# COMP-304 Winter 2023

# Review of Lecture 8

- **Room persistence library** - an abstraction layer over SQLite API
- **MVVM** architectural pattern:
- **Model** - represents the data and database functionalities
- **ViewModel** - interacts with model and also prepares observable(s) that can be observed by a View
  - Uses a **LiveData** object to hold data
- **View** - observes (or subscribe to) a ViewModel observable to get data in order to update UI elements accordingly

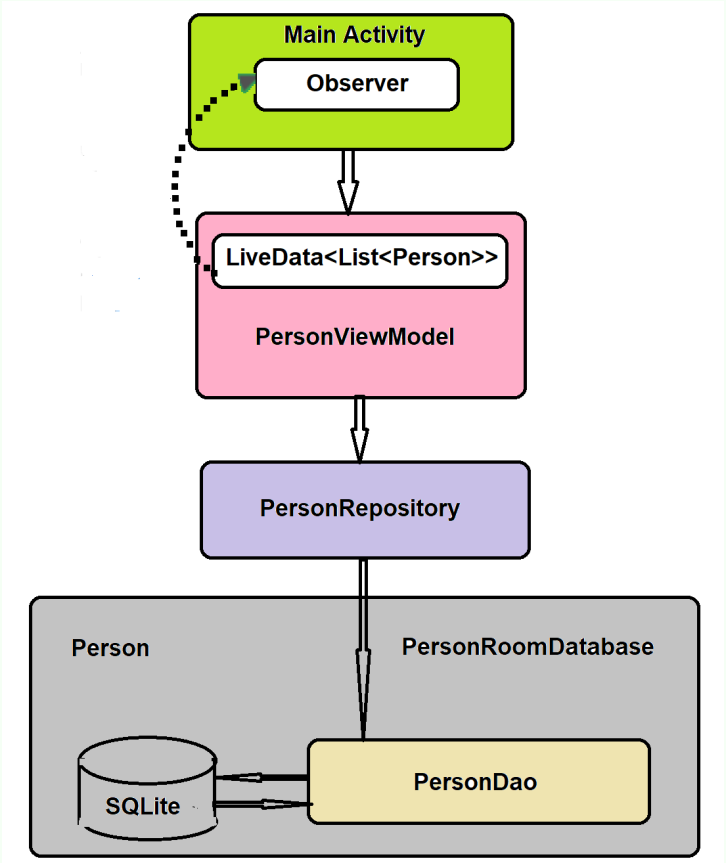
- **Architecture Components**
- **Entity** - is an annotated class that describes a database table.
- **DAO**: Data access object – a mapping of SQL queries to functions.
- **Repository**: A class that you create for managing multiple data sources.
- **ViewModel** - provides data to the UI, stands between the Repository and the UI.
  - Hides where the data originates from the UI.
  - ViewModel instances survive configuration changes.

# Review of Lecture 8

- **LiveData**: A data holder class that can be observed.
  - Always holds/caches latest version of data.
  - Notifies its observers when the data has changed.
  - LiveData is lifecycle aware.
  - UI components just observe relevant data and don't stop or resume observation.
- **Room maps** database **objects** to Java/Kotlin **objects:**
  - maps class variables to table columns
  - maps methods to SQL statements
  - Uses annotations within your class definitions
- A new abstract class that extends **RoomDatabase**, annotating it with a @Database annotation that includes a list of each of your entity classes and the current version number.

# Review of Lecture 8

- **CRUD operations with Room:**
  - **Insert -** @Insert annotation to annotate methods that will be used to insert a new object/entity instance into your database
  - **Update** - use the @Update annotation
  - **Delete** - use the @Delete annotation
  - **Select** – use @Query annotation
- **Using Room in apps**



# Working With **Firebase**/Firestore Databases

**Objectives:**

- Write Android Apps with full CRUD functionalities
- Write Android apps that manipulate a **Firebase**/Firestore database

# Firebase Realtime Database

- Firebase Realtime Database is a cloud-hosted NoSQL database:
  - **data is synced** across all clients in real time
  - remains available for queries and transactions on your device even when you lose Internet connectivity.
- It's a **NoSQL database**, it is not relational and you do not use SQL statements to interact with it.
- The **database is stored locally** on each device **as JSON files**, which are synchronized in real time to the cloud-host and in turn with every connected client.

# Checking dependencies

- In **build.gradle** (project level), add at the top:

```
buildscript {
    // ...

    dependencies {
        classpath 'com.android.tools.build:gradle:7.3.0'

        // The google-services plugin is required to parse the google-services.json file
        classpath 'com.google.gms:google-services:4.3.14'
    }
}
```

# Checking dependencies

- In **build.gradle** (module level), add:

```
plugins {
    id 'com.android.application'
    id 'org.jetbrains.kotlin.android'
    id 'com.google.gms.google-services'
}
```

# Checking dependencies

**dependencies** {

……..

implementation 'com.google.firebase:**firebase-database**:20.0.6’

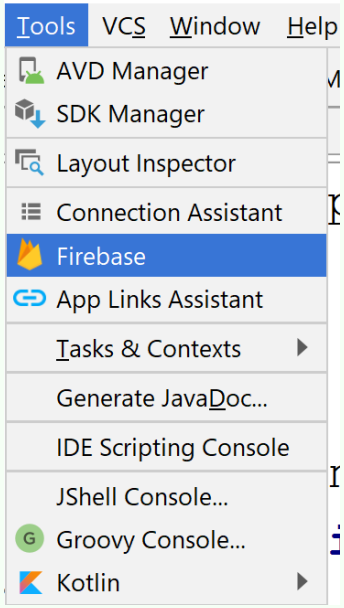
}

# Adding Firebase to Your App

- **Option 1**: Add Firebase using the Firebase console (https://firebase.google.com/docs/android/setup)
  - Create a Firebase project
  - Register your app with Firebase
  - Add a Firebase configuration file
  - Modify root-level (project-level) Gradle file (build.gradle).
  - Modify module (app-level) Gradle file (usually app/build.gradle)
  - Add Firebase SDKs to your app

# Adding Firebase to Your App

- **Option 2**: Android Studio includes a **Firebase Assistant** to simplify adding Firebase components to your app.
  - To use it, select **Tools** ⇨ **Firebase** to display the assistant window.
  - Expand the Realtime Database list item and select the hyperlinked **Save and retrieve data text**, to display the Firebase Realtime Database assistant.



# Adding Firebase to Your App

- Run assistant:

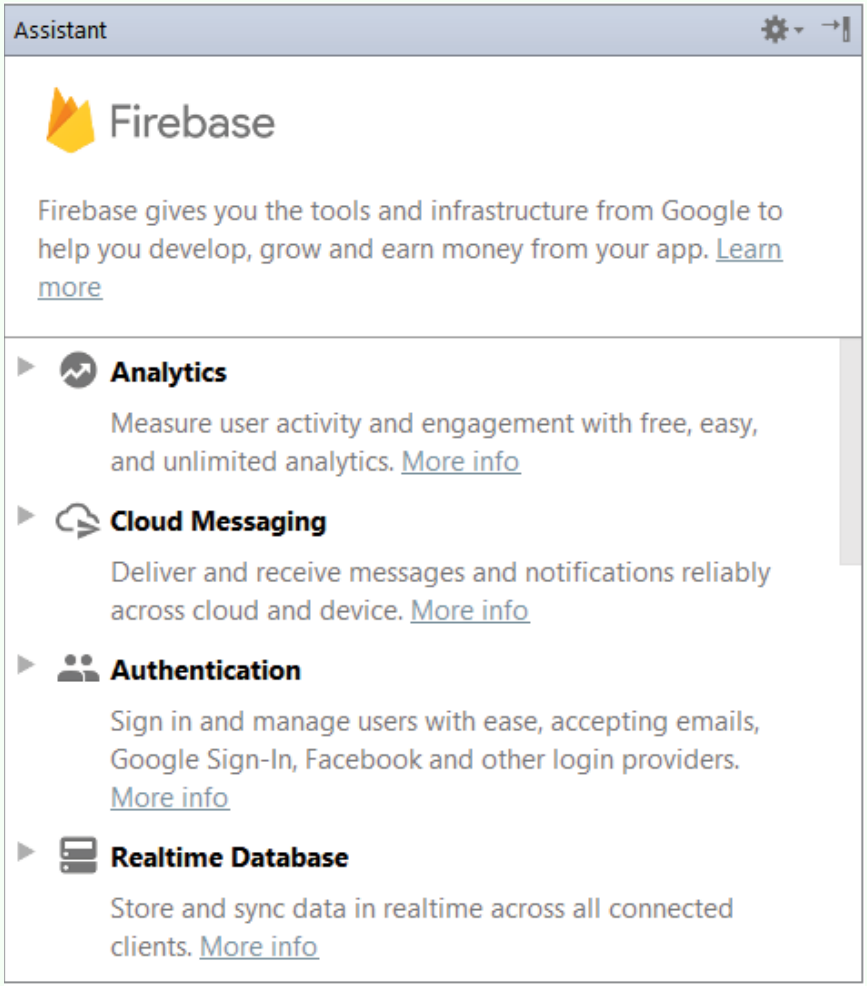



FIGURE 9-3

# Adding Firebase to Your App

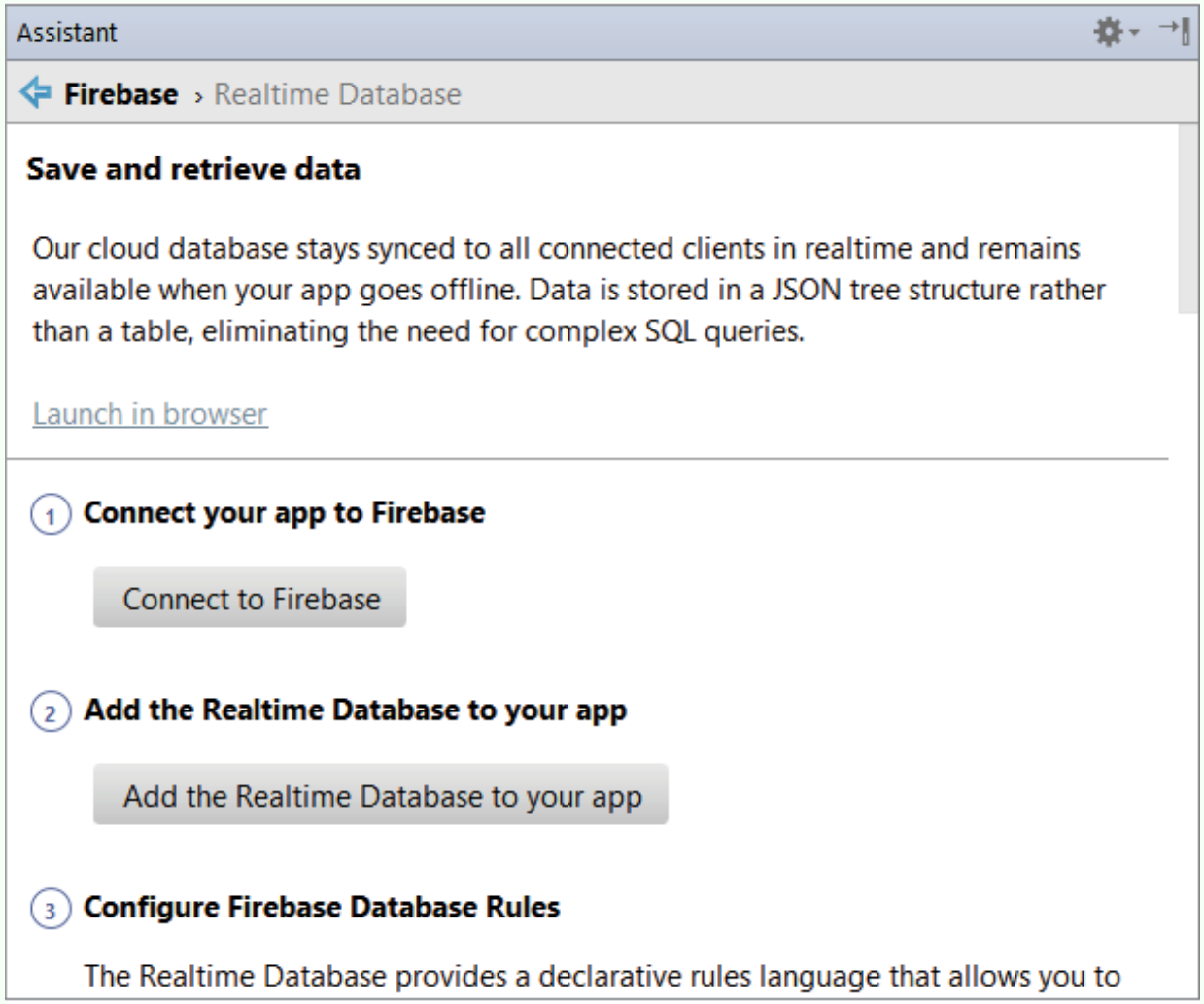
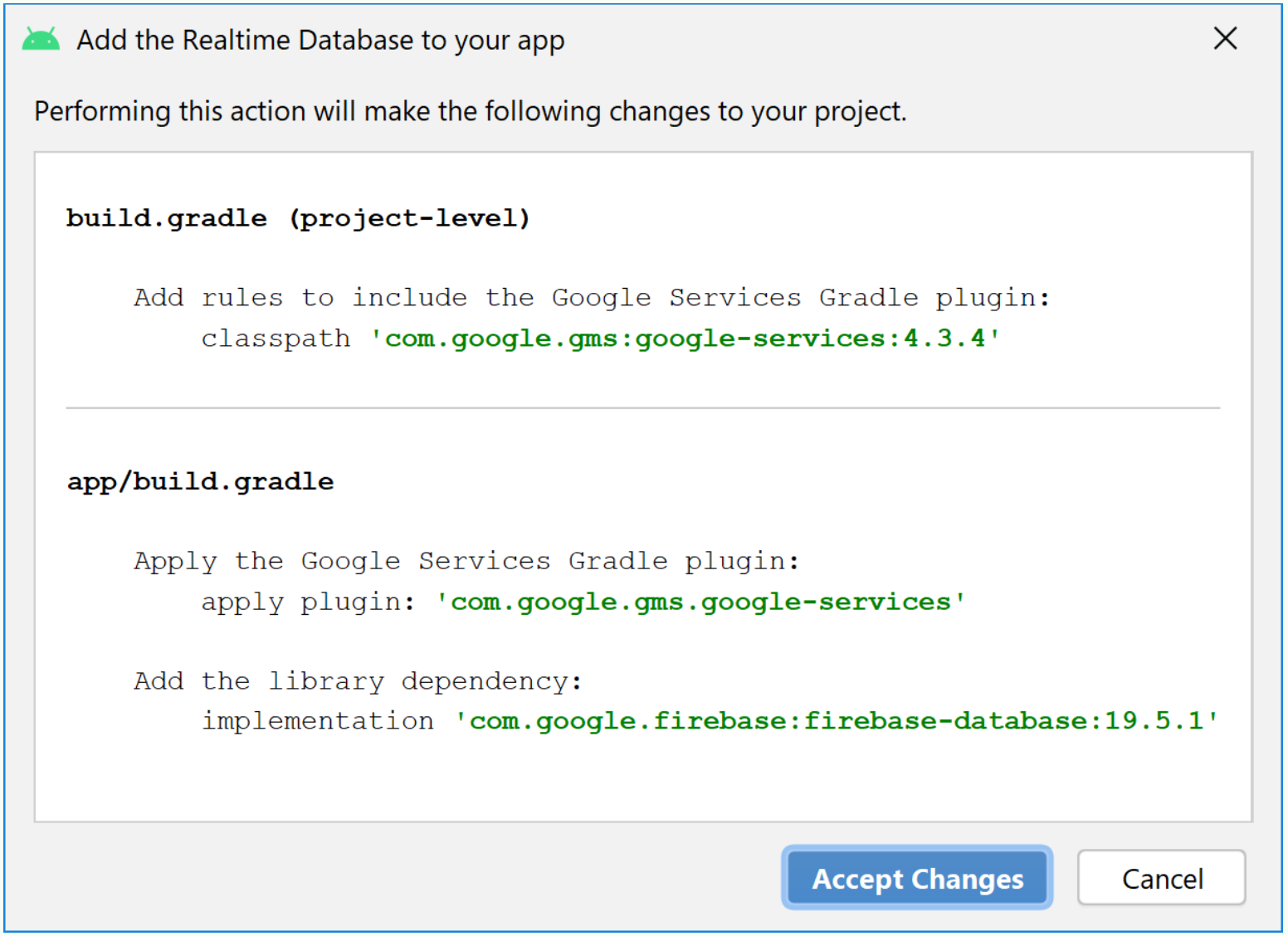



FIGURE 9-4

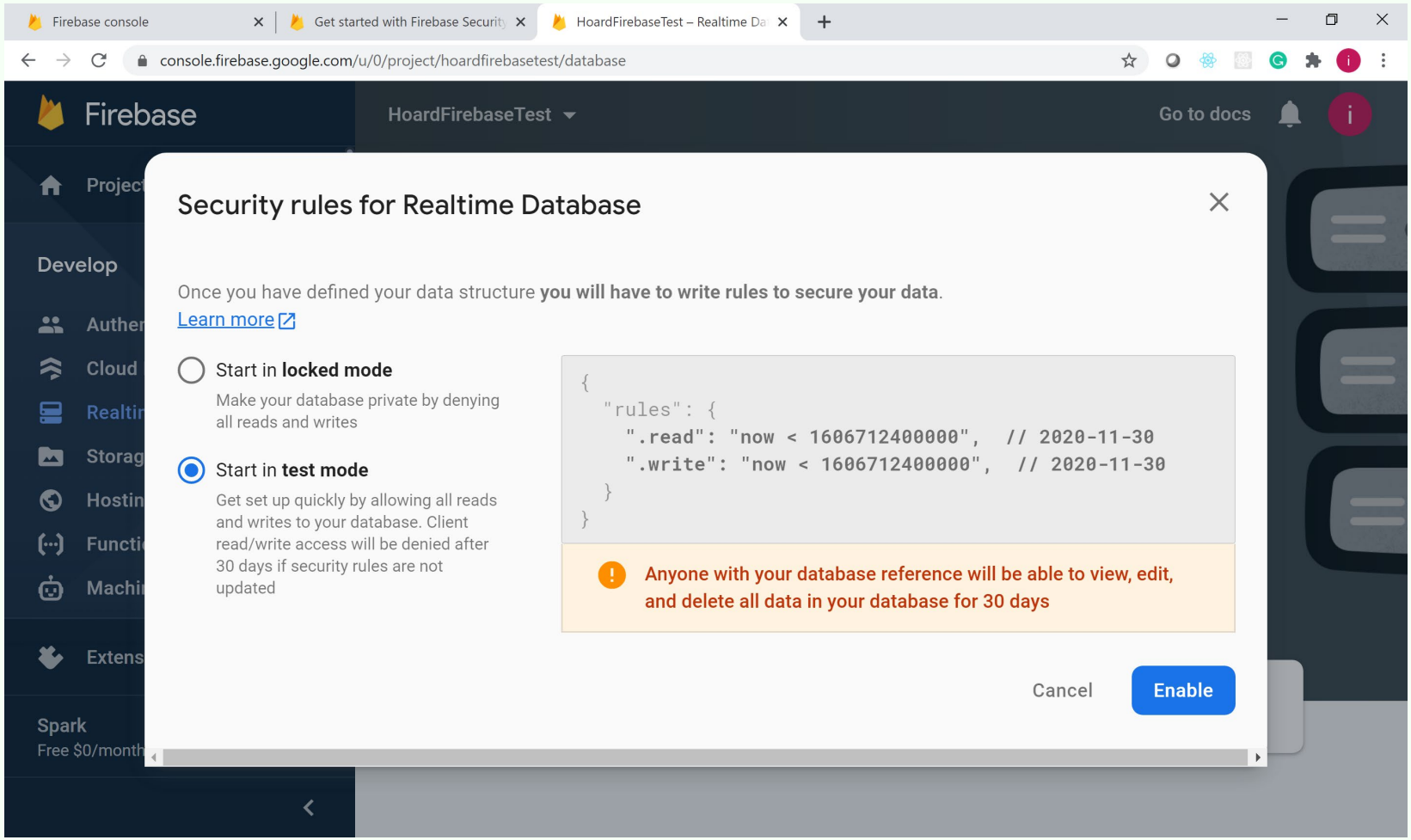
# Adding Firebase to Your App

- With your app connected, you can choose to “Add the Realtime Database to Your App”
- This will add:
  - the Firebase Gradle build script dependency to your project-level build.gradle file
  - the Firebase plug-in for Gradle
  - a dependency for the Firebase Database library to your build.gradle file.

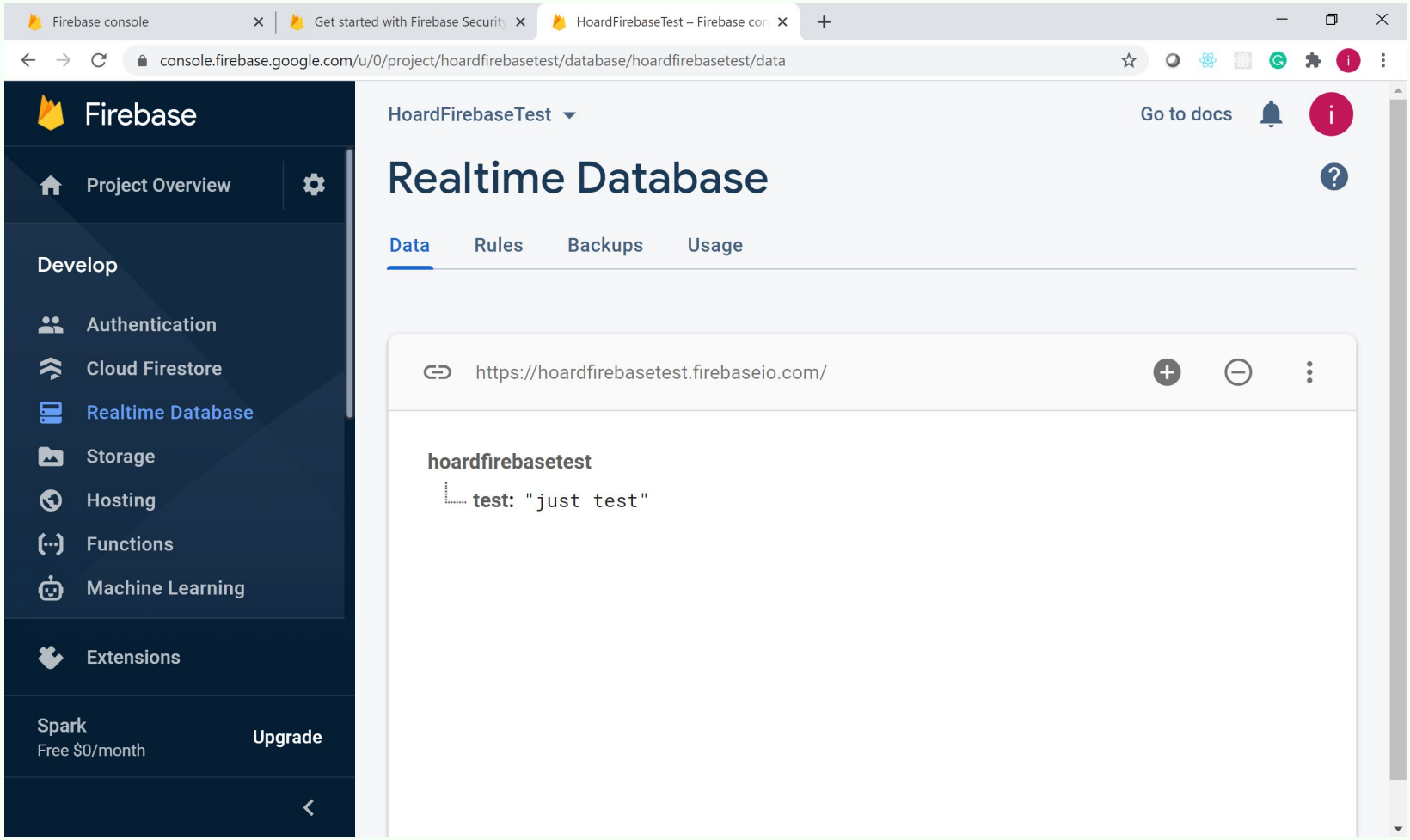
# Adding Firebase to Your App



# Create Database



# Adding Firebase to Your App



# Adding Firebase to Your App

- The Firebase Realtime Database uses **a declarative rules language** to define how your data should be accessed.
- By default, Firebase Databases require Firebase Authentication, and grant full read and write permissions to all authenticated users.
- During development it can be useful to permit full unauthenticated access to get started, allowing you to develop your database before you have completed the authentication piece.
- To set the access rules to public, switch to the *rules* tab and set the read and write elements to true: {

```
"rules": {
".read": true,
".write": true
}
}
```

# Adding data to a Firebase Realtime Database

```
class Hoard (hoardName: String?, goldHoarded: Int,
hoardAccessible: Boolean ) {
        val hoardName = hoardName
        val goldHoarded = goldHoarded
        val hoardAccessible = hoardAccessible

        override fun toString(): String {
            return "($hoardName:$goldHoarded:$hoardAccessible)"
        }

}
```

# Adding data to a Firebase Realtime Database

```
class MainActivity : AppCompatActivity() {
    private val TAG = "MainActivity"
    //
    private var database: FirebaseDatabase? = null
    private var listRootRef: DatabaseReference? = null
    var hoard: Hoard? = null
    private var editTextGold: EditText? = null
    private var editTextHoardName: EditText? = null
    private var txtMessage: TextView? = null

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        //
        editTextGold = findViewById<View>(R.id.editTextGold) as EditText
        editTextHoardName = findViewById<View>(R.id.editTextTextHoardName) as EditText
        txtMessage = findViewById<View>(R.id.txtMessage) as TextView
        //
        database = FirebaseDatabase.getInstance()
        listRootRef = database!!.getReference("hoards")
    }
```

# Adding data to a Firebase Realtime Database

```
fun sendAmount(view: View?) {
        //
        val hoardName = editTextHoardName!!.text.toString()
        val goldHoarded = editTextGold!!.text.toString().toInt()
        hoard = Hoard(
            hoardName,
            goldHoarded,
            true
        )
        val itemRootRef = listRootRef!!.child(hoard!!.hoardName.toString())
        // Set values for the properties of our hoard.
        itemRootRef.child("hoardName").setValue(hoard!!.hoardName)
        itemRootRef.child("goldHoarded").setValue(hoard!!.goldHoarded)

      itemRootRef.child("hoardAccessible").setValue(hoard!!.hoardAccessib
      le)
```

# Querying data

```
        // Read from the database
        listRootRef!!.addValueEventListener(object : ValueEventListener {
            override fun onDataChange(dataSnapshot: DataSnapshot) {
                // This method is called once with the initial value and again
                // whenever data at this location is updated.
                val key = dataSnapshot.key
                val value = dataSnapshot.value.toString()
                txtMessage!!.text = "Value is: $value"
            }
            override fun onCancelled(error: DatabaseError) {
                // Failed to read value
                Log.w(TAG, "Failed to read value.", error.toException())
            }
        })
        //
    }
}
```

# Modifying and Deleting data

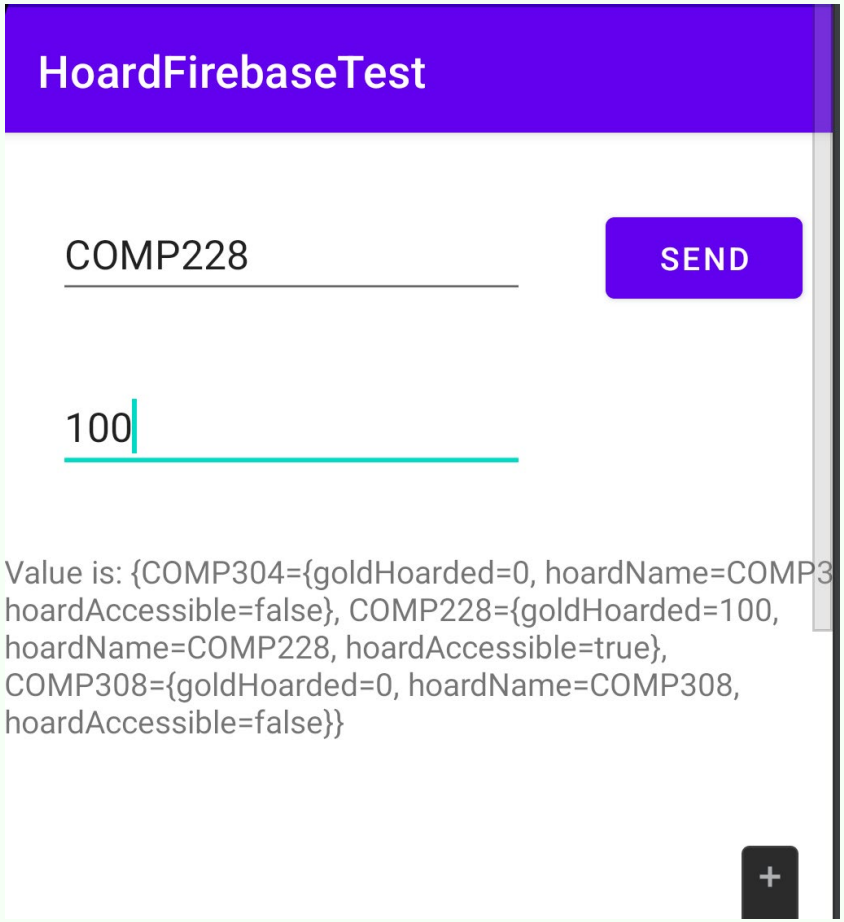
- To **modify an entry**, simply use setValue as you would to create a new entry, and the previous value(s) will be overridden with the new values.

```
val itemRootRef =
listRootRef!!.child(hoard!!.hoardName.toString())
// Set values for the properties of our hoard.
itemRootRef.child("hoardName").setValue(hoard!!.hoardName)
```

- To **delete an entry**, call removeValue on the node or element you wish to remove:

```
database = FirebaseDatabase.getInstance()
listRootRef = database!!.getReference("hoards")
listRootRef.child(hoard!!.hoardName).removeValue()
```

# Firebase Example




# Firestore

- Google has also released a new databse system, **Cloud Firestore**.
- Firestore is **a highly-scalable NoSQL cloud database** that, like Firebase Realtime Database, can be used to sync application data across servers and client apps in real time.
- Firestore is designed specifically to be highly scalable and supports more expressive and efficient querying, including shallow queries that don’t require retrieving the entire collection, and support for sorting, filtering, and limiting query returns.
- It also offers seamless integration with other Firebase and Google Cloud Platform products, **including Cloud Functions**.
- In addition to Android, web, and iOS SDKs, Firestore APIs are available in Node.js, Java, Python, and Go.

# References

- Textbook
- https://firebase.google.com/docs/android/setup
- https://firebase.google.com/docs/emulator-suite
- https://codelabs.developers.google.com/codelabs/firebase-android#0