Writing Cloud Functions In Dart

Stefan Matthias Aust Follow
May 11 · 4 min read

Photo by chuttersnap on Unsplash

Can you write Firebase Cloud Functions in Dart? Yes you can and here is how.

Firebase Setup

Install the Firebase CLI if you haven't done so already:

```
npm install -g firebase-tools
```

Then, log into the Firebase Console and create a new project (or pick one you want to use). Because later in this tutorial, we will demonstrate a function that accesses the Cloud Firestore, create such a database in our prefered region and remember that region. You must deploy your functions in the same region if you don’t want to pay for the data traffic between different regions.

Back on the terminal, also log into Firebase here:

```
firebase login
```

Then, last but not least, setup the functions project in your current directory:

```
firebase init functions
```

Choose the Firebase project you created earlier and pick “JavaScript” as language and deselect “ESLint” (which we don’t need as we don’t write JavaScript on our own). Then allow `firebase` to download all dependencies via `npm` and wait while it downloads half of the internet.

This process created an empty `firebase.json` file and a directory called `functions` which contains a typical node project with a `package.json` file, a `node_modules` directory and an `index.js` file which contains a “hello world” example.

Open `index.js` and uncomment the example code. Then test the created function to make sure, your setup was successful, running:

```
npm run serve
```

The Firebase emulator should print the URL to access the cloud function. It will even automatically pick up changes to `index.js` while running which is quite nice. Eventually, stop the server.

Switch to Dart

Next, setup the Dart project.

Install the Dart SDK if you haven't done so already.

Create a new `pubspec.yaml` file in the `functions` directory with the following content:

```
name: functions
version: 1.0.0

environment:
 sdk: ‘>=2.8.0 3.0.0’

dependencies:
 firebase_functions_interop: ^1.0.2

dev_dependencies:
 build_runner: ^1.9.0
 build_node_compilers: ^0.2.4
```

Run `pub get` to download all dependencies.

Create a directory called `node` and create a file called `index.dart` inside that folder:

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

void main() {
 functions[‘moin’] = functions
   .region('europe-west3')
   .https.onRequest(greet);
}

void greet(ExpressHttpRequest request) {
 request.response
 ..writeln('Moin, moin!')
```

```
  ..close();
 }
```

Note: You must replace `europe-west3` with your region or omit that line.

Create another file called `build.yaml` with this content:

```
targets:
 $default:
 sources:
 — node/**
 — lib/**
 builders:
 build_node_compilers|entrypoint:
 options:
 compiler: dart2js
```

Add these lines to `.gitignore`:

```
.dart_tool/
.packages
```

Now run

```
pub run build_runner build — output node:build
```

This converts `node/index.dart` to `build/index.dart.js`. The convention, to use a `node` directory instead of the usual `bin` directory has been built into the NodeJS wrapper package that is used internally. It does the heavy-lifting of translating Dart to JavaScript. You can add more files to `lib` if you like.

Then change `package.json` and add:

```
   "main": "build/index.dart.js"
```

You can remove `index.js`. The JavaScript reign is about to end.

Next, run `npm run serve` again. When accessing the shown URL, the text “Moin, moin!” should be displayed. We successfully created a cloud function in Dart. Eventually, stop the server.

Deploying the Function

Replace the content of `firebase.json` with this content:

```
{
 “functions”: {
 “ignore”: [
 “.dart_tool”,
 “.packages”,
 “**/build/*.map”,
 “**/build/packages”,
 “build.yaml”,
 “node”,
 “node_modules”,
 “pubspec.*”
 ]
 }
}
```

Let’s now deploy the cloud function to Firebase:

```
npm run deploy
```

Development Workflow

To develop locally, use `watch` instead of `build` like so:

```
pub run build_runner watch — output node:build
```

and — in a second terminal — run:

```
npm run serve
```

Then change the Dart source and observe the `build_runner` to recompile the Dart source and the Firebase develop server to reload the JavaScript function. Now, you're ready to leave the JavaScript world behind and only use Dart for both your client and server development. Happy times!

Firestore Integration

Let's define a Firestore trigger which will automatically be called every time a new document is add to the `foo` collection. It creates a similar document in the `bar` collection that has the same `name` property as the document in `foo`.

```
void main() {
 functions['foo'] = functions
 .region('europe-west3')
 .firestore
 .document('/foo/{id}').onCreate(createHook);
}

Future<void> createHook(DocumentSnapshot snapshot, EventContext
context) async {
 final name = snapshot.data.getString('name');
 final data = DocumentData.fromMap({'name': name});
 await snapshot.firestore.collection('/bar').add(data);
}
```

Note: Make sure that the function runs in the same region as your datastore! Also notice that the store region name and the function region name might not be identical.

For the last time, deploy:

```
npm run deploy
```

Now go to Firebase console, open your Firestore database and test the trigger by creating a `foo/<id>` document with a `name` property and observe the creation of a new `bar/<id>` docment with a similar `name` property. `<id>` stands for the automatically created document identifier.

Conculsion

We're building Flutter apps for fun and profit and we love to use Firebase as a simple to use backend because it is very well integrated because Google provides ready to use packages most aspects of Firebase. For most application, we also have to customize the server part. Using the same programming language really helps to make this process much nicer.

We successfully created a custom backend for a Flutter app using only Dart.

Dart Programming Flutter Firebase Firebase Cloud Functions