The Firestore Tutorial for 2020

Table of Contents

- Unsubscribing from a collection
- Getting individual documents

Changing Data with Firestore

- Adding document to a collection with .add()
- Adding a document to a collection with .set()
- Updating existing data
- Deleting data

Essential Patterns

- Working with subcollections
- Useful methods for Firestore fields
- Querying with .where()
- Ordering and limiting data

What is Firestore? Why Should You Use It?

Firestore is a very flexible, easy to use database for mobile, web and server development. If you're familiar with Firebase's realtime database, Firestore has many similarities, but with a different (arguably more declarative) API.

Here are some of the features that Firestore brings to the table:

⚡Easily get data in realtime

Like the Firebase realtime database, Firestore provides useful methods such as .onSnapshot() which make it a breeze to listen for updates to your data in real time. It makes Firestore an ideal choice for projects that place a premium on displaying and using the most recent data (chat applications, for instance).

🥞 Flexibility as a NoSQL Database

Firestore is a very flexible option for a backend because it is a NoSQL database. NoSQL means that the data isn't stored in tables and columns as a standard SQL database would be. It is structured like a key-value store, as if it was one big JavaScript object. In other words, there's no schema or need to describe what data our database will store. As long as we provide valid keys and values, Firestore will store it.

↕ Effortlessly scalable

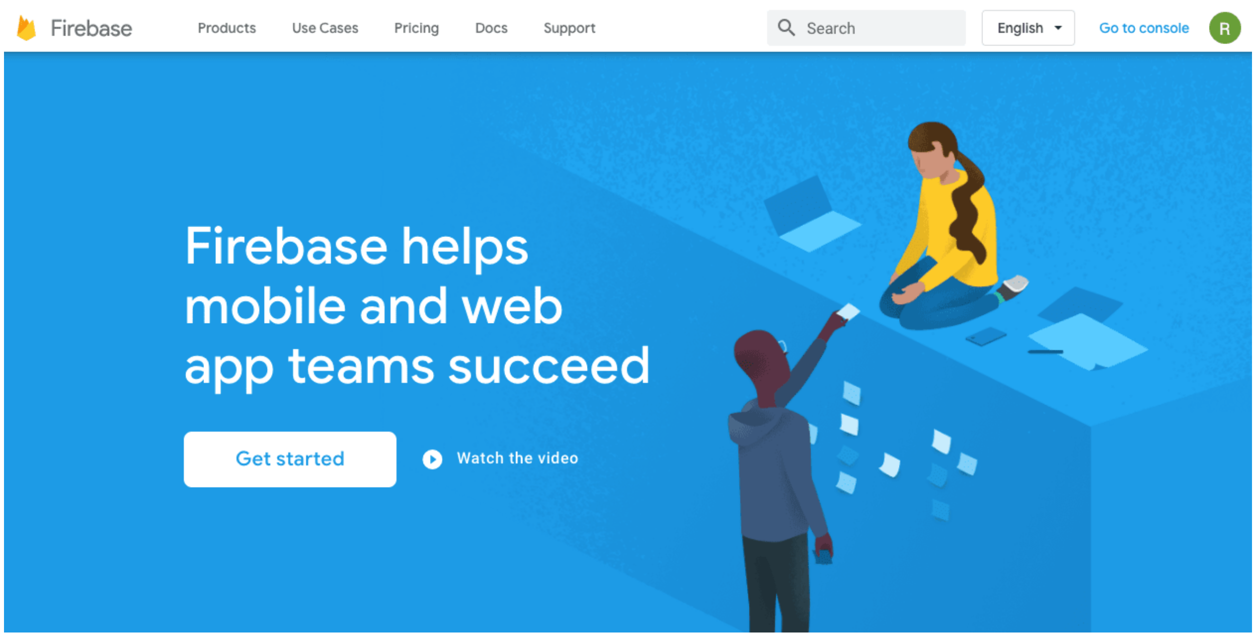
One great benefit of choosing Firestore for your database is the very powerful infrastructure that it builds upon that enables you to scale your application very easily. Both vertically and horizontally. No matter whether you have hundreds or millions of users. Google's servers will be able to handle whatever load you place upon it.

In short, Firestore is a great option for applications both small and large. For small applications it's powerful because we can do a lot without much setup and create projects very quickly with them. Firestore is well-suited for large projects due to it's scalability.

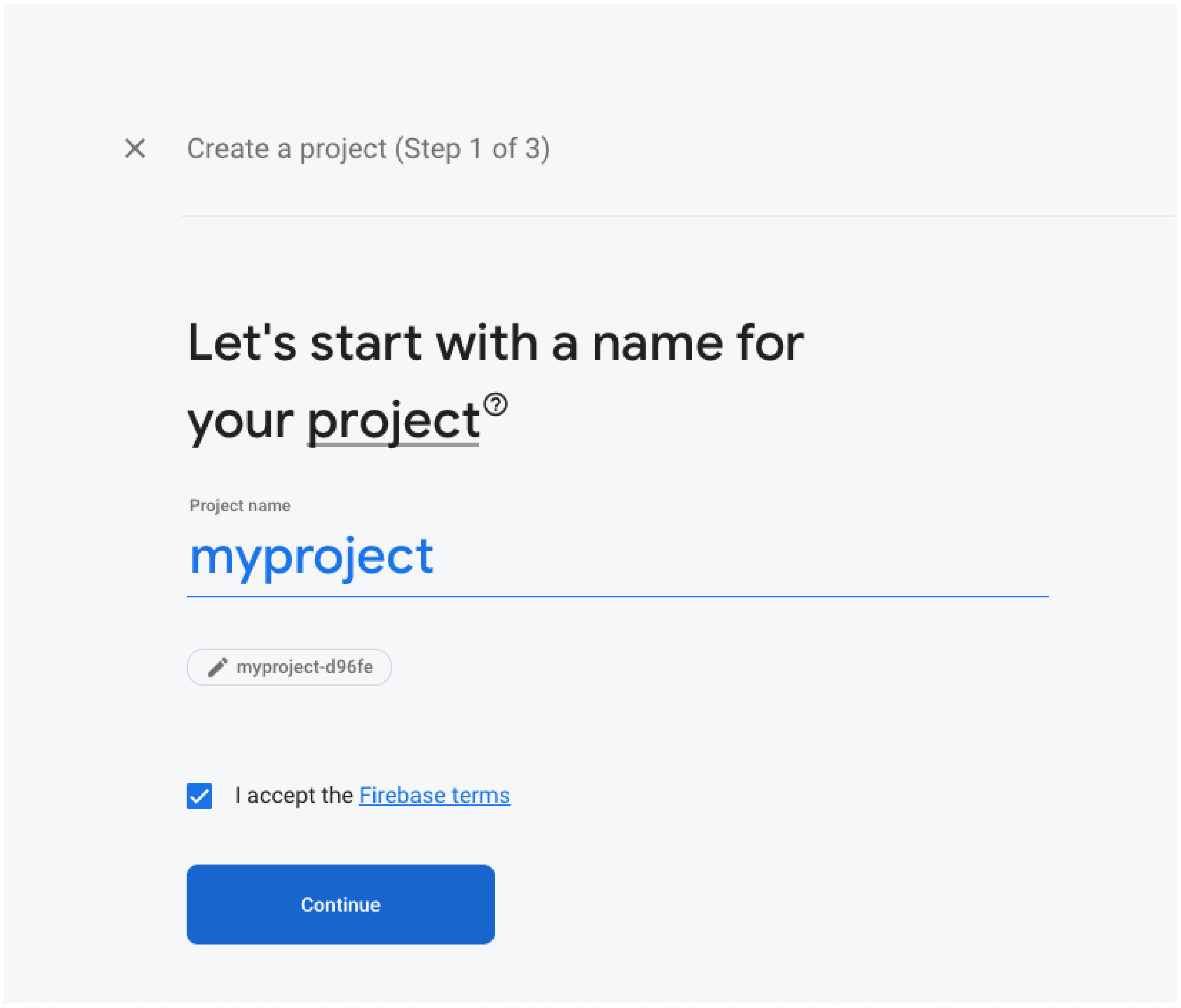
Setting Up Firestore in a JavaScript Project

> We're going to be using the Firestore SDK for JavaScript. Throughout this cheatsheet, we'll cover how to use Firestore within the context of a JavaScript project. In spite of this, the concepts we'll cover here are easily transferable to any of the available Firestore client libraries.

To get started with Firestore, we'll head to the Firebase console. You can visit that by going to firebase.google.com. You'll need to have a Google account to sign in.

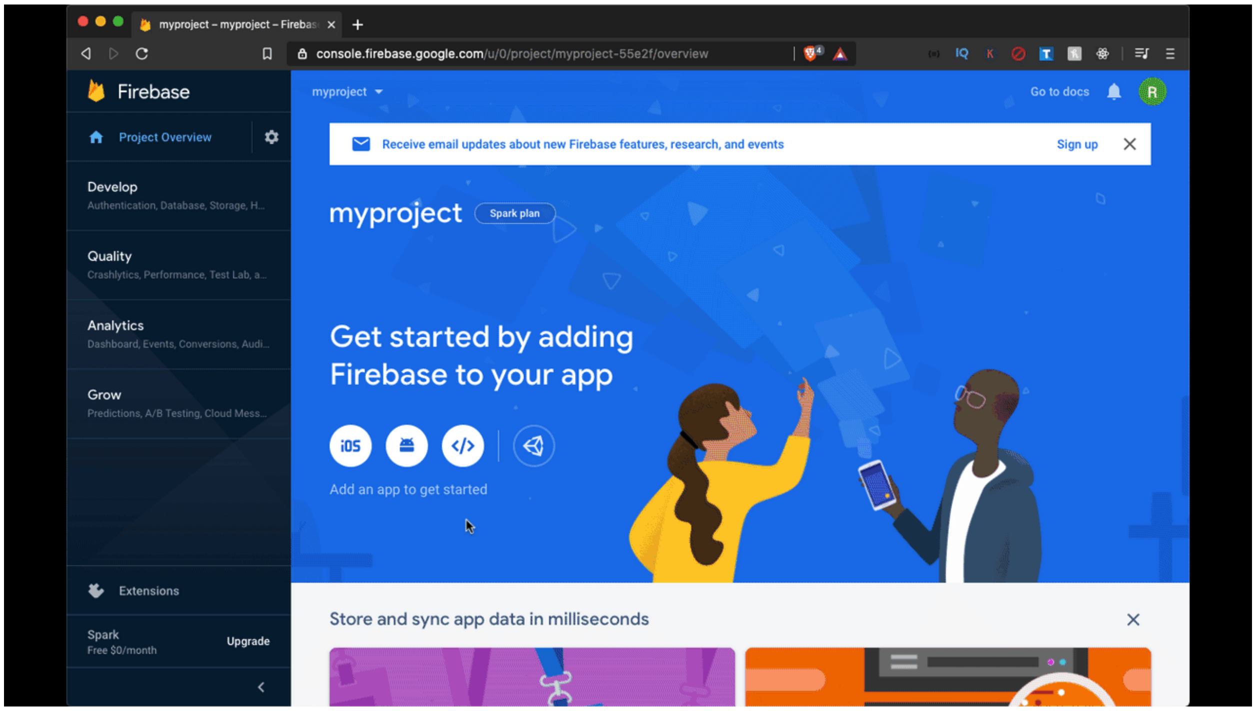


Once we're signed in, we'll create a new project and give it a name.


×
Create a project (Step 1 of 3)
Let's start with a name for your project
Project name
myproject
myproject-d96fe
I accept the Firebase terms
Continue

Once our project is created, we'll select it. After that, on our project's dashboard, we'll select the code button.

This will give us the code we need to integrate Firestore with our JavaScript project.


myproject – myproject – Firebase
console.firebase.google.com/u/0/project/myproject-55e2f/overview
Firebase
Project Overview
Develop
Authentication, Database, Storage, H...
Quality
Crashlytics, Performance, Test Lab, a...
Analytics
Dashboard, Events, Conversions, Audi...
Grow
Predictions, A/B Testing, Cloud Mess...
Extensions
Spark
Free $0/month
Upgrade
myproject
Go to docs
Receive email updates about new Firebase features, research, and events
Sign up
myproject
Spark plan
Get started by adding Firebase to your app
Add an app to get started
Store and sync app data in milliseconds

Usually if you're setting this up in any sort of JavaScript application, you'll want to put this in a dedicated file called firebase.js. If you're using any JavaScript library that has a package.json file, you'll want to install the Firebase dependency with npm or yarn.

```
// with npm
npm i firebase

// with yarn
yarn add firebase
```

Firestore can be used either on the client or server. If you are using Firestore with Node, you'll need to use the CommonJS syntax with require. Otherwise, if you're using JavaScript in the client, you'll import firebase using ES Modules.

```
// with Commonjs syntax (if using Node)
const firebase = require("firebase/app");
require("firebase/firestore");

// with ES Modules (if using client-side JS, like React)
import firebase from 'firebase/app';
import 'firebase/firestore';
```

```
var firebaseConfig = {
  apiKey: "AIzaSyDpLmM79mUqbMDBexFtOQOkSl0glxCW_ds",
  authDomain: "lfasdfkjkjlkjl.firebaseapp.com",
  databaseURL: "https://lfasdlkjkjlkjl.firebaseio.com",
  projectId: "lfasdlkjkjlkjl",
  storageBucket: "lfasdlkjkjlkjl.appspot.com",
  messagingSenderId: "616270824980",
  appId: "1:616270824990:web:40c8b177c6b9729cb5110f",
};
// Initialize Firebase
firebase.initializeApp(firebaseConfig);
```

Firestore Collections and Documents

There are two key terms that are essential in understanding how to work with Firestore: **documents** and **collections**.

Documents are individual pieces of data in our database. You can think of documents to be much like simple JavaScript objects. They consist of key-value pairs, which we refer to as **fields**. The values of these fields can be strings, numbers, Booleans, objects, arrays, and even binary data.

```
document -> { key: value }
```

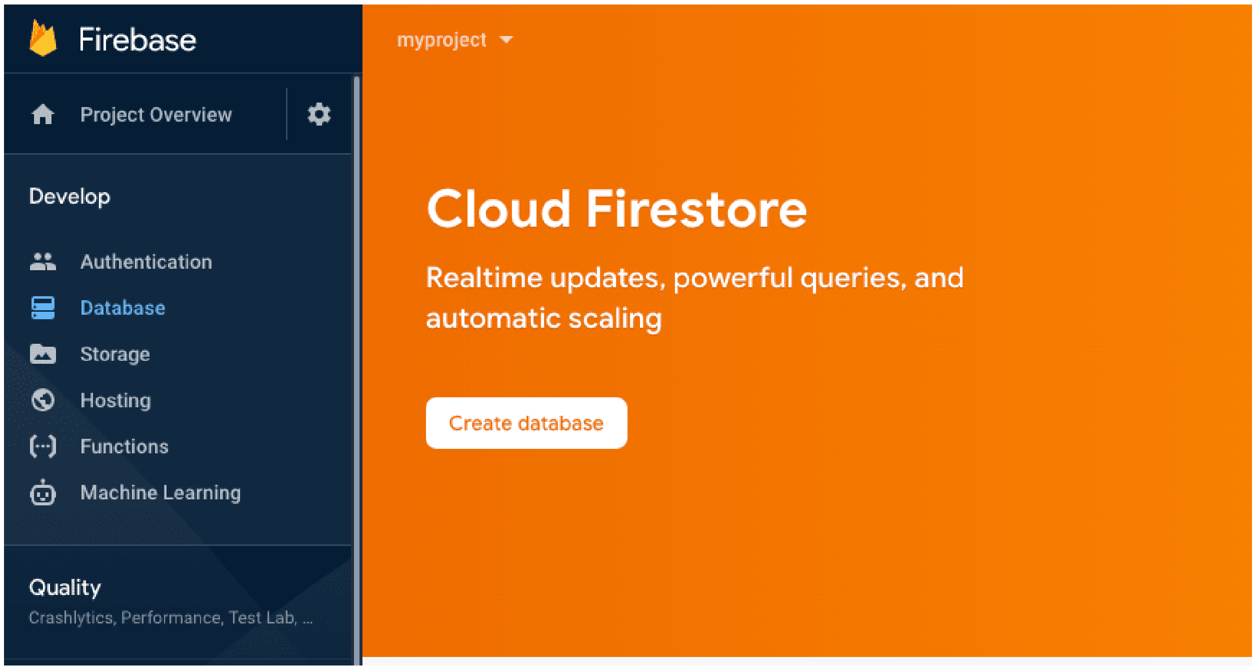
Sets of these documents of these documents are known as collections. Collections are very much like arrays of objects. Within a collection, each document is linked to a given identifier (id).

```
collection -> [{ id: doc }, { id: doc }]
```

Managing our database with the Firestore Console

Before we can actually start working with our database we need to create it.

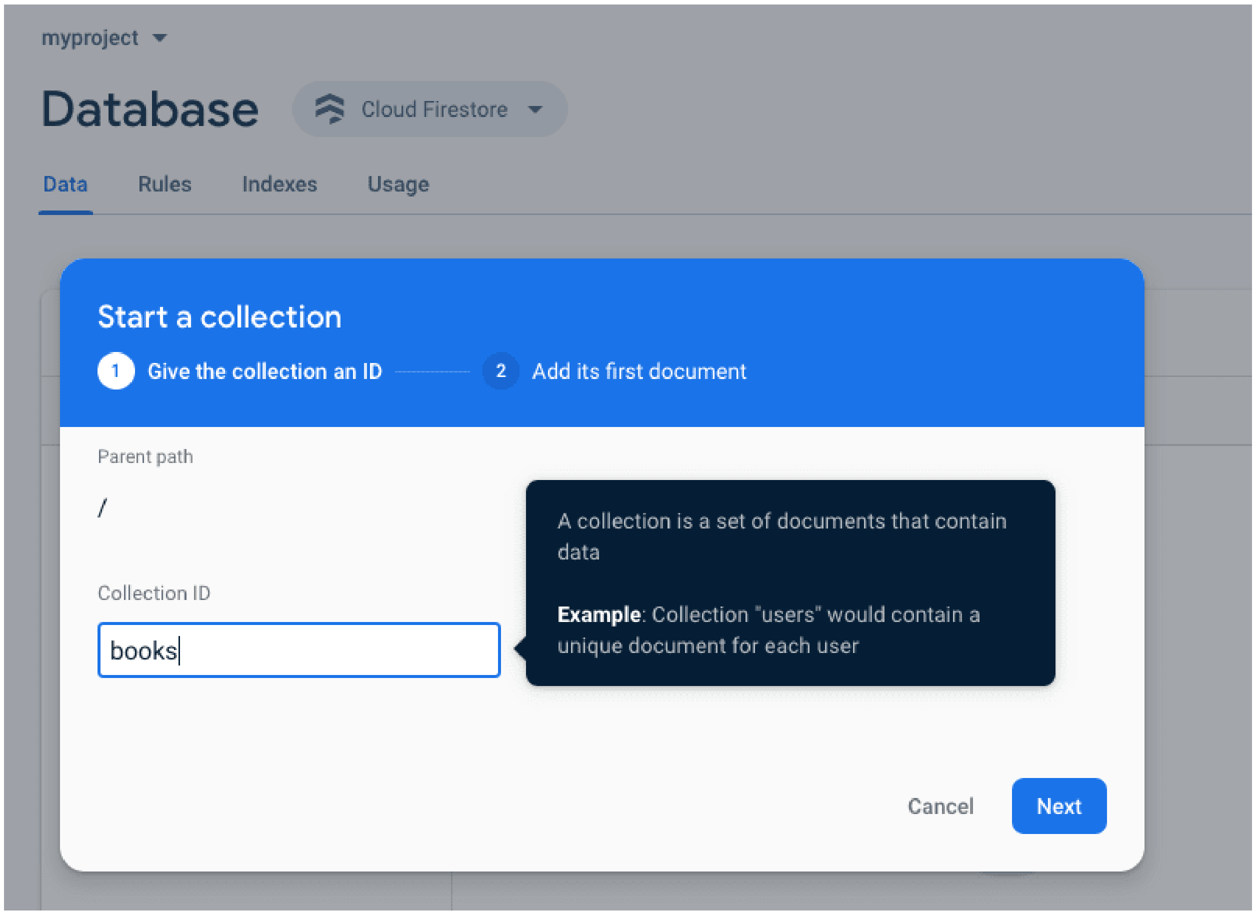
Within our Firebase console, go to the 'Database' tab and create your Firestore database.



Once you've done that, we will start in test mode and enable all reads and writes to our database. In other words, we will have open access to get and change data in our database. If we were to add Firebase authentication, we could restrict access only to authenticated users.

After that, we'll be taken to our database itself, where we can start creating collections and documents. The root of our database will be a series of collections, so let's make our first collection.

We can select 'Start collection' and give it an id. Every collection is going to have an id or a name. For our project, we're going to keep track of our users' favorite books. We'll give our first collection the id 'books'.



A collection is a set of documents that contain data

Example: Collection "users" would contain a unique document for each user

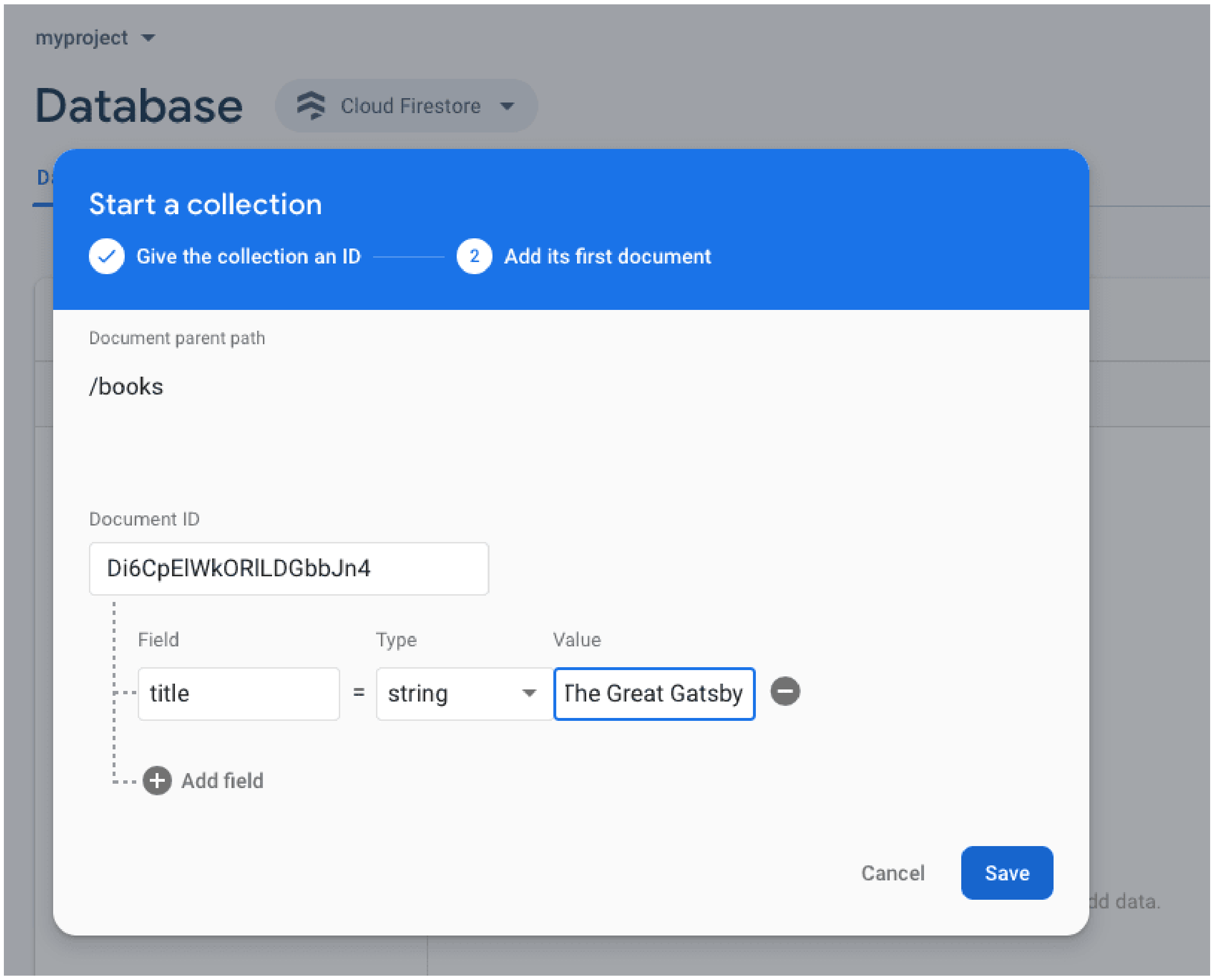
Next, we'll add our first document with our newly-created 'books' collection.

Each document is going to have an id as well, linking it to the collection in which it exists.

In most cases we're going to use an option to give it an automatically generated ID. So we can hit the button 'auto id' to do so, after which we need to provide a field, give it a type, as well as a value.

For our first book, we'll make a 'title' field of type 'string', with the value 'The Great Gatsby', and hit save.

After that, we should see our first item in our database.



Document parent path
/books

Document ID
Di6CpElWkORILDGbbJn4

Field		Type	Value
title	=	string	The Great Gatsby

Getting data from a collection with .get()

To get access Firestore use all of the methods it provides, we use `firebase.firestore()`. This method need to be executed every time we want to interact with our Firestore database.

I would recommend creating a dedicated variable to store a single reference to Firestore. Doing so helps to cut down on the amount of code you write across your app.

```
const db = firebase.firestore();
```

> In this cheatsheet, however, I'm going to stick to using the firestore method each time to be as clear as possible.

To reference a collection, we use the `.collection()` method and provide a collection's id as an argument. To get a reference to the books collection we created, just pass in the string 'books'.

```
const booksRef = firebase.firestore().collection('books');
```

To get all of the document data from a collection, we can chain on the `.get()` method.

`.get()` returns a promise, which means we can resolve it either using a `.then()` callback or we can use the async-await syntax if we're executing our code within an async function.

Once our promises is resolved in one way or another, we get back what's known as a **snapshot**.

For a collection query that snapshot is going to consist of a number of individual documents. We can access them by saying `snapshot.docs`.

From each document, we can get the id as a separate property, and the rest of the data using the `.data()` method.

Here's what our entire query looks like:

```
const booksRef = firebase
  .firestore()
  .collection("books");

booksRef
  .get()
  .then((snapshot) => {
    const data = snapshot.docs.map((doc) => ({
      id: doc.id,
      ...doc.data(),
    }));
    console.log("All data in 'books' collection", data);
    // [ { id: 'glMeZvPpTN1Ah31sKcnj', title: 'The Great Gatsby' } ]
  });
```

Subscribing to a collection with .onSnapshot()

The `.get()` method simply returns all the data within our collection.

To leverage some of Firestore's realtime capabilities we can subscribe to a collection, which gives us the current value of the documents in that collection, whenever they are updated.

Instead of using the `.get()` method, which is for querying a single time, we use the `.onSnapshot()` method.

```
firebase
  .firestore()
  .collection("books")
  .onSnapshot((snapshot) => {
    const data = snapshot.docs.map((doc) => ({
      id: doc.id,
      ...doc.data(),
    }));
    console.log("All data in 'books' collection", data);
  });
```

In the code above, we're using what's known as method chaining instead of creating a separate variable to reference the collection.

What's powerful about using firestore is that we can chain a bunch of methods one after another, making for more declarative, readable code.

Within onSnapshot's callback, we get direct access to the snapshot of our collection, both now and whenever it's updated in the future. Try manually updating our one document and you'll see that `.onSnapshot()` is listening for any changes in this collection.

Difference between .get() and .onSnapshot()

The difference between the get and the snapshot methods is that get returns a promise, which needs to be resolved, and only then we get the snapshot data.

`.onSnapshot`, however, utilizes synchronous callback function, which gives us direct access to the snapshot.

This is important to keep in mind when it comes to these different methods--we have to know which of them return a promise and which are synchronous.

Unsubscribing from a collection with unsubscribe()

Note additionally that `.onSnapshot()` returns a function which we can use to unsubscribe and stop listening on a given collection.

This is important in cases where the user, for example, goes away from a given page where we're displaying a collection's data. Here's an example, using the library React were we are calling unsubscribe within the useEffect hook.

When we do so this is going to make sure that when our component is unmounted (no longer displayed within the context of our app) that we're no longer listening on the collection data that we're using in this component.

```
function App() {
  const [books, setBooks] = React.useState([]);

  React.useEffect(() => {
    const unsubscribe = firebase
      .firestore()
      .collection("books")
      .onSnapshot((snapshot) => {
        const data = snapshot.docs.map((doc) => ({
          id: doc.id,
          ...doc.data(),
        }));
        setBooks(data);
      });
  }, []);

  return books.map(book => <BookList key={book.id} book={book} />)
}
```

Getting Individual Documents with .doc()

When it comes to getting a document within a collection., the process is just the same as getting an entire collection: we need to first create a reference to that document, and then use the get method to grab it.

After that, however, we use the `.doc()` method chained on to the collection method. In order to create a reference, we need to grab this id from the database if it was auto generated. After that, we can chain on `.get()` and resolve the promise.

```
const bookRef = firebase
  .firestore()
  .collection("books")
  .doc("glMeZvPpTN1Ah31sKcnj");

bookRef.get().then((doc) => {
  if (!doc.exists) return;
  console.log("Document data:", doc.data());
  // Document data: { title: 'The Great Gatsby' }
});
```

Notice the conditional `if (!doc.exists) return;` in the code above.

Once we get the document back, it's essential to check to see whether it exists.

If we don't, there'll be an error in getting our document data. The way to check and see if our document exists is by saying, if `doc.exists`, which returns a true or false value.

If this expression returns false, we want to return from the function or maybe throw an error. If `doc.exists` is true, we can get the data from `doc.data`.

Adding document to a collection with .add()

Next, let's move on to changing data. The easiest way to add a new document to a collection is with the `.add()` method.

All you need to do is select a collection reference (with `.collection()`) and chain on `.add()`.

Going back to our definition of documents as being like JavaScript objects, we need to pass an object to the `.add()` method and specify all the fields we want to be on the document.

Let's say we want to add another book, 'Of Mice and Men':

```
firebase
  .firestore()
  .collection("books")
  .add({
    title: "Of Mice and Men",
  })
  .then((ref) => {
    console.log("Added doc with ID: ", ref.id);
    // Added doc with ID:  ZzhIgLqELaoE3eSsOazu
  });
```

The `.add` method returns a promise and from this resolved promise, we get back a reference to the created document, which gives us information such as the created id.

The `.add()` method auto generates an id for us. Note that we can't use this ref directly to get data. We can however pass the ref to the doc method to create another query.

Adding a document to a collection with .set()

Another way to add a document to a collection is with the `.set()` method.

Where set differs from add lies in the need to specify our own id upon adding the data.

This requires chaining on the `.doc()` method with the id that you want to use. Also, note how when the promise is resolved from `.set()`, we don't get a reference to the created document:

```
firebase
  .firestore()
  .collection("books")
  .doc("another book")
  .set({
    title: "War and Peace",
  })
  .then(() => {
    console.log("Document created");
  });
```

Additionally, when we use `.set()` with an existing document, it will, by default, overwrite that document.

If we want to merge, an old document with a new document instead of overwriting it, we need to pass an additional argument to `.set()` and provide the property `merge` set to true.

```
// use .set() to merge data with existing document, not overwrite

const bookRef = firebase
  .firestore()
  .collection("books")
  .doc("another book");
```

```
bookRef
  .set({
    author: "Lev Nikolaevich Tolstoy"
  }, { merge: true })
  .then(() => {
    console.log("Document merged");

    bookRef
      .get()
      .then(doc => {
      console.log("Merged document: ", doc.data());
      // Merged document:  { title: 'War and Peace', author: 'Lev Nikolaevich Tolstoy' }
    });
  });
```

Updating existing data with .update()

When it comes to updating data we use the update method, like `.add()` and `.set()` it returns a promise.

What's helpful about using `.update()` is that, unlike `.set()`, it won't overwrite the entire document. Also like `.set()`, we need to reference an individual document.

When you use `.update()`, it's important to use some error handling, such as the `.catch()` callback in the event that the document doesn't exist.

```
const bookRef = firebase.firestore().collection("books").doc("another book");

bookRef
  .update({
    year: 1869,
  })
  .then(() => {
    console.log("Document updated"); // Document updated
  })
  .catch((error) => {
    console.error("Error updating doc", error);
  });
```

Deleting data with .delete()

We can delete a given document collection by referencing it by it's id and executing the `.delete()` method, simple as that. It also returns a promise.

Here is a basic example of deleting a book with the id "another book":

```
firebase
  .firestore()
  .collection("books")
  .doc("another book")
  .delete()
  .then(() => console.log("Document deleted")) // Document deleted
  .catch((error) => console.error("Error deleting document", error));
```

> Note that the official Firestore documentation does not recommend to delete entire collections, only individual documents.

Working with Subcollections

Let's say that we made a misstep in creating our application, and instead of just adding books we also want to connect them to the users that made them. T

The way that we want to restructure the data is by making a collection called 'users' in the root of our database, and have 'books' be a subcollection of 'users'. This will allow users to have their own collections of books. How do we set that up?

References to the subcollection 'books' should look something like this:

```
const userBooksRef = firebase
  .firestore()
  .collection('users')
  .doc('user-id')
  .collection('books');
```

Note additionally that we can write this all within a single `.collection()` call using forward slashes.

The above code is equivalent to the follow, where the collection reference must have an odd number of segments. If not, Firestore will throw an error.

```
const userBooksRef = firebase
  .firestore()
  .collection('users/user-id/books');
```

To create the subcollection itself, with one document (another Steinbeck novel, 'East of Eden') run the following.

```
firebase.firestore().collection("users/user-1/books").add({
  title: "East of Eden",
});
```

Then, getting that newly created subcollection would look like the following based off of the user's ID.

```
firebase
  .firestore()
  .collection("users/user-1/books")
  .get()
  .then((snapshot) => {
    const data = snapshot.docs.map((doc) => ({
      id: doc.id,
      ...doc.data(),
    }));
    console.log(data);
    // [ { id: 'UO07aqpw13xvlMAfAvTF', title: 'East of Eden' } ]
  });
```

Useful methods for Firestore fields

There are some useful tools that we can grab from Firestore that enables us to work with our field values a little bit easier.

For example, we can generate a timestamp for whenever a given document is created or updated with the following helper from the `FieldValue` property.

We can of course create our own date values using JavaScript, but using a server timestamp lets us know exactly when data is changed or created from Firestore itself.

```
firebase
  .firestore()
  .collection("users")
  .doc("user-2")
  .set({
    created: firebase.firestore.FieldValue.serverTimestamp(),
  })
  .then(() => {
    console.log("Added user"); // Added user
  });
```

Additionally, say we have a field on a document which keeps track of a certain number, say the number of books a user has created. Whenever a user creates a new book we want to increment that by one.

An easy way to do this, instead of having to first make a `.get()` request, is to use another field value helper called `.increment()`:

```
const userRef = firebase.firestore().collection("users").doc("user-2");

userRef
  .set({
    count: firebase.firestore.FieldValue.increment(1),
  })
  .then(() => {
    console.log("Updated user");

    userRef.get().then((doc) => {
      console.log("Updated user data: ", doc.data());
    });
  });
```

Querying with .where()

What if we want to get data from our collections based on certain conditions? For example, say we want to get all of the users that have submitted one or more books?

We can write such a query with the help of the `.where()` method. First we reference a collection and then chain on `.where()`.

The where method takes three arguments--first, the field that we're searching on an operation, an operator, and then the value on which we want to filter our collection.

We can use any of the following operators and the fields we use can be primitive values as well as arrays.

`<`, `<=`, `==`, `>`, `>=`, `array-contains`, `in`, or `array-contains-any`

To fetch all the users who have submitted more than one book, we can use the following query.

After `.where()` we need to chain on `.get()`. Upon resolving our promise we get back what's known as a **querySnapshot**.

Just like getting a collection, we can iterate over the querySnapshot with `.map()` to get each documents id and data (fields):

```
firebase
  .firestore()
  .collection("users")
  .where("count", ">=", 1)
  .get()
  .then((querySnapshot) => {
    const data = querySnapshot.docs.map((doc) => ({
      id: doc.id,
      ...doc.data(),
    }));
    console.log("Users with > 1 book: ", data);
    // Users with > 1 book:  [ { id: 'user-1', count: 1 } ]
  });
```

> Note that you can chain on multiple .where() methods to create compound queries.

Limiting and ordering queries

Another method for effectively querying our collections is to limit them. Let's say we want to limit a given query to a certain amount of documents.

If we only want to return a few items from our query, we just need to add on the `.limit()` method, after a given reference.

If we wanted to do that through our query for fetching users that have submitted at least one book, it would look like the following.

```
const usersRef = firebase
  .firestore()
  .collection("users")
  .where("count", ">=", 1);

  usersRef.limit(3)
```

Another powerful feature is to order our queried data according to document fields using `.orderBy()`.

If we want to order our created users by when they were first made, we can use the `orderBy` method with the 'created' field as the first argument. For the second argument, we specify whether it should be in ascending or descending order.

To get all of the users ordered by when they were created from newest to oldest, we can execute the following query:

```
const usersRef = firebase
  .firestore()
  .collection("users")
  .where("count", ">=", 1);

  usersRef.orderBy("created", "desc").limit(3);
```

We can chain `.orderBy()` with `.limit()`. For this to work properly, `.limit()` should be called last and not before `.orderBy()`.