Get started: write and deploy your first functions

To get started with Cloud Functions, try working through this tutorial, which starts with the required setup tasks and works through creating and deploying two related functions:

- `addMessage()`, which exposes a URL that accepts a text value and writes it to the Realtime Database.
- `makeUppercase()`, which triggers on the Realtime Database write and transforms the text to uppercase.

We've chosen Realtime Database and HTTP-triggered JavaScript functions for this sample, but you have many more options for triggering functions. See the how-to guides for other background triggers such as authentication events and analytics events, or, if you want a typed alternative to JavaScript, see Write Functions with TypeScript.

The following sections of this tutorial detail the steps required to build and deploy the sample. If you'd rather just run the code and inspect it, jump to Review complete sample code.

Set up Node.js and the Firebase CLI

You'll need a Node.js environment to write functions, and you'll need the Firebase CLI (which also requires Node.js and npm) to deploy functions to the Cloud Functions runtime.

Cloud Functions can run on either Node v6, or Node v8. If you have existing functions built on Node v6 you can continue to use that version (v6 is default). See Set runtime options to learn more.

For installing Node.js and npm, Node Version Manager is recommended. Once you have Node.js and npm installed, install the Firebase CLI via npm:

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

This installs the globally available firebase command. If the command fails, you may need to change npm permissions. To update to the latest version of `firebase-tools`, rerun the same command.

In many cases, new features and bug fixes are available only with the latest version of the Firebase CLI and the `firebase-functions` SDK. It's a good practice to frequently update both the Firebase CLI and the SDK with these commands inside the `functions` folder of your Firebase project:

```
npm install firebase-functions@latest firebase-admin@latest --save
npm install -g firebase-tools
```

Initialize Firebase SDK for Cloud Functions

When you initialize Firebase SDK for Cloud Functions, you create an empty project containing dependencies and some minimal sample code, and you choose either TypeScript or JavaScript for composing functions.

To initialize your project:

1. Run `firebase login` to log in via the browser and authenticate the firebase tool.
2. Go to your Firebase project directory.
3. Run `firebase init functions`. The tool gives you an option to install dependencies with npm. It is safe to decline if you want to manage dependencies in another way.
4. The tool gives you two options for language support:
    - Javascript
    - TypeScript. See Write Functions with TypeScript for more information.

    For this tutorial, select **JavaScript**.

After these commands complete successfully, your project structure looks like this:

```
myproject
 +- .firebaserc    # Hidden file that helps you quickly switch between
 |                 # projects with `firebase use`
 |
 +- firebase.json  # Describes properties for your project
 |
 +- functions/     # Directory containing all your functions code
      |
      +- .eslintrc.json  # Optional file containing rules for JavaScript
linting.
      |
      +- package.json  # npm package file describing your Cloud Functions
code
      |
      +- index.js      # main source file for your Cloud Functions code
      |
      +- node_modules/ # directory where your dependencies (declared in
                       # package.json) are installed
```

Firebase projects on the Spark plan can make only outbound requests to Google APIs. Requests to third-party APIs fail with an error. For more information about upgrading your project, see Pricing.

Import the required modules and initialize an app

After you have completed the setup tasks, you can open the source directory and start adding code as described in the following sections. For this sample, your project must import the Cloud Functions and Admin SDK modules using Node `require` statements. Add lines like the following to your `index.js` file:

```
// The Cloud Functions for Firebase SDK to create Cloud Functions and setup
triggers.
const functions = require('firebase-functions');

// The Firebase Admin SDK to access the Firebase Realtime Database.
```

```
const admin = require('firebase-admin');
admin.initializeApp();
```
index.js

These lines load the `firebase-functions` and `firebase-admin` modules, and initialize an `admin` app instance from which Realtime Database changes can be made. Wherever Admin SDK support is available, as it is for FCM, Authentication, and Firebase Realtime Database, it provides a powerful way to integrate Firebase using Cloud Functions.

The Firebase CLI automatically installs the Firebase and Firebase SDK for Cloud Functions Node modules when you initialize your project. To add 3rd party libraries to your project, you can modify `package.json` and run `npm install`. For more information, see Handle Dependencies.

Add the `addMessage()` function

For the `addMessage()` function, add these lines to `index.js`:

```
// Take the text parameter passed to this HTTP endpoint and insert it into
the
// Realtime Database under the path /messages/:pushId/original
exports.addMessage = functions.https.onRequest((req, res) => {
  // Grab the text parameter.
  const original = req.query.text;
  // Push the new message into the Realtime Database using the Firebase
Admin SDK.
  return admin.database().ref('/messages').push({original:
original}).then((snapshot) => {
    // Redirect with 303 SEE OTHER to the URL of the pushed object in the
Firebase console.
    return res.redirect(303, snapshot.ref.toString());
  });
});
```
index.js

The `addMessage()` function is an HTTP endpoint. Any request to the endpoint results in ExpressJS-style Request and Response objects passed to the `onRequest()` callback.

HTTP functions are synchronous (similar to callable functions), so you should send a response as quickly as possible and defer work using the Realtime Database. The `addMessage()` HTTP function passes a text value to the HTTP endpoint and inserts it into the Realtime Database under the path `/messages/:pushId/original` using the previously initialized `admin` app.

Deploy and execute `addMessage()`

To deploy and execute the `addMessage()` function, follow these steps:

1. Run this command to deploy your functions:

   ```
   $ firebase deploy --only functions
   ```

After you run this command, the Firebase CLI outputs the URL for any HTTP function endpoints. In your terminal, you should see a line like the following:

```
Function URL (addMessage): https://us-central1-
MY_PROJECT.cloudfunctions.net/addMessage
```

The URL contains your project ID as well as a region for the HTTP function. Though you don't need to worry about it now, some production HTTP functions should specify a location to minimize network latency.

2. Add a text query parameter to the `addMessage()` URL, and open it in a browser:

```
https://us-central1-
MY_PROJECT.cloudfunctions.net/addMessage?text=uppercaseme
```

The function executes and redirects the browser to the Firebase console at the database location where the text string is stored. You should see your text value displayed in the console.

After deploying and executing functions, you can view logs in the Firebase console.

Firebase projects on the Spark plan can make only outbound requests to Google APIs. Requests to third-party APIs fail with an error. For more information about upgrading your project, see Pricing.

Add the `makeUppercase()` function

For the `makeUppercase()` function, add these lines to `index.js`:

```
// Listens for new messages added to /messages/:pushId/original and creates
an
// uppercase version of the message to /messages/:pushId/uppercase
exports.makeUppercase =
functions.database.ref('/messages/{pushId}/original')
    .onCreate((snapshot, context) => {
      // Grab the current value of what was written to the Realtime
Database.
      const original = snapshot.val();
      console.log('Uppercasing', context.params.pushId, original);
      const uppercase = original.toUpperCase();
      // You must return a Promise when performing asynchronous tasks
inside a Functions such as
      // writing to the Firebase Realtime Database.
      // Setting an "uppercase" sibling in the Realtime Database returns a
Promise.
      return snapshot.ref.parent.child('uppercase').set(uppercase);
    });
```

index.js

The `makeUppercase()` function executes when the Realtime Database is written to. The `ref(path)` function defines the part of the database to listen on. For performance reasons, you should be as specific as possible.

Braces—for example, `{pushId}`—surround "parameters," wildcards that expose their matched data in the callback.

The Realtime Database triggers the `onWrite()` callback whenever data is written or updated on the given path.

Caution: Be careful to avoid any situation in which the function's result actually retriggers the function, creating an infinite loop — for example, a function triggered by writes to a specific Realtime Database path that terminates by writing to that same path. Also make sure to write functions in an idempotent way, so that they produce the same result if they run multiple times for a single event.

Event-driven functions such as Realtime Database events are asynchronous. The callback function should return either a `null`, an Object, or a Promise. If you do not return anything, the function times out, signaling an error, and is retried. See Sync, Async, and Promises.

Deploy and execute `makeUppercase()`

To complete the tutorial, deploy your functions again, and then execute `addMessage()` to trigger `makeUppercase()`.

1. Run this command to deploy your functions:

   ```
   $ firebase deploy --only functions
   ```

   If you encounter access errors such as "Unable to authorize access to project," try checking your project aliasing.

2. Using the `addMessage()` URL output by the CLI, add a text query parameter, and open it in a browser:

   ```
   https://us-central1-
   MY_PROJECT.cloudfunctions.net/addMessage?text=uppercasemetoo
   ```

   The function executes and redirects the browser to the Firebase console at the database location where the text string is stored. This write event triggers `makeUppercase()`, which writes an uppercase version of the string.

After deploying and executing functions, you can view logs in the Firebase console for Cloud Functions. If you need to delete functions in development or production, use the Firebase CLI.

Note: Deployment of functions from Firebase CLI is subject to rate limits. See Quota limits for Firebase CLI deployment.

Review complete sample code

Here's the completed `functions/index.js` containing the functions `addMessage()` and `makeUppercase()`. These functions allow you to pass a parameter to an HTTP endpoint that writes a value to the Realtime Database, and then transforms it by uppercasing all characters in the string.

```
// The Cloud Functions for Firebase SDK to create Cloud Functions and setup
triggers.
const functions = require('firebase-functions');
```

```
// The Firebase Admin SDK to access the Firebase Realtime Database.
const admin = require('firebase-admin');
admin.initializeApp();

// Take the text parameter passed to this HTTP endpoint and insert it into
the
// Realtime Database under the path /messages/:pushId/original
exports.addMessage = functions.https.onRequest((req, res) => {
  // Grab the text parameter.
  const original = req.query.text;
  // Push the new message into the Realtime Database using the Firebase
Admin SDK.
  return admin.database().ref('/messages').push({original:
original}).then((snapshot) => {
    // Redirect with 303 SEE OTHER to the URL of the pushed object in the
Firebase console.
    return res.redirect(303, snapshot.ref.toString());
  });
});

// Listens for new messages added to /messages/:pushId/original and creates
an
// uppercase version of the message to /messages/:pushId/uppercase
exports.makeUppercase =
functions.database.ref('/messages/{pushId}/original')
    .onCreate((snapshot, context) => {
      // Grab the current value of what was written to the Realtime
Database.
      const original = snapshot.val();
      console.log('Uppercasing', context.params.pushId, original);
      const uppercase = original.toUpperCase();
      // You must return a Promise when performing asynchronous tasks
inside a Functions such as
      // writing to the Firebase Realtime Database.
      // Setting an "uppercase" sibling in the Realtime Database returns a
Promise.
      return snapshot.ref.parent.child('uppercase').set(uppercase);
    });
```

index.js

Next steps

In this documentation, you can find more information on general concepts for Cloud Functions as well as guides for writing functions to handle the event types supported by Cloud Functions.

To learn more about Cloud Functions, you could also do the following:

- Read about use cases for Cloud Functions.
- Try the Cloud Functions codelab.
- Review and run code samples on GitHub.
- Review the API reference.

Video tutorial

You can learn more about Cloud Functions by watching video tutorials. In this video, you'll find detailed guidance on getting started with Cloud Functions, including Node.js and CLI setup.

Zuletzt aktualisiert: März 9, 2019