Firestore Cheatsheet

A comprehensive guide to Google Firestore, covering data modeling, querying, security rules, and common operations. Ideal for developers building scalable, real-time applications.

Data Modeling & Structure

Collections & Documents

Collections	Containers for documents. Names are strings and must be unique within the database.
Documents	Contain data as key-value pairs. Can contain nested objects and arrays.
Document ID	Unique identifier for a document within a collection. Can be auto-generated or user-defined.
Subcollections	Collections nested within documents, useful for hierarchical data structures.
Path	Structure `collection/document/collection/document/...`

Data Types

String	Textual data.
Number	Integers and floating-point numbers.
Boolean	`true` or `false`.
Timestamp	Point in time (seconds and nanoseconds).
GeoPoint	Latitude and longitude coordinates.
Array	Ordered list of values.
Map	Nested key-value pairs (object).
Null	Represents a missing or undefined value.
Reference	Pointer to another document in Firestore.

Best Practices

- Favor denormalization to optimize read performance. Minimize subcollections to avoid complex queries.
- Use batched writes for multiple operations to improve efficiency.
- Structure data to fit query patterns; consider compound indexes.

Querying Data

Basic Queries

`where()`	Filters documents based on field values. Supports `=`, `<`, `>`, `<=`, `>=`, `array-contains`, `array-contains-any`, `in`, `not-in`.
`orderBy()`	Sorts the results by a specific field. Can specify ascending or descending order.
`limit()`	Limits the number of documents returned.
`startAt()`, `startAfter()`	Paginates results, starting at a specific document or value.
`endAt()`, `endBefore()`	Paginates results, ending at a specific document or value.

Compound Queries

Combine multiple `where()` clauses for complex filtering. Requires composite indexes for range and inequality filters on different fields.

Example: `db.collection('users').where('age', '>', 25).where('city', '==', 'New York').orderBy('age').limit(10)`

Collection Group Queries

Query across all collections with the same ID, regardless of their location in the database. Useful for retrieving data from deeply nested subcollections. Requires enabling collection group index.

Example: `db.collectionGroup('comments').where('approved', '==', true)`

Limitations

- Firestore does not support `OR` queries. You must perform multiple queries and merge the results client-side.
- Inequality and range queries are limited to a single field, unless you enable multiple range filters.

Security Rules

Basic Syntax

`rules_version`	Specifies the version of the rules syntax.
`service cloud.firestore`	Declares the service to which the rules apply.
`match`	Defines the database path to which the rules apply. Can use wildcards.
`allow read, write: if condition;`	Grants read or write access if the specified condition is met.

Common Conditions

`request.auth != null`	Requires authentication.
`request.auth.uid == userId`	Checks if the authenticated user's ID matches a specific user ID.
`request.resource.data.fieldName == 'value'`	Compares the value of a field in the incoming data to a specific value.
`get(/databases/(default)/documents/users/$(request.auth.uid)).data.role == 'admin'`	Verifies user's role stored in their profile document
`exists(/databases/(default)/documents/posts/$(postId))`	Checks if a document exists at a specific path.

Example Rules

```
service cloud.firestore {
  match /databases/{database}/documents {
    match /posts/{postId} {
      allow read: if true;
      allow create: if request.auth != null;
      allow update, delete: if request.auth != null && request.auth.uid == resource.data.authorId;
    }
  }
}
```

Testing Rules

- Use the Firestore Rules Simulator in the Firebase console to test your rules before deploying them.
- Simulate various scenarios, including authenticated and unauthenticated users, and different data values.

Common Operations

CRUD Operations

Create	`db.collection('collectionName').add({ data })` or `db.collection('collectionName').doc('docId').set({ data })`
Read	`db.collection('collectionName').doc('docId').get()`
Update	`db.collection('collectionName').doc('docId').update({ data })` (updates specified fields) or `db.collection('collectionName').doc('docId').set({ data }, { merge: true })` (merges data with existing document)
Delete	`db.collection('collectionName').doc('docId').delete()`

Transactions

Ensure atomicity and consistency when performing multiple operations. Transactions automatically retry if conflicts occur.

```
db.runTransaction(async (transaction) => {
  const sfDocRef =
db.collection('cities').doc('SF');
  const sfDoc = await
transaction.get(sfDocRef);
  if (!sfDoc.exists) {
    throw 'Document does not exist!';
  }
  const newPopulation =
sfDoc.data().population + 1;
  transaction.update(sfDocRef, { population:
newPopulation });
})
```

Batched Writes

Perform multiple write operations as a single atomic unit. More efficient than individual writes.

```
const batch = db.batch();
const sfRef =
db.collection('cities').doc('SF');
batch.update(sfRef, { name: 'San Francisco'
});
const laRef =
db.collection('cities').doc('LA');
batch.delete(laRef);
await batch.commit();
```

Realtime Updates

Listen for changes to documents or collections and receive updates in real-time.

```
db.collection('cities').doc('SF')
  .onSnapshot((doc) => {
    console.log('Current data: ', doc.data());
  });
```