Application Development II

420-5A6-AB

Instructor: Talib Hussain

Day 24:

Firebase Auth

Firebase Authentication

- Use the official docs to setup your basic configuration:
  - https://firebase.google.com/docs/auth/android/firebaseui

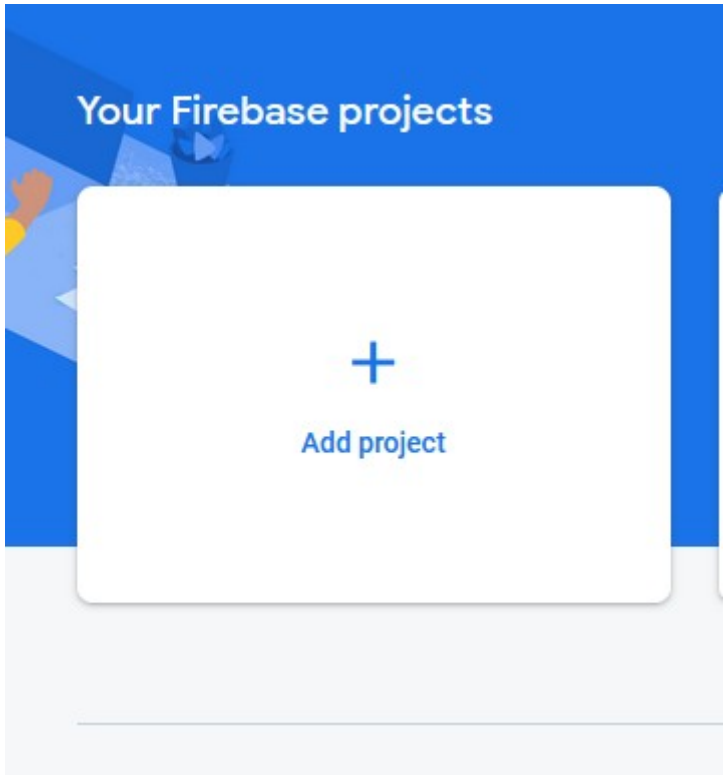
- These other links don't really have a solutions that are easy to understand and get working. But, you can consult them for ideas.
  - Simple walkthrough for setting up and using Firebase Authentication.
    - https://www.composables.com/tutorials/firebase-auth

  - A more complex tutorial from Google. Nte: Uses Hilt for dependency injection
    - https://developers.google.com/learn/pathways/firebase-android-jetpack
      - Steps 2 – 4 are most relevant
      - https://firebase.blog/posts/2022/04/building-an-app-android-jetpack-compose-firebase
      - https://firebase.blog/posts/2022/05/adding-firebase-auth-to-jetpack-compose-app
      - https://firebase.blog/posts/2022/07/adding-cloud-firestore-to-jetpack-compose-app

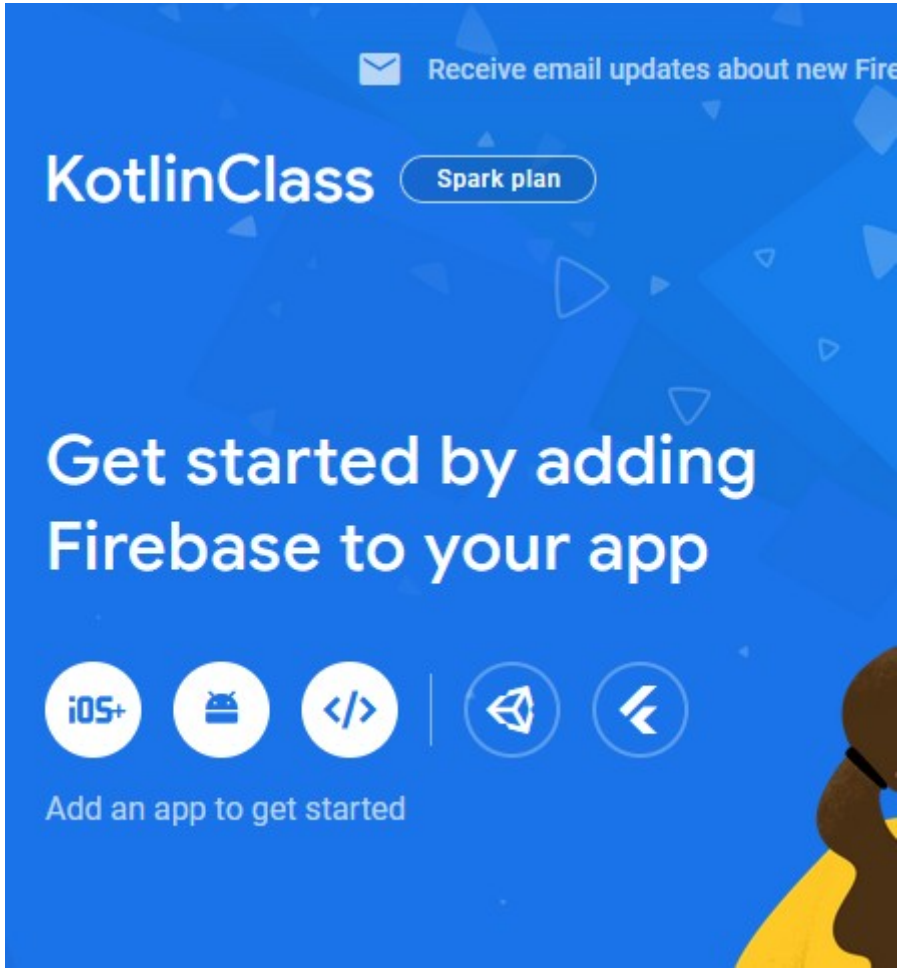
  - This codelab skips some details since it provides some code. But, may be a useful reference (up to step 4).
    - https://firebase.google.com/codelabs/build-android-app-with-firebase-compose#3

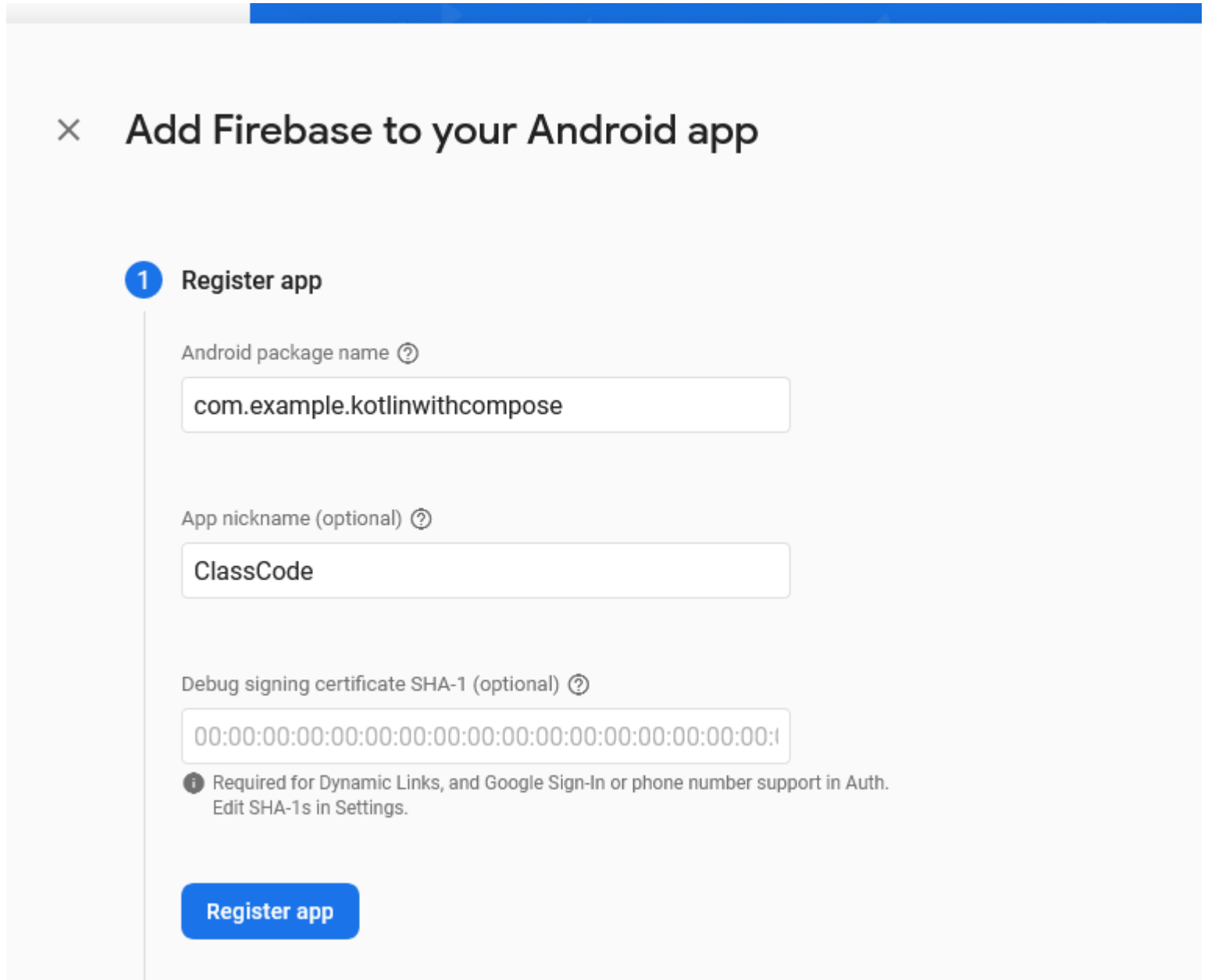
Add Project in Firebase Console

- https://console.firebase.google.com/u/0/

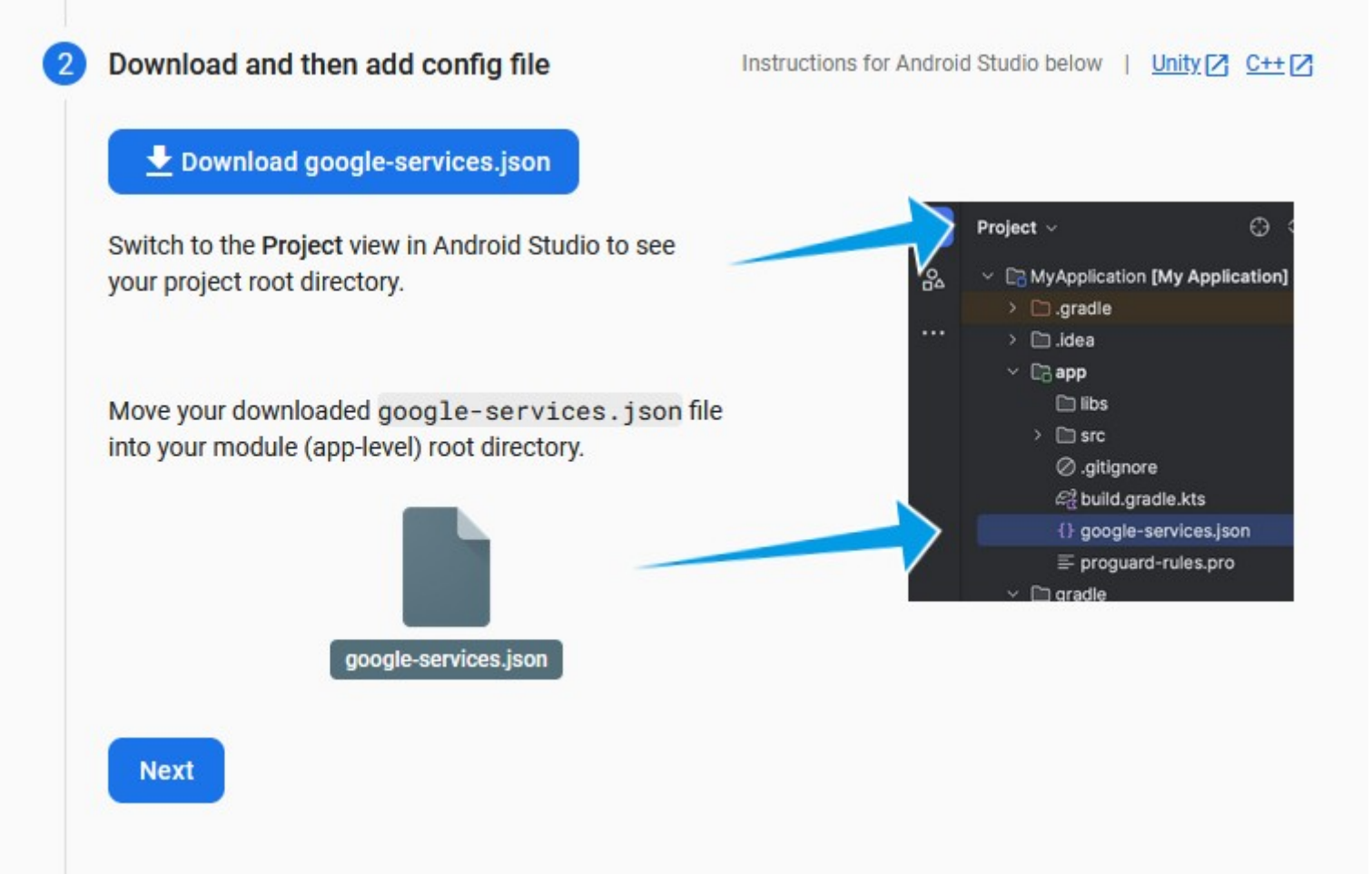


Click on Android button




Add Firebase to your Android app
1
Register app
Android package name
com.example.kotlinwithcompose
App nickname (optional)
ClassCode
Debug signing certificate SHA-1 (optional)
00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00
Required for Dynamic Links, and Google Sign-In or phone number support in Auth.
Edit SHA-1s in Settings.
Register app

- Register App
- Download google-services.json
- Put it in the app folder of your project



Root level Gradle Change

3 **Add Firebase SDK**

Instructions for Gradle | Unity C++

★ Are you still using the `buildscript` syntax to manage plugins? Learn how to add Firebase plugins using that syntax.

1. To make the `google-services.json` config values accessible to Firebase SDKs, you need the Google services Gradle plugin.

(•) Kotlin DSL (build.gradle.kts) ( ) Groovy (build.gradle)

Add the plugin as a dependency to your **project-level** `build.gradle.kts` file:

Root-level (project-level) Gradle file (`<project>/build.gradle.kts`):

```
plugins {
  // ...

  // Add the dependency for the Google services Gradle plugin
  id("com.google.gms.google-services") version "4.4.0" apply false
}
```

Module Level Gradle Change

2. Then, in your **module (app-level)** `build.gradle.kts` file, add both the `google-services` plugin and any Firebase SDKs that you want to use in your app:

Module (app-level) Gradle file (`<project>/<app-module>/build.gradle.kts`):

```
plugins {
  id("com.android.application")
  // Add the Google services Gradle plugin
  id("com.google.gms.google-services")
  ...
  }

dependencies {
  // Import the Firebase BoM
  implementation(platform("com.google.firebase:firebase-bom:32.3.1"))

  // TODO: Add the dependencies for Firebase products you want to use
  // When using the BoM, don't specify versions in Firebase dependencies
  // https://firebase.google.com/docs/android/setup#available-libraries
}
```

By using the Firebase Android BoM, your app will always use compatible Firebase library versions. Learn more

3. After adding the plugin and the desired SDKs, sync your Android project with Gradle files.

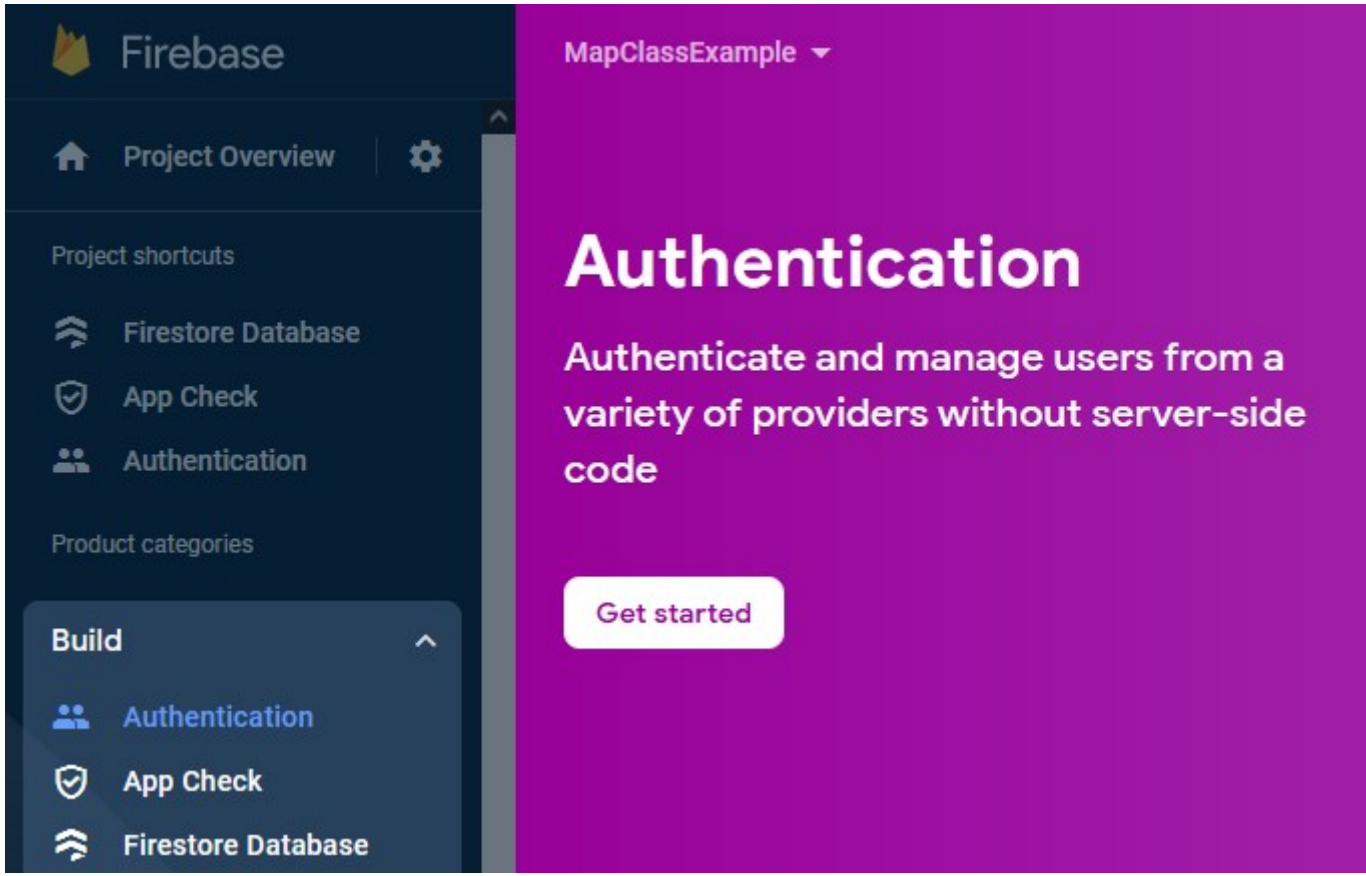
Previous Next

2. Add the dependencies for FirebaseUI to your app-level `build.gradle` file. If you want to support sign-in with Facebook or Twitter, also include the Facebook and Twitter SDKs:

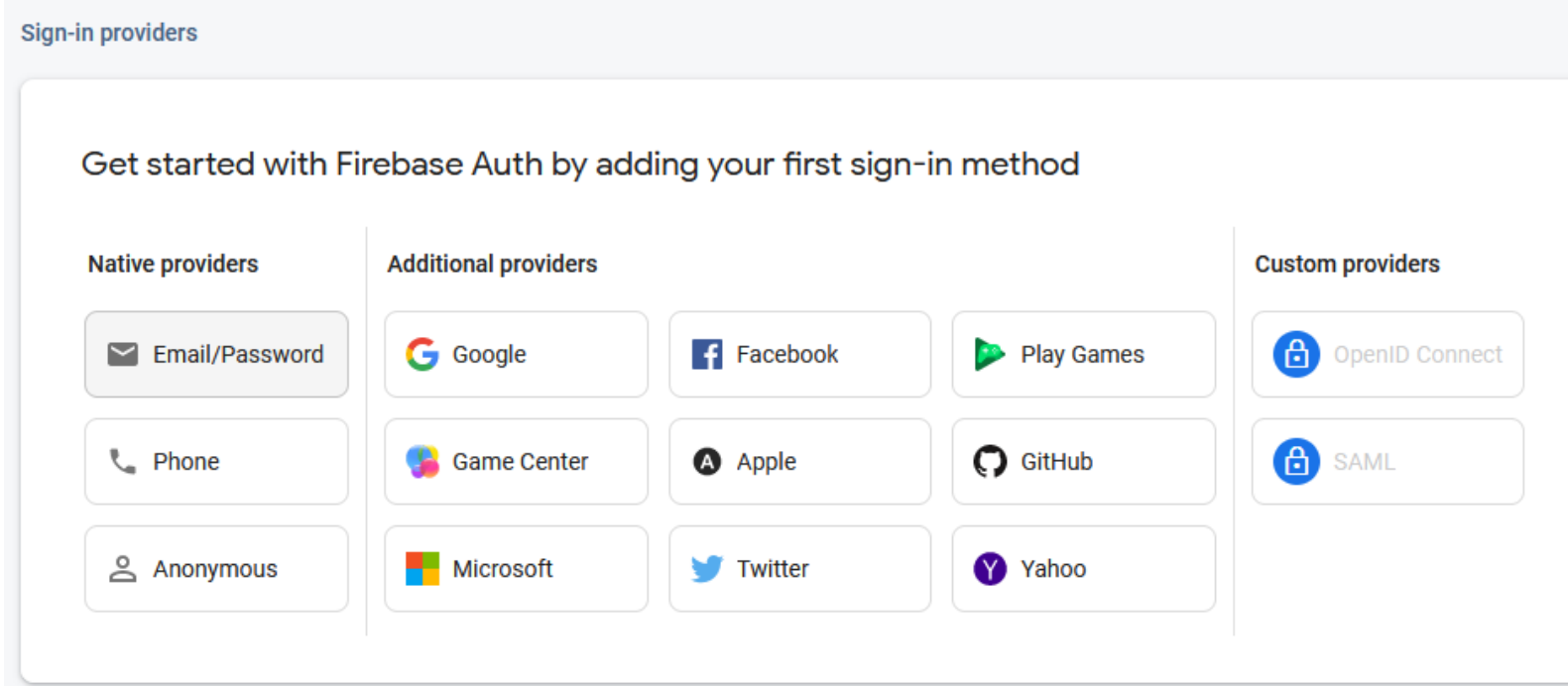
```
dependencies {
    // ...

    implementation 'com.firebaseui:firebase-ui-auth:7.2.0'
```

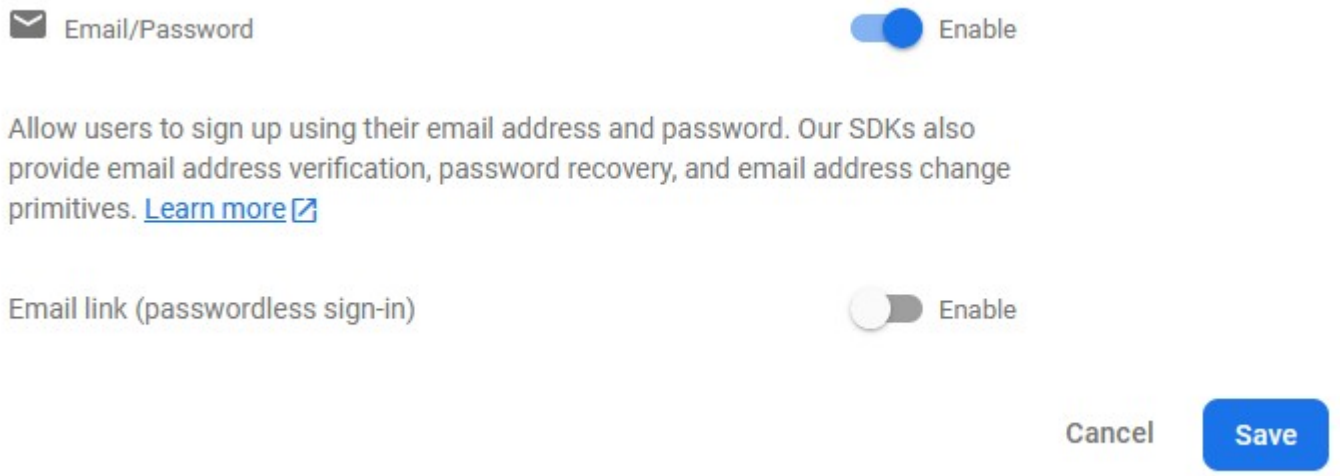
Setup on Firebase online

- Go to your console.firebase.google.com and go to the authentication page in the build menu on the left
- Click Get Started



- Choose the email/password provider and enable it.





- Teacher code with successful sign in with Email/Password is in firebaseAuth branch

Auth with email/password

- 1. Create a User class that stores user email
- 2. Create AuthRepository interface
  - currentUser(), signUp(), signIn(), signOut(), delete()
  - currentUser as a function (not a state variable) so that we have a consistent contract
- 3. Create AuthRepositoryFirebase that implements the interface
- 4. Create an AuthViewModel that accepts an AuthRepository
  - To prevent the composable functions from knowing anything about the business logic, we are going to call the Firebase Authentication API methods from the ViewModels.
- 5. Perform manual dependency injection
  - Add authRepository variable in AppModule
  - Create AuthViewModelFactory that calls Firebase.auth to instantiate
- 6. Create a composable LoginScreen that will use the ViewModel (with the factory)

Generalized Interface

- The approach below provides us with a repository that is not dependent on any particular database implementation we decide to use
  - User is our own class. Unlike, for example, FirebaseUser which is what some Firebase operations return.

```
data class User(var email: String)



interface AuthRepository {
    // Return a StateFlow so that the composable can always update when
    //   the current authorized user status changes for any reason
    fun currentUser() : StateFlow<User?>

    suspend fun signUp(email: String, password: String): Boolean

    suspend fun signIn(email: String, password: String): Boolean

    fun signOut()

    suspend fun delete()
}
```

AuthRepositoryFirebase

- Need to inject the FirebaseAuth object
- Need the initialize a flow when the repository is created so that it listens to all changes on a MutableStateFlow.
- Note that we need to convert FirebaseUser to User inside our flow since we want to pass a flow of User to our viewModel
  - i.e., the viewModel should never know about FirebaseUser

```
class AuthRepositoryFirebase(private val auth: FirebaseAuth) : AuthRepository {
    private val currentUserStateFlow = MutableStateFlow(auth.currentUser?.toUser())

    init {
        auth.addAuthStateListener { firebaseAuth ->
            currentUserStateFlow.value = firebaseAuth.currentUser?.toUser()
        }
    }
    override fun currentUser(): StateFlow<User?> {
        return currentUserStateFlow
    }
```

Helper function to convert from FirebaseUser to User

- We can put this in our AuthRepositoryFirebase as a private function since no other parts of the program ever need to use it.

```
/** Convert from FirebaseUser to User */
  private fun FirebaseUser?.toUser(): User? {
      return this?.let {
          if (it.email==null) null else
          User(
              email = it.email!!,
          )
      }
  }
```

Main operations

```
override suspend fun signUp(email: String, password: String): Boolean {
    return try {
        auth.createUserWithEmailAndPassword(email, password).await()
        return true;
    } catch (e: Exception) {
        return false;
    }
}

override suspend fun signIn(email: String, password: String): Boolean {
    return try {
        auth.signInWithEmailAndPassword(email, password).await()
        return true;
    } catch (e: Exception) {
        return false;
    }
}
```

```
override fun signOut() {
    return auth.signOut()
}

override suspend fun delete() {
    if (auth.currentUser != null) {
        auth.currentUser!!.delete()
    }
}
```

AuthViewModel

```
class AuthViewModel(private val authRepository: AuthRepository) : ViewModel() {
    // Return a StateFlow so that the composable can always update
    // based when the value changes
    fun currentUser(): StateFlow<User?> {
        return authRepository.currentUser()
    }

    fun signUp(email: String, password: String) {
        viewModelScope.launch {
            authRepository.signUp(email, password)
        }
    }
    fun signIn(email: String, password: String) {
        viewModelScope.launch {
            authRepository.signIn(email, password)
        }
    }
    fun signOut() {
        authRepository.signOut()
    }
    fun delete() {
        viewModelScope.launch {
            authRepository.delete()
        }
    }
```

AuthViewModelFactory

```
/* ViewModel Factory that will create our view model by injecting the
     authRepository from the module.
 */
class AuthViewModelFactory : ViewModelProvider.Factory {
    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        return AuthViewModel(MyApp.appModule.authRepository) as T
    }
}
```

Manual Dependency Injection

```
class AppModule(
    private val appContext: Context
) {
    /* Create appropriate repository (backed by a DataStore) on first use.
       Only one copy will be created during lifetime of the application. */
    val profileRepository : ProfileRepository by lazy {
        ProfileRepositoryDataStore(appContext)
    }
    val authRepository : AuthRepository by lazy {
        AuthRepositoryFirebase(Firebase.auth) // inject Firebase auth
    }
}
```

AuthLoginScreen

```
@Composable
fun AuthLoginScreen(authViewModel: AuthViewModel =
                        viewModel(factory= AuthViewModelFactory())
    ) {
    val userState = authViewModel.currentUser().collectAsState()

    Column {
        if (userState.value == null) {
            Text("Not logged in")
            Button(onClick = {
                authViewModel.signUp("myname@name.com", "Abcd1234!")
            }) {
                Text("Sign up via email")
            }
            Button(onClick = {
                authViewModel.signIn("myname@name.com", "Abcd1234!")
            }) {
                Text("Sign in via email")
            }
        } else {
            if (userState.value==null)
                Text("Please sign in")
            else
                Text("Welcome ${userState.value!!.email}")
            Button(onClick = {
                authViewModel.signOut()
            }) {
                Text("Sign out")
            }
            Button(onClick = {
                authViewModel.delete()
            }) {
                Text("Delete account")
            }
        }
    }
}
```

Test It Out

- Try adding and removing different users and seeing what happens in your Firebase console

Dispatchers

- By default, Kotlin will run your asynchronous routines in the main thread – the same one that your UI is running on
- It is considered best practice to inject dispatchers into your ViewModel
- A dispatcher will run suspend functions in a separate thread. There are 3 available dispatchers:
  - Dispatchers.Main - Use this dispatcher to run a coroutine on the main Android thread. This should be used only for interacting with the UI and performing quick work. Examples include calling suspend functions, running Android UI framework operations, and updating LiveData objects.
  - Dispatchers.IO - This dispatcher is optimized to perform disk or network I/O outside of the main thread. Examples include using the Room component, reading from or writing to files, and running any network operations.
  - Dispatchers.Default - This dispatcher is optimized to perform CPU-intensive work outside of the main thread. Example use cases include sorting a list and parsing JSON.
- Easy to do – just pass a Dispatcher to the launch function
  - Usually will use Dispatchers.IO to take work off the Main thread

```
viewModelScope.launch(Dispatchers.IO) {
  …
}
```

- Inside a suspend fun, you can also specify that a particular block of code will run on a different thread using withContext. E.g.,

```
// Dispatchers.Main
suspend fun get(url: String) =
   // Dispatchers.Main
   withContext(Dispatchers.IO) {
      // Dispatchers.IO
      /* perform blocking network IO here */
   }
   // Dispatchers.Main
```

- https://kotlinlang.org/docs/coroutine-context-and-dispatchers.html#dispatchers-and-threads
- https://dev.to/theplebdev/android-notes-understanding-viewmodelscopelaunch-230f
- https://developer.android.com/kotlin/coroutines/coroutines-adv
- https://medium.com/androiddevelopers/coroutines-on-android-part-i-getting-the-background-3e0e54d20bb

```
class AuthViewModel(private val authRepository: AuthRepository) : ViewModel() {
    // Return a StateFlow so that the composable can always update
    // based when the value changes
    fun currentUser(): StateFlow<User?> {
        return authRepository.currentUser()
    }

    fun signUp(email: String, password: String) {
        viewModelScope.launch(Dispatchers.IO) {
            authRepository.signUp(email, password)
        }
    }
    fun signIn(email: String, password: String) {
        viewModelScope.launch(Dispatchers.IO) {
            authRepository.signIn(email, password)
        }
    }
    fun signOut() {
        authRepository.signOut()
    }
    fun delete() {
        viewModelScope.launch(Dispatchers.IO) {
            authRepository.delete()
        }
    }
}
```

Use explicit result class for nuanced output

- Can use a sealed class for success and failure cases
- Can provide output, e.g., via a snackbar, to indicate when user actions are successful or not.

- Teacher code with use of sealed class for results is in firebaseAuthWithResults branch

- We want to distinguish between the case where there is an actual error (Failure), where the operation completed with desired outcome Success(true)(and where it completed without the desired outcome Success(false)
- We also want a special state that represents that no pertinent action is in progress

```
sealed class ResultAuth<out T> {
    data class Success<out T>(val data: T) : ResultAuth<T>()
    data class Failure(val exception: Throwable) : ResultAuth<Nothing>()
    object Inactive : ResultAuth<Nothing>()
    object InProgress : ResultAuth<Nothing>()
}
```

In AuthViewModel

- Create a StateFlow that will store the result of a call to the repository.
- Here is an example for sign up. Repeat similarly for the other cases.

```
private val _signUpResult = MutableStateFlow<ResultAuth<Boolean>?>(ResultAuth.Inactive)
val signUpResult: StateFlow<ResultAuth<Boolean>?> = _signUpResult

fun signUp(email: String, password: String) {
    _signUpResult.value = ResultAuth.InProgress
    viewModelScope.launch(Dispatchers.IO) {
        delay(3000) // TODO: Remove.  Only here to demonstrate inprogress snackbar
        try {
            val success = authRepository.signUp(email, password)
            _signUpResult.value = ResultAuth.Success(success)
        } catch (e: FirebaseAuthException) {
            _signUpResult.value = ResultAuth.Failure(e)
        } finally {
            // Reset the others since they are no longer applicable
            _signInResult.value = ResultAuth.Inactive
            _signOutResult.value = ResultAuth.Inactive
            _deleteAccountResult.value = ResultAuth.Inactive
        }
    }
}
```

In AuthScreen

- In our composable, we want to be able to show a snackbar on success or failure (or while waiting)
- For this, we need to use a launched effect that only is triggered when the result value changes.
- Here is an example for sign up. Repeat similarly for the other cases

```
val signUpResult by authViewModel.signUpResult.collectAsState(ResultAuth.Inactive)
val snackbarHostState = remember { SnackbarHostState() } // Material 3 approach

// Show a Snackbar when sign-up is successful, etc.
LaunchedEffect(signUpResult) {
    signUpResult?.let {
        if (it is ResultAuth.Inactive) {
            return@LaunchedEffect
        }
        if (it is ResultAuth.InProgress) {
            snackbarHostState.showSnackbar("Sign-up In Progress")
            return@LaunchedEffect
        }
        if (it is ResultAuth.Success && it.data) {
            snackbarHostState.showSnackbar("Sign-up Successful")
        } else if (it is ResultAuth.Failure || it is ResultAuth.Success) { // success(false) case
            snackbarHostState.showSnackbar("Sign-up Unsuccessful")
        }
    }
}
```