13-Firestore

https://firebase.google.com/

Usiamo PAAS (google) → Cloud Firestore

- Paghi per cosa usi (in questo caso se piccolo, non paghi)

Real-Time e Observability → vedo cambiamenti.

- Database disponibile facilmente in cloud
- Db a cui un'applicazione si può iscrivere in modo tale da aggiornarsi in caso di cambiamenti del db

Usiamo NoSQL db:

- Storano un dato valore ad una data chiave
  - Ogni chiave corrisponde ad un oggetto JSON
- Integrato con un web service generale
  - Area di storage dove puoi prendere risorse per la tua applicazione
- Disegnato per condividere i dati tra diverse machines
  - In modo tale da scalare numero di dati e numero di utenti concorrenti
- Questa scalabilità causa un problema di consistenza:
  - Secondo il teorema CAP
    - **Consistency → è eventuale in questo caso →** se si da abbastanza tempo ok, altrimenti si potrebbero vedere de valori intermedi
      - Se ci fosse una totale consistency:
        - Ha a che fare con isolation (di ACID)
        - Prima di una transazione ho un determinato valore valido
        - Dopo una transazione ho un altro valore valido e quello che è stato sottratto a me, è stato aggiunto ad altri (es: in pagamento)
        - I valori intermedi, non vengono visti
          - Altri utenti, vedono solo i valori iniziali o finali; non vedono quello che è successo nel mezzo.
        - NON CI SONO TRANSAZIONI CHE LAVORANO SU DATI INTERMEDI (es: sottratto dal mio account e non ancora aggiunto all'altro)
    - Availability →
    - Partition tollerance → obbligatoria

Ogni db distribuito incorre nel CAP Theory

Tipi di NoSQL DBMS:

Ci sono diverse tecnologie:

- **Key/Value stores:**
  - Hash table distribuita
  - Ogni dato ha una chiave univoca
    - Stringa unica
  - Ad ogni chiave → associato un valore
  - Valore non compreso dal server
- **Document stores**:
  - Dati hanno una chiave
  - Dato sono un documento strutturato, spesso in formato JSON
  - Es: mongoDB
- **Column data stores**:
  - Singoli record rappresentati in colonne di dato (non righe)
  - Es: Cassandra
- **Graph stores**:
  - Dati organizzati in grafi, usando nodi e links

FIRESTORE:

- Schema-less document store
    - **Dati sono organizzati in** collezioni e documenti
        - **Collezioni**:
            - Hanno un nome (es: Tasks, Teams, Users)
            - Raggruppano un insieme di dati in una maniera consistente
            - Può contenere 0+ documenti
                - Ogni documento ha una chiave unica
                    - Generata da noi se il nostro dominio ha un modo per crearla o dal sistema che garantisce assenza di collisioni
        - **Documenti**: struttura dati JSON
            - Ha chiavi testuali
            - Ha valori booleani, numerici, testuali, temporali, valori geografici
            - Valori possono anche essere liste, sotto-oggetti…
            - Un documento è limitato a dimensione di 1MB
            - ID del documento può essere generato dal sistema stesso oppure da noi, se siamo in grado di generarlo in modo tale che faccia lui (es: matricola)

EXPRESSIVE QUERYING:

Quando si effettuano le query, si possono richiedere documenti singoli specificando l'id o selezionare quelli che si vuole specificando determinati requisiti.

Se accedo al documento specificando la chiave → costo piccolo.

REALTIME UPDATES:

Firestore setta un **canale di sincronizzazione dati tra il db e ogni device** connesso

- Quando cambiano i dati → notifica a tutti i dispositivi connessi per sincronizzarsi

Quando c'è una modifica a livello locale, questa viene cachata sul mio device e poi propagata

NOTA SU **DOCUMENTI**:

- A document consists of named pieces of data

```
{
  "title" : "Mobile Application Development",
  "groups" : {
    "g1" : [ "name1", "name2", "name3", "name4" ],
    "g2" : [ "name5", "name6", "name7", "name8" ]
  },
  "tableOfContents" : ["Kotlin", "Android", "Firestore"
        "User Centered Design", "Hybrid Apps" ],
  "year" : 2024
}
```

→studenti modellati in gruppi all'interno del corso, in questo modo si ottengono direttamente i gruppi senza dover iterare sugli utenti vedendo il gruppo di appartenenza.

COLLEZIONI:

Una collezione può contenere vari documenti eterogenei.

- Un documento può contenere una sub-collection

Ogni documento è univocamente identificato dalla sua posizione nel DB → possiamo vederlo come il path nel filesystem.

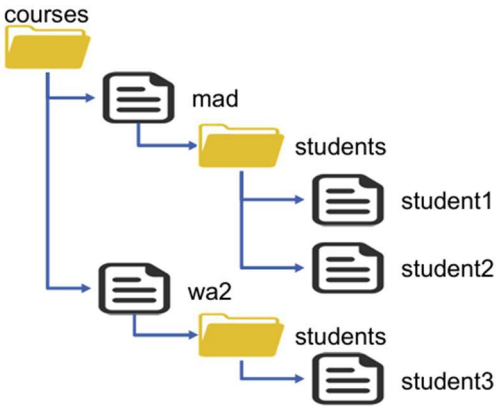
```
val mad =db.collection("courses").document("mad")
val alwaysMad = db.document("courses/mad")
```

Un dato documeto può contenere molti sub-items:

- È possibile storarli dentro il documento stesso anche se spesso non è un ottima idea

Documenti singoli hanno delle limitazioni:

- Max 1Mbye
- Max 20000 proprietà
- Nax 20 livelli di depp nesting



Una **sub-collection** è una collection assciata ad uno specifico documento (es: students a mad)

- Cosa salvo nello student di mad non ha a che fare con cosa storo nella student di wa2, sono indipendenti
  - References to items in sub-collections can be created easily
    - ```
      val student1 = db
          .collection("courses")
          .document("mad")
          .collection("students")
          .document("student1")
      ```
  - Sub-collections can be nested up to 100 levels deep

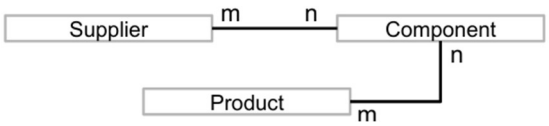
Comparazione tra strutture dati:

- Nested data in documents: ok quando abbiamo un numero limitato di dati
  - Mantenere dati insieme per veitare join pesanti
- Sub-collections: mantenere il documento padre picollo
  - Problemi in eliminazione
    - Se elimino il padre, la corrispondente sub-collection, non viene eliminata
- Root-level collections: molto flessibile e scalabile
  - Difficile gestire relazioni tra item appartenenti a collezioni sorelle
      ```

Modellare le relazioni:

Se un dato è collegato ad altri, sono disponibili 2 approcci per modellare sono possibili:

- Referenziare dati in altre collezioni
- Denormalizzare i dati



Se devo fare una **m-n** → ho informazioni duplicate

- Problema quando rimuovo le informazioni
    - Rischio inconsistenza nel caso in cui elimino da un lato e non dall'altro

...Referenziare dati in altri componenti:

- Se ho la reference di un documeto è come se avessi l'URL
    - In quanto tale, può essere memorizzato nel documento di origine per consentire il recupero del documento di destinazione
    - Facile trovare i dati ma devo essere a conoscenza che se un documento viene eliminato → ho un link pendente → cascading non è supportata
- Firestore non sopporta join tra colonne perché non le ha.
    - Devono essere emesse due richieste: una per ottenere l'insieme dei documenti di relazione, la seconda per ottenere gli elementi correlati
        - La prima richiesta implica una ricerca (è necessario costruire un indice su ogni campo di riferimento)

...Denormalizzare:

Data-duplication:

- Storo in una parte la lista dei valori, dall'altra la lista delle chiavi
- Rischio conflitti
    - Es: aggiorno solo una parte

Dunque:

- Prodotti, componenti e fornitori possono essere stoccati in raccolte separate
    - Ogni fornitore può avere un campo array che elenca l'insieme di componenti che produce
    - Ogni prodotto ha un campo array con l'insieme dei componenti di cui è composto
    - Ogni componente ha due campi di matrice: uno elenca l'insieme dei possibili fornitori, l'altro elenca l'elenco dell'insieme dei prodotti che lo utilizzano

THE FIREBASE CONSOLE:

Android studio → tools→Firebase.

```
val db = Firebase.firestore
```

Tutte le operazioni che coinvologono un db, iniziano con un oggetto di classe FirebaseFirestore

Il db è automaticamente derivata dal file google-serviceses.json (bisogna aggiungerlo al git).

Quando si ha un db, si può accedere ai dati:

- **Collection**: ci permettere di accedere
  - Ad una collezione
  - Ad un documento (mad)
- **Document**: permette di accedere ad un dato documento (mad)

```
val coll1 = db.collection("courses")
val doc1 = db.collection("courses")
           .document("mad")
val doc2 = db.document("courses/mad")
```

In kotlin i **dati** possono essere **salvati** in mappe o in classi di dato con un costruttore di default.

```
val course = HashMap<String, Any>()
course["title"] = "Mobile Application Development"
course["groups"] = hashMapOf<String, Any>(
    "g1" to listOf("name1", "name2", "name3", "name4"),
    "g2" to listOf("name5", "name6", "name7", "name8")
)
course["tableOfContents"] = listOf(
   "Kotlin", "Android", "Firestore",
   "User Centered Design", "Hybrid Apps")
course["year"] = 2024
```

```
data class Course (
 var title: String = "",
 var groups: Map<String, List<String>> = mapOf()

  @set:PropertyName("tableOfContents")
  @get:PropertyName("tableOfContents")
 var toc: List<String> = emptyList(),
 var year: Int = 0
)
```

Tutte le oper**azioni sono asincrone** → definiamo le operazioni ora ma saranno eseguite in un altro momento:

- Posso aggiungere dei listener

```
db
  .collection("courses")
  .document("mad2024")
  .set( modelData() )    //the chosen representation
  .addOnSuccessListener {
    Toast
      .makeText(this, "Data saved", Toast.LENGTH_SHORT)
      .show()
  }
  .addOnFailureListener{
    Toast
      .makeText(this, "Error", Toast.LENGTH_LONG)
      .show()
  }
```

→ **rimpiazza i vecchi dati con i nuovi**

→ **connetto dei listener**, in questo modo, quando l'operazione termina, viene invocato uno dei due listener e posso usare l'informazione ritornata dalla per svolgere ciò che voglio (Toast).

N**ota**: se mi serve sospendere l'applicazioni finchè non ottengo il risultato, devo usare awayt che causerà un'eccezione in caso di failure e ritornerà il contenuto di success se tutto ok.

L'operazione sarà suspending nel model mentre il viewModel non la vede in qesuto modo e l'unica cosa che fa è usare l'invocazione per chiamare la coroutines (Se la vita del viewModel non basta al tempo necessario per svolgere l'operazione, operazione non parte). Non bisogno preoccuparsi più di tanto in quanto memorizza ed esegue quando può.

Leggere da db:

```
db
  .collection("courses")
  .document("mad2023")
  .get()
  .addOnSuccessListener {
     res ->
       val course =
              res.toObject(Course::class.java)
       //use it as needed
  }
  .addOnFailureListener {
    Toast
      .makeText(this, "Error", Toast.LENGTH_LONG)
      .show()
  }
```

Se c'è una view connessa → refresh e aggiorna dati. **2** parti:

- **Localizzare queste proprietà nella classe indicata come target**
- **Tiping → CONVERSIONE**
  - Il modello usato da Firebase è ispirato da jvascript che non distingue i numeri per tipo mentre kotlin sì → toObject prova a mappare correttttamente:
    - Numero → double/long
    - Strings → strings
    - Dates → Timestamp
    - Boolean → Boolean
    - Json objects → Map<String, Any>
    - Josn lists → List<Any>

Listening to real time updates:

- Get mi fornisce il risultato attuale ma non aggiornamenti
- Firestore fornisce continui aggiornamenti usando una snapshot listener
  - Accetta una lambda con 2 parametri
    - **Snapshot**
      - Se !null →posso usare i dati
    - **Errore**
      - Se !null → c'è stato un errore
      - Se viene ritornato un errore, non ci saranno più return

```
val docRef = db.collection("courses").document("mad2023")
val reg = docRef.addSnapshotListener { snapshot, e ->
   if (e != null) {
      Log.w(TAG, "Listen failed.", e)
      return@addSnapshotListener
   }
   if (snapshot != null && snapshot.exists()) {
      Log.d(TAG, "Current data: ${snapshot.data}")
   } else { Log.d(TAG, "Current data: null") }
}
```

  - Quando otteniamo una nuova snapshot, prendiamo il valore e lo usiamo per aggiornare i dati
  - Se errore, usaimo questo per informare l'utente dell'errore
    - Quando un error è ritornato, non potranno essere aggiornati i dati

Si crea una connessione con il db, che resta viva finchè non la cancelliamo. Dunque, è importante che l'oggetto in cui salvo il db, venga rimossa quando si vuole chiudere la connessione:

- Si usa una **callbackFlow**<T>
  - Flow che inizia ad esistere nel momento in vui qualcuno si iscrive e riesegue se ha altre subscription.
    - Behaviour regolato dalla lambda function a lui collegata
      - Invoca **firestore**
        - booksRef
        - orderBy
      - **addSnapshotListener**
        - se ricevi qualcosa di valido → trasforma
          - salva valore
          - wrappa in success
        - se ricevi errore → wappa in Failure
      - **trySend**: ogni volta che c'è un nuovo valore, propaga il valore a chiunque stia collezionando il flow
    - se la coroutine sarò sconnessa o la connession con db chiusa, riceveremo una notifica
      - inseriamo nella lmbda: **awaytClose**:
        - se c'è una cancellazione o il subscriber si sconnette → elimina questa snapshot

```
fun getBooksFromFirestore(booksRef: CollectionReference) = callbackFlow {
    val snapshotListener = booksRef.orderBy(TITLE).addSnapshotListener { snapshot, e ->
        val booksResponse = if (snapshot != null) {
            val books = snapshot.toObjects(Book::class.java)
            Success(books)
        } else {
            Failure(e)
        }
        trySend(booksResponse)
    }
    awaitClose {
        snapshotListener.remove()
    }
}
```

Si **possono sentire documenti specifici o collezioni**:

- Collezioni: ottengo un nuovo blocco ogni volta che avviene una modifica
  - Blocco contenente l'intera collezione
    - Se piccola → ok
    - Se grossa → sconveniente
- Si possono inserire delle restrizioni:
  - `whereEqualsTo(field, value)`
  - `whereGreaterThan(field, value)`
  - `whereGreaterThanOrEqualTo(field, value)`
  - `whereLessThan(field, value)`
  - `whereLessThanOrEqualTo(field, value)`
  - `whereIn(field, valueList)`
  - `whereArrayContains(field, value)`
  - `whereArrayContainsAny(field, valueList)`

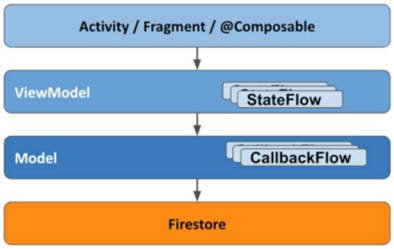
- Non si possono fare range filter su diversi campi
- Si possono fare query per equality, non per differenza
- Si possono effettuare OR tra massimo 10 uguaglianze

I dati ottenuti possono essere:

- **Ordinati** `.orderBy(fieldname)`
- **Limitati** `.limit(size)`
  - Partendo da una data posizione `.startAt(position)` and `.endAt(position)`
  - Partendo dopo una data posizione `.startAfter(position)` and `.endBefore(position)`

Firestore collegato al model:

- Tipicamente il model espone un insieme di metodi che ritornano un callbackFlow a cui iscriversi.
  - Questi flow sono visti nel viewModel come sono oppure come StateFlow e sono visti dai @Composable/altro usando CollectAsState.



FireBase supporta la **persistenza dei dati quando si è offline**: se si prova a svolgere un'operazione quando non c'è connessione ad internet → connessione ritorna fail ma è cachata e non eliminata. Successivamente, quando l'applicazione torna online, aggiorna dati da cache a online.
Se non si vuole questo comportamento, bisogna customizzare (prima di instanziare il db) le impostazioni del builder in modo tale da disabilitare qualche comportamento, settare la dimensione della cache o altro.

INDEXING DATA: Firebase console permette di creare indici in modo tale da incrementare le performance delle query.

Il Database può essere acceduto in maniera concorrente da tutti gli utenti che hanno scaricato l'applicazione:

- Se 2 utenti modificano contemporanemante un dato, db agisce con FCFS
  - Se l'app necessita di aggiornare 2 tabelle tab1.thenTab2
  - Ed entambi stanno facendo questa operazioni
    - Le operazioni richisano di sovrascrirsi e di perdere le informazioni di uno dei due
- Bisogna gestire a livello applicazione:
  - Firestore fornisce 3 operazioni sicure, da lui garantite: `increment, arrayRemove, arrayUnion`
  - Firestore provvede con **transaction**: insieme di operazioni di lettura e scrittura su uno o più documenti eseguite in maniera atomica.
    - Fai queste operazioni in sequenza e controlla che sia ok
    - Se qualcosa non va, annulla tutte le operazioni fatte in quella transazione
    - Internamente ogni oggetto contiene all'interno un **version number** che si incrementa ogni volta che un'azione di aggiornamento eseguita sull'oggetto
    - Quando devo effettuare un'operazione in una transazione, l'oggetto che accedo, è prima ispezionato prendneo il version number, poi eseguo operazione, controllo che version number non sia cambiato e aggiorno il valore
      - Se version number è cambiato, fail
    - Una transazione può essere esguita solo online, non è cachable
    - Devo ordinare le mie operazioni in modo chiaro
    - la transazione ritorna un valore:
      - se failure → undo di tutto

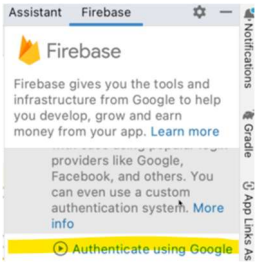
```
val sfDocRef = db.collection("cities").document("SF")

db.runTransaction { transaction ->
    val snapshot = transaction.get(sfDocRef)
    val newPop = snapshot.getDouble("population")!! + 1
    if (newPop <= 1000000) {
        transaction.update(sfDocRef, "population", newPop)
        newPop
    } else {
        throw FirebaseFirestoreException("Too many people",
                FirebaseFirestoreException.Code.ABORTED)
    }
}.addOnSuccessListener { result ->
    Log.d(TAG, "Transaction success: $result")
}.addOnFailureListener { e ->
    Log.w(TAG, "Transaction failure.", e)
}
```

→ fetcho documento
→ prendo population, dichiarando che non sarà null, e aggiungo 1
→ se nuovo valore <1mln → aggiorno

→ altrtimenti→ eccezione

Security: Permission



In alcune situazione serve capire chi sta accedendo ai nostri dati

- **autentication support**
  - email/password
  - phone
  - anonymous
    - legato al dispositivo e non all'utente
    - se un utente ha 2 cellulari, avrà due account diversi
  - Google
  - Facebook
  - etc

Di default le operazioni di lettura e scrittura necessitano che l'utente sia autenticato

- è possibile avere diverse regole per diverse collezioni/oggetti

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    // This rule allows anyone on the internet to view, edit, and delete
    // all data in your Firestore database. //
    // Make sure to write security rules for your app before that time, or
    // else your app will lose access to your Firestore database
    match /{document=**} {
      allow read, write: if request.time < timestamp.date(2020, 5, 18);
    }
  }
}
```

Per ogni collezione, si possono definire delle regole:

```
service cloud.firestore {
  match /databases/{database}/documents {

    match /courses/{course} {
      allow read: if <condition>;
      allow write: if <condition>;
    }
  }
}
```

Si possono inserire restrizioni all'intera collezione o al singolo documento

```
service cloud.firestore {
  match /databases/{database}/documents {
    match /courses/{course} {
      allow get: if <condition>;
      allow list: if <condition>;
    }
    match /courses/{course} {
      allow create: if <condition>;
      allow update: if <condition>;
      allow delete: if <condition>;
    }
  }
}
```

- Access always
  - ```
    match /collection/{id} {
      allow read, write: if true;
    }
    ```
- Authenticated users only
  - ```
    match /collection/{id} {
      allow write: if request.auth.uid != null;
    }
    ```
- Users can only see their own data
  - ```
    match /users/{userId} {
      allow read, update, delete:
      if request.auth.uid == userId;
      allow create: if request.auth.uid !=null;
    }
    ```
    ```