

EöTVöS LORÁND UNIVERSITY FACULTY OF INFORMATICS DEPARTMENT OF PROGRAMMING LANGUAGES AND COMPILERS

User friendly chat App

Turcsány-Szabó Márta Edimarf Jaka Jamba Satumbo

Supervisor Computer Science BSc

Budapest, 2020

Chapter 1

**INTRODUCTION**

Motivation

Over the past 30 years, internet access is rapidly spread across people, Popularization of smartphones which becomes part of people’s life, and making the interaction of people easiest than before, Communication, sharing of files and many others human relationships. We all have benefited from the existence of internet, but are we safe using it? besides the hacker’s attacks there is no mystery why modern social media platforms were designed to be addictive: the more we consult them, the more data they have to fuel them which enables them to grow smatter and bigger and more powerful. In the year of 2019 many platforms lost thousands of users, especially Facebook which is one of the biggest platforms in the world, after users comprehend how the company was using their personal information. It is important to mention that the term “open-source software” has been going around the internet for a while. It’s actually fairly self-explanatory, even though that might not seem immediately obvious. Basically, there are two types of software: open-source and closed-source, the difference between them is whether the end user has access to the source code. In open-source software, users can view, edit, and delete some of the code. This can be useful for modifying the software to your specific needs, embedding the software inside of an existing program or website, or just taking a few bits and pieces of the software that you need. On the other hand, closed-source software does not allow users to change any information in the source code. The most notable example may be operating software for your computer, and many others software, apps, websites which has a big position in the market place. What I would like to point is, any software, social media app, website which refuse to share their code as “open-source” is less trustful, end users should have broad knowledge on how and where they personal data is going to used and for what purpose. This is the main role of this “User friendly App”. UFCA (User Friendly Chat Application) is a chat app designed for the user safety, providing communication between them requesting few personal information. The app is composed by public groups, post session and event suggestions, strong high security with an open-source, therefore end users can have a broad knowledge of what and how they data are strongly safe and managed, if possible, contribute to it in the possible future.

The application was built above JavaScript framework named React Native, which helps writing real natively rendering mobile application for iOS and Android, based on React Facebook’s JavaScript library for building user interfaces, but instead of targeting the browser, targets mobile platforms and gives you more efficient code sharing across the mentioned platforms without scarifying the end user’s experience or application equality.

This cross-platform app development is seemingly becoming popular as the stratum of competition is exceeding higher up the order, and beyond any doubt, React Native has been identified as the most preferred cross-platform solution for the creation of both Operating Systems using a single platform. Undeniably, React Native is a real asset when it comes to improving the performances through native control modules, getting connected to the native components for both the Operating Systems and generates a code to the native APIs upfront and freely. The performance enhances due to the fact that it makes use of a different thread from UI ad the native APIs. Also, the code reusability, the possibility to integrate 90% of the native framework for reusing the codes for both the OS, helping to save a lot of time and also curtail the cost of app development as well. What is more pleasing to note that developers can also reuse the web application codes for creating mobiles app, if they are written in React.

Today, everything is going mobile and cloud, the success of the mobile app is its usage, engagement, and retention because it increases the in-app purchases and boost the app sales. And, storing mobile app data on the cloud makes a huge difference nowadays.

Here, Mobile Backend as a Service (BaaS) proves quite helpful to build functionalities using the cloud that help you offer in your mobile app. Mobile Backend as a Service (MBaas) is a medium that offers a way to link web and mobile application to the backend cloud storage and backend APIs. Commonly known as Backed as a Service (BaaS), MBaaS features like sending push notifications, integrating the cloud storage, user management, and social networks. There are various well-known MBaaS providers such as AWS, Kumulos, Meteor, Kinvey, StackMob, Applicasa, and so on, but the one use for the development of this project was Firebase.

Utilization of these tools in our project helps with many of the challenges we face, and makes the development process more efficient and structured. It also provides us with a solid infrastructure to build the application on, and to extend it with new use cases and features.

**Thesis Structure**

Chapter 2 of this thesis contains a user documentation, which provides a description of the application and its use cases, a brief overview of the technical methods used in the implementation and how it affects the user experience, and a complete user’s guide for all of the available features of the application.

Chapter 3 is a developer documentation, containing all the technical details of the application. In this chapter we specify all the tasks and challenges of the development process, and provide detailed information about the concepts and components used in the application - including how they function, and how they connect with each other to shape the functionality of the application. API documentation of the main code component of the application can be found in the appendices, which provides detailed information about the classes and functions used in the application.

Chapter 2

**USER DOCUMENTATION**

**Project Description**

The main goal of this project is to create a user-friendly chat app where users can communicate with each other safely, sharing important information, with few personal information required. Focusing on the user data protection the app only supports public groups, users can join any group from the platform using features that makes conversation more interesting such as; image and location sharing. Users can have a broadcast video call among them, the number of members is unlimited, functions can such microphone and camera can be enabling and disabling depending of user preferences.

To provide a better iteration among users there is two important session, beside the Chat groups. The `Post` session allows user to have a small social media inside the app, where they can share images and a small description about it, other users can also have access to the same post, with options to save or share with extra apps of their wish. The `Event` session is a small part of the app, that users can give some highlights of the upcoming events around the world, more information of the same events are only accessible from an extra linking provide on the description.

High user-friendly UI provided for end users, and strong data protection for they safety.

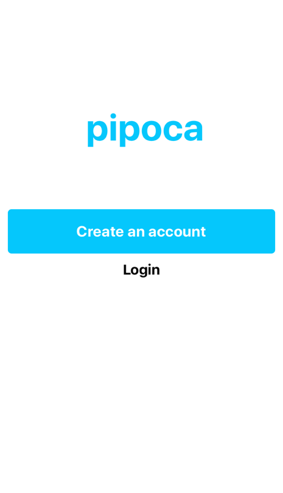
Additionally, chats are provided as public and video calls as broadcast (“live video call”), but they are implemented in the way that allows extending the application of possible implementation of the feature peer-2-peer communication, if in the future it require.

In this project, we have used React Native which is an excising framework that enables web developers to create robust mobile applications using their existing JavaScript knowledge. It offers faster mobile development, and more efficient code sharing across iOS/Android and the web, without sacrificing the end user’s experience or application quality. Using React Native solution, we are able to deliver a scalable and highly quality applications, saving in development time, cost and the effort could be cut by half.

Cloud technology is used in this project to contain and process all of the application resources, and offers on-demand storage and processing power that we require for the application. Using Firebase removes the need for buying or renting servers to hold the application, and eliminates the burden of having to maintain and configure the servers and lets us focus only on development on a reliable infrastructure.

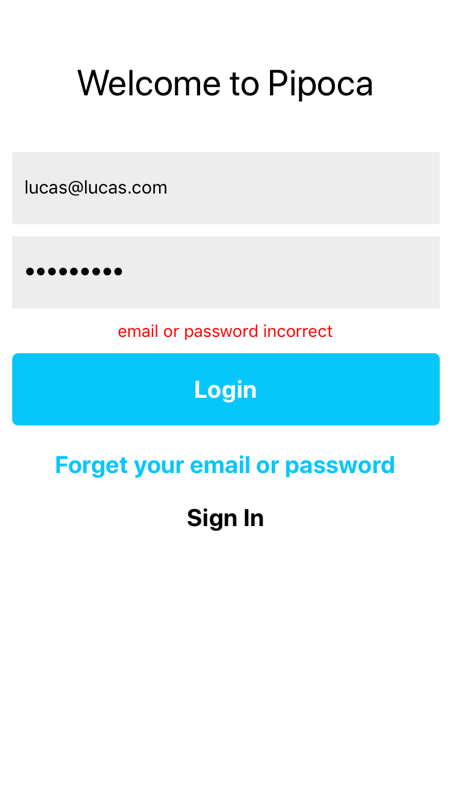
**Usage Information**

Getting Started

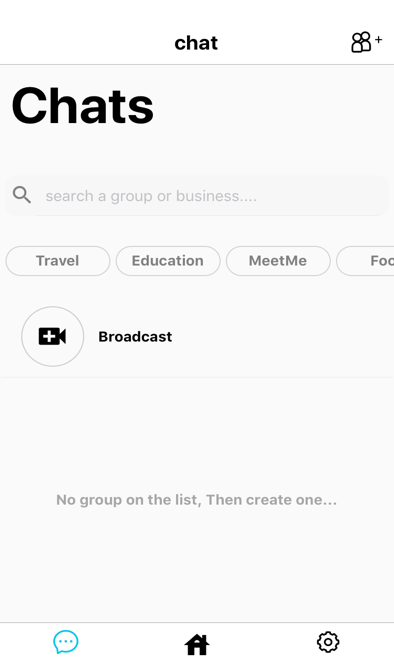


When the app is started, `Get Start` screen is present, where they will have options to create or login to their exist account. Click on the “Create an account” implies that user is not registered yet and pretends to create one. From the `Sign in` screen contains a form to be filled, requesting `email and password` as a first step. Only with valid email and pass word is possible to complete the session. After a successful API check as last form information such as name, id, location and bio will be need to finalize. The user’s `Bio` will contain as maximum length 40 characters. This information is extra from users, and they are required. With a successful completion from the required data, clicking on “Finish” a new account will be created, allowing users future unlimited access to app.

In case of having already an account clicking on “Login”, user will be asked to provide they valid `email and password`, if success on login the main screen will be present otherwise, they will face an error message similar as the figure below:

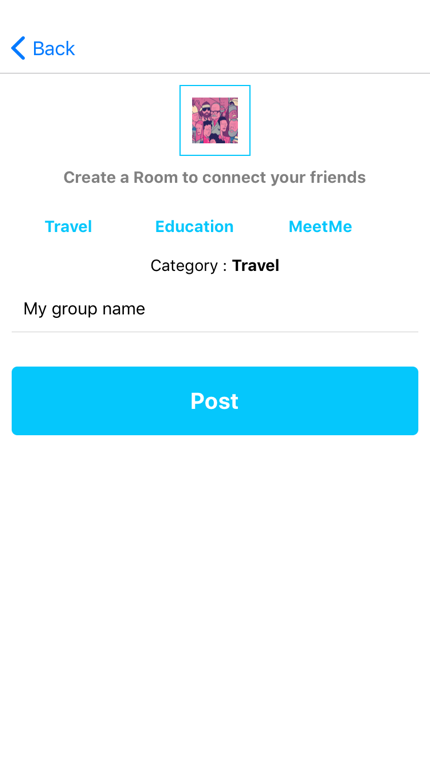
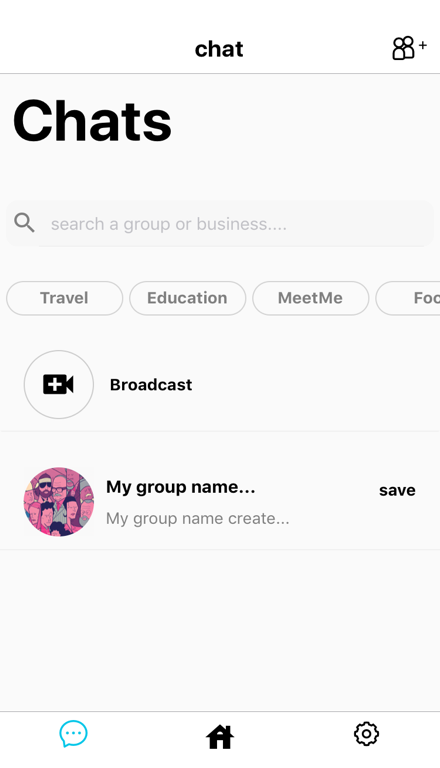
****

**Creating group**



Upon on the top right part of the screen there is an icon bottom, which

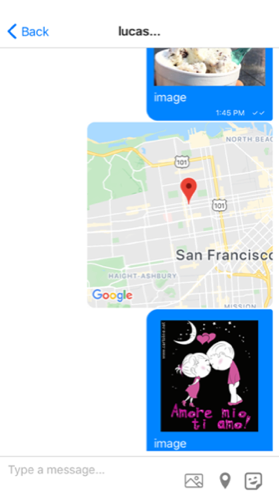
clicking on it navigates to the `Create group` screen. User will face a new screen which contains a form that requires a name and image for the group, default information is provided by the app such as: name, image and category. This information can be altered, by the user preferences. The categories are important to distinguish the topics of each groups, making easy the search of the topics in the app.

As the last step user must click on the “Post” which finalize the process and posteriorly navigate user back to the Chat screen where the new group can be found.

**Starting a conversation**

From the `Chat screen`, a list of groups can be found, to be able to start a conversation, a simple click on the desired group can navigate the user to the ` Message screen`, where they can send message, images and specially their location. Taking in consideration the user’s privacy, all of these features are possible to use if and only if users permit and agree. It is only possible to navigate through the map from the chat, future connections with Maps application will be implemented if in the future require.

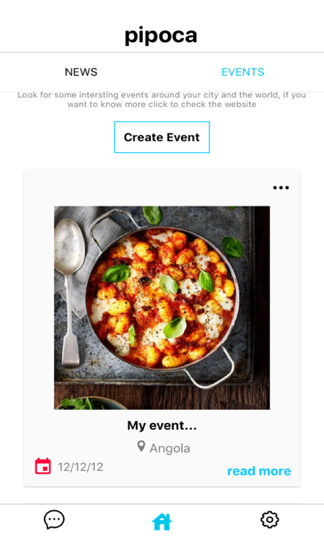
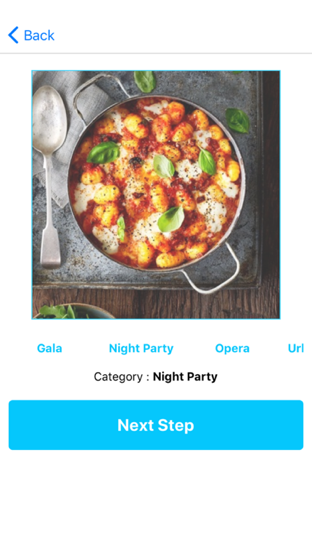
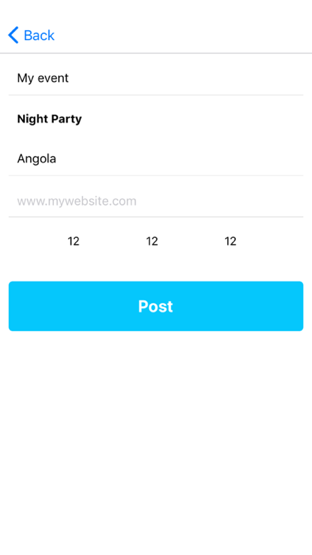
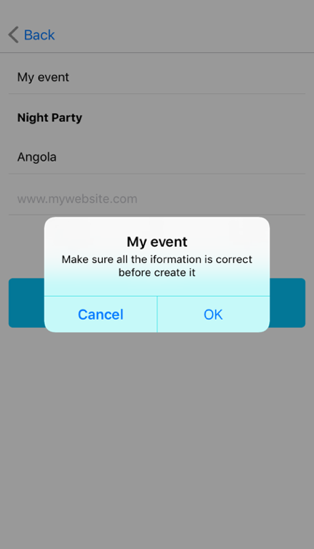
**Making a video call**

The app offers a broadcast video call, which any number of users are allowed to participate, from the Chat screen a ` Broadcast sections` is found right on the top of group list, clicking on the icon button will navigate the user to the video call session, where they can communicate using features such as microphone and camera. This function can be enabled and disabled by the user preference.

**Creating post and events**

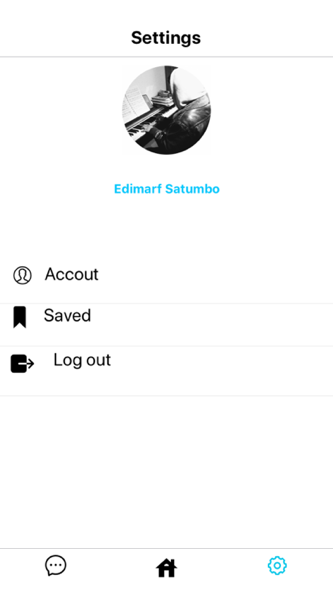
To be able to create a post or events user must navigate to `Home screen` which from the tab navigation is designed as the `home icon`. The `Home screen` is divided in two session; post and events. From the post session users: right on top of list of posts a button composed by icons can be found, clicking on the button, we will navigate to a next screen, and provide image and description for the post. Description is considered as optional rather than image which is requirement. Pressing `post` a new post will be created and prosperously navigate users to the previous screen. Some functionalities will be available to user, they will be allowed to save, share link and delete their own post.

From `Events` session, a button named `Create Event` will be located right on the top of event lists, clicking on the button, a new screen will be present, where they will have a form to fill out with necessary information such as: event name, category, location, website link and date of the event. A website and respective date of the event are required, with a successful completion clicking on the “Post” button will create a new event and navigate user to the event list.

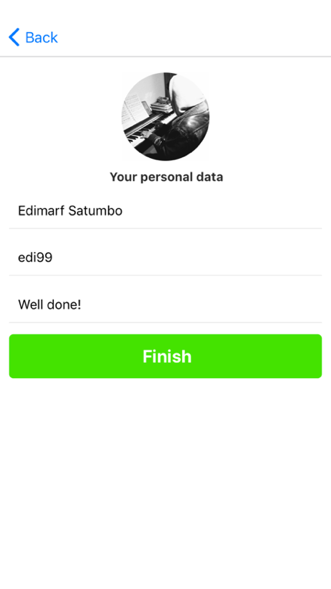
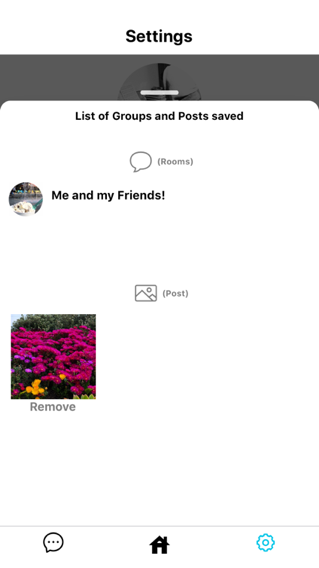
   

**Settings**

Settings is the last session of the app, where users can update and delete some private information, with the possibility to alter they personal data, saved list of post and groups, and logout the app at any wished time. User can navigate to the “Account”, where they will have access to the extra information provided during the account registration such as: user name, id, location and bio.



Important to remind that groups and posts can be saved, to be able to get access to them user can navigate to the “Saved” session, they can get the names of the favorite groups even previsualize images that they have saved.

The last part we have the “Logout” button, clicking on it means that users want to leave the app, and automatically disconnect and navigate them to Get Start session, where they can login again at any time they wish.

**Reset password**

In case user forget his password, a session is provided, where user with valid email, can update their password, by requesting a link which enables to them to reset through a website.



Chapter 3

DEVELOPER

DOCUMENTATION

**Project Description**

The goal of this project is to create a modern user-friendly platform to provide communication between and keep their information safe. This platform is offered as mobile application. The main challenges of the project can be summarized as following:

* Providing a secure way of authenticating users into the system.
* Creating a modern user-friendly interface.
* Providing video call using WebRTC technology
* Maintaining the user data and list of chat from each group to the Database.
* Creating a logical structure that can be easily maintained and extended with new features.
* Allowing user to send their own location through the chat.
* Converting images to jpg format to be able to handle them in Database.
* Implementing libraries that requires real devices, and overcome a new way of implementation
* Optimizing functionalities for a well performance of the app
* Provide the right rules to store the messages in Firebase Database.
* Share location through google API from different platforms iOS/Android
* Provide Broadcast calls with free APIs
* Moving from Expo to React Native CLI
* Reduce the work from Socket io to Firebase, message, communication between user in just one library.
* Keeping the same UI form both devices was one of the hardest parts, since two distinct platforms will have absolutely different behaviors.

**Used Technologies and Methods**

**Backend**

The backend is implemented using Firebase which is Google database, for mobile and web development, supporting Android/iOS platforms. For the video calls we used WebRTC for real-time communication capabilities that works on top of an open standards. It supports video, and generic data to be send between peers, allowing developers to build powerful voice-and video-communication solutions. The technology is available on all modern browsers as well as on native clients for all major platforms. The technologies behind WebRTC are implemented as an open web standard and available as regular JavaScript APIs in all major browsers. For native clients, like Android and iOS applications, a library is available that provides the same functionality. The WebRTC project is open-source and support by Apple, Google, Microsoft and Mozilla, amongst others, but in this project, we are focused on mobile environment.

**Frontend**

The frontend is implemented using React Native, an open-source mobile framework based on JavaScript that allows you to build natively-rendered mobile apps for iOS and Android. This framework lets us to create an application for various platforms buy using the same codebase. The fact that React Native actually renders using its host platform’s standard rendering APIs enables it to stand out from most existing methods of cross-platform application development, like Cordova or Ionic. Existing methods of writing mobile applications using combinations of JavaScript, HTML and CSS typically render suing Webviews. While this approach can work, it also comes with drawbacks, especially around performance.

For developers accustomed to working on the Web with React, this means you can write mobile apps with the performance and look and feel of a native application, while using familiar tools. React Native also represents an improvement over normal mobile development in two other areas: the developer experience and cross-platform development potential.

Working with React Native can dramatically shrink the resources required to build mobile applications. Any developer who knows how to write React code can now target the Web, iOS and Android, all with the same skillset. By removing the need to “silo” developers based on their target platform, React Native lets your team iterate more quickly, and share knowledge and resources more effectively.

**Project Resources**

In this section we will provide detailed information on each of the resources of

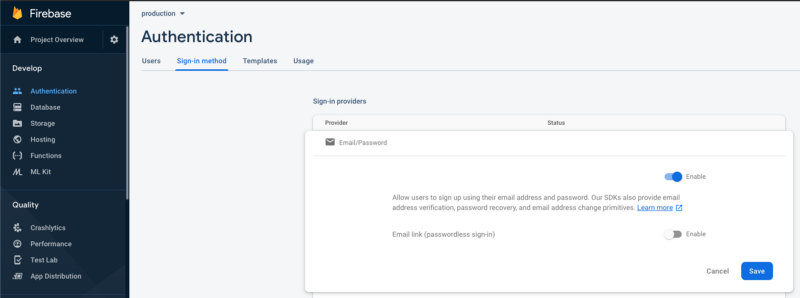
the application and how they function.

**Databases**

Firebase is a toolset from Google used to “build, improve, and grow your app”, and the tools gives you cover a large portion of the services that developers would normally have to build themselves, but don’t really want to build, because they’d rather be focusing on the experience itself. This includes things like analytics, authentication, databases, configuration, file storage, push messaging and the list goes on. The services are hosted in the cloud, and scale with little to no effort on the part of the developer.

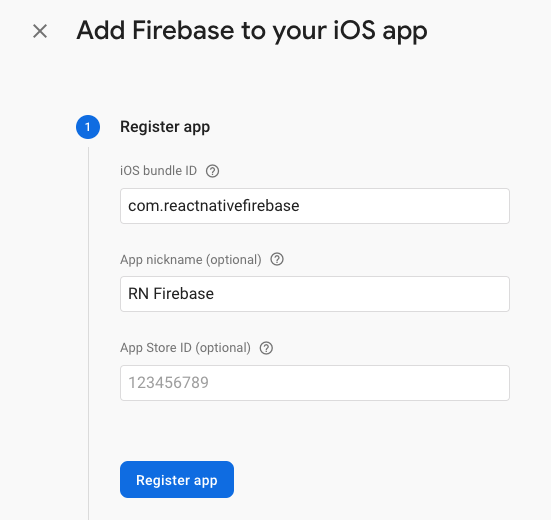
We have created a Firebase project which required a Google account at first place,

posteriorly we created a new project, exist different ways of authentication in the Firebase environment, the one used in this project, was “Email/Password” authentication, which allows users to sign up using their valid email and password, with SDKs providing email address verification, password recovery, and email address change primitives.

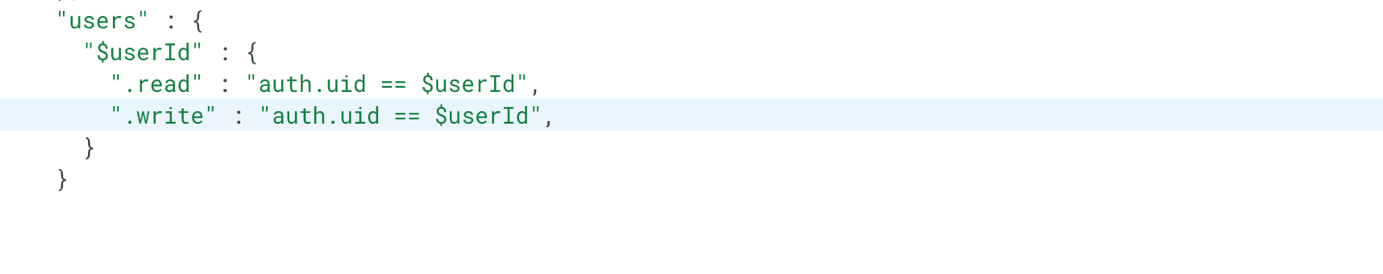


We need to enable the Email and Password verification.

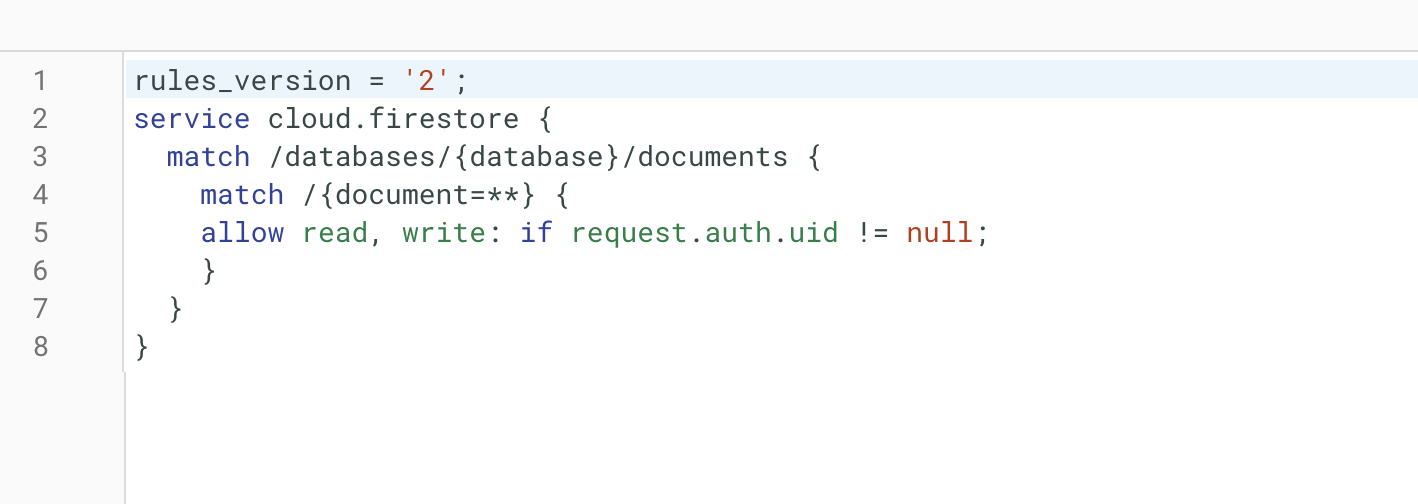
React Native works with native codes from both Platforms, for each of them we must register. Getting the ID’s both for Android and iOS we are successfully ready to set the database for usage.



As mentioned before, Firebase is NoSQL Database, the tables data has a different way of dealing, for Firebase we have `Rules`. In this session we are setting the rules for users, meaning that only users which has an account in our app is allowed to read and write his own data from Realtime Database.



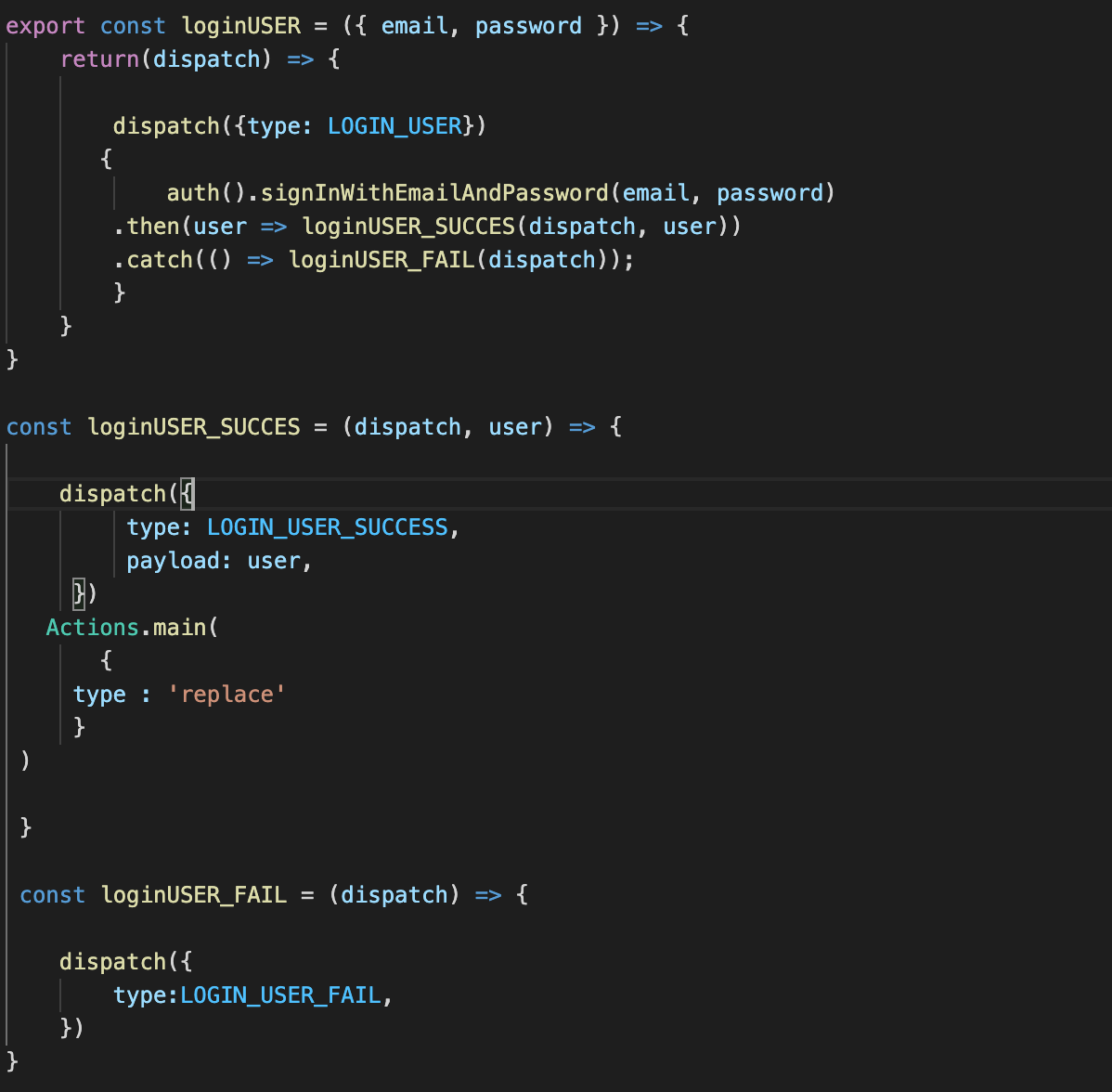
Since we are using two distinct Database for this project here, we also have an illustration of the same rules on Cloud Firebase. Meaning exactly the same thing from the Realtime Database.



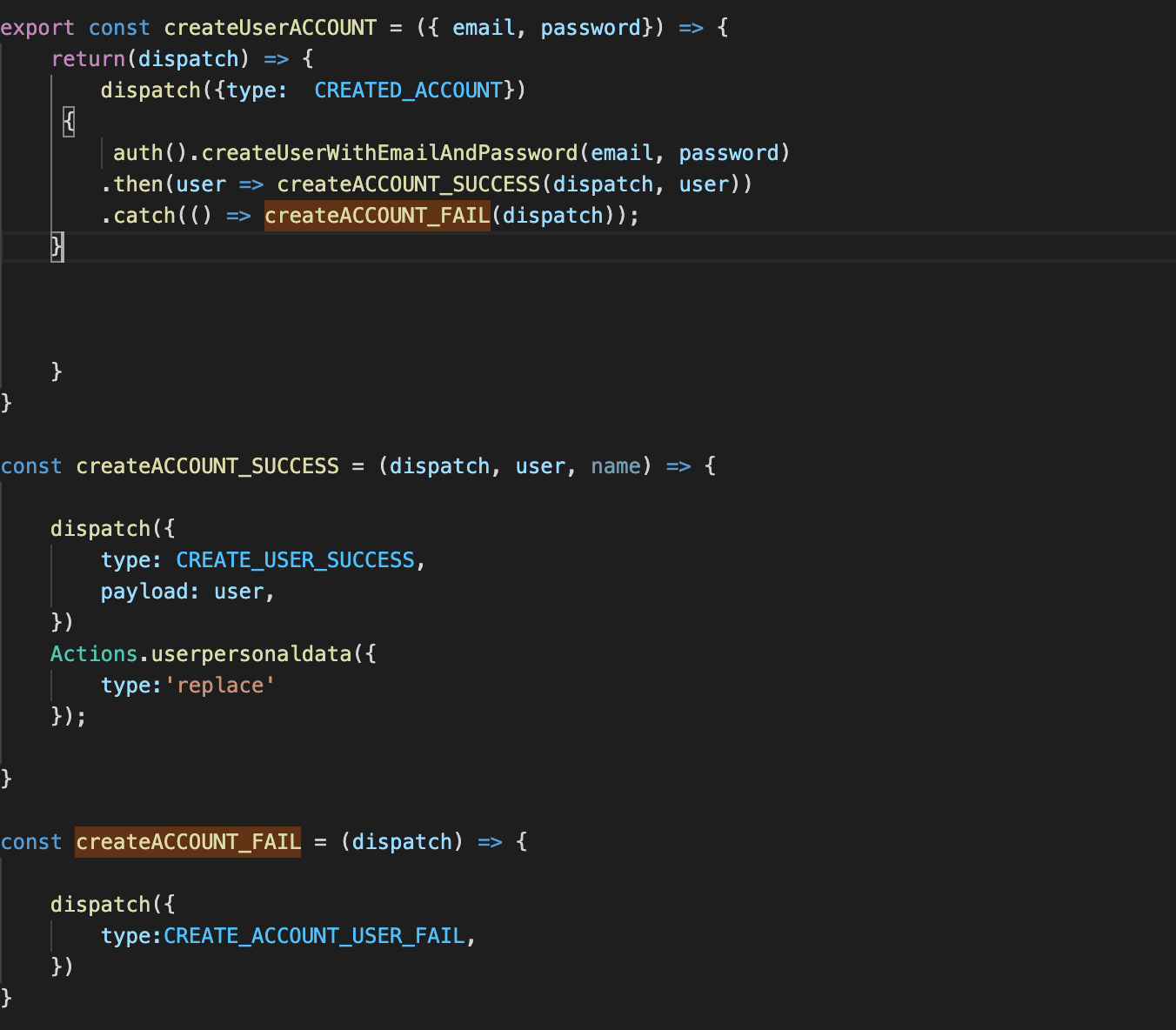
**Authentication**

For authentication session we used Redux state management, which is a predictable state container designed to help you write JavaScript apps that behave consistently across client, server, and native environments and are easy to test. While it’s mostly used as a state management tool with React, React and native any other JavaScript framework or library.

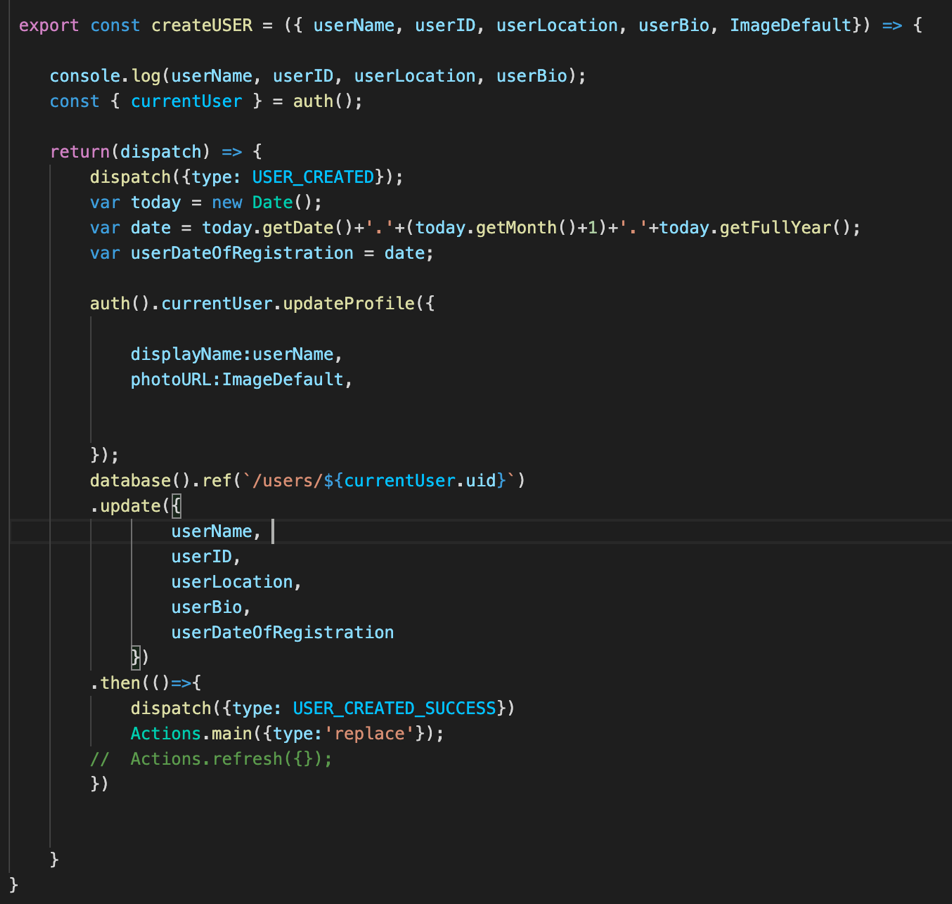
loginUSER({ email, password}) : This function requires two important information of user, their email and password. The signInWithEmailAndPasword is sending the data to the Firebase Database API checking if there already exist an account with the same email. In case of successful checking a new function is called loginUSER\_SUCCESS which posteriorly calls we move to the main screen by the Actions function, in case of unsuccessful login the loginUSER\_FAIL is called and updates the `error` state which will be displayed an error message on the screen of user.



For creating account, we followed the same logic as the login session, firstly we check if there is a valid data, and posteriorly we moved to the next screen. Here can be found below the figure.



From createUserACCOUNT function user will not be directional to the main screen yet, the Actions function will take them to the “UserPersonalDataScreen.js”, which their users will be allowed to provide more information besides of email and password data, they can set: userName, userID, userLocation, userBio as shown below:



We set the property `ImageDefault` to prevent an error from the `Image` view which does not allow null or undefined URLs, so we created a folder to hold all our default images for different categories.

**Updating user personal information**

In this session we provided an update function, which allows user to update their personal information incase if they want, **updateUSER** is taking four parameters (` userName, userID, userLocation and userBio`), each of this data can be updated separately, and posteriorly they will be sent to the database using the function `database().ref(…).then()`, which takes an reference to where data must be updated and pop the current screen after successfully updating.



**Forget email or Password**

In case user forget their email or password, there is a possible to recover it, here we used the function `**forgotPASSWORD**` which takes only email as a parameter and calls the `**sendPasswordResetEmail**` function for posteriorly send the link of the reset password page provide from Firebase system. Also, we catch some errors in case the email is not in the valid type or does not even exist in our database.

For the last step we used `emptyALL\_FIELDS` to clean up the fields after an action, we want to ensure that the same data used to login will not be the same on sign up screen after a navigation.

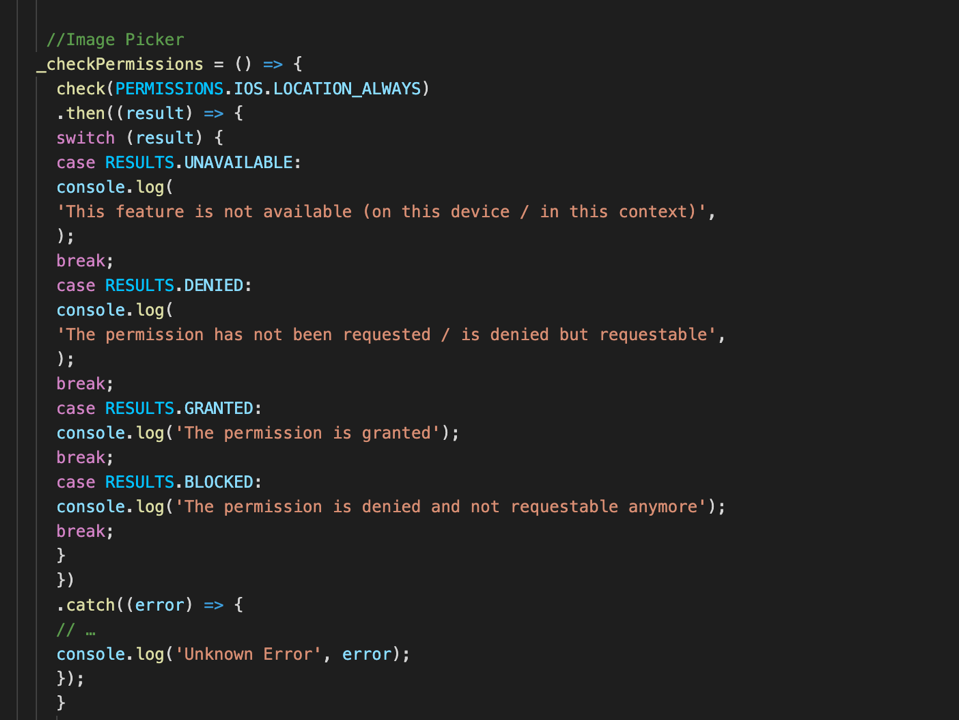


**Creating groups**

The create group screen is taking important states, which will help us knowing the real state of our progress as shown in the figure below :



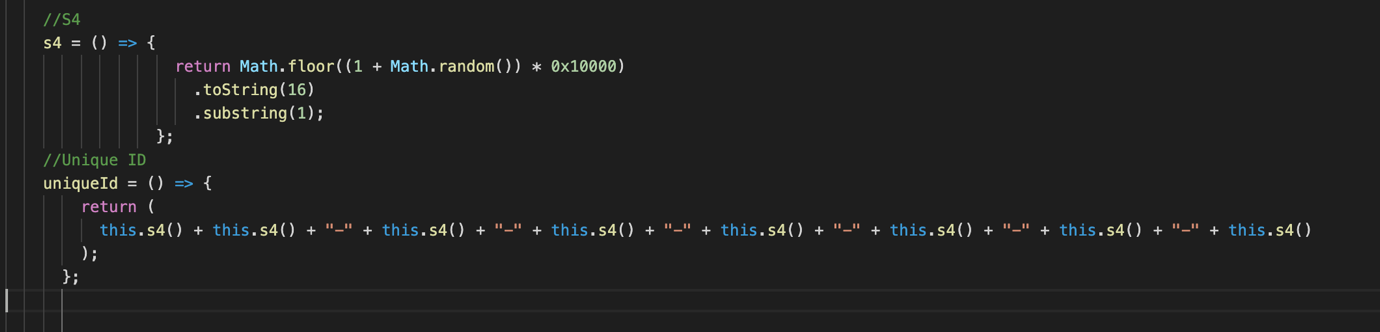
Groups are composed by: name, images and categories. We defined a default image for groups which is fetched from our default image folder located in our database, the categories state receives `Travel` as the default value. To be able to access images on devices we use the helper function called `**\_checkPermition**` which ask user permission to access the images from their devices, this function can return different states such as `DENIED` which means user denied the permission but a future request can be made, `GRANTEND` meaning user is agree and allowed the device to have access to the gallery, `BLOCK` which blocks the device to access the gallery and no future request can be made and `UNAVAILABLE` which is present when the version of the device is low and does not support the actual feature.



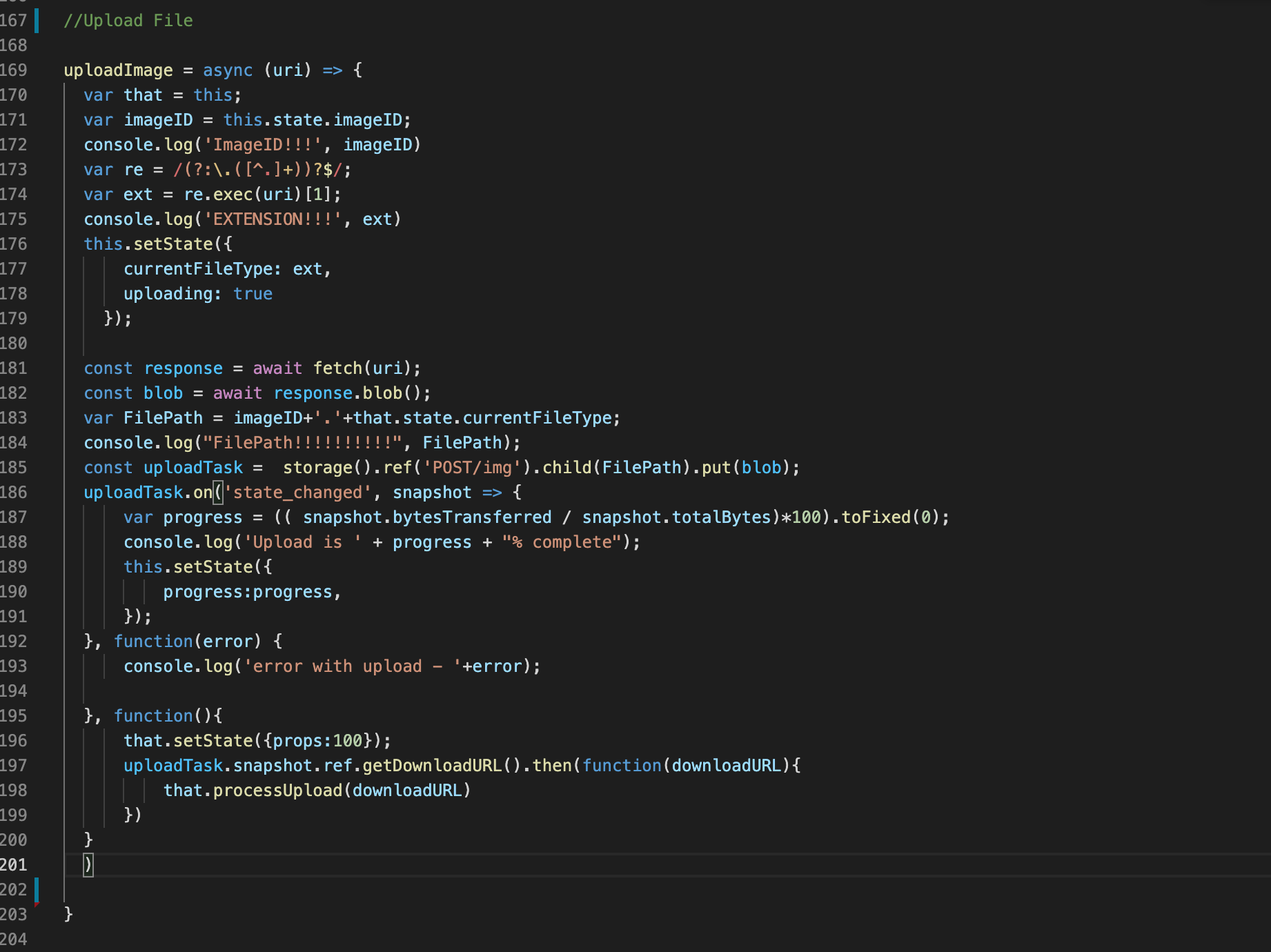
After successful request from user, we need a helper function `**findNewImage**`, this function will allow us to navigate through the device gallery and select the desired Image, also, we will have option to cancel the action, canceling we will be derationed to the priviest screen, a pop action will be called.



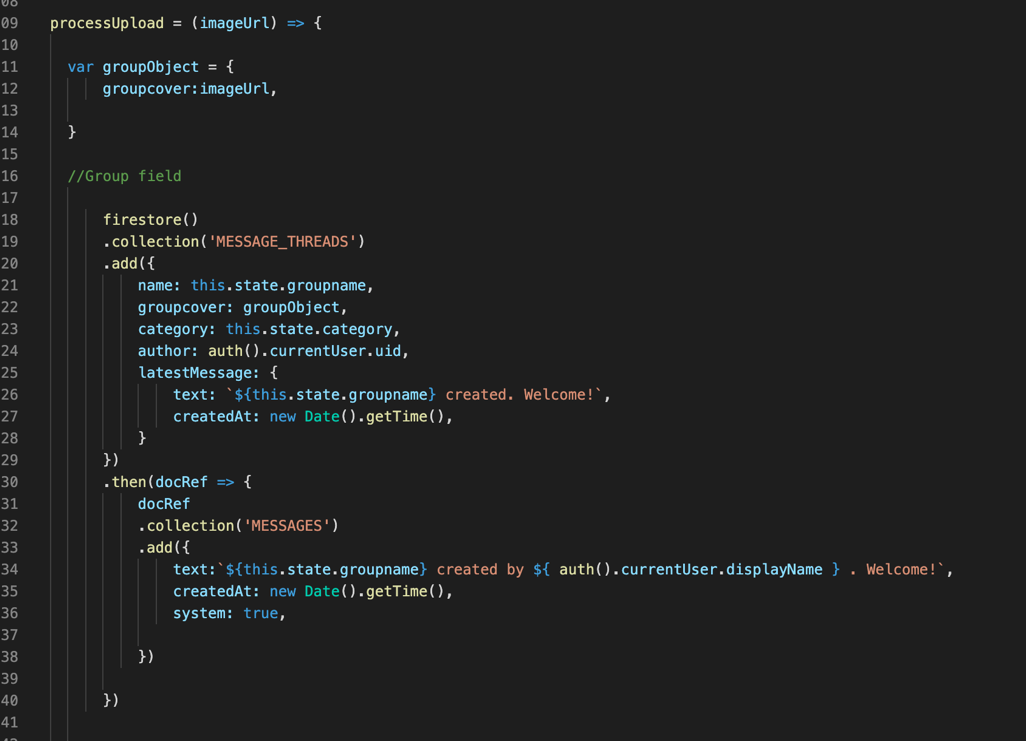
After selecting the images, to give unique ids for images we use two important functions the `S4` and `uniqueId`. The function `S4` is generating random numbers which posteriorly will be used on the `uniqueId` function, by this we are preventing two or more images containing the same Ids which would cause a conflict on the database point of view.



After this point, we upload the image on our Firebase Storage, given a reference folder as `**POST/img**`. After successful upload we download the URL from our database `**downloadURL**`, with this URL we pass to the `**processUpload**` function.



From `**processUpload**` function we pass the final properties that will be fetched to the database, such as : name, groupcover, category, author and lastestMessages. Remind that all this information are required to create a group, unless they are provide by default.

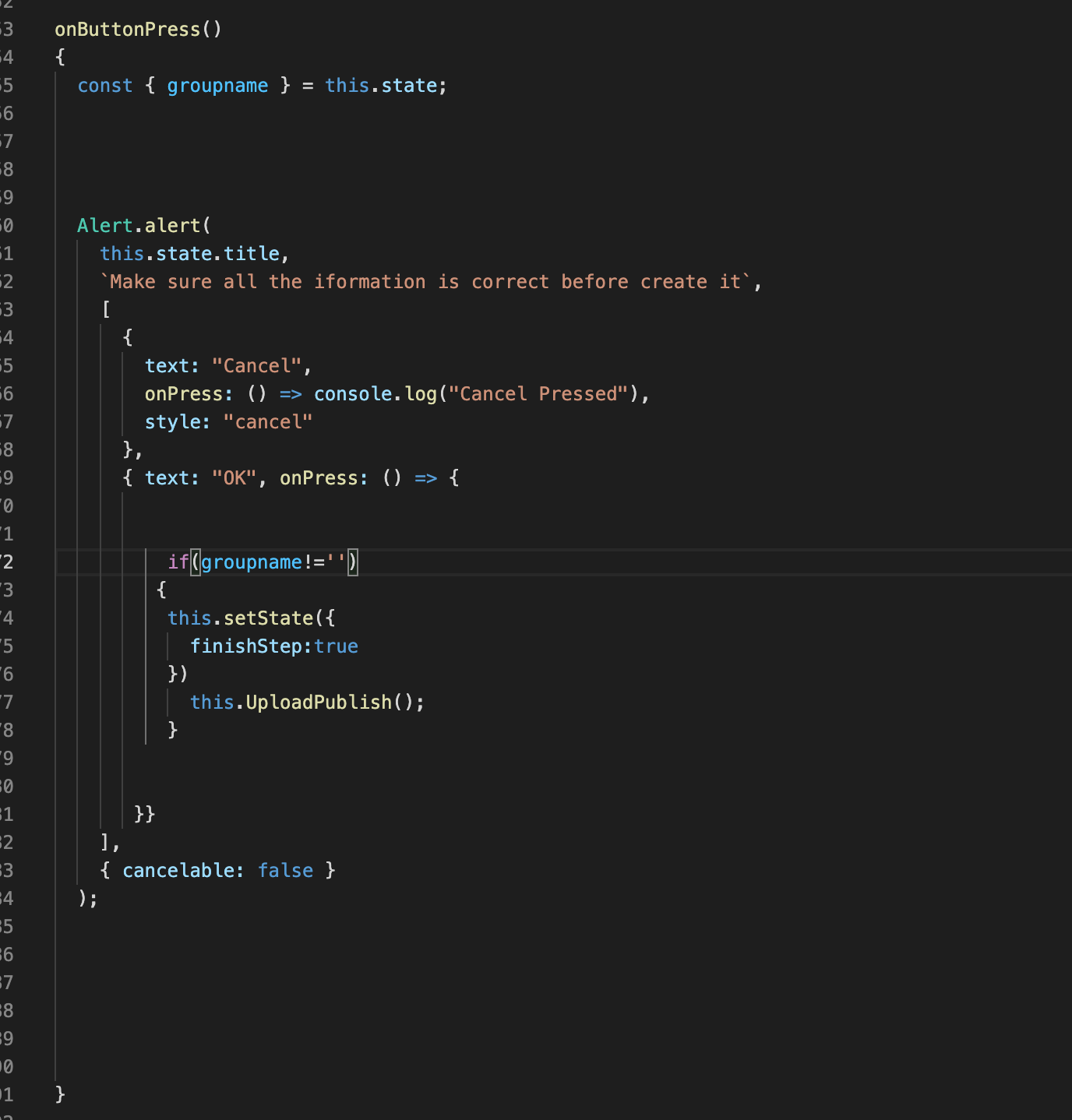


`UploadPublish` is helper method which has inside the `uploadImage` function, so, this will be called later on `onButtonPress` method.



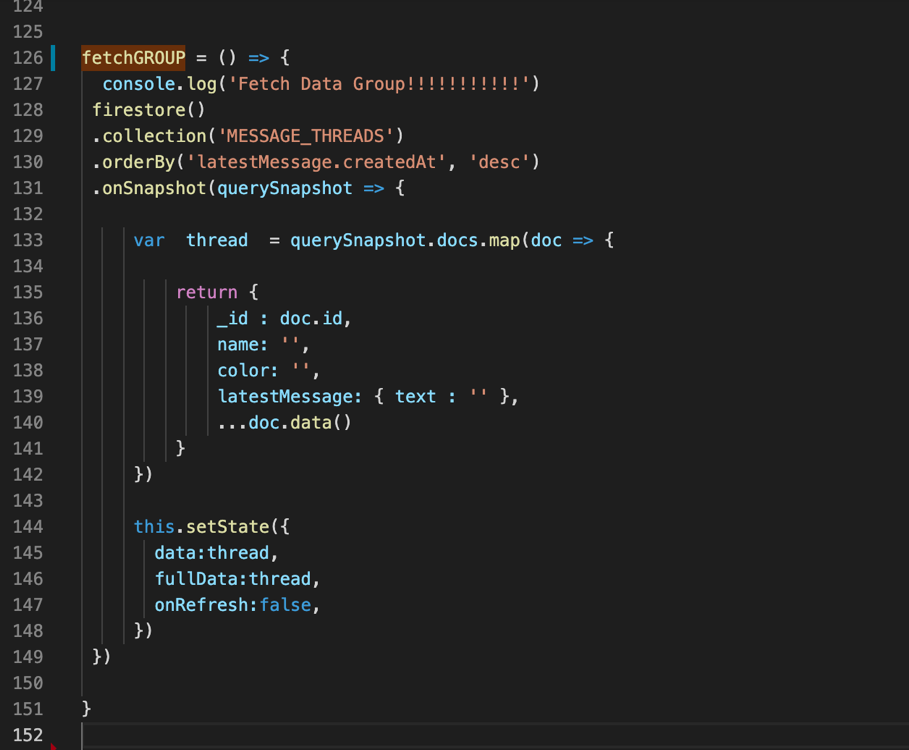
From `onButtonPress` we used an Alert function; this is to let user know if the provided data is the one

That they want to upload, because we don’t offer a possible edition after the group be created. Alert takes two important options “Cancel” and “OK”. Clicking on `Cancel` we assume that we do not want to publish yet an information may be incorrect or we do not want to create the group anymore. By clicking `OK` we say we are surely about the data that we provided and we would like to create the group. So, the group will be stored on the database and can be deleted on the future.



**Fetching groups**

For obtain the groups we that we have created, we use the function `fetchGROUP`, divis into our database in the collection named `MESSAGE\_THREADS` where we store our group information, `latestMessage` is a property which allows us to show the latest message from each group, users can read the last message sent to the group without actually open it.

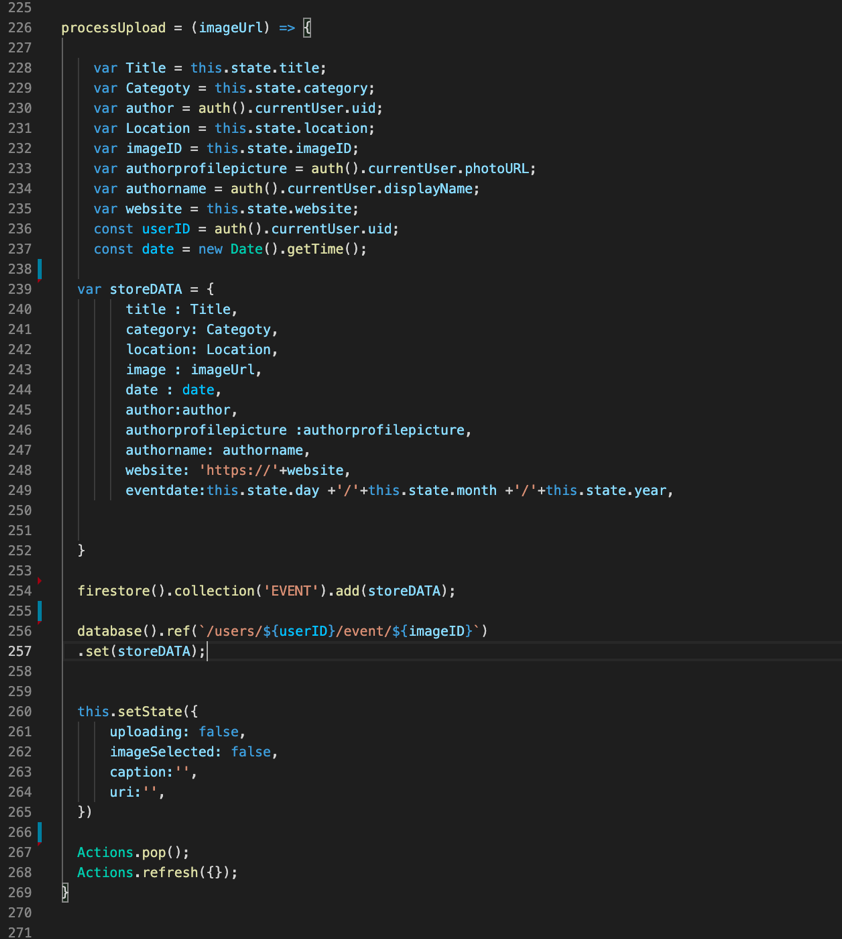


**Creating Post and Events**

The process of creating Posts and Events is really similar to the creating group section, here we will show the distinct functions that each of them possess. They all have the similar structure only the `processUpload` functions properties changes. below is shown the function of `Post` session.



Here we asked important information such as caption which is the description, the author’s name and the URL of the post.



For Event session we take important information such as event title, category, image URL, event date and if possibly a website link.

**Video call**

Nowadays video call apps have become a well-integrated habit in our lives. Here we show how incredibly fast and straightforward it can be to integrate video call functionality using React Native and Vonage Video API SDK, which is a WebRTC based solution. It supports many platforms, among these are browsers, Android, iOS, Windows and macOS. We use the API to create broadcast video sessions between our different roles.



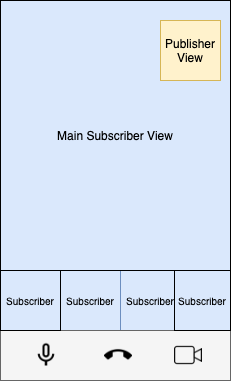
The `**JoinCall**` property on React state will trigger different views based on the value. When the App is lunched, a simple View with a “Join the Call” button will be displayed to the user. The button will trigger a state change and toggle the `**joinCall**` value to show the more complex videocall view.

**Videocall View**

Before digging into videocall View, it’s worth spending time exploring the `**opentok-react-native**` library.

The library is composed of the three main components: `**OTSession**`, `**OTPublisher**` and `**OTSubscriber** `. Each of them will interact with the native layer (iOS and Android), calling the native methods to connect, publish and subscribe. We will also need to listen to the events fired by those components, especially the session events, such as `**sessionConnected**`, `**sessionDisconnected**`, `**streamCreated**` and `**streamDestroyed**`.

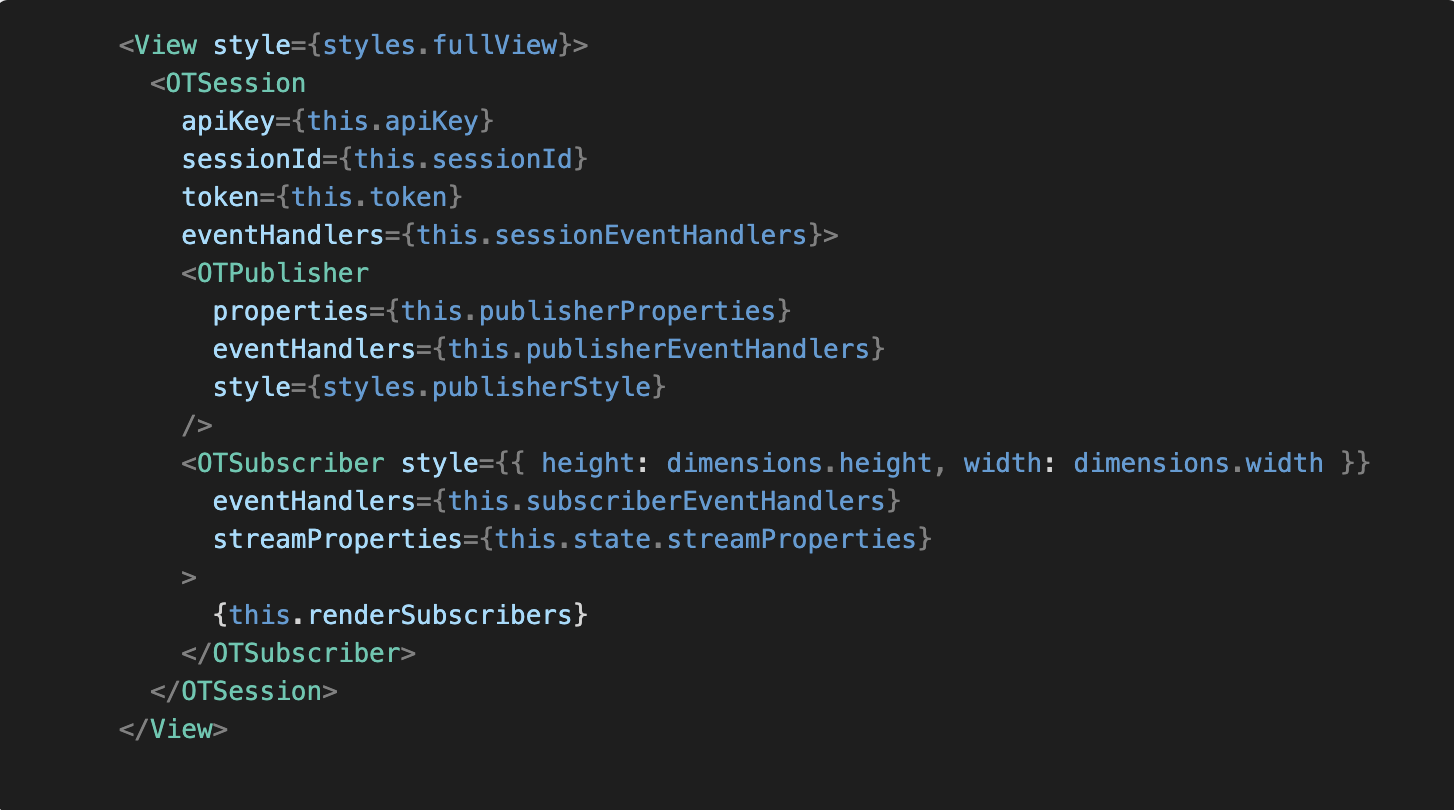
The videocall View is composed of the following components: Publisher, Subscribers and a ToolBar on the bottom side to control the microphone and camera and to end the call.



To build the above View, we need to keep track of the subscriber’s streams, the primary subscriber, and the local microphone and camera publishing state. The perfect place to store this information is the React State.



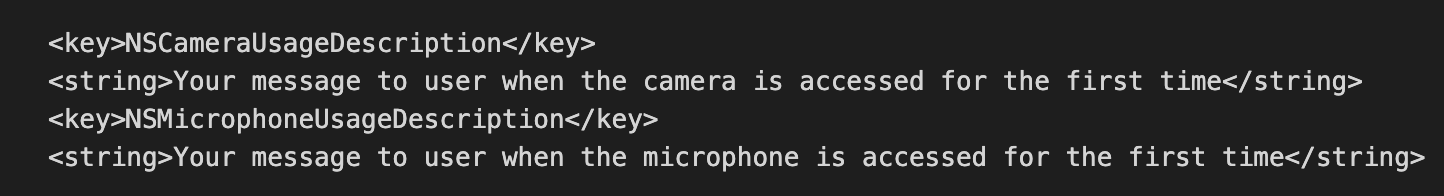
`**SubscriberIds**` array stores the subscribers within a session. Each time we receive a **streamCreated** event, it means that someone has joined the session and published a stream, so we need to add their **streamId** on the `**subscriberIds**` array. On the other hand, when we received the `**streamDestroyed**` event, we need to remove the **streamId** from the subscribers’ array.



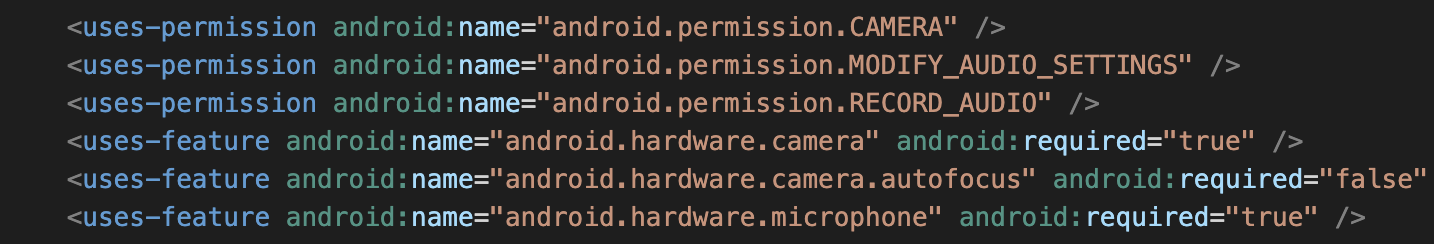
On the video view render function, we need to add `**OTSession**`, `**OTPublisher**` AND `**OTSubscriber**` from `opentok-react-native` libray. On the `**OTSession**` we set the credentials and the eventHandler function as props of the component.

The `**OTPublisher**` component will initialize a publisher and publish to the specified session upon mounting. It’s possible to specify different properties, such as camera position, resolution, and etc. In this session we will only set `**this.publisherProperties = { cameraPosition: ‘front’};**`

Ensure you have enabled both camera and microphone usage by adding the following enteres entries to your `**Info.plist**` file (iOS project):



Alternatively, for Android 6.0 add the following (newer version of `**API Level 23**`) :



The `**OTPublisher**` component has a `**streamProperty**` property which handles the publisher properties passed into the native instance. Using the React State, we can trigger changes to the Publisher instance by updating the `**this.publisherProperties**` variavle. We use this approach to implement the Toolbar with the mute/unmute functions for the Microphone and Camera. The function implementation is straightforward; it toggles the publishAudio or publishVideo value on the `**this.publisherProperties**` and the `**localPublishAudio**` and `**localPublishVideo**` to adjust the button icon based on the value.

The End Call button has a very similar approach. The `**endCall**` function toggles the `**joinCall**` value in the State and resets the View to the initial one.



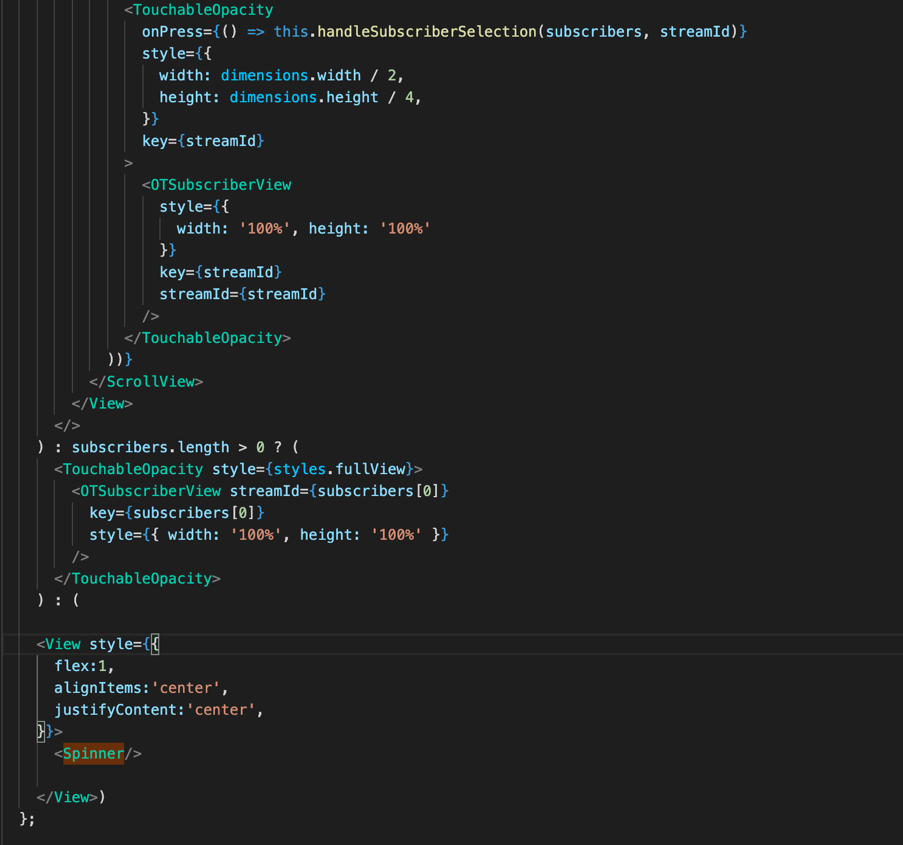


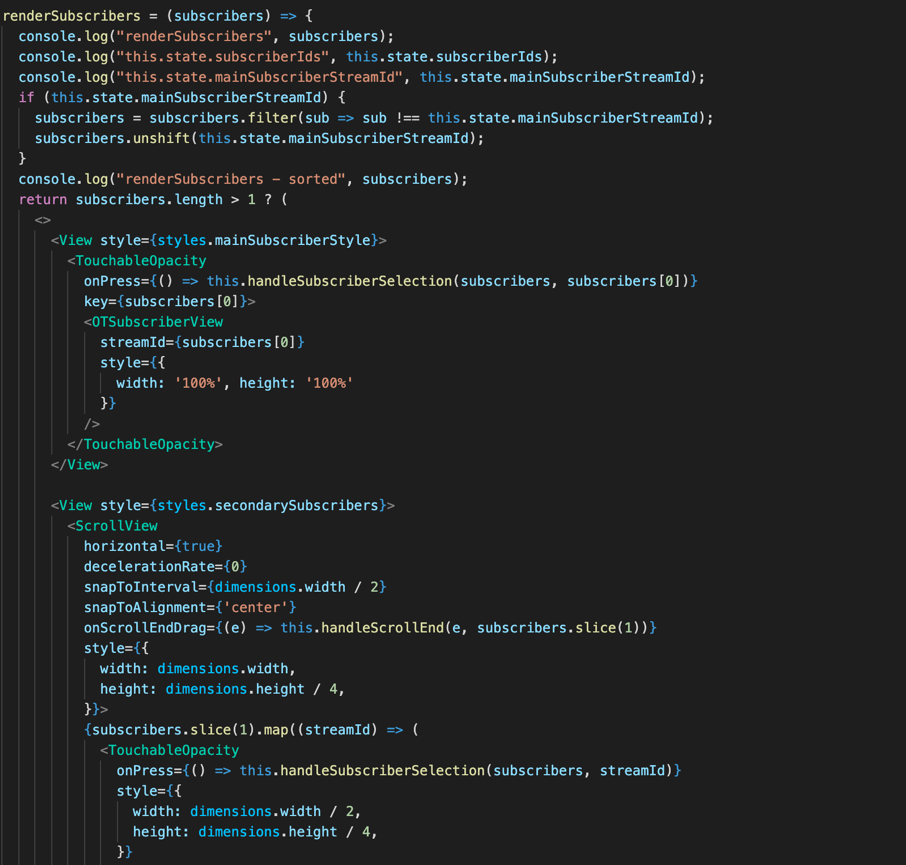
At this point we have implemented the Join Call View, the session and Publisher component, and the Toolbar. Next, we will define the View for the different possible number of subscribers. If we have no users, we are going to display a Loading spinner.

If we have only one subscriber, we will display their stream in full-screen mode.

Finally, if we have more than one user, we will show the primary subscriber in the big View (as shown in the mock-up), and the other in a Scroll View component to handle a different number of subscribers. As number could grow and challenge our device CPU and network bandwidth, we will implement optimization on each of the subscribers, such as lowering the resolution and disabling the video for the subscribers that are not visible.

Let’s explore the `**OTSubscriber**` component to handle the cases described above. First of all, as we want have control over each subscriber, we would need to implement a render for the subscribers.





We used the conditional rendering in React to handle the different cases with zero, one or N subscribers.

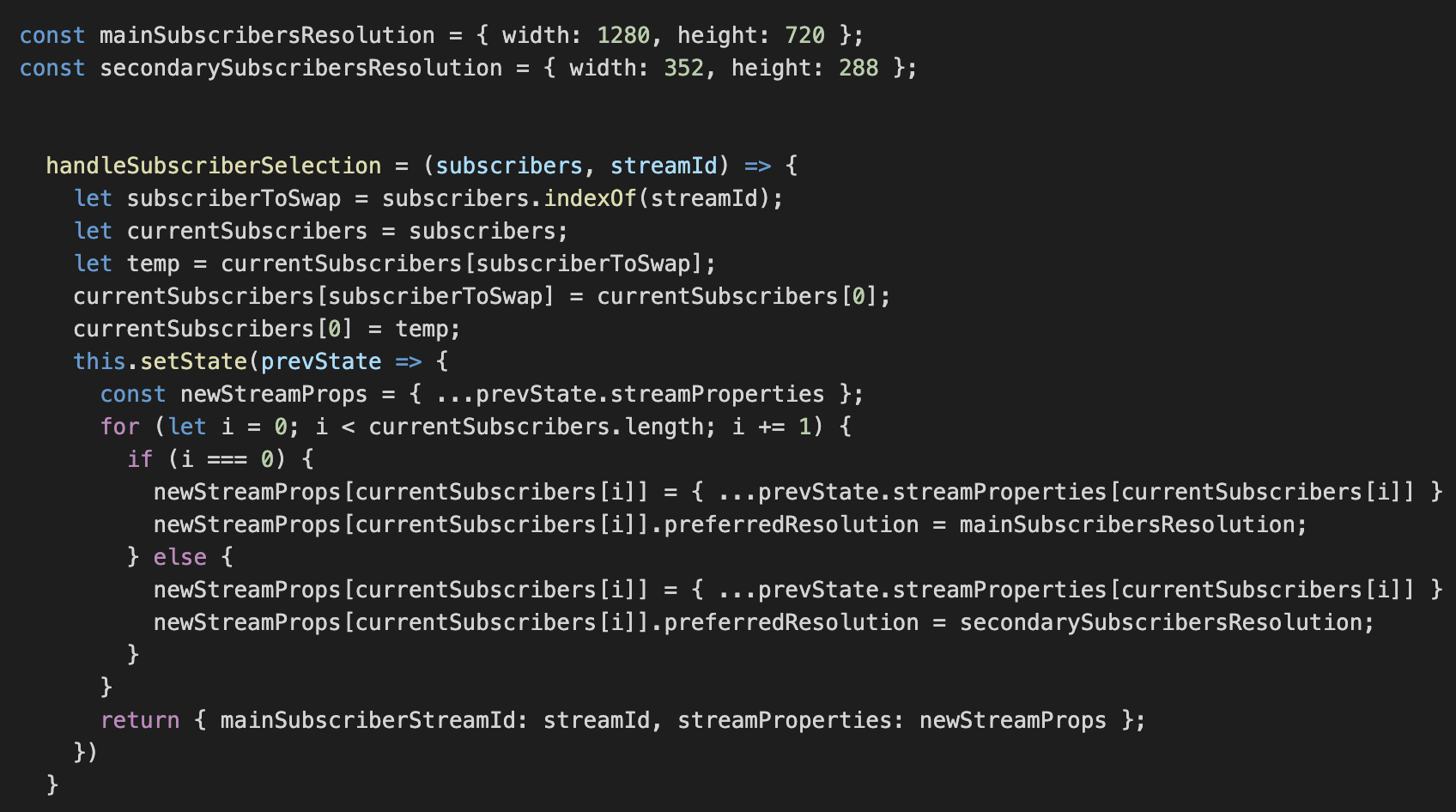
Firstly, if there not subscribers, we fall into last case, and we display a `**Spinner animation**` component.

Secondly, if there is one subscriber, we display the subscriber in a full view mode.

Lastly, the most interesting case is when the subscribers are more than one: We have a main subscriber view and a `**ScrollView**` component ion which we will feed the other subscribers. The first step is to check if we a `**mainSubscriberStreamId**`. If so, we will sort the array to have the primary subscriber as the first element. The remaining subscribers will be displayed in the `**ScrollView**` horizontal. The `**ScrollView**` component is deal for our use case, as we can show a relatively high number of subscribers without the need to change the layout, and we can detect how many subscribers are in the scroll view and how many of them are visible.

Group calls on mobile devices could be very challenging, both from the hardware and networking point of view. To deliver a good result to the end-user, an App should implement a list of best practices to handle different use case and layout. In this case, we have a main subscriber view which needs to have the best resolution possible, and the Scroll View component with the remaining subscribers in smaller thumbnails that could be optimized by lowering the received resolution. OpenTok SDKs give the developer the opportunity to set the preferred resolution and frame rate for each of the subscriber.

We implement the `**handleSubscriberSelection**` method to handle the **mainSubscriber** View and the preferred resolution. The function is on the `**TouchableOpacity**` component parent of each of the subscribers.



Based on the subscriber selected, the function moves the selected subscriber to the head of the subscribers’ array. As mentioned before, the first element on the subscriber array will be displayed in the main View. After that, we need to update the `**streamProperties**` of the `**OTSubscriber**` component to set the different preferred resolution. We set the maximum resolution (`**width**: 1280, **height**:720`) for the primary subscriber and a lower resolution for the other (` {**width**: 352, **height**: 288} `). If we also want to change the preferred frame rate, based on the layout or use case, we would only need to add the `**preferredFrameRate**` property on the `**streamProperties**` object.

Finally, we want to optimize the `**ScrollView**` component. The **ScrollView** component could have a high number of subscribers, but can only show two simultaneously. As an example, if we have five subscribers, one will be on the main subscriber view; the remaining four will be on the ScrollView. Only two of them are visible in the View, and the remaining ones will be visible only if we scroll horizontally.

The ScrollView component has an event listener called onScrollEndDrag, which is called when the user stops, dragging the scroll view and it either stops or begins to glide. We can use this event to understand which subscribers are visible and mute the video of the remaining ones. Muting the video of the non-visible stream will improve the performance of the App, and save CPU consumption and network bandwidth.



On the `onScrollEndDrag` event, we have the information about the contentOffset coordinates, which is the point at which the origin of the content view is offset from the origin of the scroll view. We will use this value to understand which streams are currently visible, diving the content offset by half of the width of the screen **(`event.nativeEvent.contentOffset.x / (dimensions.width / 2)`)**.

The result will be the first visible subscriber. At this point, we know that the visible streams are the stream in position `**firstVisibleIndex**` and `**firstVisibleIndex + 1**`. The last step is to loop the subscribers’ array and mute the video of the non-visible subscribers.

Chapter 4

CONCLUSION

In this project we achieved our goal of creating a user-friendly application and delivering the promised functionalities. The approaches chosen in the project are far from being the most straightforward solution for creating a mobile app; however, they provide a solid and scalable infrastructure and a structured development process needed for large scale complex applications.

React Native Firebase provided many useful services to make this project possible, and helped tremendously with the development process buy encapsulating much of the server configuration and allowing us to fucus on development of the application without incurring any costs. Since many of the used services were released not too long ago, they have not been widely adopted by the community, so there is relatively fewer articles and sources of information that can be found for them apart from the official documentation. Due to this, some of these tools - especially the less explored sections of them - have a steep learning curve, and it might take a while to get familiar with their definitions and methods. Because of the full-stack nature of this project, many core concepts in different fields of Computer Science had to be thoroughly explored to be able to have a clear vision of each used component’s behaviour, resulting in valuable personal learnings

on mobile development.