Campus Meet

Reynel Villabona
A_A

February 4, 2023

Contents

1 Introduction

In this section we will discuss the purpose of Campus Meet and the main features supporting this purpose.

1.1 Purpose

The purpose of Campus Meet is to connect students at university. As we can see from everyday experiences, organizing meetups with friends at university is troublesome, because every student has their unique schedule with different timeslots and different places. Especially as exchange students, with no previous friends and also completely unrelated curriculas, this app helps connecting with other students. Campus Meet allows to easily find people for lunch breaks, study sessions and other meetings with fellow students. At these meetings, new friendships can be created which can extend into other areas of the life.

1.2 Target audience

The target audience are students around the whole world. Of course, students will mostly be interested into meetups on their own campus, therefore the app supports to filter by location. As students use smartphones frequently it seems appropriate to create a mobile application for this audience. Furthermore, we assume that most students are familiar with new technologies, however we also design the user interface such that inexperienced users can intuitively navigate through the application.

1.3 Features

The following sections show which features are included in Campus Meet.

1.3.1 Sign in

Users can sign in by using their email address. They do not need to remember a password, as the login is persisted until they logout, even across device restarts. In case of changing the phone, they can easily sign in again with their email address.

1.3.2 Profiles

Users can create a profile about them, displaying relevant information such as name, study area and languages. They can edit their profile at any time. When viewing meetings or chats, they can click on the other users avatars to view their profiles.

1.3.3 Meetings

The app's meetings feature enables users to connect and network with new students, fostering new valuable personal connections. The user-friendly interface allows them to filter meetings by location, see meeting details and also modify self-created meetings. Meeting information includes a title, location, starting time, duration and optionally some notes about it. The user interface also highlight meetings which you are member of.

1.3.4 Chats

The application supports two kind of chats: meeting chats and private chats. Meeting chats created as soon as a new meeting is created and automatically include every participant of the meeting. Private chats can be created when viewing another profile and starting a new chat from there. Chats are stored locally to provide an offline experience and automatically synced with the server in realtime.

2 External services

To implement the features, we interact with following external services. As external service we view components outside of the mobile application, therefore also special systems services of the mobile phone are considered.

The purpose of using external services in mobile app development is to add functionality and capabilities to the app that would otherwise be difficult or time-consuming to implement from scratch.

We used them for storing and retrieving data , authentication, push notifications and adding map and location-based functionalities.

2.1 Services external to the phone

2.1.1 Firebase authentication

Firebase Authentication [Gooa] is use to easily add authentication functionality to the mobile and web apps. We used it to:

- Send an email link for signing in to the provided email address.
- Display a website that forwards the user to the mobile application.
- When starting the mobile application, check if the initial url is for signing in the user.
- Listen for deep link changes and sign in the user if it's a sign in url.

2.1.2 Firestore

Firestore [Gooc] is a NoSQL document-based database service provided by Firebase. It allows you to store and retrieve data in the form of documents, which are organized into collections. We used it to:

- Add new documents to the storage.
- Update existing documents either completly with all new information, or only partially by changing single fields.
- Atomically add or remove items from lists to prevent race conditions.
- Delete existing documents.
- Retrieve the data of a document and also listen for real-time changes to the data of a document or collection as trigger for Firebase Functions.

- Query and filter documents and collections according to the app requirements.

2.1.3 Firebase Functions

Firebase Functions [Goob] are used to react to changes in the Firestore database. They allow to subscribe to certain events and then execute a custom function. In particular, this was used to:

- send push notifications to all recipients when a new message is stored
- send push notification to all members when a user joined a meeting
- automatically create a chat when a meeting is created
- update chat members and metadata when someone joins or leaves a meeting
- update chat metadata when a profile is updated

2.1.4 Google Maps

Google Maps is a web mapping service [Good]. They allow to develop location search feature, allowing users to search for meetings, as well as the ability to save and share locations. We used it to:

- Display a map to select the location while creating a meeting.
- Display the location of meetings on a map including a circle showing the currently selected search radius.
- Get a human-readable address from the coordinates, so it is better readable for users.

2.1.5 Expo push notifications

Expo push notifications [Exp] provide a simple and easy way to send push notifications via a REST API. One can use this API to send notifications to specific users or groups of users, or to send broadcast notifications to all users. In the development of this app we used it to:

- Retrieve a unique push notification token per user device.
- Send notifications to specific users or groups of users, based on the unique token (token) associated with user's devices.
- Customize the appearance of notifications: the title, message, and icon that appear in a notification.

2.2 Services within the phone

The purpose of using services within the phone when developing mobile apps is to provide users with a more seamless, efficient, and engaging experience by leveraging the device's hardware and software capabilities.

2.2.1 Current location

After granting the permissions, the phone can provide apps with information about the current position of the device. We used this to:

- Simplify the selection of a meeting location by starting at the users current location.
- Develop a Location-based search: Using user's location we can provide search results that are within a specific distance to the user.

2.2.2 File system

The file system is used to store, retrieve, and manage files on a device. We used it to:

- Select and load images from the file system to allow users to upload profile pictures from their phone.

2.2.3 Camera

The camera is used to take pictures, we used this to:

- allow the user to take a photo of themself (selfie) and set it up as their profile picture.

2.2.4 Sharing

Mobile devices provide the ability to share content from one application to other channels. We used this to:

- allow to share meeting details with friends via other channels present in the phone, such as WhatsApp, Facebook, Email and more.

2.2.5 Geo url scheme

Mobile devices have special url schemes, which allow applications to register and listen to them. We used the geo scheme to:

- Open a meetings location in the configured map application of the devices, for instance Google Maps, to easily let users navigate to this position.

3 Software architecture

This section discusses the software architecture of Campus Meet. First, we will give a high-level overview of the different actors interacting with each other. Then we will give a more detailed view on the structure of the mobile application.

3.1 Overview

In our app mobile development we implemented the client-server process. For this we will shown how the S2B layer are distributed. The diagram 1 illustrates this scheme.

- **Presentation layer:** In our case it is the mobile application where the user interacts.
- **Application layer:** The application layer is located inside the mobile application as well as Firebase Auth, Firebase Functions and Expo Notifications.
- **Data layer:** Firestore is used to store and keep our data in a structured way.

3.1.1 Interactions between the layers

The mobile application and Firebase Authentication interact by exchanging information related to user authentication, session management, access control, and user data management. By using Firebase Authentication, the mobile application can leverage the authentication and authorization features provided by Firebase, allowing the mobile application to focus on the business logic related to the application's functionality.

The Firestore SDK provides an API for accessing Cloud Firestore from the mobile application and supports creating, updating, complex querying and deleting documents in a structured manner.

When a change occurs in the Firestore database, such as adding a new document or updating an existing one, a Firebase Cloud Function can be triggered to perform additional processing on the data. This allows for the creation of event-driven systems where changes to the data stored in Cloud Firestore trigger backend processing.

As finally interaction of the components, when a Firebase Cloud Function is triggered, it can use the Expo Push Notifications API to send a push notification to the mobile application. The Expo Push Notifications API requires the device token provided by the mobile application to send the push notification to the specific device.

3.2 Mobile application design

As part of the overall design of the mobile application we decided to divide it into the following components. The interaction between these components is shown in diagram 2.

- Screens and components are part of the visual design of the application.

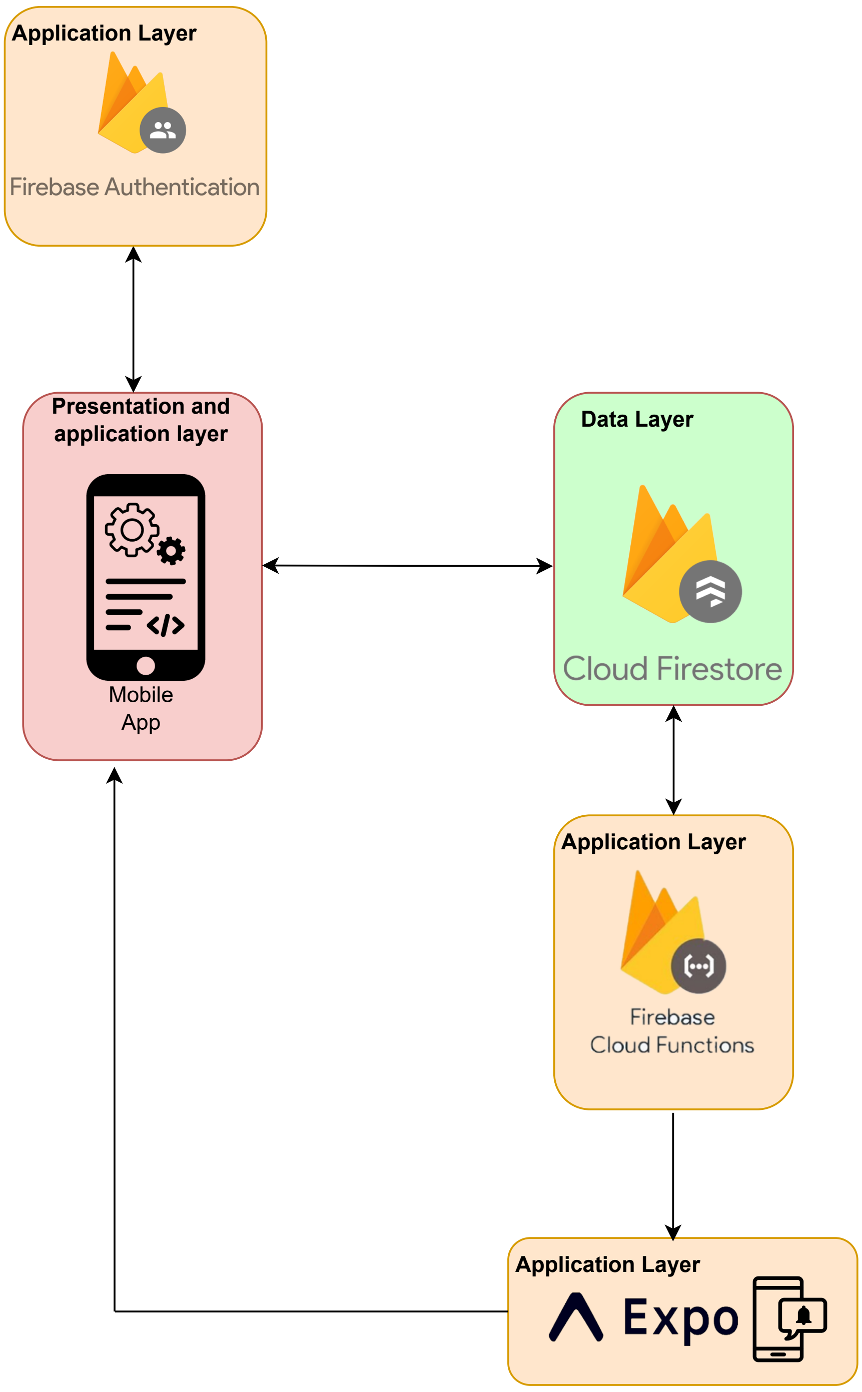


Figure 1: Interaction of different components and their respective layers

- Hooks, Context, and local databases are part of the functional design of the application and help manage state, data, and performance.
- APIs are part of the technical design of the application and help connect the application to external data and services.

The contexts provide a way to pass data and functions down the component tree without having to pass props down manually at every level. Contexts were used to provide access to the local database, to incoming push notifications as well as to synchronization mechanisms for the chat.

Hooks and contexts work together to provide a way to share data and manage state and side-effects in a flexible and modular way. The hooks take the data from the contexts and provide them in a simple interface.

Components here are using hooks to access state and life cycle methods, and are indirectly using

Contexts to access data shared across multiple components. Components are also responsible to trigger changes to contexts, such as initiating a synchronization with the server after sending a new message.

Screen components use APIs for multiple purposes: they fetch data to display them on the screen. They use them to transform raw data such as coordinates to human-readable data such as an address. And finally, they also use APIs to store and update data from entered by the users.

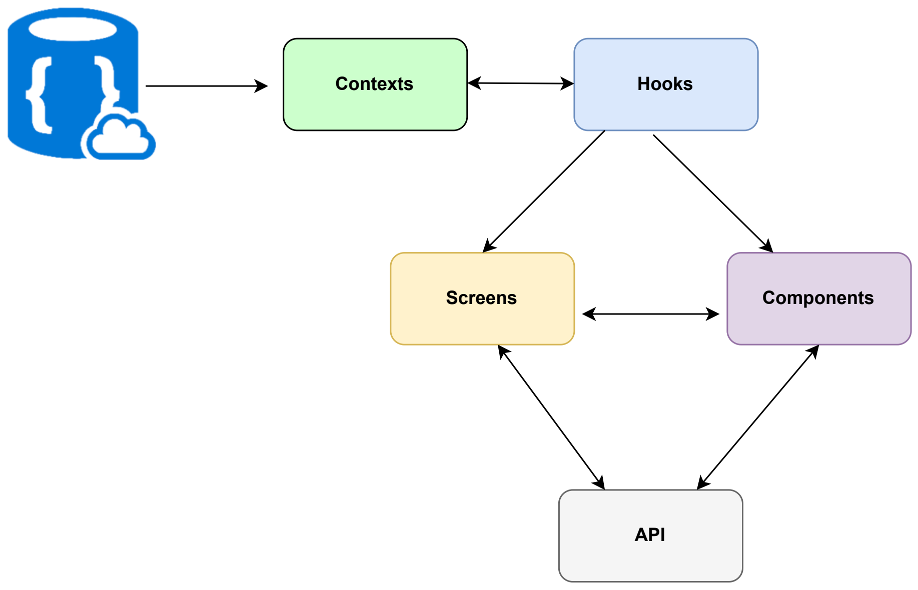


Figure 2: Interaction of different components within the mobile application

3.2.1 Components

Components in mobile apps refer to the different parts or building blocks that make up the app. These components are responsible for different aspects of the app's functionality. In our application we have some components that will be mentioned below:

- **VerifyProfileScreen**: This component acts as a guard that checks if a logged-in user already has created their profile. In case they have not it displays a screen to create the profile. In case the user already has a profile, it displays the "BottomTabNavigator" which shows the normal application.
- **CapitalizedText**: The "CapitalizedText" component applies a style to the text prop of the wrapped "Text" component, that capitalizes the first letter of each word in the text using the textTransform property.
- **Time**: The component receives a datetime object and it's passed to the toFormat method of luxon library to parse it and return the time in the format of "HH:mm".
- **RelativeDateOrTime**: The function uses a ternary operator to check if the passed in "datetime" value is today's date using a helper function "isToday" imported from "../../utils/time/datetime". If it is today's date, the function renders a component called "Time" with the passed in "datetime" value as a prop.
- **RelativeDateAndTime**: The "RelativeDate" and "Time" components are also being passed the "datetime" prop.The overall purpose of this component is to display a relative date and time based on the provided datetime.
- **ListWithDateDivider**: The "RelativeDate" and "Time" components are also being passed the "datetime" prop. The overall purpose of this component is to take a list of elements and display a formatted date divider showing to which day the different elements belong to.
- **PaddedView**: It renders a View component with a top and horizontal padding of 15 and a flex value of 1. The children passed to this component will be rendered inside this View.
- **ProfileDetails**: The component displays the profile image, name, studies, languages, and about information of the user profile.It also uses the FAB component from the react-native-

paper library to display a "Send Message" button for non-logged-in users, and an action menu for the logged-in user that allows them to log out or edit their profile.

- **NotificationReceiver**: This is a functional component that is used to handle push notifications. The component uses the useContext hook to get access to the PushNotificationContext context, which holds information about push notifications. When the user tapped on a push notification this component will forward them to the appropriate screens, e.g. the chat of the message notification which was pressed.
- **BottomTabNavigator**: The code defines a BottomTab component using the createBottomTabNavigator function, which creates a tab navigator that switches between the three tabs: Discover Meetings, Chats and Profile.
- **LocationSelector**: It allows the user to select a location on the map by clicking on it. The selected location is then passed to a callback function "onLocationChange" that can be used to update the parent component state. A marker displays the currently selected location.
- **ConfirmDialog**: The component takes in props such as title, message, visible, onConfirm and onCancel which are used to customize the dialog and handle user interactions. The component uses the Portal component to render the Dialog component as a modal.
- **MessagesContainer**: This component is the container for chat messages. It displays a chat background image which depends on the currently selected mode (dark or light mode). It wraps its children in a Scrollview to allow scrolling and ensures that when opening the component it is rendered at the bottom of the scrollview.
- **Messages**: This component renders a list of messages. The component renders each message as an item in a List component from react-native-paper. Each message item includes the sender's name, the message text, and a timestamp. Messages are styled as bubbles and are formatted on either the left or right side, depending on whether they are sent by the currently logged-in user or someone else.
- **MessageInput**: This is a functional component that is used as a message input field. It uses the useState hook from to keep track of the message input. When send is pressed, it will pass the message to the parent component via the provided callback.
- **ChatPreviews**: This component takes in a list of chat previews and displays them to the user. The component maps over the array of chat previews and creates a list item for each one, displaying information such as the chat title, last message, and unread message count.
- **Chat**: This is a functional component for a chat page in a mobile application. The component displays a list of messages, a message input field, and a title for the chat which, when clicked, navigates to either the profile of the other user in a private chat or to a meeting details page in a group chat.

3.2.2 API

The API (Application Programming Interface) of our application is used to let the mobile application communicate with the Firestore database. We divided the API in four parts according to the main functionalities of our App: Chats, Notification, Meetings and Profile.

Chats: The module makes use of the Firebase Firestore library to interact with the database. The exported functions include createPrivateChat, fetchOwnChats, getChatIdOfMeeting, getChat, markAsNewestReadMessage, which are used to create, fetch, and update chats. The module also exports an interface IChat, which defines the shape of a chat object.

Messages: The functions make use of the Firebase Firestore library, as well as some utility functions for authentication and getting the Firestore database instance.

The main functionality includes:

- **sendMessage:** which takes in a chatId and message, and sends the message to the Firestore database, by adding a new document to the "messages" subcollection of the chat document. It also calls a utility function "markAsNewestReadMessage" to update the read message date in the chat document.

- **fetchMessagesNewerThan:** which takes in a chatId and date, and fetches all messages that have a date newer than the provided date, from the Firestore database. The function returns an array of IMessage objects.
- **getChatsCollection:** which returns the ”chats” collection from Firestore
- **getMessagesCollection:** which takes in chatId and returns the ”messages” subcollection of the chat document
- **mapDataToMessage:** which takes in an id, chatId, and data and returns an IMessage object.

Meetings : This is a module containing various functions for interacting with a Firestore database, specifically for creating, fetching, updating, and deleting ”meeting” documents. The module exports several interfaces and types such as IMeetingCreationDto, IMeeting and a function for each CRUD operation (createMeeting, fetchCurrentMeetings, fetchMeeting, deleteMeeting, joinMeeting, leaveMeeting, updateMeeting) and utility functions (getMeetingsCollection, getMeetingDoc, mapDataToMeeting) that are used by these CRUD functions.It also includes methods for querying and updating arrays in the Firestore document.

Notifications : This code exports a single function ”savePushTokenToServer” that is used to save a push token to a Firebase Firestore database. The function imports two helper functions ”getCurrentUserId” and ”getDb” which are used to get the current user's ID and the Firestore database object respectively. The function then uses the ”collection” and ”doc” functions from the Firestore library to get the ”pushtokens” collection and the document for the current user's ID. Finally, the function uses the ”setDoc” function to set the document's ”token” field to the provided token.

profileAPI : This is a module containing a set of functions that interact with Firebase's Firestore and Storage services to create, read, update and delete user profiles. The functions use the Firestore SDK to perform CRUD operations on a ”profiles” collection, and the Storage SDK to handle image uploads and URLs. The functions also use helper functions from other files such as ”currentUser” and ”firestore” to handle user authentication and Firestore connections. The file exports several interfaces, such as IProfileCreationDto, IProfile, and IProfileMetadata, that define the structure of a user profile and its metadata.

3.2.3 Hooks

Hooks were used to manage the state and side-effects of a component. They allowed us to use state and other React features in functional components, rather than only in class components. This made the code easier to understand and manage, and also improved the performance by avoiding unnecessary re-renders.

We used hooks for chats, Authentication and getting user location.

Chats: Here we used different Hooks for managing messages, below we will explain each of them:

- **useStoredMessages:** This is a Hook that allows a component to access messages stored in a local database. By using the ChatsDbSyncedContext it will automatically update whenever a new message is stored. The hook takes in a chatId as an argument and returns an object with all messages from this chat.
- **useSortedChatPreviews:** This is a Hook that allows a component to access chat previews sorted by their last message's date from a local database. By using the ChatsDbSyncedContext it will automatically update whenever the chat metadata is modified or a new last message is received. The hook returns an object with a list of chat previews.
- **usePrivateChatId:** This is a Hook that allows a component to access the id of a private chat with a specific user from a local database. This allows to check whether a private chat with a user exists and, if yes, to get the corresponding chat id. By using the ChatsDbSyncedContext it will automatically update when a new private chat is created. The hook takes in a userId as an argument and returns an object with the chatId for the private chat with that user.

- **useChatMetadata:** This is a Hook that allows a component to fetch and use chat metadata from the local database. By using the ChatsDbSyncedContext it will automatically update when the chat metadata has updated. The hook takes a chatId argument, which is the id of the chat whose metadata is to be fetched. It returns an object with a single property chatMetadata that is the metadata of the specified chat.
- **useChatsSync:** This is a Hook that allows a component to trigger a synchronization of the local database with the remote server. The hook returns an object with a single property syncChats that is the reference to the sync function from the ChatsSyncingContext.

useAuthentication: This is a hook that uses the Firebase authentication API to handle user authentication. It returns an object indicating whether the user is currently logged-in, logged-out or it is not yet known. If the user is logged-in it will provide information about this user, such as the user id. It will automatically update to authentication state changes by listening to the onAuthStateChanged method from Firebase auth. When the component using this hook is unmounted, the returned function from useEffect is called to unsubscribe from the authentication state change. The hook also logs the current user's email when the authentication state changes.

useCurrentLocation: This hook allows a component to access the user's current location. The hook uses the "expo-location" package to request permission to access the user's location, and then uses the "getLastKnownPositionAsync" and "getCurrentPositionAsync" methods to retrieve the user's current coordinates (latitude and longitude). The hook also logs a message when the user denies permission to access their location.

3.2.4 Contexts

Context in React Native can be used to share data and state between components without having to pass props down through multiple levels of component nesting. This can make the component hierarchy more flexible and reduce the need for props drilling, making the code more readable and easier to maintain.

Here are the following contexts that we used:

- **ChatsDbContext:** This context is used to provide a ChatsDb object, the local database for chats, to components throughout the app. The ChatsDb object is instantiated in the ChatsDbProvider component and passed as the value of the context. The purpose of this is to make the ChatsDb object easily accessible to all components in the app without having to pass it down through props.
- **ChatsDbSyncedContext:** This context provides a ChatsDb object and a number (dbUpdatedDependency) that will be increased every time the db updates. The context is re-rendered every time the dbUpdatedDependency number changes. This allows components that use the ChatsDbSyncedContext to refetch data from the local database every time the database has been udpated.
- **ChatsSyncingContext:** This context provides a sync() function to its consumers. The sync() function downloads current chat information and new messages from an API and updates the local database with the new data. The provider component also listens for new message notifications and calls the sync() function when it receives one. Additionally, it runs the sync() function once on initial load. After synchronizing it notifies ChatsDBSyncedContext of the changes in the database to trigger a rerender of the components that rely on up-to-date data from the local database.
- **NotificationEvent:** This code defines the types and structures of push notifications sent to the application. It allows to parse a new notification to a specific type to handle it further.

4 Push Notifications

The application uses push notifications to receive important real-time updates. We send push notifications in two cases:

- a new user joined your meeting
- a chat where you are member of received a new message

These types of push notifications are described in the NotificationEvent file, which contains the event names and their corresponding data structure. When a notification is opened by the user the app automatically opens the corresponding screen, which (depending on the notification type) is either meeting details screen or the chat screen. Furthermore, components can subscribe to any incoming push notification, which is used by the chats to update in real time when a new message notification is received.

5 Chats

To provide the chats feature, a more complex functionality had to be designed. The requirements are, that the chat should be available when offline and automatically synchronize with the server when necessary. The necessary functions implemented for these requirements are discussed in the following sections.

5.1 Local database

The app uses a local sqlite database to store information about the chats and their messages. This allows the app to load messages, even when currently no internet connection is available. The database stores following information about chats where the user participates:

- **Chats:** id, title and type (private/meeting)
- **Members:** chat id, user id, user name and user image
- **Messages:** message id, chat id, sender id, message and date
- **Read:** chat id, user id and date of last message read

Access to the database is provided through the ChatsDbContext.

5.2 Synchronization

To ensure that the database has up-to-date data it needs to be synchronized with the Firestore database. In following events we trigger a synchronization:

- **App start:** this ensures that messages sent while the app was closed are loaded
- **Message received:** this ensures that push notifications of new messages will be saved in the local database.
- **Message sent:** this ensures that messages sent from the user themself are also stored in the local database.
- **Meeting joined:** this ensures that the chat of the corresponding meeting is also stored in the local database.

On a synchronization the application fetches all new messages from the server. It does so by using the Read date of the local database and only fetching messages with a newer date per chat. In addition to the messages, it also checks if the user joined new chats or any chats have been updated, and if yes, changes the database accordingly.

5.3 Live updates

The user interface needs to reflect the changes in the database. For instance, when a new message is received the chat preview should increase the new unread counter and display the new last message. To implement this use case, we created the ChatsDbSyncedContext. This context provides a reference to the database and a ”dbUpdatedDepency”. This dbUpdatedDepency counter is incremented every time a synchronization is performed. Therefore, hooks that use the database can add this counter to the useEffect dependencies and automatically update after a synchronization is performed. An example of this is shown in listing 1, where useChatMetadata passes the dbUpdatedDepency to the useEffect dependencies and thus fetches the chat metadata from the database every time the database has been updated.

```
export function useChatMetadata(chatId: string) {
  const { db, dbUpdatedDependency } = useContext(ChatsDbSyncedContext);
  const [metadata, setMetadata] = useState<IChatMetadata>();

  useEffect(() => {
    db.getChatMetadata(chatId).then(setMetadata);
  }, [db, chatId, dbUpdatedDependency]);

  return {
    chatMetadata: metadata,
  };
}
```

Listing 1: the useChatMetadata uses the dbupdatedDependency to automatically rerender on database updates

6 Design guidelines

Design guidelines are a set of principles and standards that provide a consistent approach to design and ensure that a product or service is visually appealing and functional. Their purpose is to ensure a consistent look, feel and user experience, while also providing a framework for designers to work within.

In our mobile application we used the material guidelines which we will discuss in the next section. These guidelines were useful at suggesting recommendations for the final design of the layout, typography, color, icons and touch gestures.

6.1 Material

6.1.1 Material Design

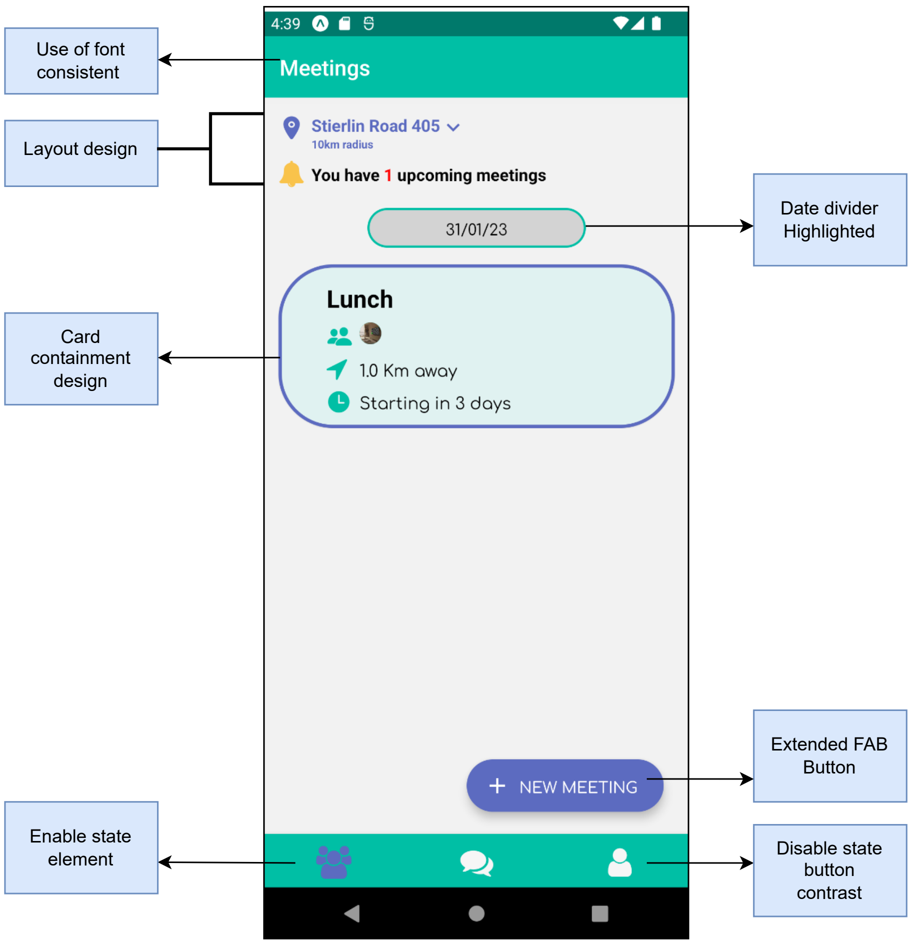
Material Design [Gooe] is a design language developed by Google in 2014. It provides a consistent visual, motion and interaction design across all platforms and devices, including mobile and web applications. Material Design features a flat, colorful, and intuitive interface inspired by the physical world and its textures, shadows, and light.

In the picture below we can observe how the advices about the mobile app-front end design were applied. The use of the font should be consistent through the whole app, this means that Headlines, subtitles, body and buttons must have the same size and weight in order to provide a neat and clean looking.

We thought about what is important to highlight in the main screen and how it should be displayed in order to get the user attention and be easy to read. Therefore, we decided to display the user current location and the amount of meetings he has subscribe to.

The Date divider has its own style that makes it easy to read and to spot if there would be more meetings, avoiding the screen to become cumbersome. On the Design of the card we show the title, up to five members avatars, the distance from your location and when is starting, this makes the card also easy to read and helps the user to make a quick decision on join or not the meeting.

On the bottom part we can see how the extended FAB button which is the main button to feed the main purpose of this mobile application is highlighted with color and elevation being this one unavoidable for the user. The contrast when an screen is activated to make easier user navigation through the app.



6.1.2 React Native Paper

React Native Paper [Cala] is a UI component library for React Native that follows Google's Material Design Guidelines. It provides a set of customizable and reusable components for building user interfaces in React Native.

The use of React native paper brings many benefits since the default properties that the components come with comply with the material design purposes and aims. This allows us to focus more on arranging the content and layout, and then put some tweaks were necessary.

Another useful feature is the easy access of a globally defined theme. With the `useTheme` hook one can get the currently selected theme and use it to obtain current colors. React Native Paper allows to adjust the default themes. On the one hand this ensures, that we use the same color across the app. On the other hand this also simplifies the process of iterating through possible themes, as a change in one place updates the theme in the whole application. And finally, this also makes it easy to implement a dark theme.

The light and dark themes we chose are displayed in listing 2.

```
export const customLightTheme: ReactNativePaper.Theme = {
  ...DefaultTheme,
  fonts: configureFonts(fontConfig),
  colors: {
    ...DefaultTheme.colors,
    primary: "#00BFA5",
    statusbar: "#00796B",
    accent: "#5C6BC0",
    surface: "white",
    bell: "#f9c34b",
    names: "#37474F",
  },
};

export const customDarkTheme: ReactNativePaper.Theme = {
  ...DarkTheme,
  fonts: configureFonts(fontConfig),
  colors: {
    ...DarkTheme.colors,
    primary: "#62374E",
    accent: "#007880",
    statusbar: "#33313B",
    bell: "#f9c34b",
    names: "#37474F",
  },
};
```

Listing 2: Configuration of themes using React Native Paper

6.2 Theming

We have used Material design theming guidelines and we searched for a contrast that helps the user to spot the things that we want that he looks at, also making the app with a good feeling with some colors easy to reminder creating a relation in the user's brain.

Here you can see the app in the use of both light and dark theme. The theme automatically adjusts according to the system settings.

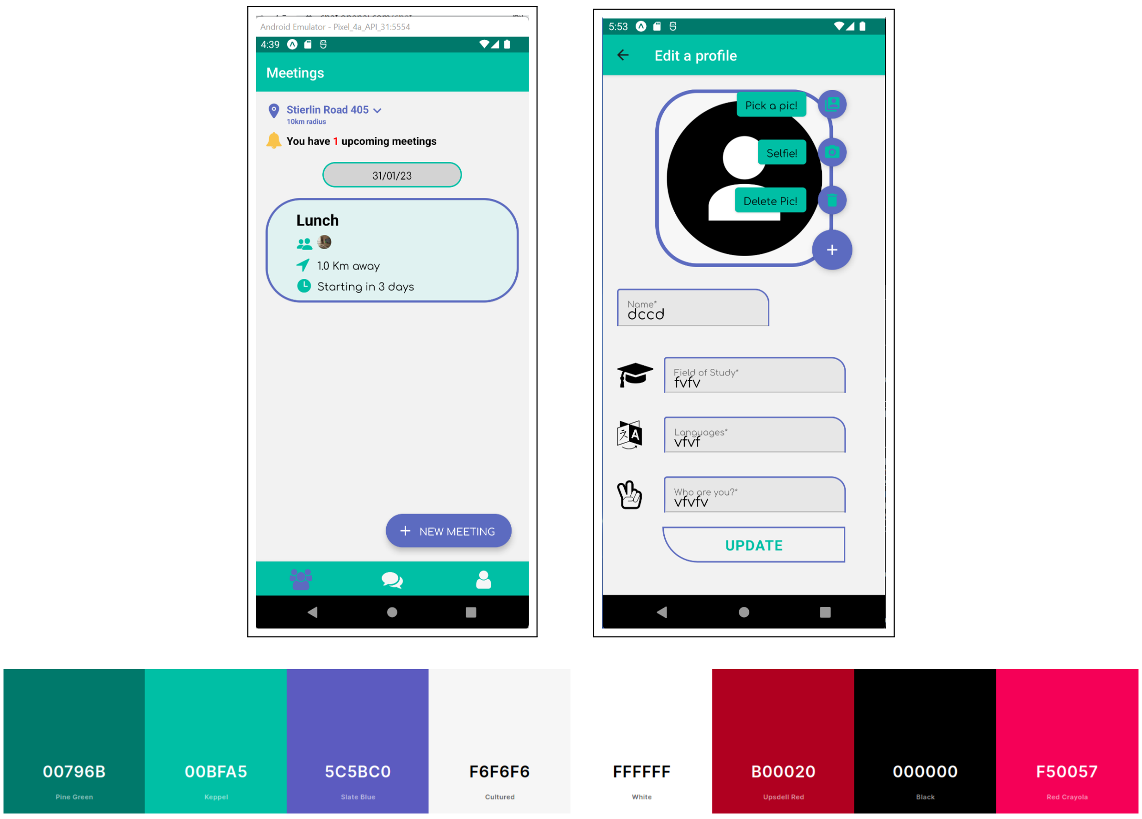


Figure 3: Light theme

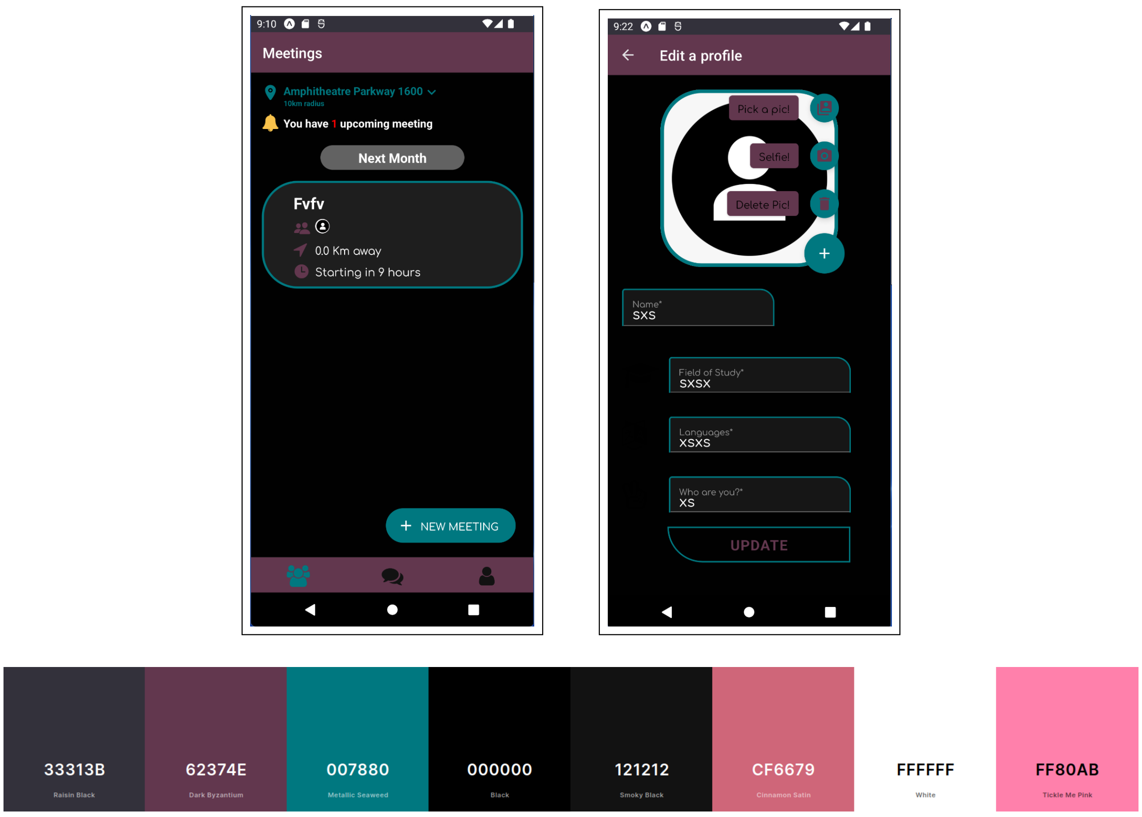


Figure 4: Dark theme

7 Test campaign

A test campaign in React Native mobile app development refers to a series of tests that are conducted to ensure that the app is working correctly and that it meets certain quality standards. In the development of our application we decided to implement Unit test, Component test and End-to-end test.

During the implementation of the tests we identified some bugs, showing us the importance and the usefulness of them. In the implementation of the tests we followed React Native guidelines [Nat], therefore we use the react-native testing library [Calb] using jest as our framework [Jes].

Within the unit and component tests we achieved a line coverage of more than 50 percent. Considering that the configuration files and files that are hard to unit test (such as the App.tsx) are all covered by the end to end tests this number represents a significant amount of the files. By running `npm run test:ci` this test suite is run, displaying the coverage details per file and creating a report at coverage/lcov-report/index.html. This report can be opened for further details.

Furthermore, we used the Arrange-Act-Assert (AAA) paradigm, in which the first step of a test arranges the test environment, the second step performs the action we want to test, and the third test asserts the correct outcome of this action given the environment. This allows to make the code easily understandable, as the different parts are well separated.

7.1 Unit tests

Unit tests in React Native are automated tests that are used to test individual units of code in a React Native app.

Here we used Jest library to test the smaller part of our code which are the function and classes to ensure that the function used was returning the data and objects we needed to get from them.

During the implementation of the unit test, when testing a function we tested it the whole workflow of it, assuring that there was not bugs along it. For this we isolate the part that we wanted to test by mocking the rest of the parts involved and testing the specific part we desired to test.

An example of a unit test is showed in listing 3, where we test that the `fetchOwnChats` function uses the currently logged in user to filter the chats stored in Firestore.

```
it("fetchOwnChats filters where member contains current user id", async () => {
  mockedFirestore.getDocs.mockResolvedValue({ docs: [] } as any);

  await fetchOwnChats();

  expect(mockedFirestore.where).toHaveBeenCalledWith(
    "members",
    "array-contains",
    MOCKED_CURRENT_USER
  );
});
```

Listing 3: Unit test, testing that fetchOwnChats uses the currently logged in user to filter the chats stored in Firestore

7.2 Component tests

Component testing in React Native refers to the practice of testing individual components of a React Native app. These components can be visual elements such as buttons, text fields, or lists, or they can be non-visual elements such as services or utility functions.

During a component test, the component is rendered and interactions are simulated by passing in the necessary props and states. The output is then analyzed to ensure that the component is rendered correctly and behaves as expected. For example, a test for a button component might check that the button is rendered, that it can be pressed, and that it triggers the correct action.

Here we mainly have tested the screens and actions that are involved within it, checking that the initially renders are the ones expected, rendering time and buttons correctly interactions. In the implementation of the component tests we used the react native testing library [Calb].

One example of a component test is shown in listing 4, where we test that the ChatScreen renders the given messages from oldest to newest, as it is common in messengers. Note that the givenMetadata and givenMessages mock the local database in a way, such that the ChatScreen component receives them for displaying.

```
it("displays messages from oldest to newest", async () => {
  givenMetadata(sampleChatMetadata);
  givenMessages(sampleMessages);

  render(
    <ChatScreen
      navigation={null as any}
      route={routeFor(sampleChatMetadata.id)}
    />
  );

  const messages = screen.queryAllByTestId(/message-\d+/);
  expect(messages).toHaveLength(3);
  expect(messages[0]).toHaveTextContent(sampleOldMessage.message);
  expect(messages[1]).toHaveTextContent(sampleMessage.message);
  expect(messages[2]).toHaveTextContent(sampleNewMessage.message);
});
```

Listing 4: Component test, testing that ChatScreen displays the messages in the right order

7.3 End-to-end tests

End-to-end (E2E) testing in React Native refers to the practice of testing the entire app from the user interaction up to the backend service. This test takes the view that a user has when interacting with the app, thus simulating the interactions and actions that a real user would perform. E2E tests are used to ensure that the app is working correctly and that all the different parts of the app are working together as expected.

In the implementation of this we used Detox https://wix.github.io/Detox/ to run the tests in the Android emulator. It will create a build of the application, install it on the android device and then interact with it as a user would do. As a backend we used the official Firebase local emulators, which mimic the behaviour of Firebase. By seeding the local emulator, we can start the tests from a complex state where some users have already created profiles and meetings and have exchanged messages. As these tests are time expensive to run, they cover the main features, such as logging in, creating a profile, creating and finding meetings and writing messages.

As an example, listing 5 shows an end to end test which will be run directly on the Android emulator. The test first navigates to a chat by clicking on a chat preview, then enters a message to the message input, presses on the send button and finally verifies that it can see the newly sent message. All of this runs on the build application without any mocks and the local Firebase emulators, thus requiring that all the different components interact properly with each other. In this case, this short test tests: The navigation to the chat. The message input handling and passing the result to the parent component. The storage of the message in Firestore. The synchronization of the Firestore chats with the local database and also that this synchronization automatically updates the displayed messages. And finally, that the displaying component realizes the message is from us and displays "You" with the corresponding message.

Taking this into consideration it is clear why these tests are time consuming, but also how useful they are to catch unobvious errors in the interplay of components even with short tests.

```
it("can send a new message", async () => {
  const message = "Hi from the test";
  await element(by.id("chat-preview-0")).tap();

  await element(by.id("message-input")).typeText(message);
  await element(by.id("send")).tap();

  // shows author and message
  await expect(element(by.id("from-1"))).toHaveText("You");
  await expect(element(by.text(message))).toBeVisible();
});
```

Listing 5: End-to-end test, testing that a user can send a new message in a chat

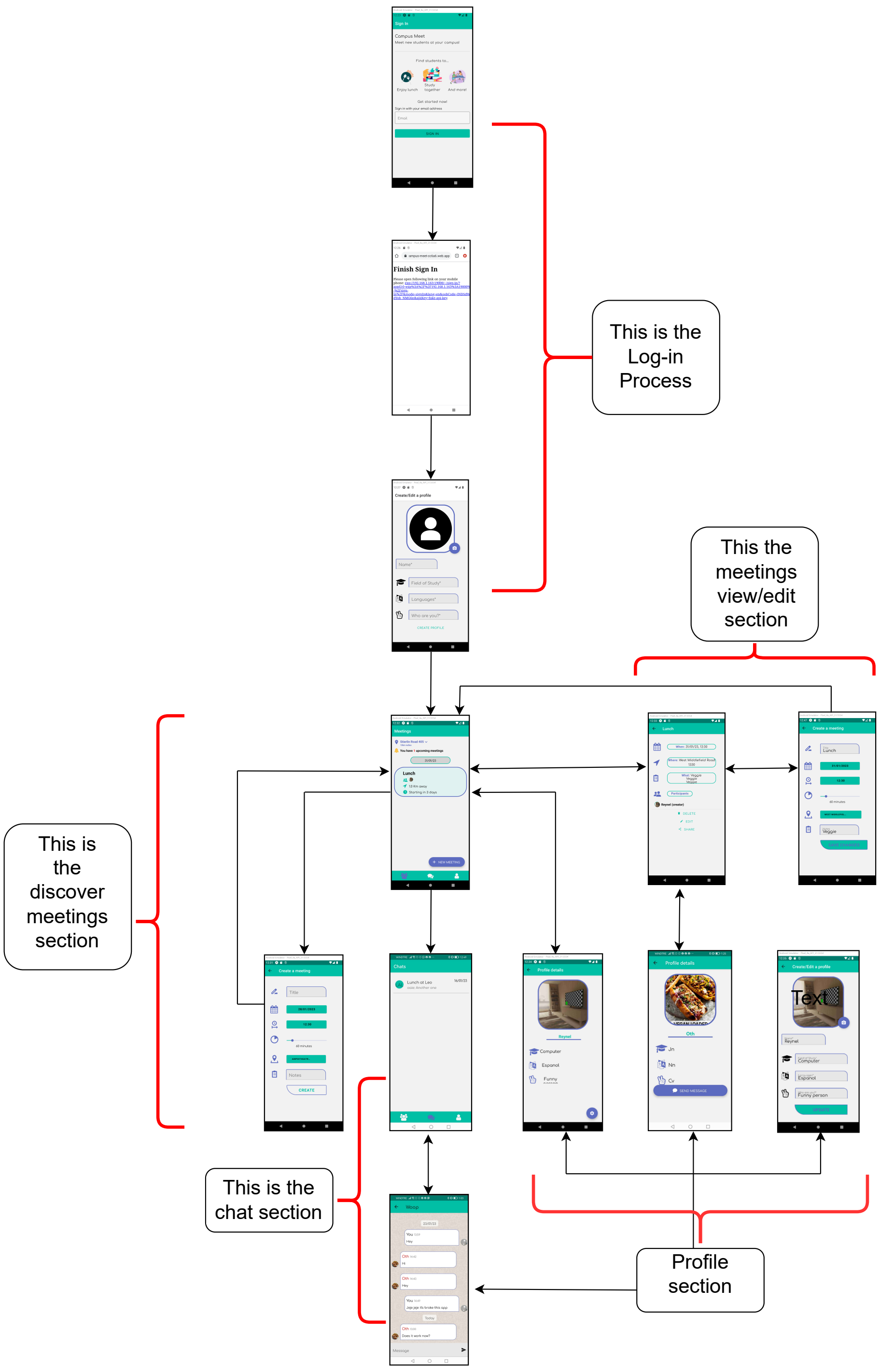
8 Screenflow

The screen-flow can give you an overview how the different screens are connected to each other and how users can navigate from screen to screen.

In our app screen-flow you can see there five main sections that you can navigate through.

1. **The Log-In process:** This includes typing your email address and opening the link received via email, and finally creating the profile
2. **Discover Meetings:** After creating the profile the user will be redirected to this section. Here they can find meetings close to them and create new meetings.
3. **View and Edit Meetings:** In this section the users can see the detailed information of a meeting and can edit their own meetings.
4. **Chats:** Here the users see their own chats and can write and receive new messages.
5. **Profiles:** In this section the user can view profiles of other people or update their own profile.

After the login process the user has a bottom navigation, where they can use three tabs to switch between discovering meetings, viewing chat previews and showing the own profile. Starting from these tabs users can navigate through the other screens. More details will follow in the screen section 9.



This is the Log-in Process
This the meetings view/edit section
This is the discover meetings section
This is the chat section
Profile section

9 Screens

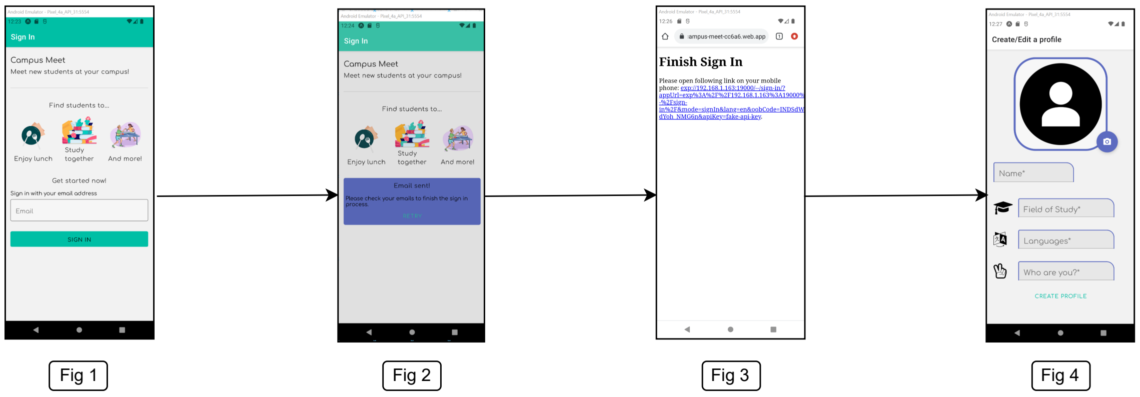
In this section we will show the screens in detail and describe how they work and connect with the others screens.

9.1 Log-in flow

Firstly, we will talk about the Log-In flow. When a new user opens the Campus Meet, the screen shown in **Fig. 1** is displayed. This screen displays general information about Campus Meet and also an email input to login. When entering and submitting the email it will display a message saying as in the **Fig. 2**: "Email sent! Please check your emails to finish the sign in process.".

When you have received the email and click on the sent link, it will take you to the pre-determined browser that you have set up in your phone and here you will see a link which is the final step to of the "Log-in" Process as you can see in the **Fig 3.**.

As last part of this you have to create a profile, filling in all the shown fields in **Fig 4.** A picture is not mandatory to upload but the text input fields are mandatory.



Finish Sign In

Please open following link on your mobile phone: exp://192.168.1.163:19000/--/sign-in/?appUrl=exp%3A%2F%2F192.168.1.163%3A19000%-%2Fsign-in%2F&mode=signIn&lang=en&oobCode=INDSdWdYoh_NMG6n&apiKey=fake-api-key.

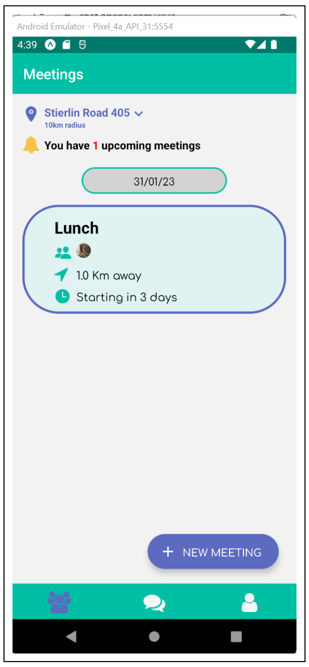
9.2 Discover Meetings Screen

This is the main Screen of our application. It shows the user all meetings that are close to them.

On the top-left part, it will show you your current location plus the radius of search that has set to look for meetings around. Clicking on it will open a map that shows the meetings as markers on the map and allows to configure the search radius with a slider.

After the location, it will show you an alert of how many meetings you have subscribed to. The meetings that you have joined will be highlighted with a ”light-purple” border to make them easy to spot on this screen. The meetings are displayed as cards and show the key information used to decide if the meeting could be interesting. This includes the title, the distance, in how many days, hours or minutes it will start and the avatars of people who already joined. If the meeting has already started it will show you in how many minutes will be finished instead. The meetings are grouped by the day and a above each group the date is shown.

On the bottom-right part, there is a FAB button ”NEW MEETING” to create new meetings. Clicking it will take you to the Create meeting screen which will be shown later. As with all other screens, the bottom allows to navigate to the other tabs.



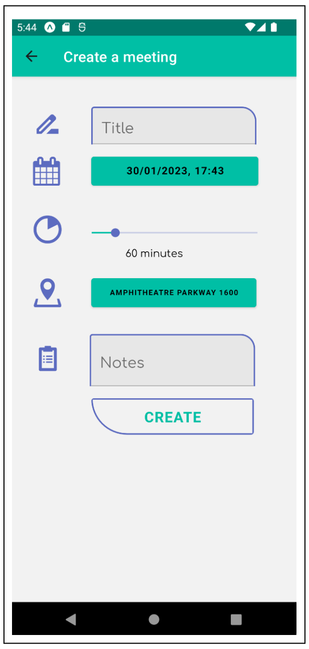
9.3 Create Meeting Screens

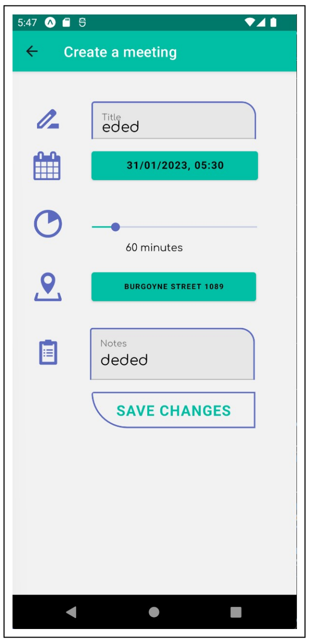
In these screens you can see all the information that you add to the meeting. Starting with a title which will be shown in the upper part of the card in the discover meetings screen. You must choose a date, time and duration of it. The buttons displays the current date and time but as soon as you change them they will save the date and time that you select.

At the end you can choose a location where the meeting is going to take place and a note for the members that in the future will be part of. The choose location button also is displaying initially your current location but it will save the chosen location.

After you have filled up all the fields you can click on "CREATE" and it will take you to the main screen showing you the just-created meeting. If you want to go back , you can do it and it will not save anything.

Additionally this screen also serves as the edit meetings screen, as it shares the same UI. When editing an existing meeting it will load the meeting information into the input fields initially. You can update the inputs and then save it with the "SAVE CHANGES" button. Only owners of meetings are able to edit meetings.

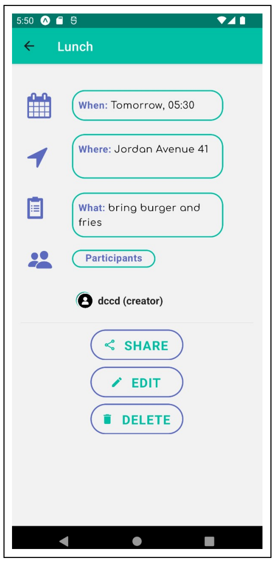


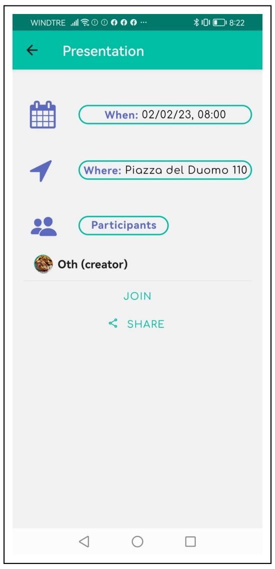


9.4 Meeting details Screen

The meeting details screen will be shown when you click in any of the cards that are in the Discover Meetings screen. In here you can see when the meeting takes place (Date and time), where it is happening (a clickable button that will take you to the maps application and display the meeting location) and finally the participants. If you click on the participants avatar it will redirect you to their profiles.

In the bottom part you can see three buttons, if you are not member of the meeting you will see a "Join" button as is shown in the right-side picture, which if you click then "GO TO CHAT" text button will pop-up. If you are already part of it like in this case then you will see a "leave" text-button which will take you out of the meeting. Additionally if you wish to share the meeting information –Title , Date, time and location– through different social media channels – Whatsapp, Facebook, etc. – you can do it by clicking on the "SHARE" text button.

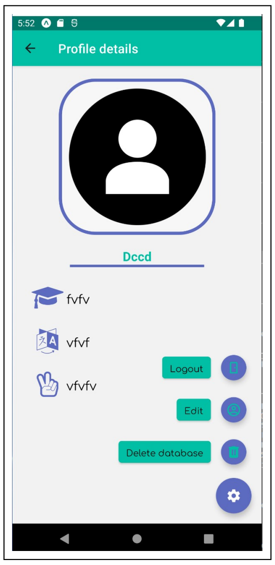


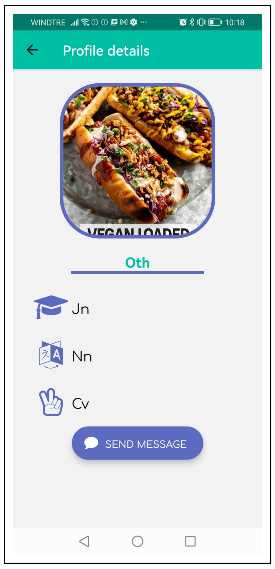


9.5 Profile details Screens

In these screens we can see our own and others profile. In your own profile you will see the screen on the left-side shown below, here you can edit the information of your profile and log-out of the app if you wish as well.

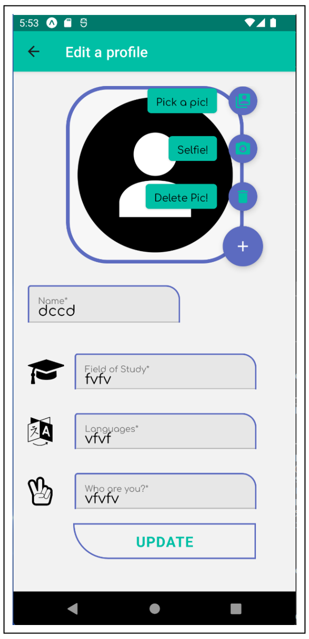
When you visit others people profile you can start a chat with them by pressing a button that is at the bottom called "SEND MESSAGE".





9.6 Edit Profile Screen

In this screen you are able to modify your own profile, you can change your name, field of study, language you speak and interest/hobbies. Also, you can change your profile picture by taking a selfie! or picking a picture you like from the gallery of your phone. If you click on the "UPDATE" button it will redirect you to the "Profile details screen" and you can see how your profile looks with the changes.



Name* dccd

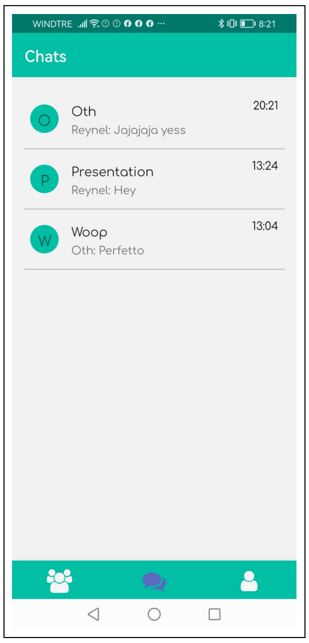
Field of Study* fvfv

Languages* vfvf

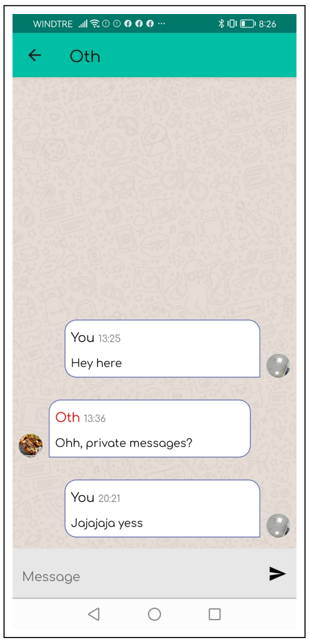
Who are you?* vfvfv

9.7 Chat Screens

In these screens, on the left side we can see the preview chats screen, in which you can see the chats of the meetings that you have joined and also the private chats you have started. This preview displays the title of the chat, the last message and the date of the last message. On the right side you can see how the chat screen look like. Messages are styled as bubbles, aligned either on the left side for messages from other people, or on the right side for your own messages. Each message bubble shows the sender, time and content next to the avatar of the sender.



O	Oth Reynel: Jajajaja yess	20:21
P	Presentation Reynel: Hey	13:24
W	Woop Oth: Perfetto	13:04



You 13:25
Hey here

Oth 13:36
Ohh, private messages?

You 20:21
Jajajaja yess

10 Firestore database scheme

This section discusses the structure and organization of data within the Firestore database. It defines the collections, documents, and fields within the database, and how they relate to each other. Following vocabulary specific to Firestore is used:

- **Collection:** a container for documents, the top-level object in the Firestore data model. Each collection can contain multiple documents.

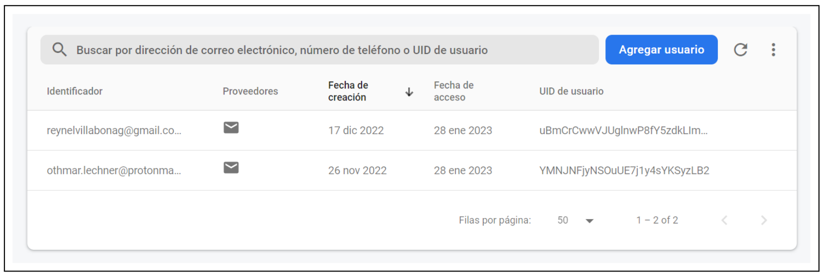
- **Document:** a container for fields which potentially can contain nested collections. Each document is identified by a unique ID.
- **Field:** a key-value pair within a document, where the key is a string and the value can be a number, string, boolean, date or another nested structure.

10.1 Profiles Database Scheme

In the picture below we can observe on the left side the Collections which, as said above, are the top level object, we assigned the name for the collection, after this the collection is compose by documents which are created for each user that log in and create a profile, then there will not be one email with two documents, this makes the ID of the document a personal ID for each user information.

The document contains different fields in this case they are the followings:

- **"About":** Refers to the Text-input "Who are you?" in the "Edit/Create profile screen". It accepts data type "String" which means a sequence of characters, such as letters, numbers and symbols.
- **"creator":** Refers to the UserID which is given by the firestore Authentication function, this is also a unique ID for each User or E-mail logged-in. It accepts data type "String".

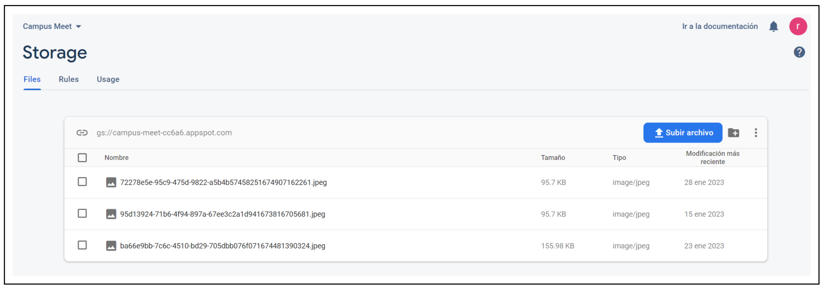


Buscar por dirección de correo electrónico, número de teléfono o UID de usuario

Identificador	Proveedores	Fecha de creación	Fecha de acceso	UID de usuario
reynelvillabonag@gmail.co...	✉	17 dic 2022	28 ene 2023	uBmCrCwwVJUglnwP8fY5zdkLIm...
othmar.lechner@protonma...	✉	26 nov 2022	28 ene 2023	YMNJNFjyNSOuUE7j1y4sYKSyzLB2

Filas por página: 50 1 – 2 of 2

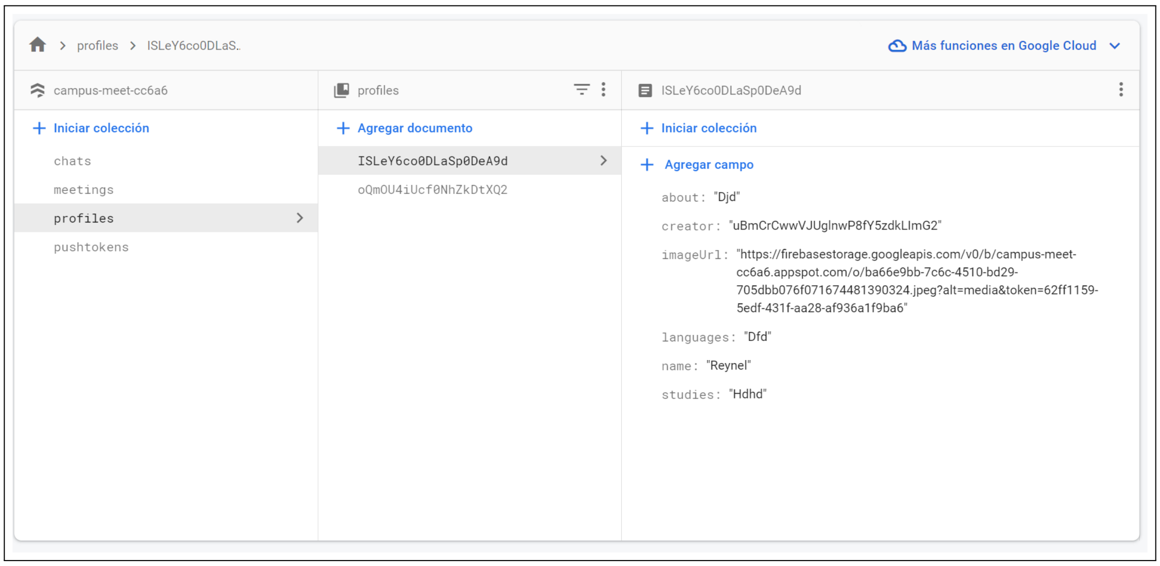
- **"imageUrl":** Makes reference to the link with which is Storage the uploaded picture. As you can see in the image below, the picture is uploaded with a "String" data type ".Jpeg", but then this "String" it attached to the link of the "Storage bucket"–gs://campus-meet-cc6a6.appspot.com/–. It accepts data type "String".



gs://campus-meet-cc6a6.appspot.com

Nombre	Tamaño	Tipo	Modificación más reciente
72278e5e-95c9-475d-9822-a5b4b57458251674907162261.jpeg	95.7 KB	image/jpeg	28 ene 2023
95d13924-71b6-4f94-897a-67ee3c2a1d941673816705681.jpeg	95.7 KB	image/jpeg	15 ene 2023
ba66e9bb-7c6c-4510-bd29-705dbb076f071674481390324.jpeg	155.98 KB	image/jpeg	23 ene 2023

- **"languages":** Makes reference to the Input-text "Languages" in the "Edit/Create profile screen". It accepts data type "String".
- **"name":** Makes reference to the Input-text "Name" in the "Edit/Create profile screen". It accepts data type "String".
- **"studies":** Makes reference to the Input-text "Field of Study" in the "Edit/Create profile screen". It accepts data type "String".



```
chats
meetings
profiles
pushtokens
```

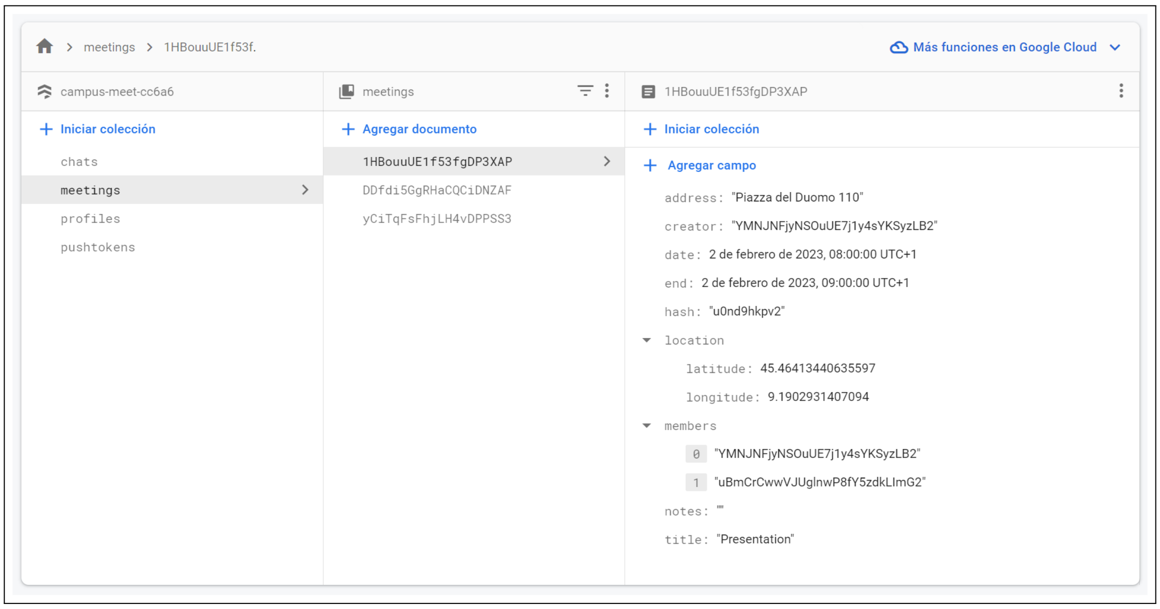
```
ISLeY6co0DLaSp0DeA9d
oQmOU4iUcf0NhZkDtXQ2
```

```
about: "Djd"
creator: "uBmCrCwwVJUglnwP8fY5zdkLImG2"
imageUrl: "https://firebasestorage.googleapis.com/v0/b/campus-meet-
          cc6a6.appspot.com/o/ba66e9bb-7c6c-4510-bd29-
          705dbb076f071674481390324.jpeg?alt=media&token=62ff1159-
          5edf-431f-aa28-af936a1f9ba6"
languages: "Dfd"
name: "Reynel"
studies: "Hdhd"
```

10.2 Meetings Database Scheme

Here, we assigned to this collection the ”meetings” name, then we can see the ID of each document that contains each meeting information, the fields of each document are the following:

- **”address”:** A textual description of the address chosen by the user on the map. It accepts data type ”String”.
- **”creator”:** Refers to the user ID that created this meeting. It accepts data type ”String”.
- **”date”:** Makes reference to the date chosen at which the meeting will start. It accepts ”Date” data type which is a JavaScript object that represents a single moment in time. It stores a date and time as a numeric value, which represents the number of milliseconds since January 1, 1970, 00:00:00 UTC (coordinated universal time).
- **”end”:** Makes reference to the date at which the meeting will end. It is calculated by adding the duration chosen by the user to the started date. It also accepts ”Date” data type.
- **”location”:** Refers to a set of two values, latitude and longitude, gotten from the Geolocation API, which is called by the method ”getCurrentPosition()”. This field of the document accepts ”Coordinates” data type.
- **”members”:** This field storage the UserID of the users that have joined the meeting. It is important to mention that this ID is associated with the E-mail registered. This accepts ”string” data type.
- **”notes”:** This field takes the Text that have been typed in the ”Notes” space in the ”Create/Edit Meeting Screen”. This accepts ”string” data type.
- **”title”:** This field in the document takes the Text that have been typed in the ”Title” space in the ”Create/Edit Meeting Screen”. This accepts ”string” data type.



meetings > 1HBouuUE1f53f.

Más funciones en Google Cloud

campus-meet-cc6a6 | meetings | 1HBouuUE1f53fgDP3XAP

Iniciar colección | Agregar documento | Iniciar colección

chats
meetings
profiles
pushtokens

1HBouuUE1f53fgDP3XAP
DDfdi5GgRHaCQCiDNZAF
yCiTqFsFhjLH4vDPPSS3

Agregar campo

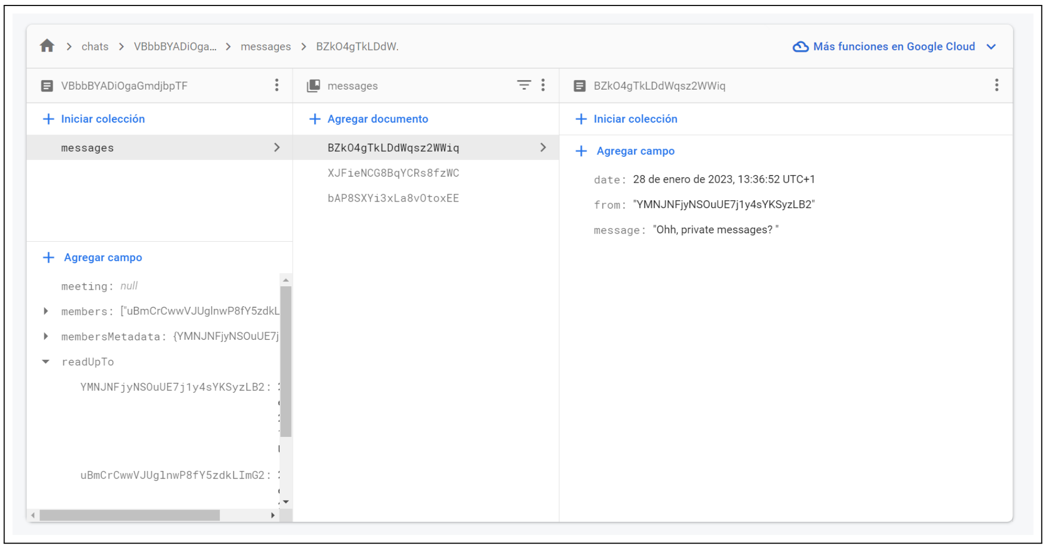
```
address: "Piazza del Duomo 110"
creator: "YMNJNFjyNSOuUE7j1y4sYKSyzLB2"
date: 2 de febrero de 2023, 08:00:00 UTC+1
end: 2 de febrero de 2023, 09:00:00 UTC+1
hash: "u0nd9hkpv2"
location
    latitude: 45.46413440635597
    longitude: 9.1902931407094
members
    0 "YMNJNFjyNSOuUE7j1y4sYKSyzLB2"
    1 "uBmCrCwwVJUglnwP8fY5zdkLImG2"
notes: ""
title: "Presentation"
```

10.3 Chats Database Scheme

The chats Database Scheme is compose by the name of the collection "chats", Each document belong to each chat created for each of the meetings. The within-data-structure of the document following structure:

- **"messages":** A subcollection of a chat. It contains all messages sent in this chat.

  Each message sent has inside "date" field referring to the specific time and date when the message was sent. "From" containing the User Id that sent this message and the "message" field which is the message itself.



chats > VBbbBYADiOga... > messages > BZkO4gTkLDdW.

Más funciones en Google Cloud

VBbbBYADiOgaGmdjbpTF | messages | BZkO4gTkLDdWqsz2WWiq

Iniciar colección | Agregar documento | Iniciar colección

messages

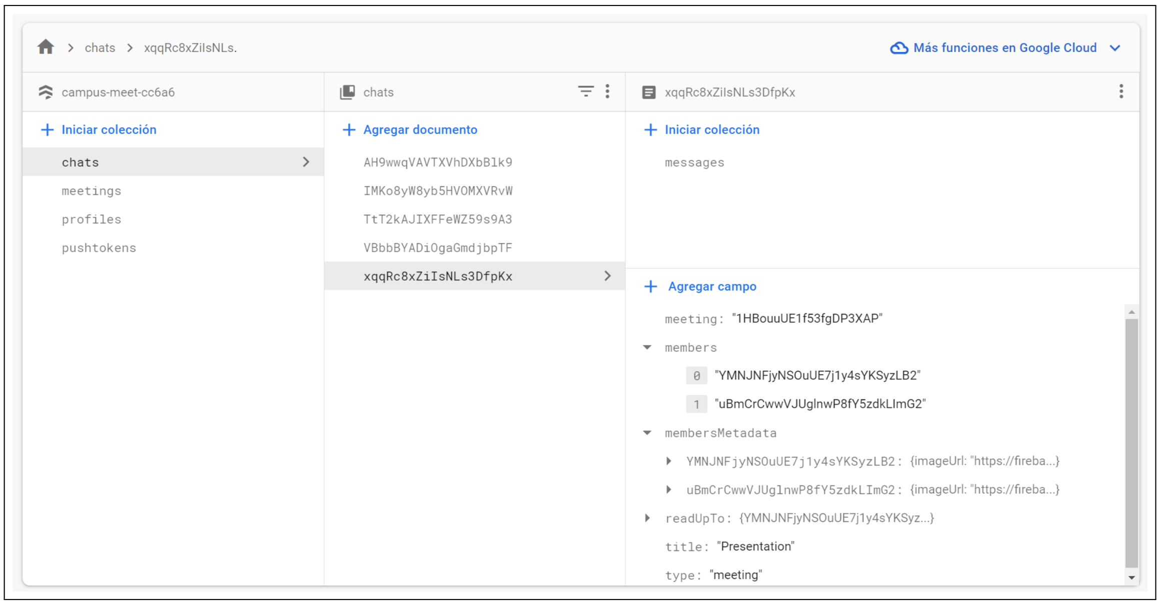
BZkO4gTkLDdWqsz2WWiq
XJFieNCG8BqYCRs8fzWC
bAP8SXYi3xLa8vOtoxEE

```
Agregar campo
meeting: null
members: ["uBmCrCwwVJUglnwP8fY5zdkL
membersMetadata: {YMNJNFjyNSOuUE7j
readUpTo
    YMNJNFjyNSOuUE7j1y4sYKSyzLB2:
    uBmCrCwwVJUglnwP8fY5zdkLImG2:
```

```
Agregar campo
date: 28 de enero de 2023, 13:36:52 UTC+1
from: "YMNJNFjyNSOuUE7j1y4sYKSyzLB2"
message: "Ohh, private messages? "
```

- **"members":** Refers to the all the members that are part of the chat, "members" field has the userIDs associated with each of the members account. It accepts data type "String[]".
- **"membersMetadata":** Contains nested fields, each field contains the userID which maps to the name and the imageUrl of each member. This makes possible to display the name and the profile picture in the chat screen, shown in the past Section.
- **"readUpto":** This is a mapping of userID to the last message they have read. The last message is stored as a date for easy retrieval of new messages.
- **"title":** It has the title given by the creator to the meeting. It accepts data type "String".

- **"type":** It has the type of chat. A chat can be "meeting" or "private". It accepts data type "String".



chats > xqqRc8xZilsNLs.

Más funciones en Google Cloud

campus-meet-cc6a6

+ Iniciar colección

chats
meetings
profiles
pushtokens

chats

+ Agregar documento

AH9wwqVAVTXVhDXbBlk9
IMKo8yW8yb5HVOMXVRvW
TtT2kAJIXFFeWZ59s9A3
VBbbBYADiOgaGmdjbpTF
xqqRc8xZiIsNLs3DfpKx

xqqRc8xZiIsNLs3DfpKx

+ Iniciar colección

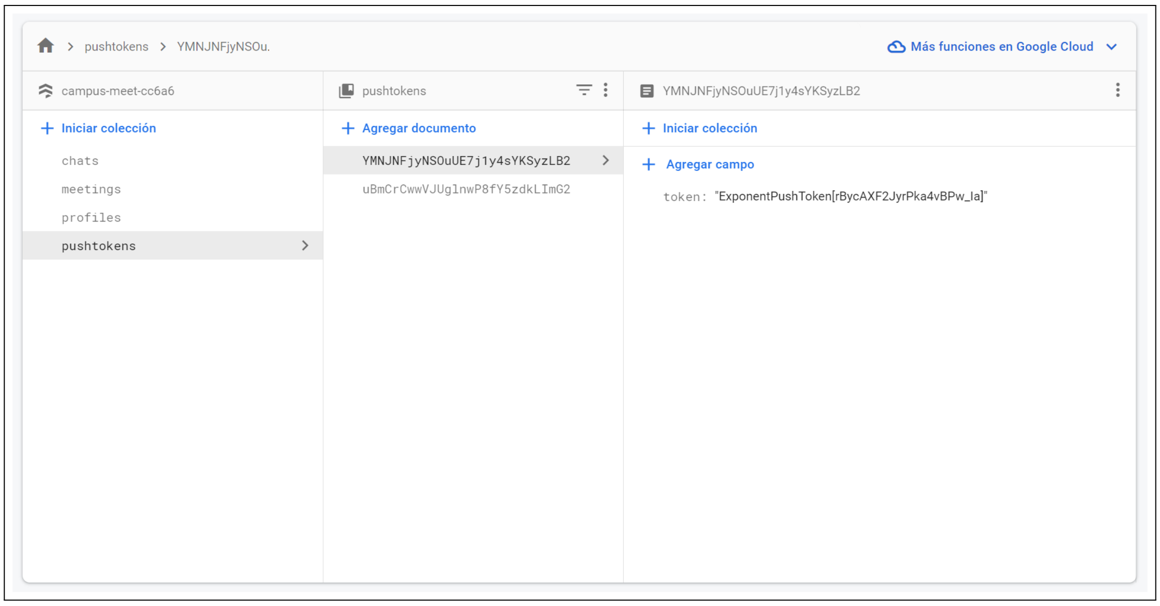
messages

+ Agregar campo

```
meeting: "1HBouuUE1f53fgDP3XAP"
▾ members
    0 "YMNJNFjyNSOuUE7j1y4sYKSyzLB2"
    1 "uBmCrCwwVJUglnwP8fY5zdkLImG2"
▾ membersMetadata
    ▸ YMNJNFjyNSOuUE7j1y4sYKSyzLB2: {imageUrl: "https://fireba...}
    ▸ uBmCrCwwVJUglnwP8fY5zdkLImG2: {imageUrl: "https://fireba...}
▸ readUpTo: {YMNJNFjyNSOuUE7j1y4sYKSyz...}
title: "Presentation"
type: "meeting"
```

10.4 Push tokens Database Scheme

The "pushtoken" collection contains documents which their IDs are each UserID, assigned when the sign-in process is complete. It maps the user IDs to the tokens used for push notifications.



pushtokens > YMNJNFjyNSOu.

Más funciones en Google Cloud

campus-meet-cc6a6

+ Iniciar colección

chats
meetings
profiles
pushtokens

pushtokens

+ Agregar documento

YMNJNFjyNSOuUE7j1y4sYKSyzLB2
uBmCrCwwVJUglnwP8fY5zdkLImG2

YMNJNFjyNSOuUE7j1y4sYKSyzLB2

+ Iniciar colección

+ Agregar campo

```
token: "ExponentPushToken[rBycAXF2JyrPka4vBPw_la]"
```

11 Development tools

11.1 Continuous integration

We used github actions to automatically run our test suite on every push. With this, in case a test should have failed we will receive an email notifying us of this error and we can start fixing it again.

11.2 Precommit linting and formatting

We used husky to setup a pre-commit hook. This hook automatically runs eslint and prettier on all changed files and fixes mistakes where possible. While prettier ensures a consistent code formatting, eslint warns about unused variables, wrong usage of react hooks and removes unused imports automatically. This allows to catch some errors early on and keep the codebase in a healthy shape. Sadly there was an error running this hook automatically on Windows, because of which some code may not be automatically linted and formatted.

11.3 Static typing

To allow better suggestions by the IDE and to prevent typing errors, where we expect a variable to have the wrong type or put a typo in a property name, we have used Typescript. Typescript is a statically typed language that compiles down to Javascript, sharing most of the syntax with Javascript. We used types wherever this is possible. Among other things this includes modelling the API responses, function parameters, parameters passed for navigation to other screens and more.

As an example, following interface is used to model chat objects that we use throughout the application.

```
export interface IChat {
  id: string;
  title: string | null;
  members: string[];
  membersMetadata: { [userId: string]: IProfileMetadata };
  readUpTo: { [userId: string]: Date };
  meeting: string | null;
  private: boolean;
}
```

References

[Cala] Callstack. *React Native Paper*. URL: https://callstack.github.io/react-native-paper/index.html.

[Calb] Callstack. *React Native Testing Library*. URL: https://callstack.github.io/react-native-testing-library.

[Exp] Expo. *Expo notifications*. URL: https://docs.expo.dev/push-notifications/overview/.

[Gooa] Google. *Firebase Authentication*. URL: https://firebase.google.com/docs/auth.

[Goob] Google. *Firebase Functions*. URL: https://firebase.google.com/docs/functions.

[Gooc] Google. *Firestore*. URL: https://firebase.google.com/docs/storage.

[Good] Google. *Google Maps*. URL: https://developers.google.com/maps/documentation/javascript.

[Gooe] Google. *Material Design*. URL: https://m2.material.io/design/introduction.

[Jes] Jest. *Jest*. URL: https://jestjs.io/docs/tutorial-react-native.

[Nat] React Native. *Testing - React Native*. URL: https://reactnative.dev/docs/testing-overview.