Cloud Firestore

> ⚠ NOTICE
>
> This page is **archived** and might not reflect the latest version of the FlutterFire plugins. You can find the latest information on firebase.google.com:
>
> - https://firebase.google.com/docs/firestore/quickstart
> - https://firebase.google.com/docs/firestore/manage-data/add-data
> - https://firebase.google.com/docs/firestore/manage-data/transactions
> - https://firebase.google.com/docs/firestore/query-data/get-data
> - https://firebase.google.com/docs/firestore/query-data/listen
> - And throughout https://firebase.google.com/docs/firestore/... Most pages that have code snippets have Flutter examples. If you find a page that's missing Flutter snippets, please file a bug.

To start using the Cloud Firestore package within your project, import it at the top of your project files:

```
import 'package:cloud_firestore/cloud_firestore.dart';
```

Before using Firestore, you must first have ensured you have initialized FlutterFire.

To create a new Firestore instance, call the `instance` getter on `FirebaseFirestore`:

```
FirebaseFirestore firestore = FirebaseFirestore.instance;
```

By default, this allows you to interact with Firestore using the default Firebase App used whilst installing FlutterFire on your platform. If however you'd like to use Firestore with a secondary Firebase App, use the `instanceFor` method:

```
FirebaseApp secondaryApp = Firebase.app('SecondaryApp');
FirebaseFirestore firestore = FirebaseFirestore.instanceFor(app: secondaryApp);
```

Collections & Documents

Firestore stores data within "documents", which are contained within "collections". Documents can also contain nested collections. For example, our users would each have their own "document" stored inside the

"Users" collection. The `collection` method allows us to reference a collection within our code.

In the below example, we can reference the collection `users`, and create a new user document when a button is pressed:

```
import 'package:flutter/material.dart';

// Import the firebase_core and cloud_firestore plugin
import 'package:firebase_core/firebase_core.dart';
import 'package:cloud_firestore/cloud_firestore.dart';

class AddUser extends StatelessWidget {
  final String fullName;
  final String company;
  final int age;

  AddUser(this.fullName, this.company, this.age);

  @override
  Widget build(BuildContext context) {
    // Create a CollectionReference called users that references the firestore collection
    CollectionReference users = FirebaseFirestore.instance.collection('users');

    Future<void> addUser() {
      // Call the user's CollectionReference to add a new user
      return users
          .add({
            'full_name': fullName, // John Doe
            'company': company, // Stokes and Sons
            'age': age // 42
          })
          .then((value) => print("User Added"))
          .catchError((error) => print("Failed to add user: $error"));
    }

    return TextButton(
      onPressed: addUser,
      child: Text(
        "Add User",
      ),
    );
  }
}
```

Read Data

Cloud Firestore gives you the ability to read the value of a collection or a document. This can be a one-time read, or provided by realtime updates when the data within a query changes.

One-time Read

To read a collection or document once, call the `Query.get` or `DocumentReference.get` methods. In the below example a `FutureBuilder` is used to help manage the state of the request:

```
class GetUserName extends StatelessWidget {
  final String documentId;

  GetUserName(this.documentId);

  @override
  Widget build(BuildContext context) {
    CollectionReference users = FirebaseFirestore.instance.collection('users');

    return FutureBuilder<DocumentSnapshot>(
      future: users.doc(documentId).get(),
      builder:
          (BuildContext context, AsyncSnapshot<DocumentSnapshot> snapshot) {

        if (snapshot.hasError) {
          return Text("Something went wrong");
        }

        if (snapshot.hasData && !snapshot.data!.exists) {
          return Text("Document does not exist");
        }

        if (snapshot.connectionState == ConnectionState.done) {
          Map<String, dynamic> data = snapshot.data!.data() as Map<String, dynamic>;
          return Text("Full Name: ${data['full_name']} ${data['last_name']}");
        }

        return Text("loading");
      },
    );
  }
}
```

To learn more about reading data whilst offline, view the Access Data Offline documentation.

Realtime changes

FlutterFire provides support for dealing with realtime changes to collections and documents. A new event is provided on the initial request, and any subsequent changes to collection/document whenever a change occurs (modification, deleted or added).

Both the `CollectionReference` & `DocumentReference` provide a `snapshots()` method which returns a `Stream`:

```
Stream collectionStream = FirebaseFirestore.instance.collection('users').snapshots();
Stream documentStream = FirebaseFirestore.instance.collection('users').doc('ABC123').snapshots();
```

Once returned, you can subscribe to updates via the `listen()` method. The below example uses a `StreamBuilder` which helps automatically manage the streams state and disposal of the stream when it's no longer used within your app:

```
class UserInformation extends StatefulWidget {
  @override
    _UserInformationState createState() => _UserInformationState();
}

class _UserInformationState extends State<UserInformation> {
  final Stream<QuerySnapshot> _usersStream =
FirebaseFirestore.instance.collection('users').snapshots();

  @override
  Widget build(BuildContext context) {
    return StreamBuilder<QuerySnapshot>(
      stream: _usersStream,
      builder: (BuildContext context, AsyncSnapshot<QuerySnapshot> snapshot) {
        if (snapshot.hasError) {
          return Text('Something went wrong');
        }

        if (snapshot.connectionState == ConnectionState.waiting) {
          return Text("Loading");
        }

        return ListView(
          children: snapshot.data!.docs.map((DocumentSnapshot document) {
          Map<String, dynamic> data = document.data()! as Map<String, dynamic>;
            return ListTile(
              title: Text(data['full_name']),
              subtitle: Text(data['company']),
            );
          }).toList(),
        );
      },
    );
  }
}
```

By default, listeners do not update if there is a change that only affects the metadata. If you want to receive events when the document or query metadata changes, you can pass `includeMetadataChanges` to the `snapshots` method:

```
FirebaseFirestore.instance
  .collection('users')
  .snapshots(includeMetadataChanges: true)
```

Document & Query Snapshots

When performing a query, Firestore returns either a `QuerySnapshot` or a `DocumentSnapshot`.

QuerySnapshot

A `QuerySnapshot` is returned from a collection query, and allows you to inspect the collection, such as how many documents exist within it, gives access to the documents within the collection, see any changes since the last query and more.

To access the documents within a `QuerySnapshot`, call the `docs` property, which returns a `List` containing `DocumentSnapshot` classes.

```
FirebaseFirestore.instance
    .collection('users')
    .get()
    .then((QuerySnapshot querySnapshot) {
        querySnapshot.docs.forEach((doc) {
            print(doc["first_name"]);
        });
    });
```

DocumentSnapshot

A `DocumentSnapshot` is returned from a query, or by accessing the document directly. Even if no document exists in the database, a snapshot will always be returned.

To determine whether the document exists, use the `exists` property:

```
FirebaseFirestore.instance
    .collection('users')
    .doc(userId)
    .get()
    .then((DocumentSnapshot documentSnapshot) {
      if (documentSnapshot.exists) {
        print('Document exists on the database');
      }
    });
```

If the document exists, you can read the data of it by calling the `data` method, which returns a `Map<String, dynamic>`, or `null` if it does not exist:

```
FirebaseFirestore.instance
    .collection('users')
    .doc(userId)
    .get()
    .then((DocumentSnapshot documentSnapshot) {
      if (documentSnapshot.exists) {
        print('Document data: ${documentSnapshot.data()}');
      } else {
        print('Document does not exist on the database');
      }
    });
```

A `DocumentSnapshot` also provides the ability to access deeply nested data without manually iterating the returned `Map` via the `get` method. The method accepts a dot-separated path or a `FieldPath` instance. If no data exists at the nested path, a `StateError`:

```
try {
  dynamic nested = snapshot.get(FieldPath(['address', 'postcode']));
} on StateError catch(e) {
  print('No nested field exists!');
}
```

Querying

Cloud Firestore offers advanced capabilities for querying collections. Queries work with both one-time reads or subscribing to changes.

Filtering

To filter documents within a collection, the `where` method can be chained onto a collection reference. Filtering supports equality checks and "in" queries. For example, to filter users where their age is greater than 20:

```
FirebaseFirestore.instance
  .collection('users')
  .where('age', isGreaterThan: 20)
  .get()
  .then(...);
```

Firestore also supports array queries. For example, to filter users who speak English (en) or Italian (it), use the `arrayContainsAny` filter:

```
FirebaseFirestore.instance
  .collection('users')
  .where('language', arrayContainsAny: ['en', 'it'])
  .get()
  .then(...);
```

To learn more about all of the querying capabilities Cloud Firestore has to offer, view the Firebase documentation.

Limiting

To limit the number of documents returned from a query, use the `limit` method on a collection reference:

```
FirebaseFirestore.instance
  .collection('users')
```

```
  .limit(2)
  .get()
  .then(...);
```

You can also limit to the last documents within the collection query by using `limitToLast`:

```
FirebaseFirestore.instance
  .collection('users')
  .orderBy('age')
  .limitToLast(2)
  .get()
  .then(...);
```

Ordering

To order the documents by a specific value, use the `orderBy` method:

```
FirebaseFirestore.instance
  .collection('users')
  .orderBy('age', descending: true)
  .get()
  .then(...);
```

Start & End Cursors

To start and/or end a query at a specific point within a collection, you can pass a value to the `startAt`, `endAt`, `startAfter` or `endBefore` methods. You must specify an order to use cursor queries, for example:

```
FirebaseFirestore.instance
  .collection('users')
  .orderBy('age')
  .orderBy('company')
  .startAt([4, 'Alphabet Inc.'])
  .endAt([21, 'Google LLC'])
  .get()
  .then(...);
```

You can further specify a `DocumentSnapshot` instead of a specific value, by passing it to the `startAfterDocument`, `startAtDocument`, `endAtDocument` or `endBeforeDocument` methods. For example:

```
FirebaseFirestore.instance
  .collection('users')
  .orderBy('age')
  .startAfterDocument(documentSnapshot)
  .get()
```

```
.then(...);
```

Query Limitations

Cloud Firestore does not support the following types of queries:

- Queries with range filters on different fields, as described in the previous section.
- Logical OR queries. In this case, you should create a separate query for each OR condition and merge the query results in your app.
- Queries with a != clause. In this case, you should split the query into a greater-than query and a less-than query. For example, the query clause `where("age", isNotEqualTo: 30)` is not supported, however you can get the same result set by combining two queries, one with the clause `where("age", isLessThan: 30)` and one with the clause `where("age", isGreaterThan: 30)`

Writing Data

The Firebase Documentation provides some great examples on the best practices to structuring your data. It is recommended that you read the guide before building your database.

For more information on what is possible when writing data to Firestore, please refer to this documentation

Typing CollectionReference and DocumentReference

By default, Firestore references manipulate a `Map<String, dynamic>` object. The downside is that we lose type safety. One solution is to use `withConverter`, which will modify methods like `CollectionReference.add` or `Query.where` to be type-safe.

A common usage of `withConverter` is when combined with a serializable class, such as:

```
class Movie {
  Movie({required this.title, required this.genre});

  Movie.fromJson(Map<String, Object?> json)
    : this(
        title: json['title']! as String,
        genre: json['genre']! as String,
      );

  final String title;
  final String genre;

  Map<String, Object?> toJson() {
    return {
```

```
      'title': title,
      'genre': genre,
    };
  }
}
```

We can then use `withConverter` to manipulate a collection of movies like so:

```
final moviesRef = FirebaseFirestore.instance.collection('movies').withConverter<Movie>(
      fromFirestore: (snapshot, _) => Movie.fromJson(snapshot.data()!),
      toFirestore: (movie, _) => movie.toJson(),
    );
```

Finally, we can use our new `moviesRef` variable to perform read and write operations:

```
Future<void> main() async {
  // Obtain science-fiction movies
  List<QueryDocumentSnapshot<Movie>> movies = await moviesRef
      .where('genre', isEqualTo: 'Sci-fi')
      .get()
      .then((snapshot) => snapshot.docs);

  // Add a movie
  await moviesRef.add(
    Movie(
      title: 'Star Wars: A New Hope (Episode IV)',
      genre: 'Sci-fi'
    ),
  );

  // Get a movie with the id 42
  Movie movie42 = await moviesRef.doc('42').get().then((snapshot) => snapshot.data()!);
}
```

Adding Documents

To add a new document to a collection, use the `add` method on a `CollectionReference`:

```
class AddUser extends StatelessWidget {
  final String fullName;
  final String company;
  final int age;

  AddUser(this.fullName, this.company, this.age);

  @override
  Widget build(BuildContext context) {
    // Create a CollectionReference called users that references the firestore collection
    CollectionReference users = FirebaseFirestore.instance.collection('users');
```

```
    Future<void> addUser() {
      // Call the user's CollectionReference to add a new user
      return users
          .add({
            'full_name': fullName, // John Doe
            'company': company, // Stokes and Sons
            'age': age // 42
          })
          .then((value) => print("User Added"))
          .catchError((error) => print("Failed to add user: $error"));
    }

    return FlatButton(
      onPressed: addUser,
      child: Text(
        "Add User",
      ),
    );
  }
}
```

The `add` method adds the new document to your collection with a unique auto-generated ID. If you'd like to specify your own ID, call the `set` method on a `DocumentReference` instead:

```
CollectionReference users = FirebaseFirestore.instance.collection('users');

Future<void> addUser() {
  return users
    .doc('ABC123')
    .set({
      'full_name': "Mary Jane",
      'age': 18
    })
    .then((value) => print("User Added"))
    .catchError((error) => print("Failed to add user: $error"));
}
```

Calling `set` with an id that already exists on the collection will replace all the document data. You can also specify `SetOptions(merge: true))` on the query, and this will merge the existing document with the data passed into the `set()` :

```
CollectionReference users = FirebaseFirestore.instance.collection('users');

Future<void> addUser() {
  return users
    // existing document in 'users' collection: "ABC123"
    .doc('ABC123')
    .set({
      'full_name': "Mary Jane",
      'age': 18
      },
      SetOptions(merge: true),
    )
    .then(
```

```
      (value) => print("'full_name' & 'age' merged with existing data!")
    )
    .catchError((error) => print("Failed to merge data: $error"));
}
```

Updating documents

Sometimes you may wish to update a document, rather than replacing all of the data. The `set` method above replaces any existing data on a given `DocumentReference`. If you'd like to update a document instead, use the `update` method:

```
CollectionReference users = FirebaseFirestore.instance.collection('users');

Future<void> updateUser() {
  return users
    .doc('ABC123')
    .update({'company': 'Stokes and Sons'})
    .then((value) => print("User Updated"))
    .catchError((error) => print("Failed to update user: $error"));
}
```

The method also provides support for updating deeply nested values via dot-notation:

```
CollectionReference users = FirebaseFirestore.instance.collection('users');

Future<void> updateUser() {
  return users
    .doc('ABC123')
    .update({'info.address.zipcode': 90210})
    .then((value) => print("User Updated"))
    .catchError((error) => print("Failed to update user: $error"));
}
```

Field values

Cloud Firestore supports storing and manipulating values on your database, such as Timestamps, GeoPoints, Blobs and array management.

To store `GeoPoint` values, provide the latitude and longitude to the GeoPoint class:

```
CollectionReference users = FirebaseFirestore.instance.collection('users');

Future<void> updateUser() {
  return users
    .doc('ABC123')
    .update({'info.address.location': GeoPoint(53.483959, -2.244644)})
    .then((value) => print("User Updated"))
    .catchError((error) => print("Failed to update user: $error"));
```

```
}
```

To store a Blob such as an image, provide a `Uint8List`. The below example shows how to get an image from your `assets` directory and nest it in the `info` object in Firestore.

```
CollectionReference users = FirebaseFirestore.instance.collection('users');

Future<void> updateUser() {
  return rootBundle
    .load('assets/images/sample.jpg')
    .then((bytes) => bytes.buffer.asUint8List())
    .then((avatar) {
      return users
        .doc('ABC123')
        .update({'info.avatar': Blob(avatar)});
      })
    .then((value) => print("User Updated"))
    .catchError((error) => print("Failed to update user: $error"));
}
```

Removing Data

To delete documents with Cloud Firestore, you can use the `delete` method on a `DocumentReference`:

```
CollectionReference users = FirebaseFirestore.instance.collection('users');

Future<void> deleteUser() {
  return users
    .doc('ABC123')
    .delete()
    .then((value) => print("User Deleted"))
    .catchError((error) => print("Failed to delete user: $error"));
}
```

If you need to remove specific properties from within a document rather than the document itself, you can use the `delete` method with the `FieldValue` class:

```
CollectionReference users = FirebaseFirestore.instance.collection('users');

Future<void> deleteField() {
  return users
    .doc('ABC123')
    .update({'age': FieldValue.delete()})
    .then((value) => print("User's Property Deleted"))
    .catchError((error) => print("Failed to delete user's property: $error"));
}
```

Transactions

Transactions are a way to ensure that a write operation only occurs using the latest data available on the server. Transactions never partially apply writes, and writes execute at the end of a successful transaction.

Transactions are useful when you want to update a field based on its current value, or the value of another field. If you want to write multiple documents without using the documents current state, a batch write should be used.

When using transactions, note that:

- Read operations must come before write operations
- Transactions will fail when the client is offline, they cannot use cached data

An example of where a transaction could be used would be in an application where a user can subscribe to a channel. When a user presses the subscribe button, a "subscribers" field in a document increments. Without using Transactions, we would first need to read the existing value, and then increment that value using two separate operations.

On a high traffic application, the value on the server could have already changed by the time the write operation sets a new value, causing the number to be inconsistent.

Transactions remove this issue by atomically updating the value of the server. If the value changes whilst the transaction is executing, it will retry, ensuring the value on the server is used, rather than the client value.

To execute a transaction, call the `runTransaction` method:

```
// Create a reference to the document the transaction will use
DocumentReference documentReference = FirebaseFirestore.instance
  .collection('users')
  .doc(documentId);

return FirebaseFirestore.instance.runTransaction((transaction) async {
  // Get the document
  DocumentSnapshot snapshot = await transaction.get(documentReference);

  if (!snapshot.exists) {
    throw Exception("User does not exist!");
  }

  // Update the follower count based on the current count
  // Note: this could be done without a transaction
  // by updating the population using FieldValue.increment()

  int newFollowerCount = snapshot.data()['followers'] + 1;

  // Perform an update on the document
  transaction.update(documentReference, {'followers': newFollowerCount});
```

```
  // Return the new count
  return newFollowerCount;
})
.then((value) => print("Follower count updated to $value"))
.catchError((error) => print("Failed to update user followers: $error"));
```

In the above example, if the document changes at any point during the transaction, it will retry up-to five times.

You should not directly modify application state inside of the transaction, as the handler may execute multiple times. You should instead return a value at the end of the handler, updating application state once the transaction has completed.

If an exception is thrown within the handler, the entire transaction will be aborted.

Batch write

Firestore lets you execute multiple write operations as a single batch that can contain any combination of `set`, `update`, or `delete` operations.

First, create a new batch instance via the `batch` method, then perform the operations on the batch, and then commit it once ready. The below example shows how to delete all documents in a collection in a single operation:

```
CollectionReference users = FirebaseFirestore.instance.collection('users');

Future<void> batchDelete() {
  WriteBatch batch = FirebaseFirestore.instance.batch();

  return users.get().then((querySnapshot) {
    querySnapshot.docs.forEach((document) {
      batch.delete(document.reference);
    });

    return batch.commit();
  });
}
```

Data Security

It is important that you understand how to write rules in your Firebase console to ensure that your data is secure. Please follow the Firebase Firestore documentation on security.

Access Data Offline

Configure Offline Persistence

Firestore provides out of the box support for offline capabilities. When reading and writing data, Firestore uses a local database which automatically synchronizes with the server. Cloud Firestore functionality continues when users are offline, and automatically handles data migration when they regain connectivity.

This functionality is enabled by default, however it can be disabled if needed. The `settings` must be set before any Firestore interaction is performed:

```
// Web.
await FirebaseFirestore.instance.enablePersistence();

// All other platforms.
FirebaseFirestore.instance.settings =
    Settings(persistenceEnabled: false);
```

If you want to clear any persisted data, you can call the `clearPersistence()` method.

```
await FirebaseFirestore.instance.clearPersistence();
```

Calls to update settings or clearing persistence must be carried out before any other usage of Firestore. If called afterwards, they will take effect on the next Firestore claim (e.g. restarting the application).

Configure Cache Size

When persistence is enabled, Firestore caches every document for offline access. After exceeding the cache size, Firestore will attempt to remove older, unused data. You can configure different cache sizes, or disable the removal process:

```
// The default value is 40 MB. The threshold must be set to at least 1 MB,
// and can be set to Settings.CACHE_SIZE_UNLIMITED to disable garbage collection.

FirebaseFirestore.instance.settings =
    Settings(cacheSizeBytes: Settings.CACHE_SIZE_UNLIMITED);
```

Disable and Enable Network Access

It is possible to disable network access for your Firestore client. While network access is disabled, all Firestore requests retrieve results from the cache. Any write operations are queued until network access is re-enabled.

```
await FirebaseFirestore.instance.disableNetwork()
```

To re-enabled network access, call the `enableNetwork` method:

```
await FirebaseFirestore.instance.enableNetwork()
```

Emulator Usage

If you are using the local Firestore emulators, then it is possible to connect to this using the `useFirestoreEmulator` method. Ensure you pass the correct port on which the Firebase emulator is running on.

Ensure you have enabled network connections to the emulators in your apps following the emulator usage instructions in the general FlutterFire installation notes for each operating system.

```
Future<void> main() async {
  WidgetsFlutterBinding.ensureInitialized();
  await Firebase.initializeApp();

  // Ideal time to initialize
  FirebaseFirestore.instance.useFirestoreEmulator('localhost', 8080);
  //...
}
```

Data Bundles

If you have setup a data bundle and are currently serving it, you may load data bundles into your app.

```
// Use a package like 'http' to retrieve bundle.
import 'package:http/http.dart' as http;
final response = await http.get(url);

// Convert the 'bundle.txt' string in the response to an Uint8List instance.
Uint8List buffer = Uint8List.fromList(response.body.codeUnits);

// Load bundle into cache.
LoadBundleTask task = FirebaseFirestore.instance.loadBundle(buffer);

// Use .stream API to expose a stream which listens for LoadBundleTaskSnapshot events.
task.stream.listen((taskStateProgress) {
  if(taskStateProgress.taskState == LoadBundleTaskState.success){
    //bundle is loaded into app cache!
  }
});
```

```
// If you do not wish to .listen() to the stream, but simply want to know when the bundle has been loaded. Use .last API:
await task.stream.last;

// Once bundle is loaded into cache, you may query for data by using the GetOptions() to specify data retrieval from cache.
QuerySnapshot<Map<String, Object?>> snapshot = await FirebaseFirestore.instance
  .collection('cached-data')
  .get(const GetOptions(source: Source.cache));
```

Named Query

If you have loaded a bundle with a named query, you may retrieve the query snapshot by calling the `namedQueryGet()` API:

```
// Assuming you've loaded a data bundle that includes a named query called 'sports-teams':

// Query snapshot loaded from cache.
QuerySnapshot<Map<String, Object?>>  snapshot = await
FirebaseFirestore.instance.namedQueryGet('sports-teams', options: const GetOptions(source:
Source.cache));
```

Distributed Counters

To support more frequent counter updates, create a distributed counter. Each counter is a document with a subcollection of `shards`, and the value of the counter is the sum of the value of the `shards`.

Write throughput increases linearly with the number of `shards`, so a distributed counter with 10 `shards` can handle 10x as many writes as a traditional counter.

> NOTE
>
> An alternative solution is to use **Firebase extensions**, please refer to the Ditributed Counters extension and how to set it up here.

A distributed counter collection would look like this:

```
// counters/${ID}
class Counter {
  final int numShards;

  Counter(this.numShards);
}
```

```
// counters/${ID}/shards/${NUM}
class Shard {
  final int count;

  Shard(this.count);
}
```

The following code initializes a distributed counter:

```
Future<void> createCounter(DocumentReference ref, int numShards) async {
  WriteBatch batch = FirebaseFirestore.instance.batch();

  // Initialize the counter document
  batch.set(ref, {'numShards': numShards});

  // Initialize each shard with count=0
  for (var i = 0; i < numShards; i++) {
    final shardRef = ref.collection('shards').doc(i.toString());
    batch.set(shardRef, {'count': 0});
  }

  // Commit the write batch
  await batch.commit();
}
```

To increment the counter, choose a random shard and increment the `count`:

```
Future<void> incrementCounter(DocumentReference ref, int numShards) async {
  // Select a shard of the counter at random

  final shardId = Random().nextInt(numShards).toString();
  final shardRef = ref.collection('shards').doc(shardId);

  // Update count
  await shardRef.update({'count': FieldValue.increment(1)});
}
```

To get the total count, query for all shards and sum their `count` fields:

```
Future<int> getCount(DocumentReference ref) async {
  // Sum the count of each shard in the subcollection
  final shards = await ref.collection('shards').get();

  int totalCount = 0;

  shards.docs.forEach(
    (doc) {
      totalCount += doc.data()['count'] as int;
    },
  );

  return totalCount;
```

```
}
```

Refer to the official documentation on distributed counters for more details.

Edit this page