Android Mobile Application Development
Week 7: Firebase Authentication

Firebase Authentication
https://firebase.google.com/docs/auth
Firebase provides sign-in flows for email/password, email link, phone authentication, Google Sign-In, Facebook Login, Twitter Login, and GitHub Login.
To sign a user in:

1. Get authentication credentials from the user
2. Pass these credentials to Firebase Authentication
3. Firebase will verify the credentials and return a response to the client

Firebase Console
In your Firebase console go into Build | Authentication and in the Sign-in tab enable whichever provider you'll be using.
In Firebase go into your project and click on the gear and go to Project Settings. Scroll down and confirm that your app is listed.
You'll also need to add the SHA1 fingerprint of your app to your Project Settings.
https://developers.google.com/android/guides/client-auth
Open a terminal and run the keytool utility provided with Java to get the SHA-1 fingerprint of the certificate.
Mac:
keytool -list -v -alias androiddebugkey -keystore ~/.android/debug.keystore
Windows:
keytool -list -v -alias androiddebugkey –keystore %USERPROFILE%\.android\debug.keystore

The default password for the debug keystore is android
Then the fingerprint is printed to the terminal. Copy the SHA1 fingerprint.

If you don't have the JDK installed, or it can't be found in your path, you'll get an error as keytool is part of the JDK (can be found in the bin directory). Either install it or add it to your path. The JDK is embedded in Android Studio but it might not be added to your path. Downloading it might be the simplest way to deal with this error.

Scroll down in project settings and add a fingerprint. The debug fingerprint is enough to get this working.

Security Rules
https://firebase.google.com/docs/firestore/security/get-started?authuser=0
Firestore lets you define the security rules for the collections and documents in your database.

https://firebase.google.com/docs/firestore/security/rules-structure?authuser=0
Firestore security rules always begin with the following declaration:

```
service cloud.firestore {
  match /databases/{database}/documents {
    // ...
  }
}
```

Basic rules consist of a `match` statement specifying a document path and an `allow` expression detailing when reading the specified data is allowed:
The `match /databases/{database}/documents` declaration specifies that rules should match any Cloud Firestore database in the project. Currently each project has only a single database named `(default)`.
For all documents in all collections:
`match /{document=**}`

All match statements should point to documents, not collections. A match statement can point to a specific document, as in `match /cities/SF` or use the wildcard {} to point to any document in the specified path, as in `match /cities/{city}`.

https://firebase.google.com/docs/firestore/security/rules-conditions?authuser=0
You can set up conditions for your security rules. A condition is a boolean expression that determines whether a particular operation should be allowed or denied. Use security rules for conditions that check user authentication, validate incoming data, or access other parts of your database.
Allow statements let you target your rules for read, write, delete, etc.
This rule allows authenticated users to read and write all documents in the cities collection:

```
service cloud.firestore {
  match /databases/{database}/documents {
    match /cities/{city} {
      allow read, write: if request.auth.uid != null;
    }
  }
}
```

When we set our database up in test mode it opened read and write access open to the public for all documents in our database.
Now that we're going to want to use authentication let's set up our security rules so a user must be authenticated to write to the database. We'll continue to allow public access to read from the database.

```
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read;
      allow write: if request.auth.uid != null;
    }
  }
}
```

You must Publish your rules to save the changes.
You can test your Firestore security rules in the console. In the database rules tab there is a simulator you can use to test different types of actions on different parts of your database with different authentication rules.

Note that once you've changed the rules if you run your app without implementing authentication you won't see any data.

Google Authentication

To implement authentication you must add the Firebase Authentication and Google Play services to your app by adding the following dependencies to your app Gradle file.

```
implementation 'com.google.firebase:firebase-auth-ktx'
implementation 'com.google.android.gms:play-services-auth:20.4.0'
```

Recipes app

I'm adding Google authentication to the Recipes app (Recipes auth).
Add the Firebase Authentication and Google Play services libraries dependencies to your app Gradle file.
Your app Gradle file should have these dependencies:

```
implementation platform('com.google.firebase:firebase-bom:31.1.0')
implementation 'com.google.firebase:firebase-firestore-ktx'
implementation 'androidx.lifecycle:lifecycle-viewmodel-compose:2.5.1'
implementation 'com.google.firebase:firebase-auth-ktx'
implementation 'com.google.android.gms:play-services-auth:20.4.0'
```

Now we're ready to implement authentication.
We'll need some new strings. Add these to strings.xml.

```
<string name="login">Login</string>
<string name="logout">Logout</string>
<string
name="title_activity_google_sign_in">GoogleSignInActivity</string>
<string name="signin_success">Successfully signed in user</string>
<string name="signin_fail">Sign in unsuccessful</string>
<string name="action_logged_out">Logged Out</string>
<string name="action_login">Login</string>
```

App Bar

Create a composable to render a TopAppBar. The TopAppBar has slots for a title, navigation icon, and actions. The actions parameter adds icons arranged in a row at the end of the TopAppBar. Currently there's no support for the overflow menu (3 vertical dots) but we can use an action slot and a DropdownMenu to present our menu.

In the screens package create a new file called AppBar. We define a state variable `dropDownMenuExpanded` to track the state of the menu.

```
@Composable
fun AppBar(){
    var dropDownMenuExpanded by remember {mutableStateOf(false)}

    TopAppBar(
        title = {Text(text = stringResource(id =
R.string.app_name))},
        actions ={
            // options icon (vertical dots)
            IconButton(onClick = {dropDownMenuExpanded = true})
            {
                Icon(imageVector = Icons.Outlined.MoreVert,
```

```
contentDescription = "Options")
            }
                DropdownMenu(
                    expanded = dropDownMenuExpanded,
                    onDismissRequest = { dropDownMenuExpanded = false
},
                ) {
                    DropdownMenuItem(onClick = { dropDownMenuExpanded
= false }) {
                        Text(stringResource(id = R.string.login))
                    }
                    DropdownMenuItem(onClick = { dropDownMenuExpanded
= false }) {
                        Text(stringResource(id = R.string.logout))
                    }
                }
        }
    )
}
```

We set up our Scaffold in RecipeScreen so update it to include a topBar.

```
Scaffold(
    ...
    topBar = { AppBar() }
...}
```

Run it so you can see the app bar and the menu. Open the menu and rotate the phone. What do you notice?
Change `dropDownMenuExpanded to use rememberSaveable` instead of `remember` and the menu position will be saved during a device configuration change.

Now let's implement Google authentication.

One of the nice things about using Google authentication is that you can leverage the UI that Google provides for signing in, the ability to choose or add an account.

Create a new Compose activity called GoogleSignInActivity
New | Compose |Empty Compose Activity
Leave Launcher Activity unchecked.

GoogleSignInActivity.kt
All authentication logic will be in GoogleSignInActivity.kt
Delete Greeting() and the preview function as we won't be using those.

We need a few class level variables. Note that the `lateinit` keyword is used for variables that aren't given a default value so we must initialize them before using them.

```
//main entry for Firebase authentication
private val auth: FirebaseAuth = Firebase.auth
//Google signin API
```

```
private lateinit var googleSignInClient: GoogleSignInClient
//launch an activity with a result
private lateinit var authResultLauncher:
ActivityResultLauncher<Intent>
```

And a state variable to store the Firebase user.

```
private var currentUser by mutableStateOf<FirebaseUser?>(null)
```

Let's put all the setup in a method named googleAuthSetup(). Here we build the GoogleSignInOptions, get the GoogleSignIn client using the context(this), and set up the ActivityResultLauncher. You might be wondering where this `default_web_client_id` string comes from since we didn't add this string. This is a default string that Google play services (we added this dependency) processes in the google-services.json file that you downloaded from Firebase and added to your project.

```
private fun googleAuthSetup() {
    val gso =
GoogleSignInOptions.Builder(GoogleSignInOptions.DEFAULT_SIGN_IN)
        .requestIdToken(getString(R.string.default_web_client_id))
        .requestEmail()
        .build()

    googleSignInClient = GoogleSignIn.getClient(this, gso)

    authResultLauncher =

registerForActivityResult(ActivityResultContracts.StartActivityForRes
ult()) { result ->
            val data: Intent? = result.data
            val task =
GoogleSignIn.getSignedInAccountFromIntent(data)
            try {
                // Google Sign In was successful, authenticate with
Firebase
                val account =
task.getResult(ApiException::class.java)!!
                Log.d("auth", "firebaseAuthWithGoogle:
${account.id}")
                firebaseAuthWithGoogle(account.idToken!!)
            } catch (e: ApiException) {
                // Google Sign In failed
                Log.d("auth", "firebaseAuthWithGoogle: failed
${e.message}")
            }
        }
}
```

In onCreate() we set up the authentication listener and use the result to access the current user. We also call googleAuthSetup() which is needed to initialize our two variables marked as lateinit.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    auth.addAuthStateListener { auth ->
        currentUser = auth.currentUser
    }
    googleAuthSetup()
}
```

Now we'll create a method that will actually call Google signin using the account token that we defined in `GoogleSignInOptions`. If signin is successful we set the current user, present a Toast and then call finish() which destroys the activity and automatically returns the user to the previous activity.

```
private fun firebaseAuthWithGoogle(idToken: String) {
    val credential = GoogleAuthProvider.getCredential(idToken, null)
        auth.signInWithCredential(credential)
            .addOnCompleteListener{ task ->
                if (task.isSuccessful) {
                    // Sign in success
                    currentUser = auth.currentUser
                    Toast.makeText(this,
"${getString(R.string.signin_success)} ${currentUser?.displayName}",
Toast.LENGTH_LONG).show()
                    finish()
                } else {
                    // Sign in fails
                    Toast.makeText(this,
"${getString(R.string.signin_fail)}", Toast.LENGTH_LONG).show()
                }
            }
}
```

Now we create a method that will get the sign in intent and then launch the Google sing in.

```
private fun googleLogIn() {
    val signInIntent = googleSignInClient.signInIntent
    authResultLauncher.launch(signInIntent)
}
```

And then we call googleLogIn() after googleAuthSetup()in onCreate().

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    // Log.d("auth", "${auth.currentUser}")
    auth.addAuthStateListener { auth ->
        currentUser = auth.currentUser
    }
    googleAuthSetup()
    googleLogIn()
}
```

MainActivity.kt

In MainActivity we need a function that will start the GoogleSignInActivity.

```
fun login() {
    val intent = Intent(this, GoogleSignInActivity::class.java)
    startActivity(intent)
}
```

Now we need to pass that from MainActivity to ScreenSetup, RecipeScreen, and AppBar so we can call it when the user taps Login in the menu.
In onCreate()

```
ScreenSetup(loadWebPage = ::startDetailActivity, login = {login()})

@Composable
fun ScreenSetup(loadWebPage: (String) -> Unit, login: () -> Unit){
    RecipeScreen(loadWebPage = loadWebPage, login)
}
```

In RecipeScreen()

```
fun RecipeScreen(loadWebPage: (String) -> Unit, login: () -> Unit) {
...
Scaffold(
    backgroundColor = MaterialTheme.colors.surface,
    topBar = { AppBar(login) },
...)
}
```

In AppBar()

```
fun AppBar(login: () -> Unit){
...
DropdownMenuItem(onClick = {
    login()
    dropDownMenuExpanded = false
}) {
    Text(stringResource(id = R.string.login))
}
```

Now run the app and the Login menu calls login() which starts the GoogleSignInActivity and runs Google authentication.

Logout

Logging out is very simple. In GoogleSignInActivity add a function to call signOut() and then finish().

```
private fun googleLogout() {
    auth.signOut()
    Toast.makeText(this, "${getString(R.string.action_logged_out)}",
Toast.LENGTH_LONG).show()
    finish()
}
```

In MainActivity we create a logout() function.

```
fun logout(){
    val intent = Intent(this, GoogleSignInActivity::class.java)
    startActivity(intent)
}
```

Since both login and logout call GoogleSignInActivity we need a way to tell the activity if we want to log in our out. We'll add a string that gets passed to the intent.

```
fun login() {
    val intent = Intent(this, GoogleSignInActivity::class.java)
    intent.putExtra("googleAuth", "login")
    startActivity(intent)
}

fun logout(){
    val intent = Intent(this, GoogleSignInActivity::class.java)
    intent.putExtra("googleAuth", "logout")
    startActivity(intent)
}
```

Then in GoogleSignInActivity we extract the name of the string and use the value to determine if we need to log in or out.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    auth.addAuthStateListener { auth ->
        currentUser = auth.currentUser
    }
    googleAuthSetup()
    val googleAuth = intent.getStringExtra("googleAuth")
    when (googleAuth){
        "login" -> googleLogIn()
        "logout" -> googleLogout()
    }
}
```

Now we need to pass the logout function from MainActivity to ScreenSetup, RecipeScreen, and AppBar so we can call it when the user taps Logout in the menu, same as we did for Login.

In MainActivity in onCreate()

```
ScreenSetup(loadWebPage = ::startDetailActivity, login = {login()},
logout = {logout()})

@Composable
fun ScreenSetup(loadWebPage: (String) -> Unit, login: () -> Unit,
logout: () -> Unit){
    RecipeScreen(loadWebPage = loadWebPage, login)
}
```

In RecipeScreen()

```
fun RecipeScreen(loadWebPage: (String) -> Unit, login: () -> Unit,
logout: () -> Unit) {...
Scaffold(
    backgroundColor = MaterialTheme.colors.surface,
    topBar = { AppBar(login, logout) },
...)
}
```

In AppBar()

```
fun AppBar(login: () -> Unit, logout: () -> Unit){
...
DropdownMenuItem(onClick = {
    logout()
    dropDownMenuExpanded = false
}) {
    Text(stringResource(id = R.string.logout))
}
```

Now run the app and the Logout menu calls logout() which starts the GoogleSignInActivity and signs the user out.
Note that in testing the app the device will remember that you're logged in. So to test the whole authentication chain again you either need to remove your Google account from the device, uninstall the app, or use a device where you're not signed in.

This is a basic example to introduce you to Firebase authentication on Android. If you want to offer the users different login methods you'd want to create a screen where they can choose from google, Facebook, email/password, etc.
If different accounts have different data in Firebase you'd need to organize your database differently so recipes are specific to a user.