen. It shows us all the movies that have been chosen by a predetermined percentage of users with the corresponding name, rating, and genre. From this view, tapping any of the film containers leads the viewer to the details screen (similarly, as on the swiping screen, tapping on the back button on the details screen will take us back to the list of matches).

The overview of described functionality can also be seen in the OKRs table provided below in the report.

Implementation

Our app's implementation as well as users' interactions with it is divided in two parts: one is creating and setting up a session (or joining it, for that matter); the other one is finding the movie to watch by swiping movie cards left and right. The entire app is written *(as the course name suggests)* in Kotlin. We have done our best to write well-structured code with detailed comments, so it would be easier to understand.

General structure

App's classes are divided into packages to keep the code more structured:

- `api` package contains classes that interact with The Movie DB's API;
- `data` package holds objects with data used throughout the app's lifecycle;
- `database` package encapsulates classes that communicate directly with the Firebase Realtime Database;
- `models` package contains classes for serialization and deserialization of data from the API and the database as well as ViewModel classes;
- `start` and `swipe` packages respectively incorporate fragments and activities for the two phases of user interaction with the app;
- `ui` package holds data adapters and other helper classes used for displaying data and UI elements.

Navigation

To make the structure of our app more intuitive, we divided the navigation into two graphs. The first graph consists of the Start, Filter, Genre, Join and Share screens. The second contains only the Swipe, Details and Matches screens. That way we managed to separate the authentication and the session creation from the actual swiping functionality and could implement and test each of these parts independently. Another perk of creating 2 navigation graphs was the ability to implement both parts simultaneously.

Interactive UI Elements

Swipe Elements

The movie cards — swipe elements — are implemented as follows: in the `SwipeFragment` class, the list of not yet swiped movies is pulled from the `MoviesBatchViewModel`. The `swipeDeck` contains all the swipe cards. `DeckAdapter` class acts as its adapter. It loads a list of movies, implements basic adapter functions, and fills each card with basic information about the movie — title, poster, genres, and a short description. By setting an event callback on the swipe deck in the `SwipeFragment`, we were able to overwrite default swipe handling. In the case of a swipe, the current movie object is set to

swiped in the instance of `MoviesBatchViewModel` *(and therefore in the database)*, so that it's not shown again to a user who's already swiped it. If a card has been swiped right, the match count of the movie is increased as well. Furthermore, we check if the number of users who've swiped right exceeds the minimum number of right swipes required to count a movie as a match. When there are no more cards left on the stack, the next batch is loaded from the database.

Go-Back Alerts

Since users are sometimes restricted from going back to the previous screen, we needed "back button alerts" to prompt whether they really want to close the app. This functionality has been implemented in the `AlertDialogBuilder` class. It can not only create a dialog, but also override the `handleOnBackPressed()` method of the current activity, so the alert will be shown before the app is closed. However, in some cases this basic functionality was not enough, and we had to extend its functionality a bit instead of simply using the functions of the class. For example, at the matches fragment we wanted to make sure the alert is only shown if enough matches are found and the app has fulfilled its purpose, otherwise user is simply navigated back.

Data

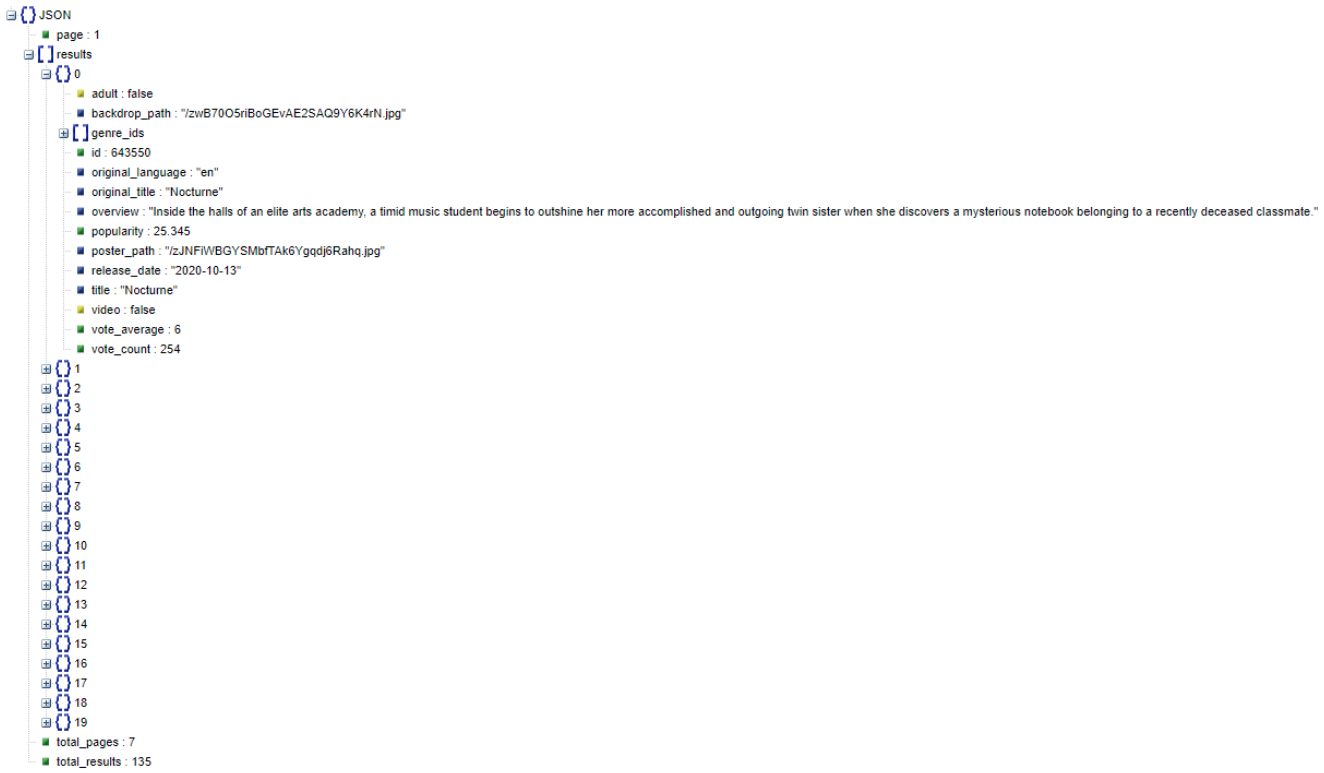
Kotlin has special classes called objects that only allow one instance of themselves to be initialized. We are using these classes to store data that is accessed from different screens. There are 3 objects in our project:

- `movieapp.data.GenresData` stores a cached list of genres. It is initialized together with the `movieapp.start.MainActivity` and offers methods to search for genres by their name or API ID. Genres are used for filtering movies and translating genre IDs from API's responses into their names.
- `movieapp.data.LanguagesData` is very similar to the `GenresData`. The only difference is, instead of loading its data from the API, this object has a static string in JSON format with a list of language names with their codes in ISO 639-1 format. These are used for filtering the movies.
- `movieapp.data.SessionData` contains all the necessary information about the active session: session ID; unique device ID; a flag indicating if a user is the session's host; a timestamp of the start of the session; movie filter; session options; a list of active users in the session; index of the movies batch currently shown to the user; and a list with database IDs of all loaded movie batches.

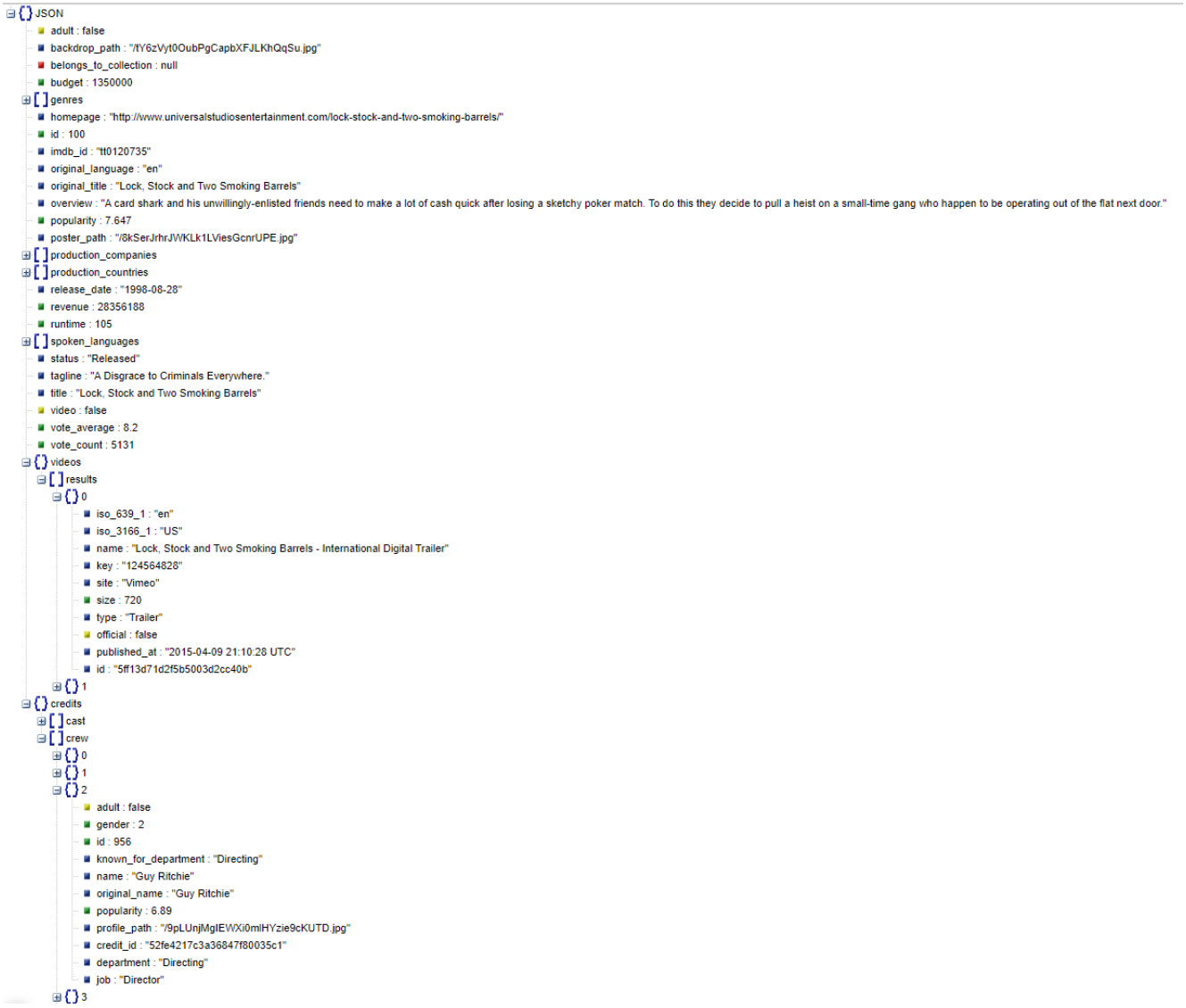
The Movie DB's API

Obviously, we didn't collect the information about the movies ourselves. Instead, we have decided to use an open API provided by The Movie DB. We have chosen to use this API for its well-written documentation, straightforward request structure, fast responses, and detailed information about more than 66 million movies. To communicate with the API endpoint, we are using a type-safe HTTP client called Retrofit. Communication with the API consists of 2 classes: `movieapp.api.Api.kt` is an interface that defines URL paths and parameters as well as response types for deserialization (their structure can be found in `movieapp.models.ApiResponses.kt`) and `movieapp.api.MoviesRepository.kt` is responsible for handling the asynchronous responses. There are 3 important methods for working with the API:

- `getGenres()` loads a list of all the genres used by TMDB in order to be able to later convert API's internal genre IDs to their names. This method is called once on the app's launch to load and cache the latest list of genres and their ID. These genres are also later dynamically loaded onto the Genres screen in the filter.
- `getMovies()` takes filtering attributes and response's page's number as parameters. This function is used to load movie suggestions by a defined filter from the API. The API's response contains a list of pages with 20 filtered movies each. Each page becomes a batch of movies that will be shown to the user; the `page` parameter is used to iterate through them. Loaded movies are then saved to the database to be available to session's users. This method is called once the host creates the session to load the first batch of movies, and then is called each time any user finishes swiping the last loaded batch of movies.



- `getMovieDetails(api_id, append_to_results, fragment)` loads more detailed information about a movie defined by `api_id` than the information available from the `getMovies()` function. `append_to_results` parameter defines a list of extra attributes like movie credits or videos associated with the movie. This method is called once a user presses on a movie or a match card and a new instance of `DetailsFragment` is created. This instance is passed to the function, so once the data has been asynchronously loaded it would be displayed on the fragment.



Firebase Realtime Database

The database is an essential piece of the app's architecture to coordinate sessions. All the information is stored in a `sessions` node, that contains a list of session's data.

A new session node is created at the moment session's host clicks "Submit" on the Filter screen after having configured the session's options and movies' filter. The node contains:

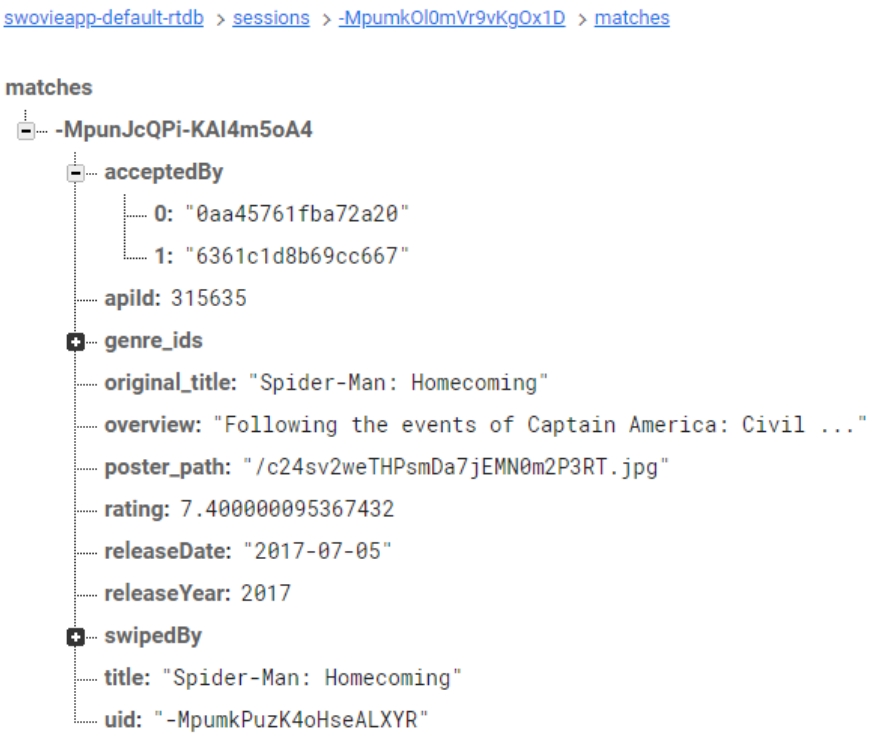
- session's status;
- a list of loaded movie batches' database IDs;
- movie filter configuration;
- a movies node that contains the information about each movie in each loaded batch;
- session's options configuration;
- a timestamp of start of the session;
- a list of session's active users.

There are 3 classes responsible for communicating with the Firebase Realtime Database from within the app. The main one is `movieapp.database.Database.kt`. It contains functions to:

- create a session called by the app of session's host after session's configuration was submitted;
- join a session which is called from the `JoinFragment` when a new user tries to join a session with a session ID, join link or a QR code. It validates user's input and either navigates user to the `SwipeFragment` if join was successful, or displays an error message in the `JoinFragment`, describing what went wrong;
- save a newly loaded batch of movies to the database;
- clear matches, that is called each time a new user joins the session, so only the movies that got an appropriate number of votes become matches;
- load session's data into the `SessionData` object and add `onDataChange` listeners to keep data in sync;

- leave a session to remove device's ID from the list of active users, so even if a user leaves the session matches would still be evaluated correctly;
- get references that are used in LiveData classes.

There are also 2 classes that extend `LiveData` class — `movieapp.database.MatchesLiveData` and `movieapp.database.MoviesBatchLiveData`. These classes load matches and movie batches respectively to show them to the user and handle data changes for new matches and swiped movie cards.



OKRs

Mission: Create an app to help a group of people to decide which movie to watch together

		Priority	Assignee	Status
Objective:	Allow users to create a session	**P1**		**75%**
	Give different options to share session with others	P1	Carlos	100%
	~~Allow the host to stop a session~~	~~P1~~	~~Kirill~~	~~0%~~
	Allow to select filtering options	P2	Carlos	100%
	Enable setting session options	P3	Carlos	100%
Objective:	Allow users to join a session	**P1**		**100%**
	Enter per session ID	P1	Kirill	100%
	Allow leaving the session	P1	Kirill	100%
	Enter per join link	P2	Carlos	100%
	Enter per scanning QR code	P3	Carlos	100%
Objective:	Create app's UI	**P2**		**64%**
	Draw layouts for each screen	P1	Lisa	100%
	Create app icon	P2	Lisa	100%
	Add swiping animations	P2	Carlos	100%
	Add animation for new matches	P2	Carlos	0%
	Add transaction animations	P4	Kirill	20%
Objective:	Create movie cards design	**P1**		**100%**
	Draw layout of a card	P2	Lisa	100%
	Add information about the movie to a card	P1	Kirill	100%
Objective:	Apply configured movie filter to loading movies	**P2**		**84%**
	Filter by genres	P1	Kirill	100%
	Filter by rating	P1	Kirill	100%
	Filter by release year	P1	Kirill	100%
	Filter by duration	P2	Kirill	100%
	Filter by language	P3	Kirill	100%
	Add advanced search	P4	Kirill	5%
Objective:	Display detailed information about a movie	**P3**		**100%**
	Display detailed information while swiping	P1	Nugzari	100%
	Display detailed information for matches	P1	Nugzari	100%
Objective:	Show same movies to all users in the session	**P1**		**33%**
	Load filtered movies	P1	Kirill	100%

	Show movies in different order	P2	Carlos	0%
	Reload movies if filter was changed during swiping	P3	Kirill	0%
Objective:	Add social engagement	**P4**		**100%**
	Share a download link to the app with friends	P1	Oto	100%
	Add a button to share matches	P1	Oto	100%
Objective:	Implement the design	**P3**		**88%**
	"Start" screen	P1	Temo	100%
	"Filter" screen	P2	Carlos	65%
	"Genres" screen	P3	Kirill	100%
	"Share" screen	P2	Temo	100%
	"Join" screen	P2	Oto	100%
	"Swiping" screen	P1	Nugzari	70%
	"Matches" screen	P1	Tsotne	90%
	"Details" screen	P2	Tsotne	80%
Objective:	Do all the copywriting	**P3**		**75%**
	Write welcome screen	P2	Lisa	100%
	Write explanations for how to use the app	P1	Kirill	0%
	Write template message for sharing matches	P3	Lisa	100%
	Write template message for sharing a join link	P1	Carlos	100%
Objective:	Test the application	**P3**		**90%**
	Test "normal" usage	P1	Lisa	100%
	Test "edge" cases	P1	Kirill	100%
	Test losing connection	P2	Carlos	70%

Possible Extensions

It is not possible to come up with a plan, create a perfect app, and test it thoroughly within 10 days, so although we have done our best, there is still a lot of room for improvement.

The main issue that needs to be addressed is Swovie's sensitivity to the quality of the Internet connection. Network interruptions and low bandwidth may result in dropped API requests, lost API responses, or delayed data exchange with the database. These factors lead to errors and delays in app's responsiveness, and impair users' experience with the app. A possible solution to this problem has several aspects to it.

Firstly, caching information about the movies would allow to avoid network data exchange at all for the most popular movies that are shown more often. This method however requires a close analysis to properly balance the ratio of cached information's usability in terms of the frequency of its usage to the phone's internal storage usage for storing it. In its own turn, this can be achieved by only caching movie's details for movies with rating above a certain threshold since they are shown more often; or deleting the information if it has not been accessed for a certain amount of time.

Secondly, a network connection monitoring thread would help. It would measure the network's speed in the background and handle connectivity losses and lags with a wide range of measures: from resending requests and notifying the user of Internet connection problems to blocking app's usage completely until the connection is reestablished in severe cases. Monitoring the Internet connection's speed would generally allow to mitigate arising issues that an unstable connection could cause and to provide a better user experience.

Finally, a peer-to-peer architecture would also improve app's stability and reliability. The idea here would be to allow inter-device communication using Bluetooth or local Wi-Fi network, since in most cases Swovie's users would be sitting in the same room. Devices lacking stable and quick Internet connection would be able to use other devices in the same session as their gateways into the open Internet, or simply request data that better-equipped devices have already loaded and cached. This method, as complicated as it is, would not only allow to completely avoid any delays or problems caused by the network for as long as at least one device in the session has a stable connection, but also allow for all-but-one-device offline sessions, were all app instances use the single online phone as their hub for loading the movies information from outside and then only exchange data locally to decide on a movie.

However, stability improvements are not the only direction in which the app can be refined. We have also discussed allowing users to use an advanced movies filter with extra parameters, i.e. production companies, directors, watch providers, actors, and keywords. This change would not require any major changes to the app's architecture, most changes would have to be done to the filter UI.

Another nice feature would be to allow users to swipe only on movies that are currently shown in cinemas in their country.

Finally, we did not have the time to implement layout design completely. We'd also like to polish Swovie's looks and make it more intuitive and appealing to its users. Key UI details that need further improvement are the sliders on the Filter screen. We would need to adjust their scale and units of measurement, i.e. set the movie's duration in "HH:MM" format rather than in minutes. We would have also liked to test the app's behavior and appearance on different screen sizes and with different Android versions and builds.