



# Curso de Cloud Firestore para Android

---

Santiago Carrillo



Autenticación  
(Anónima)



Cloud  
Firestore



Autenticación  
(Anónima)

10:32

Realtime Trader

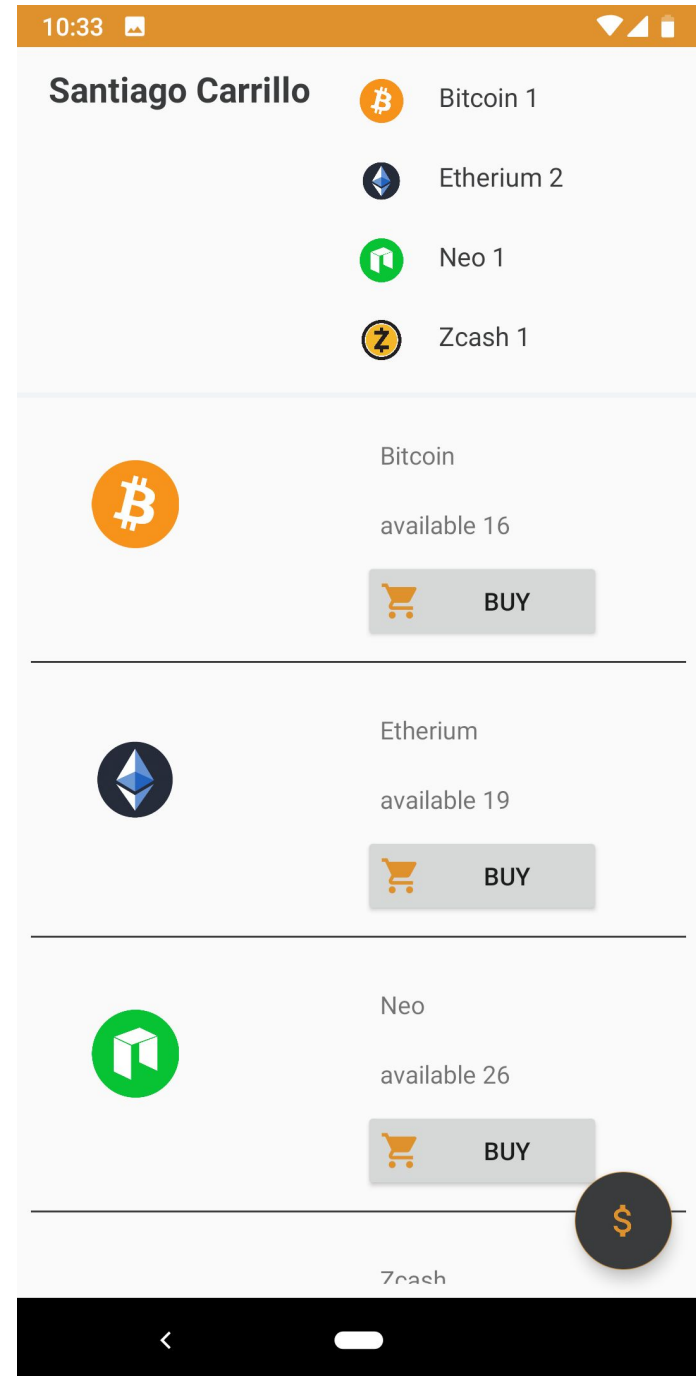
\$

username

START



Cloud  
Firestore



---

# Apps Rápidas sin preocuparse de la infraestructura



Firebase



---

# Apps Mejores





### **Machine Learning Kit**

Machine Learning para apps móviles



### **Cloud Firestore**

Almacena y sincroniza datos a escala global



### **Cloud Function**

Ejecute código en el backend sin gestionar servidores



### **Autenticación**

Autenticación simple y segura



### **Hosting**

Ofrece recursos Web con velocidad y seguridad



### **Cloud Storage**

Almacena y accede archivos en la escala de Google



### **Realtime Database**

Almacena y sincroniza datos en milisegundos

---

# Apps de Mayor Calidad







### **Crashlytics**

Prioriza y repara fallos con reportes de errores en tiempo real



### **Monitor de Desempeño**

Recibe datos del desempeño de tus Apps

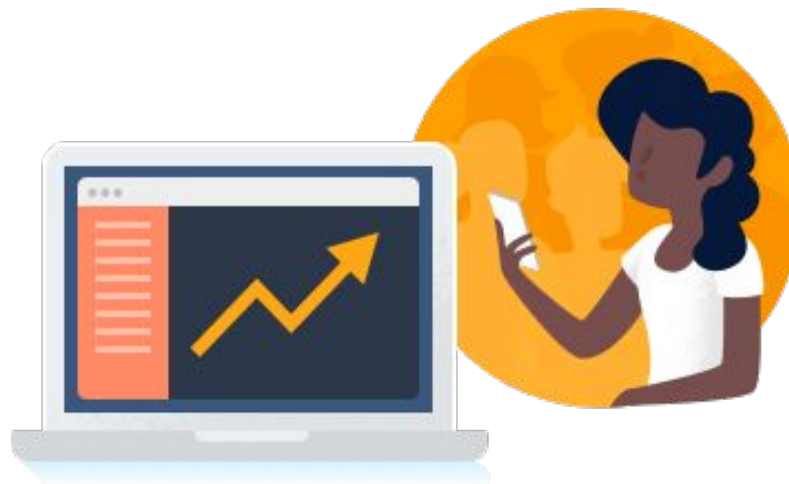


### **Test Lab**

Prueba tus aplicaciones en dispositivos de Google

---

# Crece tu Negocio





### **In-App Messaging**

Conectate con tus usuarios por medio de mensajes de contexto



### **Cloud Messaging**

Envía mensajes dirigidos y notificaciones.



### **Google Analytics**

Obtén analítica de tu app gratis e ilimitada



### **Configuración Remota**

Modifica tu App sin necesidad de despliegue de nueva versiones



### **Predicciones**

Segmentación inteligente de usuarios basada en comportamiento predictivo.



### **Links Dinámicos**

Maneja el crecimiento usando deep links



### **A/B Testing**

Optimiza la experiencia de tu App a través de experimentación.



### **App Indexing**

Maneja el tráfico de búsqueda de tu App móvil

---

# Creación y configuración de tu cuenta



---

# Proyecto base Android e integración con Firebase



---

# Autenticación Anónima

Documentación Oficial

---

# Autenticación Anónima Firebase



LoginActivity

`signInAnonymously()`



Auth



---

# Entendiendo Cloud Firestore



---

# Conociendo el modelo de datos de Cloud Firestore

Documentación Oficial

# Modelo de Datos

The screenshot displays a database interface with a hierarchical structure. At the top, a database named 'test-90da8' is shown. Below it, a collection named 'cryptos' is selected, showing a list of documents: 'users', 'bitcoin', 'ethereum', 'neo', and 'zcash'. The 'bitcoin' document is expanded, revealing its fields: 'available' (29), 'imageUrl' (a URL to a Bitcoin logo), and 'name' ('Bitcoin').

```
test-90da8
```

- + Add collection
- cryptos
  - + Add document
  - bitcoin
    - ethereum
    - neo
    - zcash
- users

```
bitcoin
```

- + Add collection
- + Add field
- available: 29
- imageUrl: "https://en.bitcoin.it/w/images/en/2/29/BC\_Logo.png"
- name: "Bitcoin"

# Documentos

## Sencillos

bitcoin

+ Add collection

+ Add field

available: 29

imageUrl: "https://en.bitcoin.it/w/images/en/2/29/BC\_Logo\_.png"

name: "Bitcoin"

## Complejos

Santiago Carrillo

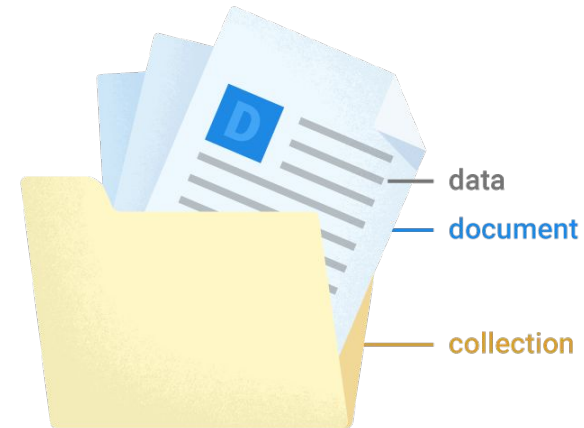
+ Add collection



+ Add field

cryptosList: [{available: 2, documentId...}]

username: "Santiago Carrillo"

# Colecciones



 cryptos	 bitcoin
<a href="#">+ Add document</a>	<a href="#">+ Add collection</a>
bitcoin >	<a href="#">+ Add field</a>
etherium	available: 29
neo	imageUrl: "https://en.bitcoin.it/w/images/en/2/29/BC_Logo_.png"
zcash	name: "Bitcoin"

---

# Referencias

```
val bitcoinDocumentRef =  
db.collection("cryptos").document("bitcoin")
```

```
val bitcoinDocumentRef =  
db.document("cryptos/bitcoin")
```

```
val cryptosCollectionRef = db.collection("cryptos")
```

# Subcolecciones

☰ Santiago Carrillo

+ Add collection

+ Add field

▼ cryptosList

▼ 0

available: 2

documentId: "bitcoin"

imageUrl: "https://en.bitcoin.it/w/images/en/2/29/BC\_Logo\_.png"

name: "Bitcoin"

▶ 1 {available: 1, documentId:...}

▶ 2 {available: 2, documentId:...}

▶ 3 {available: 0, documentId:...}

username: "Santiago Carrillo"

# Tipos de Datos

**Add a document**

Parent path  
`/users`

Document ID ?

Auto-ID

Field	Type	Value
<input type="text"/>	<div><div>string</div><div>number</div><div>boolean</div><div>map</div><div>array</div><div>null</div><div>timestamp</div><div>geopoint</div><div>reference</div></div>	<input type="text"/>

+ Add field



---

# Índices

Si no hay índices cada consulta realizará una búsqueda ineficiente de dato por dato y no permitirá que esta escale.

## INDEX

E00127  
E01234  
E03033  
E04242  
E10001  
E10297  
E16398  
E21437  
E27002  
E41298  
E43128  
E63535

## TABLE

Tyler	Bennett	E10297
John	Rappl	E21437
George	Woltman	E00127
Adam	Smith	E63535
David	McClellan	E04242
Rich	Holcomb	E01234
Nathan	Adams	E41298
Richard	Potter	E43128
David	Motsinger	E27002
Tim	Sampair	E03033
Kim	Arlich	E10001
Timothy	Grove	E16398

---

# Tipos de Índices

## *Campo único*

almacena el orden de todos los documentos en una colección de un sólo campo.

## *Compuestos*

almacena el orden de todos los documentos en una colección que contiene *subcampos*.

---

# Modos Índices

Index mode	Description
Ascending ↑	Supports <, <=, ==, >=, and > query clauses on the field and supports sorting results in ascending order based on this field value.
Descending ↓	Supports <, <=, ==, >=, and > query clauses on the field and supports sorting results in descending order based on this field value.
Array-contains	Supports <a href="#">array_contains</a> query clauses on the field.

---

# Auto indexación

Por defecto se crean los siguientes índices:

- **Campos simples** (no colecciones) - dos índices de campo único (ascendente y descendente).
- **Diccionarios** - > dos índices campo único (ascendente y descendente) por cada subcampo.
- **Colecciones** -> Índice array-contains.

---

# Índices *campo único*

Este tipo de índice te permite realizar consultas usando:

- Comparadores `<`, `<=`, `==`, `>=`,
- Arreglos `array_contains`

```
citiesRef.where("name", "==", "Bogotá")
```

```
citiesRef.where("population", "<", 100000)
```

```
citiesRef.where("regions", "array-contains", "Amazonas")
```

---

# Índices *compuestos*

Almacenan un mapeo ordenado de todos los documentos en una colección.

```
citiesRef.where("country", "==", "Colombia").orderBy("population", "asc")
```

```
citiesRef.where("country", "==", "Colombia").where("population", "<", 3000)
```

```
citiesRef.where("country", "==", "Colombia").where("population", ">", 3000)
```

Collection	Fields indexed
cities	↑ (or ↓) country, ↑ population



---

# Creación y Gestión de Datos

---

# Estructura de Datos

## Colecciones



alovelace

name :

first : "Ada"

last : "Lovelace"

born : 1815

rooms :

0 : "Software Chat"

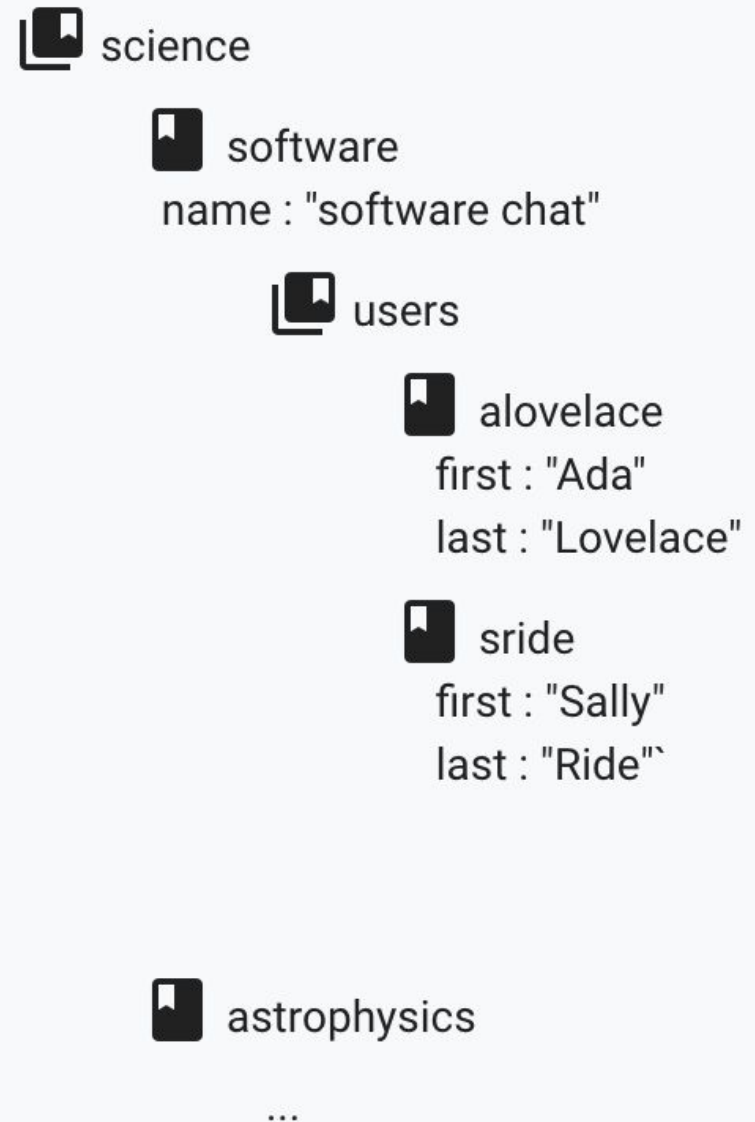
1 : "Famous Figures"

2 : "Famous SWEs"

---

# Estructura de Datos

## Subcolecciones



---

# Estructura de Datos

Colecciones a nivel de la raíz



---

# Creación de Documentos

```
val city = HashMap<String, Any>()  
city["name"] = "Bogotá"  
city["state"] = "Cundinamarca"  
city["country"] = "Colombia"
```

```
db.collection("cities").document("BOG")  
    .set(city)  
    .addOnSuccessListener {  
        Log.d(TAG, "DocumentSnapshot creado exitosamente!") }  
    .addOnFailureListener { e ->  
        Log.w(TAG, "Error escribiendo documento", e) }
```

\*\*\* Si el documento no existe será creado. Si el documento existe será sobreescrito\*\*\*

---

# Merge de Documentos

```
// Actualizar un campo, el documento "BOG" es  
creado si no existe.
```

```
val data = HashMap<String, Any>()  
data["capital"] = true
```

```
db.collection("cities").document("BOG")  
    .set(data, SetOptions.merge())
```

---

# Tipos de Datos

```
val docData = HashMap<String, Any?>()  
docData["stringExample"] = "Hello world!"  
docData["booleanExample"] = true  
docData["numberExample"] = 3.14159265  
docData["dateExample"] = Date()  
docData["listExample"] = arrayListOf(1, 2, 3)  
docData["nullExample"] = null
```

```
val nestedData = HashMap<String, Any>()  
nestedData["a"] = 5  
nestedData["b"] = true
```

```
docData["objectExample"] = nestedData
```

---

# Agregar un Documento

Nuevo Documento con ID definido:

```
db.collection("cities")  
.document("new-city-id").set(data)
```



---

# Agregar un Documento

Nuevo Documento con ID autogenerado:

```
val data = HashMap<String, Any>()  
data["name"] = "Bogotá"  
data["country"] = "Colombia"  
  
db.collection("cities")  
    .add(data)
```

---

# Actualizar un Documento

```
val bogotaRef = db.collection("cities").document("BOG")

// Actualizar el campo "isCapital" de la ciudad 'BOG'
bogotaRef
    .update("capital", true)
    .addOnSuccessListener {
        Log.d(TAG, "DocumentSnapshot successfully
updated!") }
    .addOnFailureListener {
        e -> Log.w(TAG, "Error updating document", e)
    }
```

---

# Actualizar Subdocumentos

```
// Asuma que el documento contiene:  
// { name: "Frank",  
//   favorites: { food: "Pizza", color:  
"Blue", subject: "recess" },  
//   age: 12  
// }
```

```
// Para actualizar el atributo age y favorite color:  
db.collection("users").document("frank")  
    .update(  
        "age", 13,  
        "favorites.color", "Red")
```

---

# Actualizar Listas

```
val bogotaRef =  
db.collection("cities").document("BOG")
```

```
// Agregar una nueva región al arreglo "regions".  
colombiaRef.update("regions",  
FieldValue.arrayUnion("Amazonas"))
```

```
// Remover una región del arreglo "regions".  
colombiaRef.update("regions",  
FieldValue.arrayRemove("Amazonas"))
```

---

# Implementación FirestoreService



---

# Transacciones

```
val bogDocRef = db.collection("cities").document("BOG")

db.runTransaction { transaction ->
    val snapshot = transaction.get(bogDocRef)
    val newPopulation = snapshot.getDouble("population")!! + 1
    transaction.update(bogDocRef, "population", newPopulation)
    null
}.addOnSuccessListener { Log.d(TAG, "Transaction success!") }
    .addOnFailureListener { e -> Log.w(TAG, "Transaction
failure.", e) }
```

---

# Escritura en Batch

```
// Obtener un nuevo batch de escritura
```

```
val batch = db.batch()
```

```
// Asignar el valor de 'BOG'
```

```
val bogotaRef = db.collection("cities").document("BOG")
```

```
batch.set(bogotaRef, City())
```

```
// Actualizar el atributo population de 'BOG'
```

```
val bogotaRef = db.collection("cities").document("BOG")
```

```
batch.update(bogotaRef, "population", 1000000L)
```

---

# Escritura en Batch

```
// Eliminar la ciudad 'BOG'  
val bogotaRef = db.collection("cities").document("BOG")  
batch.delete(bogotaRef)  
  
// Realizar Commit del batch  
batch.commit().addOnCompleteListener {}
```



---

# ¿Cuándo usar Transacciones o Batch?

1. Migración de datos
2. Muchas operaciones simultáneas de escritura sobre una misma colección.

---









# Lectura de Datos

---

# Lectura de Datos

Existen dos formas de lectura de datos con Firestore:

- **Directa:** Llamado a un método para obtener los datos
- **Observación:** Asignar un listener para ser notificado en modificaciones de los datos

<div><div> users</div><div></div></div>	<div><div> Santiago Carrillo</div></div>
<div><div><div><div> Add document</div><div>Andres Lopez</div><div>Camila Rojas</div><div>Santiago Carrillo &gt;</div></div></div></div>	<div><div><div><div> Add collection</div></div><div><div> Add field</div><div><div><div> cryptosList: [{available: 2, documentId...}]</div><div>username: "Santiago Carrillo"</div></div></div></div></div></div>

---

# Lectura Única de Datos

```
val docRef = db.collection("cities").document("BOG")
docRef.get()
    .addOnSuccessListener { document ->
        if (document != null) {
            val city = document.toObject(City::class.java)
        } else {
            Log.d(TAG, "No such document")
        }
    }
    .addOnFailureListener { exception ->
        Log.d(TAG, "get failed with ", exception)
    }
```

---

# Lectura de Colecciones

```
//todos los documentos en una colección
db.collection("cities")
    .get()
    .addOnSuccessListener { result ->
        for (document in result) {
            Log.d(TAG, document.id +
                " => " + document.data)
        }
    }
    .addOnFailureListener { exception ->
        Log.d(TAG, "Error getting documents: ",
exception)
    }
```

---

# Lectura de Colecciones

```
//varios documentos que cumplan criterio
db.collection("cities")
    .whereEqualTo("capital", true)
    .get()
    .addOnSuccessListener { documents ->
        for (document in documents) {
            Log.d(TAG, document.id + " => "
                + document.data)
        }
    }
    .addOnFailureListener { exception ->
        Log.w(TAG, "Error getting documents: ", exception)
    }
```

---

# Suscripción a Actualizaciones en Realtime

//Observador a cambios en un documento específico

```
val docRef = db.collection("cities").document("BOG")
docRef.addSnapshotListener(EventListener
<DocumentSnapshot> { snapshot, e ->
    if (e != null) {
        Log.w(TAG, "Listen failed.", e)
        return@EventListener
    }

    if (snapshot != null && snapshot.exists()) {
        Log.d(TAG, "Current data: " + snapshot.data)
    } else {
        Log.d(TAG, "Current data: null")
    }
})
```



---

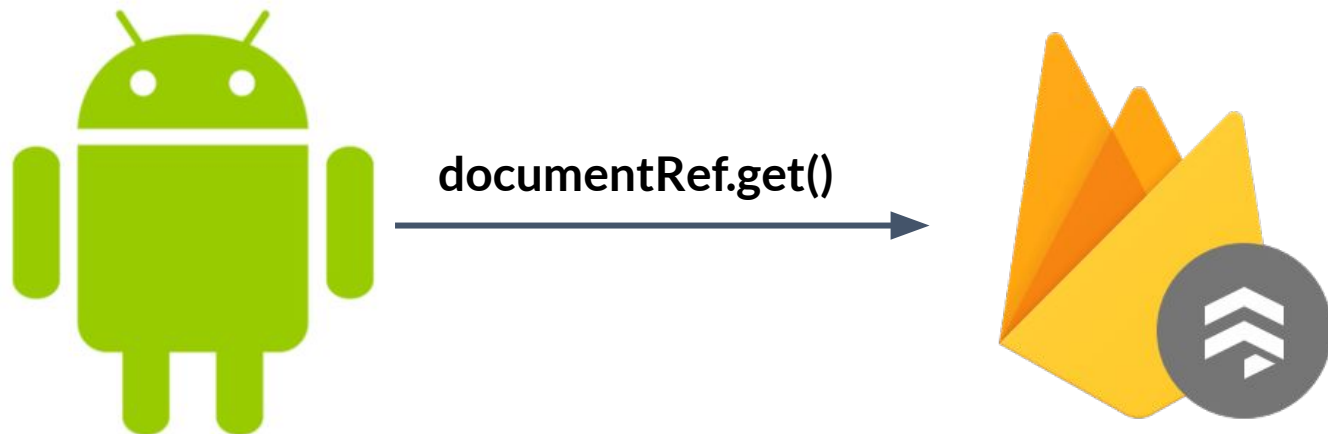
# Suscripción a Actualizaciones en Realtime

```
//Observador a eventos que cumplen cierto criterio
```

```
db.collection("cities")  
    .whereEqualTo("state", "CA")  
    .addSnapshotListener(  
        EventListener<QuerySnapshot> {})
```

---

# Implementación Lectura Datos FirestoreService



---

# Consultas Sencillas

```
// Crear referencia a las ciudades de la colección  
val citiesRef = db.collection("cities")
```

```
// Crear referencia a query sobre la colección  
val query = citiesRef.whereEqualTo("name", "Bogotá")
```

```
// Obetener todas las ciudades capitales  
val capitalCities =  
db.collection("cities").whereEqualTo("capital", true)
```

---

# Consultas Sencillas

```
// Filtros
```

```
citiesRef.whereEqualTo("name", "Bogotá")
```

```
citiesRef.whereLessThan("population", 100000)
```

```
// Operador array_contains
```

```
val citiesRef = db.collection("cities")
```

```
citiesRef.whereArrayContains("regions", "Amazonas")
```

---

# Consultas Compuestas

```
citiesRef.whereEqualTo("country", "Colombia")  
  .whereEqualTo("name", "Bogotá")  
citiesRef.whereEqualTo("country",  
"Colombia").whereLessThan("population", 1000000)
```

```
//Rango de filtros válido (un sólo campo)  
citiesRef.whereGreaterThanOrEqualTo("country", "Colombia")  
  .whereLessThanOrEqualTo("country", "Suecia")  
citiesRef.whereEqualTo("country", "Colombia")  
  .whereGreaterThan("population", 1000000)
```



---

# Consultas Compuestas

```
citiesRef.whereEqualTo("country", "Colombia")  
.whereEqualTo("name", "Bogotá")  
citiesRef.whereEqualTo("country", "Colombia")  
.whereLessThan("population", 1000000)
```



//Rango de filtros inválido (en campos diferentes)

```
citiesRef.whereGreaterThanOrEqualTo("country", "Colombia")  
.whereGreaterThan("population", 100000)
```

---

# Orden y Límites de Datos

```
//Consultar los 3 primeros registros ordenados por nombre  
citiesRef.orderBy("name").limit(3)
```

```
//Ordenar de manera descendente  
citiesRef.orderBy("name", Query.Direction.DESENDING).limit(3)
```

```
//Ordenar por multiples campos  
citiesRef.orderBy("state").orderBy("population", Query.Direction.DESENDING)
```

```
//Combinación con la cláusula where()  
citiesRef.whereGreaterThan("population",  
100000).orderBy("population").limit(2)
```

```
//Rango del filtro y orderBy en el mismo campo  
citiesRef.whereGreaterThan("population", 100000).orderBy("population")
```

```
//Rango del filtro y orderBy en diferente campo  
citiesRef.whereGreaterThan("population", 100000).orderBy("country")
```



---

# Paginación y Cursores de Datos

Use los métodos *startAt()* o *startAfter()* para definir el punto de partida de la consulta

```
// Obtener todas las ciudades con population >=
1,000,000, ordenados por population,
db.collection("cities")
    .orderBy("population")
    .startAt(1000000)
```

```
// Obtener todas las ciudades con population <=
1,000,000, ordenados por population,
db.collection("cities")
    .orderBy("population")
    .endAt(1000000)
```



---

# Paginación y Cursores de Datos

Use el snapshot del documento para definir el cursor

```
// Obtener data de "San Francisco"
db.collection("cities").document("SF")
    .get()
    .addOnSuccessListener { documentSnapshot ->
// Obtener todas las ciudades con un población mayor
que la de San Francisco.
        val biggerThanSf =
db.collection("cities")
            .orderBy("population")
            .startAt(documentSnapshot)
    }
```

---

# Acceso Offline a Datos

Configure la persistencia offline

```
val settings =  
    FirebaseFirestoreSettings.Builder()  
        .setPersistenceEnabled(true)  
        .build()  
db.firestoreSettings = settings
```

---

# Acceso Offline a Datos

Monitoreo de Datos Offline Agregando  
MetadataChanges.INCLUDE

```
db.collection("cities")  
  .whereEqualTo("name", "Bogota")  
  
  .addSnapshotListener(MetadataChanges.INCLUDE,  
    EventListener<QuerySnapshot>
```

---

# Prueba Acceso Offline

```
db.disableNetwork().addOnCompleteListener {  
    // Do offline things  
    // ...  
}
```

```
db.enableNetwork().addOnCompleteListener {  
    // Do online things  
    // ...  
}
```

---

# Gestión de Índices

Para garantizar un desempeño óptimo Firestore requiere índices para cada consulta. Los índices requeridos para la mayoría de consultas son creados de manera automática

---

# Gestión de Índices

1. Ingrese a la sección **Database** de la consola de Firebase.
2. Ingrese al tab **Indexes** tab y de clic en **Add Index**.
3. Ingrese el nombre del nombre de la colección y el conjunto de campos que quiere ordenar y ser indexados
4. Clic **Create**.

## Add composite index



Composite indexes are required for queries that include specific values and a range or order.



### Recommended

Instead of defining a composite index manually, run your query in your app code to get a link for generating the required index.

[LEARN MORE](#)

In collection(s)...

collection

Order documents by...

1	field1	<input checked="" type="radio"/> ascending	<input type="radio"/> descending
2	field2	<input type="radio"/> ascending	<input checked="" type="radio"/> descending

[ADD FIELD](#)

CANCEL

CREATE INDEX

Creation time depends on the size of your data

---

# Reglas y Seguridad



---

# Escritura de Reglas

Todas las reglas se crean usando la declaración *match* que identifica los documentos en la base de datos y permite acceso o no a estos usando la expresión *allow*

```
service cloud.firestore {  
  match /databases/{database}/documents {  
    match /<some_path>/ {  
      allow read, write: if <some_condition>;  
    }  
  }  
}
```

// Permitir acceso de lectura/escritura read/write en todos los documentos a cualquier usuario autenticado

```
service cloud.firestore {  
  match /databases/{database}/documents {  
    match /{document=**} {  
      allow read, write: if request.auth.uid != null;  
    }  
  }  
}
```

```
// Denegar acceso de lectura/escritura a todos los  
usuarios en cualquier condición
```

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

```
// Permitir acceso de lectura/escritura a todos los  
    usuarios en cualquier condición
```

```
// Alerta: **NUNCA** usar esta regla en producción.
```

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

---

# Despliegue de Reglas

Antes de poder utilizar Firestore es necesario realizar el deploy de las reglas de seguridad. Las reglas se pueden configurar de dos formas:

- Firebase CLI
- Consola Firebase

---

# Firebase CLI

```
// Configurar Firestore en tu proyecto, esto crea un  
archivo.rules
```

```
firebase init firestore
```


```
// Editar el archivo generado .rules con las reglas  
deseadas
```

```
// Deploy del archivo .rules
```

```
firebase deploy --only firestore:rules
```

# Consola Firestore

Database

 Cloud Firestore ▾

Data

Rules

Indexes

Usage

★ Jan 29, 2019 • 10:47 PM

○ Jan 29, 2019 • 5:49 PM

1

2

3

4

5

6

7

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

---

# Estructurando Reglas de Seguridad

Las reglas siempre comienzan con la siguiente declaración:

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



---

# Lectura y Escritura

```
service cloud.firestore {  
  match /databases/{database}/documents {  
  
    // Match cualquier documento en la colección 'cities'  
    match /cities/{city} {  
      allow read: if <condition>;  
      allow write: if <condition>;  
    }  
  }  
}
```

---

# Operaciones Granulares

```
service cloud.firestore {  
  match /databases/{database}/documents {  
    // Regla de lectura dividida en reglas de get y list  
    match /cities/{city} {  
  
      // Aplica a peticiones de lectura de un solo documento  
      allow get: if <condition>;  
  
      // Aplica a peticiones de lectura de consultas y colecciones  
      allow list: if <condition>;  
    }  
  }  
}
```

---

# Operaciones Granulares

```
// Una regla de escritura puede estar dividida en create,  
update, y delete
```

```
match /cities/{city} {
```

```
    // Aplica a escritura de documentos no existentes
```

```
    allow create: if <condition>;
```

```
    // Aplica a escritura de documentos existentes
```

```
    allow update: if <condition>;
```

```
    // Aplica a operaciones de delete
```

```
    allow delete: if <condition>;
```

```
}}
```

```
}
```

---

# Reglas con Jerarquía

```
service cloud.firestore {
```

```
  match /databases/{database}/documents {
```

```
    match /cities/{city} {
```

```
      allow read, write: if <condition>;
```

```
// Definición explícita de reglas para subcolección 'neighborhood'
```

```
      match /neighborhoods/{neighborhood} {
```

```
        allow read, write: if <condition>;
```

```
      }
```

```
    }
```

```
  }
```

```
}
```

---

# Reglas con Jerarquía

//Se usan *match* anidados y la sentencia es relativa al path del *match* exterior

```
service cloud.firestore {  
  match /databases/{database}/documents {  
    match /cities/{city} {  
      match /neighborhoods/{neighborhood} {  
        allow read, write: if <condition>;  
      }  
    }  
  }  
}
```

---

# Condiciones para Reglas de Seguridad

## Autenticación

```
service cloud.firestore {  
  match /databases/{database}/documents {  
    // Permitir sólo a usuarios acceder los documentos en  
    la colección "cities" únicamente  
    si están autenticados.  
    match /cities/{city} {  
      allow read, write: if request.auth.uid != null;  
    }  
  }  
}
```

---

# Condiciones para Reglas de Seguridad

```
service cloud.firestore {  
  match /databases/{database}/documents {  
    // Asegurarse que el uid del usuario que realiza la  
    // petición haga match con el del usuario  
    // La expresión wildcard {userId} permite que la variable  
    // userId esté disponible en la regla  
    match /users/{userId} {  
      allow read, update, delete: if request.auth.uid ==  
      userId;  
      allow create: if request.auth.uid != null;  
    }  
  }  
}
```

---

# Condiciones para Reglas de Seguridad

```
service cloud.firestore {  
  match /databases/{database}/documents {  
    // Asegurarse que todas las ciudades tienen un valor  
    // positivo para el campo population y el nombre no es  
    // cambiado  
    match /cities/{city} {  
      allow update: if request.resource.data.population > 0  
        && request.resource.data.name ==  
        resource.data.name;  
    }  
  }  
}
```



---

# Condiciones para Reglas de Seguridad

## Validación de Datos

```
service cloud.firestore {  
  match /databases/{database}/documents {  
    // Permitir al usuario leer datos si el documento tiene  
    // el campo 'visibility' con el valor 'public'  
    match /cities/{city} {  
      allow read: if resource.data.visibility == 'public';  
    }  
  }  
}
```

---

# Condiciones para Reglas de Seguridad

## Acceso a otros Documentos

```
service cloud.firestore {  
  match /databases/{database}/documents {  
    match /cities/{city} {  
      // Asegurarse de que el documento 'users' existe para el  
      // usuario que hizo la petición antes de permitir escrituras  
      // a la colección 'cities'  
      allow create: if  
exists(/databases/{database}/documents/users/{request.auth.uid}))
```

---

# Condiciones para Reglas de Seguridad

## Acceso a otros Documentos

```
// Permitir al usuario borrar ciudades si el documento  
del usuario tiene el campo 'admin' con el valor 'true'
```

```
    allow delete: if  
get(/databases/$(database)/documents/users/$(request.auth.  
uid)).data.admin == true
```

```
}
```

```
}
```

```
}
```

---

# Condiciones para Reglas de Seguridad

## Funciones personalizadas

```
service cloud.firestore {  
  match /databases/{database}/documents {  
    // True si el usuario está autenticado o si la data  
    // solicitada es 'public'  
    function signedInOrPublic() {  
      return request.auth.uid != null ||  
      resource.data.visibility == 'public';  
    }  
  }  
}
```

---

# Condiciones para Reglas de Seguridad

## Funciones personalizadas

```
match /cities/{city} {  
  allow read, write: if signedInOrPublic();  
}
```

```
match /users/{user} {  
  allow read, write: if signedInOrPublic();  
}  
}  
}
```

---

# Escenarios Comunes de Reglas Inseguras

//Escenario 1: Cualquier usuario puede sobrescribir la base de datos!!

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

---

# Escenarios Comunes de Reglas Inseguras

```
service cloud.firestore {  
  match /databases/{database}/documents {  
    // Permitir únicamente al propietario de los datos  
    autenticado el acceso  
    match /some_collection/{document} {  
      allow read, write: if request.auth.uid ==  
request.resource.data.author_uid  
    }  
  }  
}
```

# Escenarios Comunes de Reglas Inseguras

```
service cloud.firestore {  
  match /databases/{database}/documents {  
    // Permitir acceso de lectura público, pero sólo los  
    // propietarios del contenido tienen acceso de escritura  
    match /some_collection/{document} {  
      allow read: if true  
      allow write: if request.auth.uid ==  
        request.resource.data.author_uid  
    }  
  }  
}
```



---

# Escenarios Comunes de Reglas Inseguras

//Escenario 2: Cualquier usuario autenticado puede leer y escribir la base de datos!!

```
service cloud.firestore {  
  match /databases/{database}/documents {  
    match /some_collection/{document} {  
      allow read, write: if request.auth.uid != null;  
    }  
  }  
}
```

---

# Escenarios Comunes de Reglas Inseguras

```
service cloud.firestore {  
  match /databases/{database}/documents {  
    // Permitir acceso público de lectura, pero únicamente los  
    // propietarios del contenido pueden realizar sobrescribir  
    match /some_collection/{document} {  
      allow read: if true  
      allow write: if request.auth.uid ==  
        request.resource.data.author_uid  
    }  
  }  
}
```

---

# Acceso Cerrado a Datos

Mientras se está desarrollando el App es recomendable que el acceso a los datos esté bloqueado

```
// Denegar acceso de lectura/escritura a todos los usuarios bajo cualquier condición
```

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

---

# Pruebas de Reglas de Seguridad

Utiliza el Cloud Firestore emulador para probar tus reglas de seguridad:



## Simulator



Simulation type

update



Location

/databases/(default)/documents

/databases/{database}/documents

Data ?

Build document

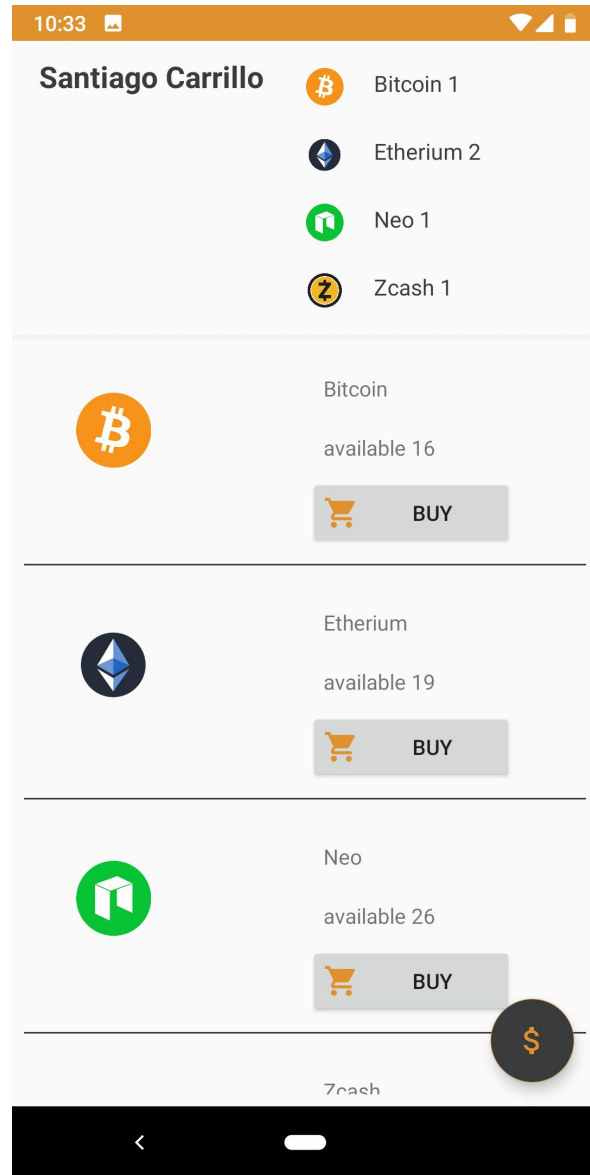
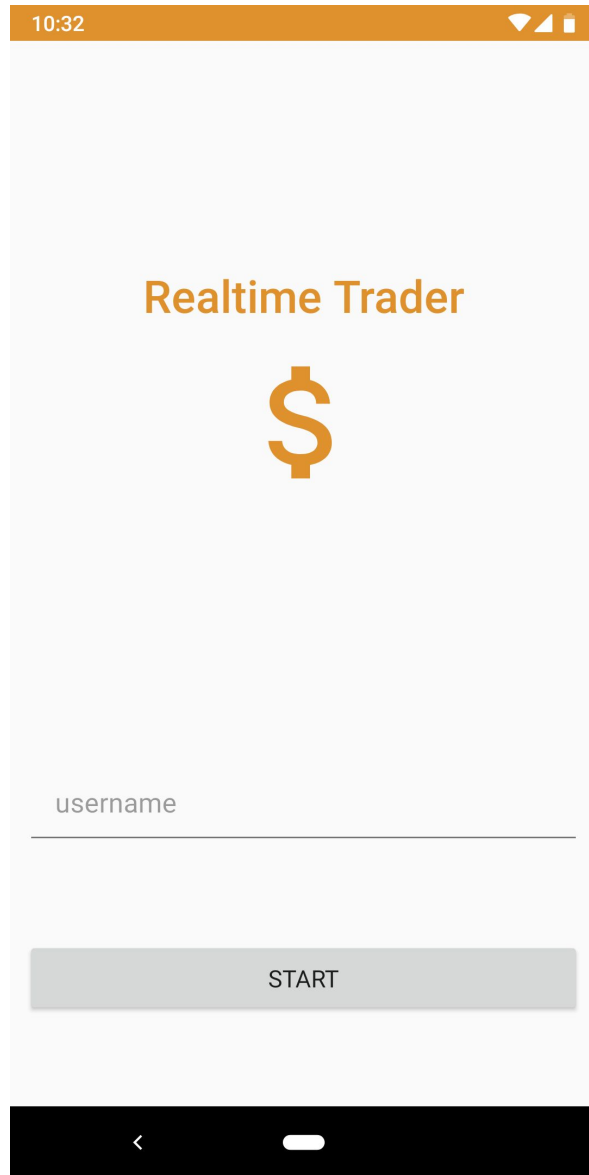
```
{"__name__": "/databases/(default)/documents/databases/{database}/documents/foo", "data": {"thing": "sauce"}}
```

Run

```
1 service cloud.firestore {
2   match /databases/{database}/documents {
3     match /{document=**} {
4       allow read, write: if false;
5     }
6   }
7 }
```

---

# Trading App en Realtime



---

# Adaptador de Criptomonedas

1. CryptosAdapterListener
2. CryptosAdapter
3. RecyclerView + FirestoreService



---

# Panel Info Criptomonedas

1. Carga Listado Criptomonedas  
Usuario
2. Carga Vistas Información  
Criptomonedas

---

# Actualizaciones Realtime Criptomonedas

1. Observación cambios Lista  
Criptomonedas
2. Observación cambios Criptomonedas de  
Usuario

---

# Generación Aleatoria de Criptomonedas

1. Usando la Función Random en Kotlin

---

# Repaso del Curso



## Firebase



Autenticación  
(Anónima)



Cloud  
Firestore

# Modelo de Datos

- Colecciones
- Documentos
- Referencias

The screenshot displays the MongoDB Atlas interface for a database named 'test-90da8'. The left sidebar shows the database structure with a collection named 'cryptos' and a sub-collection 'users'. The main panel shows the 'cryptos' collection with a document named 'bitcoin'. The document fields are:

- available: 29
- imageUrl: "https://en.bitcoin.it/w/images/en/2/29/BC\_Logo.png"
- name: "Bitcoin"

---

# Crear un Documento

Nuevo Documento con ID definido:

```
db.collection("cities")  
.document("new-city-id").set(data)
```

Nuevo Documento con ID autogenerado:

```
db.collection("cities")  
.document().set(data)
```

---

# Lectura de Datos

- **Directa:** Llamado a un método para obtener los datos
- **Obsevación:** Asignar un listener para ser notificado en modificaciones de los datos

```
citiesRef.where("name", "==", "Bogotá")
```

```
citiesRef.where("population", "<", 100000)
```

```
citiesRef.where("regions", "array-contains", "Amazonas")
```



---

# Reglas de Seguridad

```
service cloud.firestore {  
  match /databases/{database}/documents {  
  
    match /cryptos/{crypto} {  
      allow read: if <condition>;  
      allow write: if <condition>;  
      allow update: if <condition>;  
    }  
  }  
}
```

---

# Modos Índices

Index mode	Description
Ascending ↑	Supports <, <=, ==, >=, and > query clauses on the field and supports sorting results in ascending order based on this field value.
Descending ↓	Supports <, <=, ==, >=, and > query clauses on the field and supports sorting results in descending order based on this field value.
Array-contains	Supports <a href="#">array_contains</a> query clauses on the field.

# Revisemos el proyecto del curso

