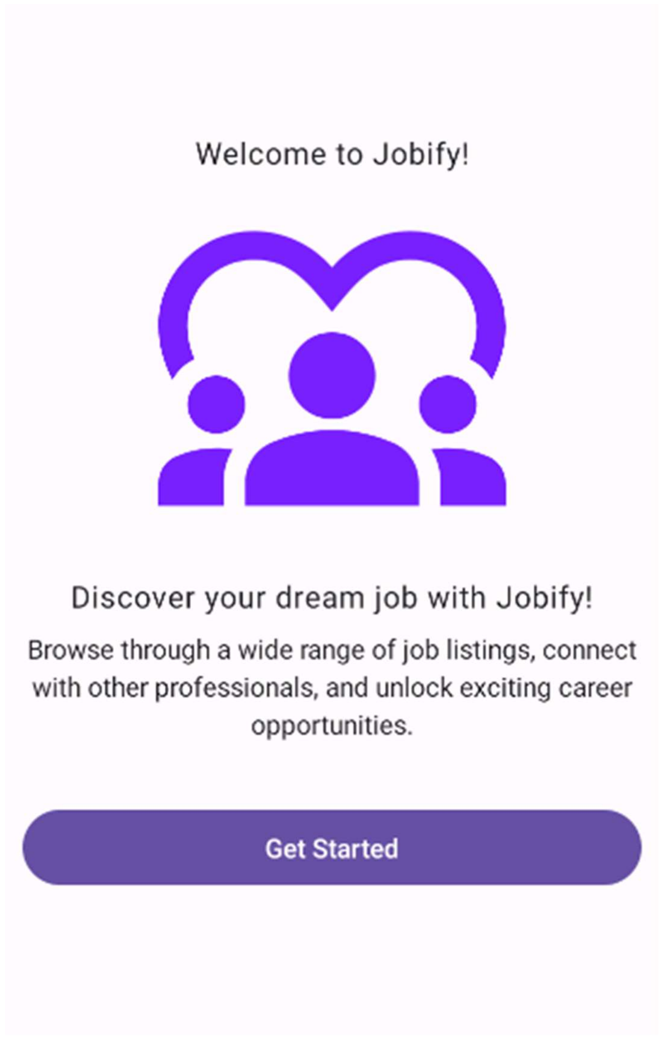
Report and Documentation (Part 2)

Overview of project

The idea of my app is a platform which allows for companies to put up job listings, with all the details, like the salary, the job description and the hours. It will also allow people looking for jobs to easily search for suitable jobs quickly. The primary target audience will be any corporation or startup company looking for employees, and people aged 18-40 searching for new jobs or internship opportunities.

It will be useful for them as it helps them in searching for new jobs or just simply looking to recruit others for companies they manage or for a freelance job. The interaction that other users will have on the app will also be to a high level, making communication much easier through the app.

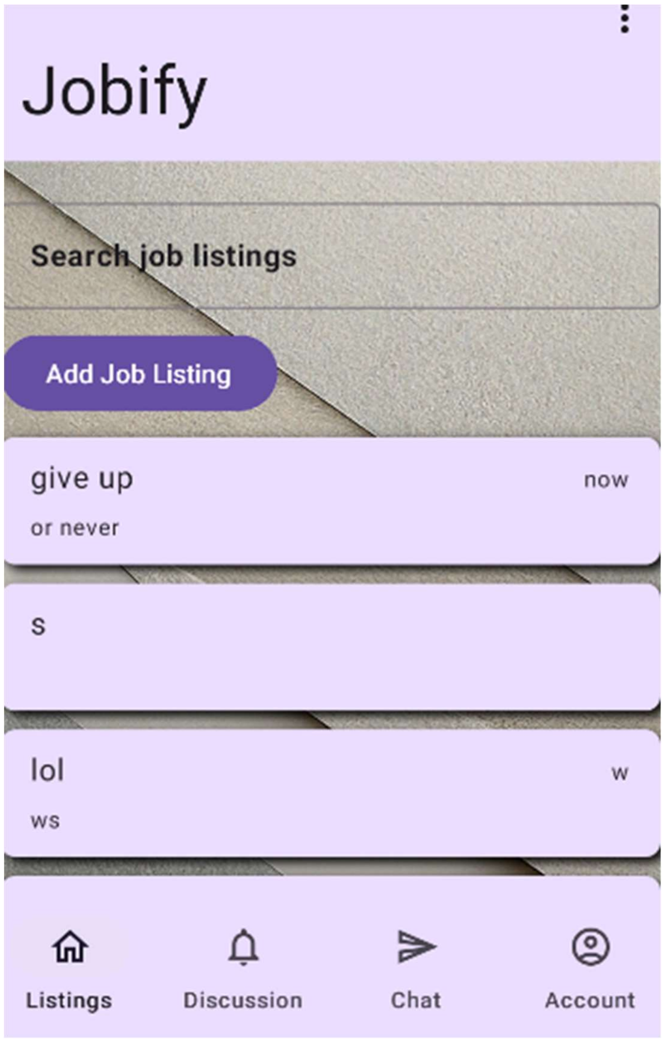
Documentation of App


Welcome to Jobify!

Discover your dream job with Jobify!

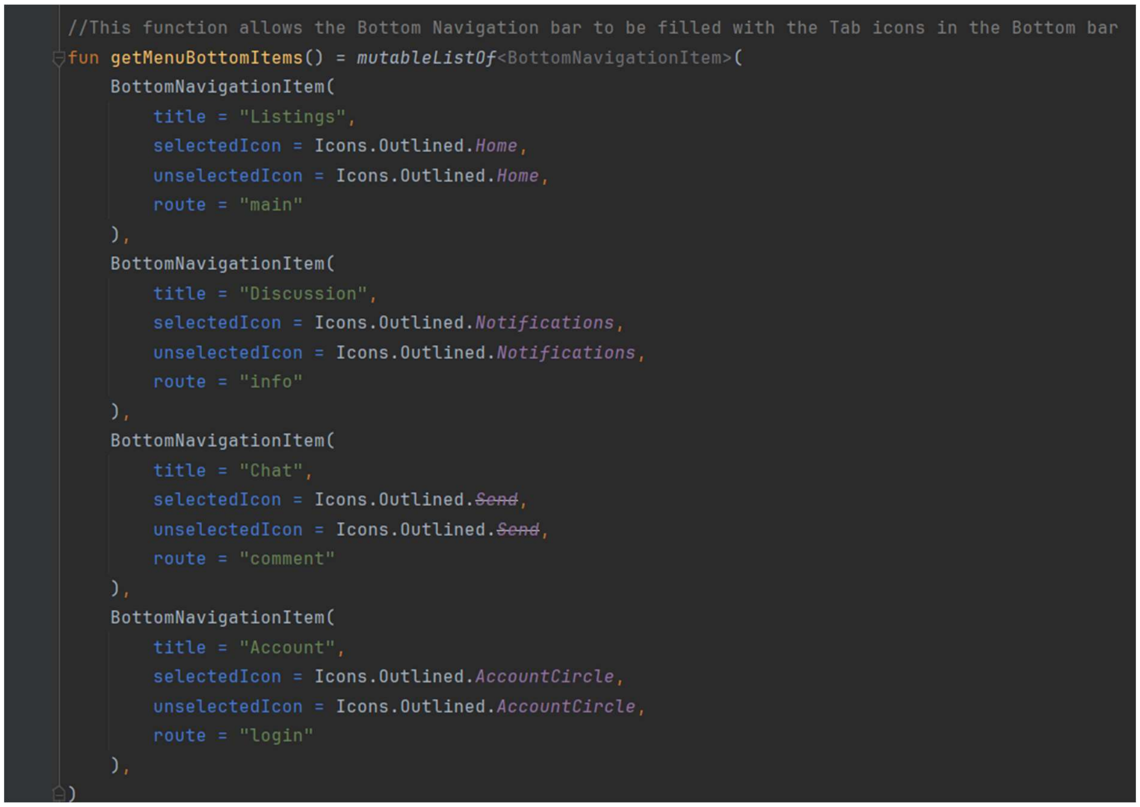
Browse through a wide range of job listings, connect with other professionals, and unlock exciting career opportunities.

When using the app for the first time, the user will see an onboarding screen that will not be shown again after the first time. Clicking “Get Started” simply takes the user to the home page.



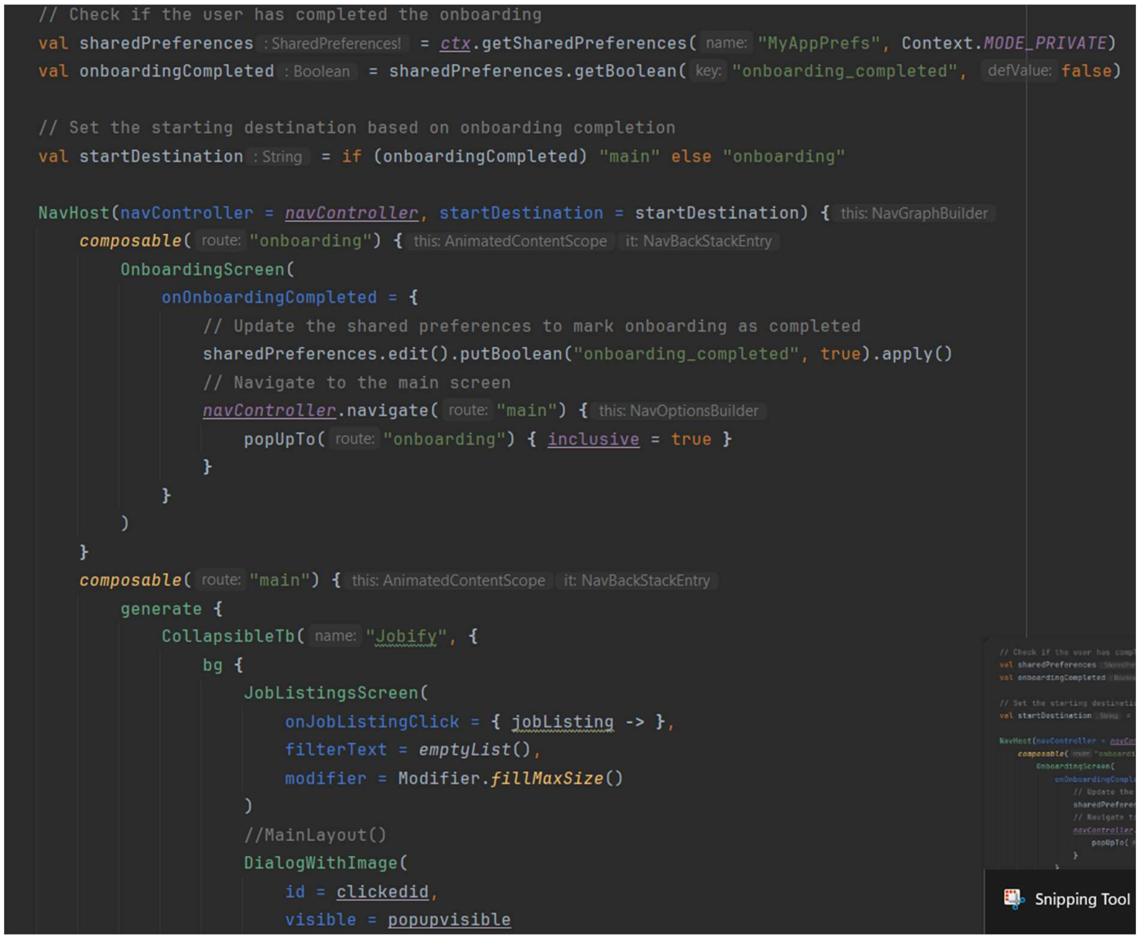
This is the home page the users will see. It features a collapsible toolbar with the app's name on it, and an overflow menu. The home screen features a list of job listings put up by users of the app, with some placeholder listings placed for testing purposes. It even features a search bar which the user can use to search for specific listings based on the title, location, company, or tags.

At the bottom is a navigation bar with clear and descriptive icons. The user can use this navigation bar to navigate through the app easily



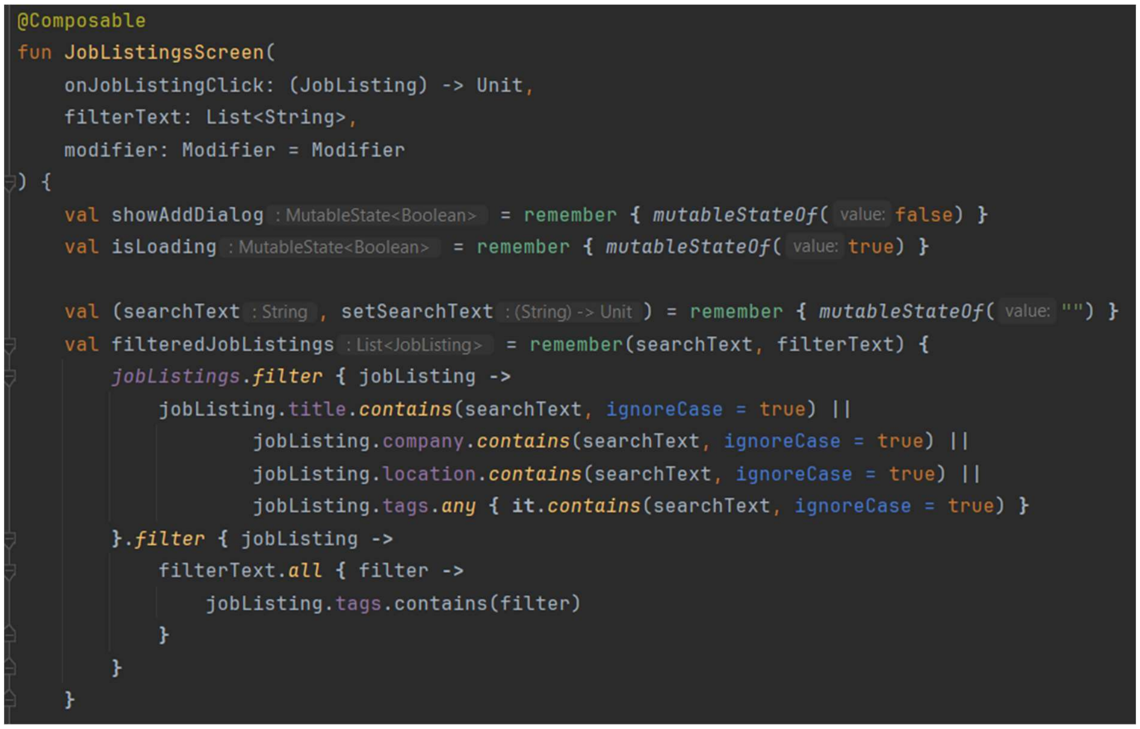
```kotlin
//This function allows the Bottom Navigation bar to be filled with the Tab icons in the Bottom bar
fun getMenuBottomItems() = mutableListOf<BottomNavigationItem>(
    BottomNavigationItem(
        title = "Listings",
        selectedIcon = Icons.Outlined.Home,
        unselectedIcon = Icons.Outlined.Home,
        route = "main"
    ),
    BottomNavigationItem(
        title = "Discussion",
        selectedIcon = Icons.Outlined.Notifications,
        unselectedIcon = Icons.Outlined.Notifications,
        route = "info"
    ),
    BottomNavigationItem(
        title = "Chat",
        selectedIcon = Icons.Outlined.~~Send~~,
        unselectedIcon = Icons.Outlined.~~Send~~,
        route = "comment"
    ),
    BottomNavigationItem(
        title = "Account",
        selectedIcon = Icons.Outlined.AccountCircle,
        unselectedIcon = Icons.Outlined.AccountCircle,
        route = "login"
    ),
)
```

This is the code allowing the Bottom Navigation bar to be filled with the icons. It uses a mutable list of BottomNavigationItem, with the title of the tab, icon, and its route.



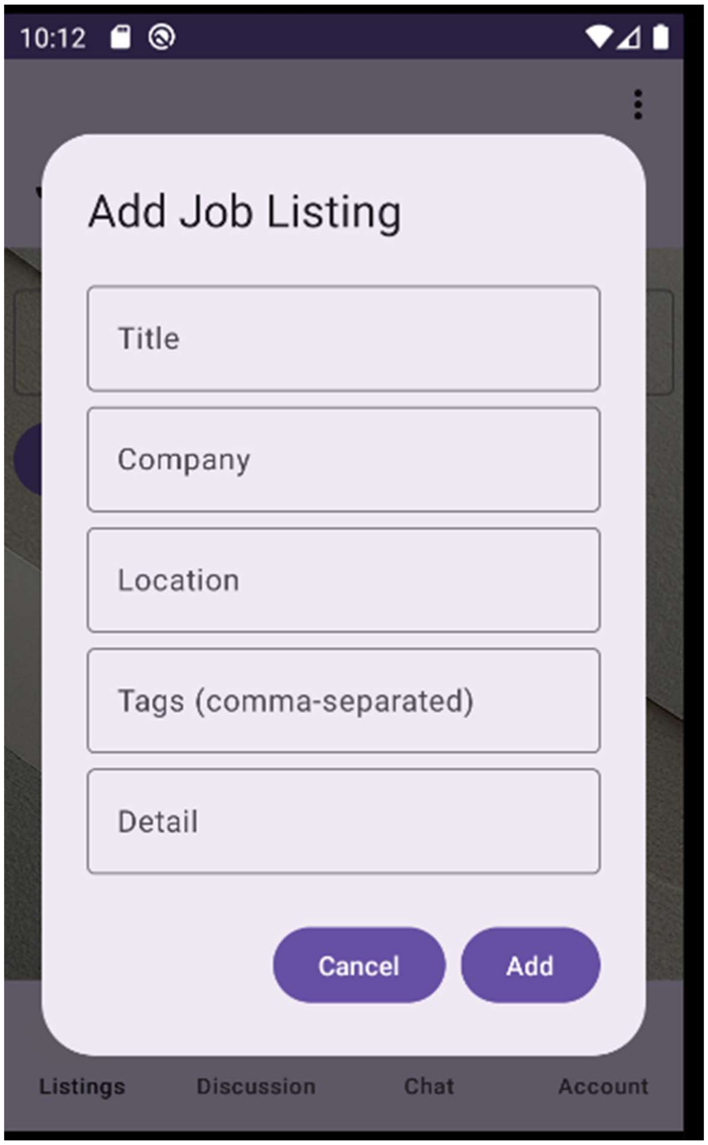
```
// Check if the user has completed the onboarding
val sharedPreferences : SharedPreferences! = ctx.getSharedPreferences( name: "MyAppPrefs", Context.MODE_PRIVATE)
val onboardingCompleted : Boolean = sharedPreferences.getBoolean( key: "onboarding_completed", defValue: false)

// Set the starting destination based on onboarding completion
val startDestination : String = if (onboardingCompleted) "main" else "onboarding"

NavHost(navController = navController, startDestination = startDestination) { this: NavGraphBuilder
    composable( route: "onboarding") { this: AnimatedContentScope it: NavBackStackEntry
        OnboardingScreen(
            onOnboardingCompleted = {
                // Update the shared preferences to mark onboarding as completed
                sharedPreferences.edit().putBoolean("onboarding_completed", true).apply()
                // Navigate to the main screen
                navController.navigate( route: "main") { this: NavOptionsBuilder
                    popUpTo( route: "onboarding") { inclusive = true }
                }
            }
        )
    }
    composable( route: "main") { this: AnimatedContentScope it: NavBackStackEntry
        generate {
            CollapsibleTb( name: "Jobify", {
                bg {
                    JobListingsScreen(
                        onJobListingClick = { jobListing -> },
                        filterText = emptyList(),
                        modifier = Modifier.fillMaxSize()
                    )
                    //MainLayout()
                    DialogWithImage(
                        id = clickedid,
                        visible = popupvisible
```

And this is part of the code that is behind the navigation. It also shows part of the logic behind the onboarding screen, using Shared Preferences for it. It uses NavHost in order to navigate between tabs seamlessly, using the routes specified. The onboarding screen is part of it, and is the first shown, before never being shown again after it is shown once, instead navigating to “main”, the home screen.

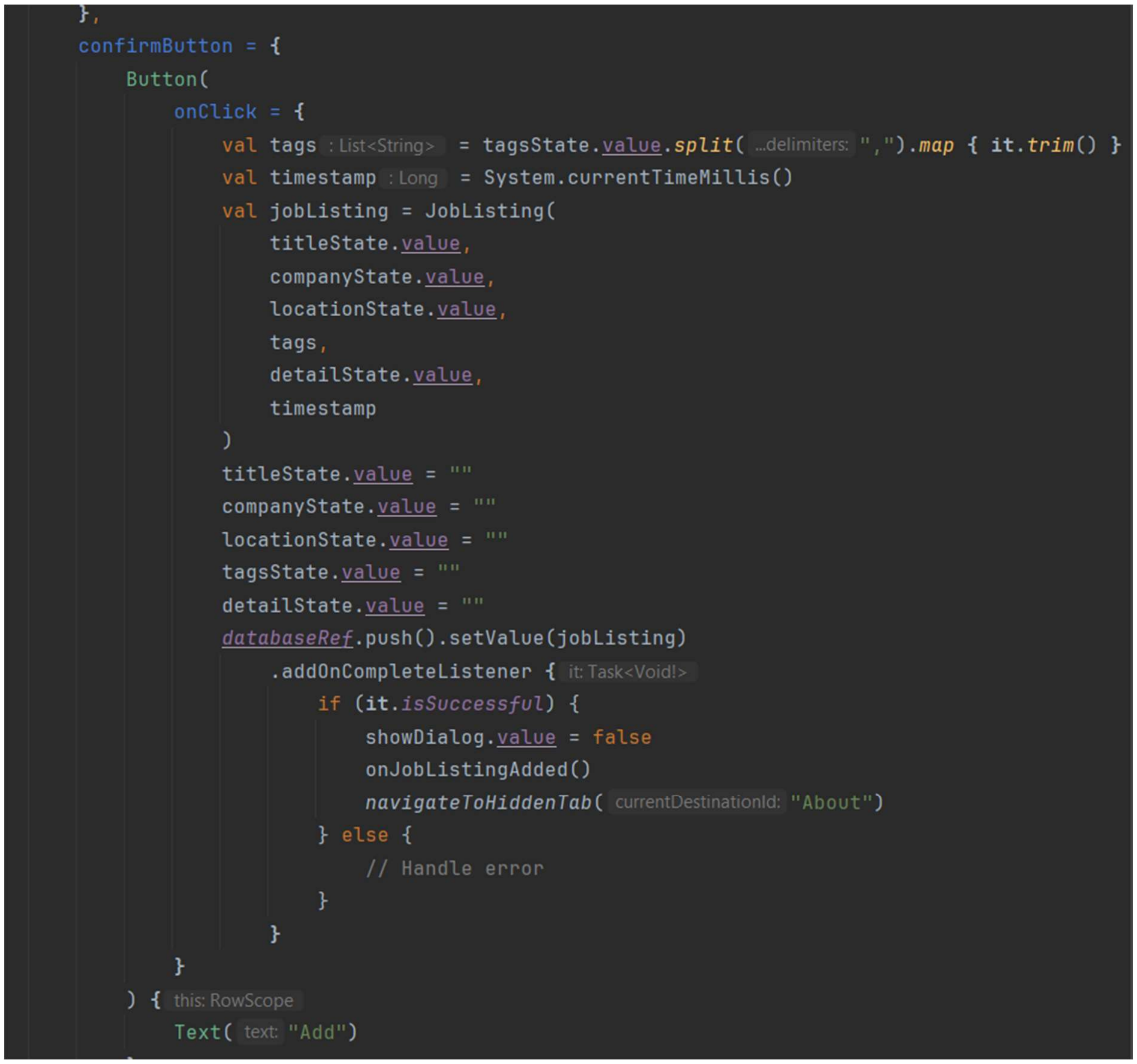


```kotlin
@Composable
fun JobListingsScreen(
    onJobListingClick: (JobListing) -> Unit,
    filterText: List<String>,
    modifier: Modifier = Modifier
) {
    val showAddDialog : MutableState<Boolean>  = remember { mutableStateOf( value: false) }
    val isLoading : MutableState<Boolean>  = remember { mutableStateOf( value: true) }

    val (searchText : String , setSearchText : (String) -> Unit ) = remember { mutableStateOf( value: "") }
    val filteredJobListings : List<JobListing>  = remember(searchText, filterText) {
        jobListings.filter { jobListing ->
            jobListing.title.contains(searchText, ignoreCase = true) ||
                    jobListing.company.contains(searchText, ignoreCase = true) ||
                    jobListing.location.contains(searchText, ignoreCase = true) ||
                    jobListing.tags.any { it.contains(searchText, ignoreCase = true) }
        }.filter { jobListing ->
            filterText.all { filter ->
                jobListing.tags.contains(filter)
            }
        }
    }
```

This is the code responsible for the searching of the job listings. Note that it uses mutable state variables in order to update the listings live as the user is typing in a query.

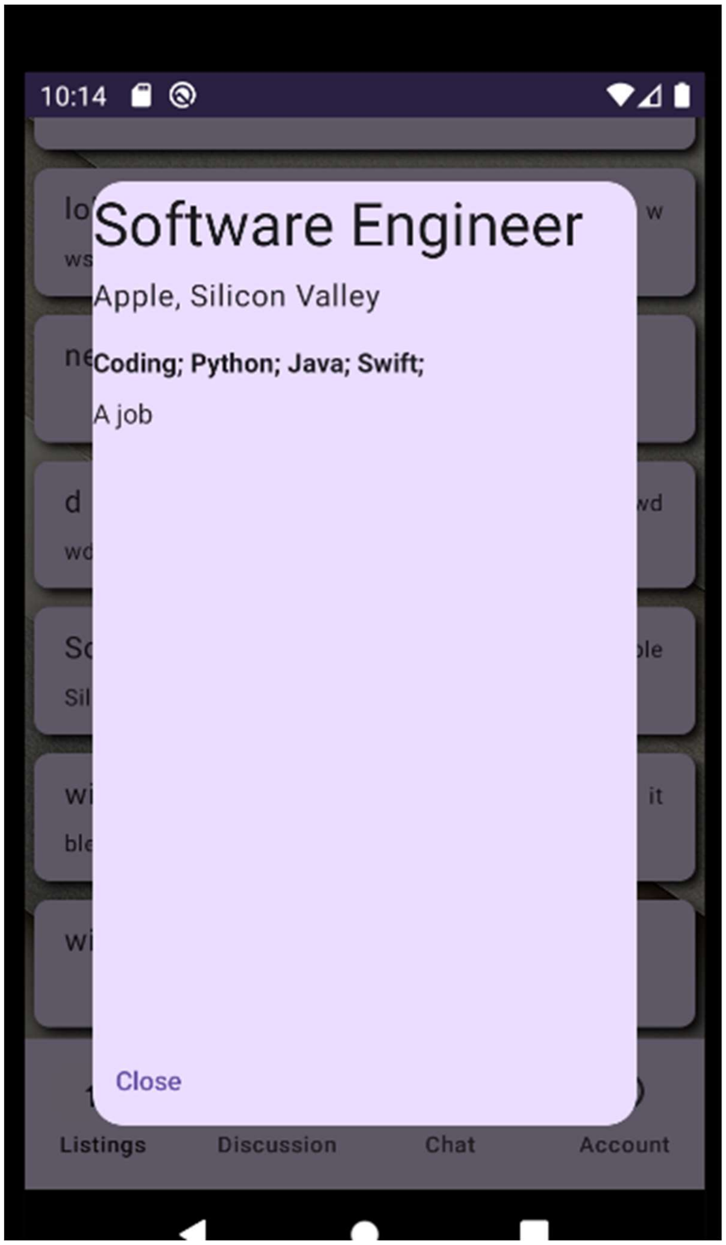


Once the user clicks on the button labelled “Add Job Listing”, the user will see a dialog box. The user can then fill in the fields on the dialog box. Upon doing so, the user’s job listing will be added to the screen, and will appear at the very top, as the listings are sorted such that the most recent ones will appear first. The user can also cancel the dialog if they change their minds.



```
},
confirmButton = {
    Button(
        onClick = {
            val tags : List<String>  = tagsState.value.split( ...delimiters: ",").map { it.trim() }
            val timestamp : Long  = System.currentTimeMillis()
            val jobListing = JobListing(
                titleState.value,
                companyState.value,
                locationState.value,
                tags,
                detailState.value,
                timestamp
            )
            titleState.value = ""
            companyState.value = ""
            locationState.value = ""
            tagsState.value = ""
            detailState.value = ""
            databaseRef.push().setValue(jobListing)
                .addOnCompleteListener { it: Task<Void!>
                    if (it.isSuccessful) {
                        showDialog.value = false
                        onJobListingAdded()
                        navigateToHiddenTab( currentDestinationId: "About")
                    } else {
                        // Handle error
                    }
                }
        }
    ) { this: RowScope
        Text( text: "Add")
```

This is the responsible code. Note the usage of firebase database in adding a new job listing. All the data from the Text Fields are taken and put into Firebase Database, and then updated in the screen.

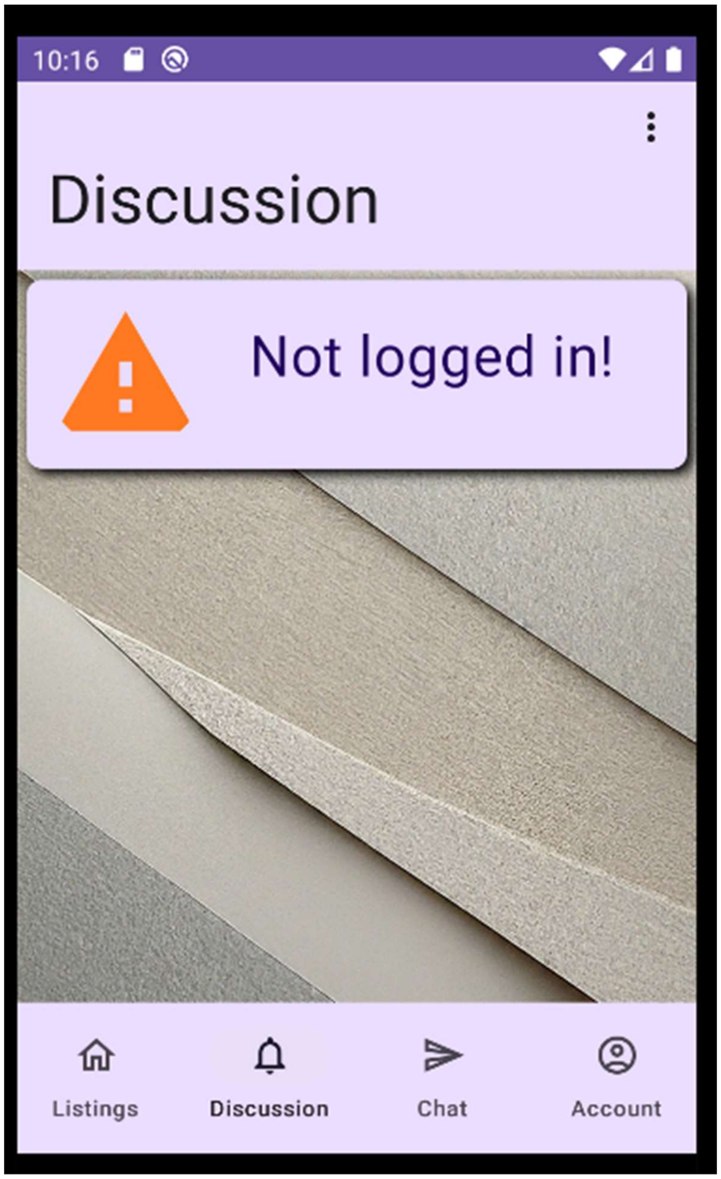

Software Engineer

Apple, Silicon Valley

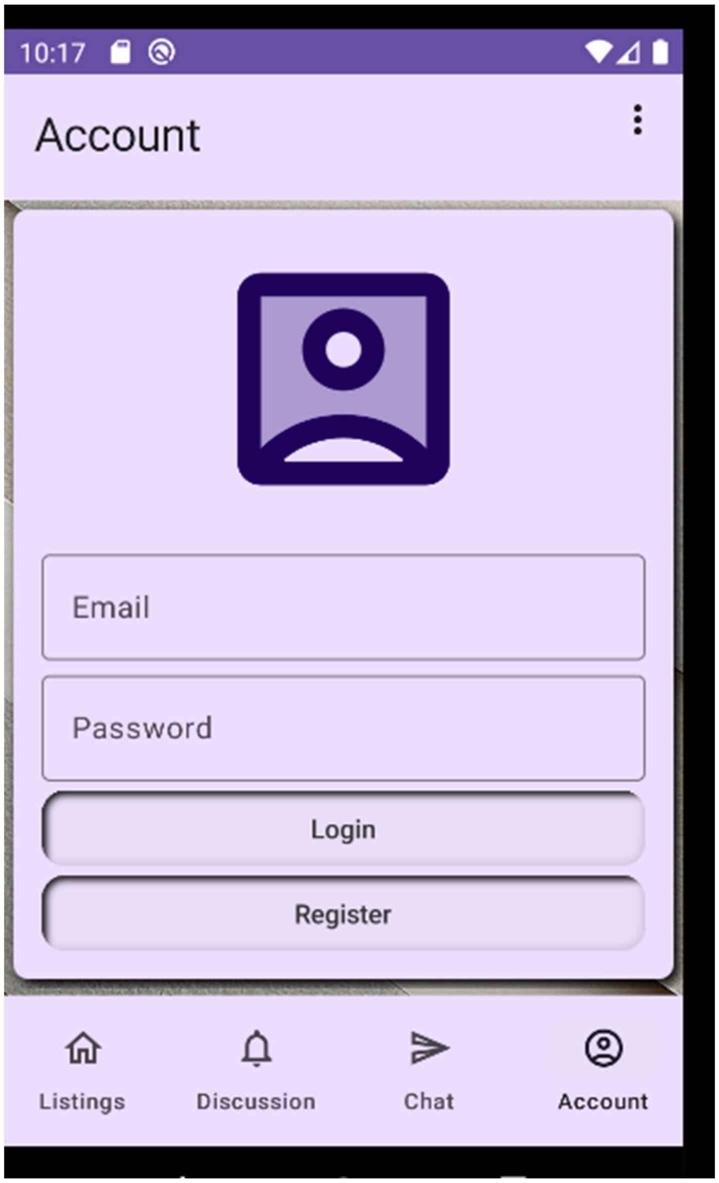
Coding; Python; Java; Swift;

A job

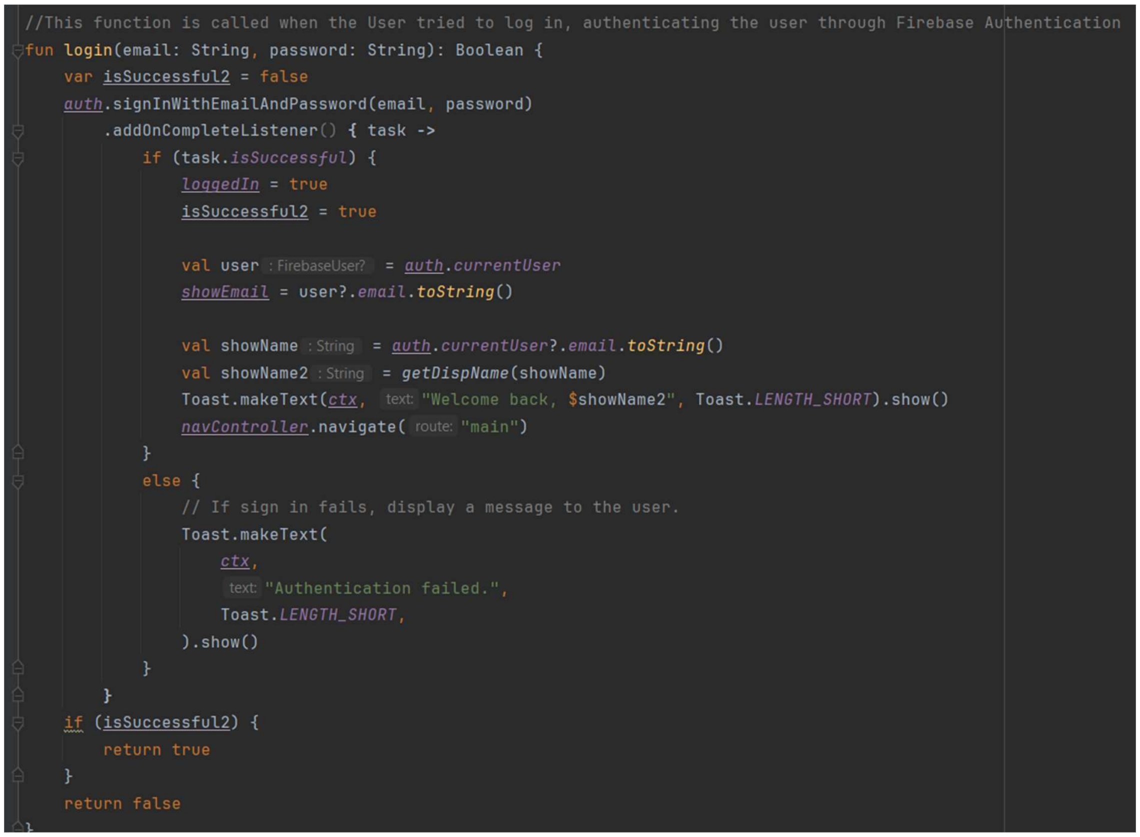
The user will see the above when they click on a job listing. A dialog box opens up, with additional details about the job. Generally, contact information would be included in the job details. The interface is set up nicely, with the most important information having the biggest font weight, and the details having a smaller font weight. There is also a button to close the dialog at the bottom.



This is the Discussion tab. As the user is not logged in at this point, this is the message that will show. The same goes for the Chat tab. I chose not to implement this on the Listings tab as being logged in was not necessary for that tab, but is for the Discussion and Chat tab.

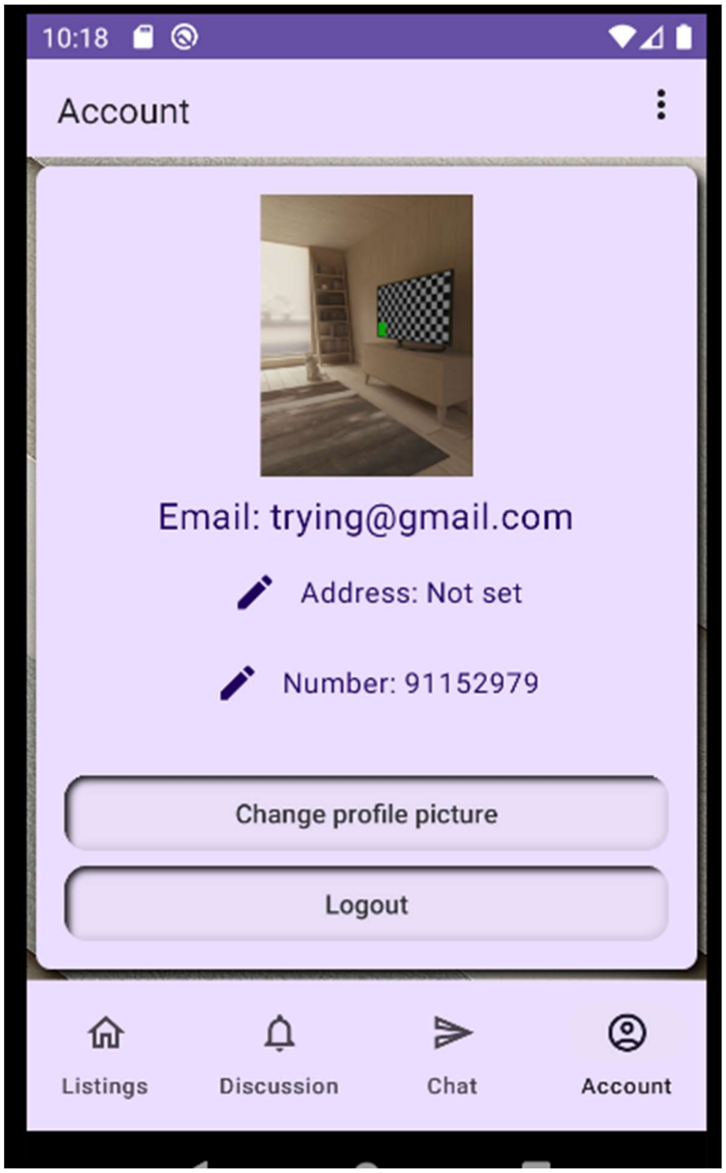


This is the Account tab. Currently, it shows a screen for the user to either log in or register, using their emails. If a user with the email already exists, the user will not be allowed to register, and will get a Toast indicating so. Similarly, the user cannot log in with an email that does not exist.

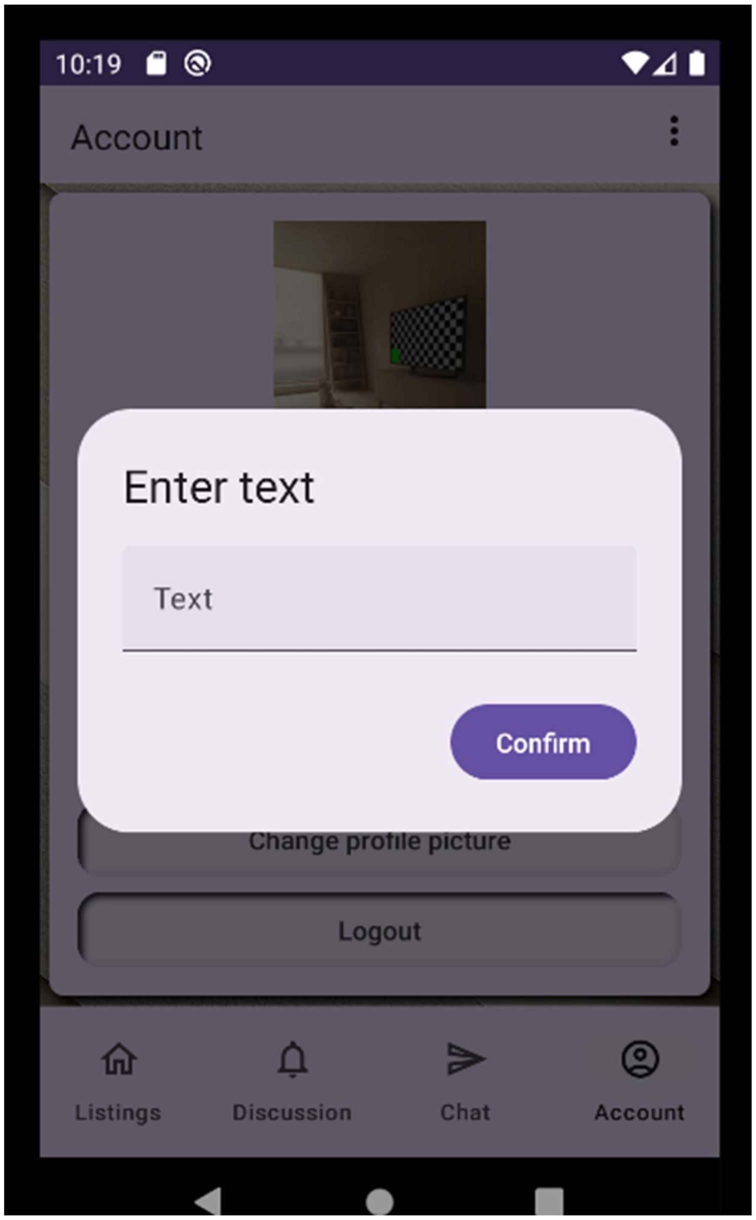


```kotlin
//This function is called when the User tried to log in, authenticating the user through Firebase Authentication
fun login(email: String, password: String): Boolean {
    var isSuccessful2 = false
    auth.signInWithEmailAndPassword(email, password)
        .addOnCompleteListener() { task ->
            if (task.isSuccessful) {
                loggedIn = true
                isSuccessful2 = true

                val user : FirebaseUser?  = auth.currentUser
                showEmail = user?.email.toString()

                val showName : String  = auth.currentUser?.email.toString()
                val showName2 : String  = getDispName(showName)
                Toast.makeText(ctx,  text: "Welcome back, $showName2", Toast.LENGTH_SHORT).show()
                navController.navigate( route: "main")
            }
            else {
                // If sign in fails, display a message to the user.
                Toast.makeText(
                    ctx,
                     text: "Authentication failed.",
                    Toast.LENGTH_SHORT,
                ).show()
            }
        }
    if (isSuccessful2) {
        return true
    }
    return false
}
```

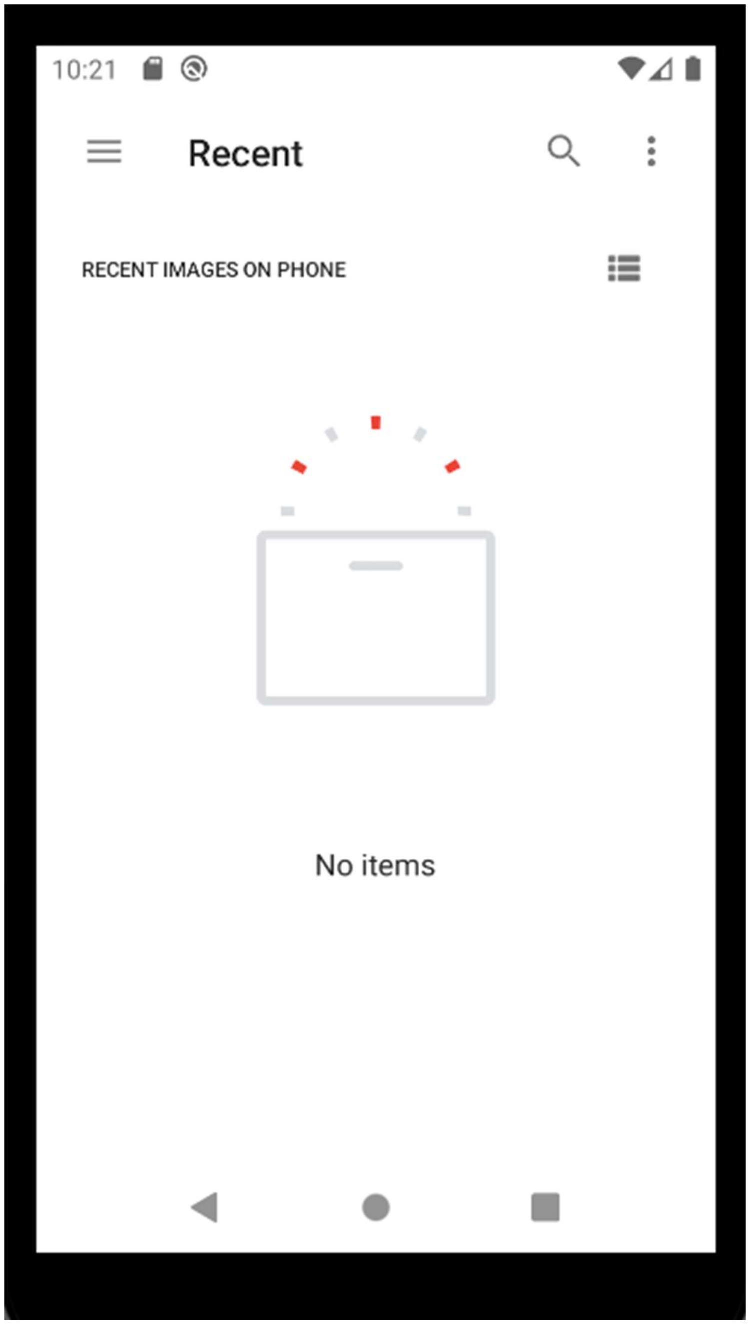
This is the code for the logic of logging in, depending heavily on Firebase Authentication. The logic for registering is very similar as well. As signInWithEmailAndPassword() is a built in function by Firebase Authentication, I simply used it with the user's email and password, without the need for declaring my own function.



This is the screen that is shown after the user is logged in. There is a profile picture, an address, a number, and an option to change their profile picture or log out.

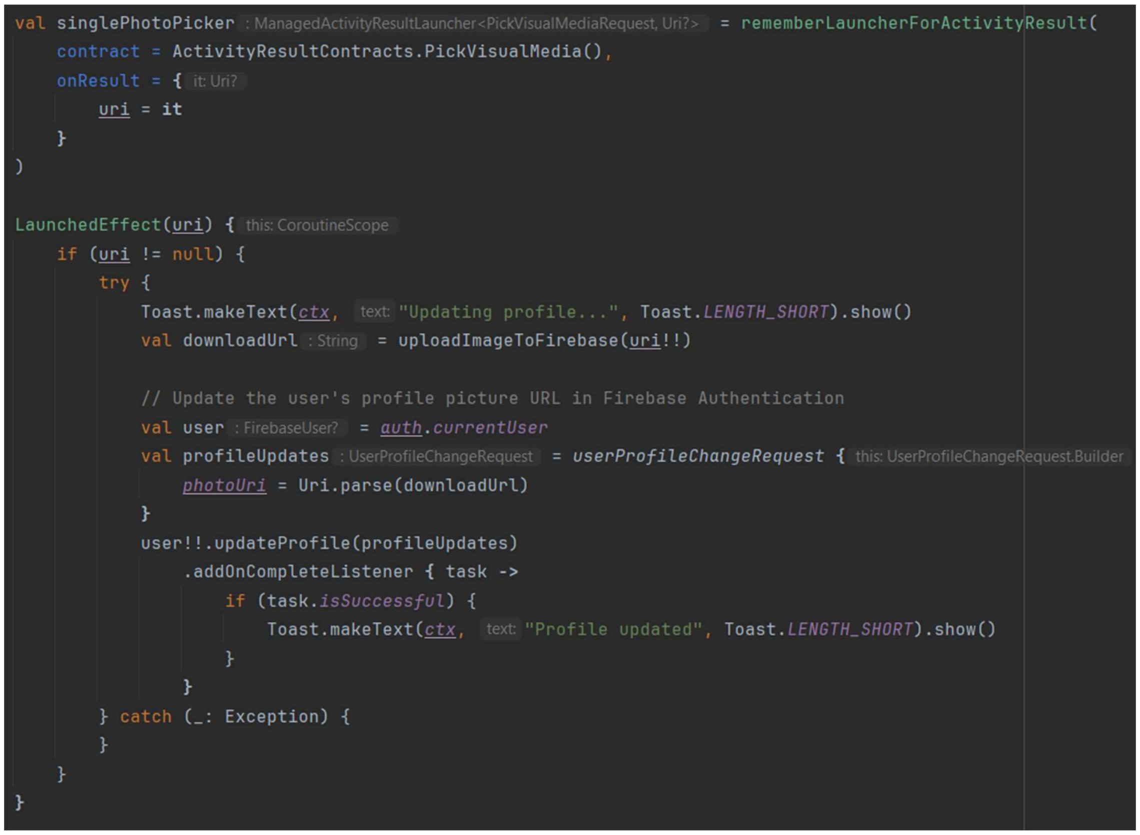


When clicking on the Pen icon beside the address or the number, the user will see the above. However, the dialog does output a warning Toast if the user inputs a non-numerical string for the number dialog.



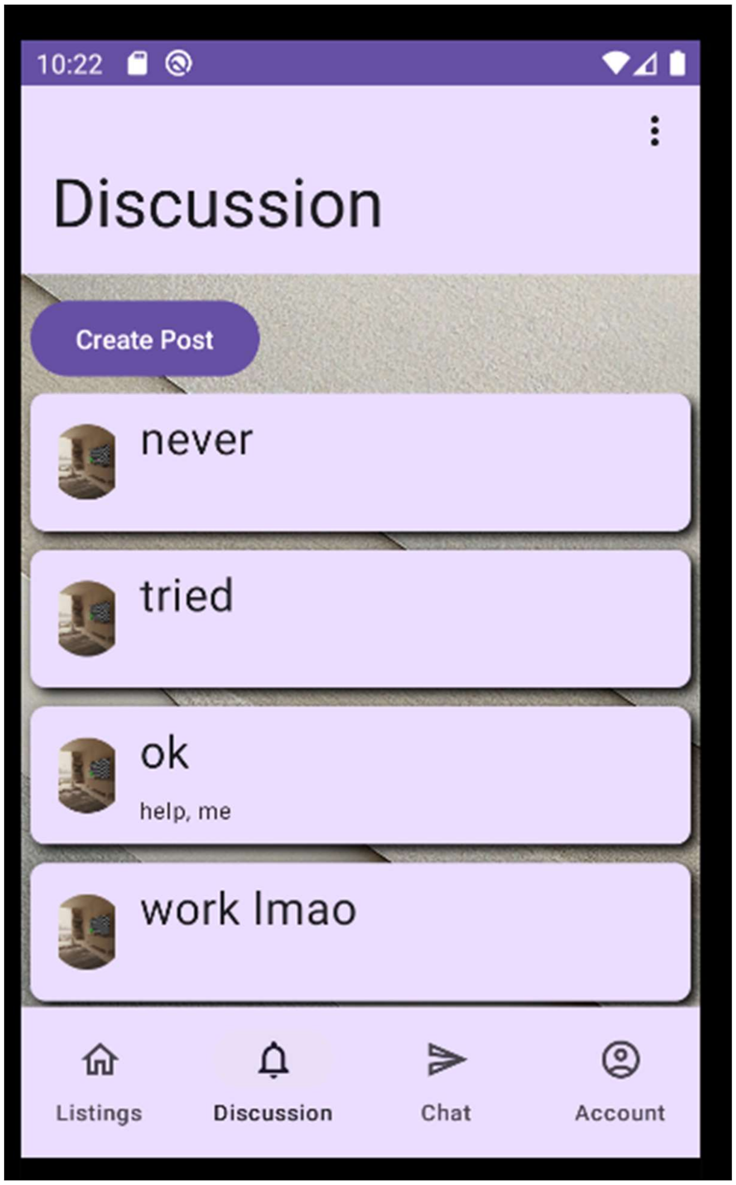
The user will see the above when clicking on the “Change profile picture” button. The user can select a picture, which will be uploaded into firebase and updated for the user.

Upon clicking on the “Logout” button, the user will simply be logged out and the app will navigate back to the home screen.

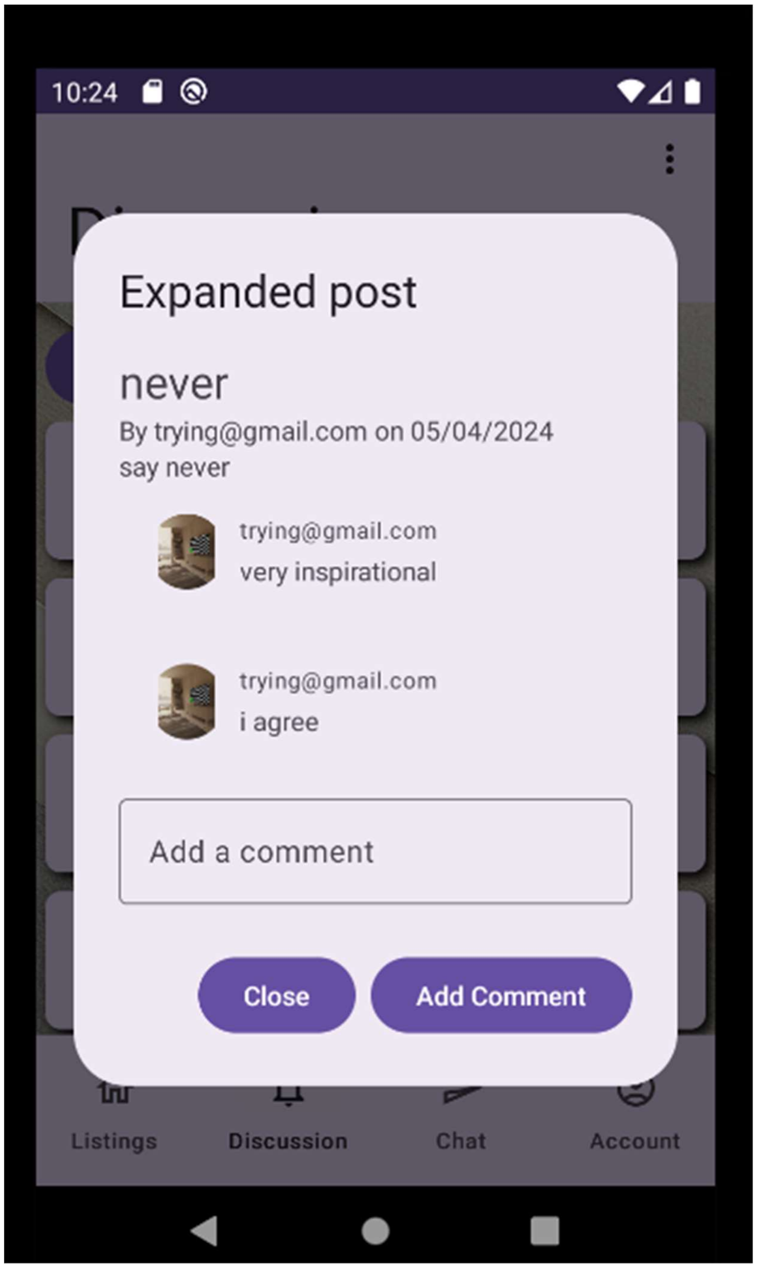


```
val singlePhotoPicker : ManagedActivityResultLauncher<PickVisualMediaRequest, Uri?> = rememberLauncherForActivityResult(
    contract = ActivityResultContracts.PickVisualMedia(),
    onResult = { it: Uri?
        uri = it
    }
)

LaunchedEffect(uri) { this: CoroutineScope
    if (uri != null) {
        try {
            Toast.makeText(ctx,  text: "Updating profile...", Toast.LENGTH_SHORT).show()
            val downloadUrl : String = uploadImageToFirebase(uri!!)

            // Update the user's profile picture URL in Firebase Authentication
            val user : FirebaseUser? = auth.currentUser
            val profileUpdates : UserProfileChangeRequest = userProfileChangeRequest { this: UserProfileChangeRequest.Builder
                photoUri = Uri.parse(downloadUrl)
            }
            user!!.updateProfile(profileUpdates)
                .addOnCompleteListener { task ->
                    if (task.isSuccessful) {
                        Toast.makeText(ctx,  text: "Profile updated", Toast.LENGTH_SHORT).show()
                    }
                }
        } catch (_: Exception) {
        }
    }
}
```

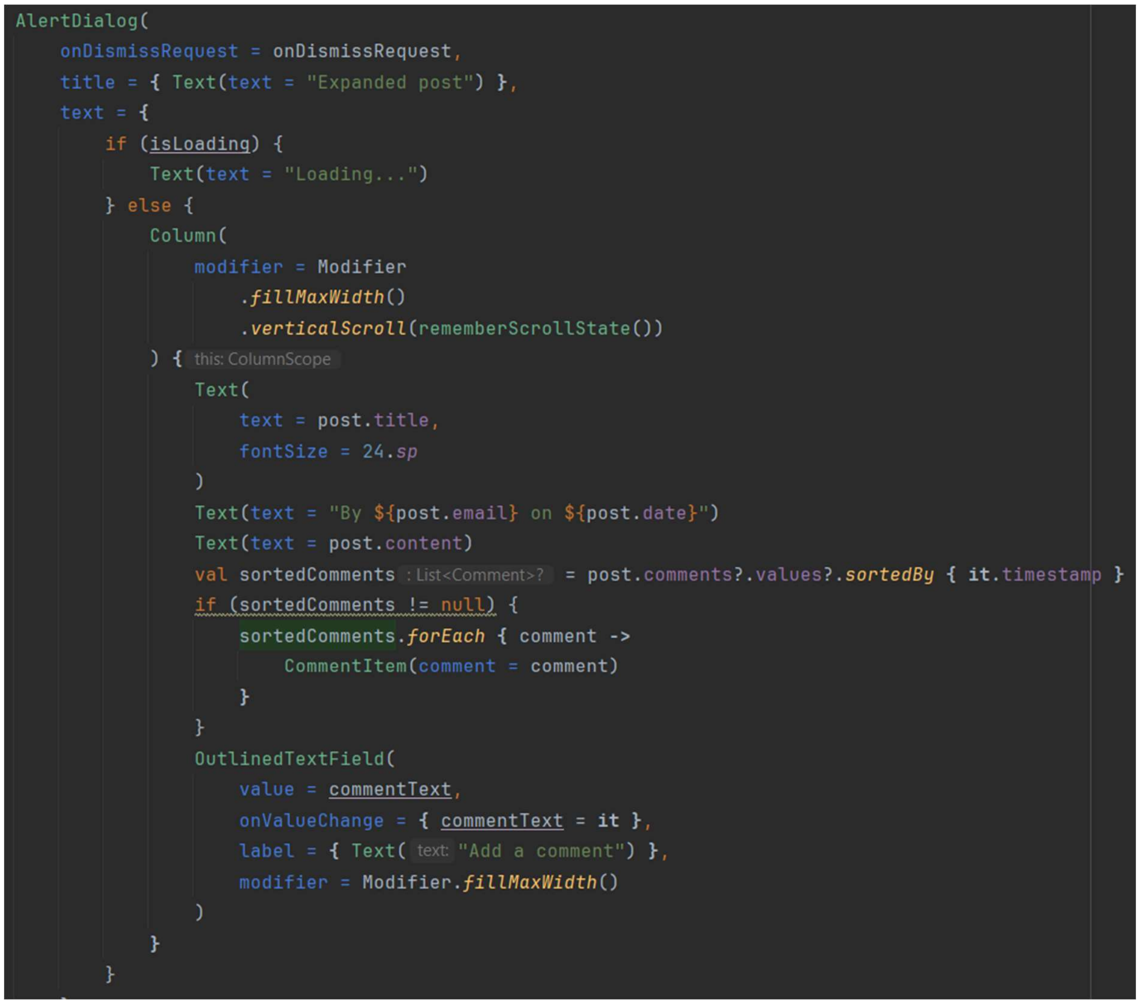
This is the code for the changing of the profile picture for the user. Note the usage of Firebase Authentication and Firebase Storage, to store the reference to the image and the actual image respectively. Here, LaunchedEffect, which is under Coroutines, is used, and so is Single Photo Picker for picking a visual media from the phone’s local storage.



This is the discussion tab, now accessible to the user as they are logged in. The user can see several posts, with an image. These posts are by other users, and the image is the other user's profile picture.

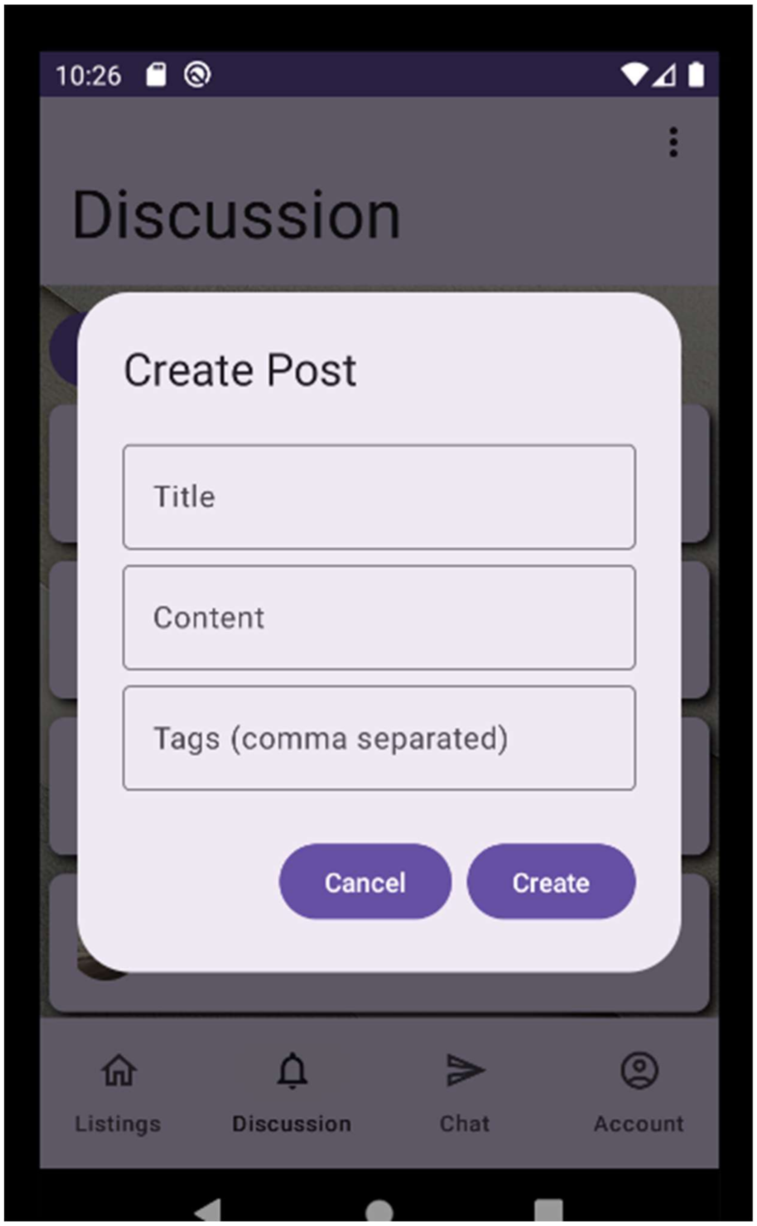


This is what the user will see when they click on one of the posts. The post is expanded into a dialog, with the details ("say never"), and even comments by other users. The user themselves can add a comment, using the "Add a comment" text field and the "Add Comment" button. The comments are sorted such that the most recent one appears at the bottom.

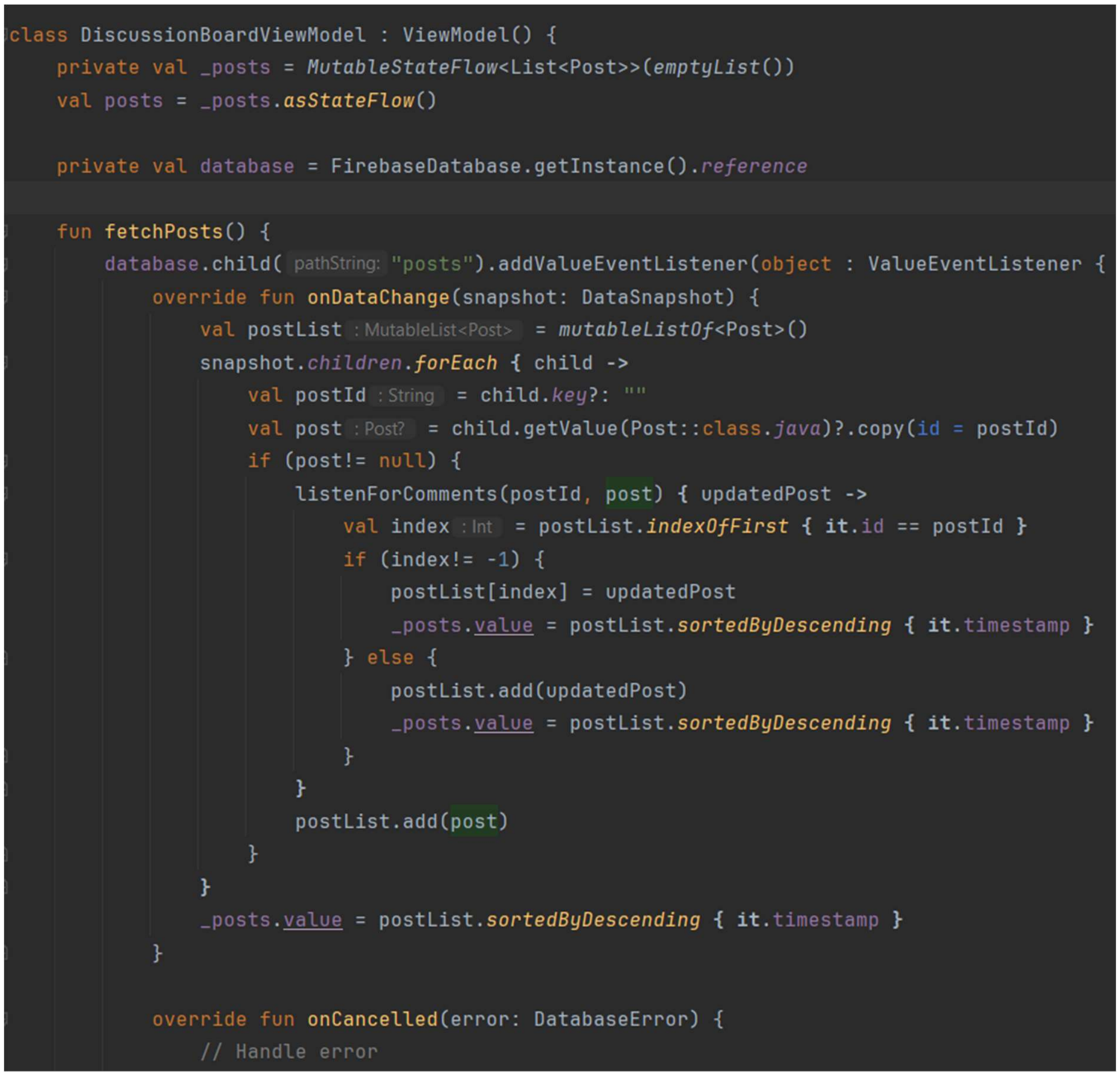


```
AlertDialog(
    onDismissRequest = onDismissRequest,
    title = { Text(text = "Expanded post") },
    text = {
        if (isLoading) {
            Text(text = "Loading...")
        } else {
            Column(
                modifier = Modifier
                    .fillMaxWidth()
                    .verticalScroll(rememberScrollState())
            ) { this: ColumnScope
                Text(
                    text = post.title,
                    fontSize = 24.sp
                )
                Text(text = "By ${post.email} on ${post.date}")
                Text(text = post.content)
                val sortedComments : List<Comment>? = post.comments?.values?.sortedBy { it.timestamp }
                if (sortedComments != null) {
                    sortedComments.forEach { comment ->
                        CommentItem(comment = comment)
                    }
                }
                OutlinedTextField(
                    value = commentText,
                    onValueChange = { commentText = it },
                    label = { Text( text: "Add a comment") },
                    modifier = Modifier.fillMaxWidth()
                )
            }
        }
```

This is the code for the Dialog box, along with the comments. Note the sorting of comments. This is a decently simple Jetpack Compose function for an AlertDialog.

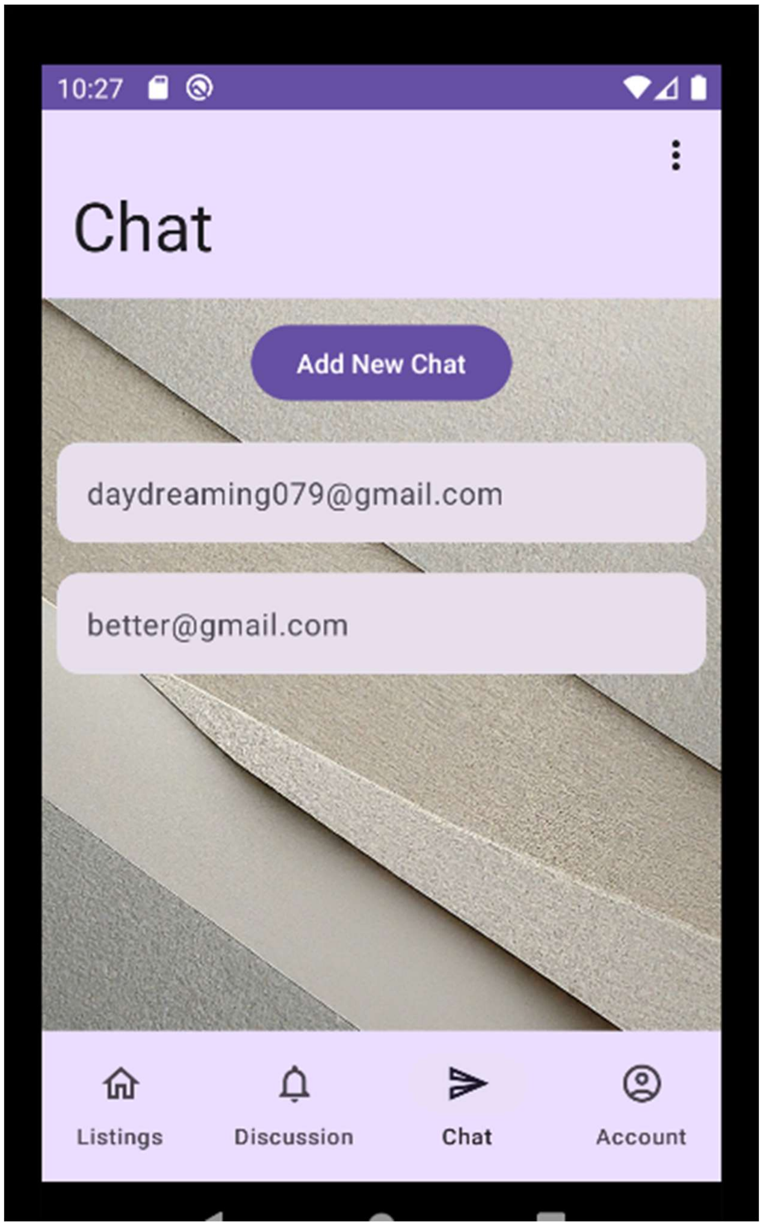


This is what the user will see when they click on the "Create Post" button. They get to create a post with a title, content and even tags. The tags will be shown in the Post when the Posts are in the Discussion screen, but not when the Post is expanded. The posts are set up such that the most recent post will be shown at the very top.

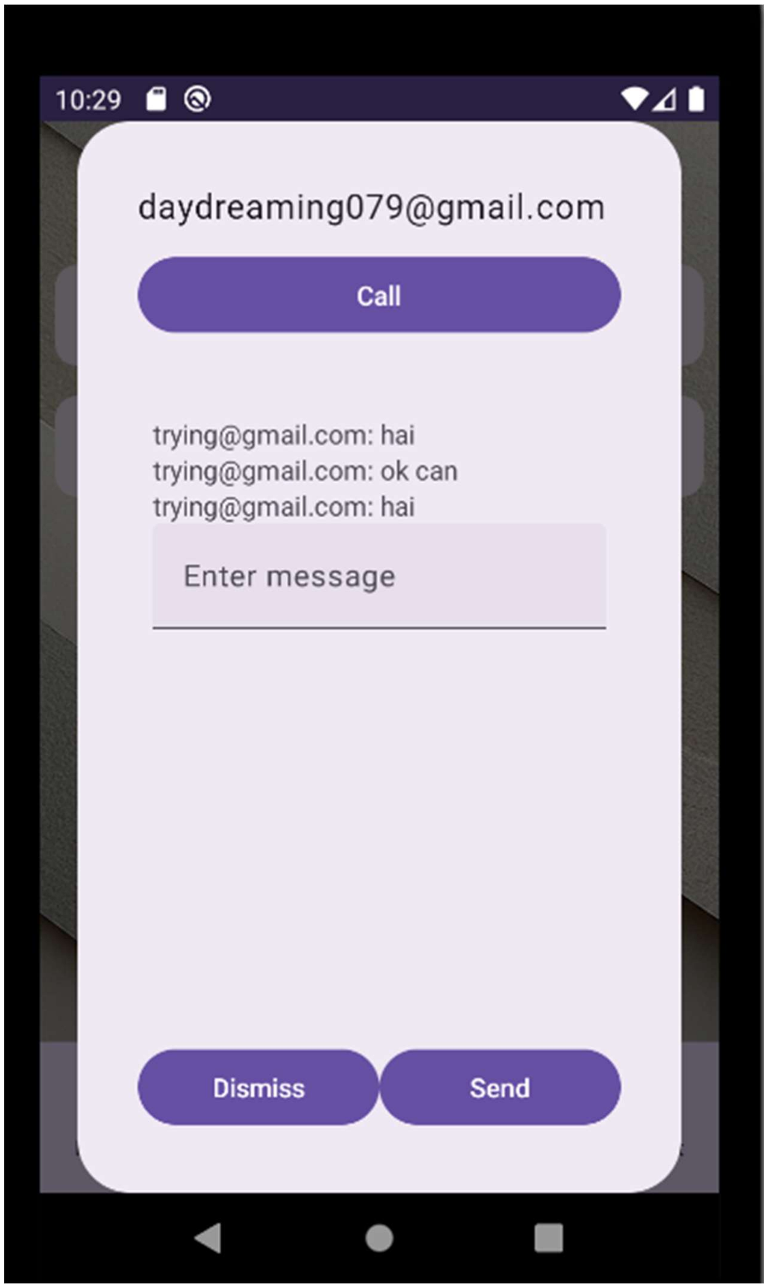


```
class DiscussionBoardViewModel : ViewModel() {
    private val _posts = MutableStateFlow<List<Post>>(emptyList())
    val posts = _posts.asStateFlow()

    private val database = FirebaseDatabase.getInstance().reference

    fun fetchPosts() {
        database.child( pathString: "posts").addValueEventListener(object : ValueEventListener {
            override fun onDataChange(snapshot: DataSnapshot) {
                val postList : MutableList<Post> = mutableListOf<Post>()
                snapshot.children.forEach { child ->
                    val postId : String = child.key?: ""
                    val post : Post? = child.getValue(Post::class.java)?.copy(id = postId)
                    if (post!= null) {
                        listenForComments(postId, post) { updatedPost ->
                            val index : Int = postList.indexOfFirst { it.id == postId }
                            if (index!= -1) {
                                postList[index] = updatedPost
                                _posts.value = postList.sortedByDescending { it.timestamp }
                            } else {
                                postList.add(updatedPost)
                                _posts.value = postList.sortedByDescending { it.timestamp }
                            }
                        }
                        postList.add(post)
                    }
                }
                _posts.value = postList.sortedByDescending { it.timestamp }
            }

            override fun onCancelled(error: DatabaseError) {
                // Handle error
```

This is a key feature of the logic of the discussion board. It uses a Discussion Board View Model to link it to Firebase effectively and thus get and upload the data from and to Firebase respectively. This makes the code much easier to work with and understand, and easily links the code to Firebase.



This is what the user will see upon clicking on the “Chat” tab. There are already some preexisting chats from previous testing.

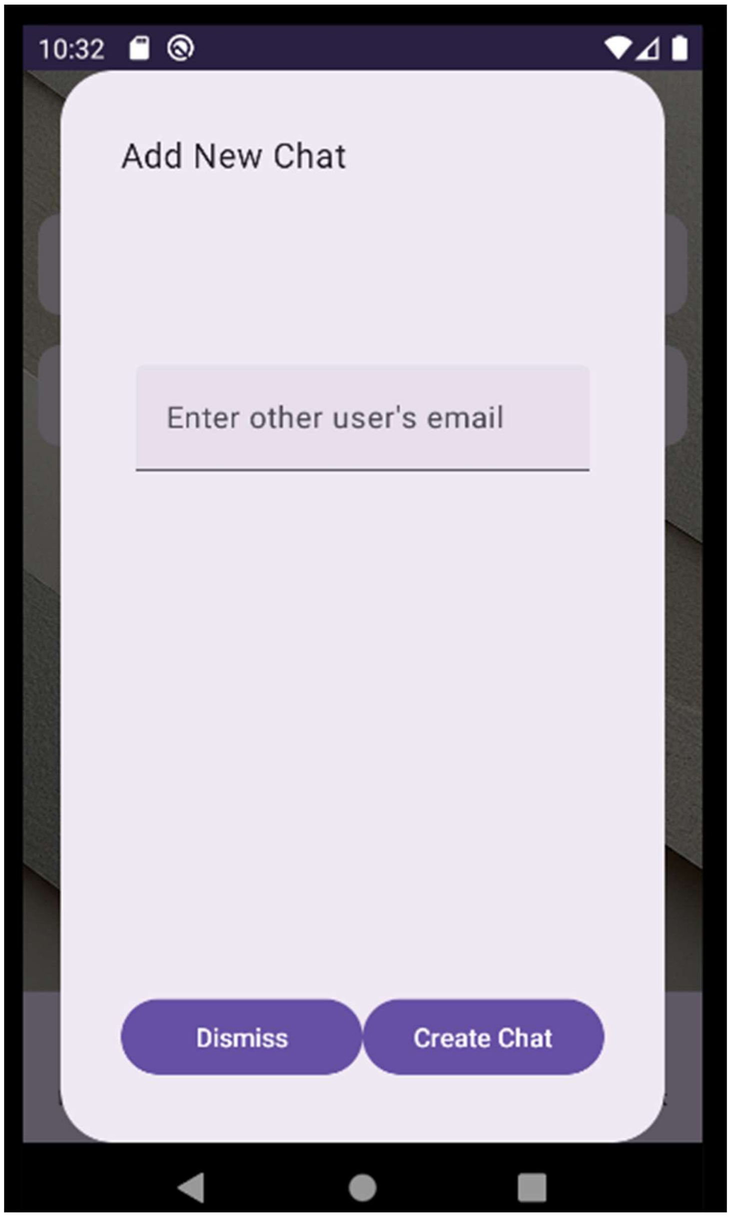

daydreaming079@gmail.com

trying@gmail.com: hai
trying@gmail.com: ok can
trying@gmail.com: hai

This is what the user will see when they click on a chat. It opens a dialog with the email, a call button, and the chat conversation. Using this, users can communicate with each other on the app itself, without the need for an external app. The usage of this is quite simple and self-explanatory to users. The user simply needs to type in a message into the Text Field, and click send. They can also dismiss the dialog box to exit the conversation.

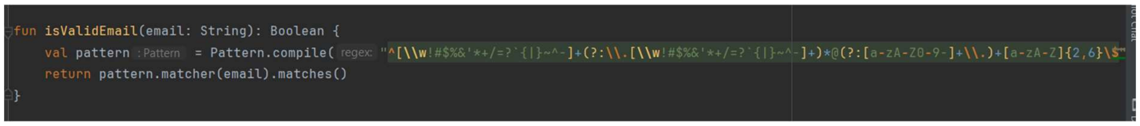
The user can even call the user they are chatting with! This is, of course, provided that the other user has set a number, and if they have not, a warning Toast will be outputted.

The call takes place on an external app, and the “Call” button simply takes the user to the external app with the number already placed into the dial pad.

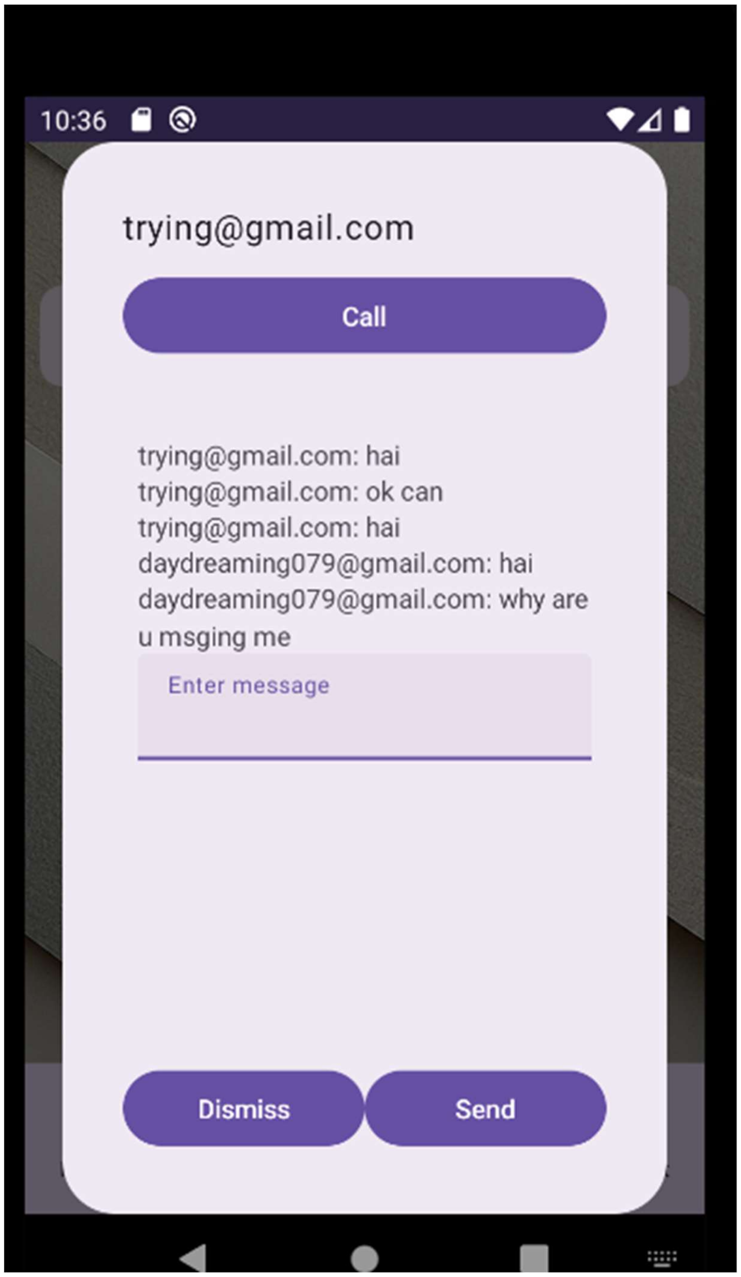


This is what the user will see when they click on the button “Add New Chat”. The user will be able to enter the email of the user they wish to chat with, and click on the button “Create Chat” in order to start the conversation, or “Dismiss” if they change their minds. The email is used as it makes it much more convenient, as it has been seen that the email of other users is, most of the time, the only contact information they have, as seen earlier from the “Discussion” tabs, where user’s emails would be below posts they made of above comments they make.

Note that if the user enters an invalid email, a warning Toast is outputted. This is done using a regex, shown below



```
fun isValidEmail(email: String): Boolean {
    val pattern :Pattern  = Pattern.compile( regex: "^[\\w!#$%&'*+/=?`{|}~^-]+(?:\\.[\\w!#$%&'*+/=?`{|}~^-]+)*@(?:[a-zA-Z0-9-]+\\.)+[a-zA-Z]{2,6}\$"
    return pattern.matcher(email).matches()
}
```

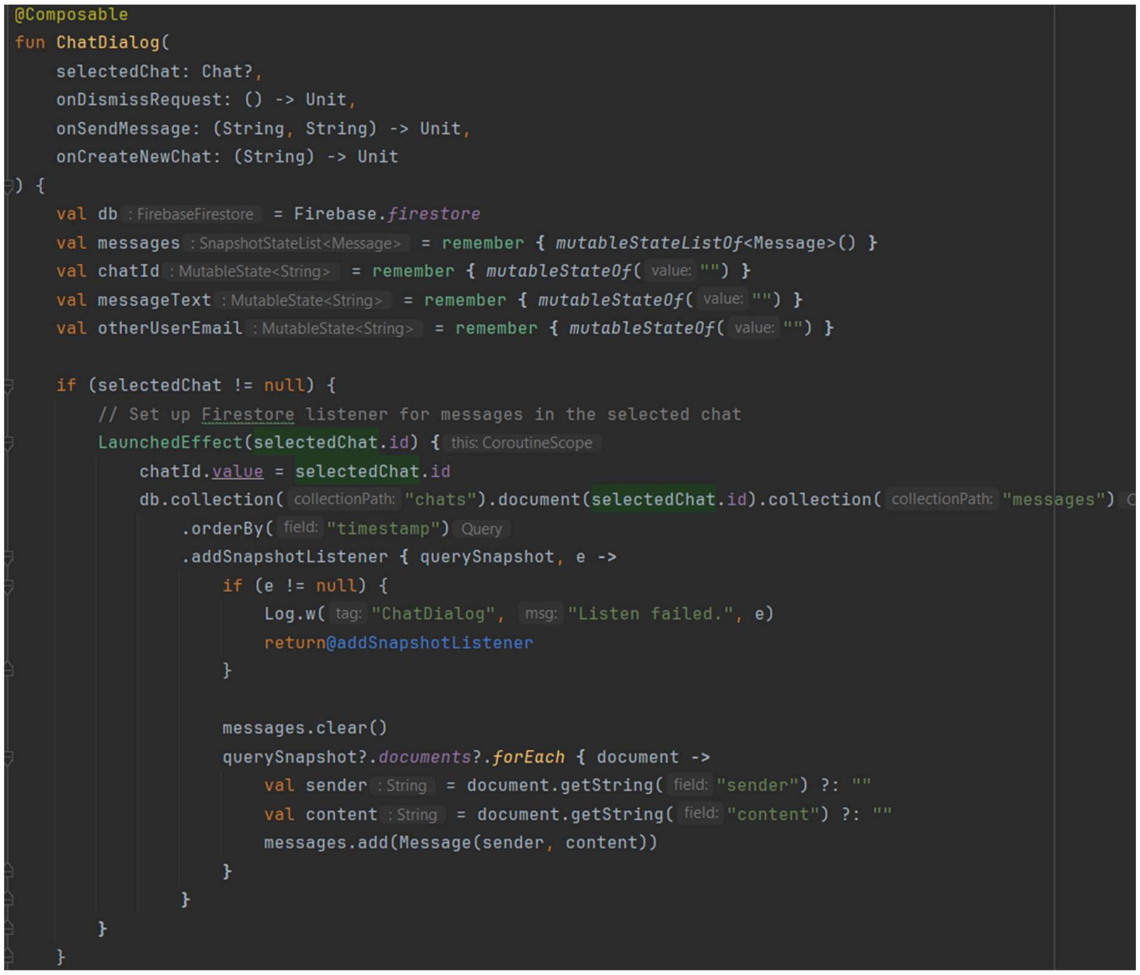


10:36

trying@gmail.com

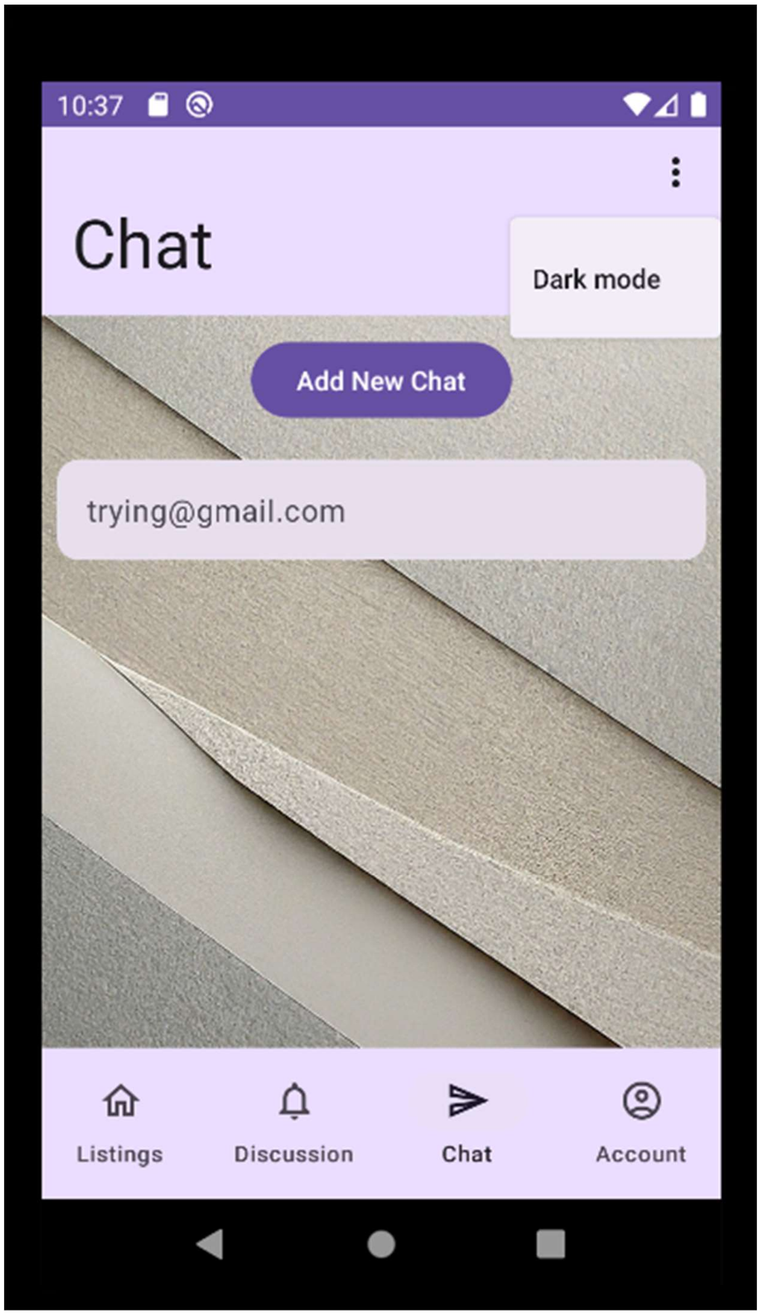
trying@gmail.com: hai
trying@gmail.com: ok can
trying@gmail.com: hai
daydreaming079@gmail.com: hai
daydreaming079@gmail.com: why are u msging me

This is the chat conversation from the perspective of the other user, daydreaming079@gmail.com

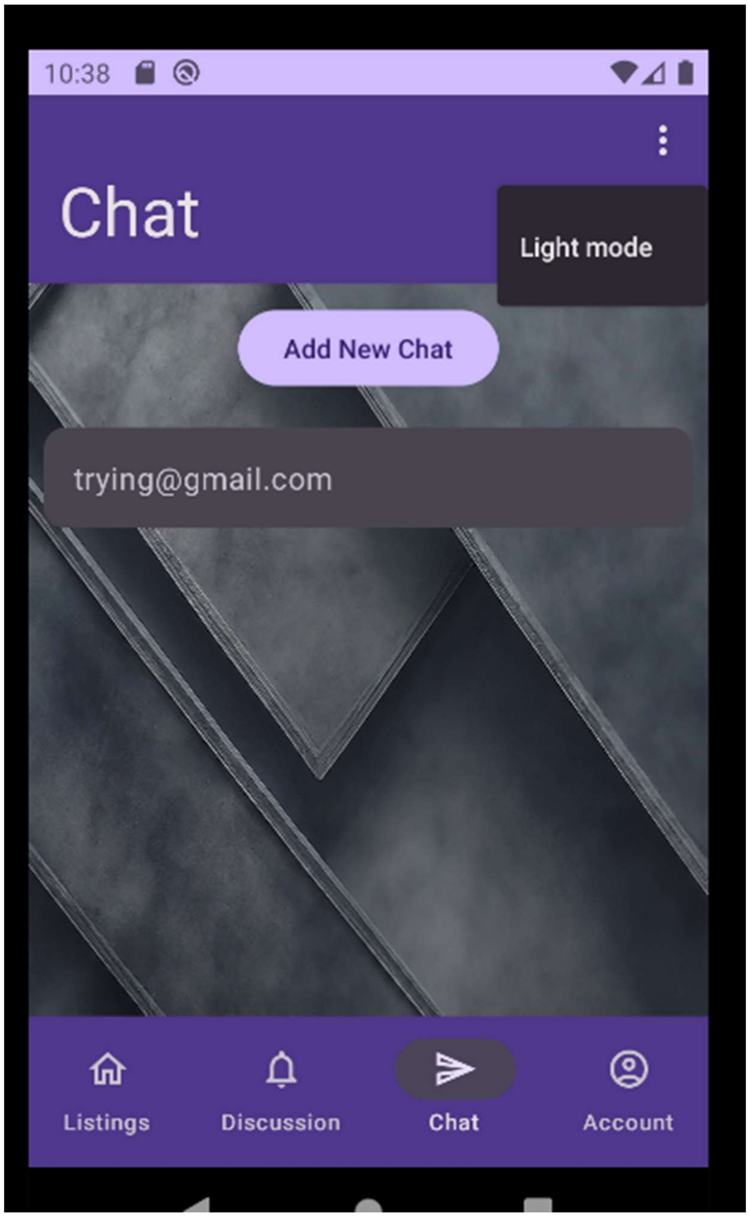


```
@Composable
fun ChatDialog(
    selectedChat: Chat?,
    onDismissRequest: () -> Unit,
    onSendMessage: (String, String) -> Unit,
    onCreateNewChat: (String) -> Unit
) {
    val db : FirebaseFirestore  = Firebase.firestore
    val messages : SnapshotStateList<Message>  = remember { mutableStateListOf<Message>() }
    val chatId : MutableState<String>  = remember { mutableStateOf( value: "") }
    val messageText : MutableState<String>  = remember { mutableStateOf( value: "") }
    val otherUserEmail : MutableState<String>  = remember { mutableStateOf( value: "") }

    if (selectedChat != null) {
        // Set up Firestore listener for messages in the selected chat
        LaunchedEffect(selectedChat.id) { this: CoroutineScope
            chatId.value = selectedChat.id
            db.collection( collectionPath: "chats").document(selectedChat.id).collection( collectionPath: "messages") C
                .orderBy( field: "timestamp") Query
                .addSnapshotListener { querySnapshot, e ->
                    if (e != null) {
                        Log.w( tag: "ChatDialog",  msg: "Listen failed.", e)
                        return@addSnapshotListener
                    }

                    messages.clear()
                    querySnapshot?.documents?.forEach { document ->
                        val sender : String  = document.getString( field: "sender") ?: ""
                        val content : String  = document.getString( field: "content") ?: ""
                        messages.add(Message(sender, content))
                    }
                }
        }
    }
```

This is the part of the code for the Chat conversation. Note the use of Firebase Firestore instead of Firebase Database. Firestore is much preferable due to its speed at updating the data, and its extra available storage which is needed especially for a storage-heavy feature such as chatting. Note the usage of many mutable state variables for live updating.

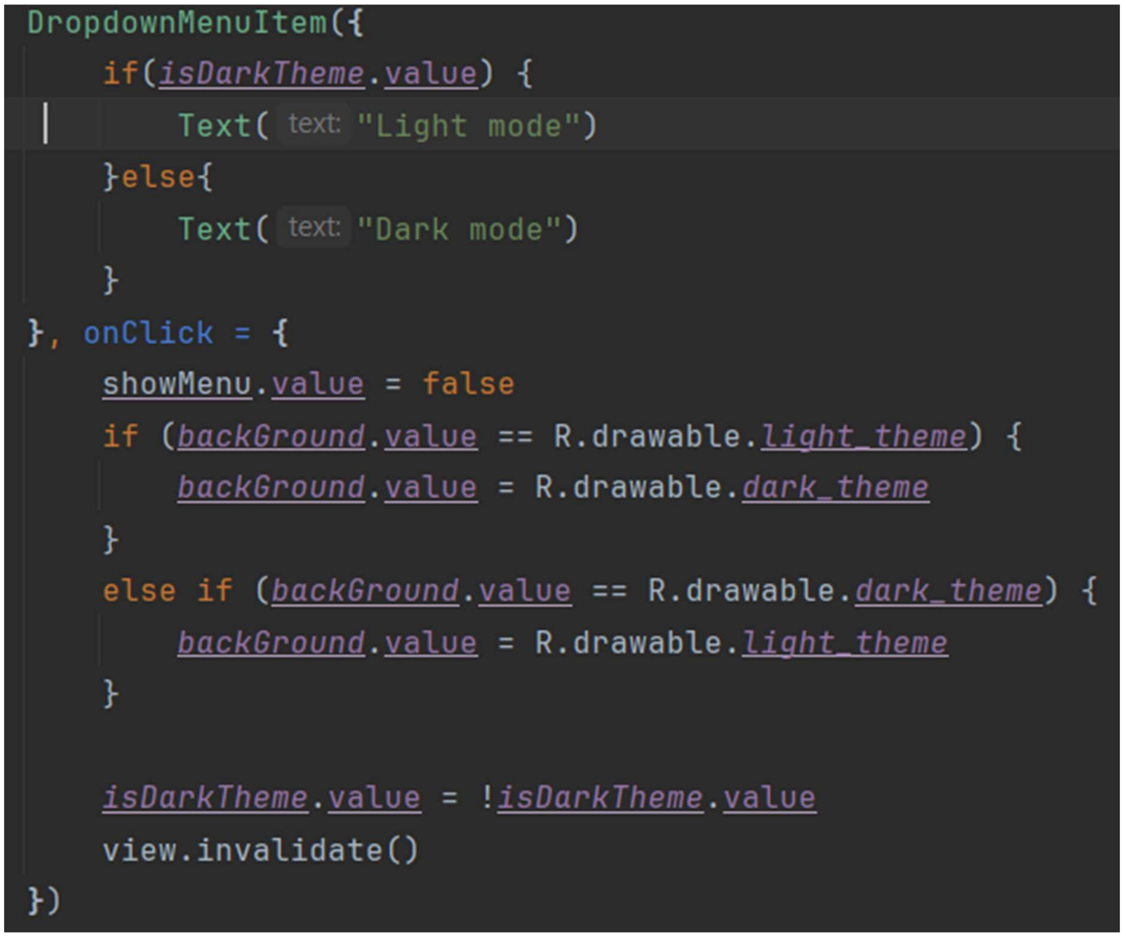


Finally, the overflow menu contains an option to toggle between light and dark mode. The app is currently in light mode, so the item shows “Dark mode”.



Now, it is in dark mode. Note the clean complement of colours and even the change in the background, from a "light" background to a "dark" one.

Below is the code for that feature. It simply toggles the colour scheme between light and dark mode, and even the background as well.



```kotlin
DropdownMenuItem({
    if(isDarkTheme.value) {
        Text( text: "Light mode")
    }else{
        Text( text: "Dark mode")
    }
}, onClick = {
    showMenu.value = false
    if (backGround.value == R.drawable.light_theme) {
        backGround.value = R.drawable.dark_theme
    }
    else if (backGround.value == R.drawable.dark_theme) {
        backGround.value = R.drawable.light_theme
    }

    isDarkTheme.value = !isDarkTheme.value
    view.invalidate()
})
```

Usability Study and User Feedback

First tester: Kieran Chai Kai Ren (16)

Kieran Chai was the first tester for the app, who is also my schoolmate. The study was conducted by sending him the zip file of the project, and allowing him to run it and test it. He was asked to test the app overall, and specifically the chat feature, as the chat feature is difficult to test as one person with one device. His feedback was recorded over Whatsapp and Discord.

<u>Testing step 1: Testing the Authentication feature</u>

Kieran found that the Authentication feature worked well, and would output descriptive Toasts to the user when necessary.

<u>Testing step 2: Testing the Discussion and Job Listings feature</u>

Kieran found that these features were quite functional, and would rarely, if at all, display signs of lag, despite using the data from Firebase. He did, however, point out that the Job listings and Discussions were not ordered by timestamps, and seemed to be randomly ordered. This was due to the autogeneration of keys in the Firebase Database, and sorting among themselves by the key. This issue was promptly fixed, by giving the Listings and Posts timestamps, and sorting them using it.

<u>Testing step 3: Testing the Chatting function</u>
Kieran found that the Chatting function worked well as intended, as the Chat conversation went very smoothly between my instance of Android and Kieran's Android phone he used for testing. He did, however, think that the Conversation should not show up for other users if one user created a chat conversation between them, and it should instead take a Whatsapp-like approach, only showing up for other users once the first user sent a message. He also noted the inability to use other keyboard items like emojis.

Kieran found that the UI was minimalistic and simple, which caters to many people, as it is simple enough that users will not feel overwhelmed yet neat enough to draw in users. He also noted the consistency in the colour themes, maintaining a purple/white theme which he found to be aesthetically pleasing.

The app worked well for him overall, however he did point out a few significant bugs that I was prompt in patching.

Second tester: Samantha (22)

Samantha Lee, a cousin of mine and also recent university graduate, was the second tester for the app. She was provided with the APK file and asked to install it on her Android device. Samantha was instructed to explore the app's features and provide feedback on the overall user experience. Her feedback was obtained over Whatsapp.

<u>Testing step 1: Testing the Authentication feature</u>
Samantha found the authentication process straightforward and user-friendly. She noted the clear instructions and the descriptive error messages when she entered incorrect credentials.

<u>Testing step 2: Testing the Discussion and Job Listings feature</u>
Samantha enjoyed browsing through the job listings and discussions. However, she noted that the search functionality could be improved. She suggested implementing filters or categories to make it easier to find relevant listings or discussions, which I promptly implemented.

<u>Testing step 3: Testing the Chatting function</u>
Samantha had some difficulties with the chatting feature initially. She mentioned that it wasn't clear how to start a new conversation or add new contacts. Once she figured it out, she found the chat interface clean and easy to use. However, she suggested adding the ability to share media files or documents within the chat.

<u>Testing step 4: Comments on UI</u>
Samantha appreciated the clean and modern design of the app's user interface. She found the color scheme pleasing and the overall layout intuitive. However, she suggested adding more visual cues or tooltips to help new users navigate the app more easily.

Overall, Samantha found the app useful and functional, but she identified a few areas for improvement, particularly in terms of discoverability and additional features.

Third tester: Theingi (40)

Theingi, my mother, was the third tester for the app. She was given the emulator on my computer to test the app. She was asked to explore the app's ease of use for older users, and her feedback was obtained in real life.

<u>Testing step 1: Testing the ease of use</u>
Theingi found that the icons were not particularly very helpful in helping her navigate through the app, as they were not very descriptive icons. This testing was, however, conducted early into the app development phase, and I have since improved the choice of icons.

<u>Testing step 2: Testing the Discussion and Job Listings feature</u>
Theingi was particularly interested in the Discussion feature, noting somewhat its similarity to the app Reddit, which she uses often. She, however, commented that clicking on the user's email should directly lead to opening a chat with them, somewhat like in Reddit, in order to make It easier. She also claimed that the Discussion should be split across communities like in Reddit Subreddits.

Overall, Theingi found the app useful and functional, but she identified a few areas for improvement, particularly in terms of UI and additional features.

Fourth tester: Alex (52)

Alex, my father, was the fourth tester for the app. He was given the emulator on my computer to test the app. He was asked to explore the app's ease of use for older users, and also the main functionalities of the app like the Discussion and Chat especially, and his feedback was obtained in real life.

<u>Testing step 1: Testing the ease of use</u>
Alex thought the icons were quite simple yet descriptive, and matched icons seen in a lot of popular apps used nowadays, like Whatsapp and Instagram.

<u>Testing step 2: Chatting feature</u>

Alex was particularly interested in the Chatting feature, as he thought it quite impressive to build a chat app within an Android app. However, he pointed out the lag in the Chat function when messages were sent or received, which was mainly due to the code looking through Firebase Database for all the messages, which took up a lot of time. His comments were the main reason I decided to switch the chats function to using Firebase Firestore instead, instead of combining it with all the other data already using Firebase Database.

<u>Testing step 2: Discussion</u>

Alex thought the Discussion feature was helpful and interesting to users, but he thought that maybe an upvoting feature should be added, like in Reddit, to upvote comments or posts, such that the top comments will be seen first, as the Discussion feature is mainly for asking and answering queries. He did, however, appreciate the Discussion feature.

Overall, Alex found the app useful and functional, but he identified a few areas for improvement, particularly in the Discussion feature, requesting more useful functions.

Fifth tester: Emma (82)

Emma, my grandmother, was the fifth tester for the app. She was given the emulator on my computer to test the app. She was asked to explore the app's ease of use for older users, and her feedback was obtained in real life.

<u>Testing step 1: Testing the ease of use</u>
Emma thought the icons was nice, instead of using words, as she claimed words would be difficult for her to see and thus use the app properly with. She did however suggest more descriptive or bigger icons, or even an option to change the size of icons or words to cater to users with poorer eyesight. She even suggested using voice commands for users unable to use their hands to navigate the app properly. She did, however, find that the app was quite nice for navigation and its UI catered well to her, comparing it to many of the apps she used, like Facebook and Wechat. She claimed she would definitely benefit from an enlarged UI, and thus suggested a zooming feature.

Overall, Emma found the app useful and functional, but she identified a few areas for improvement, particularly in the UI and ease of use for a wide variety of users.

Reflections

Through this project, I realise that I have become decently proficient in both XML and Jetpack Compose for Android Studio, and I now use them based on their benefits. I found that XML is easier in general, as it has more support due to more users, the notes using XML to teach, and my experience in XML like languages like IntelliJ Java in Year 3. XML also had the advantage of speed over Jetpack Compose.

However, Jetpack Compose was definitely more useful for long term projects, with its modularity and its neat organisation. It uses a declarative method and functions, making the project much neater and easier to look at for large projects, as compared to tens of XML files which would overwhelm someone.

I also realised that I have become quite proficient at Firebase, being able to utilise many of its features, like Authentication, Storage, Realtime Database, Firebase Firestore. Firebase definitely has the advantage over many No SQL databases in Android studio due to its compatibility with Android studio and its security.

I further also learned more about material design and what kind of material design would cater to many users and please them. In the past, I would have used a vibrant colour scheme and bright contrasting colours but I now use complementing colour schemes and calmer colours, which definitely has seen an improvement in the feedback I received from my testers compared to in Year 3 OOP.

One of the main difficulties I faced was learning how to integrate Firebase with my code. The Firebase documentation, although well written, was still quite vague and thus I had to figure out a lot on my own, even using a View Model to link my code to Firebase. Firebase was definitely difficult to learn and set up at the start, but becomes much easier once I had some experience.

My other main difficulty was figuring out how to get the screens in my app to update live with the user interactions. This was especially tough for Jetpack Compose and Jetpack Compose only recomposes a function when a parameter passed in has changed, and thus I had to look far and wide for other methods to recompose it. Eventually I had to learn to use Coroutines in Jetpack Compose, which, while difficult to grasp, was definitely worth using in the end.

If more time was given, I would definitely have implemented many of the features my testers talked about, like a more “Reddit-like” Discussion page, with multiple communities, a choice to change the design of the app like the colour scheme and the text or icon size, and I might even have redone the project in XML just for the extra speed for the user in using the project, even if it meant having to deal with an overwhelming number of XML filed. I might even have added more features to my project such as allowing employers to pay employees over the app to make it more convenient.