Firebase

ANDROID

What is Firebase?

- **Firebase** is a **mobile** and **web app** development platform
- Provides developers with a plethora of **tools** and **services** to
  - help them develop high-quality apps,
  - grow their user base
  - earn more profit.

Develop & test your app

- Realtime Database
- Cloud Firestore
- Authentication
- Cloud Storage
- Test Lab
- Performance Monitoring
- Crashlytics
- Cloud Functions
- Hosting

Grow & engage your audience

- Analytics
- Dynamic Links
- Invites
- AdWords
- Cloud Messaging
- Remote Config
- Predictions
- App Indexing
- AdMob

What is Firebase?

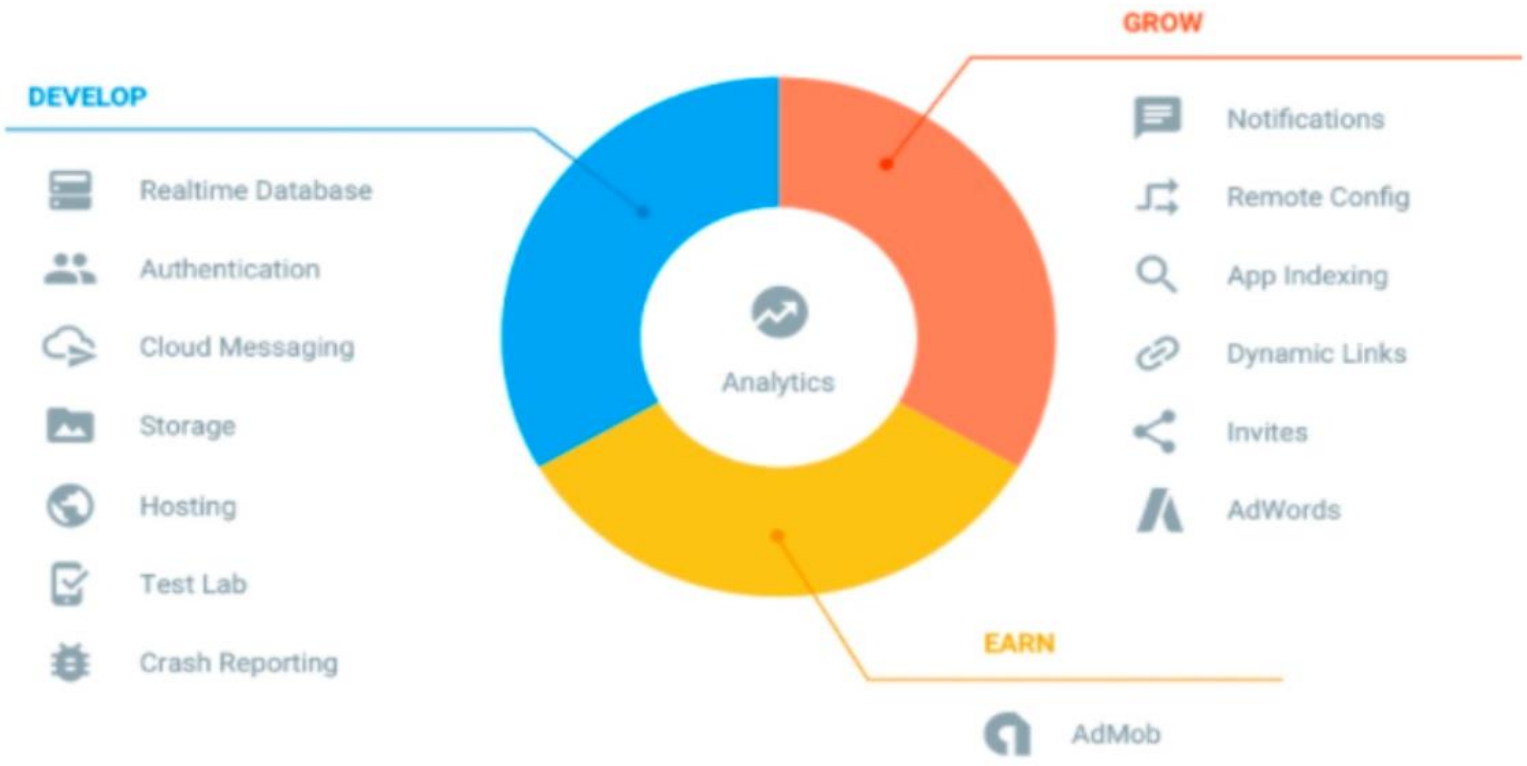
- **Firebase** is a service that takes care of all the back-end of applications.
- The **processing** of the **data**, the **statistics** relative to the use, the management of the **permissions** of the application on each smartphone, etc.

A Brief History

- 2011, it was a **startup** called **Envolve**.
- As Envolve, it provided developers with an **API**.
- Developers were using Envolve to **sync application data** such as a game state in real time across their users.
- In April 2012, Firebase was created as a separate company that provided **Backend-as-a-Service (BaaS)** with real-time functionality.
- After it was **acquired by Google in 2014**, Firebase rapidly evolved into the multifunctional system of a **mobile** and **web** platform that it is today.

Firebase Services

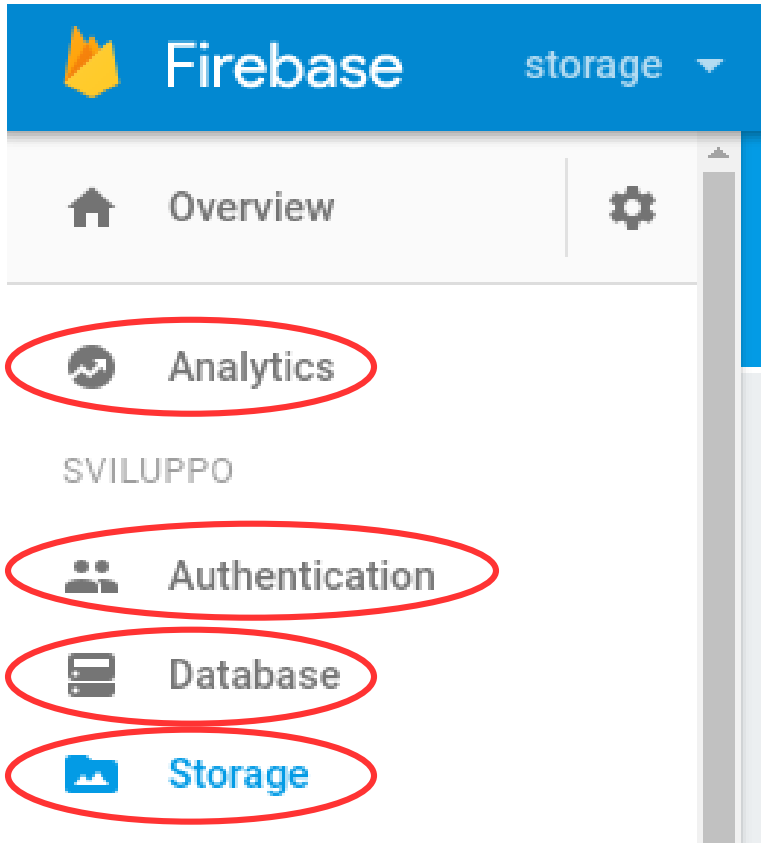
- As Firebase Services can be divided into two groups:
- Develop & test your app
- Grow & Engage your audience



Database

Android Firebase Samples

- A collection of quickstart samples demonstrating the Firebase APIs on Android: https://github.com/firebase/quickstart-android/
- Some interesting samples:
  - Analytics
  - Auth
  - Storage
  - Database



Real time database

- The **Firebase Realtime Database** is a
  - cloud-hosted **NoSQL** database
  - lets you **store** and **sync** between your users in realtime.

Table-to-JSON

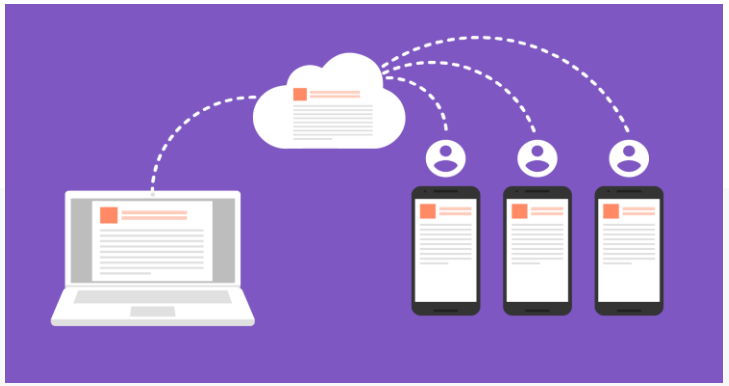
First Name	Last Name	Points
Jill	Smith	50
Eve	Jackson	94
John	Doe	80

JSON

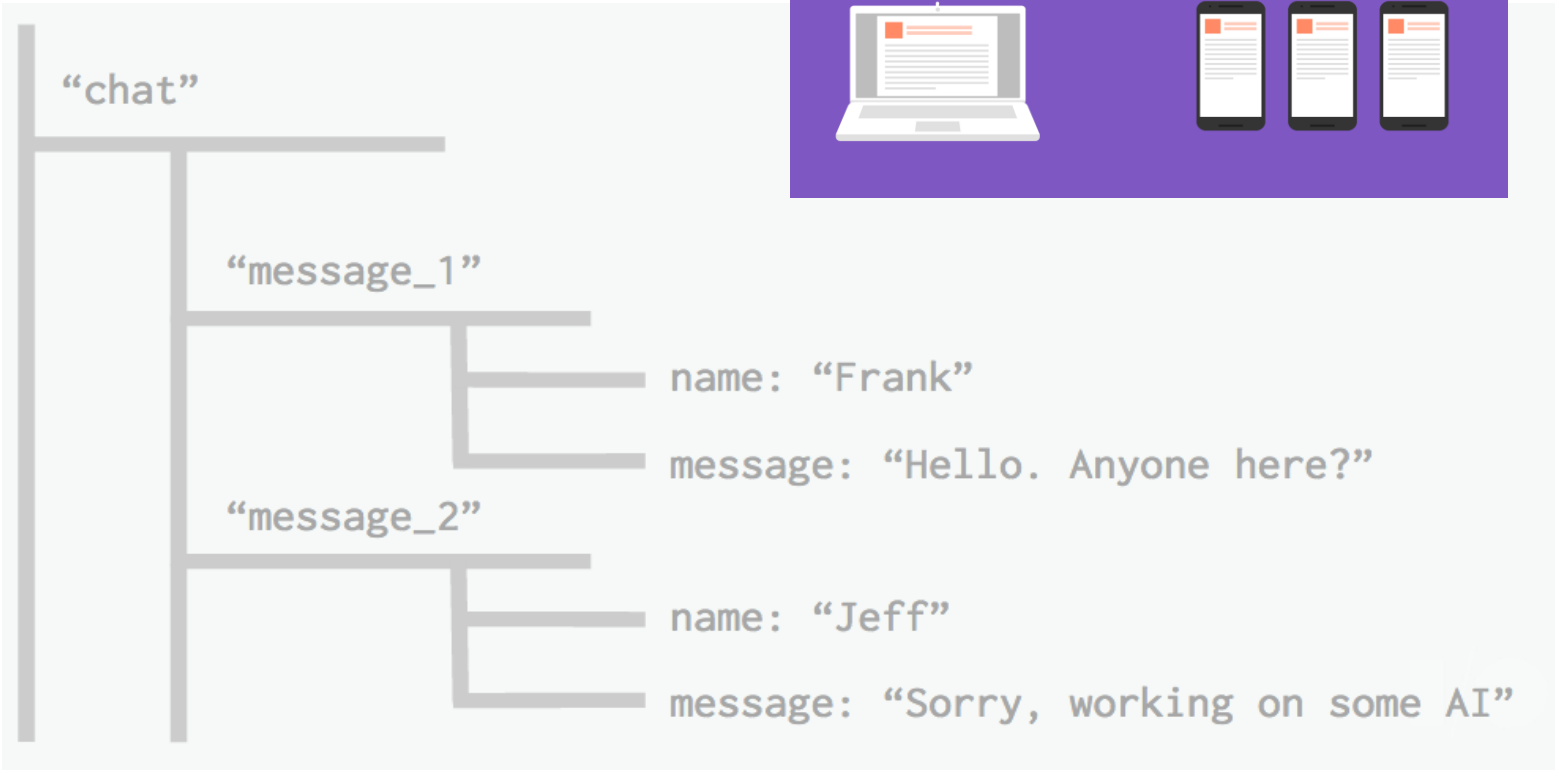
rst Name":"Jill","Last Name":"Smith","Score":"50"},
rst Name":"Eve","Last Name":"Jackson","Score":"94"},
rst Name":"John","Last Name":"Doe","Score":"80"}]

Real time database

- The Realtime Database is really just **one big JSON object** that the developers can manage in realtime.



- With just a **single API**, the Firebase database provides your app with
- the **current value** of the data
- **any updates** to that data.



Real time database

- The Realtime Database uses WebSocket technology under the hood **for synchronization**
- All the connected clients can receive realtime update when the data in the server changes and vice versa,
- Updates within milliseconds latency
- Bidirectional communication

Offline support

- Automatically syncs to the server and pushes to other devices when network comes back
- When your **users go offline**, the Realtime Database SDKs use local cache on the device to serve and store changes.
- When the device comes online, the local data is automatically synchronized.

How to Connect Android Studio Project With Firebase

- Go to https://console.firebase.google.com and create a new project
- Connect the project with the Android Project using Firebase Assistant
- Create the database using the console.firebase... web page

```
// Write a message to the database
val database = FirebaseDatabase.getInstance("https://<YOUR_DB URL>")
val myRef = database.getReference("message")

myRef.setValue("Hello, World!")
```

```
https://fir-kotlin2022-default-rtdb.firebaseio.com/
    message: "Hello, World!"
```

Read data from database

```
myRef.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 value = dataSnapshot.getValue(String::class.java)
    Log.d(TAG, "Value is: $value")
    tvMessage.text = value
  }

  override fun onCancelled(error: DatabaseError) {
    // Failed to read value
    Log.w(TAG, "Failed to read value.", error.toException())
  }
})
```

Firebase Database: Read Data

There are three ways by which you can **listen** to your data:

- **`ChildEventListener`**: Child events are triggered in response to specific operations that happen to the children of a node
- **`ValueEventListener`**: will return the entire list of data as a single DataSnapshot, which you can then loop over to access individual children.
- **`ListenerForSingleValueEvent`**: Same as ValueEventListener but is triggered only once and then doesn't trigger again.

Read Data example: ChildEventListener

- Get all phone numbers of users
- First retrieve the users DataSnapshot

```
//Get datasnapshot at "users" root node
DatabaseReference ref = FirebaseDatabase.getInstance().getReference().child("users");
ref.addChildEventListener(new ChildEventListener() {
    @Override
    public void onChildAdded(DataSnapshot dataSnapshot, String s) {
        collectPhoneNumbers((Map<String,Object>) dataSnapshot.getValue());
    }
    @Override
    public void onChildChanged(DataSnapshot dataSnapshot, String s) { }
    @Override
    public void onChildRemoved(DataSnapshot dataSnapshot) { }
    @Override
    public void onChildMoved(DataSnapshot dataSnapshot, String s)
    @Override
    public void onCancelled(DatabaseError databaseError) { }
});
```

- Then loop through users, accessing their map and collecting the phone field.

```
private void collectPhoneNumbers(Map<String,Object> users) {

    ArrayList<Long> phoneNumbers = new ArrayList<>();

    //iterate through each user, ignoring their UID
    for (Map.Entry<String, Object> entry : users.entrySet()){
        //Get user map
        Map singleUser = (Map) entry.getValue();
        //Get phone field and append to list
        phoneNumbers.add((Long) singleUser.get("phone"));
    }
}
```

Read Data example: ValueEventListener

- Get all phone numbers of users
- First retrieve the users DataSnapshot

```
//Get datasnapshot at "users" root node
DatabaseReference ref = FirebaseDatabase.getInstance().getReference().child("users");
ref.ValueEventListener(    new ValueEventListener() {
    @Override
    public void onDataChange(DataSnapshot dataSnapshot) {
        //Get map of users in datasnapshot
        collectPhoneNumbers((Map<String,Object>) dataSnapshot.getValue());
    }

    @Override
    public void onCancelled(DatabaseError databaseError) {
        //handle databaseError
    }
});
```

- Then loop through users, accessing their map and collecting the phone field.

```
private void collectPhoneNumbers(Map<String,Object> users) {

    ArrayList<Long> phoneNumbers = new ArrayList<>();

    //iterate through each user, ignoring their UID
    for (Map.Entry<String, Object> entry : users.entrySet()){
        //Get user map
        Map singleUser = (Map) entry.getValue();
        //Get phone field and append to list
        phoneNumbers.add((Long) singleUser.get("phone"));
    }
}
```

Read Data example: ListenerForSingleValueEvent

- Get all phone numbers of users
- First retrieve the users DataSnapshot

```
//Get datasnapshot at "users" root node
DatabaseReference ref = FirebaseDatabase.getInstance().getReference().child("users");
ref.addListenerForSingleValueEvent(    new ValueEventListener() {
    @Override
    public void onDataChange(DataSnapshot dataSnapshot) {
        //Get map of users in datasnapshot
        collectPhoneNumbers((Map<String,Object>) dataSnapshot.getValue());
    }

    @Override
    public void onCancelled(DatabaseError databaseError) {
        //handle databaseError
    }
});
```

- Then loop through users, accessing their map and collecting the phone field.

```
private void collectPhoneNumbers(Map<String,Object> users) {

    ArrayList<Long> phoneNumbers = new ArrayList<>();

    //iterate through each user, ignoring their UID
    for (Map.Entry<String, Object> entry : users.entrySet()){
        //Get user map
        Map singleUser = (Map) entry.getValue();
        //Get phone field and append to list
        phoneNumbers.add((Long) singleUser.get("phone"));
    }
}
```

Firebase Database: Write Data

There are two ways by which you can write data to your db:

- Updating value: You can use **`setValue(..)`** to save data to a specified reference, replacing any existing data at that path or pass a custom Java object.

```
queryDatabase.child("<node_name>").child(<node_id>).child("<field_name>").setValue(<field_value>);
// or
queryDatabase.child("<node_name>").child(<node_id>).setValue(<new_object>);
```

- Update specific children simultaneously: To simultaneously write to specific children of a node without overwriting other child nodes, use the **`updateChildren(..)`** method.

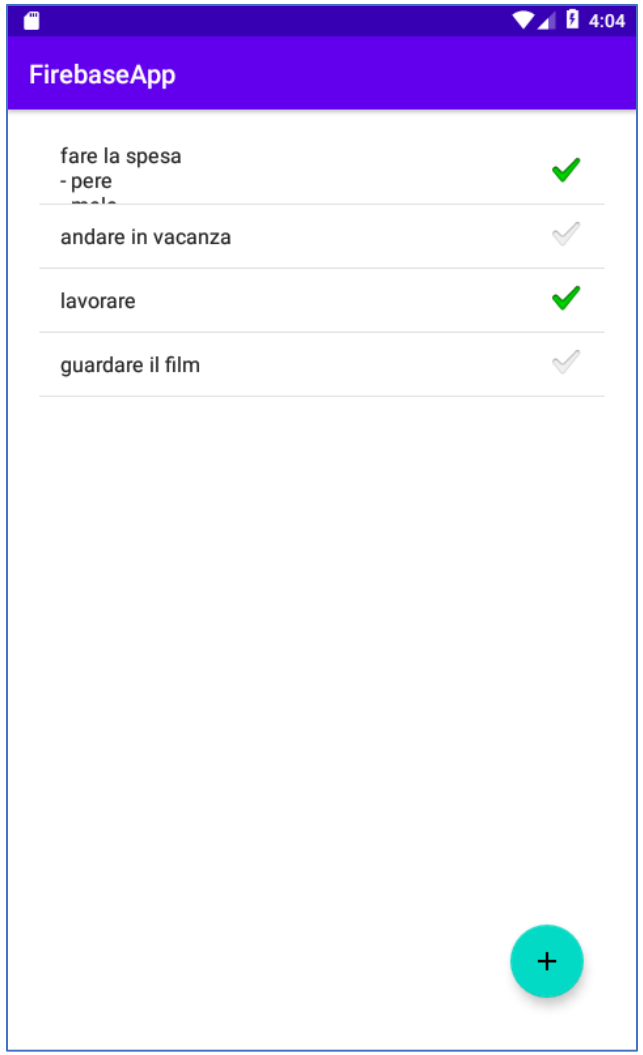
Firebase Database: Write Data

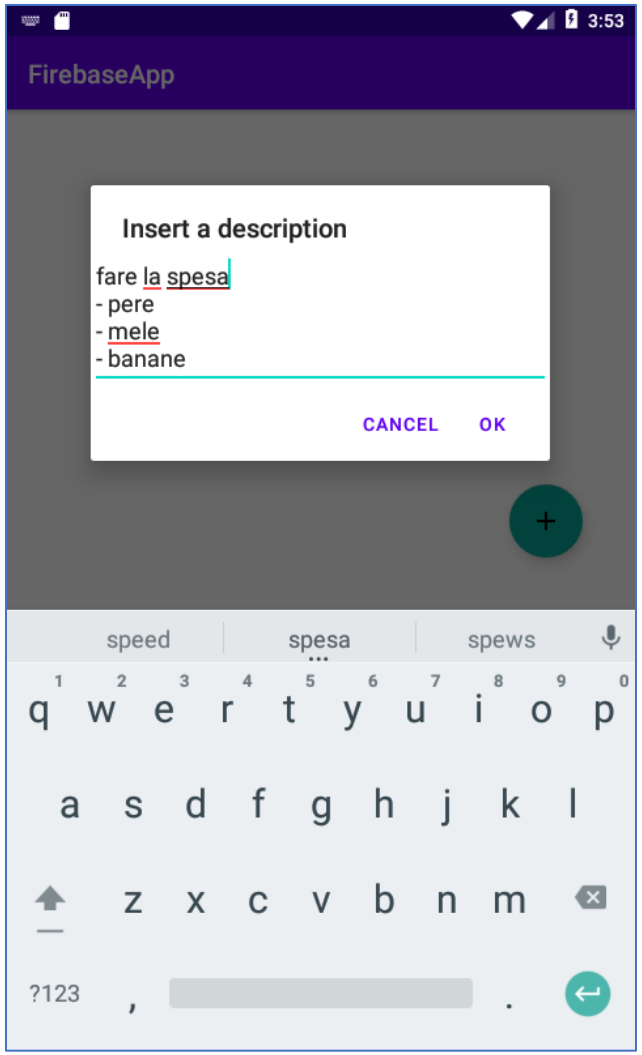
- There are two ways by which you can write data to your db:
- Updating value: ...
- Update specific children simultaneously: To simultaneously write to specific children of a node without overwriting other child nodes, use the **`updateChildren(..)`** method.

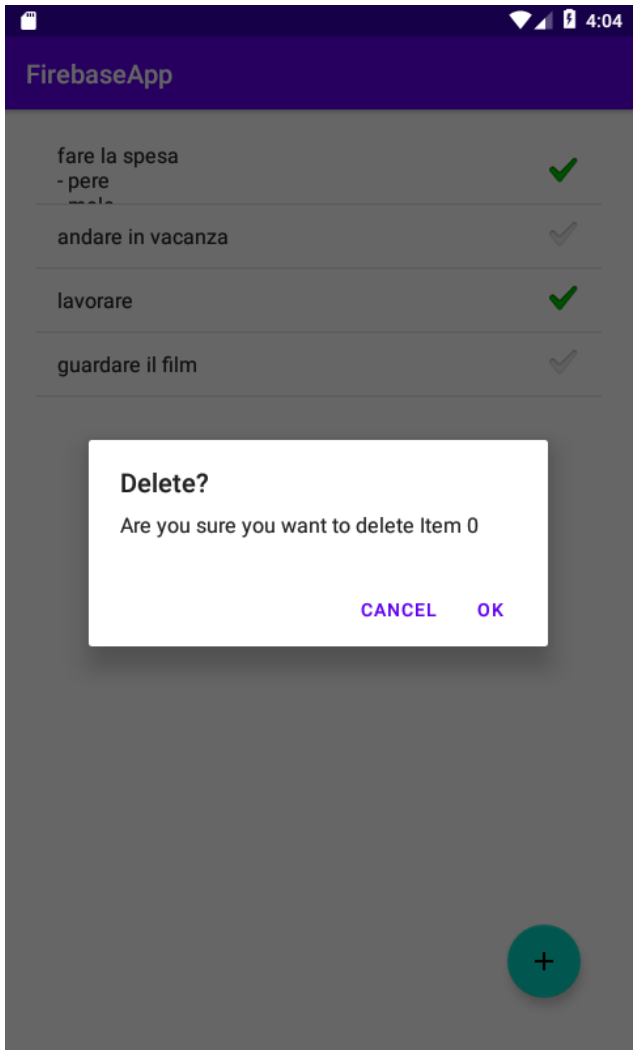
```
private fun onStarClicked(uid: String, key: String) {
    val updates: MutableMap<String, Any> = HashMap()
    updates["posts/$key/stars/$uid"] = true
    updates["posts/$key/starCount"] = ServerValue.increment(1)
    updates["user-posts/$uid/$key/stars/$uid"] = true
    updates["user-posts/$uid/$key/starCount"] = ServerValue.increment(1)
    database.updateChildren(updates)
}
```

https://firebase.google.com/docs/database/android/read-and-write

Example

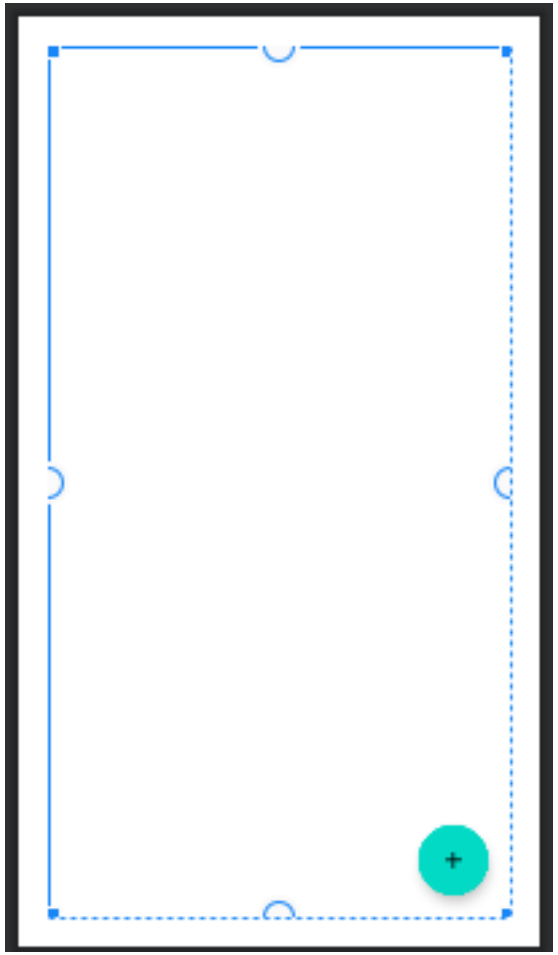

4:04
FirebaseApp
fare la spesa
- pere
andare in vacanza
lavorare
guardare il film


3:53
FirebaseApp
Insert a description
fare la spesa
- pere
- mele
- banane
CANCEL
OK
speed
spesa
spews


4:04
FirebaseApp
fare la spesa
- pere
andare in vacanza
lavorare
guardare il film
Delete?
Are you sure you want to delete Item 0
CANCEL
OK


todo_items
02-5-2022_04:01:18
checked: true
descr: "fare la spesa - pere - mele - banane"
key: "02-5-2022_04:01:18"
02-5-2022_04:02:51
checked: false
descr: "andare in vacanza"
key: "02-5-2022_04:02:51"
02-5-2022_04:03:22
checked: true
descr: "lavorare"
key: "02-5-2022_04:03:22"
02-5-2022_04:04:09
checked: false
descr: "guardare il film "

Layout



```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://sch
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity"
    android:layout_margin="24dp">

    <ListView
        android:id="@+id/lv_items"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:choiceMode="multipleChoice"/>

    <com.google.android.material.floatingactionbutton.FloatingActionButton
        android:id="@+id/fab"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="end|bottom"
        android:layout_margin="16dp"
        android:contentDescription="submit"
        android:src="@drawable/ic_add_24"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

DBHelper & ToDoItem class to store data

```
class FirebaseRealtimeDBHelper {
    companion object {
        // A Firebase reference represents a particular location in your Database and
        // can be used for reading or writing data to that Database location.
        private var firebaseDbRef = FirebaseDatabase
            .getInstance( url: "https://fir-kotlin2022-default-rtdb.firebaseio.com/")
            .getReference( path: "todo_items")

        // read data from DB
        fun readTodoItems(todoEventListener: ChildEventListener) {
            firebaseDbRef.addChildEventListener(todoEventListener)
        }

        // Update or insert a ToDoItem node in Firebase
        fun setTodoItem(key: String, toDoItem: ToDoItem) {
            firebaseDbRef.child(key).setValue(toDoItem)
        }

        // Delete a ToDoItem node in Firebase
        fun removeTodoItem(key: String) {
            firebaseDbRef.child(key).removeValue()
        }
    }
}
```

```
data class ToDoItem(
    var descr: String?,
    var checked: Boolean?,
    var key: String) {

    constructor() : this( descr: "", checked: false,  key: "")

    override fun toString(): String {
        return descr.toString()
    }
    fun set(todo: ToDoItem?) {
        descr = todo?.descr
        checked = todo?.checked
        key = todo?.key.toString()
    }
}
```

MainActivity

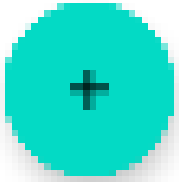
```kotlin
class MainActivity : AppCompatActivity() {
    private val TAG = "MainActivity"
    private val data: MutableList<ToDoItem> = ArrayList()
    private lateinit var adapter: ArrayAdapter<ToDoItem>
    private lateinit var listViewItems: ListView

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        adapter = ArrayAdapter( context: this,
            android.R.layout.simple_list_item_checked, data)
        listViewItems = findViewById<ListView>(R.id.lv_items)
        listViewItems.adapter = adapter

        FirebaseRealtimeDBHelper.readTodoItems(getTodoEventListener())
        val fab = findViewById<FloatingActionButton>(R.id.fab)
        fab.setOnClickListener(getFabClickListner())

        listViewItems.setOnItemLongClickListener {...}
        listViewItems.setOnItemClickListener{...}
    }
```

```kotlin
listViewItems.setOnItemLongClickListener { parent, view, pos, id ->
    val adb = AlertDialog.Builder( context: this@MainActivity)
    adb.setTitle("Delete?")
    adb.setMessage("Are you sure you want to delete Item $pos")
    adb.setNegativeButton( text: "Cancel",  listener: null)
    adb.setPositiveButton( text: "Ok") { dialog, which ->
        FirebaseRealtimeDBHelper.removeTodoItem(data[pos].key)
    }
    adb.show()
    true  ^setOnItemLongClickListener
}
listViewItems.setOnItemClickListener{ parent, view, pos, id ->
    val checked = (view as CheckedTextView).isChecked
    data[pos].checked = checked
    FirebaseRealtimeDBHelper.setTodoItem(data[pos].key, data[pos])
}
```

```kotlin
private fun getFabClickListner(): View.OnClickListener? {
    val lister = View.OnClickListener {...}
    return lister
}

private fun getTodoEventListener(): ChildEventListener {
    // used to receive events about changes in the child
    // locations of a given DatabaseReference
    val childEventLister = object: ChildEventListener{...}
    return childEventLister
}
```

FloatingActionButton

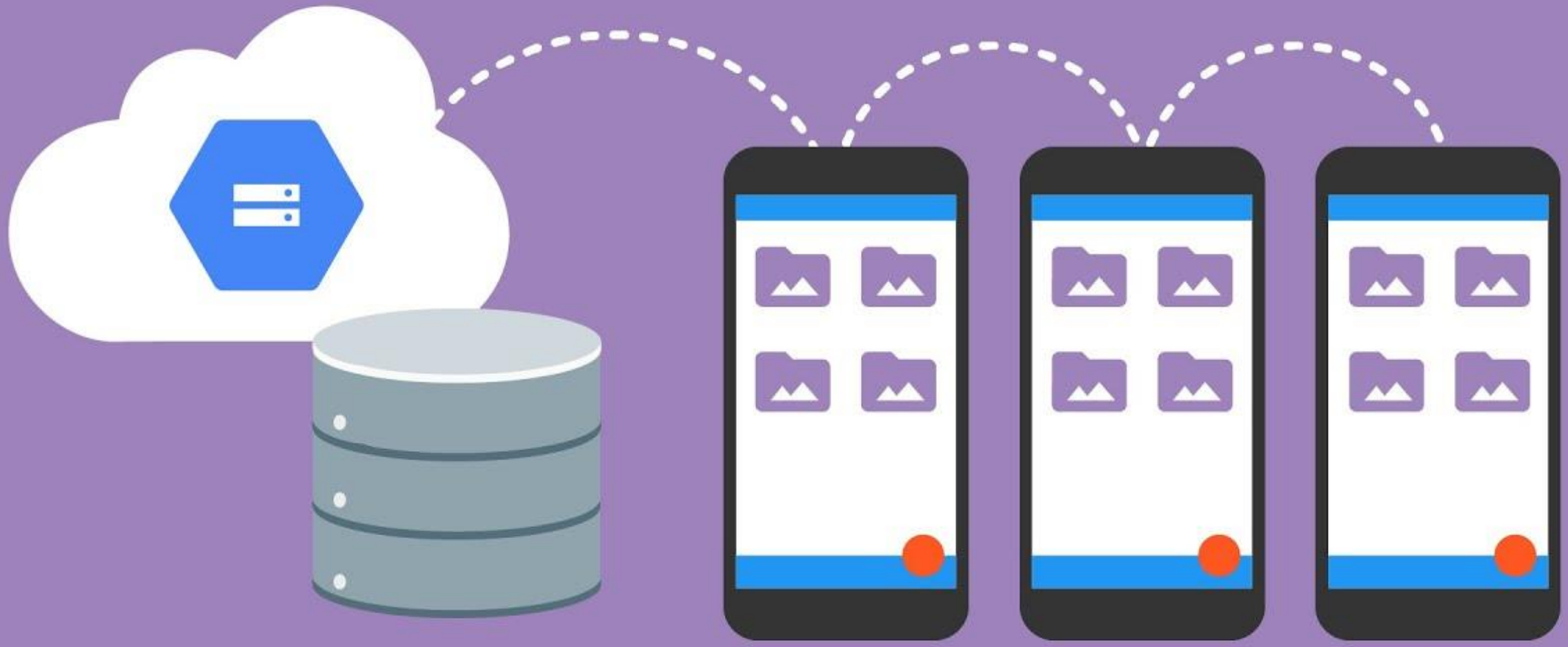
```
private fun getFabClickListner(): View.OnClickListener? {
    val lister = View.OnClickListener {  it: View!
        val adb = AlertDialog.Builder( context: this@MainActivity)
        adb.setTitle("Insert a description")

        val input = EditText( context: this@MainActivity)
        adb.setView(input)

        adb.setPositiveButton( text: "OK")
        { dialog, which ->
            val text = input.text.toString()
            val sdf = SimpleDateFormat( pattern: "dd-M-yyyy_hh:mm:ss")
            val newTodo = ToDoItem(text,  checked: false, sdf.format(Date()))
            FirebaseRealtimeDBHelper.setTodoItem(newTodo.key, newTodo)
        }
        adb.setNegativeButton( text: "Cancel")
        { dialog, which ->
            dialog.cancel()
        }
        adb.show()
    }
    return lister
}
```



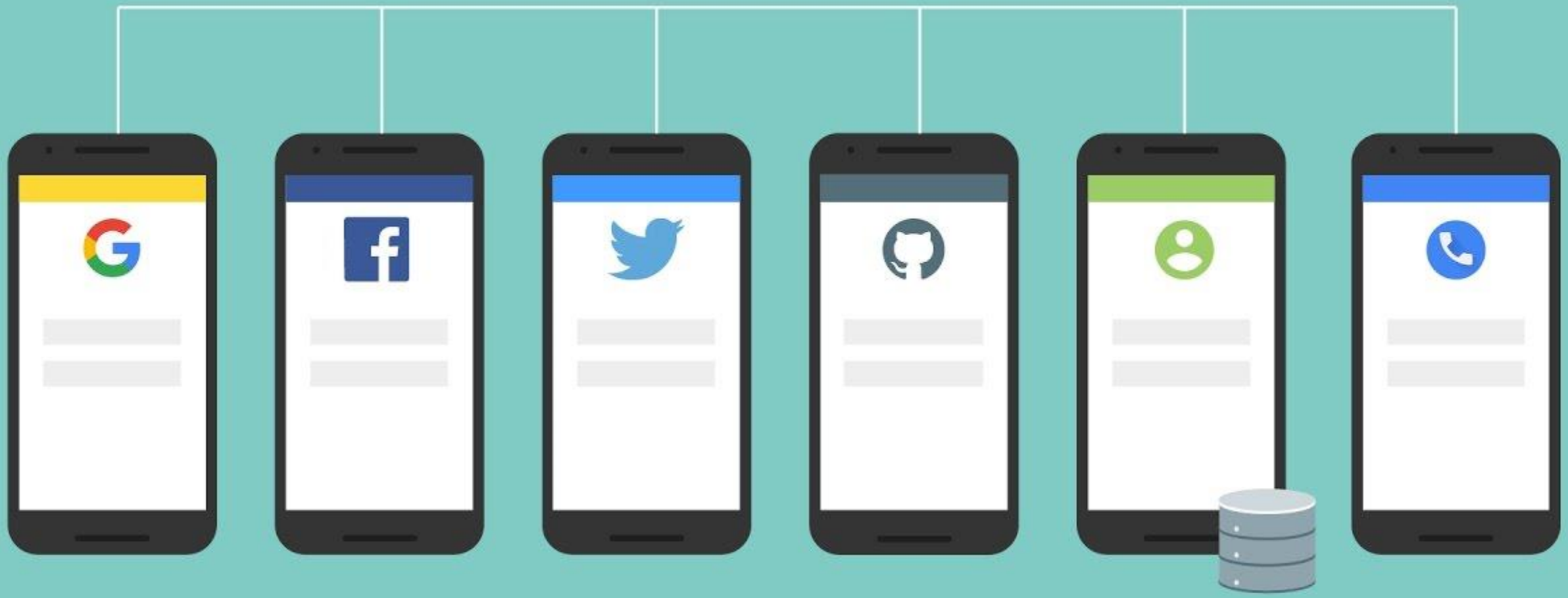
Read data...

```
override fun onChildMoved(dataSnapshot: DataSnapshot, p1: String?) {
    // This method is triggered when a child location's priority changes.
}
// This method will be triggered in the event that this listener either
// failed at the server, or is removed as a result of the security...
override fun onCancelled(databaseError: DatabaseError) {
    Log.w(TAG, msg: "postComments:onCancelled", databaseError.toException())
    Toast.makeText( context: this@MainActivity, text: "Failed to load comment.",
        Toast.LENGTH_SHORT).show()
}
```

```
private fun getTodoEventListener(): ChildEventListener {
    // used to receive events about changes in the child
    // locations of a given DatabaseReference
    val childEventLister = object: ChildEventListener{
        override fun onChildAdded(dataSnapshot: DataSnapshot, previousChildName: String?) {
            val todo = dataSnapshot.getValue(ToDoItem::class.java)
            // add new ToDoItem
            data.add(todo!!)
            val todoIndex = data.indexOf(todo)
            listViewItems.setItemChecked(todoIndex, value: data[todoIndex].checked == true)
            adapter.notifyDataSetChanged()
        }

        override fun onChildChanged(dataSnapshot: DataSnapshot, previousChildName: String?) {
            val todo = dataSnapshot.getValue(ToDoItem::class.java)
            // find modified ToDoItem
            val todoIndex = data.indexOf(todo)
            data[todoIndex].set(todo)
            listViewItems.setItemChecked(todoIndex, value: data[todoIndex].checked == true)
            adapter.notifyDataSetChanged()
        }

        override fun onChildRemoved(dataSnapshot: DataSnapshot) {
            val todo = dataSnapshot.getValue(ToDoItem::class.java)
            data.remove(todo)
            adapter.notifyDataSetChanged()
        }
```



Storage



Authentication

Firebase Authentication

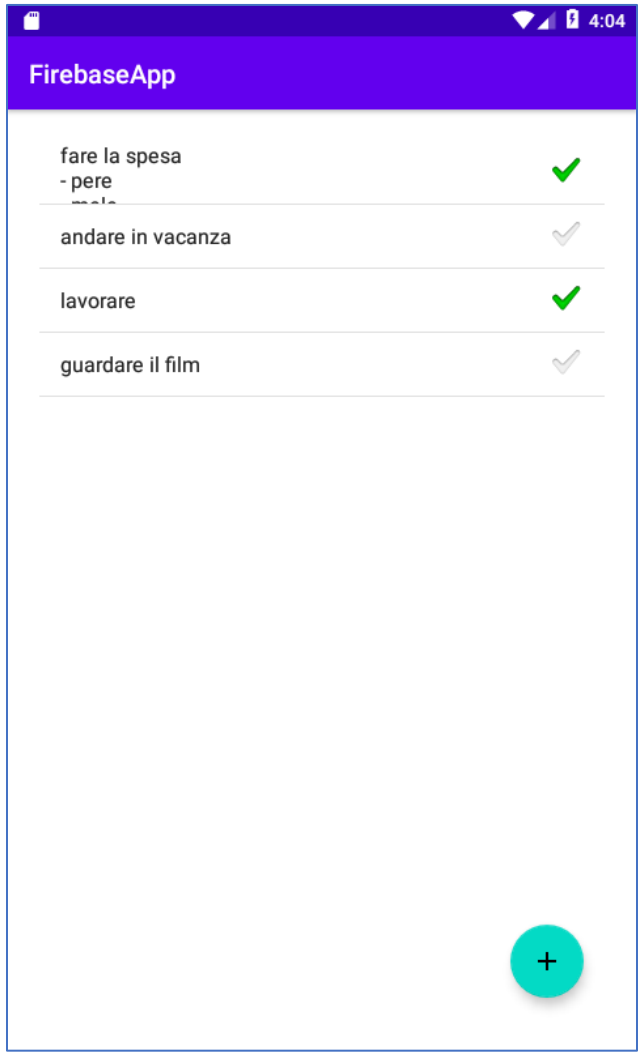
- **Integrate** Realtime Database with Firebase Authentication
- **Simple** and intuitive authentication process.
- To set up your own authentication system: **months**
- But if you use Firebase: **10 lines of code**

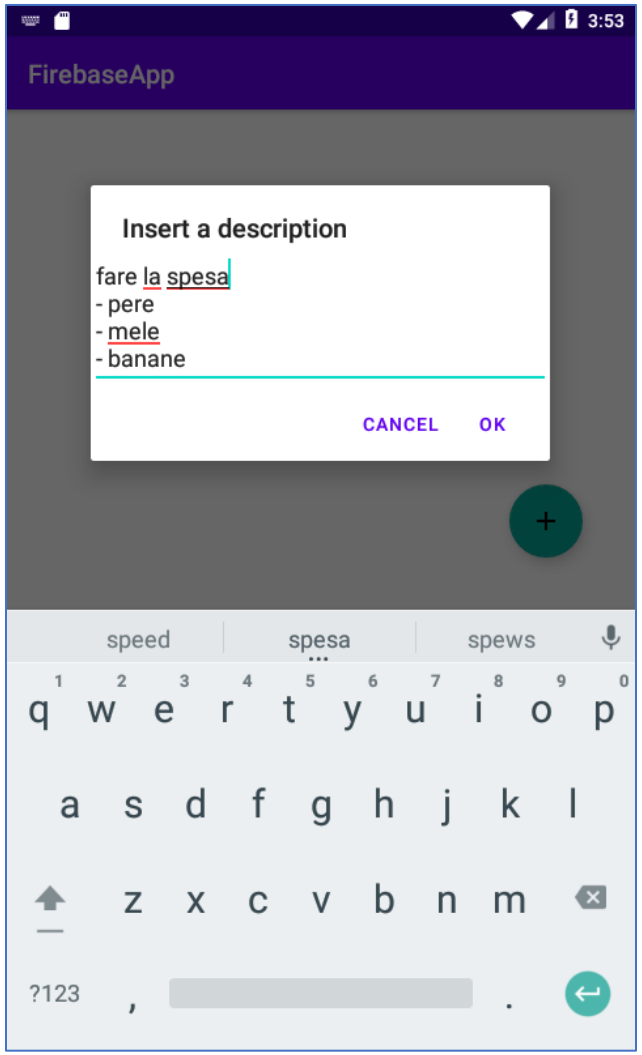
Support for login

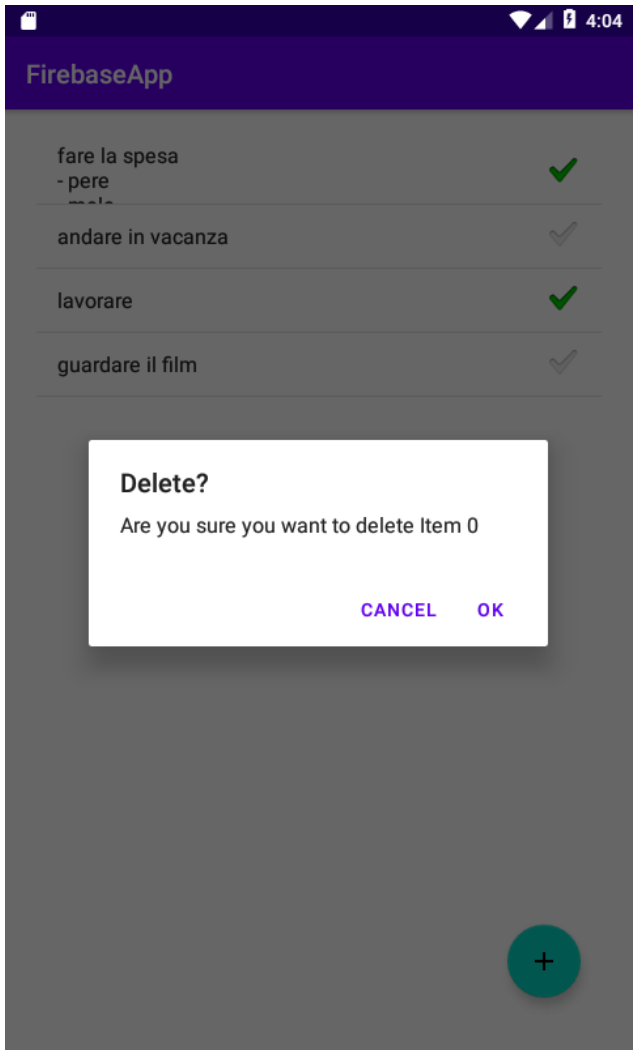
- Common Auth providers
  - Email & Password
  - Phone numbers
  - Google
  - Facebook
  - Twitter
  - Github
  - And more..

- Your own **custom auth tokens** Auth and user management

Example without user interface







Login / Logout

```
class MainActivity : AppCompatActivity() {
  private lateinit var auth: FirebaseAuth
  ...
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    // Initialize Firebase Auth
    auth = Firebase.auth
    ...
  }
```

```
override fun onStart() {
  super.onStart()
  // Check if user is signed in (non-null)
  val currentUser = auth.currentUser
  if(currentUser == null) {
    loginUser("ignazio.gallo@uninsubria.it", "ciao123")
    // val intent = Intent(this, LoginActivity::class.java)
    // startActivity(intent)
    // finish()
  }
}
```

```
override fun onStop() {
  super.onStop()
  auth.signOut()
  Log.w(TAG, "User signOut")
}
```

- Try to automatically login the user in `onStart()`
- And logout in `onStop()`

Login or SignUp

```
private fun loginUser(email: String, password: String) {
    auth.signInWithEmailAndPassword(email, password)
    .addOnCompleteListener(this) { task ->
        if (task.isSuccessful) {
            // Sign in success, update UI with the signed-in user's information
            Log.d(TAG, msg: "signInWithEmail:success")
              val intent = Intent(this, MainActivity::class.java)
              startActivity(intent)
              finish()
        } else {
            // If sign in fails, display a message to the user and try to create it
            Log.w(TAG, msg: "signInWithEmail:failure", task.exception)
            val builder = AlertDialog.Builder( context: this)
            with(builder)
            { this: AlertDialog.Builder
                setTitle("Authentication for ${email} failed")
                setMessage(task.exception?.message)
                setPositiveButton( text: "OK", listener: null)
                show()  ^with
            }
            createUser( userName: "Ignazio",
                 email: "ignazio.gallo@uninsubria.it", password: "ciao123"
            )
        }
    }
}
```

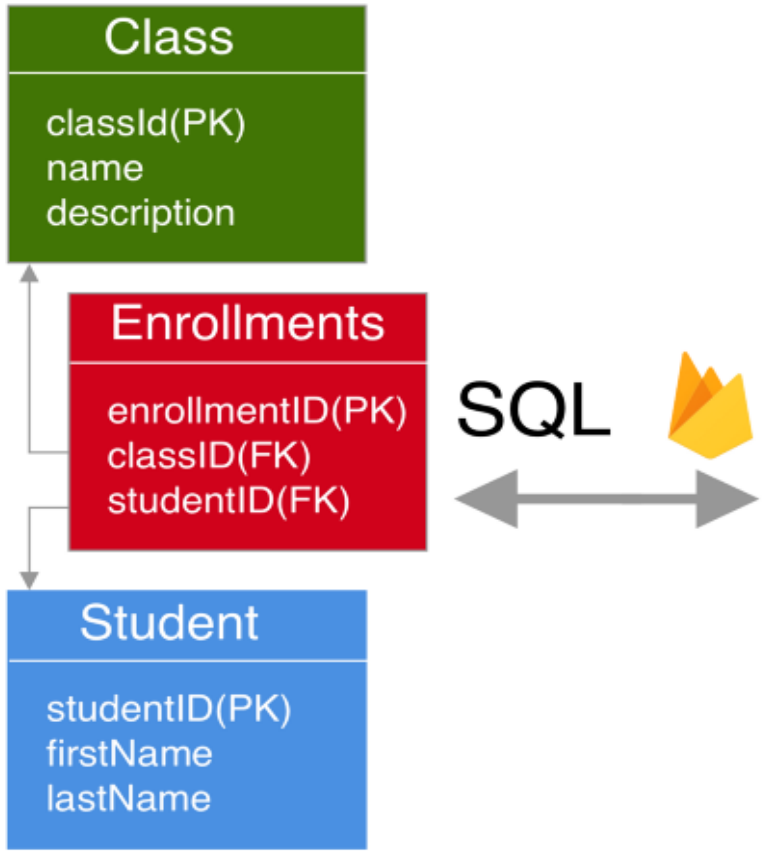
Login or SignUp

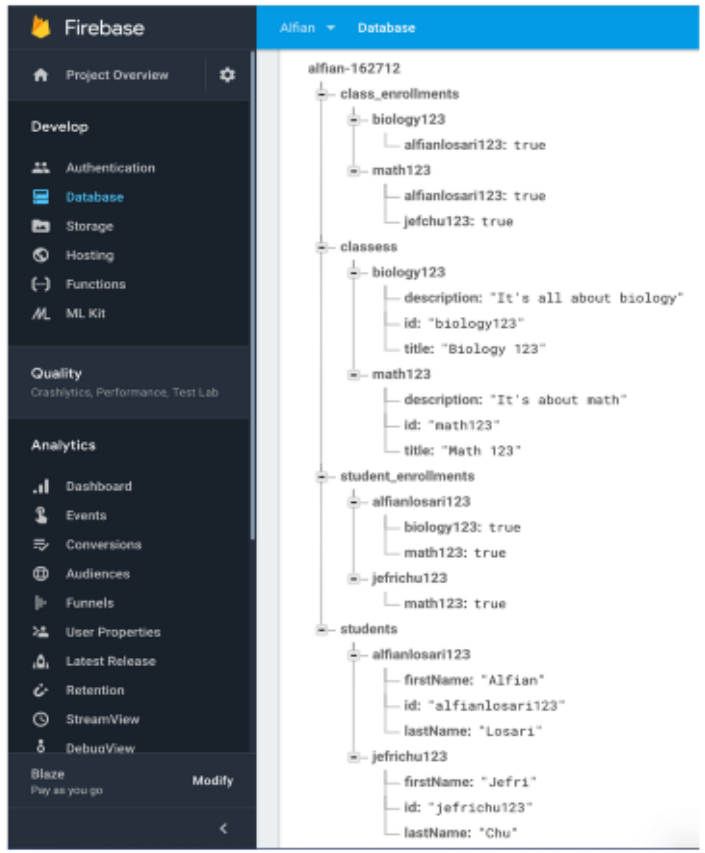
```
private fun createUser(userName: String, email: String, password: String) {
    auth.createUserWithEmailAndPassword(email, password)
    .addOnCompleteListener(this) { task ->
        if (task.isSuccessful) {
            // Sign in success, update UI with the signed-in user's information
            Log.d(TAG, msg: "createUserWithEmail:success")
            Toast.makeText(baseContext, text: "Authentication success.",
                Toast.LENGTH_SHORT).show()
        } else {
            // If sign in fails, display a message to the user.
            Log.w(TAG, msg: "createUserWithEmail:failure", task.exception)
            Toast.makeText(
                baseContext, text: "Authentication failed.",
                Toast.LENGTH_SHORT
            ).show()

            finish()
        }
    }
}
```

Structure Your Database

Firebase Realtime Database schema

- Realtime Database provides freedom for developers to design the schema for their application because of its NoSQL nature





- The developers still need to design the schema carefully so their app can really scale efficiently

JSON tree

- All Firebase Realtime Database data is stored as JSON objects.
- Unlike a SQL database, there are no tables or records.
- When you add data to the JSON tree, it becomes a node in the existing JSON structure with an associated key.

- *For example:*
  to store a basic profile and contact list.
- *A typical user profile is located at a path, such as /**users**/$uid.*
- *The user alovelace might have a database entry that looks something like this*

```
{
  "users": {
    "alovelace": {
      "name": "Ada Lovelace",
      "contacts": { "ghopper": true },
    },
    "ghopper": { ... },
    "eclarke": { ... }
  }
}
```

Best practices: Avoid nesting data

- Firebase Realtime Database allows nesting data up to 32 levels deep, but…
- When you fetch data at a location in your database, you retrieve all of its child nodes.
- When you grant someone read or write access at a node in your database, you also grant them access to all data under that node.
- Therefore, in practice, it's best to **keep your data structure as flat as possible**.

```
{
  // This is a poorly nested data architecture, because iterating
  // the children of the "chats" node to get a list of
  // conversation titles requires potentially downloading hundreds
  // of megabytes of messages
  "chats": {
    "one": {
      "title": "Historical Tech Pioneers",
      "messages": {
        "m1": { "sender": "ghopper",
                "message": "Relay malfunction found. Cause: moth." },
        "m2": { ... },
        // a very long list of messages
      }
    },
    "two": { ... }
  }
}
```

Best practices: Avoid nesting data

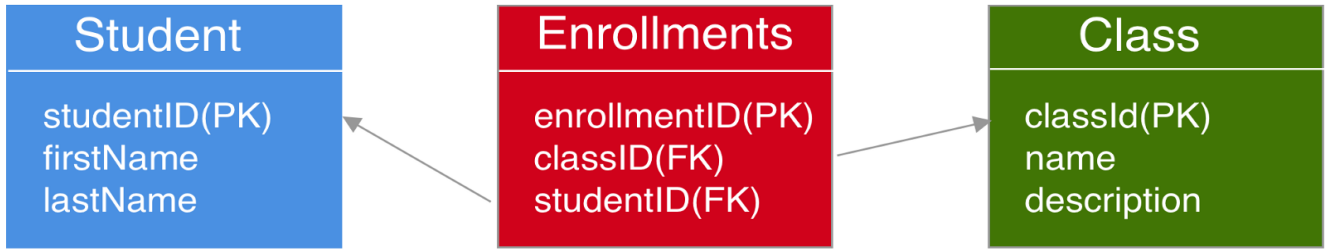
```
{ // Chats contains only meta info about each conversation stored
  // under the chats's unique ID
  "chats": {
    "one": {
      "title": "Historical Tech Pioneers",
      "lastMessage": "ghopper: Relay malfunction found. Cause: moth.",
      "timestamp": 1459361875666
    },
    "two": { ... },
    "three": { ... }
  },
```

```
  // Conversation members are easily accessible and stored by
  // chat conversation ID
  "members": {
    // we'll talk about indices like this below
    "one": {
      "ghopper": true,
      "alovelace": true,
      "eclarke": true
    },
    "two": { ... },
    "three": { ... }
  },
```

```
  // Messages are separate from data we may want to iterate quickly
  // but still easily paginated and queried, and organized by chat
  // conversation ID
  "messages": {
    "one": {
      "m1": {
        "name": "eclarke",
        "message": "The relay seems to be malfunctioning.",
        "timestamp": 1459361875337
      },
      "m2": { ... },
      "m3": { ... }
    },
    "two": { ... },
    "three": { ... }
  }
}
```

Case study: Many to Many Relationship

- A single student can register to many classes
- A class can have many students.
- In **SQL** Database: use join Table called enrolments.



- How do we translate the SQL design schema into the Firebase Realtime Database JSON tree?

Antipattern

- We might try to create 2 top level nodes, a
  - classes and students,
- then inside each class/student child we embed the classes or students in each child like so:
- Cons
  - When we query for the children for example student1 or class1, all the data will also get fetched from the server although we don't need to use the data.
  - We need to query each child class or students to get their enrolments event though we don't need the metadata for the child.

```
classes:
  class1:
    students:
      student1: true
  class2:
    students:
      student1: true
      student2: true
students:
  student1:
    classes:
      class1: true
      class2: true
  student2:
    classes:
      class2: true
```

Using Schema Denormalization (Recommended)

- Denormalize and flatten the data into 4 top level nodes:

1. classes: Store the metadata for each class as the children.
2. students: Store the metadata for each student as the children.
3. class_enrolments: Store the relationship between each child class and students as the children. We use this to lookup the student enrolments for a class.
4. student_enrolments: Store the relationship between each child student and classes as the children. We use this to lookup the classes enrolments for a student.

```
classes:
  class1:
    ...
  class2:
    ...
students:
  student1:
    ...
  student2:
    ...
class_enrolments:
  class1:
    student1: true
    student2: true
  class2:
    student2: true
student_enrolments:
  student1:
    class1: true
  student2:
    class1: true
    class2: true
```