Application Development II

420-5A6-AB

Instructor: Talib Hussain

Day 27:

Firestore CRUD

"Deploying" for Testing...(Firebase Hosting)

- Want to deploy, but gets complication with Play store (need developer account, approvals process, etc.)
- Instead we want to "deploy" to Firebase to support user testing.
  - https://firebase.google.com/docs/hosting/github-integration
- Breaks down the steps:
  - https://blog.logrocket.com/android-ci-cd-using-github-actions/
- Some details on secrets and explains a few things more clearly
  - https://proandroiddev.com/create-android-release-using-github-actions-c052006f6b0b
- https://www.kodeco.com/19407406-continuous-delivery-for-android-using-github-actions
- https://dustn.dev/post/2022-02-21-build-a-cicd-pipeline-using-github-actions/
- Actually running the test app
  - https://quickresource.quickseries.com/knowledge-base/installing-your-test-app-on-android-firebase/

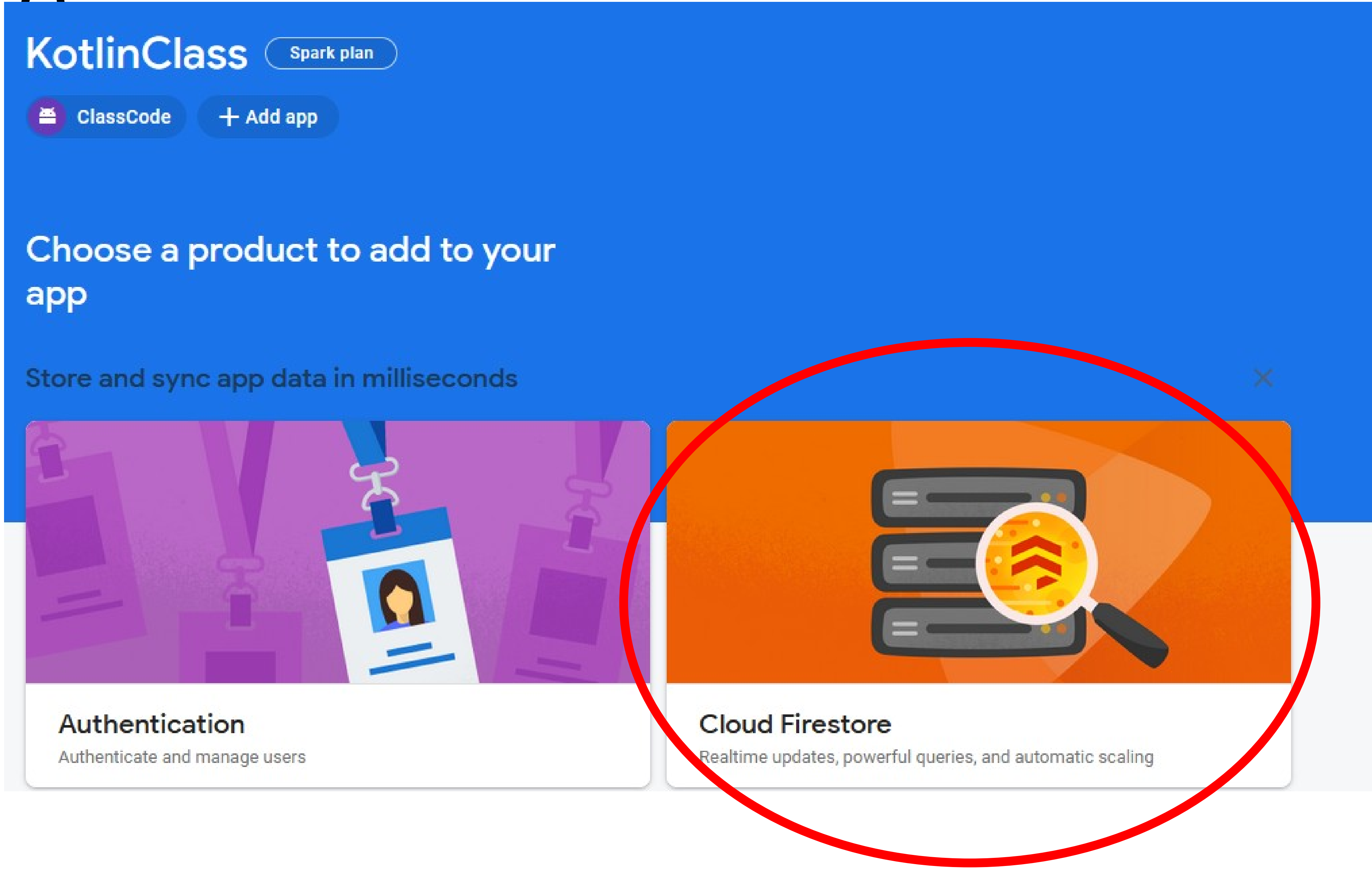
Firestore CRUD

- Haven't found a great resource yet, but this link seems reasonable (go to Part II, though it does use Hilt). Will try to work this up for you in slides later.
  - https://medium.com/@emmanuelmuturia/firebase-in-jetpack-compose-authentication-adding-data-to-cloud-firestore-a6a8e5ebee19
- Other links, but may be misleading
  - https://developer.android.com/kotlin/ktx
  - https://firebase.google.com/docs/firestore/quickstart#kotlin+ktx_1
  - https://firebase.google.com/docs/firestore/query-data/get-data

- In module build.gradle.kts, add Firebase Cloud Firestore dependency
    - implementation("com.google.firebase:firebase-firestore-ktx:24.6.0")

Add Firestore

- In your project page on console.firebase.google.com, add "Cloud Firestore"



Choose a product to add to your app

Store and sync app data in milliseconds

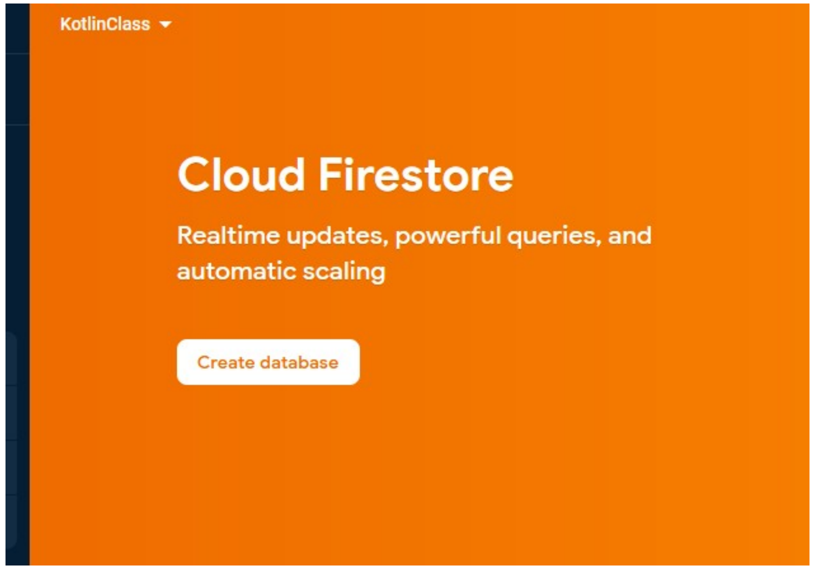
Authentication

Authenticate and manage users

Cloud Firestore

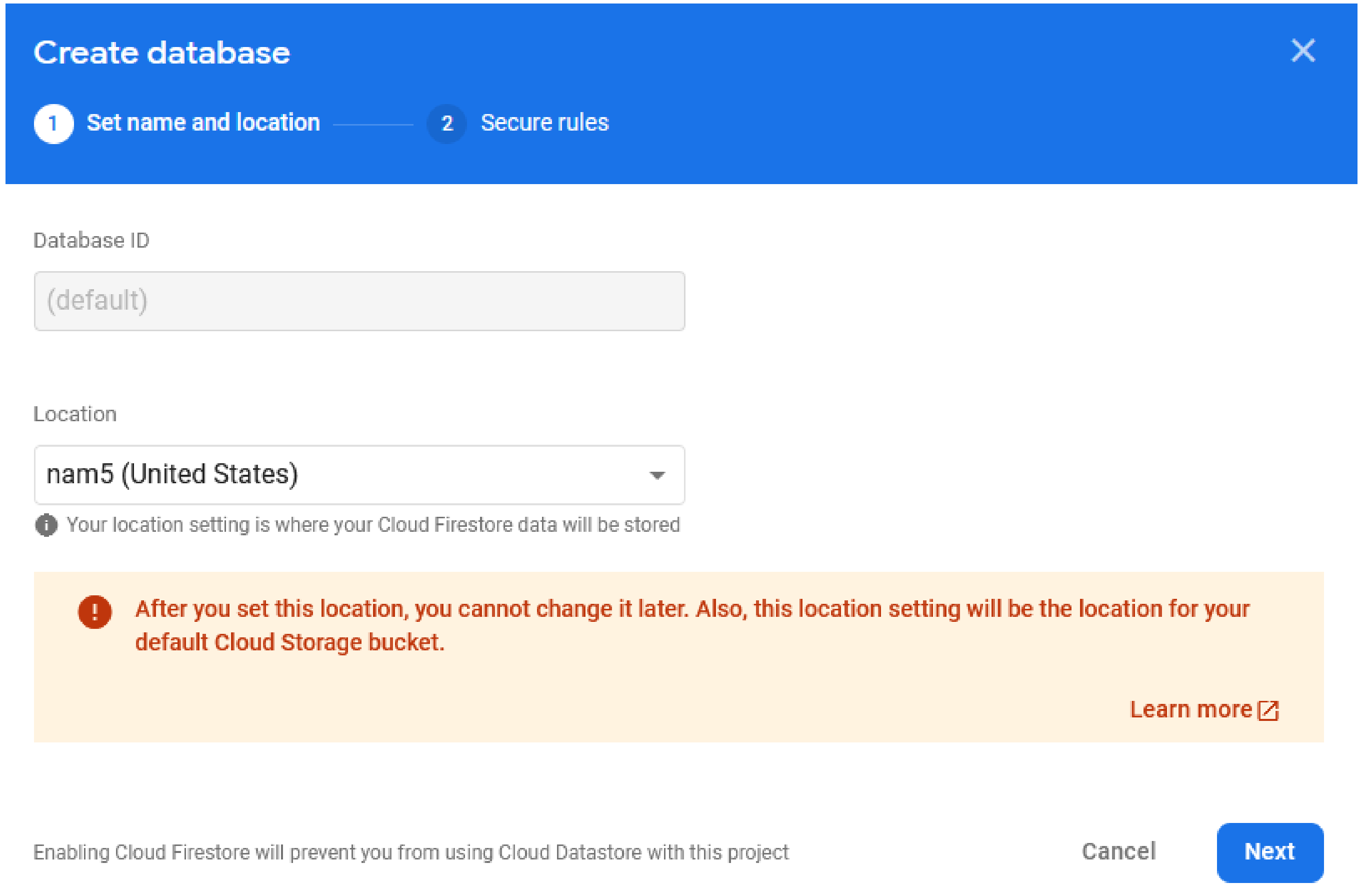
Realtime updates, powerful queries, and automatic scaling

- Create Database


KotlinClass

Cloud Firestore

Realtime updates, powerful queries, and automatic scaling


Create database

Database ID

(default)

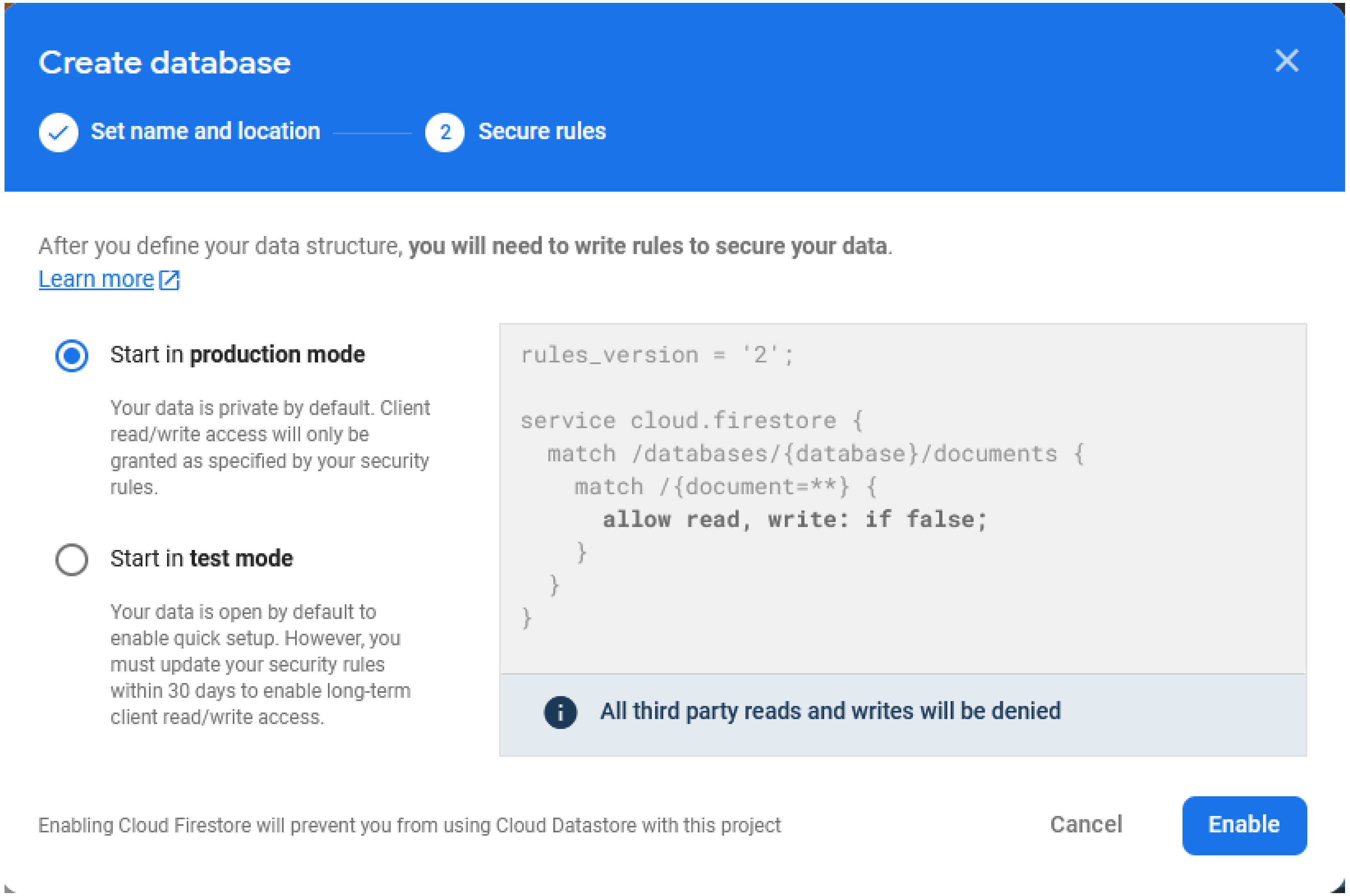
Location

nam5 (United States)

Your location setting is where your Cloud Firestore data will be stored

After you set this location, you cannot change it later. Also, this location setting will be the location for your default Cloud Storage bucket.

Enabling Cloud Firestore will prevent you from using Cloud Datastore with this project


Create database

Set name and location — 2 Secure rules

After you define your data structure, **you will need to write rules to secure your data**.
Learn more

[✓ Start in **production mode**]

Your data is private by default. Client read/write access will only be granted as specified by your security rules.

Start in **test mode**

Your data is open by default to enable quick setup. However, you must update your security rules within 30 days to enable long-term client read/write access.

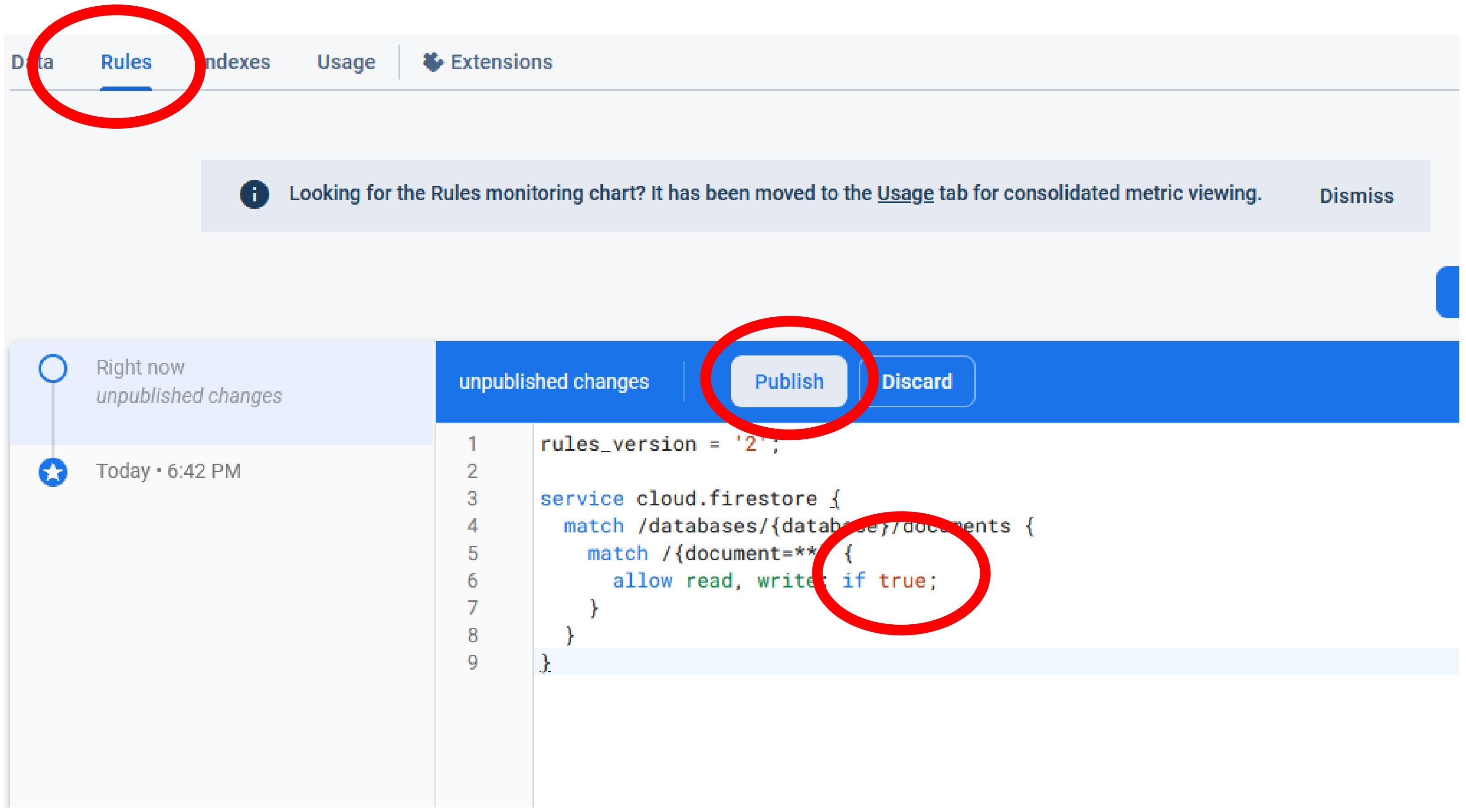
```
rules_version = '2';

service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if false;
    }
  }
}
```

All third party reads and writes will be denied

Enabling Cloud Firestore will prevent you from using Cloud Datastore with this project

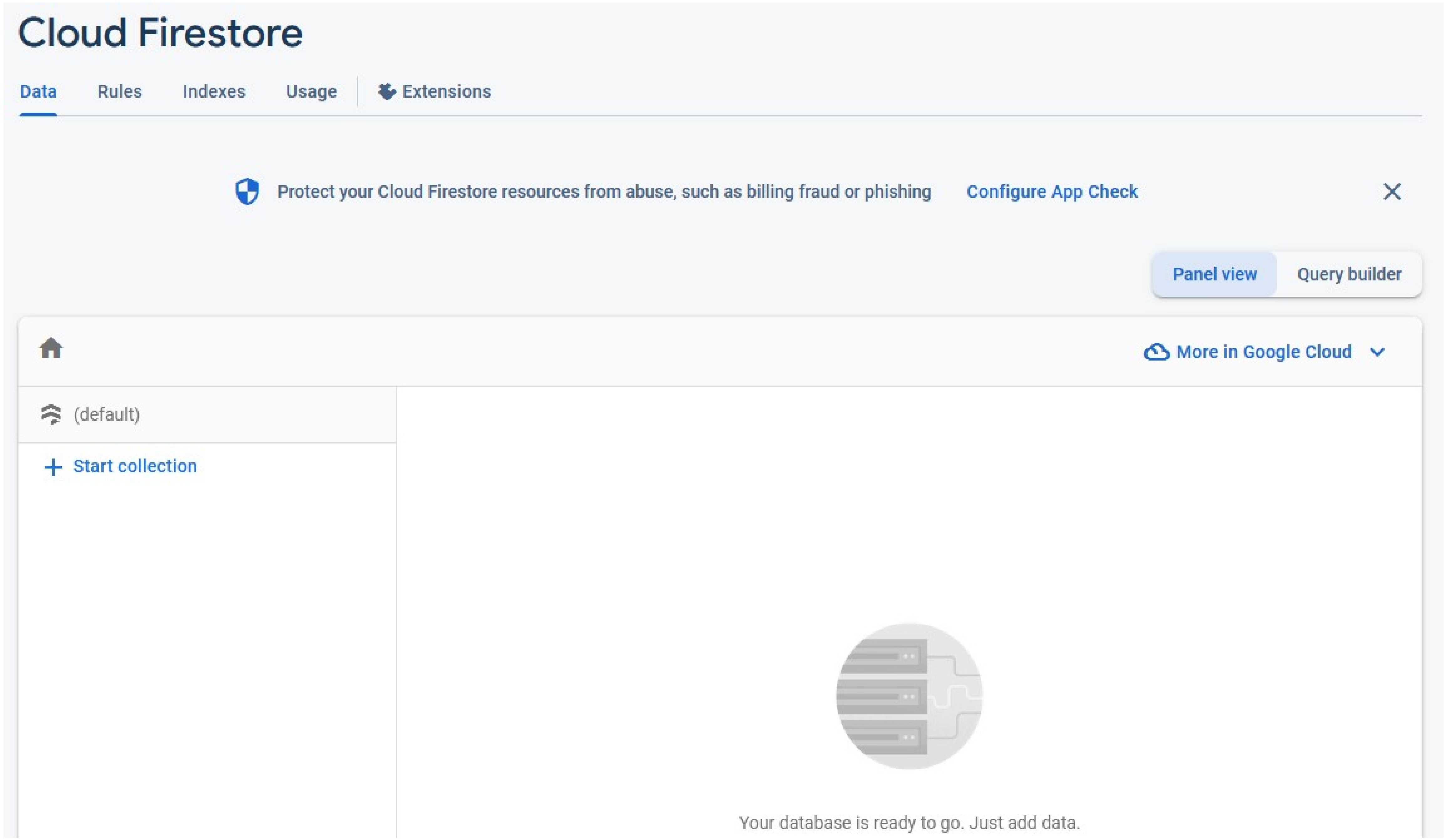
Allow Access

- For now, give open access to your database
  - In the rules tab, set the rule to "if true"
  - Then publish the rule.



Looking for the Rules monitoring chart? It has been moved to the Usage tab for consolidated metric viewing.

```
rules_version = '2';

service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if true;
    }
  }
}
```



Your database is ready to go. Just add data.

ProfileRepositoryFirestore

- Let's use Firestore to implement our ProfileRepository
- Inject in AppModule:

```
 val profileRepository : ProfileRepository by lazy {
       ProfileRepositoryFirestore(FirebaseFirestore.getInstance())
   }
```

- Create ProfileRepositoryFirestore:

```
class ProfileRepositoryFirestore (val db: FirebaseFirestore) : ProfileRepository {
   val dbProfile: CollectionReference = db.collection("Profile")


}
```

Save Profile (using set)

- https://firebase.google.com/docs/firestore/manage-data/add-data
- When you use set() to create a document, you must specify an ID for the document to create
  - e.g., db.collection("cities").document("my-city-id").set(data)
- We are storing only a single profile at a time, so let's use a unique document name to refer to it

```
val profileId = "main-profile"
  override suspend fun saveProfile(profileData: ProfileData) {
      dbProfile.document(profileId).set(profileData)
  }
```

- set() will also update the document with the given id if it already exists.

- (Note: There is also an "add()" function. When you use add() to create a document, Cloud Firestore auto-generates an ID for you)

Listeners

- We can define listeners to handle successful and failed calls to the db.
  - addOnSuccessListener, addOnFailureListener

```
dbProfile.document(profileId).set(profileData)
        .addOnSuccessListener {
           println("Profile saved.")
        }
        .addOnFailureListener { e ->
           println("Error saving profile: $e")
        }
```

Get Profile

- https://firebase.google.com/docs/firestore/query-data/get-data
- To simply get a data value from Firestore, we can use the get() method on a particular document

```
db.collection("cities").document("my-city-id").get()
```

- This will return a DocumentSnapshot that we can handle in a success listener
  - For example, convert it to the type of object we are querying for

```
...get().addOnSuccessListener { documentSnapshot ->
    val city = documentSnapshot.toObject(City::class.java)}
```

- This will get the currently stored value, by request
- The function toObject will perform a conversion of the data for us into the object type specified using the ::class operator.

Get Real-Time Changes (SnapshotListener)

- But, for our use case, we'd like to get our profile as a flow so that we can respond to changes in the saved value.
- To do this, we first can add a "Snapshot" listener to the document so that we can get notified of changes in real time
  - In the snapshot listener, we have a lambda with two values – the snapshot and any error

```
val docRef = db.collection("cities").document("my-city-id ")
        docRef.addSnapshotListener { snapshot, error ->
            if (error != null) {
                println("Listen failed: $error")
                return@addSnapshotListener
            }

            if (snapshot != null && snapshot.exists()) {
                println("Current data: ${snapshot.data}")
            } else {
                println("Current data: null")
            }
        }
```

Get Changes as Flow using callbackFlow

- callbackFlow is a flow builder function that lets you convert callback-based APIs into flows
- A callbackFlow internally uses a subscription to a listener callback
  - It also must include a special operation called awaitClose that gets executed when the flow is closed of cancelled. This operation will remove the subscription.

```
override suspend fun getProfile(): Flow<ProfileData>
        = callbackFlow {

    val subscription = dbProfile.document("main-profile").addSnapshotListener{
        ...
     }

awaitClose { subscription.remove() }
}
```

- https://developer.android.com/kotlin/flow
- https://blog.canopas.com/use-firestore-and-firebase-realtime-database-with-kotlin-flow-76a8f260e31a
- https://medium.com/mobile-app-development-publication/keep-your-kotlin-flow-alive-and-listening-with-callbackflow-c95e5dd545a

Update Flow using TrySend

- Finally, for the callbackFlow to update the flow in response to the listener callback, it uses trySend() to "send" the new value
  - trySend() "sends" a value into a Kotlin "channel" (but is not a suspending function)
  - Similar to "emit" into a "flow"
  - We use this here because callbackFlow uses a channel internally
  - Don't worry too much about the details, just use it… ⏪

  - https://medium.com/mobile-app-development-publication/kotlins-flow-channelflow-and-callbackflow-made-easy-5e82ce2e27c0

```
override suspend fun getProfile(): Flow<ProfileData> = callbackFlow {
    val docRef = dbProfile.document("main-profile")
    val subscription = docRef.addSnapshotListener{ snapshot, error ->
        if (error != null) {
            // An error occurred
            println("Listen failed: $error")
            return@addSnapshotListener
        }
        if (snapshot != null && snapshot.exists()) {
            // The user document has data
            val profile = snapshot.toObject(ProfileData::class.java)
            if (profile != null) {
                println("Real-time update to profile")
                trySend(profile)
            } else {
                println("Profile is / has become null")
                trySend(ProfileData()) // If there is no saved profile, then send a default object
            }
        } else {
            // The user document does not exist or has no data
            println("Profile does not exist")
            trySend(ProfileData()) // send default object
        }
    }
    awaitClose { subscription.remove() }
}
```

Delete

- To delete a document, use the delete() function. You can use the success/failure listeners here too.

```
override suspend fun clear() {
    dbProfile.document(profileId)
        .delete()
        .addOnSuccessListener { println("Profile successfully deleted!") }
        .addOnFailureListener { error -> println("Error deleting profile: $error") }
}
```

Storing Lists/Sets

- Branch: firestoreCRUDlist
- Our profile example we've used so far had a single profile representing a single set of preferences to store
  - Mostly since we learned it in the context of a preferences DataStore
- More generally, we'll want to use firebase to store multiple objects and retrieve/update those objects
  - E.g., users, products, profiles, etc.
- Let's create a slightly different repository called UserProfileRepository
- It will store the same ProfileData objects as our current ProfileRepository, but will store multiples….
- Copy & Rename ProfileRespository and ProfileRepositoryFirestore
- In UserProfileRepositoryFirestore, use a new collection name
  - val dbUserProfiles: CollectionReference = db.collection("UserProfiles")
- In AppModule, we'll need to inject the firestore instance from MyApp so that we can use it in both repositories.

```
class MyApp: Application() {

    /* Always be able to access the module ("static") */
    companion object {
        lateinit var appModule: AppModule
    }

    /* Called only once at beginning of application's
lifetime */
    override fun onCreate() {
        super.onCreate()
        appModule = AppModule(this, Firebase.auth,
FirebaseFirestore.getInstance())
    }
}
```

```
class AppModule(
    private val appContext: Context,
    private val auth: FirebaseAuth,
    private val firestore: FirebaseFirestore
) {
    /* Create appropriate repository (backed by Firebase) on first use.
       Only one copy will be created during lifetime of the application.
*/
    val profileRepository : ProfileRepository by lazy {
        ProfileRepositoryFirestore(firestore)
    }
    val userProfileRepository : UserProfileRepository by lazy {
        UserProfileRepositoryFirestore(firestore)
    }
    val authRepository : AuthRepository by lazy {
        AuthRepositoryFirebase(auth) // inject Firebase auth
    }
}
```

UserProfileRepository Interface

- In saveProfile() implementation, use the name of the provided ProfileData parameter as the document id
  - dbProfile.document(profileData.name).set(profileData)
- Change clear() to delete(name: String) and delete the document using the name parameter as the id
  - dbProfile.document(name).delete()
- Change getProfile() to getProfile(name: String)
  - val docRef = dbProfile.document(name)
- Add a getProfiles() function to our interface that returns Flow<List<ProfileData>>

```
interface UserProfileRepository {
    suspend fun saveProfile(oldName: String, profileData: ProfileData)
    suspend fun getProfile(name: String): Flow<ProfileData>
    suspend fun getProfiles(): Flow<List<ProfileData>>
    suspend fun delete(name: String)
}
```

```
class UserProfileRepositoryFirestore (val db: FirebaseFirestore) : UserProfileRepository {
    val dbUserProfiles: CollectionReference = db.collection("UserProfiles")
    override suspend fun saveProfile(oldName: String, profileData: ProfileData) {
        // We are storing only a single profile at a time, so use a unique document name to refer to it
        dbUserProfiles.document(oldName).set(profileData)
            .addOnSuccessListener {
                println("Profile saved.")
            }
            .addOnFailureListener { e ->
                println("Error saving profile: $e")
            }
    }

   override suspend fun delete(name:String) {
        dbUserProfiles.document(name)
            .delete()
            .addOnSuccessListener { println("Profile $name successfully deleted!") }
            .addOnFailureListener { error -> println("Error deleting profile $name: $error") }
    }
```

```
override suspend fun getProfile(name: String): Flow<ProfileData> = callbackFlow {

    val docRef = dbUserProfiles.document(name)
    val subscription = docRef.addSnapshotListener{ snapshot, error ->
        if (error != null) {
            // An error occurred
            println("Listen failed: $error")
            return@addSnapshotListener
        }
        if (snapshot != null && snapshot.exists()) {
            // The user document has data
            val profile = snapshot.toObject(ProfileData::class.java)
            if (profile != null) {
                println("Real-time update to profile")
                trySend(profile)
            } else {
                println("Profile is / has become null")
                trySend(ProfileData()) // If there is no saved profile, then send a default object
            }
        } else {
            // The user document does not exist or has no data
            println("Profile does not exist")
            trySend(ProfileData()) // send default object
        }
    }
    awaitClose { subscription.remove() }
}
```

```
override suspend fun getProfiles(): Flow<List<ProfileData>> = callbackFlow {


    // Listen for changes on entire collection
    val subscription = dbUserProfiles.addSnapshotListener{ snapshot, error ->
        if (error != null) {
            // An error occurred
            println("Listen failed: $error")
            return@addSnapshotListener
        }
        if (snapshot != null) {
            // The collection has documents, so convert them all to ProfileData objects
            val profiles = snapshot.toObjects(ProfileData::class.java)
            if (profiles != null) {
                println("Real-time update to profile")
                trySend(profiles)
            } else {
                println("Profiles has become null")
                trySend(listOf<ProfileData>()) // If there is no saved profile, then send a default object
            }
        } else {
            // The user document does not exist or has no data
            println("Profiles collection does not exist")
            trySend(listOf<ProfileData>()) // send default object
        }
    }
    awaitClose { subscription.remove() }
  }
```

- See branch (firestoreCRUDlist) for more details.

- ViewModel:

  Create variables to track the flow containing the list

  In an init block, collect the flow to get things started.

```
private val _allProfiles = MutableStateFlow(listOf<ProfileData>())
  // public getter for the state (StateFlow)
  val allProfiles: StateFlow<List<ProfileData>> = _allProfiles.asStateFlow()

  init {
      viewModelScope.launch {
          userProfileRepository.getProfiles().collect { allProfiles ->
              _allProfiles.value = allProfiles
          }
      }
  }
```

- In a Screen composable, collect the flow list as state, then use it in a LazyColumn

```
val allProfiles by myViewModel.allProfiles.collectAsState()
```

Simple query: whereEqualTo()

- You can also query the Firestore collections using a variety of queries
  - Self-study…
- https://firebase.google.com/docs/firestore/query-data/queries
- E.g.,

```
db.collection("cities")
    .whereEqualTo("capital", true)
    .get()
```

- You can make certain operations dependent upon the results of a query.
- For example, get() the first document whose name field matches the target. Then change the value of that document using set()

```
dbUserProfiles.whereEqualTo("name", oldname).limit(1).get()
          .addOnSuccessListener { snapshot ->
            for (document in snapshot) {
              // will only be 1 at most due to limit(1)
              val docId = document.id
              dbUserProfiles.document(docId).set(profileData)
                .addOnSuccessListener {
                  println("Profile for $oldname updated.")
                }
                .addOnFailureListener {
                  println("Failed to update profile for $oldname.")
                }
            }
          }
          .addOnFailureListener { e ->
            println("Error saving profile for $oldname: $e")
          }
```