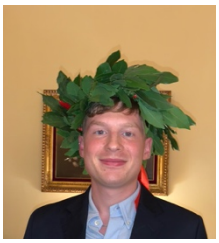
Design Document

Design and implementation of mobile applications

Professor Luciano Baresi

Academic Year: 2022/2023

Team members



Riccardo Casciotti

riccardo.casciotti@mail.polimi.it

Master degree in computer science

Ivan Crusco

ivan.crusco@mail.polimi.it

Master degree in computer science

Contents

1. Introduction

1.1 Purpose

This design document has the purpose of illustrating how the application "SpotiQuiz" we developed for the course "Design and Implementation of mobile application" held at Politecnico di Milano has been built, which are the goals we wanted to achieve with it and how we implemented them.

1.2 Acronyms definitions and abbreviations

DB = database, the place in which we store the data we want to keep for every session.

API = Application Programming Interface, a way in which is possible to communicate with another systems.

OS = Operating System, the main software that manages the device hardware and software resources, it provides services to other programs and applications.

UX = User eXperience, the way in which is defined how a user interacts with a product, system or application.

UI = User Interface, the components through which the interaction between user and application happens.

App = application.

SpotiQuiz = the name of the app, we will use It to refer to the final product.

Bio = biography.

ID = identifier of an entity.

User = the person that uses the application.

Uid = it's the ID of the user, which distinguishes him/her from the other users.

Nationality = the nationality of the user given by Spotify API. We use it to provide a local and a global ranking.

BestScore = the best score that a user has made for a quiz. It is used as a personal information but also to create a ranking.

Score = the score of the current quiz going on.

NumOfQuiz = a parameter to remember how many quizzes the user has played since he started using the application.

CorrectAnswers = a parameter to check how many correct answers a user has given during the quizzes.

WrongAnswers = a parameter to check how many wrong answers a user has given during the quizzes.

Level = the level of a user, to keep a trace of his/her progression while using the application and to display in the rankings.

Experience = the experience of a user, a parameter that grows every time he/she plays a quiz. After it grows after a certain threshold, a new level is reached.

RefreshToken = a string related to a user which is necessary to do request related to him/her to the Spotify API.

AccessToken = a string related to a user which expires after 1 hour.

Quiz Screens = with the term quiz screen, we refer to the interface we use to let the user play the different type of quizzes that we created.

Screen A = quiz screen related to the first game mode.

Screen B = quiz screen related to the second game mode.

Screen C = quiz screen related to the third game mode.

Screen D = quiz screen related to the fourth game mode.

Random mode screens = the equivalent of screens A, B, C and D, but for the mixed game mode, with a slightly different layout that comprehend the question.

1.3 Document structure

- Chapter 1: introduction to the document and its purpose
- Chapter 2: overview of the application and of its functionalities

- Chapter 3: architectural and technical overview
- Chapter 4: images of the various screen and the flow of the various interactions
- Chapter 5: test campaign and results of it
- Chapter 6: references

2. Application functionalities and features

The primary goal of the app is to let its users play quizzes based on their tastes on Spotify.

Starting the application, the user will have to login into his/her Spotify account through a web view.

If the user has not an account, we suggest him/her to create one, since It is strictly necessary to use the app properly.

After accepting the terms and conditions from Spotify, he/she will be redirected to the home page of the application.

On his/her home page, the user will find information as his/her level, number of quizzes done until that moment and profile picture taken from Spotify.

If the user has not a picture on Spotify, we provide a placeholder as a picture.

It will be possible to click on the profile picture to see the user page, in which there are more detailed information related to the experience of the user, the percentage of correct and wrong answers given and the best score done.

From the user page it will be possible to return to the home page thanks to a return button.

From the home page, the user can return to the login page through a logout button, or decide to go to the ranking page, to the event page or to the quiz page thanks to dedicated buttons.

The ranking page will display a ranking based on the overall best scores of the users, both locally and globally.

From the ranking page the user can return to the home page.

The event page will display music events that will happen in the zone where the user lives in the following days.

To have this functionality, the user will have to give the authorization to the localization of the device.

From the event page the user can return to the home page.

In the quiz page, the user will find the different types of quiz available and a pool of artists that he/she could like based on his/her preferences on Spotify.

From the quiz page it will be possible to return to the home page thanks to a specific button.

The different quizzes will be presented with different buttons and a different name.

There will be 5 types of quiz: artists, songs, album, "Sarabanda!" and random.

Clicking on the button related to the quiz, the user will go in an information page in which a more detailed description of the game mode will be available.

From the information page, the user can decide if he/she wants to play clicking on a play button or come back to the quiz page with a return button.

Once playing the quiz, the user will be able to give one out of 4 answers clicking on the corresponding button.

Once the answer is given the user will be redirected to the result screen in which he/she will see the actual score of the quiz, and if the answer he/she gave was correct or not.

From the result screen, the user can decide to return to the info page or if to continue the quiz with the next questions through 2 different buttons.

The quiz will be done in an endless way.

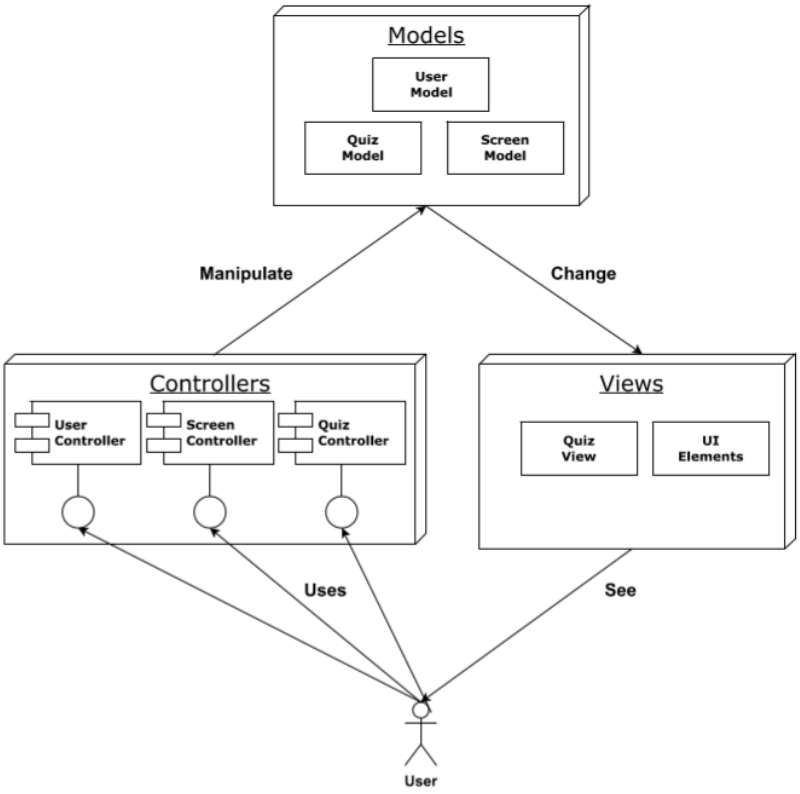
Here a list of feature that the application will have:

- A Firebase database to store consistent data related to the user.
- Adaptanility with regards to the screen orientation
- Different layouts depending on the device used.
- External API to get user's preferences on Spotify
- External API to get events in the user's area
- 6 different localizations: italian, english, spanish, french, german, russian.
- Geolocation of the user to get the device current position.

3. Architectural design

3.1 Baseline

We used MVC as design pattern reference for the architecture, with models for users and screens, views for every page of the applications showing relevant information and interactions, and controllers for the population of the screens, the creation and update of the users and the creation of the different quizzes.



3.2 Technologies

The functionalities of SpotiQuiz were possible thanks to the extensive use of technologies such as REST API services which allowed to retrieve data from external servers and manipulate them to create a seamless user experience, and on board hardware interfaces peculiar to the devices itself like the GPS chip to use the user's whereabouts for a personalized experience. These technologies are fully exploited by

integrating libraries which allowed solutions to interact with their functionalities. More in depth the libraries allowed http interactions, test functionalities and web navigations. Finally, some functionalities of the project are based on the BLOC (Business Logic Component) design pattern to deal with delicate and important events such as login and authentication of a user.

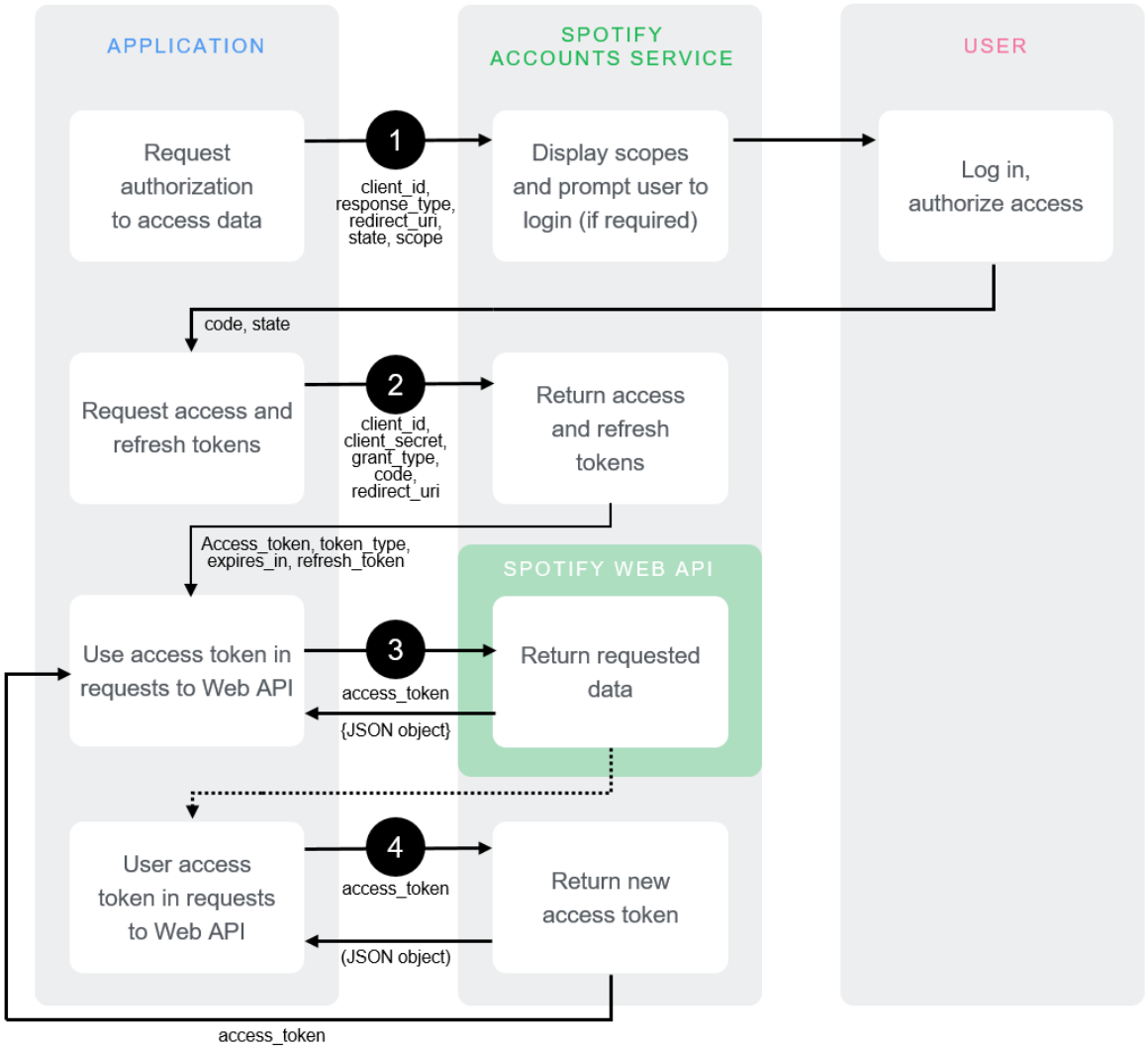
3.3.1 REST API

A REST (Representational State Transfer) API is a set of rules and conventions for building web services that follow the principles of the REST architectural style. REST is an architectural style that emphasizes scalability, simplicity, and statelessness in distributed systems. REST APIs are widely used to enable communication and data exchange between different systems over the internet.

3.3.1.1 Spotify REST API

The Spotify REST API is used extensively throughout the whole application to retrieve data based on the user's Spotify profile and on the user's preferences in terms of music genres, artists and albums.

The Spotify API uses a very secure and complex mechanism of authentication for both the application and the user who wants to use it. In particular, it is an authentication schema which is based on tokens of two main kinds: the first kind is an access token which is useful to authenticate the API calls and the second kind is a refresh token which is used to request a new access token after expiration of the previous one. The following diagram shows the complete authentication flow:



Spotify offers different endpoints in the API service which allow to retrieve specific data also based on the authorization scopes given by the user at login time. In particular SpotiQuiz uses the following ones:

- get user's information
- get related-artists
- get artist's albums
- get user's followed artists
- get artist form id
- get artist's top tracks

The first endpoint is used to retrieve the main information about the logged in user such as the profile image and the username. The other endpoints are used to retrieve data about the music's taste of the user and other data such as the followed artists and the related albums of the artist along with a list of the top tracks on the platform.

All of the raw data is then processed and formatted to be used on the creation of the quizzes. All the api calls are implemented in a file called “api_calls.dart”, following an example of one of the functions which uses the endpoints:

```dart
Future<List<model.Artist>> get_followed_artists() async {
  accessToken =  getAccessToken();

  final artistsInfo = await http.get(
      Uri.parse("https://api.spotify.com/v1/me/following?type=artist"),
      headers: {
        "Authorization": 'Authorization: Bearer $accessToken',
        "content-type": "application/x-www-form-urlencoded"
      });

  final artistsJson = json.decode(artistsInfo.body);

  List<model.Artist> artists = [];
  for (var i = 0; i < List.from(artistsJson["artists"]["items"]).length; i++) {
    var curr_artist = List.from(artistsJson["artists"]["items"])[i];

    final res = create_artist(curr_artist);
    artists.add(res);
  }

  return artists;
}
```

As we can see the access is retrieved in a secure manner using the “getAccessToken” function and then, though the external library “http” a get is performed to get the data about the followed artists of the specific user. The raw object is then processed and formatted into a Dart model called “artist.dart” (placed inside the “models” folder).

3.3.1.2 Firebase API

Firebase is a comprehensive suite of cloud-based tools and services provided by Google for developing and managing web and mobile applications. It offers a range of features, including real-time database, hosting, authentication, cloud storage, messaging, and more. Firebase also provides a powerful API that developers can use to integrate Firebase services into their applications.

Firebase API allows developers to interact with Firebase services programmatically, enabling them to read and write data, authenticate users, send push notifications, and

perform other operations. The Firebase API is available for various platforms, including JavaScript, Android, iOS, and several server-side environments.

The SpotiQuiz application, out of the many services Firebase offers, uses the Firebase Cloud Firestore API which is a flexible, scalable NoSQL database and it enables to store, retrieve, and query data using documents and collections. In particular, this services is extensively used in the application to store data about the users who logged in the platform using their Spotify account and so to allow to store information about their quizzes and the points earned with the quizzes necessary to create the standings of all the users who play and interact with SpotiQuiz.

3.3.1.3 Geogoding with OpenCageData

Geocoding is the process of converting addresses or place names into geographic coordinates (latitude and longitude). It allows you to transform textual location information into usable data for mapping and location-based applications. This services is particularly useful when processing the raw data in coordinates from the GPS chip of the actual device. In fact, a problem faced during development was on how to convert the coordinates into an actual place or city and this API does just that. The following picture contains the actual API call:

```
Response position_data = await http
    .get(Uri.parse("https://api.opencagedata.com/geocode/v1/json?q=$position&key=$geoCodingKey"));
```

Where the geoCodingKey is the token used to access the service and the position variable contains the coordinates to be converted. The value returned is a JSON containing the information about the place specified such as name of the city the county and continent.

In addition, this API service has made itself very useful to use another service which allows SpotiQuiz to fetch the events and concerts in the city where the user is situated.

3.3.1.4 Concerts – Artists Events Tracker from RapidApi

The Concerts - Artists Events Tracker API it is a service used by the platform to retrieve information about social events in the area of the user's whereabouts. It is provided by RapidAPI which is an online platform and marketplace that allows developers to discover, test, and integrate a wide range of APIs (Application Programming Interfaces) into their applications. It acts as a central hub where developers can find APIs from different providers, manage their API keys, and access API documentation and SDKs (Software Development Kits) for various programming languages.

The application uses this API services in conjunction with the GPS location of the device that the user is using SpotiQuiz from. The code for the API call is the following:

```
events_api_called = true;
final eventsInfo = await http.get(
    Uri.parse(
        "https://concerts-artists-events-tracker.p.rapidapi.com/location?name=$city&minDate=$minDate&maxDate=$maxDate&page=1"),
    headers: {'X-RapidAPI-Key': eventsKey, 'X-RapidAPI-Host': eventsHost});

final eventsJson = json.decode(eventsInfo.body);
//final eventsJson = json.decode('{"data":[{"@context":"http://schema.org","@type":"MusicEvent","description":"Daniele Silvestri
if(eventsJson["data"] != null)
{
for (var i = 0;
    i <
        (List.from(eventsJson["data"]).length < 20
            ? List.from(eventsJson["data"]).length
            : 20);
    i++) {
  print(List.from(eventsJson["data"])[i]);      You, 4 weeks ago • All quiz modes complete + events api logic + loca…
  var curr_event = List.from(eventsJson["data"])[i];
  events.add(format_event(curr_event));
}
// print(eventsJson);
// print(events.toString());
}
events_call = events;
```

Where the "city" variable specifies the events' city and the "minDate" and "maxDate" specify the date interval for the events to be found.

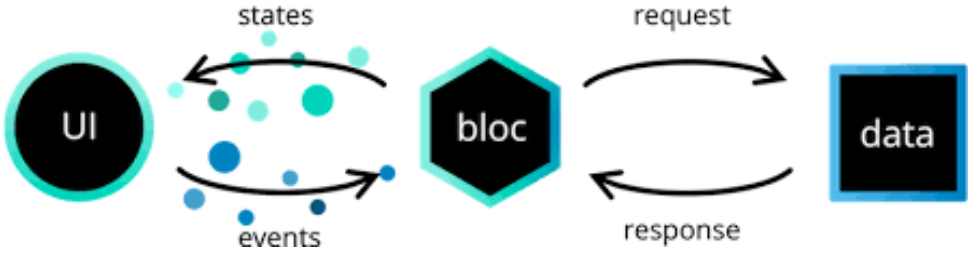
3.3.2 External libraries

External libraries played a fundamental role into the integration and seamless functioning of certain services offered by the SpotiQuiz platform. Following the explanation of some of the main libraries used in the project.

- **Flutter Intl**: this library was used to implement the language localization based on the language of the device which the application is running on.
- **Flutter Bloc**: to integrate the BLOC pattern into the project.
- **Flutter Integration Test**: used to produce the unit tests, widgets tests and integration tests.
- **Flutter Webview**: library which allows to open a web browser inside the application's window, in particular it has been used to go through the login process managed by the Spotify Web API.
- **Http**: library used extensively to make the API calls by allowing the application to use http requests and gather the raw response.

3.3.3 Flutter BLOC

Flutter BLoC (Business Logic Component) is a popular state management pattern and library for building Flutter applications. BLoC helps in separating business logic from the UI layer, making code more manageable, reusable, and testable. It follows a reactive programming approach and leverages streams to handle state changes.



This pattern has been largely used in the application to manage the authentication of a user and the login phase. In particular, whenever a user has not been identified yet the status inside the application is set to unauthenticated. Once the login process starts another BLOC model is used to manage the login process itself and be sure that everything goes smoothly. Thus, the login BLOC sets the status of the login to "in progress" and once the whole procedure is completed it sets its status to "completed". The implementation of such login process is due to the complexity of the interaction with the Spotify API, since the application needs to open a web browser and communicate with the Spotify servers it is appropriate that the whole process is adequately supervised and followed. Once the login phase is completed the authentication bloc sends an event which modifies the authentication status of the current user from unauthenticated to authenticated. At this point the application loads the homepage and the user can use all the functionalities available. Following the implementation of the authentication BLOC:

```dart
class AuthenticationBloc
    extends Bloc<AuthenticationEvent, AuthenticationState> {
  User user = User.empty;
  List<User> userByNation = List<User>.empty();
  List<User> userGlobal = List<User>.empty();

  AuthenticationBloc({
    required AuthenticationRepository authenticationRepository,
    required UserRepository userRepository,
  })  : _authenticationRepository = authenticationRepository,
        _userRepository = userRepository,
        super(const AuthenticationState.unknown()) {
    on<_AuthenticationStatusChanged>(_onAuthenticationStatusChanged);
    on<AuthenticationLogoutRequested>(_onAuthenticationLogoutRequested);
    _authenticationStatusSubscription = _authenticationRepository.status.listen(
      (status) => add(_AuthenticationStatusChanged(status)),
    );
  }

  final AuthenticationRepository _authenticationRepository;
  // ignore: unused_field
  final UserRepository _userRepository;
  late StreamSubscription<AuthenticationStatus>
      _authenticationStatusSubscription;

  @override
  Future<void> close() {
    _authenticationStatusSubscription.cancel();
    _authenticationRepository.dispose();
    return super.close();
  }
```

```dart
  Future<void> _onAuthenticationStatusChanged(
    _AuthenticationStatusChanged event,
    Emitter<AuthenticationState>
        emit, // emit new state of the current BLOC, different from
  ) async {
    switch (event.status) {
      case AuthenticationStatus.unauthenticated:
        return emit(const AuthenticationState.unauthenticated());
      case AuthenticationStatus.authenticated:
        return emit(
          user != User.empty
              ? AuthenticationState.authenticated(user)
              : const AuthenticationState.unauthenticated(),
        );
      case AuthenticationStatus.unknown:
        return emit(const AuthenticationState.unknown());
    }
  }

  void _onAuthenticationLogoutRequested(
    AuthenticationLogoutRequested event,
    Emitter<AuthenticationState> emit,
  ) {
    _authenticationRepository.logOut();
  }
}
```

4. Design

We wanted the look of SpotiQuiz to be minimalist and to recall the colours and feels of Spotify.

For this reason, we opted for an app with simple and meaningful buttons and the same palette made by green, white and black colours taken from the official Spotify documentation.

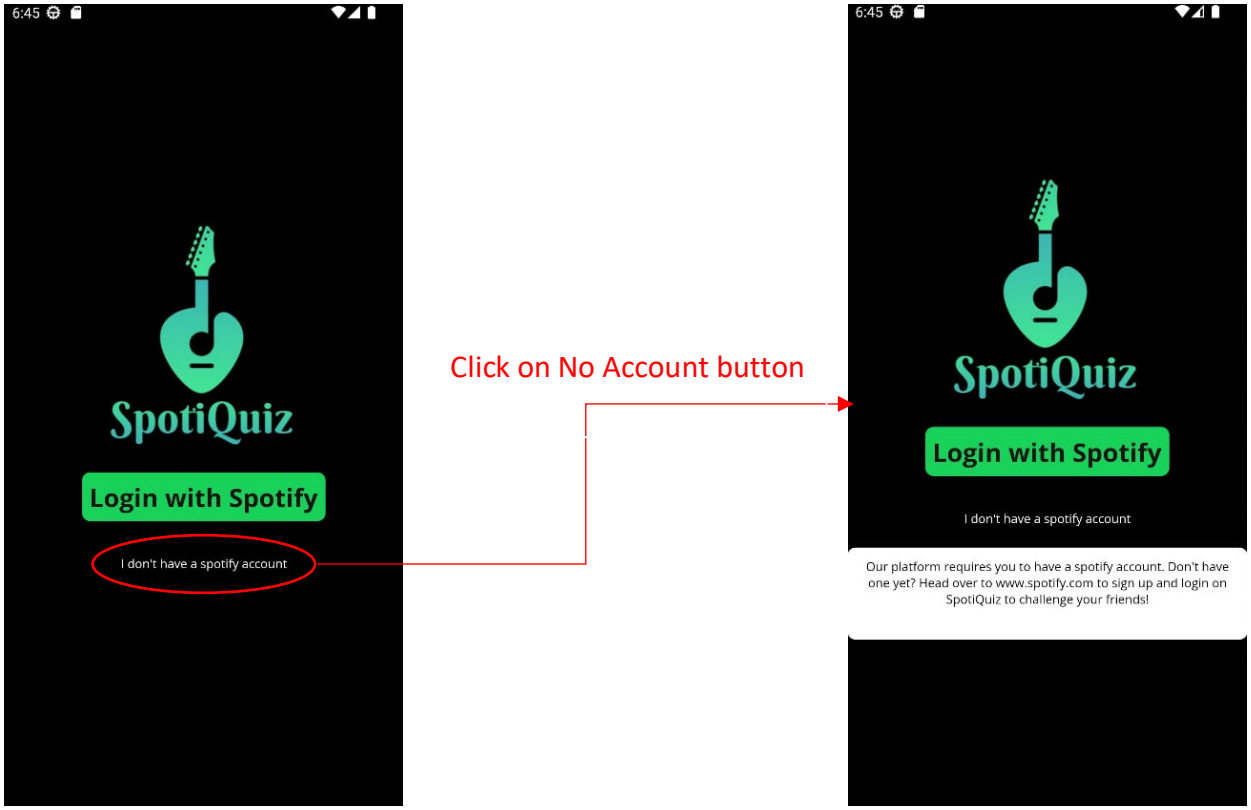
We developed two different layouts, one for smartphones and the other one for tablets, that we will present in the following with some screenshots of the app running.

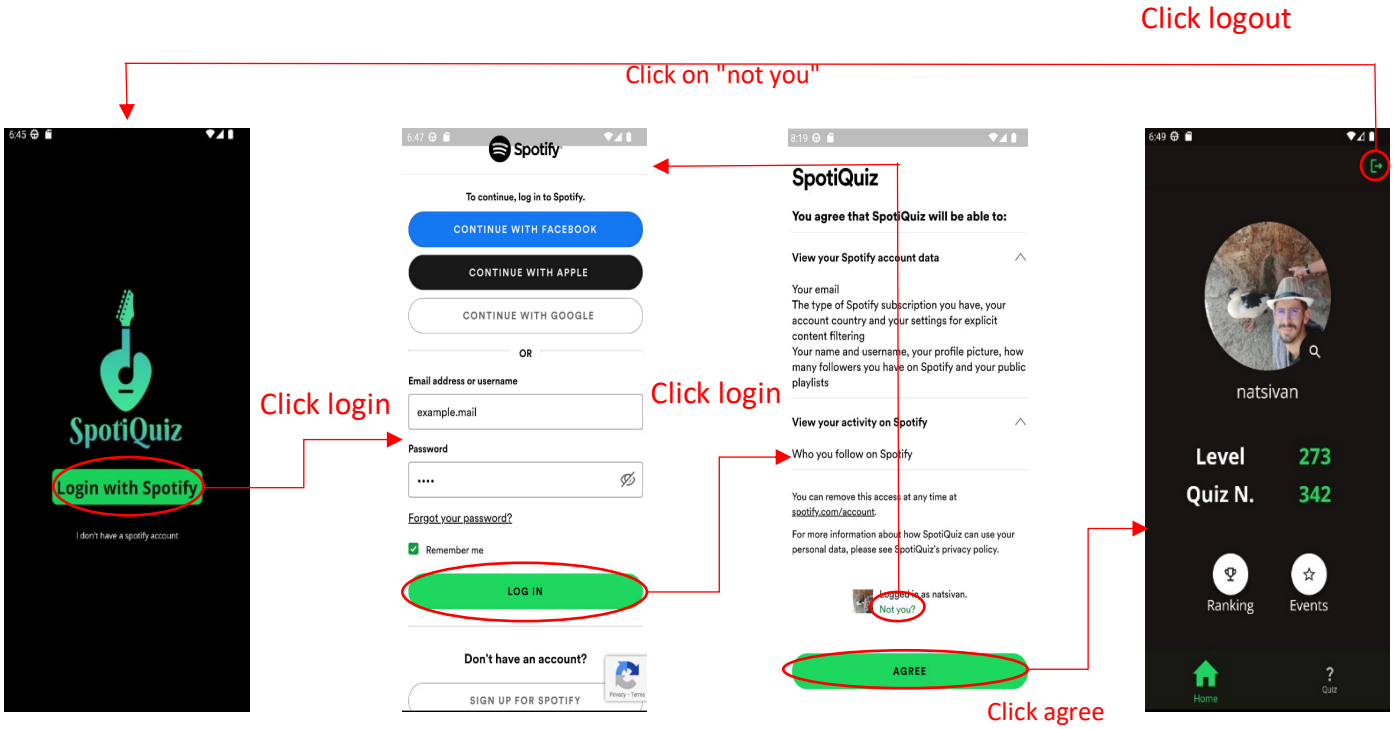
Having in mind the principles of responsiveness and adaptability, the application will continue to be ready to use when we change the orientation of the device, and for the most relevant screens as the quiz we also changed the layout to make it easier to use.

4.1 UX Diagrams (Flow of interactions between the elements)

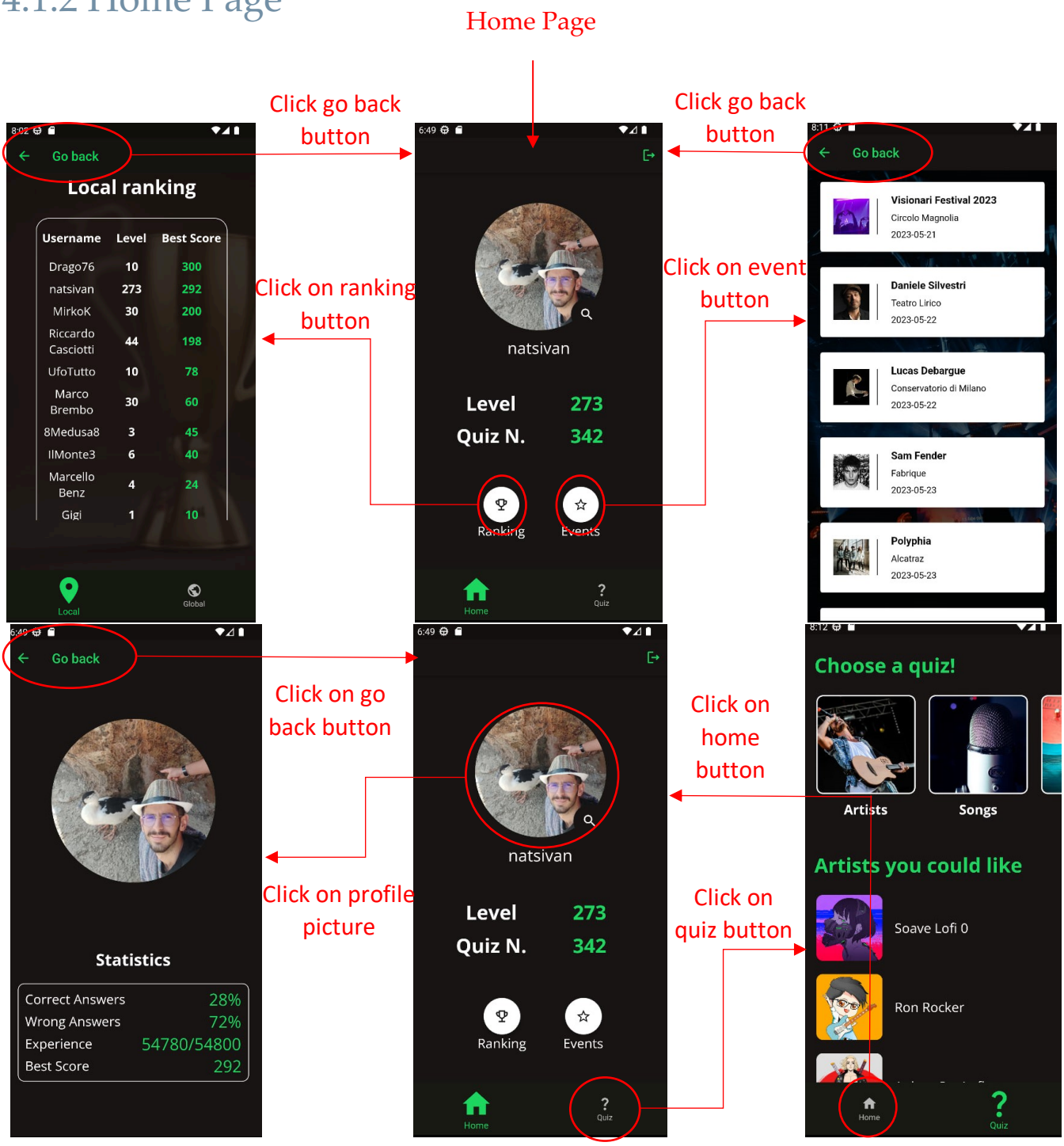
In this section we will show the various interactions that the user can do while in the application.

4.1.1 Login and authorization phase

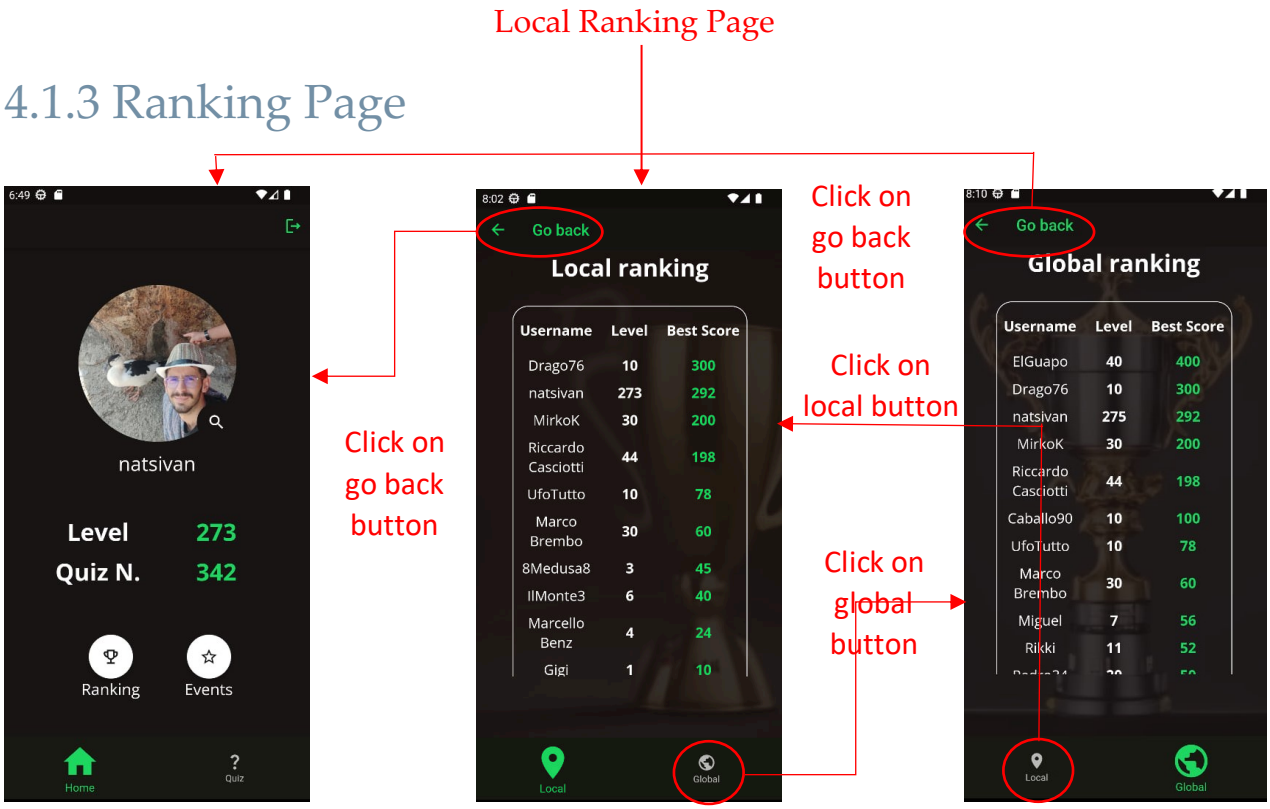




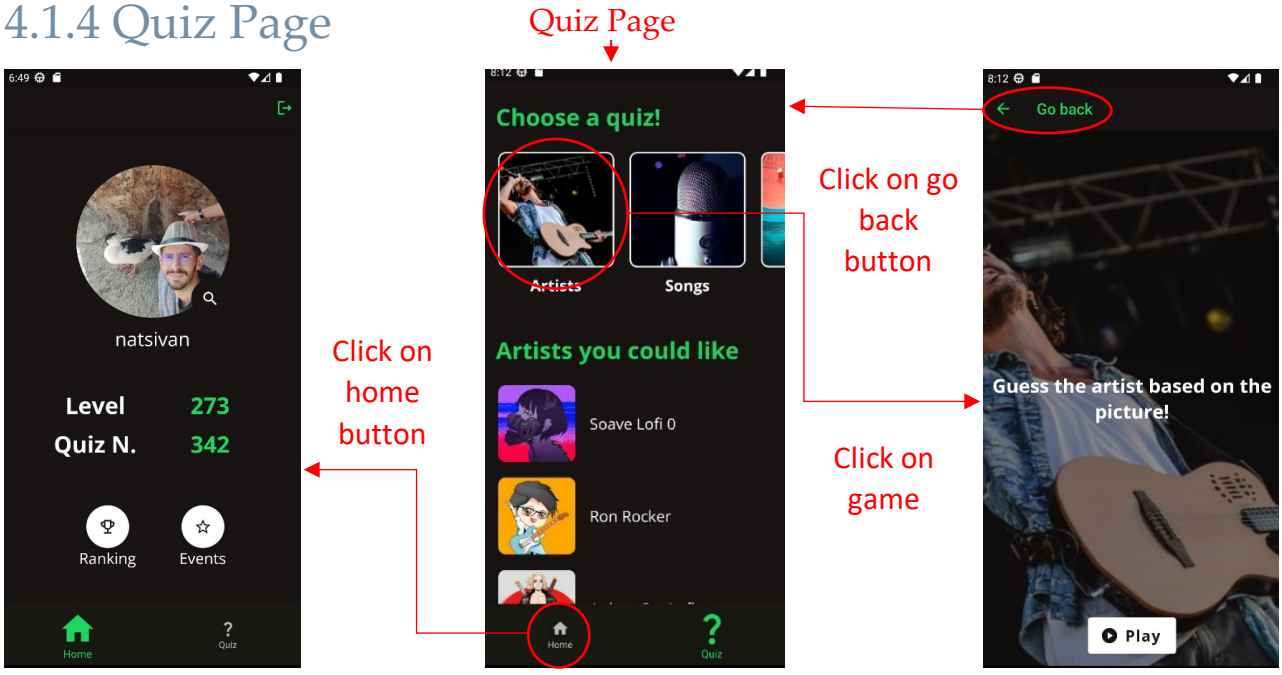
4.1.2 Home Page



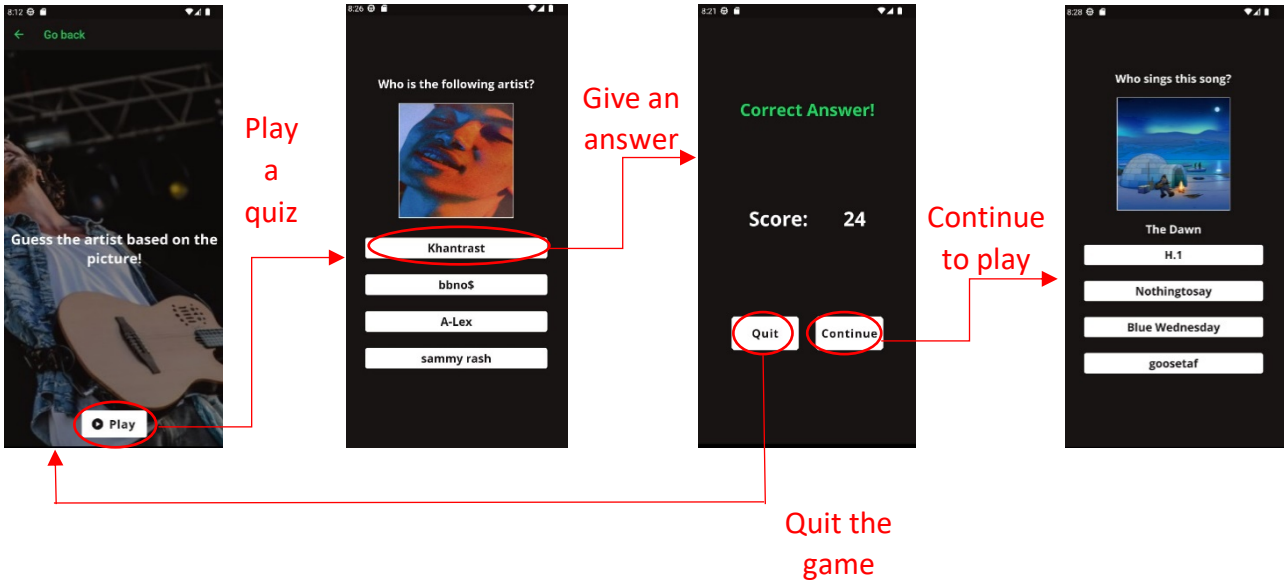
4.1.3 Ranking Page



4.1.4 Quiz Page



4.1.5 Quiz Logic



4.2 User Interfaces

In this section we will show the various screens of the application explaining their purpose.

Since we significantly changed the layout for the tablet version, we will put both the smartphone and tablet layouts.

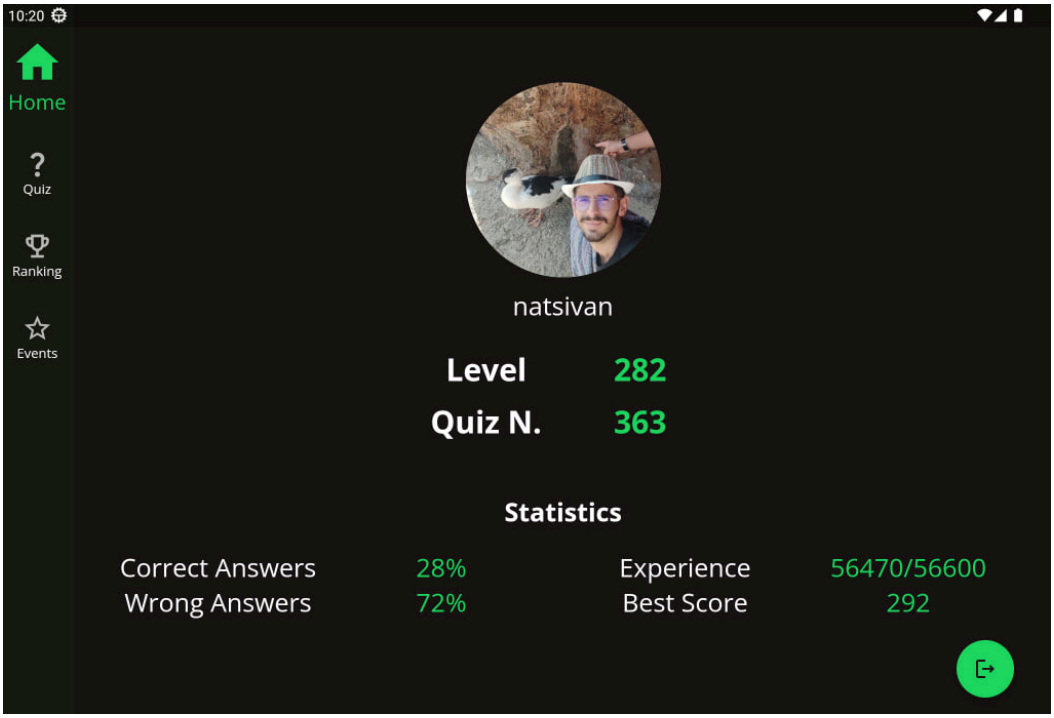
Furthermore, for some relevant screens we will show the difference in terms if layout when we change the orientation of the device.

4.2.1 Login Page

The application starts from the login page. From here, the user can press on the login button to enter the app or, if he/she doesn't have a spotify account, can presso on the botton under the login in which we explain how to create one

4.2.2 Home Page

Tablet



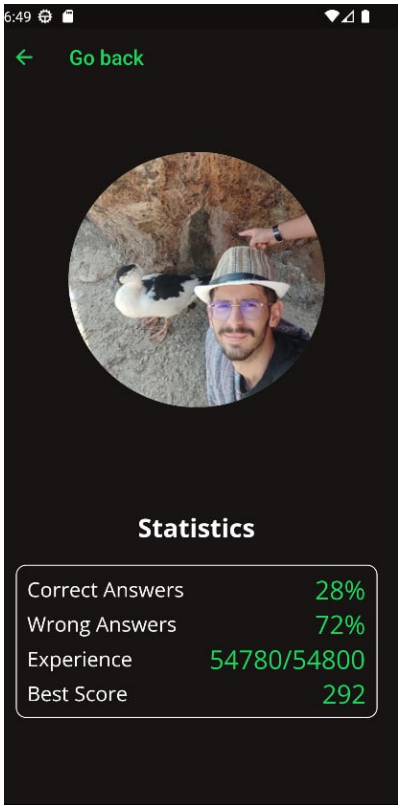
The first page after logging in.

Here the user can see basic information related to the account.

It is possible to logout thanks to the button on the screen, and to see more detailed information clicking on our profile picture.

From here, the user can also go to the quiz page, the event page and the ranking page.

4.2.3 User Page



Page with more details for the user, regarding his/her overall performance related to the quizzes played. This page is available only on the smartphone, since we decided to integrate it directly in the home page for the tablet version, in order to use the space available making it easier for the user to access these data.

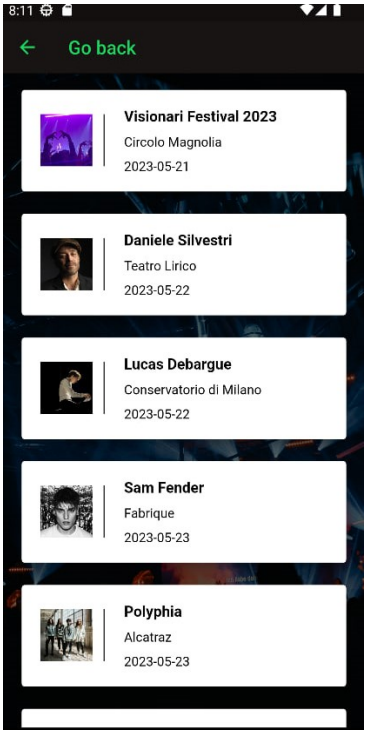
4.2.4 Ranking Page

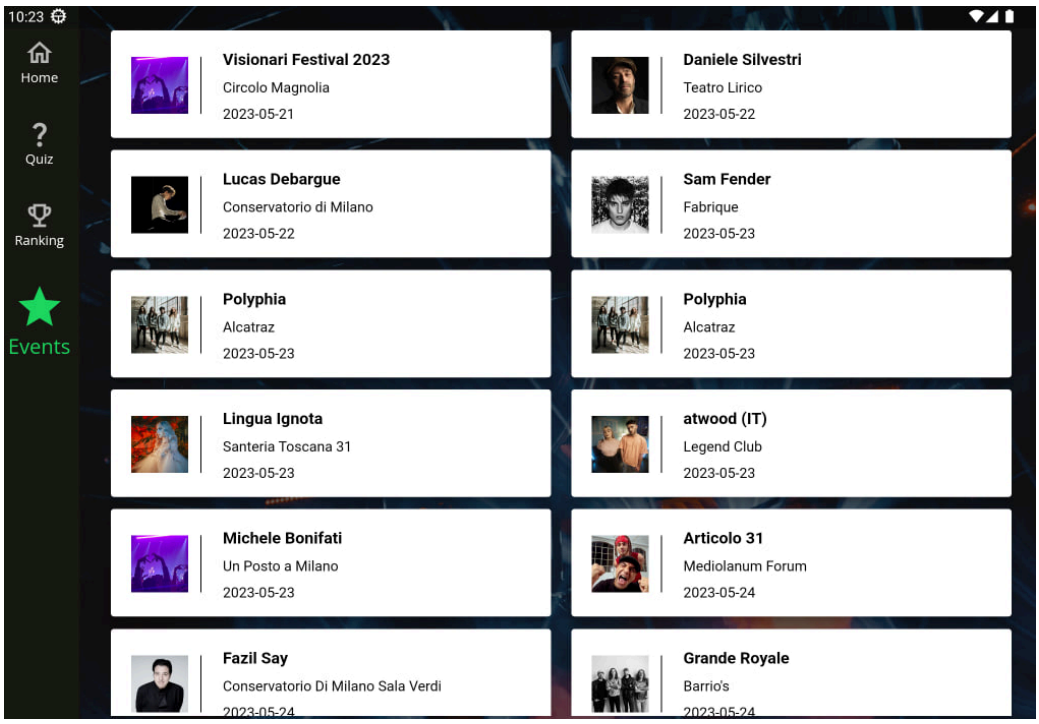
Tablet

The ranking page is where the user can see how well he/she is playing with respect to other players locally or around the world. In the tablet version, both the rankings are available in the same page, while in the smartphone version the user can easily switch between a ranking and the other thanks to the buttons at the bottom of the screen.

4.2.5 Event Page

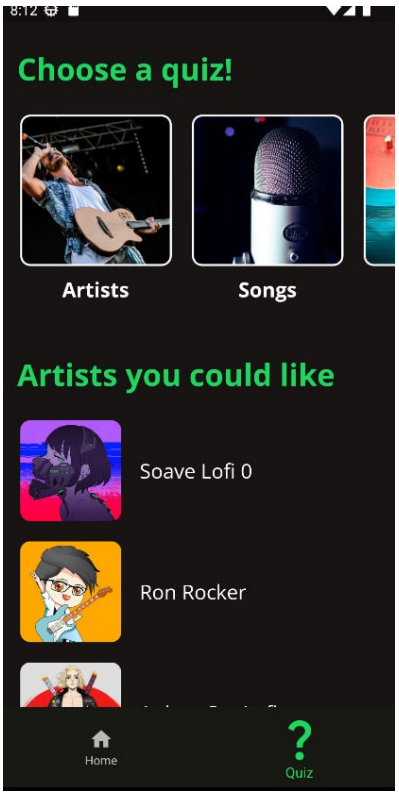
Tablet



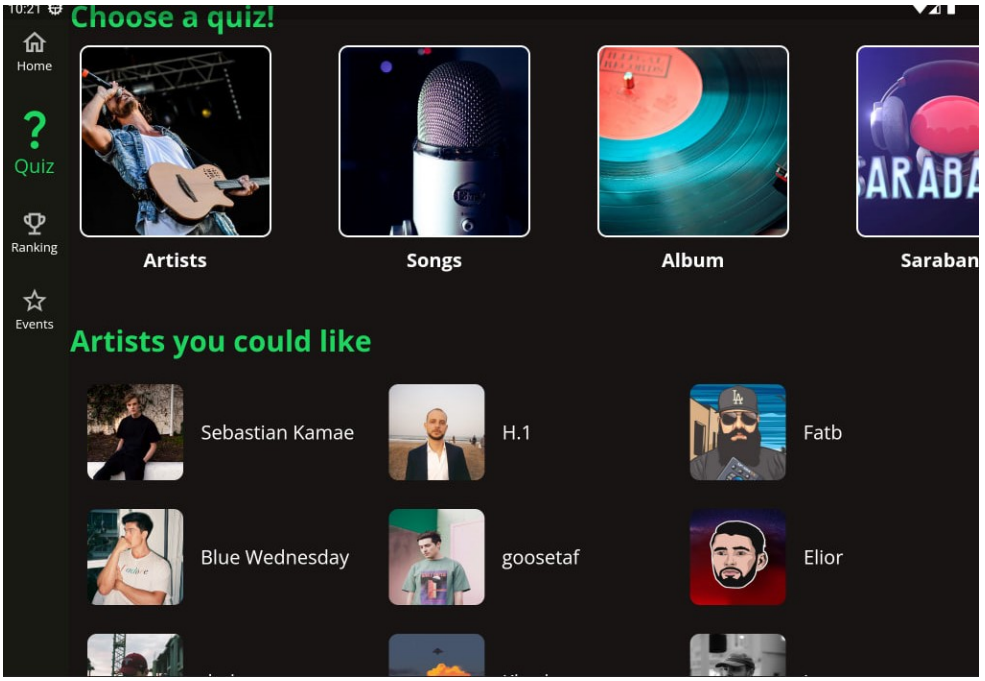


The event page is where the user can see which are the music events taking place near his/her zone in that period of time. Since SpotiQuiz is an app related to music, with this functionality passionate people can always stay up to date.

4.2.6 Quiz Page



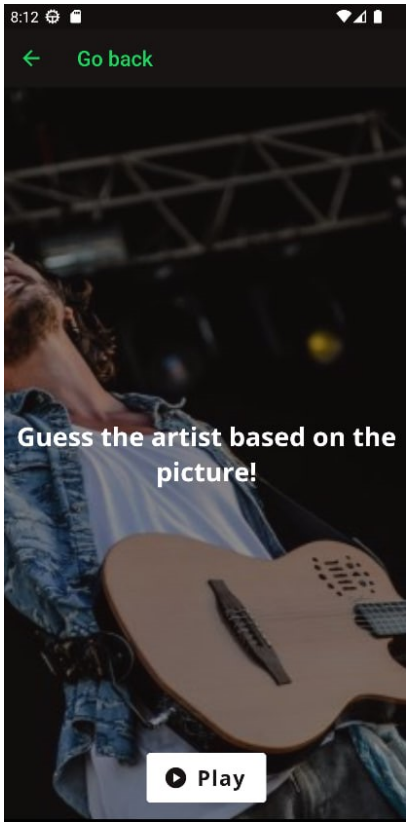
Tablet



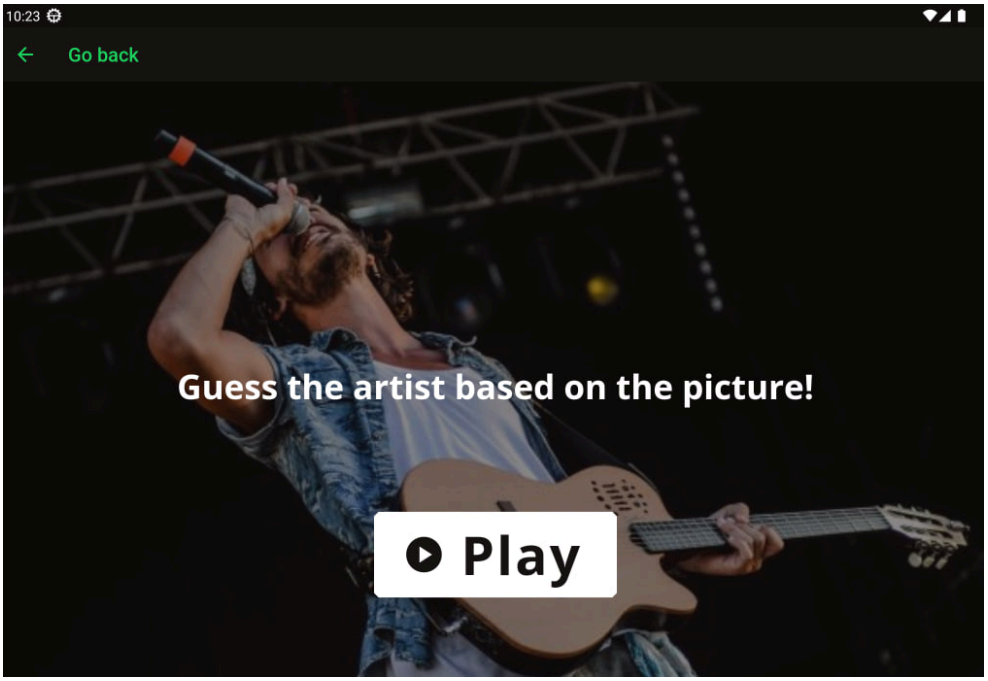
The quiz page is where the user can actually choose the quiz he/she wants to play.

We created four types of quiz and a mode which mix together all of them. Here, the user can also see which are the artists suggested based on his/her preferences on Spotify.

4.2.7 Info Page



Tablet

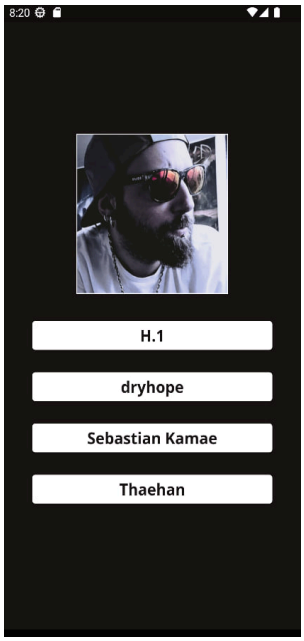


The info page is where the user can see which type of questions he will get once the game will start. From here, he/she can decide to play or to go back to the quiz page.

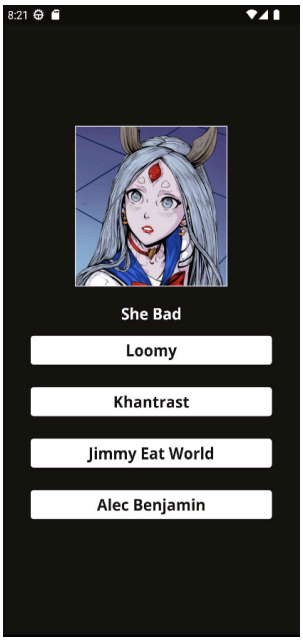
4.2.8 Quiz Screens

Smartphone (Vertical Layout)

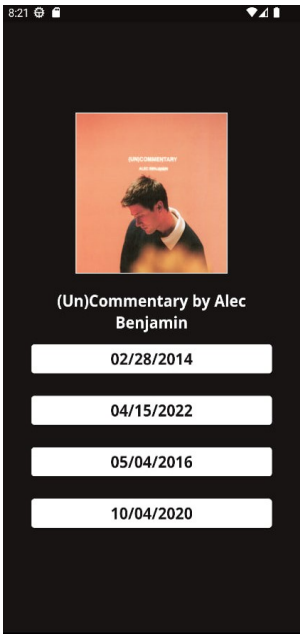
Screen A



Screen B



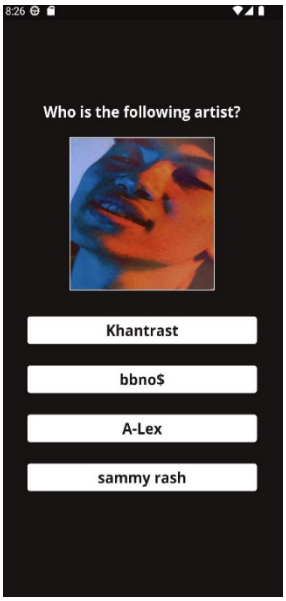
Screen C



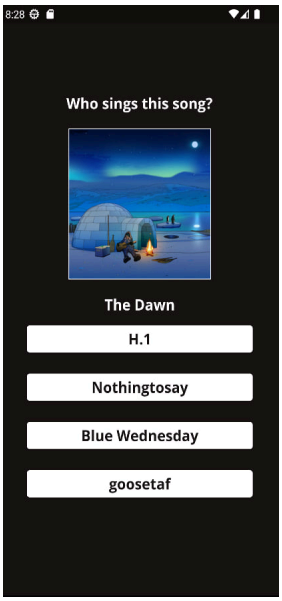
Screen D

Random mode screens

Screen A



Screen B



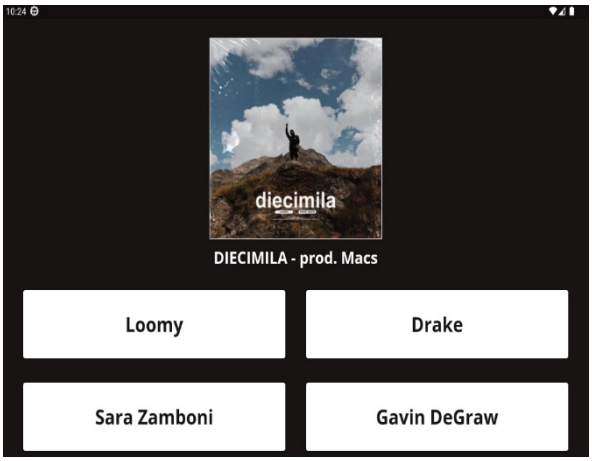
Screen C

Screen D

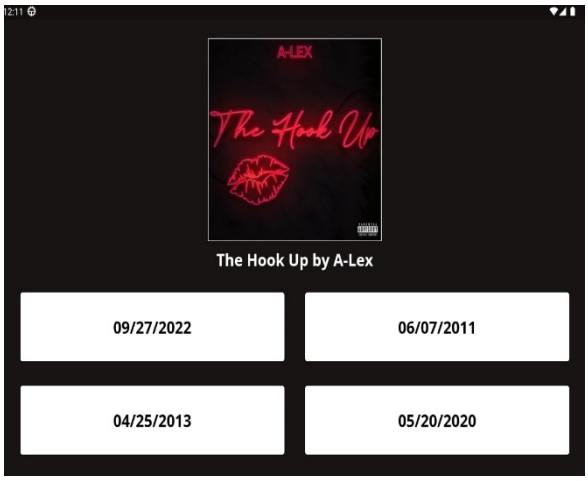
Tablet

Screen A

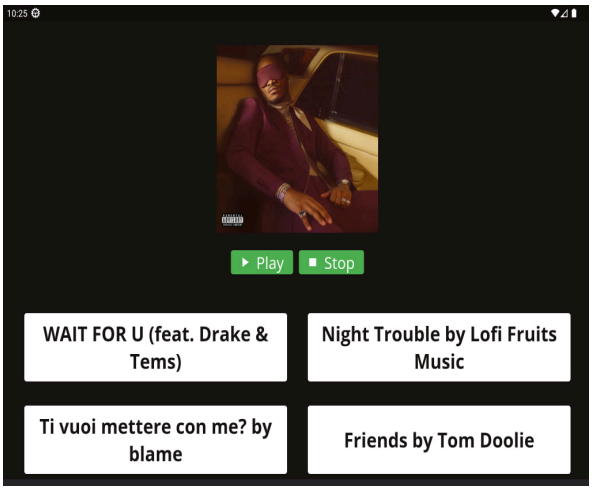
Screen B



Screen C

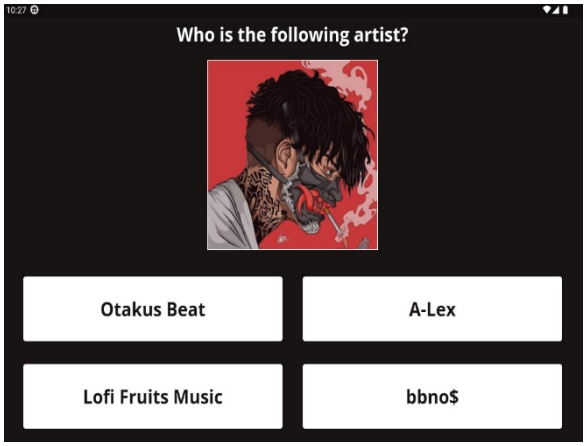


Screen D

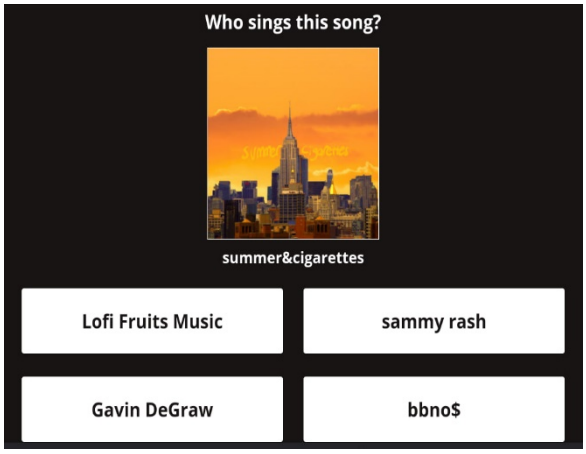


Random mode screens

Screen A

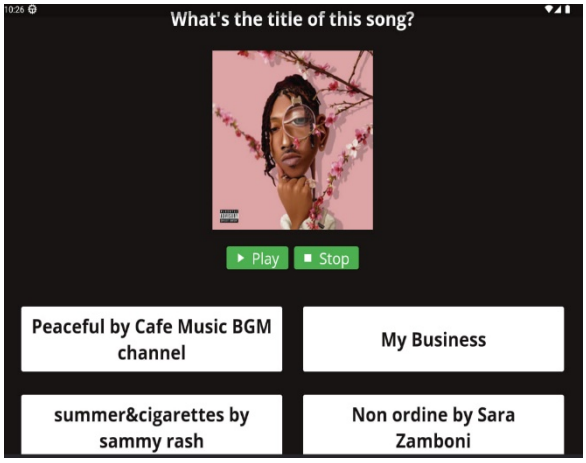


Screen B



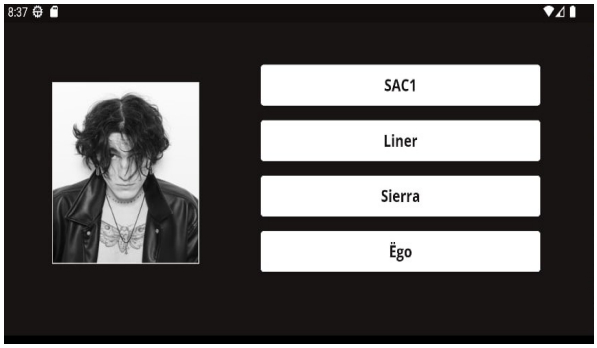
Screen C

Screen D

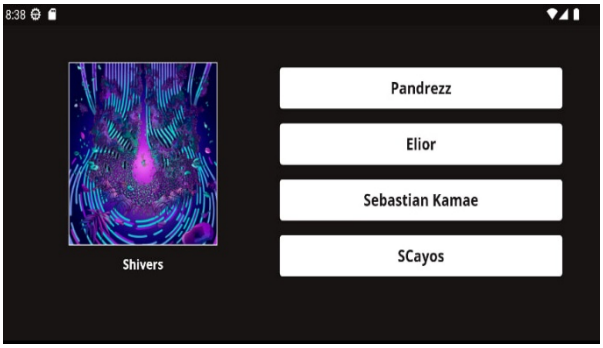


Smartphone (Horizontal Layout)

Screen A

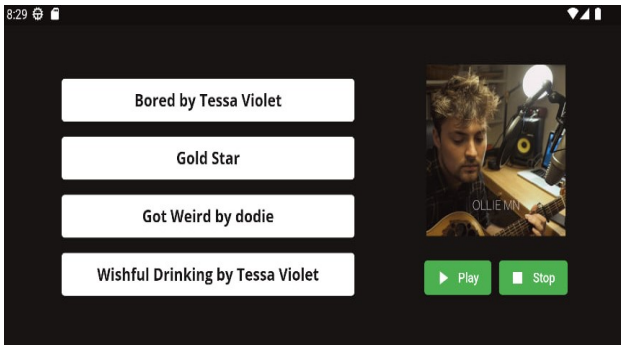


Screen B



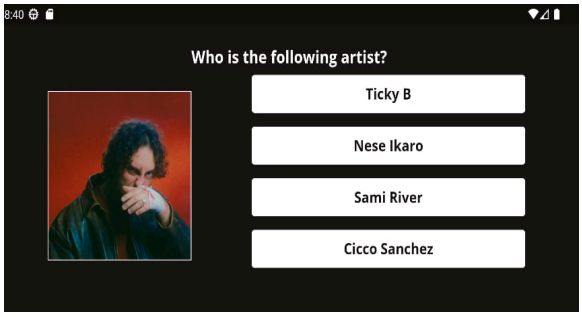
Screen C

Screen D

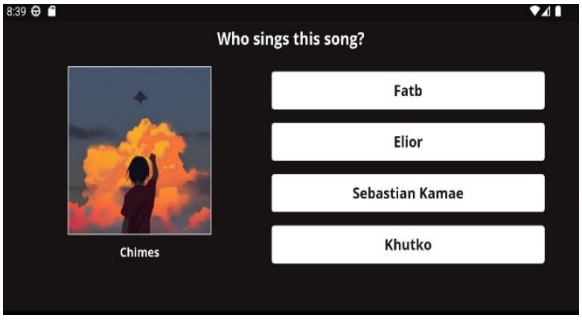


Random mode screens

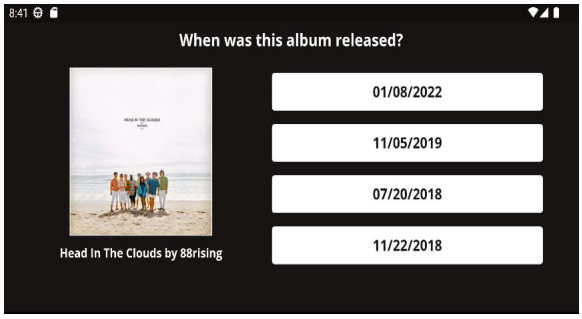
Screen A



Screen B



Screen C



Screen D

The quiz screens are the core of the quiz experience.

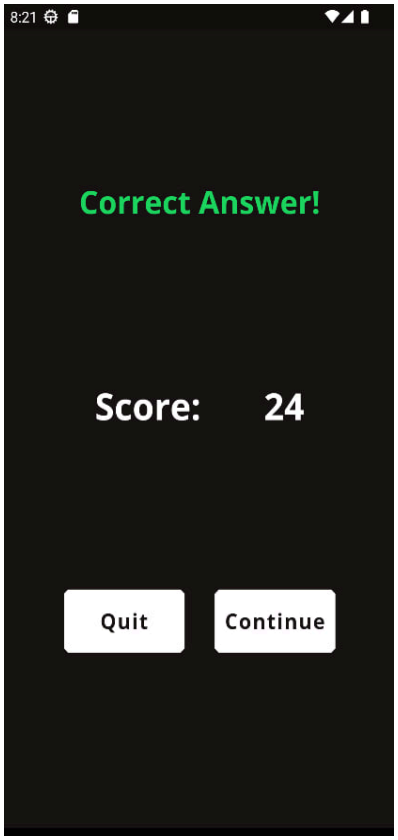
Based on the quiz the player selected, we have four different type of screens with different questions regarding singer, songs and albums.

We differentiated between the normal screens and the "random mode screens" because when the player select the random mode he/she will not know in advance which type of question will be next, so in this specific case we build the screens attaching to them also the actual text of the question.

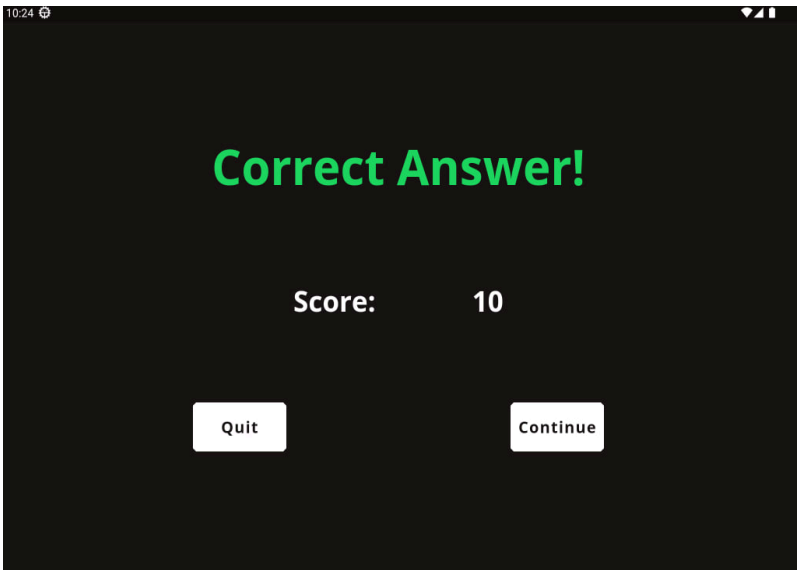
For this part of the app, we created a layout for the horizontal and vertical orientation of the smartphone and also for the tablet, since it's the place where the user will most likely spend the majority of the time.

4.2.9 Result Screens

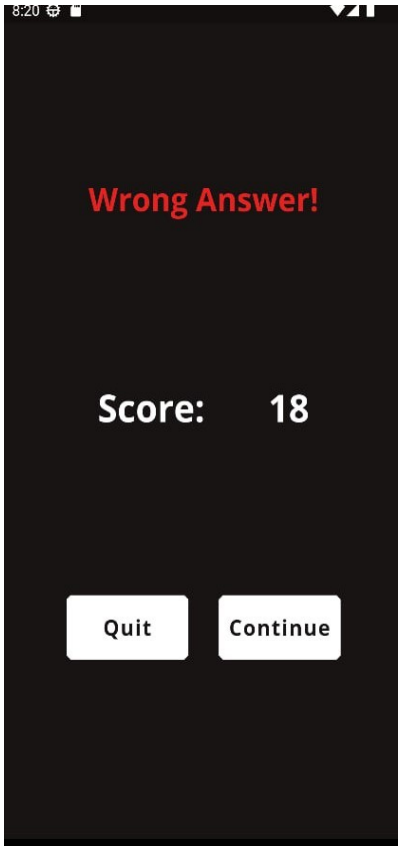
Correct (Smartphone)



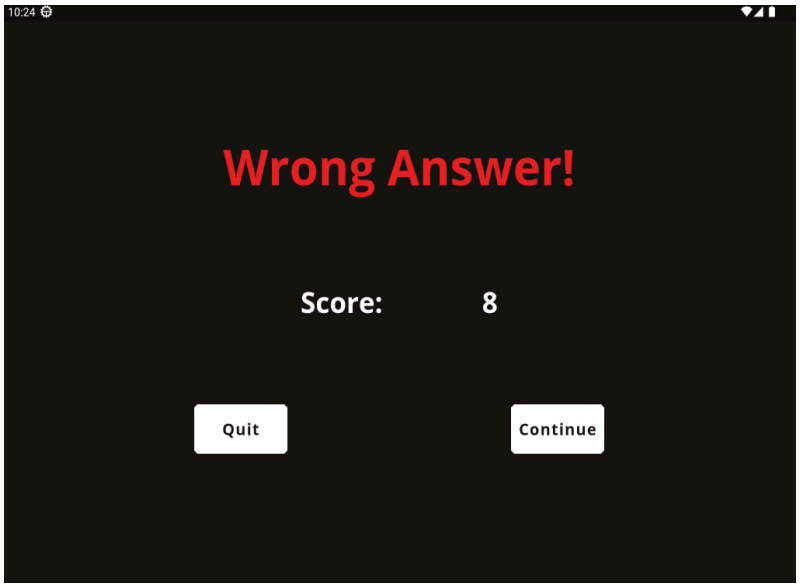
Correct (Tablet)



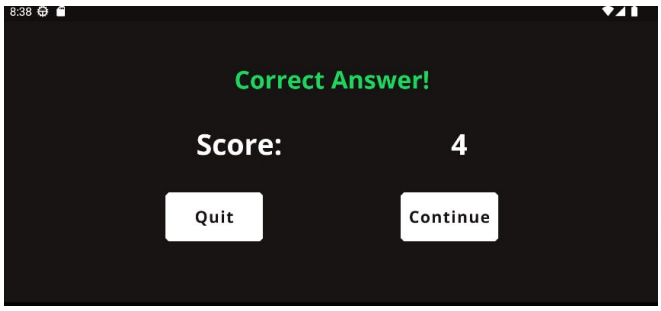
Wrong (Smartphone)

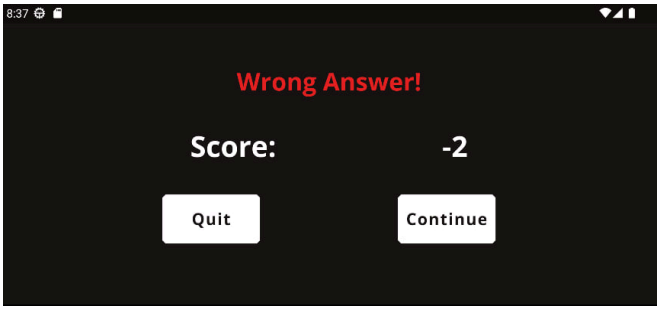


Wrong (Tablet)



Horizontal Layout (Smartphone)





The result screens are the pages following an answer to a quiz. From here, the player can decide to continue with a new question or to return to the previous page.

5. Tests

As for the test campaign, since SpotiQuiz is an app mainly based on interactions with the user, we decided to use integration testing to check that every possible interaction with the widgets went on the correct path and that every view was working as intended.

Starting from the Login Page, we tested all the views for both the tablet and the smartphone versions.

With these integration tests, every time we test a new view of the application, we are passing again through all the views before that one, doublechecking everything.

In the following we will put the images of the tests completed briefly assessing what was their purpose.

We will also put a snippet of code representing a test to give an idea of how they were performed.

We do tests to check every widget in the view and every possible interaction, in this way we are sure that if a test fails, it does for one single reason.

5.1 Example of test

```dart
testWidgets('QuizScreen Layout: Answer 1', (WidgetTester tester) async {
  //setup
  app.main();

  await tester.pumpAndSettle(const Duration(seconds: 5));

  final Finder buttonLogin = find.byKey(const Key('LoginButton'));

  //do

  await tester.tap(buttonLogin);
  await tester.pumpAndSettle(const Duration(seconds: 2));

  final Finder buttonQuiz = find.byIcon(Icons.question_mark);

  await tester.tap(buttonQuiz);
  await tester.pumpAndSettle(const Duration(seconds: 2));

  final Finder buttonGame = find.byKey(const Key("FirstQuizButton"));

  await tester.tap(buttonGame);
  await tester.pumpAndSettle(const Duration(seconds: 2));

  final Finder buttonPlay = find.byKey(const Key("PlayButtonInfoPage"));

  await tester.tap(buttonPlay);
  await tester.pumpAndSettle(const Duration(seconds: 5));

  //test
  expect(find.byKey(const Key("AnswerQuestion1")), findsOneWidget);
});
```

In this example, we pass through the Login Page, click on the Login Button, then on the button to go in the Quiz Page through the Home Page. From the Quiz Page, we press the button to Open the first type of quiz and we press the button to play in the Game Info Page. Once we start the quiz, we expect to find in the page the widget representing the first answer.

5.2 Integration and Widget Tests Smartphone

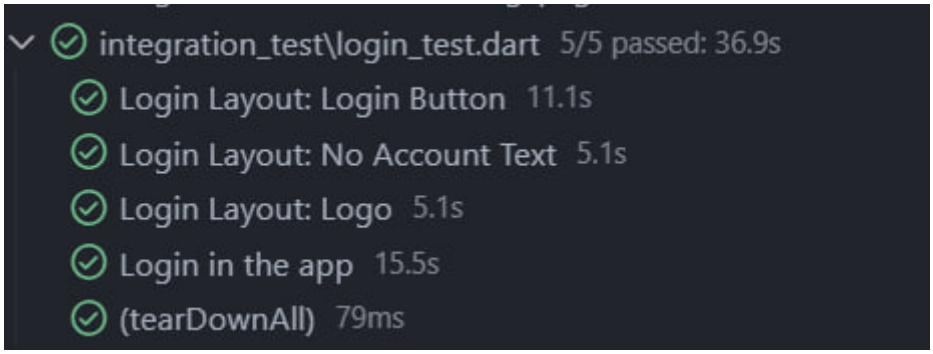
As for the smartphone tests, we did 118 tests in total.

These are done keeping in mind which were the possible interactions between the user and the application and which widgets a user can expect to see in every page of the application.

In every test, we always check that every widget that should be present in the page is displayed, then proceeding in testing the possible interactions.

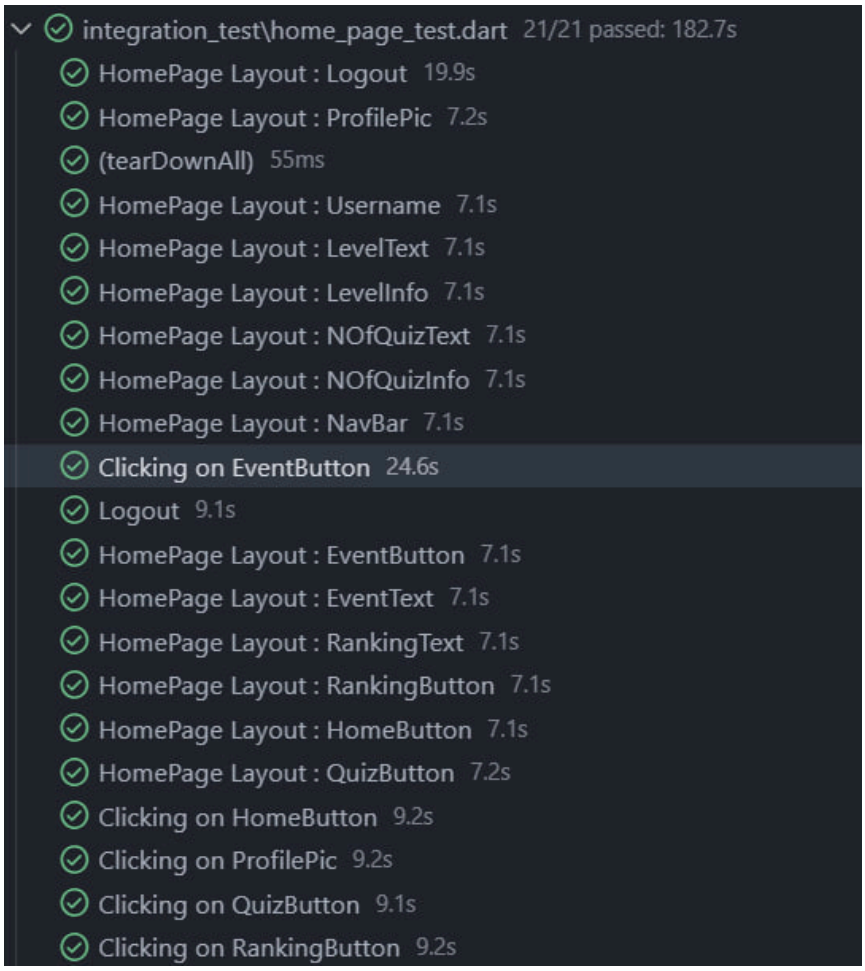
For ease of reading, in the following subsections we skip the description of the layout checks, focusing on the interactions tested.

5.2.1 Login Page



Login in the app: after pressing the login button, we must be redirected to the Home Page of the app.

5.2.2 Home Page



Logout: after pressing on the logout button, we must be redirected to the Login Page.

Clicking on EventButton: after pressing on the event button, we must be redirected to the Event Page.

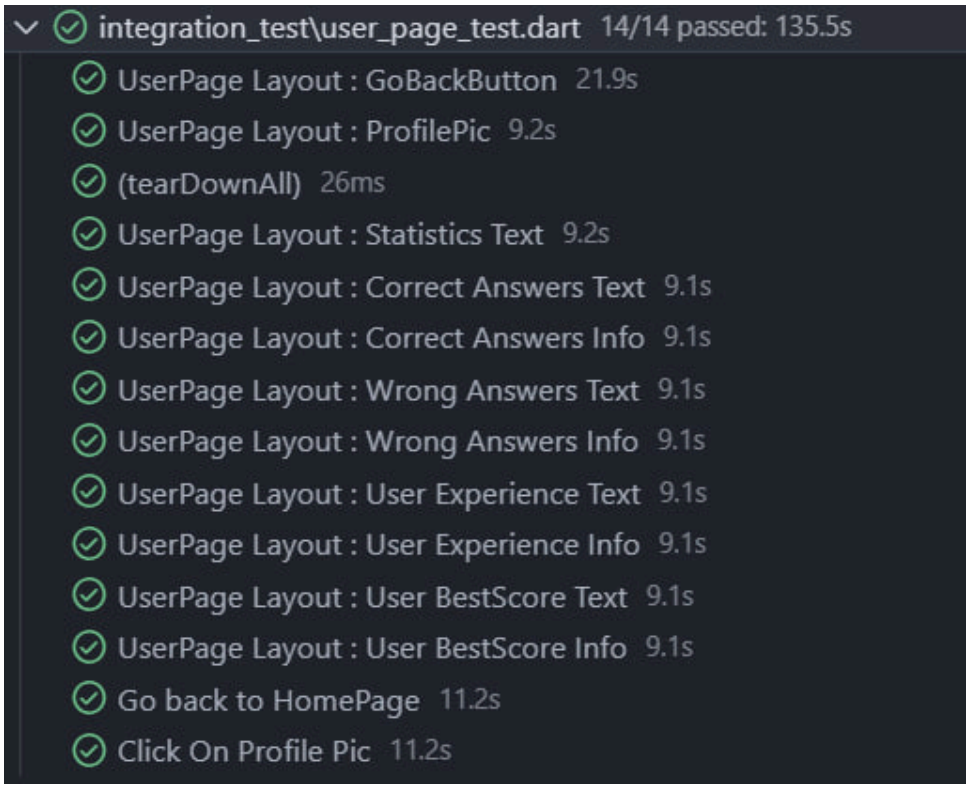
Clicking on HomeButton: after pressing on the home button, nothing should happen, since we are already in the correct page.

Clicking on ProfilePic: after pressing on the user's profile picture, we must be redirected to the User Page.

Clicking on QuizButton: after pressing on the quiz button, we must be redirected to the Quiz Page.

Clicking on RankingButton: after pressing on the ranking button, we must be redirected to the Ranking Page.

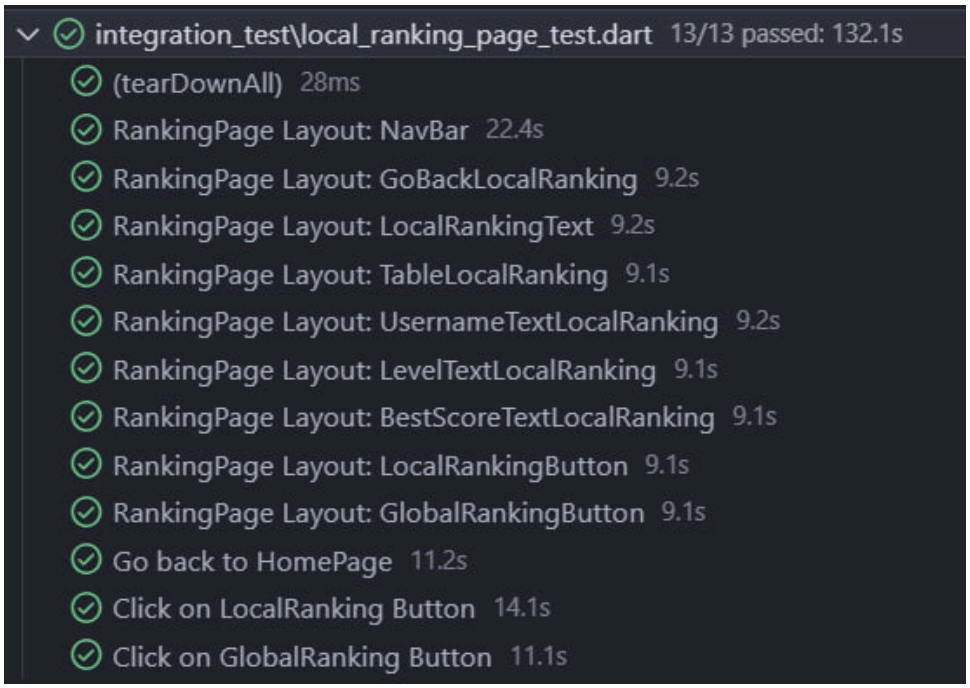
5.2.3 User Page



Go back to HomePage: after pressing on the go back button, we must be redirected to the Home Page.

Click On Profile Pic: after pressing on the user's profile picture, nothing should happen, since we are already in the correct page.

5.2.4 Local Ranking Page

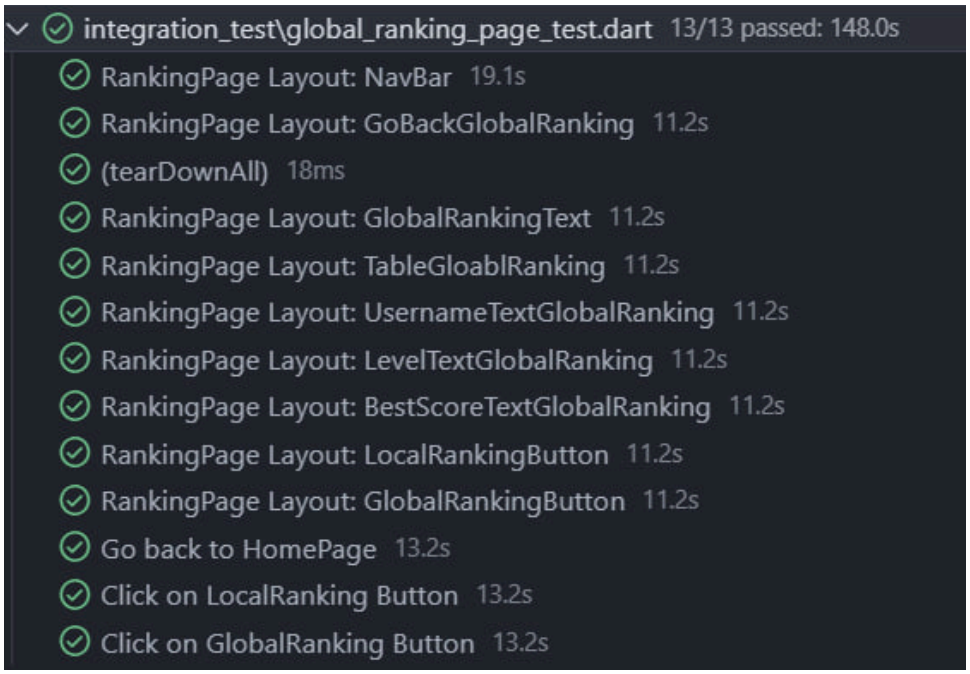


Go back to HomePage: after pressing on the go back button, we must be redirected to the Home Page.

Click on LocalRanking Button: after pressing on the local ranking button, nothing should happen, since we are already in the correct page.

Click on GlobalRanking Button: after pressing on the global ranking button, we must be redirected to the Global Ranking Page.

5.2.5 Global Ranking Page

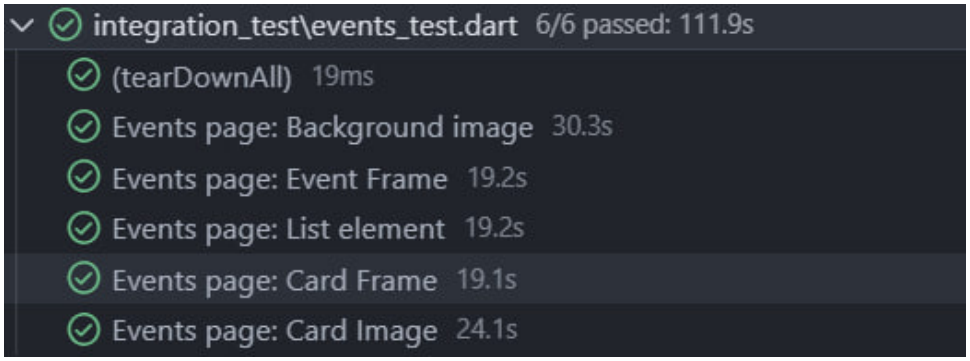


Go back to HomePage: after pressing on the go back button, we must be redirected to the Home Page.

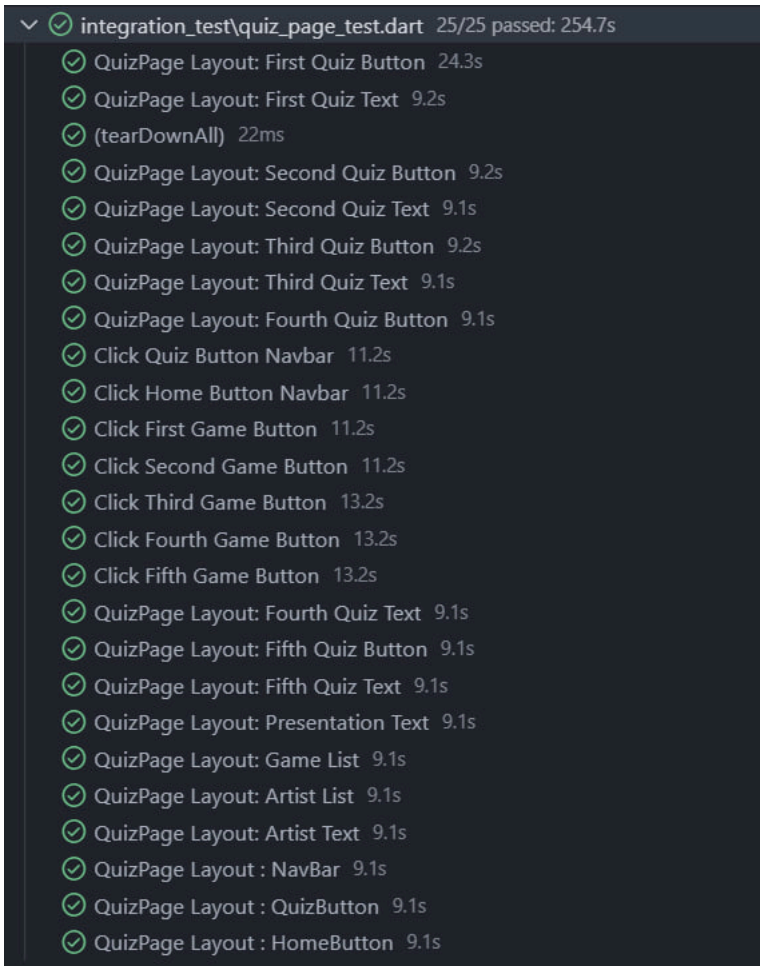
Click on LocalRanking Button: after pressing on the local ranking button, we must be redirected to the Local Ranking Page.

Click on GlobalRanking Button: after pressing on the global ranking button, nothing should happen, since we are already in the correct page.

5.2.6 Event Page



5.2.7 Quiz Page

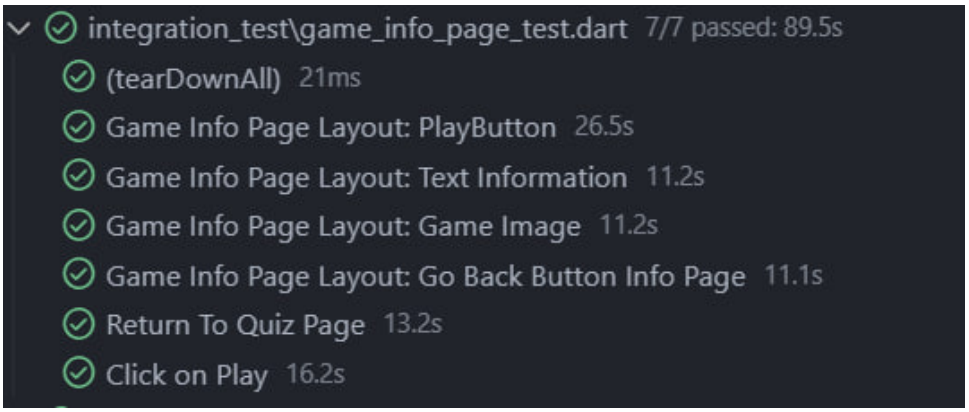


Click Home Button Navbar: after pressing on the go home button, we must be redirected to the Home Page.

Click Quiz Button Navbar: after pressing on the go quiz button, nothing should happen since we are in the correct page.

Click Game Button: after pressing on any of the buttons for the game modalities, we must be redirected to the Game Info Page related to the quiz we clicked on.

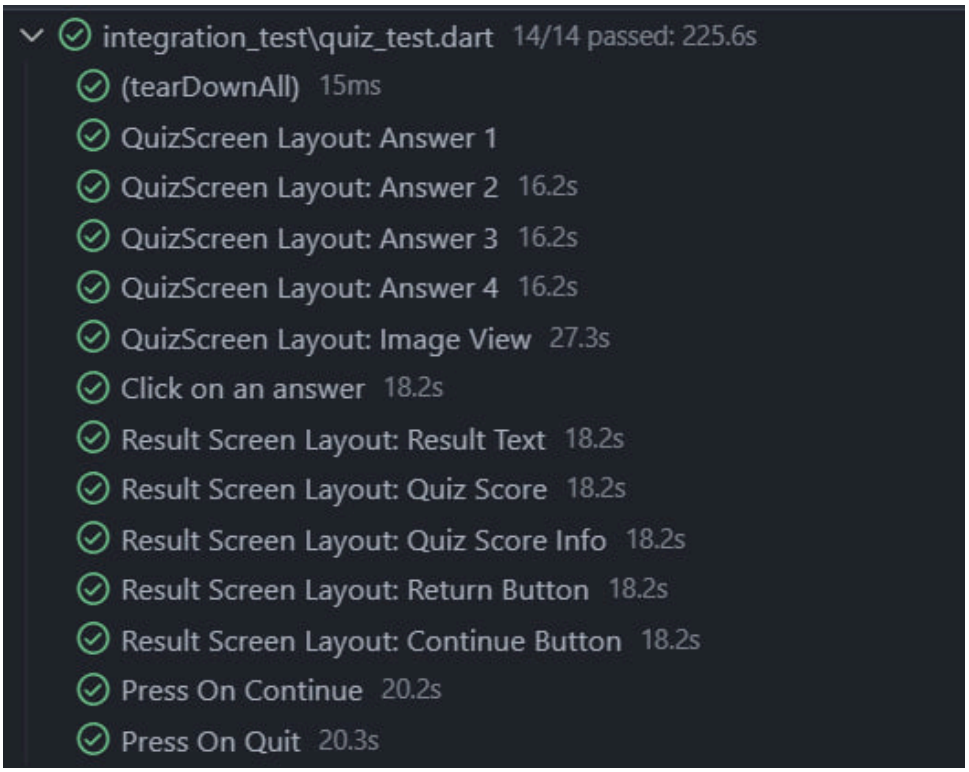
5.2.8 Game Info Page



Return To Quiz Page: after pressing on the go back button, we must be redirected to the Quiz Page.

Click on Play: after pressing on the play button, we must be redirected to the quiz.

5.2.9 Quiz Logic



Click on an answer: when I click on an answer, I must be redirected to the result screen.

Press On Continue: after pressing on the continue button, the quiz must continue with the next question.

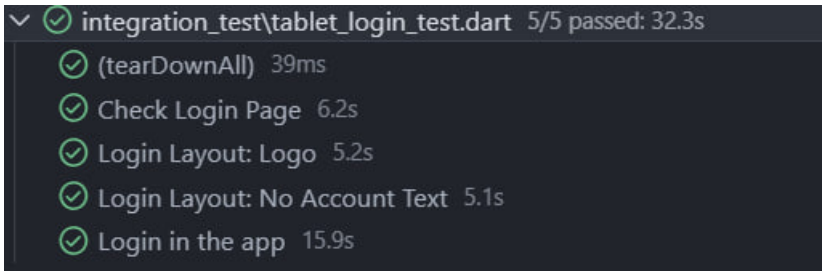
Press On Quit: after pressing on the quit button, the quiz must end and we must be redirected to the Game Info Page.

5.3 Integration and Widget Tests Tablet

As for the tablet tests, we did 113 tests in total.

The principles followed are the same that we used for the smartphone tests.

5.3.1 Login Page



Login in the app: after pressing the login button, we must be redirected to the Home Page of the app.

5.3.2 Home Page

Logout: after pressing on the logout button, we must be redirected to the Login Page.

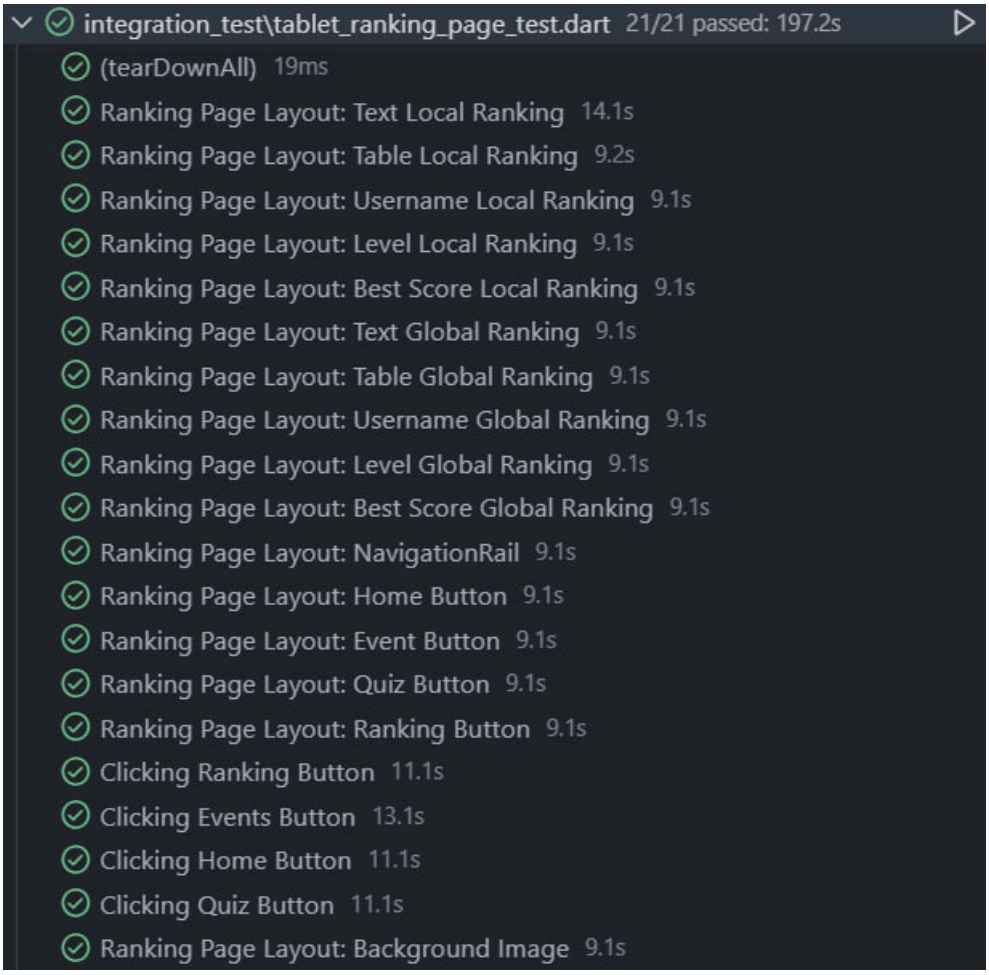
Click Event Button: after pressing on the event button, we must be redirected to the Event Page.

Click Home Button: after pressing on the home button, nothing should happen, since we are already in the correct page.

Click Quiz Button: after pressing on the quiz button, we must be redirected to the Quiz Page.

Click Ranking Button: after pressing on the ranking button, we must be redirected to the Ranking Page.

5.3.3 Ranking Page



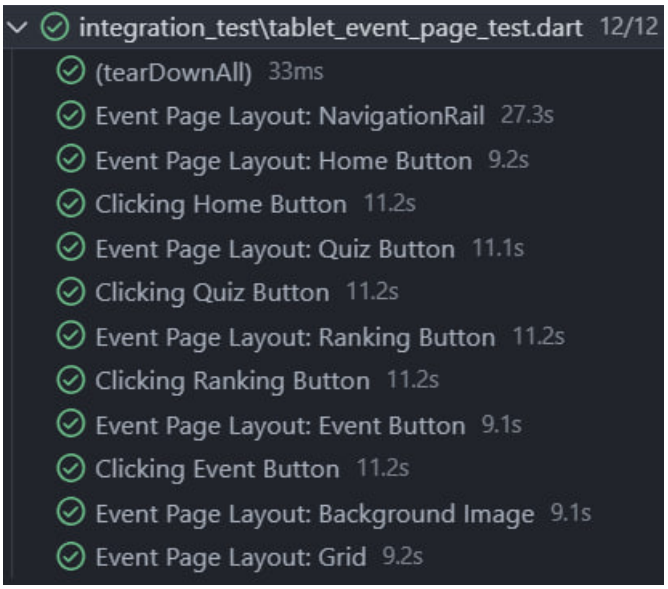
Clicking Event Button: after pressing on the event button, we must be redirected to the Event Page.

Clicking Home Button: after pressing on the home button, we must be redirected to the Home Page.

Clicking Quiz Button: after pressing on the quiz button, we must be redirected to the Quiz Page.

Clicking Ranking Button: after pressing on the ranking button, nothing should happen, since we are already in the correct page.

5.3.4 Event Page



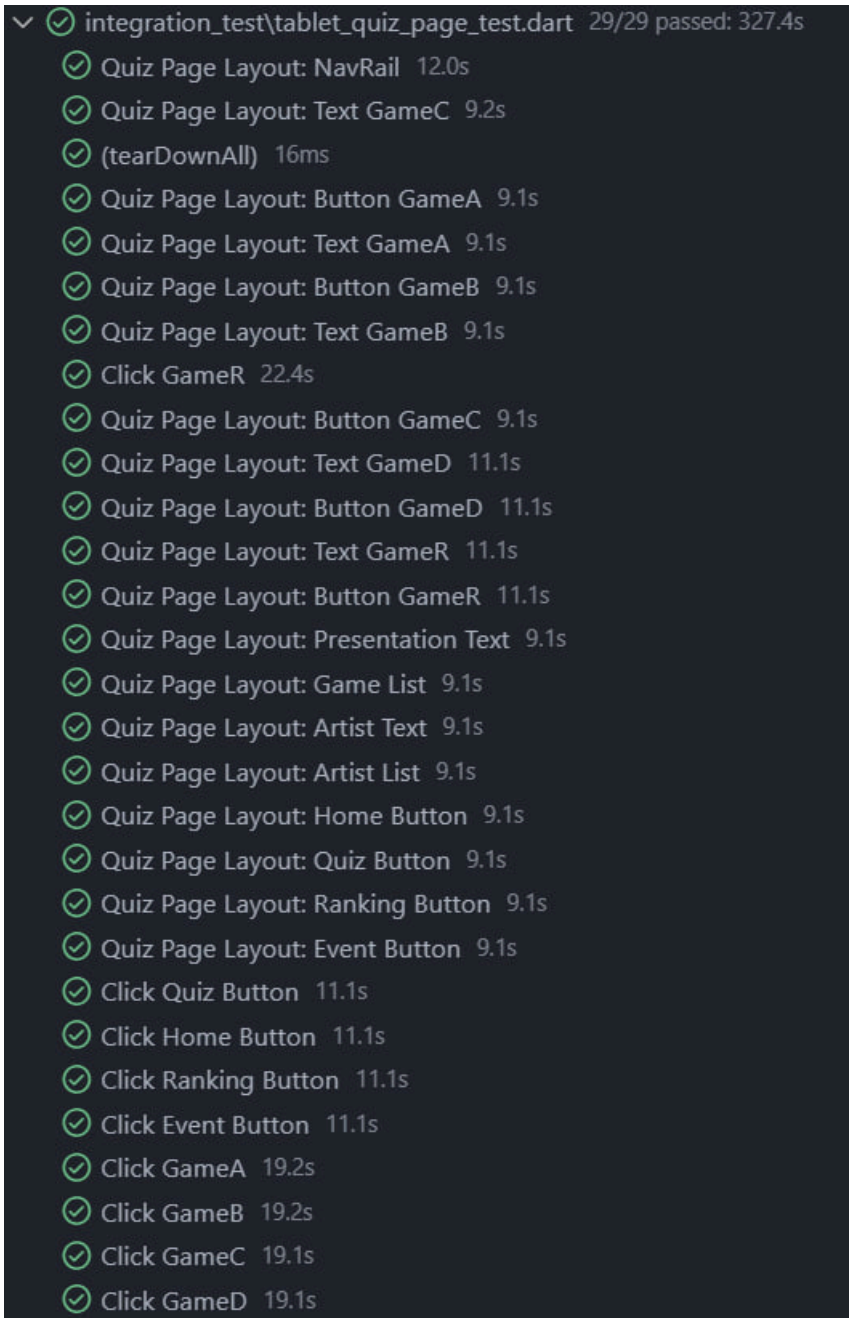
Clicking Event Button: after pressing on the event button, nothing should happen, since we are already in the correct page.

Clicking Home Button: after pressing on the home button, we must be redirected to the Home Page.

Clicking Quiz Button: after pressing on the quiz button, we must be redirected to the Quiz Page.

Clicking Ranking Button: after pressing on the ranking button, we must be redirected to the Ranking Page.

5.3.5 Quiz Page



Clicking Event Button: after pressing on the event button, we must be redirected to the Event Page.

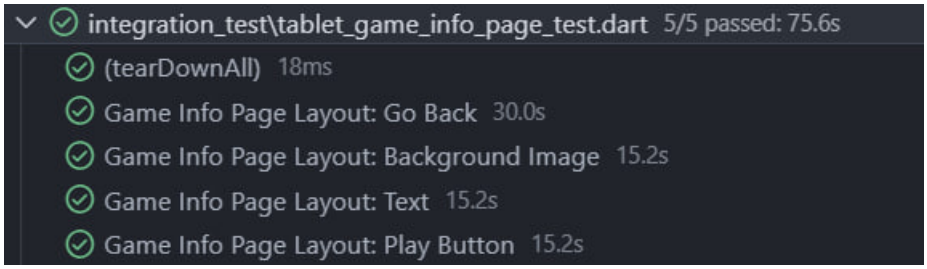
Clicking Home Button: after pressing on the home button, we must be redirected to the Home Page.

Clicking Quiz Button: after pressing on the quiz button, nothing should happen, since we are already in the correct page.

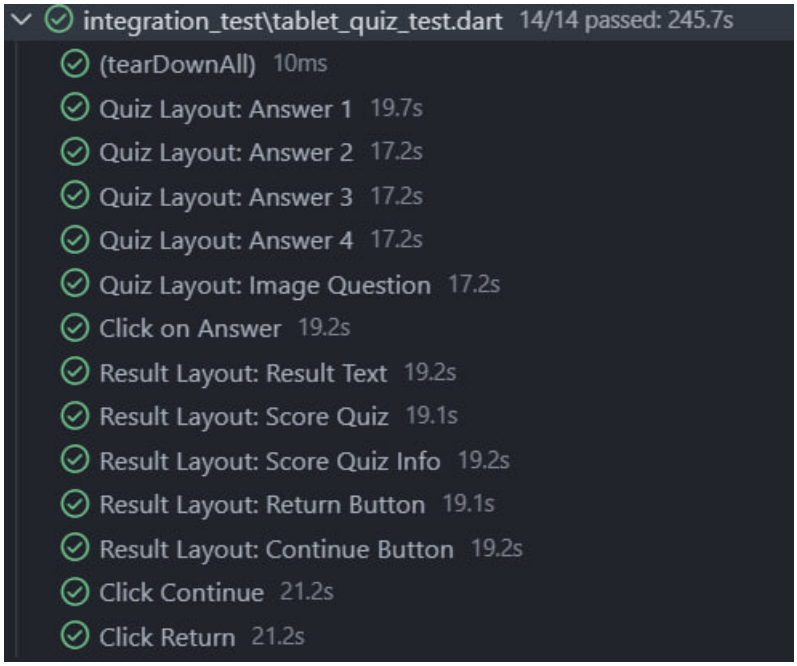
Clicking Ranking Button: after pressing on the ranking button, we must be redirected to the Ranking Page.

Click Game: after pressing on the button of a quiz, we must be redirected to the Game Info Page of that game.

5.3.6 Game Information Page



5.3.7 Quiz Logic



Click on Answer: when I click on an answer, I must be redirected to the result screen.

Click Continue: after pressing on the continue button, the quiz must continue with the next question.

Click Return: after pressing on the quit button, the quiz must end and we must be redirected to the Game Info Page.

5.4 Unit Tests

Unit tests are a type of software testing that focuses on verifying the correctness of individual components or units of code. A unit refers to the smallest testable part of a software application, such as a function, method, or class.

Overall, unit tests play a crucial role in the software development process, helping ensure the correctness and reliability of individual code units before they are integrated into larger components or the overall system.

5.4.1 REST API

The API services the platform uses are accessed through the use of auxiliary functions which manage the http calls and work on the raw data received from the REST APIs in JSON format. Since the whole platform works upon these services it is very important that each functionality is tested, thus the unit tests are done on all the main functions which comprise the following:

```
Future<List<model.Event>> get_events_on_position(String? position)
Future<List<model.Artist>> get_related_artists(String artist_id)
Future<List<model.Album>> get_artist_albums(String artist_id)
Future<List<model.Artist>> get_followed_artists()
Future<List<model.Artist>> get_artist_quizpage()
Future<model.Artist> get_artist(String artist_id)
Future<List<model.Track>> get_top_tracks(String artist_id)
```

In particular:

- “get_events_on_position” manages the events loaded on the events page
- “get_related_artists” manages to load all the artists related to a particular artist given its id and it is extensively used in the quiz generation process to always guarantee a personalized experience.
- “get_artist_albums” loads the albums of an artist given its id, used in the quiz generation process.
- “get_followed_artists” loads the followed artists of the current user.
- “get_ artist_quizpage” create the and loads the artists models to be displayed in the quiz homepage.
- “get_artists” loads the information of a given artist and it is used in the quiz generation process.
- “get_top_tracks” loads the tracks of a given artist.

All these tests are simulated in two different scenarios: Success and Failure, to analyze the behavior of the application in both cases.

5.4.2 Questions

The questions make up the entirety of the quiz functionalities and thus a big percentage of the whole application. So to guarantee an exceptional user experience and reliability of the whole project every single auxiliary function which manages the creation of the questions has been thoroughly tested. Following a list of the functions tested:

```
generate_a()
generate_b()
generate_c()
generate_d()
```

5.4.3 Quiz

The creation of a quiz is managed through the use of auxiliary functions which manage not only the number of questions created but also their type and diversification. The unit tests have been done on the following functions:

```
generate_quiz("A");
generate_quiz("B");
generate_quiz("C");
generate_quiz("D");
generate_quiz("R");
```

Each letter indicates a different type of quiz.

5.4.4 Database

SpotiQuiz works largely using a Firestore API database to manage the storage of users' data such as username, profile picture, number of quizzes played and so on. The unit tests have been done on the most important database interactions:

```
test('getUsers()', () async {…

test('Call success: getUsersByNation()', () async {…

test('Call failure: getUsersByNation()', () async {…

test('Call success: getByID()', () async {…

test('Call failure: getByID()', () async {…

test('NumOfQuiz updated: UpdateAfterQuizOnDB()', () async {…

test('Wrong Answers updated: UpdateAfterQuizOnDB()', () async {…

test('Correct Answers updated: UpdateAfterQuizOnDB()', () async {…

test('Best Score updated: UpdateAfterQuizOnDB()', () async {…

test('Experience updated: UpdateAfterQuizOnDB()', () async {…

test('Level updated: UpdateAfterQuizOnDB()', () async {…

test('apiGetUser()', () async {…
}
```

All the unit tests are performed using a success scenario and a failure scenario.

5.5 Results

With a total of 266 tests, we were able to assess the quality of the final product.

During the development process, testing helped us in detecting some problems, like the unproper call to some asynchronous function or the presence of future functions that weren't properly waited by the caller.

Bibliography

[1] https://developer.spotify.com/documentation/web-api

[2] https://developer.spotify.com/documentation/design#using-our-colors

[3] https://docs.flutter.dev/

[4] https://pub.dev/packages/intl

[5] https://docs.flutter.dev/testing/integration-tests

[6] https://firebase.google.com/docs/database?hl=it

[7] https://www.photopea.com/

[8] https://www.craiyon.com/