**Cloud Firestore**

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

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

FirebaseApp secondaryApp = Firebase.app('SecondaryApp');

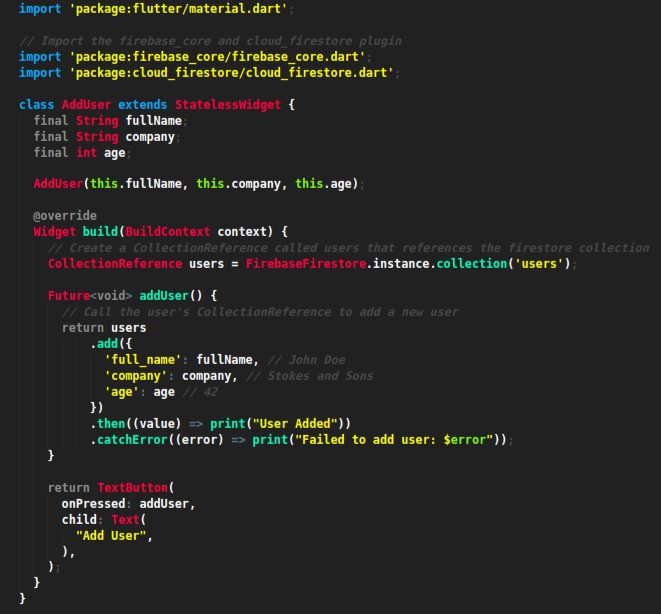
FirebaseFirestore firestore = FirebaseFirestore.instanceFor(app: secondaryApp);

**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`](!cloud\_firestore.FirebaseFirestore.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:**

****

**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

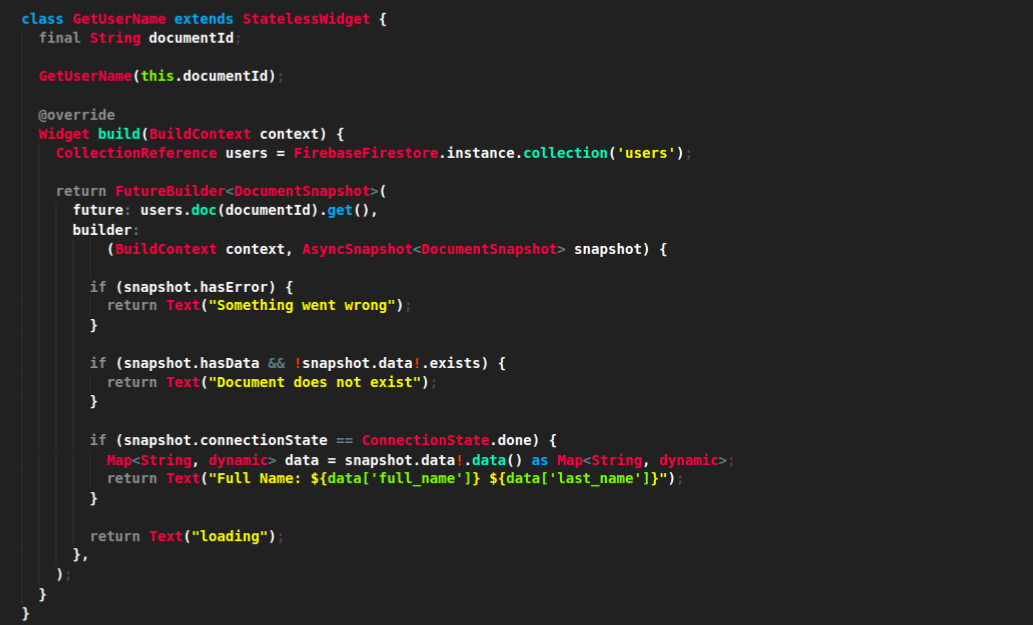
provided by realtime updates when the data within a query changes.

**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:

****

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

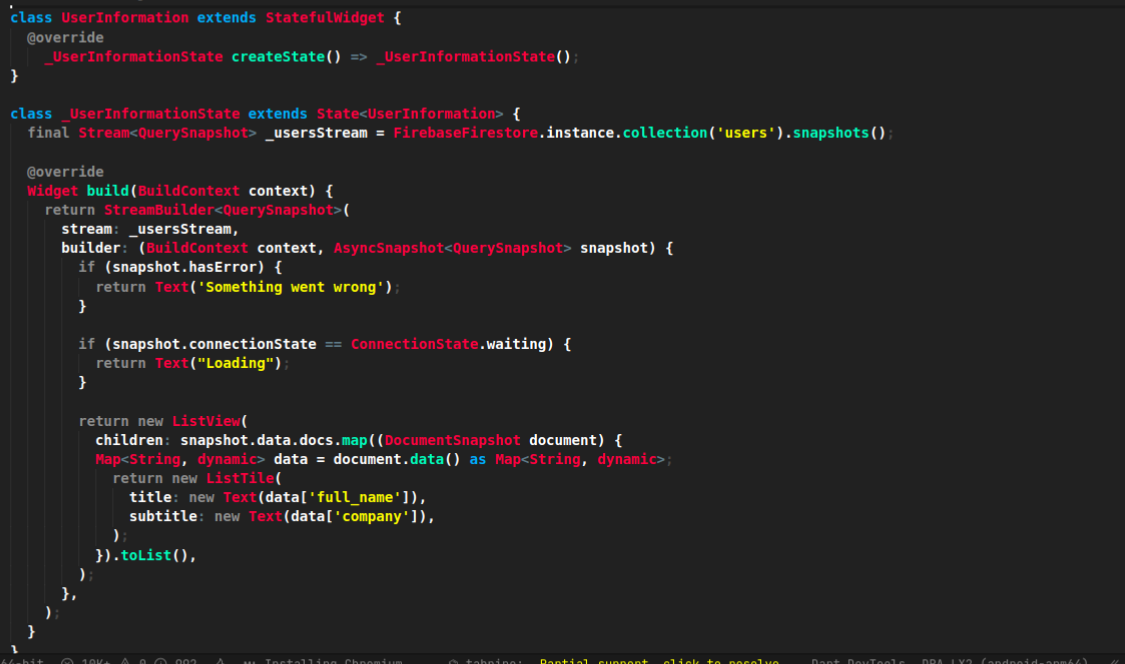
or added).

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

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`].

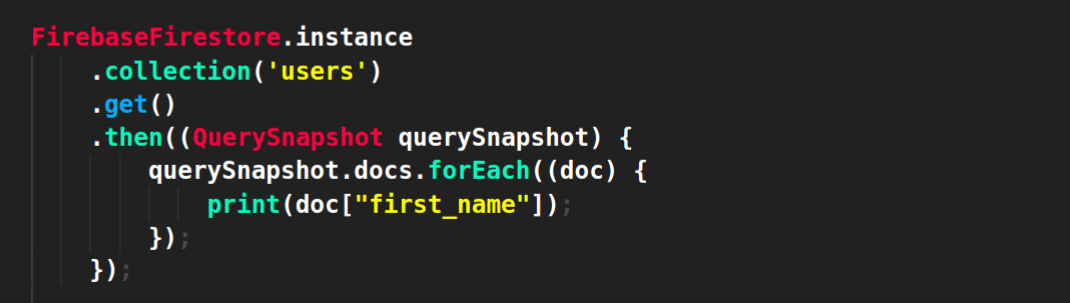
**QuerySnapshot**

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.



**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, for filter

users where their age is greater than 20:

FirebaseFirestore.instance

.collection('users')

.where('age', isGreaterThan: 20)

.get()

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(...);

**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 cause, 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](https://firebase.google.com/docs/firestore/manage-data/structure-data) 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](https://firebase.google.com/docs/firestore/manage-data/add-data)

**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,**

**};**

**}**

**}**

**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* 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.

**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"))

}

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"));

}