An Introduction to Google Firebase

Lecture 5

Goal of this lecture

- State of the art solution for quick and efficient app development.
- Support by Google technologies (QoS, security, interoperability).
- Develop backend solutions without the need for backend developers.

Content (1/2)

- What is Firebase?
- What features does it offer?
- What platforms does it support?
- Overview of the **Realtime database**
- Support for **user authentication** (client vs. server)
- Overview of Cloud Storage
- Database and Storage Rules
- Notifications

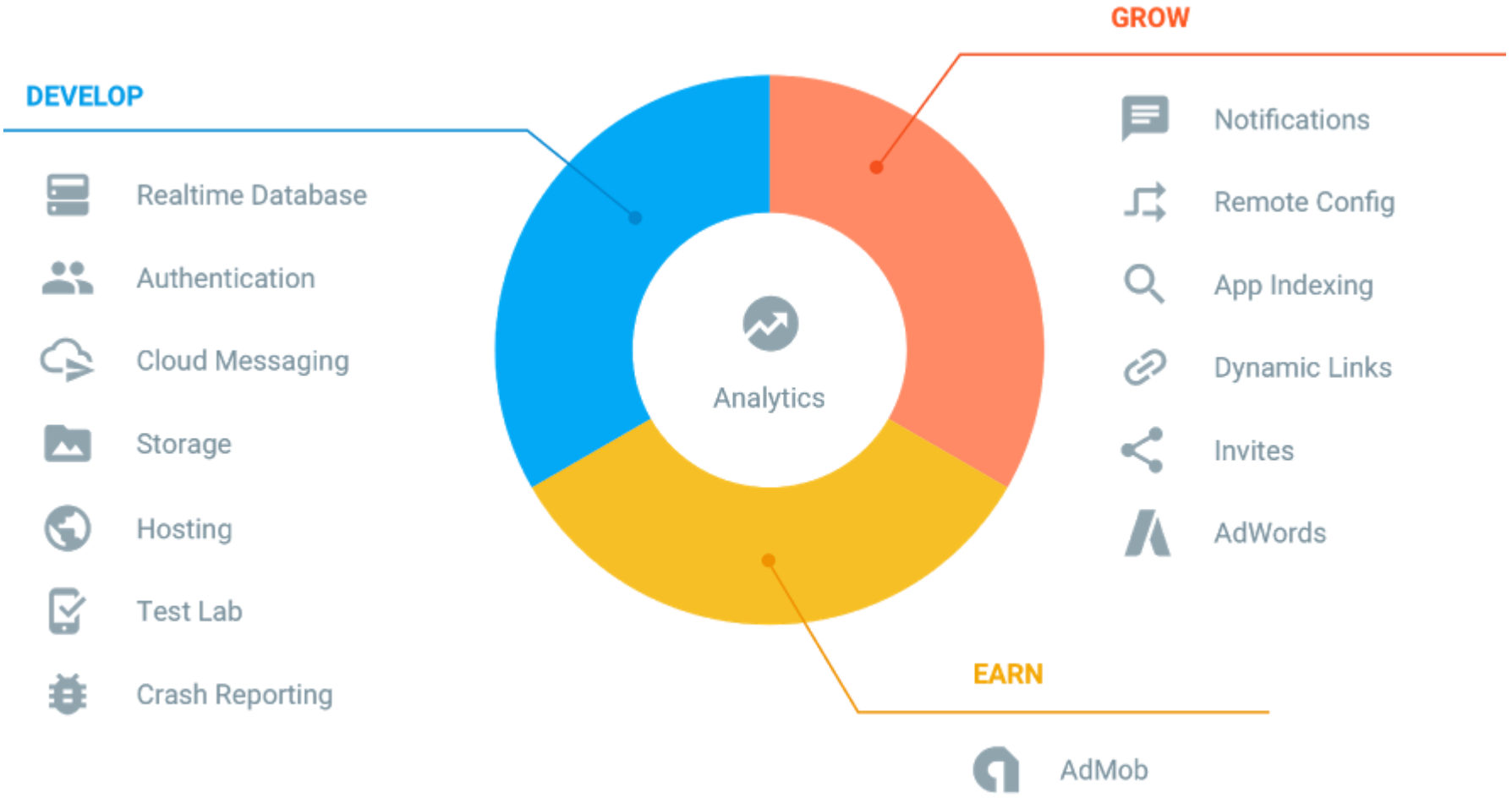
Content (2/2)

- Analytics.
- Firestore and cloud functions
- ~~Integration in academia~~
  - ~~Agora & student projects~~
- ~~Integration in industry~~
  - ~~Canvy~~
- Why choose it over other backend solutions?
- Overview towards the future.

What is Firebase?

- Mobile platform that helps you quickly develop apps, grow your user base, and earn more money.
- Firebase is made up of complementary features that you can mix to fit your needs.

What is Firebase?



What features does it offer?

- Analytics – gather insight about app usage and user behavior.
- Development
  - Cloud messaging
  - Authentication
  - Realtime database, Cloud Firestore
  - Cloud Storage
  - Hosting, Test lab, Crashlytics
- Growth
  - Remote config, Notifications, App indexing, Dynamic links, Invites, Cloud Functions
- Earning
  - AdMob – target audiences with ads

What platforms does it support?

Full online documentation, Code lab, API reference, Samples

- C++ (for Android & iOS)
- Unity
- Flutter
- Firebase Admin SDK (Node.js, Admin Java SDK)
- Python

Real-time database

A cloud-hosted NoSQL database

Data is:

- stored as JSON
- synced across devices in a matter of *ms*
- available when the app goes offline

Console (login with the MSA account)

Realtime database. How does it work?

- Data is persisted locally.
- Even while offline, real-time events continue to fire, giving the end user a responsive experience.
- Conflicts are merged automatically when connectivity is regained.
- NoSQL database with different optimizations and functionality compared to a relational database => need to structure data accordingly.

Realtime database. Structuring data.

Example to get list of chat titles (threads): iterate all data (members, messages)

```
{
 // bad structure
 "chats": {
   "one": {
     "title": "Historical Tech Pioneers",
     "messages": {
       "m1": { "sender": "ghopper", "message": "Relay malfunction found.
Cause: moth." },
       "m2": { ... },
       // a very long list of messages
     }
   },
   "two": { ... }
 }
}
```

Realtime database.

- Get list of chats (titles)
- Get members in each chat
- Get message thread for each chat

```
{
 // better structure → flattening
 "chats": {
   "one": {
     "title": "Historical Tech Pioneers",
     "lastMessage": "ghopper: Relay
malfunction found. Cause: moth.",
     "timestamp": 1459361875666
   },
   "two": { ... },
   "three": { ... }
 },
```

```
 // members info
 "members": {
   "one": {
     "ghopper": true,
     "alovelace": true,
     "eclarke": true
   },
   "two": { ... },
   "three": { ... }
 },

 // messages info
 "messages": {
   "one": {
     "m1": {
       "name": "eclarke",
       "message": "The relay seems to
be malfunctioning.",
       "timestamp": 1459361875337
     },
     "m2": { ... },
     "m3": { ... }
   },
   "two": { ... },
   "three": { ... }
 }
}
```

Realtime database. Set data and listen for changes.

- Database reference: node (path) on which R/W operations can be done

```
databaseRef.child("messages")
.child("one")
.child("m1")
.child("timestamp")
```

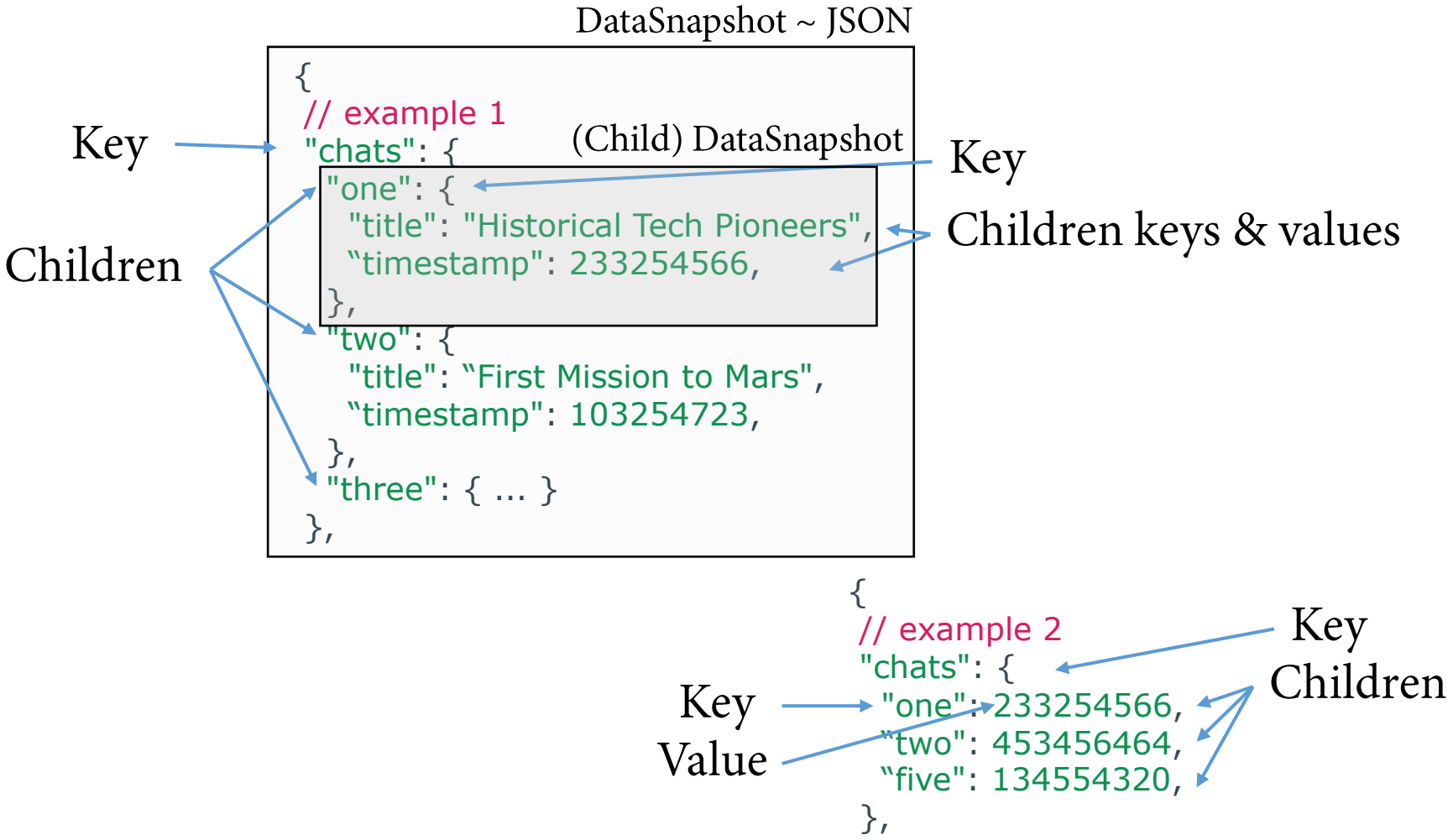
OR any combination using '+' and '/'

```
databaseRef.child("messages/one/"+"m1")
.child("timestamp")
```

Three types of listeners:

- ValueEventListener: returns all data below node.
- SingleValueEventListener: returns all data below node, once.
- ChildEvenentListener : returns only modified child node under current node.

Realtime database. Key, Value & Child.



Realtime database. Set data and listen for changes.

- Listening for changes on a node

```
ValueEventListener msgListener = new ValueEventListener() {
    @Override
    public void onDataChange(DataSnapshot dataSnapshot) {
        // Get Post object and use the values to update the UI
        Message msg = dataSnapshot.getValue(Message.class);
        // ...
    }

    @Override
    public void onCancelled(Dat
        // Getting Message failed
        Log.w(TAG, "loadMessage
        // ...
    }
};
databaseRef.addValueEventListener(msgListener);
```

Triggered every time something changes below the databaseRef node.

All data below the node will be re-downloaded

Realtime database. Direct mapping to model class

```
@IgnoreExtraProperties
public class Message {
    private String name;
    private String message;
    private long timestamp;

    public Message() {
        // Default constructor required for calls to
DataSnapshot.getValue(Mesage.class)
    }

public Message(String name, String message, long timestamp) {
        // …
     }

}
```

Realtime database. Set data and listen for changes.

- Listening for changes on a node

```
ChildEventListener childEventListener = new ChildEventListener() {
   @Override
   public void onChildAdded(DataSnapshot dataSnapshot, String
previousChildName) {...}
   @Override
   public void onChildChanged(DataSnapshot dataSnapshot, String
previousChildName) {...}
   @Override
   public void onChildRemove
   @Override
   public void onChildMoved(
previousChildName) {...}
};
ref.addChildEventListener(ch
```

Triggered every time something changes below the databaseRef node.

Only modified child below the node will be re-downloaded

Realtime database. Set data and listen for changes.

➢Writing data

1. Overwrite key with single value

```
databaseRef.child("messages").child("one").child("m1").c
hild("message").setValue("This is a test message");
```

2. Overwrite key with object value

```
Message msg = new Message(name, message, timestamp);
databaseRef.child("messages").child("one").child("m1")
.setValue(msg);
```

Realtime database. Set data and listen for changes.

➢Writing data

3. Update key with values list

```
Map<String, Object> childUpdates = new HashMap<>();
    childUpdates.put("name", name);
    childUpdates.put("mesasage", "This is a test message");
    childUpdates.put("timestamp", getTime());
    databaseRef.child("messages").child("one").child("m1")
.updateChildren(childUpdates);
OR  childUpdates.put("messages/one/m1/timestamp", getTime());
    databaseRef.updateChildren(childUpdates);
```

Realtime database. Offline persistence.

- Firebase apps automatically handle temporary network interruptions for you.
- Queues r/w operations locally

```
FirebaseDatabase.getInstance().setPersistenceEnabled(true);
```

Transactions are not persistent across app restarts (use other/custom solution...)

Client user authentication

Support for different providers:

- Email/password (managed by Firebase)
- Phone number
- Google (+Play Games), Facebook, Twitter, GitHub
- Anonymous

Templates for: email verification, password reset, change email

Client user authentication

- Listener for auth events:

```
mAuthListener = new FirebaseAuth.AuthStateListener() {
    @Override
    public void onAuthStateChanged(@NonNull FirebaseAuth firebaseAuth)
{
        FirebaseUser user = firebaseAuth.getCurrentUser();
        if (user != null) {
            // User is signed in

        } else {
            // User is signed out

        }
    }
};
```

```
onStart() {

FirebaseUser user =
FirebaseAuth.getInstance().getCurrentUser();

}
```

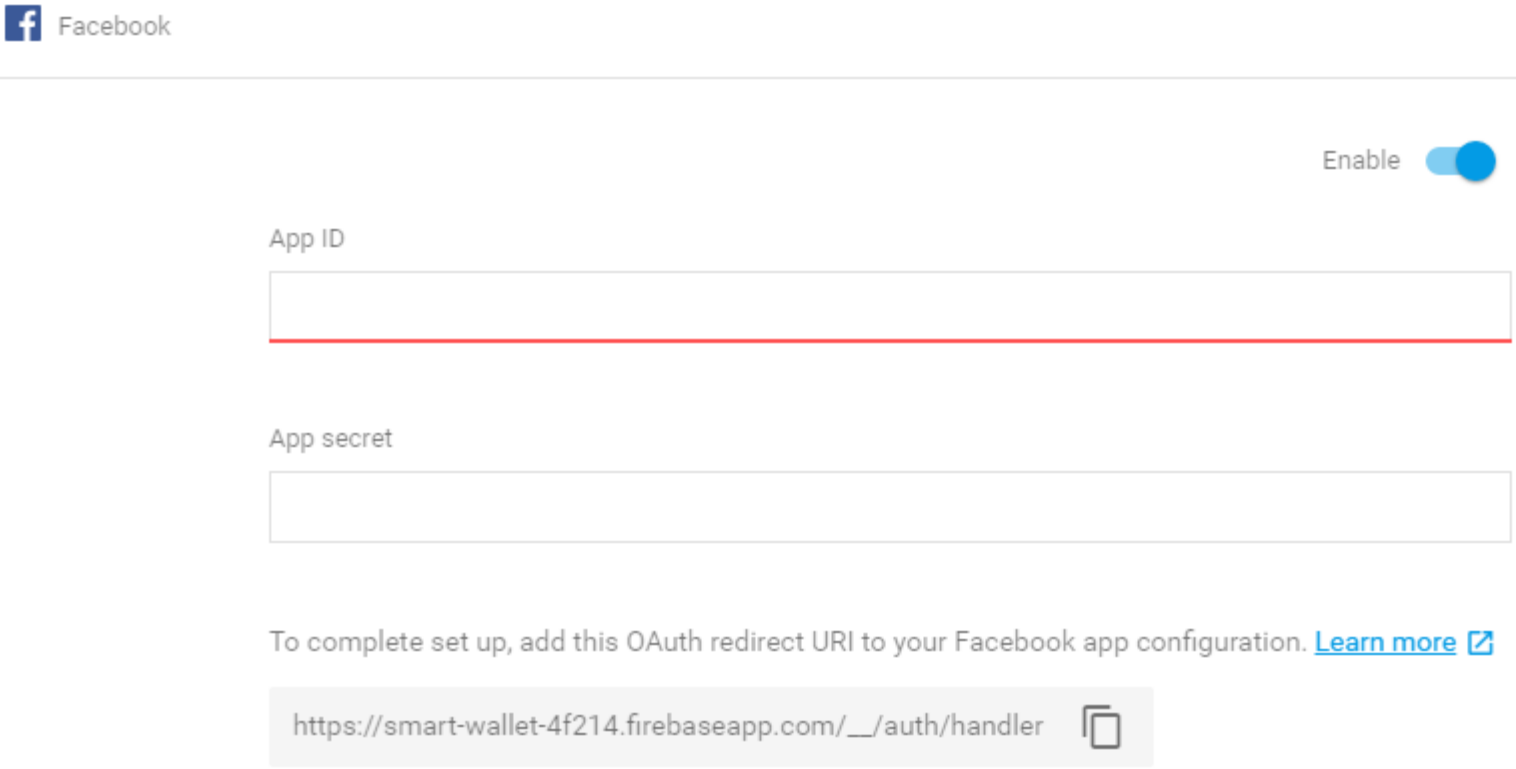
Client user authentication

- User management operations with callbacks:

```
mAuth = FirebaseAuth.getInstance();

mAuth.createUserWithEmailAndPassword(email, password)
mAuth.signInWithEmailAndPassword(email, password)
mAuth.sendPasswordResetEmail(email)


user.updateProfile(profileUpdates)
user.updateEmail("user@example.com")
user.sendEmailVerification()
user.updatePassword(newPassword)
user.reauthenticate(credential)

user.delete()
```

Client user authentication

- Use APP/API key and secret for providers other than Google/Firebase:



Console

Server user authentication

- Integrate Admin Auth API with own servers
- Java SDK, Node.js SDK, REST API
- Service account needed

Server user authentication

- User management: programmatic access to users; it offers additional operations (compared to Firebase console)
- Node.js, Java, Python, Go, C# SDKs
- Retrieve, create, update and remove user credentials
- Example (Node.js):

```
admin.auth().getUserByEmail(email)
 .then(function(userRecord) {
   // success
 })
 .catch(function(error) {
   // fail
});
```

Server user authentication

- Custom authentication: server composes different claims for a user account
- Example (Java):

```
String uid;
HashMap<String, Object> claims = new HashMap<String, Object>();
claims.put("paidAccount", true);

FirebaseAuth.getInstance().createCustomToken(uid, claims)
    .addOnSuccessListener(new OnSuccessListener<String>() {
        @Override
        public void onSuccess(String token) {
            // send retrieved token back to requester
        }
});
```

Server user authentication

- Custom authentication: client uses retrieved custom server token to authenticate current user with Firebase
- Example (Java):

```
FirebaseAuth.getInstance().signInWithCustomToken(token)
        .addOnCompleteListener(this, new OnCompleteListener<AuthResult>() {
            @Override
            public void onComplete(@NonNull Task<AuthResult> task) { }
        }
});
```

Server user authentication

- Identity verification: server can perform Firebase operations on behalf of the user
- Example (Java):

```
String idToken; // sent by client
FirebaseAuth.getInstance().verifyIdToken(idToken)
    .addOnSuccessListener(new OnSuccessListener<FirebaseToken>() {
        @Override
        public void onSuccess(FirebaseToken decodedToken) {
            String uid = decodedToken.getUid();
        }
});
```

Cloud Storage

- Same hierarchical structure as Firebase Realtime database.
- Used for storing and retrieving user-generated content.
  - Large binary data (e.g., photos, videos)
- **Robust**: an upload/download restarts where it was initially stopped.
- **Secure**: it can integrate with Firebase Authentication or use a declarative security model.
- **Scalable**: it's backed by Google Cloud Storage.

Storage

- Create a reference to the storage resource:

```
String uid;

StorageReference rootRef =
FirebaseStorage.getInstance().getReference().getRoot();

StorageReference imageRef =
rootRef.child("images").child(uid).child("avatar.png");
OR
StorageReference imageRef = rootRef.child("images/“ + uid +
“/avatar.png");
```

Storage

- Upload a local file to Firebase:

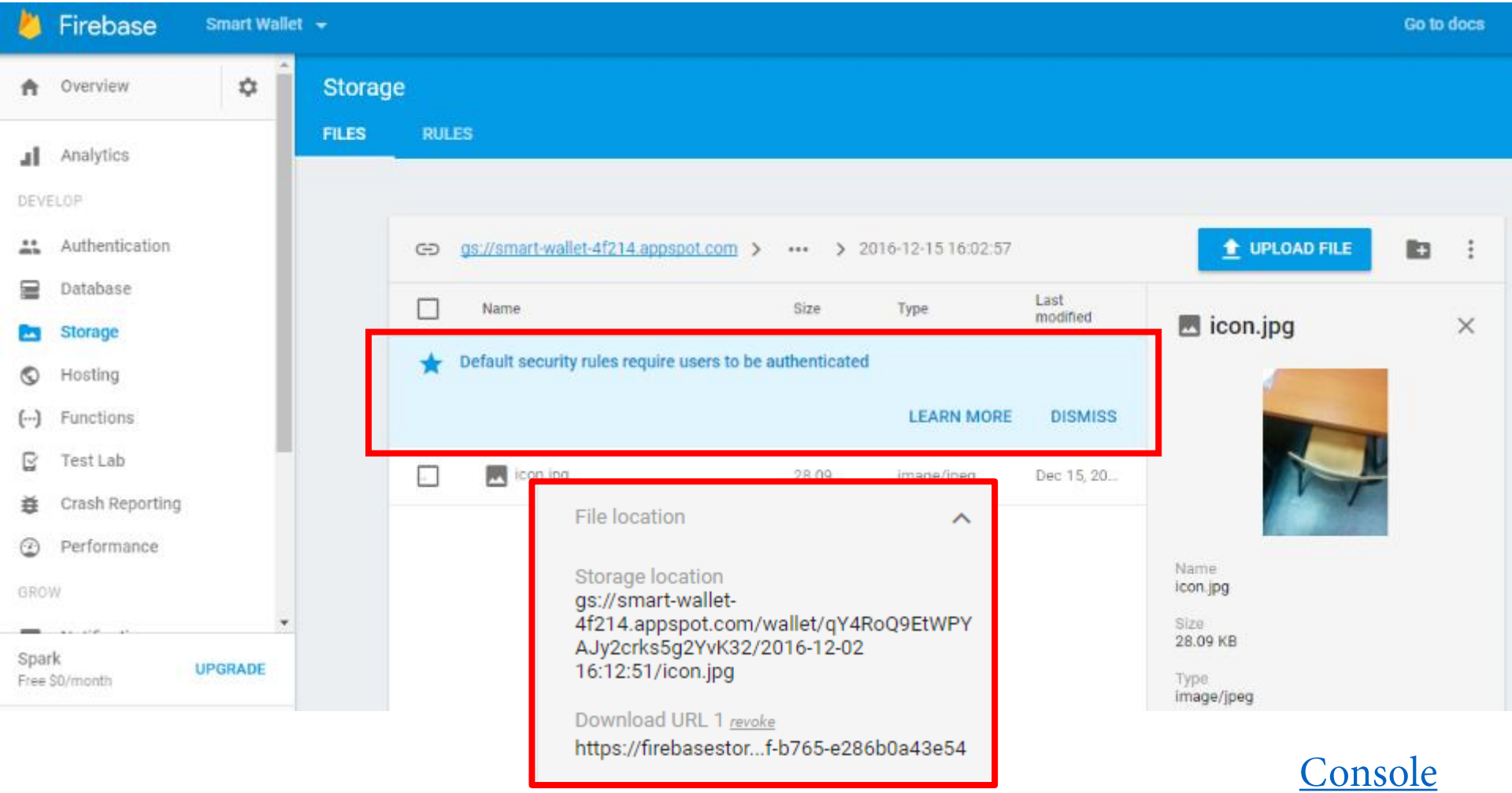
```
File file;
imageRef.putFile(file)
    .addOnFailureListener(new OnFailureListener() {
          @Override
          public void onFailure(Exception exception) { // handle
failure }
     })
    .addOnSuccessListener(new
OnSuccessListener<UploadTask.TaskSnapshot>() {
          @Override
          public void onSuccess(UploadTask.TaskSnapshot taskSnapshot) {
               Uri downloadUrl = taskSnapshot.getDownloadUrl();
          }
     });
```

Storage

- Download a file from Firebase:

```
File file;
imageRef.getFile(file)
     .addOnFailureListener(new OnFailureListener() {
          @Override
          public void onFailure(Exception exception) { // handle
failure }
      })
      .addOnSuccessListener(new
OnSuccessListener<FileDownloadTask.TaskSnapshot>() {
          @Override
          public void onSuccess(FileDownloadTask.TaskSnapshot
taskSnapshot) {    // content downloaded       }
       });
```

Storage. *Backend*



Console

Database Rules

- Supports .read, .write, .validate, .indexOn
- .read and .write rules cascade
- .read and .write rules shallower in the database override deeper rules
- .validate rules do not cascade
- E.g., a chats node:

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

Console

Database Rules

- Enforce that only authenticated users can write and read:

```
{
 "rules": {
   “chats": {
     ".read": "auth != null",
     ".write": "auth != null"
   }
 }
}
```

Database Rules

- Enforce write only for “owner”:

```
{
 "rules": {
   “chats": {
     "$uid": {
       ".write": "$uid === auth.uid“,
       ".read": true
     }
   }
 }
}
```

Database Rules

- Apply different validations on new data:

```
{
 "rules": {
   “chats": {
     "$uid": {
       ".validate": "newData.hasChildren(['msg',
'timestamp']) && newData.child('msg').val().length < 100"
     }
   }
 }
}
```

Database Rules

- Index children based on timestamp:

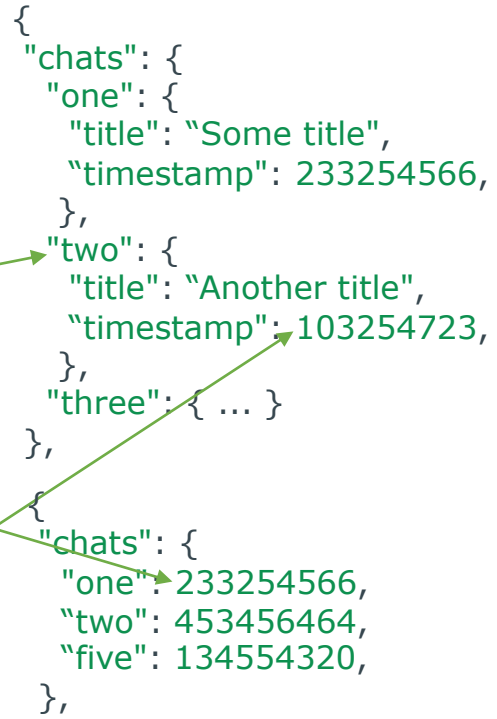
```
{
 "rules": {
   “chats": {
     "$uid": {
       ".indexOn": ["timestamp"]
     }
   }
 }
}
```

A node's **key** is indexed automatically, so there is no need to index it explicitly.

Sorting & filtering data

Use Query class to retrieve data sorted by:

- child key – `orderByKey()`
- child value – `orderByValue()`
- value of a child key – `orderByChild()`



Filter the sorted result to a specific number of results, or a range of keys or values.

Filtering on client is expensive → use indexOn rule

Sorting data example

Synch data on client, ordered by number of likes for each post:

```
"user-posts": {
  "user2": {
     "post1": {
        "msg" : "Hello world",
        "likes" : "1",
        "timestamp": "11:34:27"
        "sentTo" : "user3"
     },
     "post2": {
        "msg" : "I'm hungry",
        "likes" : "22",
        "timestamp": "12:14:67"
        "sentTo" : "user3"
     },
  }
}
```

Approach 1: sorting on the client

Download whole posts list to local array, then sort

- Possibly huge overhead

```
String myUid = “user2”;
databaseReference.child("user-posts").child(myUid)
.addListenerForSingleValueEvent(new ValueEventListener() {
    public void onDataChange(DataSnapshot dataSnapshot) {
        // get list of user posts
        List<UserPost> posts = new ArrayList<>();
        for (DataSnapshot snapshot : dataSnapshot.getChildren()) {
            UserPost post = snapshot.getValue(UserPost.class);
            posts.add(post);
        }
        // sort by likes
        Collections.sort(posts, new Comparator<UserPost>() {
            public int compare(UserPost p1, UserPost p2)
                { return p1.likes.compareTo(post2.likes);  }
        });
    } });
```

Approach 2: sorting on the server

OrderByChild & limit to first N entries, then download sorted list

- Minimal overhead, less data traffic, less client processing

```
String myUid = “user2”;
Query userPostsByLikesRef = databaseReference.child("user-
posts").child(myUid).orderByChild(“likes").limitToFirst(100);

userPostsByLikesRef.addChildEventListener(new ChildEventListener() {
    // TODO ...
});
```

Filtering data

Combine any of the limit or range methods with an order-by method when constructing a query.

```
limitToFirst()   Start from beginning of the ordered list of results.
limitToLast()    Start from the end of the ordered list of results.
startAt()        Items >= to the specified key, depending on the order-
by method chosen.
endAt()          Items <= to the specified key, depending on the order-
by method chosen.
equalTo()        Items == to the specified key or value.
```

Filtering data example

You may combine multiple limit or range functions.

For example, you can combine the startAt() and endAt() methods to limit the results to a specified range of values.

```
// no ordering applied
var postsRef1 = firebase.database().ref('user-posts').limitToLast(100);

// get first 100 posts by likes, regardless of the number of likes
var postsRef2 = firebase.database().ref('user-posts')
        .orderByValue('likes')
        .limitToFirst(100);

// get all posts with [10, 100] likes
var postsRef3 = firebase.database().ref('user-posts')
        .orderByValue('likes')
        .startAt(10)
        .endAt(100);
```

Indexing Data

- orderByChild – explicit indexing by *height, length*

```
“uid_eX434frefr”: {
  “dinosaurs”: {
  "lambeosaurus": {
    "height" : 2.1,
    "length" : 12.5,
    "weight": 5000
  },
  "stegosaurus": {
    "height" : 4,
    "length" : 9,
    "weight" : 2500
  }
 }
}
```

```
{
  "rules": {
     “$uid”: {
       "dinosaurs": {
          ".indexOn": ["height", "length"]
       }
     }
  }
}
```

Indexing Data

- orderByChild
- Retrieve a list of *dinosaurs* sorted by their *height*:

```
var myUserId = firebase.auth().currentUser.uid;
var topDinosRef = firebase.database().ref(myUserId +
'/dinosaurs').orderByChild('height');
```

Indexing Data

- orderByValue – explicit indexing by *scores*

```
{
  "scores": {
    "bruhathkayosaurus" : 55,
    "lambeosaurus" : 21,
    "linhenykus" : 80,
    "pterodactyl" : 93,
    "stegosaurus" : 5,
    "triceratops" : 22
  }
}
```

```
{
  "rules": {
    "scores": {
      ".indexOn": ".value"
    }
  }
}
```

Storage Rules

- Specify authorization rules per file and per path that determine access to the files in your app.
- The default Storage Security Rules require Firebase Authentication in order to perform any read or write operations on all files

```
service firebase.storage {
 match /b/<firebase-storage-bucket>/o {
   // flat representation "friends/<UID>/avatar.png"
   match /friends/{userId}/{allPaths=**} {
     allow write: if request.auth.uid == userId;
     allow read: if request.auth != null;
   }
 }
}
```

Storage Rules

- Firebase Security Rules for Cloud Storage can also be used for data validation, including validating file **name** and **path**, as well as file **metadata** properties such as contentType and size.

```
service firebase.storage {
 match /b/{bucket}/o {
   match /images/{imageId} {
     // Only allow uploads of any image file that's less than 5MB
     allow write: if request.resource.size < 5 * 1024 * 1024
            && request.resource.contentType.matches('image/.*');
   }
 }
```

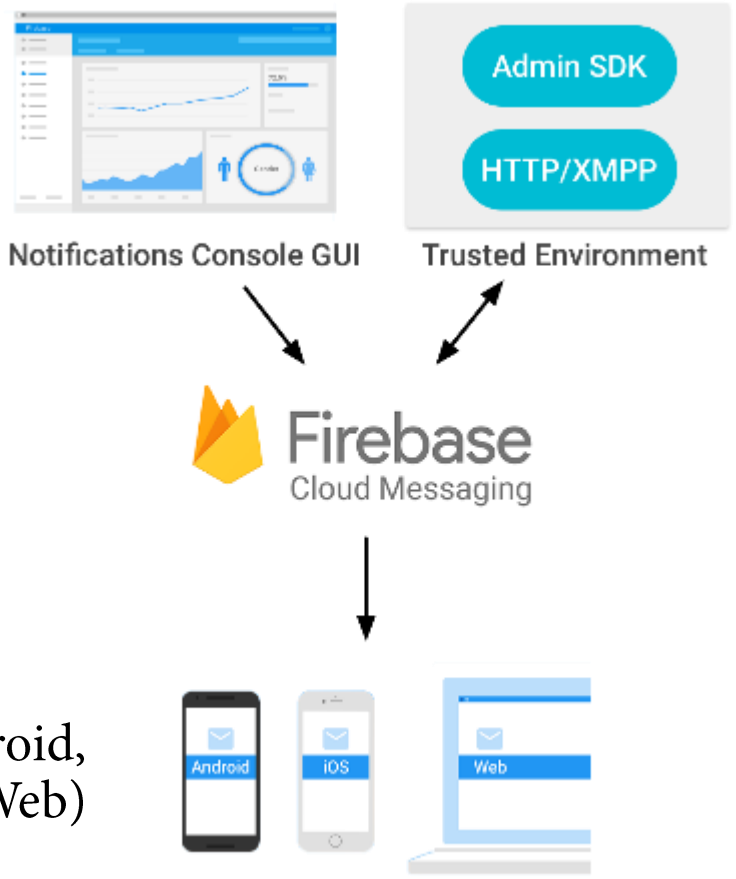
Notifications (using Firebase)

- Built on Firebase Cloud Messaging (FCM)
- Messages at no cost (4 KBytes)
- Fast user notifications (e.g. for marketing campaigns)
- Integrates with Firebase Analytics
- Battery efficient (it's what Google states)
- Broadcast to: single, group of devices, or #subscribed to topic

Demo: https://github.com/firebase/quickstart-android/tree/master/messaging

Notifications

Browser/
Firebase
console



Firebase cloud
functions / App server

Client app (Android,
iOS, Web)

Notifications

Send notifications to:

- A single device – identified by its unique registration token
- Multiple devices: registered to topic, *or* all owned by same user

```
FirebaseMessaging.getInstance().subscribeToTopic("news");
```

Receive Notifications

Extend FirebaseMessagingService.

Override onMessageReceived and onDeletedMessages.

It should handle any message within 10 seconds of receipt. After that, Android does not guarantee execution.

If the app needs more time to process a message, use the Firebase Job Dispatcher.

➔ The notification is delivered to the device's system tray, and the data payload is delivered in the extras of the intent of your launcher Activity.

Firebase Functions

Run backend code in response to events triggered by Firebase features and HTTPS requests.

Code (JavaScript, TypeScript) is stored on Google's cloud and runs in a managed environment.

→ No need to manage and scale own servers.

- Capabilities

  Integration – can respond to events generated by Realtime DB, Firestore, Auth, Analytics, Storage, Pub/Sub triggers, http triggers.

  Zero maintenance – scaling of resources is done automatically.

  Logic is kept private – secure from reverse engineering or tampering on the client side.

Function lifecycle

1. Write code for new function, select an event provider (e.g. Realtime Database), and define the conditions for the function to execute.
2. Deploy the function (Firebase connects it to the selected event provider automatically).
3. When a matching event is generated by the provider, the function code is invoked.
4. If the function is busy handling many events, Google creates more instances to handle work faster.
5. When deploying updated function code, all instances for the old version are cleaned up and replaced by new instances.
6. When deleting the function, all instances are cleaned up, and the connection between the function and the event provider is removed.

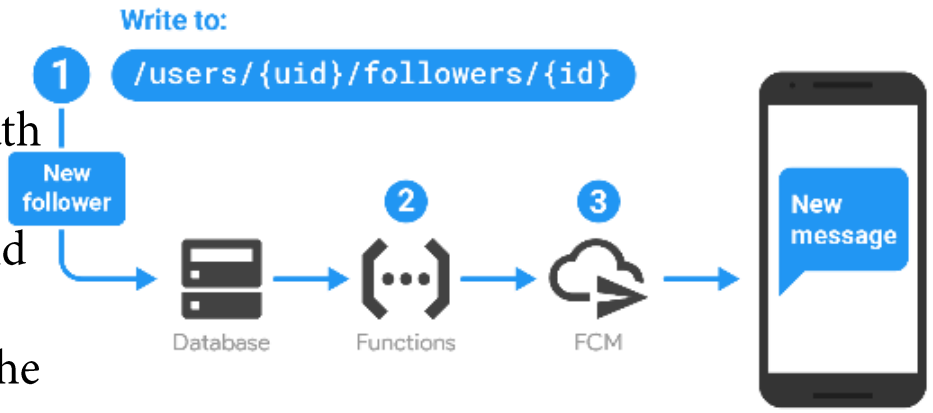
Cloud functions use cases

1. Notify users when something interesting happens.
2. Perform database sanitization and maintenance.
3. Execute intensive tasks in the cloud instead of in your app.
4. Integrate with third-party services and APIs.

Cloud functions use cases

1. Notify users when something interesting happens.

E.g., an app allows users to follow one another's activities in the app, such as being *followed* by another user.



→The function triggers on writes to DB path where followers are stored.

→The function composes a message to send via FCM (cloud messaging).

→FCM sends the notification message to the user's device.

With functions (1) the target user doesn't need to listen on the followers path, (2) receives a notification on updates without client side code.

Cloud functions use cases

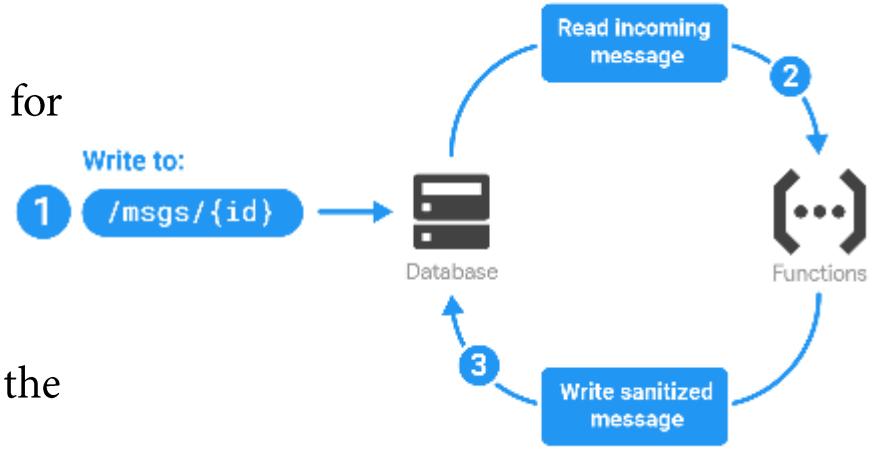
2. Perform database sanitization and maintenance.

E.g., in a chat room app using the Realtime DB, you may monitor write events and eliminate inappropriate or profane text from users' messages.

→ The function's database event handler listens for write events on a specific path, and retrieves event data containing the text of any chat messages.

→ The function processes the text to detect any inappropriate language.

→ The function writes the updated text back to the database.



With functions (1) the target user doesn't receive the inappropriate text at all, (2) nor does he have to implement to client-side filtering, then update the DB.

Cloud functions use cases

3. Execute intensive tasks in the cloud instead of in your app.

E.g., listen for image uploads to *Storage*, download the image to the instance running the function, modify it, and upload it back to *Storage*.

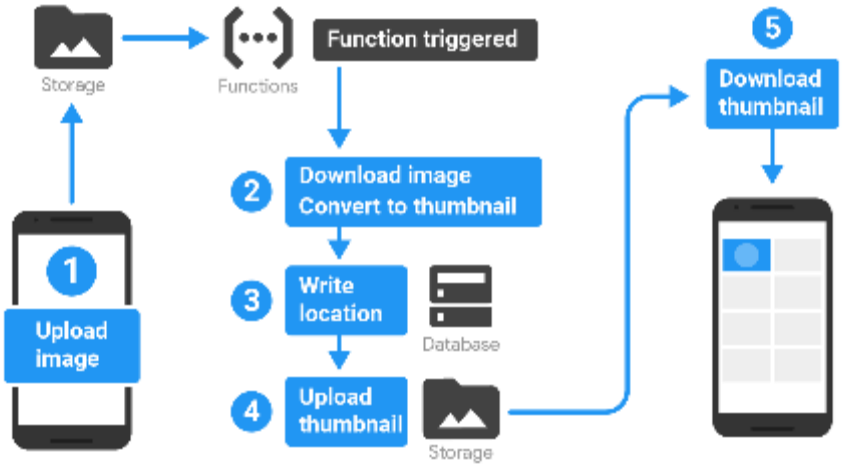
→ A function triggers when an image file is uploaded to Storage.

→ The function downloads the image and creates a thumbnail version of it.

→ The function writes that thumbnail location to the database, so a client app can find and use it.

→ The function uploads the thumbnail back to Storage in a new location.

→ The app downloads the thumbnail link.



With functions (1) the target user doesn't need to download the large files, (2) and process them.

Cloud functions use cases

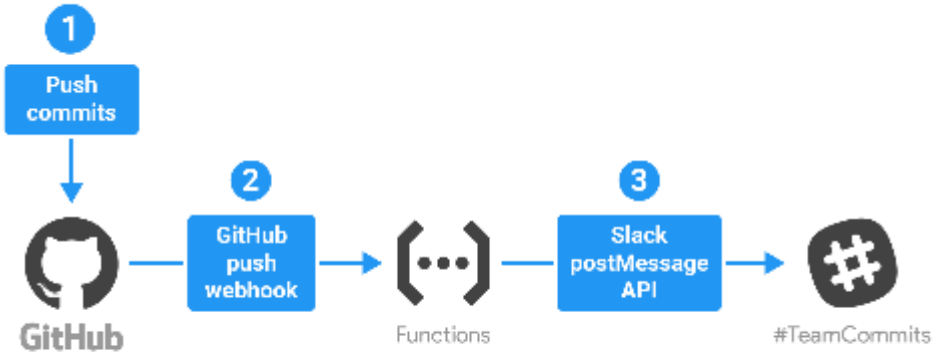
4. Integrate with third-party services and APIs.

E.g., an app used for collaboration on development could post GitHub commits to a workgroup chat room.

→A user pushes commits to a GitHub repo.

→An HTTPS function triggers via the GitHub webhook API.

→The function sends a notification of the commit to a team Slack channel.



Cloud functions implementation

→Through Firebase CLI (requires Node.js and npm library manager).

→Cloud Functions run Node 10, 12, 14 (i.e. it is recommended to develop locally with the same version).

→Install the Firebase CLI via npm: `npm install -g firebase-tools`

→Development via Node, deployment via firebase CLI.

→Full example here.

Analytics

Free app measurement solution that provides insight on app usage and user engagement.

Any app which integrates with Firebase will have Analytics enabled

→ No need to use any other features (e.g. database, auth)

- Reports of how your users behave, useful for decisions regarding app marketing and performance optimizations.
- Capabilities

  Unlimited /free reporting for up to 500 distinct events

  Custom audiences can be defined based on device data, custom events, or user properties. These audiences can be used with other Firebase features when targeting new features or notification messages.

Analytics implementation

1. Connect app with Firebase

   Automatically view data in console.

2. Log custom data

   Log custom events that make sense for your app, e.g. in-app purchases, user achievements etc.

3. Create audiences

   Define the audiences that matter to your app through the Firebase console.

4. Target audiences

   Use your custom audiences to target messages, promotions, or new app features using other Firebase features, such as FCM, and Remote Config.

Analytics implementation

The SDK logs two primary types of information:

- Events – what is *happening* in the app (e.g. user actions, system events, errors).
- User properties – attributes defined to describe segments of the user base (e.g. language preference, geographic location).

```
compile 'com.google.firebase:firebase-core:16.0.5'
// Obtain the FirebaseAnalytics instance
private FirebaseAnalytics mFirebaseAnalytics; …
mFirebaseAnalytics = FirebaseAnalytics.getInstance(this);
Bundle bundle = new Bundle();
bundle.putString(FirebaseAnalytics.Param.ITEM_ID, id);
bundle.putString(FirebaseAnalytics.Param.ITEM_NAME, name);
bundle.putString(FirebaseAnalytics.Param.CONTENT_TYPE,
"image");
mFirebaseAnalytics.logEvent(FirebaseAnalytics.Event.SELECT_
CONTENT, bundle);
```

Events collected by analytics

Some events are automatically collected by Firebase Analytics:

first_open, in_app_purchase, user_engagement, session_start, app_update, app_remove, os_update, app_clear_data, app_exception, notification_*, dynamic_link_* etc.

Some user properties are automatically collected by Firebase Analytics:

Age, Gender, App Store, App Version, Country, Device info, Interests, Language, OS Version etc.

Cloud Firestore

Similar to the Firebase Realtime Database:

→ data in sync across clients, offline & responsive support.

Also offers integration with other Firebase and Google Cloud Platform products, including Cloud Functions.

→Flexible, hierarchical data structures stored in documents, organized into collections. Documents can contain complex nested objects in addition to sub-collections.

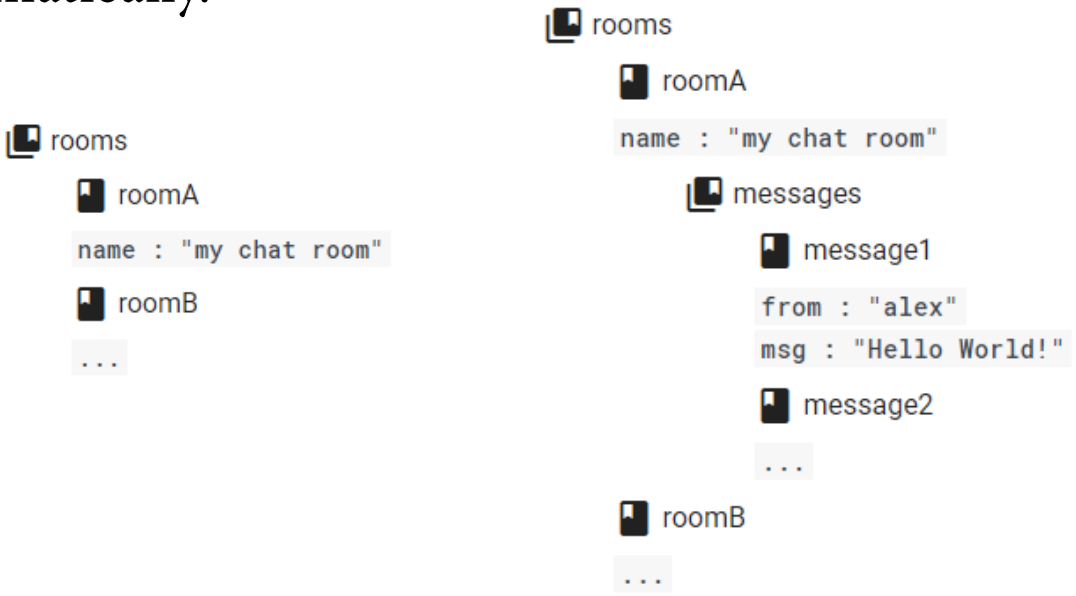
→Use powerful queries to retrieve individual, specific documents or to retrieve all the documents in a collection that match your query parameters.

Cloud Firestore data model

- NoSQL available via native SDKs for: iOS, Android, web, Node.js, Java, Python, Go, and REST&RPC APIs.
- Organized as { Collections {Documents, subcollections {…}}}
- Documents can be primitives or nested objects.
- Each document contains a set of key-value pairs; optimized for storing large collections of small documents.
- Supported data types for values: boolean, number, string, geo point, binary blob, and timestamp. You can also use arrays or nested objects, called maps, to structure data within a document.

Cloud Firestore data model

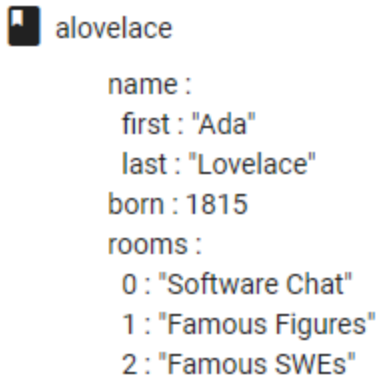
- Collections contain documents, and are automatically created.
- If all documents in a collection are deleted, the collection is removed automatically.



- Cannot have *doc* in *doc* or *coll* in *coll*, only *coll-doc-coll-doc* etc!

Cloud Firestore structuring data

Documents in subcollections can contain subcollections as well, allowing you to further nest data. You can nest data up to 100 levels deep.



1. Nested data in documents

Advantages: simple, fixed lists of data

Limitations: can't run queries on nested lists;

isn't as scalable as other options, in time;

can lead to slower document retrieval times.

Use case: e.g. a chat app, for storing a user's top 3 most visited chat rooms as a nested list in their profile.

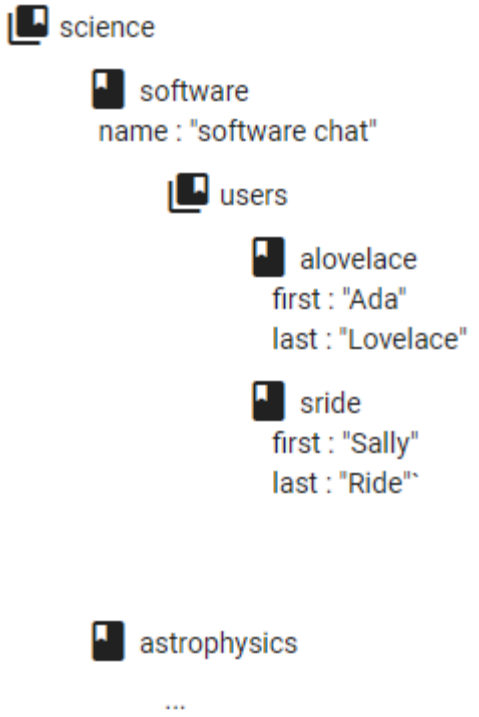
Cloud Firestore structuring data

2. Subcollections

Advantages: the size of the parent document doesn't change; full query capabilities on subcollections.

Limitations: not easy to delete subcollections, or perform compound queries across subcollections.

Use case: In the same chat app, you might create collections of users or messages within chat room documents.



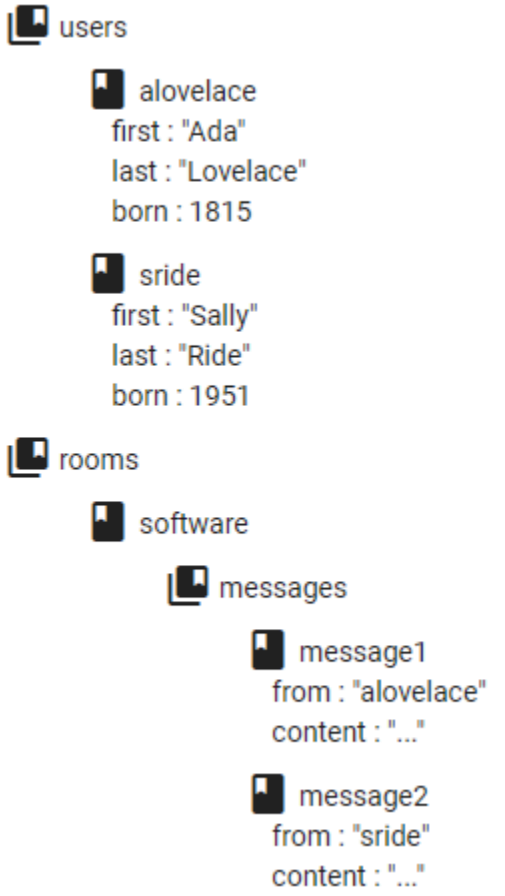
Cloud Firestore structuring data

3. Root-level collections

Advantages: most flexibility and scalability, along with powerful querying within each collection.

Limitations: getting data that is naturally hierarchical might become increasingly complex as your database grows.

Use case: In the same chat app, you might create one collection for users and another for rooms and messages.



Cloud Firestore reading and writing

- Initialize Firestore reference:

```
FirebaseFirestore db = FirebaseFirestore.getInstance();
```

- Add data to document with **add**:

```
Map<String, Object> user = new HashMap<>();
user.put("first", "Ada");
user.put("last", "Lovelace");
user.put("born", 1815);
db.collection("users")
        .add(user)
        .addOnSuccessListener(new OnSuccessListener<DocumentReference>()
{           @Override
            public void onSuccess(DocumentReference documentReference) {
                Log.d(TAG, "DocumentSnapshot added with ID: " +
                 documentReference.getId());
            }
        })
        .addOnFailureListener(new OnFailureListener() {
            @Override
            public void onFailure(@NonNull Exception e) { }}
        });
```

Cloud Firestore reading and writing

- Initialize Firestore reference:

```
FirebaseFirestore db = FirebaseFirestore.getInstance();
```

- Add data to document with **set**:

```
CollectionReference users = db.collection("users");

Map<String, Object> data1 = new HashMap<>();
data1.put("first", "Ada");
data1.put("last", "Lovelace");
data1.put("born", 1815);
users.document("AL").set(data1); OR
User user = new User(…);
users.document("AL").set(user);
```

Cloud Firestore reading and writing

- Read data from document:

```
db.collection("users")
        .get() (.whereEqualTo("born", 1999))
        .addOnCompleteListener(new OnCompleteListener<QuerySnapshot>() {
            @Override
            public void onComplete(@NonNull Task<QuerySnapshot> task) {
                if (task.isSuccessful()) {
                    for (DocumentSnapshot document : task.getResult()) {
                      Log.d(TAG,document.getId()+":"+document.getData());
                     }
                } else {
                    Log.w(TAG, "Error getting documents.",
task.getException());
                }
            }
        });
```

A listener on a collection retrieves the documents inside as a list.

Cloud Firestore queries

Example data: collection of countries in the EU.

```
CollectionReference countryRef = db.collection(“countries");
countryRef.whereEqualTo(“language", “latin");
countryRef.whereLessThan("population", 100000);

countryRef.whereGreaterThanOrEqualTo("name", “Macedonia");

countryRef.whereEqualTo(“language", “latin")
.whereLessThan("population", 1000000);
```

Cloud Firestore compound queries

Example data: collection of countries in the EU.

```
CollectionReference countryRef = db.collection("countries");
```

You may combine where(==) with range comparisons (<, >) on any field, but not range filters on two fields!

```
countryRef.whereGreaterThanOrEqualTo("language", "latin")
        .whereLessThanOrEqualTo("language", "slavic");
countryRef.whereEqualTo("language", "slavic")
        .whereGreaterThan("population", 1000000);
```

```
countryRef.whereGreaterThanOrEqualTo("language",
"slavic").whereGreaterThan("population", 100000);
```

Pricing plans

	Spark	~~Flame~~	Blaze
	Free (hobby)	$25/month (growing apps)	Pay as you go
Free products: auth, analytics, app indexing, dyn links, invites, remote config, FCM, crash reporting	yes	yes	Yes
Realtime database	100 con 1GB stored 10GB/m/dld	100K 2.5GB 20GB/m	100K $5/GB $1/GB/m
Storage	5GB 1GB/day 20K uplds/day 50K dlds/day	50GB 50GB/day 100K/day 250K/day	$0.026/GB $0.12/GB $0.05/10K $0.004/10K

Firebase BaaS questions before using it

- Is there any guaranteed service level? Have there been any outages?
- Is the service really scalable? Some services say they are scalable just because they run on Google/Amazon cloud services.
- Can the service deal with surges in use, like e.g. when a service is launched?
- How easy is it to actually see your data for reporting, admin and backup purposes? If not, you will need to write extra software to do these things.
- How much will it cost in the long term if you really get many users?
- How long is the service likely to be around? How is it funded?