eecs441§3&4

Chatter with Sign-in and Jetpack Compose Kotlin

Cover Page

DUE Wed, 11/10, 2 pm

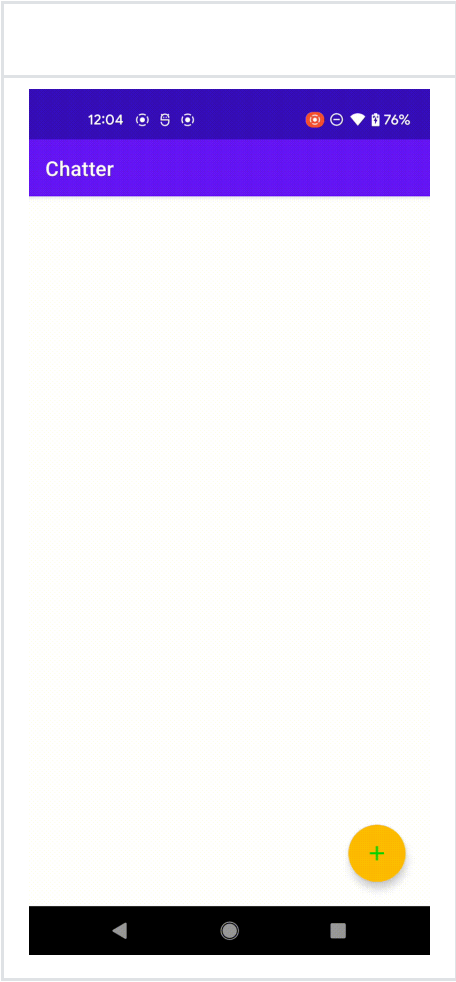
We again have two goals with this labs: First, to introduce you to concurrent programming using coroutines. Second, to add Google Sign-in with biometric authentication.

Gif demo

Post chatts with Google Sign-in:

Note: Annotations in orange describe user actions or screens not recorded by the screen recorder and are not part of the app.

> Right click on the gif and open in a new tab to get a full-size view. To view the gif again, please hit refresh on the browser (in the new tab where the gif is opened).



Preparatory work

Recall that you made a copy of the Jetpack Compose version of basic Chatter in the previous lab and named the folder `lab4`. In Android Studio's `File > Open` choose `YOUR_LABSFOLDER/lab4/kotlinJpCChatter` to start work on this lab.

As in previous labs, we'll collect all the extensions we'll be using into one file. Create a new Kotlin file called `Extensions.kt` and put the same `toast()` extension to `Context` from the previous lab in it.

Part I: Converting `ChattStore` to use Coroutines

We have used the Volley and OkHttp3 networking libraries. In this lab we will use the Retrofit networking library to take advantage of its support for coroutines. Retrofit was built on top of OkHttp3 by the same company that built OkHttp3 (Square Inc.).

First, add additional dependencies to your module/app gradle build file:

```
    dependencies {
        ...
        implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-android:1.5.2'
        implementation 'com.squareup.okhttp3:okhttp:5.0.0-alpha.2'
        implementation 'com.squareup.retrofit2:retrofit:2.9.0'
    }
```

Retrofit is set up differently from Volley and OkHttp3: whereas in Volley and OkHttp3 we constructed each API Url as we're about to make an HTTP request, in Retrofit we create, ahead of time, an interface in which each API Url is encoded into its own method. Add the following interface to your `ChattStore.kt`, **outside** the `ChattStore` object:

```
interface ChatterAPIs {
    @GET("getchatts/")
    suspend fun getchatts(): Response<ResponseBody>

    @POST("postchatt/")
    suspend fun postchatt(@Body requestBody: RequestBody): Response<ResponseBody>
}
```

All of the `ChatterAPIs` interface methods are suspending functions, which means they can only be called from another suspending function or a coroutine scope. We **redefine** the `ChattStore` object to implement `CoroutineScope` by delegation to `MainScope()`:

```
object ChattStore: CoroutineScope by MainScope() {
```

To use Retrofit, we first need to create a Retrofit client and customized it to our `ChatterAPIs` interface above. **Inside** your `ChattStore` object, add the following properties below the `serverUrl` property:

```
private val retrofit = Retrofit.Builder()
    .baseUrl(serverUrl)
    .build()
private val chatterAPIs = retrofit.create(ChatterAPIs::class.java)

private val retrofitExCatcher = CoroutineExceptionHandler { _, error ->
    Log.e("Retrofit exception", error.localizedMessage ?: "NETWORKING ERROR")
}
```

We will use `retrofitExCatcher` as the exception handler for the `ChatterAPIs` methods.

We now convert `postChatt()` to be a suspending function, by prepending the keyword `suspend` to its declaration:

```
suspend fun postChatt(chatt: Chatt) {
```

Keep the declaration of `jsonObj` in `postChatt()` but **replace** the declaration of `postRequest` with:

```
    val requestBody = JSONObject(jsonObj).toString().toRequestBody("application/json".toMediaTyp

    lateinit var response: Response<ResponseBody>
    withContext(retrofitExCatcher) {
        // Use Retrofit's suspending POST request and wait for the response
        response = chatterAPIs.postchatt(requestBody)
    }
    if (!response.isSuccessful) {
        Log.e("postChatt", response.errorBody()?.string() ?: "Retrofit error")
        // Android Studio false positive WARNING on .string()
        // https://github.com/square/retrofit/issues/3255
    } else {
        getChatts()
    }
```

In the previous lab, we passed `getChatts()` as a completion to the Volley `JsonObjectRequest()`. With the use of coroutines in Retrofit, we can call `getChatts()` after Retrofit's `chatterAPIs.posthatt()` returns, in a direct/sequential style. Since we're not using Volley anymore, **remove** the three lines of code dealing with Volley's `queue` in `postChatt()`.

> Android Studio could give a false positive WARNING on the call to `.string()` above. You can safely ignore the warning.

Now that `postChatt()` has been converted into a suspending function, we need to also convert its caller. The method `postChatt()` is called only once in `PostView`. Search for the one occurrence of `postChatt` in `PostView`. Then replace the call to `postChatt()` with:

```
        MainScope().launch {
            postChatt(Chatt(username, message))
        }
```

Since the call to `postChatt()` is from the `onClick` event handler, which is not a suspending function and cannot be converted into a suspending function, we have to put the call to `postChatt()` inside a coroutine scope. We use MainScope() here since we do not want the `chatt` posting to be cancelled when we leave `PostView`.

Next we convert `getChatts()`. Thanks to the reactive framework of Jetpack Compose, our timeline view gets updated automatically whenever the `chatts` array is updated. We no longer need to pass a completion handler to `getChatts()`. We now remove the completion handler and replace the use of Volley with a call to suspending Retrofit:

```
fun getChatts() {
    launch(retrofitExCatcher) {
        // Use Retrofit's suspending GET request and wait for the response
        val response = chatterAPIs.getchatts()
        if (response.isSuccessful) {
            val chattsReceived = try {
                JSONObject(response.body()?.string() ?: "").getJSONArray("chatts")
                // Android Studio false positive WARNING on .string()
                // https://github.com/square/retrofit/issues/3255
            } catch (e: JSONException) {
                JSONArray()
            }

            chatts.clear()
            for (i in 0 until chattsReceived.length()) {
                val chattEntry = chattsReceived[i] as JSONArray
                if (chattEntry.length() == nFields) {
                    chatts.add(
                        Chatt(
                            username = chattEntry[0].toString(),
                            message = chattEntry[1].toString(),
                            timestamp = chattEntry[2].toString()
                        )
                    )
                } else {
                    Log.e(
                        "getChatts",
                        "Received unexpected number of fields: " + chattEntry.length()
                            .toString() + " instead of " + nFields.toString()
                    )
                }
            }
        }
    }
}
```

We have not made `getChatts()` a suspending function to avoid having to call it inside a coroutine scope in the rest of the app. Instead, we use the `MainScope()` in `ChattStore` to launch a coroutine to call the Retrofit suspending function inside `getChatts()` itself. This coroutine could still be running when `getChatts()` returns to its caller. Again, we rely on the reactive nature of Jetpack Compose to update the timeline view when the coroutine launched by `getChatts()` finally completes its run and the `chatts` array has been updated.

We have no more need for the Volley `queue` anywhere in `ChattStore`. You can remove the `queue` property from your `ChattStore` object. Remove the following line:

```
private lateinit var queue: RequestQueue
```

You can also remove the Volley dependency from your Module's `build.gradle` file. Remove:

```
implementation 'com.android.volley:volley:1.2.0'
```

We're done with the conversion to suspending Retrofit!

Part II: Chatter with Google Sign-in and Biometric Authentication

We now add authentication to the `Chatter` app. We will be using OAuth 2.0 for authentication, in the guise of OpenID Connect, as implemented by Google Identity Platform. In other words, we'll add Google Sign-in to `Chatter`. Google Sign-in is not the only OAuth2.0-based identity authentication provider, Apple, Facebook, Twitter, WeChat, among others, all use OAuth 2.0. We've chosen Google Sign-in for illustrative purposes as it seems to be the most popular and accessible. Following popular sites such as stackoverflow and reddit, we've also designed `Chatter` to require authentication only for posting `chatts`, not for viewing them.

In addition to the two APIs `getchatts` and `postchatt` we will be adding a new API, `adduser`, for user sign in. We will also modify the `postchatt` API, which must now carry an authenticated credential to post a `chatt`. Due to this change, we will rename the `postchatt` API to `postauth`. Unlike previous labs, we will start with modifications to the front end before we make changes to the back end because we need the Google ID Token the front end will retrieve from Google Sign-In to test the back end. We will also start with learning about Google Sign-in before exploring changes to the API.

The front end of the app will mostly be written in Jetpack Compose. To experiment with Compose and AndroidView interoperability, we have retained the use of Google Sign-In button.

> ⚠ **DISCLAIMER:** this lab is not an exercise in designing a secure authentication protocol. It is only meant to familiarize you with some of the authentication tools available in the mobile development environment. The protocol implemented here has not been vetted by a security expert.

Integrating Google Sign-in

The following instructions are largely based on Start integrating Google Sign-In into your Android app, simplified and elaborated upon. See also: Google Sign-in for Android.

Add Google Play Services

In your app-level gradle file, `/Gradle Scripts/build.gradle (Module: kotlinJpCChatter.app)`, add Google Play services as a dependency. Also add the dependency for the biometric package:

```
dependencies {
    ...
    implementation 'com.google.android.gms:play-services-auth:19.2.0'
    implementation 'androidx.biometric:biometric:1.2.0-alpha03'
}
```

Configure a Google API Console Project

Follow this link: Configure a Google API Console project to set up a Google Cloud Platform Console account. (You will need a gmail address, not a umich email address, to set up a Google Cloud Platform Console account.)

At the linked page,

1. Click the big blue `Configure a project` button.
2. Enter your project name, in this case we will use `kotlinJpCChatter`.
3. Click `Next` and when prompted to `Configure your OAuth client` enter your project name yet again.
4. The next step is "Where are you calling from?". Choose `Android`.
5. This will prompt for a `Package name`. Open your `AndroidManifest.xml` file. Your package name will be listed, for example, as:

   ```
   <?xml version="1.0" encoding="utf-8"?>
   <manifest xmlns:android="http://schemas.android.com/apk/res/android"
       ...
       package="edu.umich.YOUR_UNIQNAME.kotlinJpCChatter">
       ...
   ```

   where `YOUR_UNIQNAME` will be your actual uniqname. Copy and paste the package name. The package name of your Android Studio project must match EXACTLY the package name used in the Google API Console.

▶ applicationID

1. You also need to enter the `SHA-1 signing certificate`. To get your certificate, on your laptop enter:

MacOS on Terminal:

```
laptop$ keytool -v -list -keystore ~/.android/debug.keystore
```

Windows on PowerShell:

```
PS laptop> keytool -v -list -keystore ~\.android\debug.keystore
```

▶ Don't have keytool?

When prompted for password, just hit return/enter. You should see amongst the output three lines that start like this:

```
Certificate fingerprints:
         SHA1: XX:XX:XX:...:XX
         SHA256: YY:YY:YY:...:YY
```

Cut and paste the SHA1 certificate to the Google API Console. Click `CREATE`.

▶ Don't see SHA1?

1. Click `Download Client Configuration`. This will download a `credentials.json` file to your computer (most likely into your `Downloads` folder). You won't be needing this file other than for your record.
2. Copy your `Client ID` to the clipboard and add it to your `/app/res/values/strings.xml`, replacing `YOUR_APP'S_CLIENT_ID` with your actual client ID:

```
<resources>
    ...
    <string name="clientID">YOUR_APP'S_CLIENT_ID</string>
</resources>
```

▶ OAuth vs. Web client ID

And that's all you need to obtain a Google Client ID.

Add Google Sign-In to `Chatter`

Recall that we design `Chatter` to allow users to view `chatts` without requiring authentication. We will authenticate users only when they post a `chatt`.

Here's the authentication flow:

1. In `MainView`, when the user launches `PostView` for the first time after launching the app, we prompt the user for biometric check to load previous session's `chatterID` from Android's `SharedPreferences`.
2. In `PostView`, we check whether the user has a valid `chatterID`. If so, they can go ahead and post a `chatt`.
3. Otherwise, we launch `SigninView`.
4. In `SigninView`, we first check if the user is signed in. If so, we check the validity of their ID Token, which also automatically refreshes it if it's no longer valid.

5. If the user is not signed in, we launch Google Sign-In and let the user obtain a new ID Token.
6. Once the user has a valid ID Token, we contact the `Chatter` back end with the ID Token and Client ID to obtain a `chatterID`.
7. Upon receiving a new `chatterID` from the back end, we perform another biometric check to update the `chatterID` in Android's `SharedPreferences`.
8. We use `chatterID` during its lifetime to post `chatt`s without further checking the user's sign-in status.

> ❗ **WARNING:** With Google Sign-In, signing out only signs the user out of the app, it does not log the user out of Google on the device. Subsequently, all the user has to do to sign back in on the app is to select their account. Google will not challenge them for password again. Apparently, this is OAuth 2.0 standard-compliant behavior, including, for example for Twitter sign-in. The user is thus left vulnerable on public computers (see github and stackoverflow postings on this topic). Further, if the app is killed or force closed from outside the app, the user will not be signed out. The only way to sign a user out from the device is through the user's `Manage your Google Account` button on a browser. Navigate to `Security > Your devices`. Click on the three vertical dots to the upper right of your device and choose `Sign out`.

As you can see from steps 1 and 7 above, our only use of the biometric check is to control access to the `chatterID` stored across invocations of the app. It doesn't make the sign-in process itself any more secure. If you don't have `chatterID` stored from a previous run of the app, you can still sign in with Google and post `chatt`s normally. Even without a stored `chatterID`, if your previous Google Sign-in has not expired, you can also still post `chatt` without being prompted to sign in again, as per standard Google Sign-in behavior. We will initially implement `chatterID` without storing it in Android `SharedPreferences`. After we have a working `chatterID`, we will return to steps 1 and 7 and implement `SharedPreferences` storage and biometric checks.

`ChatterID`

Create a new Kotlin Object file called `ChatterID.kt`. We store the `chatterID` we will obtain from the `Chatter` back end in a **singleton** called `ChatterID`:

```
object ChatterID {
    var expiration = Instant.EPOCH
    var id: String? = null
        get() {
            return if (Instant.now() >= expiration) null else field
        }
        set(newValue) {
            field = newValue
        }
}
```

`ChatterID.id` is `null` when either the user hasn't obtained a `chatterID` from the back end or the ID has expired.

> Please review the lecture slides for the definition of a singleton if you're not sure what it means.

PostView

When `PostView()` is launched, we first check whether we have a valid `ChatterID`. If not, and this is not a recomposition, or re-creation due to configuration/orientation change, of `PostView()`, we navigate to `SigninView` to obtain a `ChatterID`:

```
var isLaunching by rememberSaveable { mutableStateOf(true) }

LaunchedEffect(Unit) {
    if (isLaunching) {
        isLaunching = false
        id ?: run {
            navController.navigate("SigninView")
        }
    }
}
```

▶ Navigation transition animation

For the sake of defensive coding, we want to ensure that no `chatt` is posted without a valid `chatterID`. Look for the call to `postChatt()` in `PostView`. It should be in an `onClick = {}` block. Add the following code above the line `enableSend = false`, inside the `onClick = {}` block:

```
id ?: run {
    context.toast("Error signing in. Please try again.")
    navController.popBackStack("MainView", inclusive=false)
}
```

That's all the changes we need to make to `PostView()`.

MainActivity and MainView

So that we can navigate to `SigninView` from `PostView`, add the following inside the `NavHost(){}` block of your `MainActivity`'s `onCreate()`:

```
composable("SigninView") {
    SigninView(this@MainActivity, navController)
}
```

Since `SigninView` uses experimental coroutine API, Android Studio would like you to tag the `MainActivity` class with the annotation:

```
@ExperimentalCoroutinesApi
```

Put the annotation right above the declaration of the `MainActivity` class.

In your `MainView()` composable, in the `LaunchedEffect() {}` block, replace the line `getChatts(context) {}` with just `getChatts()`. Similarly, modify the call to `getChatts()` in the `onRefresh` action block to:

```
            onRefresh = {
                getChatts()
                isRefreshing = false
            }
```

SigninView

We will encapsulate the Google Sign-in functionalities within its own composable. Create a new Kotlin file, `SigninView.kt`. Declare the following `SigninView()` composable in the file, and instantiate a client of Google Sign-In in the `SigninView()` composable. As part of the instantiation, we set sign-in options that specify the information we want returned when a user is signed in:

```
@ExperimentalCoroutinesApi
@Composable
fun SigninView(context: Context, navController: NavHostController) {

    // Build a GoogleSignInClient with specified GoogleSignInOptions.
    val signinClient = GoogleSignIn.getClient(context,
        GoogleSignInOptions.Builder(GoogleSignInOptions.DEFAULT_SIGN_IN)
            // DEFAULT_SIGN_IN options request user's ID and basic profile.
            // ID Token is not part of DEFAULT_SIGN_IN and must be additionally
            // requested, as shown. Email address, if desired, must also be
            // requested with .requestEmail(/*void*/), not done here.
            .requestIdToken(stringResource(clientID))
            .build())
}
```

> If Android Studio complains about not knowing `clientID`, add to the top of your `SigninView.kt` file, in the `import` block:
>
> ```
> import YOUR_PACKAGENAME.R.string.clientID
> ```
>
> where `YOUR_PACKAGENAME` is the name of your package, which you can find listed at the top of the file.

Next add the following sign-in result handler function. This function and all the code in this section belongs **inside** the `SigninView()` composable:

```
    fun handleSignInResult(completedTask: Task<GoogleSignInAccount>) {
        val account = try {
            completedTask.getResult(ApiException::class.java)
        } catch (e: ApiException) {
            // The ApiException status code indicates the detailed failure reason.
            // Refer to the GoogleSignInStatusCodes class reference for more information:
            // https://developers.google.com/android/reference/com/google/android/gms/auth/api/signi
            context.toast("Failed Google SignIn ${e.localizedMessage}\nIs application.id in Module's
            navController.popBackStack(route="MainView",inclusive = false)
            return
```

```
        }

        account?.let {
            // Successful SignIn, addUser to Chatter back end, obtain chatterID
            MainScope().launch {
                if (addUser(context, it.idToken)) {
                    navController.popBackStack("PostView", inclusive = false)
                } else {
                    context.toast("Sign in problem.  Please try again.")
                    navController.popBackStack("MainView", inclusive = false)
                }
            }
        }
    }
```

If `handleSignInResult()` cannot obtain an account, inform user that Google Sign-In has failed and return them to the `MainView` to try again. Otherwise, Sign-In succeded and we try to register user with `Chatter` back end. If registration succeeded, we return user to `PostView` so that they can post a `chatt`. Otherwise, we return user to `MainView`.

> Android Studio will complain that it doesn't know of `addUser()` and will likely mark the whole `addUser() {}` block in red. This is ok, we'll implement `addUser()` later.

Unlike in a `class` definition where the ordering of function declarations does not matter, the ordering does matter in a composable function, which is why we needed to declare the above handler function before we can use it. We now use the handler to create an `ActivityResultContracts` to start the Google Sign-In activity:

```
    val forSigninResult =
        rememberLauncherForActivityResult(ActivityResultContracts.StartActivityForResult()) { result
            handleSignInResult(GoogleSignIn.getSignedInAccountFromIntent(result.data))
        }
```

Since we are in a composable that can be recomposed at any time, we use `rememberLauncherForActivityResult()` instead of `registerForActivityResult()` in creating the contract so that it persists across recompositions. Also note that `rememberLauncherForActivity()` can only be invoked directly from inside a `@Composable` function.

Once have we created the `forSigninResult` activity-result contract, we check whether the user is signed in. If the user is **not** signed in, we set up a standard Google Sign-In button that launches the Google Sign-In activity when clicked. The Google Sign-In activity is launched with the `forSigninResult` contract. Add the following code directly below the `forSiginResult` variable:

```
    // Is the user signed in?
    getLastSignedInAccount(context) ?: run {
        // User is not signed in
        Box(contentAlignment = Alignment.Center,
            modifier = Modifier.fillMaxSize(1f)) {
            // Set the size of the signinButton
```

```
            AndroidView(factory = { context ->
                SignInButton(context).apply {
                    setSize(SignInButton.SIZE_STANDARD)
                    setOnClickListener {
                        // and call GoogleSignIn() when the button is clicked
                        forSigninResult.launch(signinClient.signInIntent)
                    }
                }
            })
        }
        return
    }
```

The `SignInButton()` as defined in Google's Sign-In SDK is a traditional Android View UI element, not a composable. We wrap this UI element inside `AndroidView()` composable, which then allows us to wrap it inside a `Box()` composable inside our `SigninView()` composable.

> In addition to `AndroidView()`, there is also `AndroidViewBinding()` which allows you to inflate a layout with ViewBinding.

One last thing to add to `SigninView()`: if the user is signed in, "silently" check the freshness of the user's ID Token or obtain a new one if necessary. "Silently" means the user is not prompted for additional login data. Add the following code directly below the above, still inside the `SigninView()` composable function:

```
        // User is SignedIn, refresh idToken
        signinClient.silentSignIn().addOnCompleteListener(context.mainExecutor) { task ->
            handleSignInResult(task)
        }
```

That's all the code we need for the `SigninView()` composable. Now we work on `addUser()`.

Add user to `Chatter` back-end service

`adduser` API

We add a new API to the `Chatter` back end. When a user signs in and submits their ID Token from Google, the back-end API `adduser` will receive the token, make sure it hasn't expired, generate a new `chatterID`, and return it, along with its lifetime, to the user.

API:

```
/adduser/
<- clientID, idToken
-> chatterID, lifetime 200 OK
```

The data format `adduser` expects is:

```
{
    "clientID": "YOUR_APP'S_CLIENT_ID",
    "idToken": "YOUR_GOOGLE_ID_TOKEN"
}
```

addUser()

The front-end method `addUser()` creates a JSON object containing (1) the app's OAuth 2.0 Client ID you've previously created and (2) the passed in `idToken`. It then sends the JSON object to `Chatter`'s backend with a POST request.

The back-end server will verify the presented `idToken` with Google. If verification is successful, the back end returns a `chatterID`. Subsequently, the back end will identify the user by this `chatterID`, for the lifetime of the `chatterID`. If the token cannot be validated for whatever reason, `addUser()` returns `false`.

Add the back-end `adduser` API to the `ChatterAPIs` interface used to create our Retrofit client. In `ChattStore.kt`, find `interface ChatterAPIs {` and add the following method to the interface:

```
@POST("adduser/")
suspend fun adduser(@Body requestBody: RequestBody): Response<ResponseBody>
```

Now add the following `addUser()` method to your `ChattStore` class:

```
@ExperimentalCoroutinesApi
suspend fun addUser(context: Context, idToken: String?): Boolean {
    var userAdded = false

    idToken ?: return userAdded

    val jsonObj = mapOf(
        "clientID" to context.getString(clientID),
        "idToken" to idToken
    )
    val requestBody = JSONObject(jsonObj).toString().toRequestBody("application/json".toMediaTyp

    withContext(retrofitExCatcher) {
        // Use Retrofit's suspending POST request and wait for the response
        val response = chatterAPIs.adduser(requestBody)

        if (response.isSuccessful) {
            val responseObj = JSONObject(response.body()?.string() ?: "")
            // obtain chatterID from back end
            id = try { responseObj.getString("chatterID") } catch (e: JSONException) { null }
            expiration = Instant.now().plusSeconds(try { responseObj.getLong("lifetime") } catch

            id?.let {
                userAdded = true
                // will save() chatterID later
            }
        } else {
            Log.e("addUser", response.errorBody()?.string() ?: "Retrofit error")
        }
```

```
        }
        return userAdded
    }
```

> If Android Studio complains about not knowing `clientID`, add to the top of your `ChattStore.kt` file, in the `import` block:
>
> ```
> import YOUR_PACKAGENAME.R.string.clientID
> ```
>
> where `YOUR_PACKAGENAME` is the name of your package, which should also be listed at the top of the file.

Upon receiving a `chatterID` from the `Chatter` back-end server, `addUser()` stores it in the `ChatterID` singleton along with a computed expiration time. If we did not retrieve a valid `chatterID` from the back end, we return `false`.

We have made `addUser()` a suspending function so that the caller will wait for sign in to complete before proceeding to the next step. Recall that when a suspending function returns, all of its work is done.

`postauth` API

The `postauth` API, replacing the `postchatt` API, requires that `chatterID` be sent along with each `chatt`. It first verifies that the `chatterID` exists in the database and has not expired. That being the case, the new `chatt`, along with the user's username (retrieved from the database) will be added to the `chatts` database. Otherwise, HTTP error 401 will be returned to the front end.

API for `postauth`:

```
/postauth/
<- chatterID, message
-> {} 200 OK
```

The data format `postauth` expects is:

```
{
    "chatterID": "YOUR_CHATTER_ID",
    "message": "Chitt chatts"
}
```

Modified `postChatt()`

We need to make two changes in `ChattStore.kt`:

1. Replace `@POST("postchatt/")` in the `ChatterAPIs` interface to `@POST("postauth/")`.
2. In `postChatt()`, replace:

```
    "username" to chatt.username,
```

with:

```
    "chatterID" to id,
```

If the `Chatter` back end does not recognize the `chatterID` as valid, it will return an error.

Completing the back end

At this point, if you haven't completed your back end, we suggest you switch gear and work on your back end. You'll need the ID Token obtained from Google Sign-In to test your back end, which is why we had you work on the front end up to this point first. The ID Token is the second argument passed to `addUser()` above. You can have Android Studio print it out for you using `Log.d()`, for example. Now that you can obtain an ID Token from Google, you can switch to work on the back end before completing the rest of your front end.

- Chatter with Sign-in Back End

With your back end completed, you should now be able to post `chatts` to your Google Sign-In integrated `Chatter` back end. Congratulations!

`SharedPreferences` with biometric access control

Returning to the front end: we store the `chatterID` obtained from the back end to Android `SharedPreferences`, with biometric checks. Recall that the only purpose of the biometric check is to control access to the stored `chatterID` across invocations of the app. It doesn't make the sign-in process itself any more secure. If you don't have `chatterID` stored from a previous run of the app, you can still sign in with Google and post `chatt` normally. Even without a stored `chatterID`, if your previous Google Sign-in has not expired, you also can still post `chatt` without being prompted to sign in again, as per standard Google Sign-in behavior.

Let's first figure out how to store `chatterID` in `SharedPreferences`, in plain text. Then we will look at how to encrypt `chatterID` for storage. Finally, we will control access to the stored `chatterID` with biometric authentication.

`SharedPreferences` storage

In `MainView`, when the user attempts to launch `PostView` for the first time, we try to load the previous session's `chatterID` from Android's `SharedPreferences`. In `MainView`, search for:

```
navController.navigate("PostView")
```

and replace it with a launch of the suspending `open()` method. We navigate to "PostView" after `open()` returns:

```
MainScope().launch {
    open(context)
    navController.navigate("PostView")
}
```

Since the function `open()` uses an experimental coroutine API, Android Studio would like you to tag `MainView()` with the annotation, `@ExperimentalCoroutinesApi`. Put this annotation right above the declaration of the composable `MainView()`.

open()

We define `open()` as a suspending method of the `ChatterID` object. Add the following code **inside** the `ChatterID` object, in `ChatterID.kt`, below all of its current contents:

```
private const val ID_FILE = "Chatter"
private const val KEY_NAME = "ChatterID"
private const val INSTANT_LENGTH = 24
private const val IV_LENGTH = 12

suspend fun open(context: Context) {
    if (expiration != Instant.EPOCH) { // this is not first launch
        return
    }

    context.getSharedPreferences(ID_FILE, Context.MODE_PRIVATE)
        .getString(KEY_NAME, null)?.let {
            expiration = Instant.parse(it.takeLast(INSTANT_LENGTH))
            id = it.dropLast(INSTANT_LENGTH)
        }
    }
}
```

> It's ok if Android Studio grays out the `IV_LENGTH` variable. We'll use it later. We don't need `open()` to be a suspending method for now, but we will when we do biometric check later.

`ChatterID`'s expiration being `Instant.EPOCH` indicates that `open()` is called for the first time after app launch. In which case, the code loads or creates a new `SharedPreferences` file, whose name is stored in `ID_FILE`. Then it searches for the item whose key/name is stored in `KEY_NAME`. If no such item exists, it returns the second argument to `.getstring()` (in this case, `null`).

If `.getString()` doesn't return `null`, the code goes on to store the last `INSTANT_LENGTH` elements of the returned `String` as of type `Instant`, representing the expiration time of the stored `chatterID`. Elements preceding the expiration time is the `chatterID` itself.

> Operations on `SharedPreferences` is relatively expensive and it is common to concatenate Strings so that they can be operated on in one fell swoop.

`SharedPreferences` update

Every time we obtain a new `chatterID` from the back end, in `ChattStore.addUser()`, we want to update the entry associated with `KEY_NAME` in our `SharedPreferences`.

`save()`

Let's define a `save()` method in the `ChatterID` object:

```
suspend fun save(context: Context) {
    id?.let {
        val idVal = id+expiration.toString()
        context.getSharedPreferences(ID_FILE, Context.MODE_PRIVATE)
            .edit().putString(KEY_NAME, idVal).apply()
    }
}
```

which creates a thread-safe editor with `.edit()`, concatenates the `chatterID` with expiration time, updates `SharedPreferences` with the new item using `.putString()`, and asynchronously commits the changes to persistent storage (`.apply()` is an asynchronous version of `.commit()`).

> Again, we have declared `save()` a suspending function for use with biometric check later.

Every time we obtain a new `chatterID` from the back end, we want to save it to `SharedPreferences`.

In `ChattStore.addUser()` in file `ChattStore.kt`, find and replace the comment, `// will save() chatterID later` with:

```
            save(context)
```

That's all the changes we need to make to store `chatterID` to support `SharedPreferences`! You can now test your implementation of `SharedPreferences` storage by closing your `Chatter` app after making a post, re-launching it within your `chatterID` lifetime (which you set in the back end), and it should allow you to post without requiring you to login again.

To help you test your code, you may want to add the following `delete()` method to the `ChatterID` object:

```
fun delete(context: Context) {
    val folder = File(context.getFilesDir().getParent()?.toString() + "/shared_prefs/")
    val files = folder.list()
    files?.forEach {
        context.getSharedPreferences(it.replace(".xml", ""), Context.MODE_PRIVATE)
            .edit().clear().apply() // clear each preference file from memory
        File(folder, it).delete()   // delete the file
    }
}
```

and call it whenever you want to delete your `SharedPreferences` entry; for example, right before you call `open()` in `MainView`. You can also clear your `SharedPreferences` completely by long holding the app icon

in your app drawer, choose `App info > Storage & cache > Clear storage`. Note that this will also clear your `Google Signin` state in addition to your `SharedPreferences`.

You may also want to play with shorter or longer `chatterID` lifetime in the back end and see whether you're asked to login again when expected.

> ❗ Before moving on, make sure you delete the unencrypted `ChatterID` entry in your `SharedPreference` either by calling the `delete()` function above or by clearing your app storage on device per the instructions above. Else your app will **crash** when it gets an unencrypted entry when it expects an encrypted entry later in this lab.

Encrypting `ChatterID`

In this section we explore manually encrypting/decrypting our `SharedPreferences` entry. We will be using the Android `KeyStore` to hold our keys, with the keys used being tightly integrated with the biometric authentication mechanism. The following instructions are heavily based on Using BiometricPrompt with CryptoObject: how and why.

> ▶ The Android KeyStore trusted execution environment and EncryptedSharedPreferences

Let's create a new class named `ChatterKeyStore` and put it in a new eponymously named Kotlin file. We also define some constants we will be using in the class:

```
private const val KEY_STORE = "AndroidKeyStore"
private const val KEY_ALGORITHM = KeyProperties.KEY_ALGORITHM_AES
private const val KEY_BLOCK_MODE = KeyProperties.BLOCK_MODE_GCM
private const val KEY_PADDING = KeyProperties.ENCRYPTION_PADDING_NONE // GCM requires no padding

class ChatterKeyStore(val authenticate: Boolean) {
}
```

We will need two helper functions, `getKey()` and `createCipher()`. The method `getkey()` asks Android `KeyStore` to generate a key given your specifications. Put the following in your `ChatterKeyStore` class:

```
    private fun getKey(keyName: String): SecretKey {
        return KeyStore.getInstance(KEY_STORE)
            .apply {
                load(null)
            }.getKey(keyName, null) as? SecretKey
            ?: KeyGenerator.getInstance(KEY_ALGORITHM, KEY_STORE)
                .apply {
                    init(
                        KeyGenParameterSpec.Builder(keyName,
                            KeyProperties.PURPOSE_ENCRYPT or KeyProperties.PURPOSE_DECRYPT)
                            .setBlockModes(KEY_BLOCK_MODE)
                            .setEncryptionPaddings(KEY_PADDING)
                            .setKeySize(256)
```

```
                        .setUserAuthenticationRequired(authenticate)
                        .build()
                )
            }.generateKey()
    }
```

Given a key, you can then create a `Cipher` engine. You need one cipher to perform encryption and, for each decription operation, you'll need a different a cipher (due to the need to feed the initialization vector (IV) into the decryptor). Add the following `createCipher()` function to your `ChatterKeyStore` class:

```
    fun createCipher(keyName: String, iv: ByteArray? = null): Cipher {
        return Cipher.getInstance("$KEY_ALGORITHM/$KEY_BLOCK_MODE/$KEY_PADDING")
            .apply {
                iv?.let {
                    init(DECRYPT_MODE, getKey(keyName), GCMParameterSpec(128, it))
                } ?: run {
                    init(ENCRYPT_MODE, getKey(keyName))
                }
            }
    }
```

We're done with `ChattKeyStore.kt`.

To test your `KeyStore` functionalities, we now perform encryption/decryption of your `SharedPreferences` entries (still without biometric authentication). First, add the following variable declaration in your `ChatterID`, above the `open()` method definition:

```
    private val keyStore = ChatterKeyStore(authenticate = false) // false if encryption without auth
```

Replace the line `id = it.dropLast(INSTANT_LENGTH)` in the `context.getSharedPreferences()` code block of your `ChatterID.open()` with:

```
                val idVal = it.dropLast(INSTANT_LENGTH).toByteArray(ISO_8859_1)
                val iv = idVal.takeLast(IV_LENGTH).toByteArray()
                val idEnc = idVal.dropLast(IV_LENGTH).toByteArray()
                val decryptor = keyStore.createCipher(KEY_NAME, iv)
                id = String(decryptor.doFinal(idEnc))
```

and replace the line `val idVal = id+expiration.toString()` in your `ChatterID.save()` with:

```
            val encryptor: Cipher by lazy { keyStore.createCipher(KEY_NAME) }
            val idEnc = encryptor.doFinal(it.toByteArray(ISO_8859_1))
            val idVal = String(idEnc + encryptor.iv, ISO_8859_1)+expiration.toString()
```

Unlike `open()`, which we run only once upon app launch, we could be executing `save()` multiple times during the lifetime of the app, once every `chatterID` lifetime. Since we only need one instance of the `encryptor` cipher, we create it using the `lazy()` delegate.

▶ by lazy() delegation

Without going into the workings and interfacing of the cryptographic operations, one programmatic detail you want to be careful about is the conversion of `ByteArray` to `String` and back. Since encrypted byte array contains non-printable characters, the encoding that can preserve every byte intact is ISO-8859-1 used here (which, incidentally, is also used by Android's `Base64.encodeToString`.

You now have encrypted `SharedPreferences` entry! You should perform the same tests you did with your plaintext `SharedPreferences` implementation earlier. The encrypted version should behave exactly the same as the plaintext one.

Biometric check

To use biometric check, make sure that you have enabled screen lock and you have enrolled your biometric info with your device.

We are now ready to add biometric authentication.

⚠ First, change the declaration of your `MainActivity` in `MainActivity.kt` to inherit from `FragmentActivity()` (or `AppCompatActivity()`) instead of `ComponentActivity()`. The `AuthPrompt()` interface requires the context to be a `FragmentActivity`. It is not yet compatible with `ComponentActivity()`.

To enable authentication check, in your `ChattKeyStore.getKey()` method add the line:

```
                .setUserAuthenticationParameters(0, KeyProperties.AUTH_BIOMETRIC_STRONG
```

between `.setUserAuthenticationRequired(authenticate)` and `.build()`.

Then in your `ChatterID` object, find the declaration of `keyStore` and change its `authenticate` parameter to `true`. Right below the declaration of the `keyStore` property, add the following properties, exception handler, and the suspending `authenticate()` method:

```
    private lateinit var bioAuthPrompt: Class3BiometricOrCredentialAuthPrompt
    private lateinit var authPrompt: AuthPrompt

    private val authPromptExCatcher = CoroutineExceptionHandler { _, error ->
        authPrompt.cancelAuthentication()
        Log.e("AuthPrompt exception", error.localizedMessage ?: "authentication cancelled")
    }

    @ExperimentalCoroutinesApi
    private suspend fun Class3BiometricOrCredentialAuthPrompt.authenticate(host: AuthPromptHost, cry
        suspendCancellableCoroutine { cont ->
            authPrompt = startAuthentication(host, crypto, object : AuthPromptCallback() {
                override fun onAuthenticationSucceeded(
```

```
                activity: FragmentActivity?,
                result: BiometricPrompt.AuthenticationResult
            ) {
                super.onAuthenticationSucceeded(activity,result)
                cont.resume(result, null)
            }

            override fun onAuthenticationError(activity: FragmentActivity?, @BiometricPrompt.Aut
                super.onAuthenticationError(activity, errorCode, errString)
                if (errorCode == BiometricPrompt.ERROR_USER_CANCELED) {
                    cont.resume(null, null)
                }
            }
            // default to super for onAuthenticationFailed()
        })
        cont.invokeOnCancellation { authPrompt.cancelAuthentication() }
    }
```

The `authenticate()` extension converts the callback-based `startAuthentication()` to a suspending function by using `suspendCancellableCoroutine()`. The function `suspendCancellableCoroutine()` (and its cousin `suspendCoroutine()`) returns the management of continuation, that has been handled by the compiler, back to the developer.

We declare the `authenticate()` extension private to `ChatterID` instead of exposing it publicly in `Extensions.kt` because it updates `authPrompt`, which we'd like to keep private to `ChatterID`. In the exception handler `authPromptExCatcher`, we cancel the failed authentication through `authPrompt` which was populated by `startAuthentication()` in `authenticate()`.

▶ authenticate() by biometric-ktx:1.2.0-alpha03

Here's the modified `open()` to perform biometric check, to replace the one you have in `ChatterID`:

```
@ExperimentalCoroutinesApi
suspend fun open(context: Context) {
    if (expiration != Instant.EPOCH) { // this is not first launch
        return
    }

    val status = BiometricManager.from(context).canAuthenticate(BIOMETRIC_STRONG)
    if (status != BiometricManager.BIOMETRIC_SUCCESS) {
        context.toast("Skipping biometric authentication ($status)")
        return
    }

    bioAuthPrompt = Class3BiometricOrCredentialAuthPrompt.Builder("Biometric ID").apply {
        setSubtitle("for kotlinChatter")
        setDescription("To manage persistent ChatterID")
        setConfirmationRequired(false)
    }.build()

    context.getSharedPreferences(ID_FILE, Context.MODE_PRIVATE)
        .getString(KEY_NAME, null)?.let {
```

```
            val idExp = Instant.parse(it.takeLast(INSTANT_LENGTH))

            val idVal = it.dropLast(INSTANT_LENGTH).toByteArray(ISO_8859_1)
            val iv = idVal.takeLast(IV_LENGTH).toByteArray()
            val idEnc = idVal.dropLast(IV_LENGTH).toByteArray()
            val decryptor = keyStore.createCipher(KEY_NAME, iv)

            val cryptoObject = BiometricPrompt.CryptoObject(decryptor)
            withContext(authPromptExCatcher) {
                val authResult = bioAuthPrompt.authenticate(
                    AuthPromptHost(context as FragmentActivity),
                    cryptoObject
                )
                authResult?.cryptoObject?.cipher?.run {
                    id = String(doFinal(idEnc))
                    expiration = idExp
                } ?: run {
                    context.toast("KeyStore not read")
                }

            }
        }
}
```

When we call `open()` at the very beginning of our app run, we check that the device has biometric authentication enabled. We prepare the biometric prompt panel that will be used every time we do a biometric check, in both `open()` and `save()`. `Class3` biometric prompt allows us to perform biometric authenticated cryptographic operation. We have opted to enable fall-back to PIN or pattern in lieu of biometric authentication, you could choose otherwise with `Class3BiometricAuthPrompt()` instead.

And here's the modified `save()` to perform biometric check, to replace the one in your `ChatterID`. It should look similar enough to `open()`, except that we're doing encryption instead of decryption here:

```
@ExperimentalCoroutinesApi
suspend fun save(context: Context) {
    val status = BiometricManager.from(context).canAuthenticate(BIOMETRIC_STRONG)
    if (status != BiometricManager.BIOMETRIC_SUCCESS) {
        context.toast("Skipping biometric authentication ($status)")
        return
    }

    id?.let {
        val encryptor: Cipher by lazy { keyStore.createCipher(KEY_NAME) }

        val cryptoObject = BiometricPrompt.CryptoObject(encryptor)
        withContext(authPromptExCatcher) {
            val authResult = bioAuthPrompt.authenticate(
                AuthPromptHost(context as FragmentActivity),
                cryptoObject
            )
            authResult?.cryptoObject?.cipher?.run {
                val idEnc = doFinal(it.toByteArray(ISO_8859_1))
                val idVal = String(idEnc + iv, ISO_8859_1) + expiration.toString()

                context.getSharedPreferences(ID_FILE, Context.MODE_PRIVATE)
                    .edit().putString(KEY_NAME, idVal).apply()
```

```
                } ?: run {
                    context.toast("KeyStore not updated")
                }
            }
        }
    }
```

And with that, we're done with lab4 and all the labs for the course! Don't forget to test the last version of your app with biometric authentication as you did earlier versions.

Testing

To use the Android emulator to simulate biometric check, follow the instructions in our Getting Started with Android Development.

You may need to sign out from your Google account on your device to test Google Sign-In. **On your development host** (**not** on your device), open up your browser and go to a Google property, e.g., Gmail. At the upper right of your browser, click on your avatar icon and tap on `Manage your Google Account` button on the drop-down menu. Once in your `Google Account`, click on the `Security` menu on the left. In `Security`, scroll down until you see `Your devices` card. At the bottom of the card, click on `Manage your devices`. Find your device and click on the 3-vertical dot menu and select `Sign out`.

Submission guidelines

Unlike in previous labs, there are a few **CRUCIAL** extra step to do before you push your lab to GitHub:

- Copy `debug.keystore` in (`~/.android/` for Mac or `C:\Users\<CurrentUser>\.android\` for Windows) to your `lab4` folder.
- Put a copy of your `SHA1 certificate` (in the format of `xx:xx:xx:...`) you used to obtain your Client ID in the `README.md` file in your `lab4` folder.
- Make sure you have completed the back-end part and have pushed your changes to the back-end code to your `441` GitHub repo.

Without these we won't be able to run your app.

> **IMPORTANT:** If you work in team, remember to put your team mate's uniqnames in lab4 folder's `README.md` so that we'd know. Otherwise, we could mistakenly thought that you were cheating and accidentally report you to the Honor Council, which would be a hassle to undo.

Enter your uniqname (and that of your team mate's) and the link to your GitHub repo on the Lab Links sheet. The request for teaming information is redundant by design. If you're using a different GitHub repo from previous lab's, invite `eecs441staff@umich.edu` to your GitHub repo.

Push your lab4 to its GitHub repo as set up at the start of this spec. Using GitHub Desktop to do this, you can follow the steps below:

- Open GitHub Desktop and click on `Current Repository` on the top left of the interface
- Click on your `441` GitHub repo
- Add Summary to your changes and click `Commit to master`
- If you have a team mate and they have pushed changes to GitHub, you'll have to click `Pull Origin` and resolve any conflicts before . . .
- Finally click on `Push Origin` to push changes to GitHub

Go to GitHub website to confirm that your project files for lab4 have been uploaded to GitHub repo under folder `lab4` .

References

- Add Google Sign-in to Your Android App
- GoogleSignIn.silentSignIn
- The overall Google Identity Platform
- Object (Singleton)
- Instant Android instantaneous point on the time-line.
- Android Views in Compose
  - AndroidView
  - AndroidViewBinding
- Compose in Android Views "You must attach the ComposeView to a ViewTreeLifecycleOwner. The ViewTreeLifecycleOwner allows the view to be attached and detached repeatedly while preserving the composition. ComponentActivity, FragmentActivity and AppCompatActivity are all examples of classes that implement ViewTreeLifecycleOwner."

Coroutines

- How to make POST, GET, PUT and DELETE requests with Retrofit using Kotlin
- Kotlin Coroutines and Retrofit – A Practical Approach to Consuming REST APIs in Android by `MainScope
- suspendCancellableCoroutine

`SharedPreferences`

- SharedPreferences is your answer to simple storage
- Shared Preferences
- getSharedPreferences
- Shared Preferences Editor
- EncryptedSharedPreferences isUserAuthenticationRequired not working properly

Biometric Authentication

- Using BiometricPrompt with CryptoObject: how and why
- How you should secure your Android's app biometric authentication
- Show a biometric authentication dialog
- What is the difference between CBC and GCM mode?
- What exactly does the "NoPadding" parameter do in the Cipher class?
- Class3BiometricOrCredentialAuthPrompt.authenticate
- BiometricPrompt in android is not supporting exit on back button

Appendix: imports

Prepared for EECS 441 by Benjamin Brengman, Alexander Wu, Ollie Elmgren, Wendan Jiang, Yibo Pi, and Sugih Jamin

Last updated: August 3rd, 2021