

A thick dark blue vertical bar runs down the left side of the slide. A blue arrow points from this bar towards the title. In the bottom left corner, there are several thin, curved lines in dark blue and light grey.

FIT2095 Semester 2 2019

Exploring Serverless Web Applications

Image Gallery Application

Norman Chen

20/10/2019

Contents

Serverless Computing	2
Advantages.....	3
Disadvantages	3
Serverless Functions	4
Using Google Cloud Functions	4
Selecting a project to use.....	4
Creating a cloud function.....	5
Testing our cloud function	9
Integrating Google Cloud Functions into an Angular application.....	10
Serverless Databases and Storage	12
Firebase Database.....	12
Similarities and differences between Firebase Realtime Database and Firebase Firestore.....	12
Using Google Firebase Firestore	13
Creating an instance.....	13
Adding collections and documents via the Firebase web interface	16
Integrating Firebase Firestore into an Angular application	19
Firebase Firestore CRUD operations.....	21
Read	21
Create.....	23
Update.....	24
Delete.....	26
Firebase Storage	28
Using Firebase Storage.....	28
Creating a bucket	28
Adding files via the Firebase web interface	29
Integrating Firebase Storage into an Angular application	30
Connecting Google Cloud Functions, Firebase Firestore and Firebase Storage	33
Further Exploration	34
Further Challenges	34
References	34

Serverless Computing

At a first glance, serverless computing might sound like servers are not required to run and serve our web applications. In fact, going “serverless” means that the responsibility of meeting all the operational requirements of our application and running various components of our application such as code execution, databases and disk storage is delegated solely to third-party cloud service providers. This means that the developer does not need to worry about the server anymore, hence leading to the term “serverless”. Serverless applications are applications which consist of various combinations of third-party cloud services which work together cohesively to deliver a seamless application experience to the end-user:

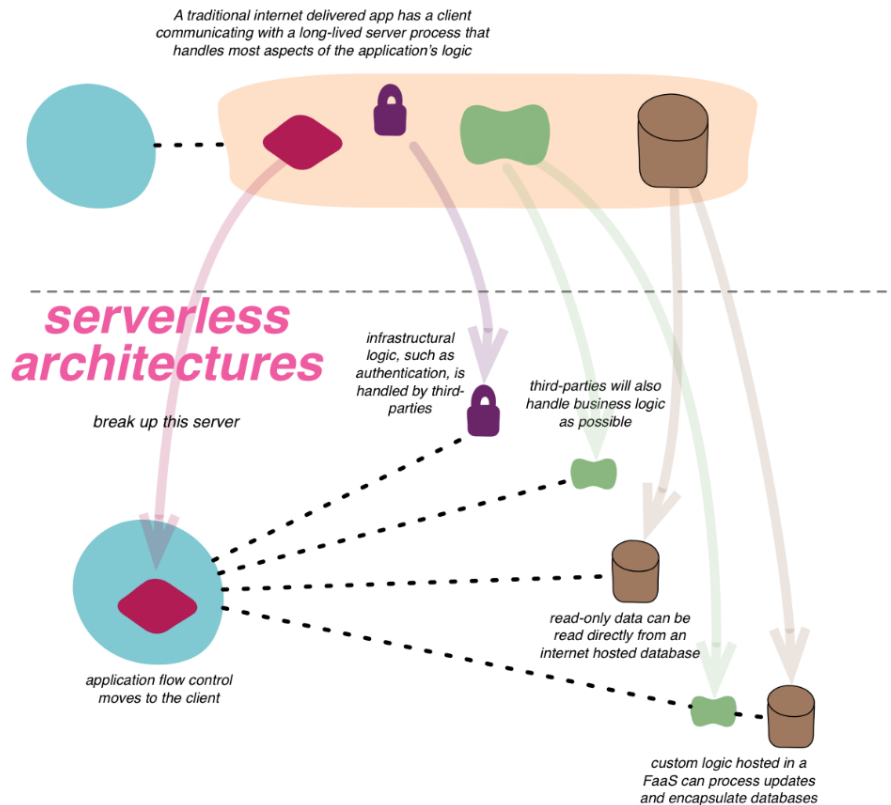


Figure 1 - Traditional Server Architecture vs. Serverless Architecture

The main difference between cloud and serverless computing is that serverless computing is an extension of the cloud computing concept. With serverless computing, application developers can quickly write and deploy their applications to the cloud without needing to purchase, manage, maintain or even know about the underlying hardware and network infrastructure required to keep their application continuously up and running. Typically, server resources are dynamically allocated on an on-demand basis, and developers will only pay for what is used. In the cloud computing model however, the user is still charged for running a virtual machine server even when the server is idle and not serving any user requests.

The major third-party cloud service providers are currently Amazon (AWS), Google (GCP) and Microsoft (Azure). These providers offer a lot of functionalities as services which our applications can consume. These cloud services are usually abbreviated with the suffix “-aaS”. For example, Function as a Service (FaaS), Database as a Service (DBaaS) and Storage as a Service (STaaS). Nowadays, competition between cloud service providers to attract developers is intense, as all of them are offering their own all-in-one mobile and web application platform with excellent integration between all their various cloud services being the major selling point. For example, Google’s Firebase (<https://firebase.google.com/>).

Advantages

1. Scalability and elasticity

Applications usually do not have consistent user traffic and resource usage loads throughout the duration of the application's lifetime. For example, Moodle has much more user traffic during swot vac compared to when the examination period has just ended. With serverless, we do not need to do anything at all to respond or adapt to user traffic fluctuations. Our web application's servers and resources will automatically scale up or down as required.

2. Pay only for what is used

With traditional server models, the user is billed according to how much time they were allocated a specific resource. This means that we will still need to pay the same amount whether we have fully utilised the resources allocated to us or not. Under the serverless model, users are only billed for actual usage of resources regardless of how much time has passed since the application was deployed. This is like changing from a mobile plan that is of a monthly fixed cost, to another mobile plan that charges per phone call, text message or byte of mobile data used.

3. Reusability

As cloud services can be accessed from anywhere, it is much easier to reuse existing functionalities across various software projects and web applications. With serverless, it is possible to split up a big monolithic (single piece) function into several smaller functions which we call microservices. For example, in an e-commerce website application, we may have a service that keeps track of the user's shopping cart, and we can reuse this functionality in other services such that based on their shopping cart contents, we can quickly create a recommendation service to recommend related and popular items, and a discounts calculator service that calculates special discounts the user is eligible for.

4. No need for server management and maintenance

As we do not own the servers, we do not need to worry about performing any form of maintenance such as applying critical software patches, firewall configuration, or making sure software packages are properly configured. This frees up a lot of time for developers, allowing them to focus and dedicate more time towards developing their application.

Disadvantages

1. Performance of serverless functions can be inconsistent

Serverless applications are usually served by one of the many handler instances (containers) on the cloud provider's servers and do not get billed by time but rather by usage. After some time spend idling, our application's containers will be deactivated until our application makes requests again as cloud providers need to optimise utilisation of their servers to ensure acceptable service performance for all customers. When a new request is made when the container has been deactivated, it will take longer to get a response as a new container has to boot up (known as "cold start").

2. Architectural complexity and vendor lock-in

Each cloud service provider has their own implementation and limitations. In order to use their services, we will need to adapt the architecture and design of our application to work around these limitations. Additionally, the more reliant we are on a specific cloud provider, the harder it is to migrate away from them. For example, billing rates could go up and we might then be forced to pay more than we would have if we were to use other providers.

3. Security and privacy concerns

Typically, a cloud service server will be concurrently shared between multiple customers. If there is a server misconfiguration, there is a chance that a customer's data could be exposed to other customers on the same server. Also, as the servers belong to the provider and not to us, we have no control over how the provider will store our data and have no visibility whether they are managing the privacy and security of our data correctly or not.

Serverless Functions

Serverless functions are pieces of code that can be called remotely and executed by a client or web application. Instead of making a single server responsible for all our HTTP REST requests like what we have previously done throughout the semester, we can have all these functions individually handled by different cloud servers when we use serverless function services. If any of our REST endpoints are experiencing heavy traffic, the performance of the rest of our REST endpoints will not be affected.

Currently, the main serverless function providers are Google Cloud Functions, AWS Lambda and Azure Functions. For the purposes of this guide, we will be using Google Cloud Functions.

Using Google Cloud Functions

Throughout this guide, we will be implementing a simple product image gallery application which is commonly found in e-commerce/online retail websites. We will be creating a useful, practical and non-trivial cloud function for this image gallery application which allows you to resize images by sending HTTP POST requests containing an image file of your choosing, together with values for the height and width (in pixels) the new image should have.

Having an image resizing function in this scenario is useful as we may be required to create thumbnails of a known, fixed size to meet certain website visual design requirements, to allow users to preview an image before viewing the full resolution image and most importantly, to save network bandwidth when small, low resolution image is more than sufficient for display.

Selecting a project to use

Navigate to <https://console.cloud.google.com/>.

1. You can use either an existing Google Cloud project or create a new one.

If you would like to select an existing project, click on the dropdown box on the navigation bar as shown below, and a pop-up menu window will appear. Select your desired project from the list.

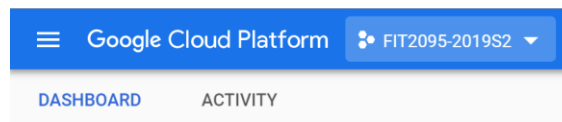
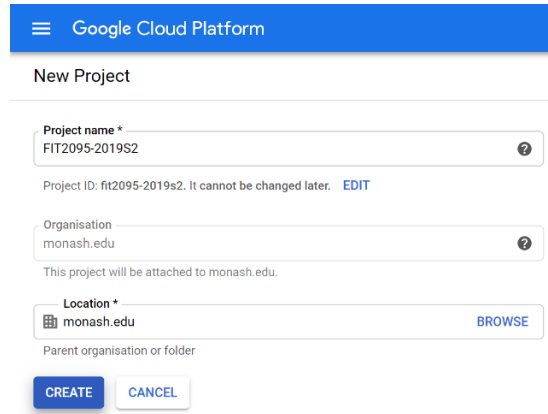


Figure 2 - GCP Navigation Bar

Otherwise, you can create a new project by clicking the “New Project” button at the top-right corner of the same pop-up menu. You will then see the following project creation screen.



Google Cloud Platform

New Project

Project name *
FIT2095-2019S2

Project ID: fit2095-2019s2. It cannot be changed later. [EDIT](#)

Organisation
monash.edu

This project will be attached to monash.edu.

Location *
monash.edu [BROWSE](#)

Parent organisation or folder

[CREATE](#) [CANCEL](#)

Figure 3 - Project Creation Screen

Creating a cloud function

1. Click on the menu icon on the left side of the navigation bar and navigate to the Cloud Functions area of GCP as shown below.

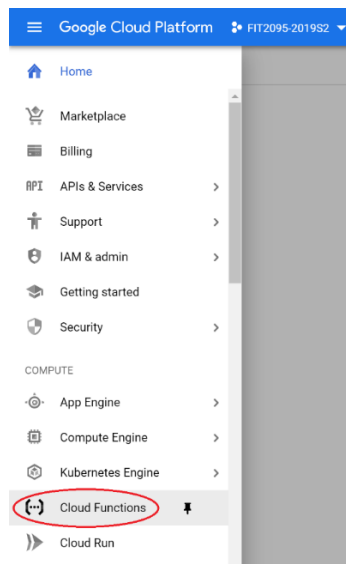


Figure 4- Navigating to Cloud Functions via GCP Main Menu

2. You will then see the following menu screen. Wait for it to finish loading, and then click “Create function”.

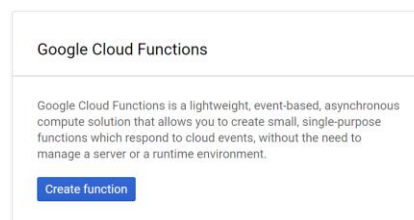



Figure 5 - Initial Create Function Screen

3. The function creation screen will appear as below. Make the following changes:

Name: *createThumbnail*
Memory allocated: *128 MB*
Function to execute: *resizeImage*

Cloud Functions | Create function

Name 
function-1

Memory allocated
256 MB

Trigger
HTTP

URL
https://us-central1-fit2095-2019s2.cloudfunctions.net/function-1

Authentication
☒ Allow unauthenticated invocations
Tick this if you are creating a public API or website.


This is a shortcut to assign the IAM Invoker role to the special identifier allUsers. You can use IAM to edit this setting after the function has been created.

Source code
☒ Inline editor
☐ ZIP upload
☐ ZIP from Cloud Storage
☐ Cloud Source repository

Runtime
Node.js 8

index.js package.json

```
1 /**  
2  * Responds to any HTTP request.  
3  *  
4  * @param {!express:Request} req HTTP request context.  
5  * @param {!express:Response} res HTTP response context.  
6  */  
7 exports.helloWorld = (req, res) => {  
8   let message = req.query.message || req.body.message || 'Hello World'  
9   res.status(200).send(message);  
10 };  
11
```

Function to execute 
helloWorld

[Environment variables, networking, timeouts and more](#)

Create Cancel

Figure 6 - Cloud Function Creation Screen

4. Replace the entire contents of the *index.js* tab of the online code editor section with the code below.

```
// Image files can be quite large and hence, must be sent over multiple packets as a multipart/form-data request.
// The 'busboy' library is used for parsing multipart/form-data requests as Node.js does not have this feature built-in.
const path = require('path');
const os = require('os');
const fs = require('fs');
const Busboy = require('busboy');
const sharp = require('sharp');

exports.resizeImage = (req, res) => {
  // This is needed to allow the endpoint to be called from other origins (domains) other than cloudfunctions.net (Google's domain for GCF).
  // For more information about CORS policies/security: https://en.wikipedia.org/wiki/Cross-origin_resource_sharing
  res.set('Access-Control-Allow-Origin', '*');

  // Only allow this function to be called with the POST request method.
  if (req.method !== 'POST') {
    return res.status(405).end();
  }

  const busboy = new Busboy({ headers: req.headers });
  let newDimensions = { // Default dimensions of output image if not specified.
    newWidth: 100,
    newHeight: 100
  };

  let filePromise;
  let imageFilepath;
  let imageMimeType;

  // This segment will process each field uploaded.
  busboy.on('field', function (fieldname, val, fieldNameTruncated, valTruncated, encoding, mimetype) {
    // Validate names of provided fields.
    if (!Object.keys(newDimensions).includes(fieldname)) {
      res.status(400).json({
        "error": "Invalid field specified. Valid field values are ${Object.keys(newDimensions).toString()}.",
        "received": fieldname
      });
      return;
    }

    newDimensions[fieldname] = parseInt(val);
  });

  // This segment will process each file uploaded.
  busboy.on('file', function (fieldname, file, filename, encoding, mimetype) {
    // Validate that only one file has been sent in the request.
    // As image MIME type would be undefined if we have not processed the file, validate by checking if MIME type has a value.
    if (imageMimeType) {
      res.status(400).json({
        "error": "Only one file can be specified at a single time."
      });
      return;
    }

    // Ignore files which are not images.
    if (!mimetype.includes("image")) {
      res.status(415).json({
        "error": "The uploaded file \"${filename}\" is not an image.",
        "received": mimetype
      });
      return;
    }

    imageMimeType = mimetype;

    // The image data will gradually come in multiple parts, so we write to a new file using a stream.
    // The file is saved into the function execution instance's in-memory file system.
    // There is a hard limit of 10MB on the capacity of the in-memory file system, per execution.
    imageFilepath = path.join(os.tmpdir(), filename);
    const writeStream = fs.createWriteStream(imageFilepath);
    file.pipe(writeStream);

    // File creation is asynchronous.
    // Return a Promise so we can wait for the new file to be fully saved to disk.
    filePromise = new Promise((resolve, reject) => {
      file.on('end', () => {
        writeStream.end();
      });
      writeStream.on('finish', resolve);
      writeStream.on('error', reject);
    });
  });
});
```



```
// This segment is executed when the file and all fields have been processed.
// Image resizing is only started when the image has been fully saved to disk.
busboy.on('finish', () => {
  filePromise
    .then(() => {
      return sharp(imageFilepath)
        .resize(newDimensions.newWidth, newDimensions.newHeight)
        .toBuffer();
    })
    .then((dataBuffer) => {
      res.set({ 'Content-Type': imageMimeType });
      res.status(200).send(dataBuffer);
    })
    .catch(err => {
      res.status(500).send({
        "error": err.message
      });
    });
});

busboy.end(req.rawBody);
};
```

You might notice that the structure of the code above is similar to how we have been using Express.js where we provided references to the callback function to be executed in our *app.get()*, *app.post()*, *app.put()* and *app.delete()* methods.

The key difference is that for any Google Cloud Function endpoint, the same function will serve as the callback (be executed) regardless of the request method (GET, POST, PUT, DELETE, etc.) of that request. Hence, we will need to check the *req.method* value to get the request method used by the user, and then use the appropriate logic to service that request.

Important: Note that our function name is *resizeImage* (see `exports.resizeImage` above). The function name needs to be the same as the name specified in the “Function to execute” field.

5. Replace the entire contents of the *package.json* tab of the online code editor section with the code below.

```
{
  "dependencies": {
    "busboy": "^0.3.1",
    "sharp": "^0.23.1"
  }
}
```

Google Cloud Functions allow us to use Node.js as the runtime environment for our cloud function, and this means we can use NPM packages too. Here, we only need to specify the dependencies section of our regular *package.json*.

6. Create!

Cloud Functions

Overview

CREATE FUNCTION

REFRESH

DELETE

COPY

Filter functions

Columns

<input type="checkbox"/>	Name ^	Region	Trigger	Runtime	Memory allocated	Executed function	Last deployed
<input checked="" type="checkbox"/>	createThumbnail	us-central1	HTTP	Node.js 8	128 MB	resizeImage	16/10/2019, 04:33

Figure 7 - Created Cloud Functions Screen

Testing our cloud function

1. Get the URL of the newly created cloud function.

Click on the name of the function (“createThumbnail”), then navigate to the “Trigger” tab and copy the URL.

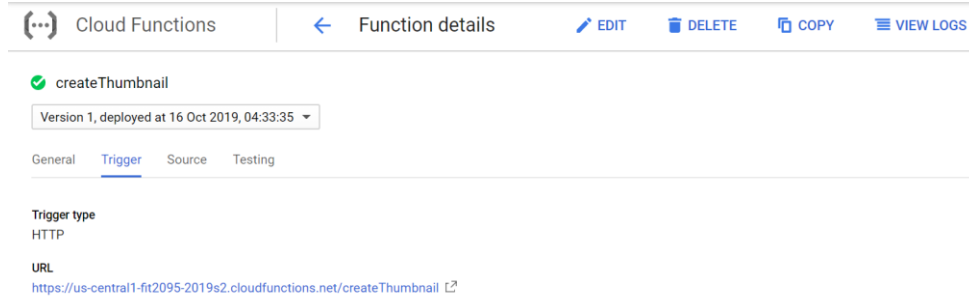


Figure 8 - Cloud Function Details - Trigger Tab

2. Use Postman to send a test request.

Set the request method to POST. Then, under the “Body” tab, select the “form-data” radio button.

Three parameters/keys will need to be specified: **newWidth**, **newHeight** and **file** (the name of the **file** key does not matter and you can use any other name that you wish. However, the key names for image width and height must be **newWidth** and **newHeight** respectively).

Set the type of both **newWidth** and **newHeight** keys to be “Text”, then set the **file** key to be of type “File”. Then, specify the **newWidth** and **newHeight** values in pixels, and then select the file for the **file** key using the file selector menu that appears when you click “Select Files”.

Click “Send” and verify that the cloud function is working correctly. The first ever request might take a bit longer compared to subsequent requests; this is normal.

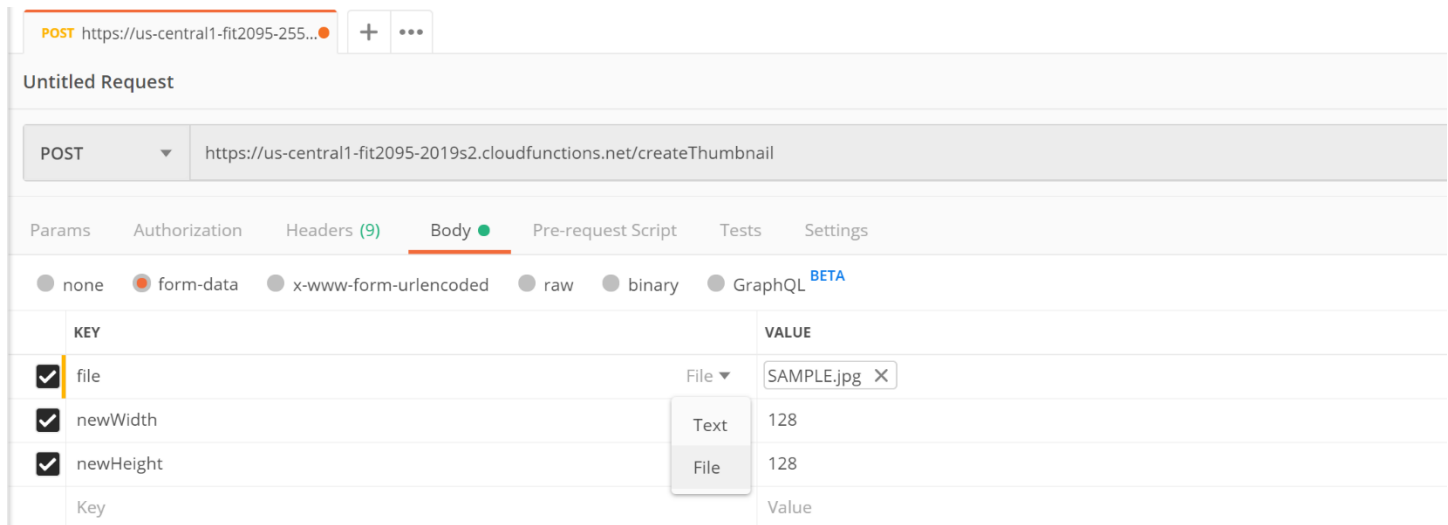


Figure 9 - Configuring Test Request in Postman

Integrating Google Cloud Functions into an Angular application

1. Import HttpClientModule and FormsModule in *app.module.ts*.

Add the following import statements:

```
import { HttpClientModule } from '@angular/common/http;  
import { FormsModule } from '@angular/forms';
```

Add the following line in the *imports* array under the *@NgModule* decorator:

```
HttpClientModule,  
FormsModule
```

2. Import HttpClient into *<your-component>.component.ts*, where *<your-component>* is the component name.

```
import { HttpClient } from '@angular/common/http';
```

3. Add the following TypeScript code snippet to the exported class of *<your-component>.component.ts*, where *<your-component>* is the component name. Replace the URL value with the endpoint URL you have just obtained earlier.

```
selectedFile: File;  
newWidth: number;  
newHeight: number;  
  
thumbnail: string | ArrayBuffer;  
  
constructor(private http: HttpClient) { }  
  
onSelectFile(files: FileList) {  
  this.selectedFile = files.item(0);  
}  
  
onSubmitForm() {  
  const formData = new FormData();  
  formData.append('file', this.selectedFile, this.selectedFile.name);  
  formData.append('newWidth', this.newWidth.toString());  
  formData.append('newHeight', this.newHeight.toString());  
  
  const url = "<REPLACE ME WITH THE CLOUD FUNCTION ENDPOINT URL>";  
  this.http.post<any>(url, formData, { responseType: 'blob' as 'json' }).subscribe(  
    (res) => {  
      // Convert image which is in binary form, into a format understandable by the <img> HTML element  
      const reader = new FileReader();  
      reader.readAsDataURL(res);  
      reader.onloadend = () => {  
        this.thumbnail = reader.result;  
      }  
    },  
    (err) => {  
      this.thumbnail = null;  
    }  
  );  
}
```

4. Add the following HTML code snippet to `<your-component>.component.html`, where `<your-component>` is the component name.

```
<form>
  <label>Select File:</label>
  <input type="file" name="file" (change)="onSelectFile($event.target.files)" required> <br />

  <label>New Width:</label>
  <input type="number" name="newWidth" [(ngModel)]="newWidth" required> <br />

  <label>New Height:</label>
  <input type="number" name="newHeight" [(ngModel)]="newHeight" required> <br />

  <button (click)="onSubmitForm()">Submit</button>
</form>

<div *ngIf="thumbnail">
  <h2>Image:</h2>
  <img *ngIf="thumbnail" [src]='thumbnail' style="border:1px solid black" />
</div>

<div *ngIf="thumbnail === null">
  <h2>Error!</h2>
</div>
```

Serverless Databases and Storage

With serverless databases, you do not need to worry about optimising database queries (*to a certain extent*) for performance, especially at a large scale. You also do not need to worry about writing your own SQL queries, as the queries are usually abstracted away and provided to you for use via a well-defined list of simple-to-use functions or API. Serverless databases automatically perform their own backups, indexing and internally store data in significantly efficient ways that have already been rigorously tested by the database service provider.

With serverless storage, you also do not need to worry about going over the storage space as you would if you were maintaining your own VM or server. The storage service provider stores your data in a container with scalable capacity, called a bucket. You may also create multiple buckets according to your web application's needs. File uploaded and stored this way may also be able to leverage the provider's content delivery network, where your uploaded files will be cached in several servers over multiple geographical locations around the world. This will lead to faster access and download times for your users all over the world.

For serverless databases services, we have options ranging from Google Firebase Firestore or Firebase Realtime Database, Amazon's DynamoDB to Microsoft's Azure CosmosDB. Several serverless storage options also exist, such as Google Firebase Storage, Amazon S3 (Simple Storage Service) and Microsoft Azure Storage. Throughout the rest of this guide, we will explore the use of serverless databases using Firebase Firestore and serverless storage using Firebase Storage. Note that a Firebase project is also a Google Cloud Platform project.

Firebase Database

In this section, we will extend our image gallery application by utilising Firebase Firestore to maintain a database of products and their respective images, where the images are already existing on the Internet (not uploaded by the user).

Similarities and differences between Firebase Realtime Database and Firebase Firestore

With Google Firebase, there are actually two DBaaS offerings available to users, namely Realtime Database and Firestore. Both Realtime Database and Firestore are document-based (NoSQL) databases like MongoDB and have generous free usage tiers which are more than enough for hobby projects.

The main difference between these two offerings is that Realtime Database stores all data in a single JSON object or tree, whereas Firestore uses the documents and collections model which we are already familiar with. Firestore is the successor of the Realtime Database and thus, offers users a better set of features, performance and scalability compared to Realtime Database. Most importantly, usage of Firestore is charged at a much lower rate compared to Realtime Database.

For more information regarding the two DBaaS offerings: <https://firebase.google.com/docs/firestore/rtdb-vs-firestore>

Using Google Firebase Firestore

Creating an instance

1. Create a Firebase project.

Navigate to <https://console.firebase.google.com/>, and add a new project. Notice that the list of project names is pre-populated with your existing Google Cloud Platform project(s). Select the GCP project that we have just created in the previous Selecting a project to use. Accept the terms and conditions, click “Continue”, leave the billing plan as Blaze and click “Confirm plan”. The Blaze plan automatically includes a generous amount of free monthly usage.

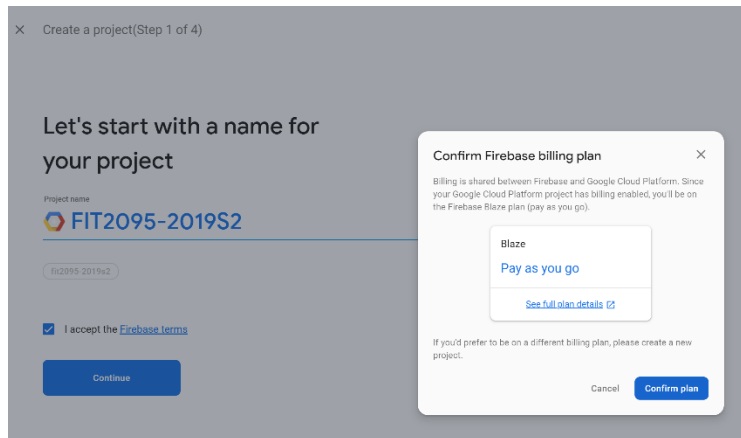


Figure 10 - Creating A New Firebase Project - Step 1

You will see a few more screens. One informs us that the same resources are shared between Firebase and Google Cloud Platform, and the following two are about setting up Google Analytics for our project. Click on “Continue” for the first two screens, and “Add Firebase” on the third.

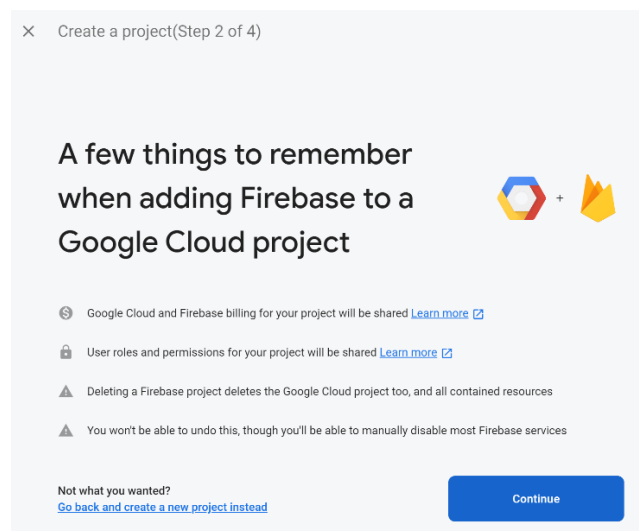


Figure 11 - Creating a New Firebase Project - Step 2

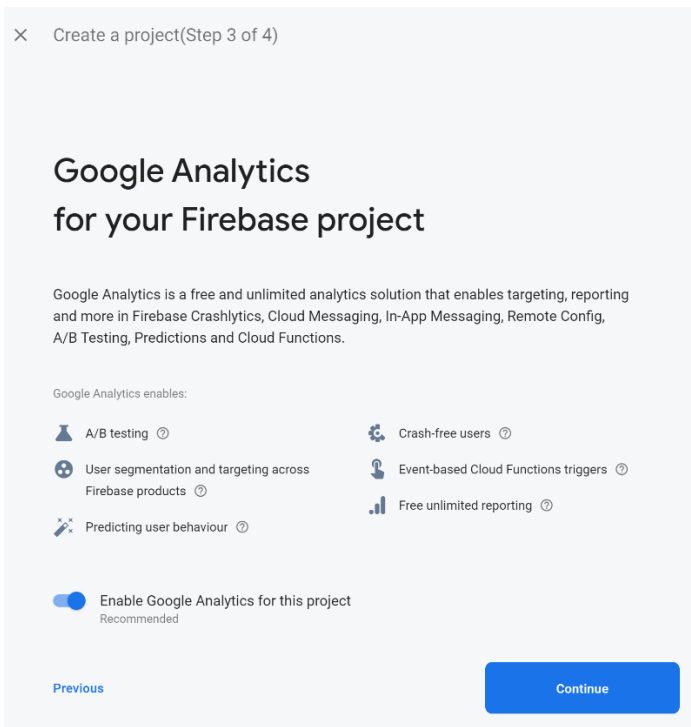


Figure 12 - Creating a New Firebase Project - Step 3

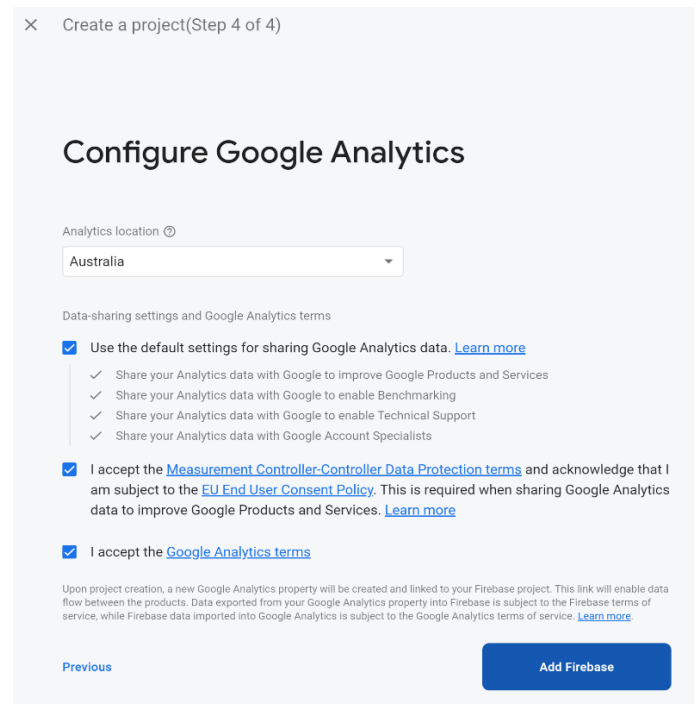


Figure 13 - Creating a New Firebase Project - Step 4

When the project has been created, click “Continue” and you will see the Firebase project console/overview page, which looks significantly different from the GCP console.

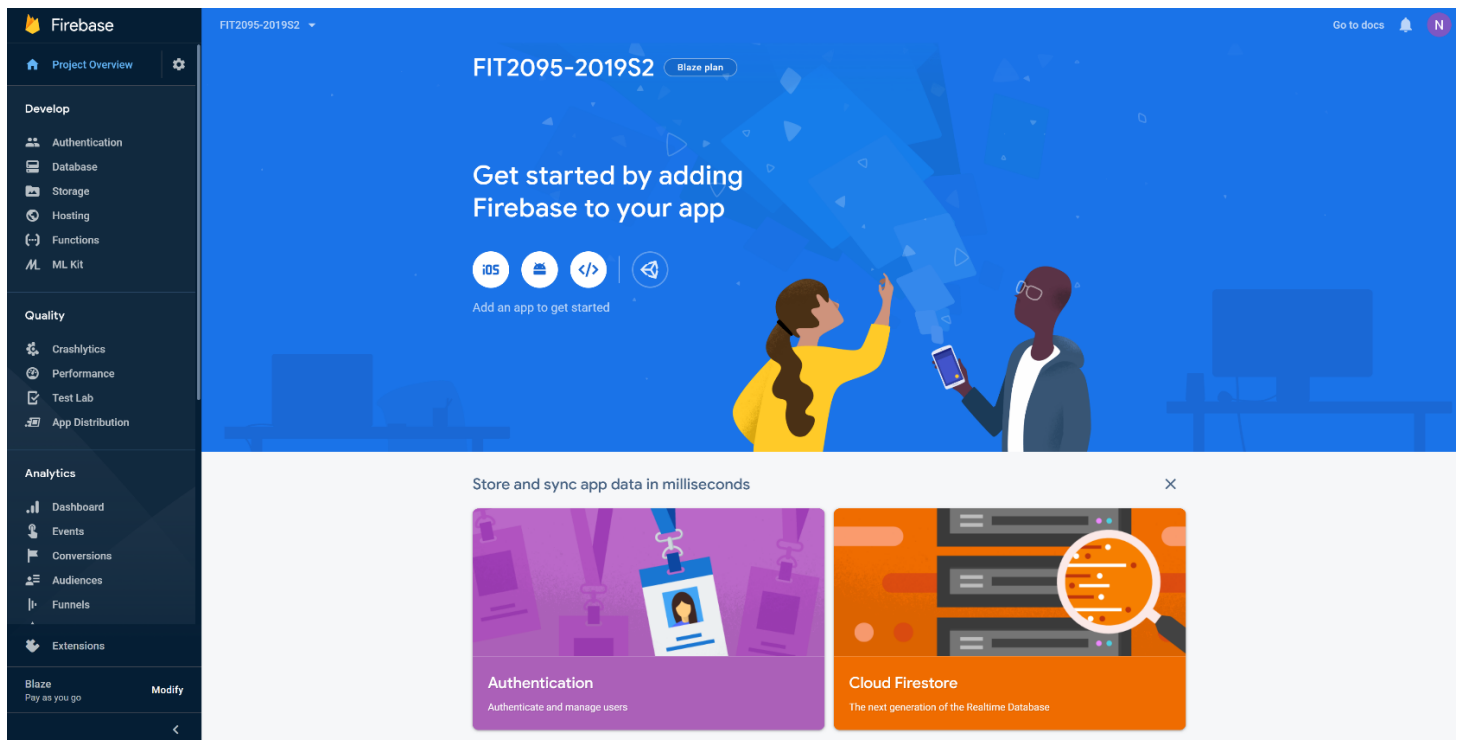


Figure 14 - Firebase Project Console/Overview Page

2. Create a Firestore database.

Under the “Development” tab in the navigation menu on the left side of the project console/overview page, select “Database”, and then click “Create database” under the *orange* Cloud Firestore section. Be careful **not** to accidentally create a new Realtime Database instead. For Firestore security and location settings, select the “Start in test mode” option, and leave the location as default. For the purposes of this guide only, we will be using test mode.

Warning: Test mode is not suitable for production use! Usually, you will need to restrict Firestore access only to authenticated users when in production by using the following line (or similar) in the database rules:

allow read, write: if request.auth != null;

For more information on security rules: <https://firebase.google.com/docs/rules/rules-language?authuser=0>

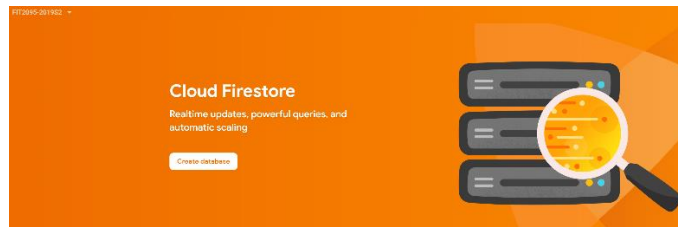


Figure 15 - Create Cloud Firestore

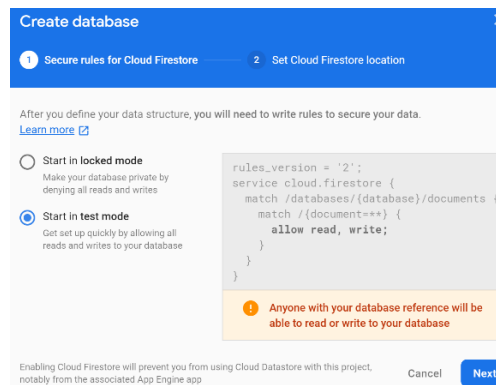


Figure 16 - Cloud Firestore Creation Options

Once the Firestore database has been created, you will see the following screen.

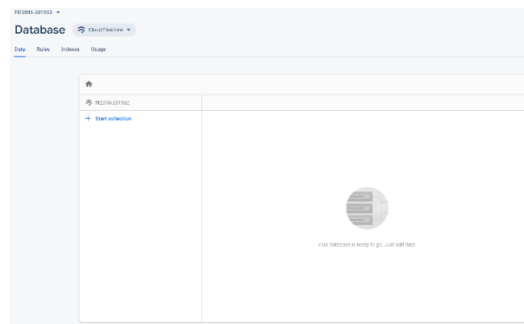


Figure 17 – Cloud Firestore Database Viewer

1. Create a products collection and document.

In the Firestore viewer (see Figure 17), click on “Start collection”, and name the new collection as “products”. Then, add the first product document with the **name**, **price**, **stock** and **description** fields as shown below in Figure 18. For each field, you may specify a *non-empty* value of your own choosing. Leave the Document ID field blank so that a random ID will be generated as the product document is saved.

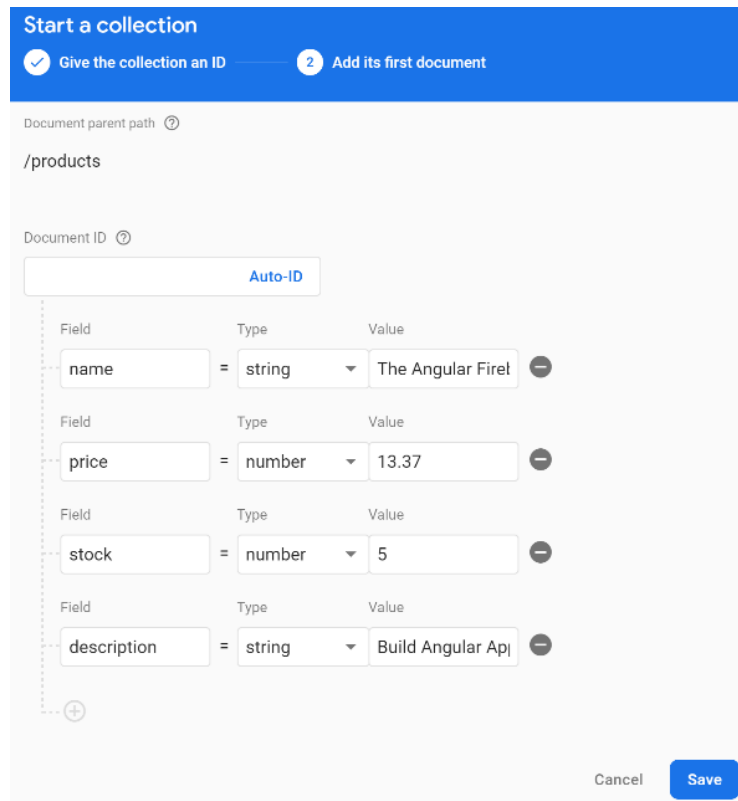
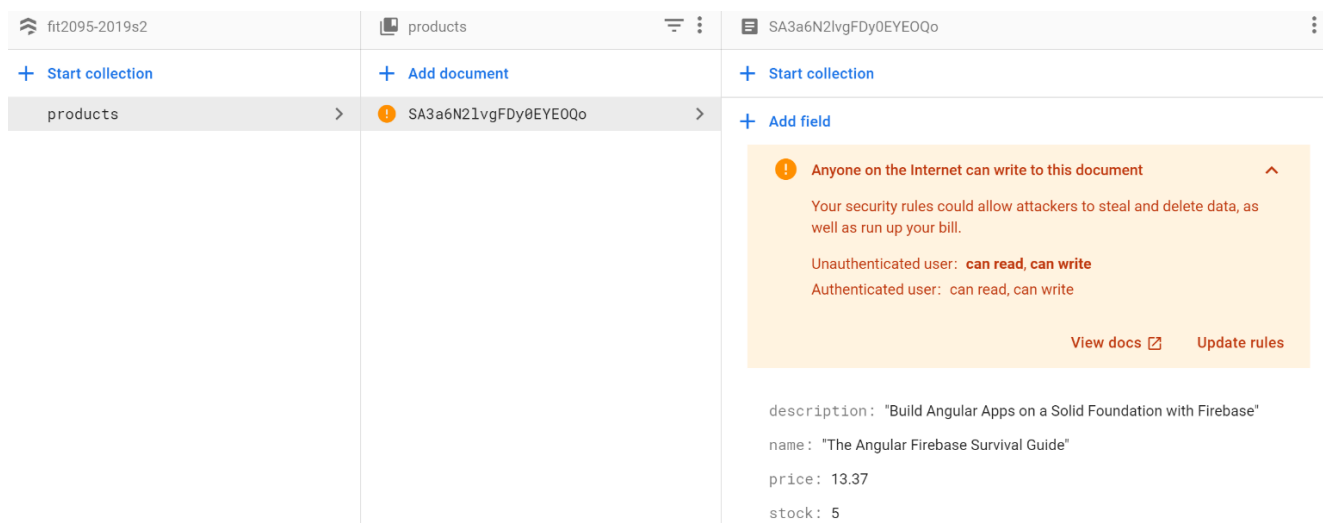


Figure 18 - Creating Products Collection and Document

Upon saving, you will see the following. Copy the ID of the newly created product document.



fit2095-2019s2	products	SA3a6N2lvvFDy0EYEOQo
+ Start collection	+ Add document	+ Start collection
products >	SA3a6N2lvvFDy0EYEOQo >	+ Add field

Anyone on the Internet can write to this document

Your security rules could allow attackers to steal and delete data, as well as run up your bill.

Unauthenticated user: **can read, can write**

Authenticated user: can read, can write

[View docs](#) [Update rules](#)

description: "Build Angular Apps on a Solid Foundation with Firebase"

name: "The Angular Firebase Survival Guide"

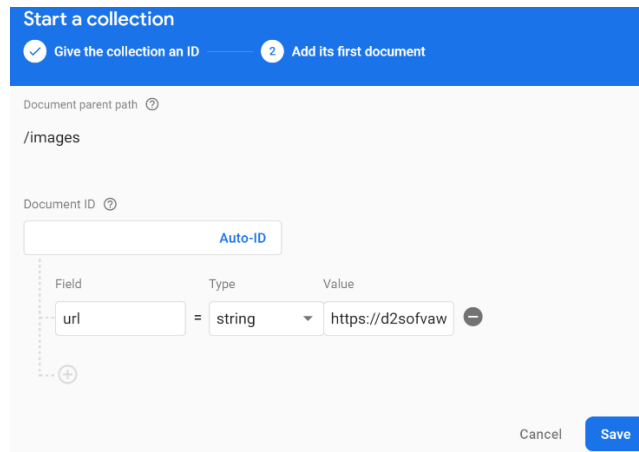
price: 13.37

stock: 5

Figure 19 - Products Collection and Document Created

2. Create an images collection and document.

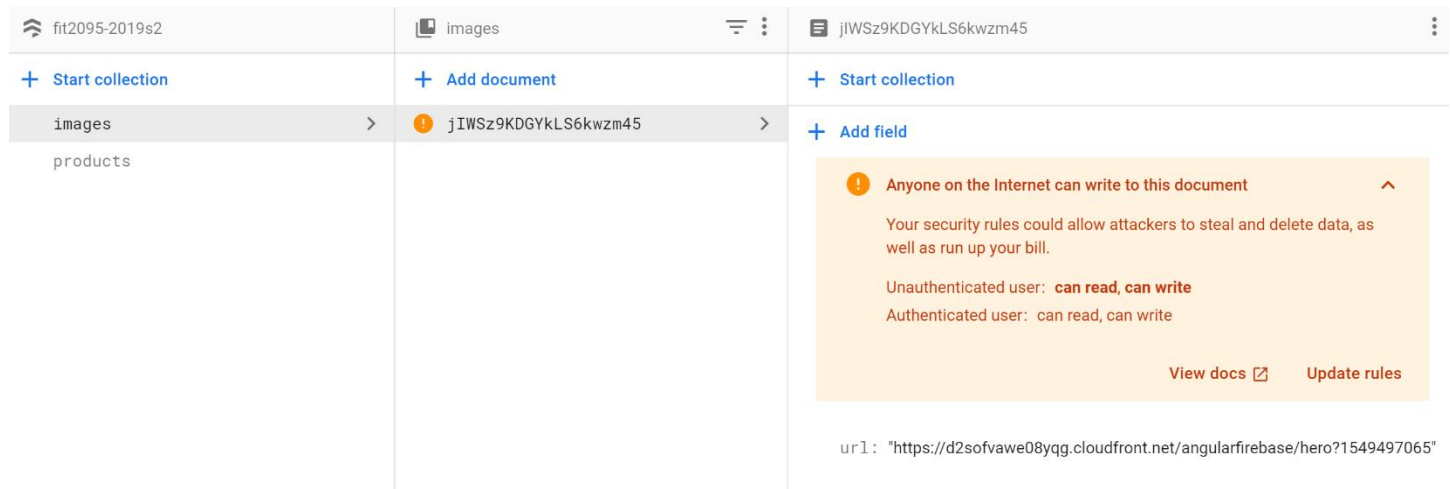
In the Firestore viewer (see Figure 17), click on “Start collection” again and name the new collection as “images”. Then, add the first image document with the **url** field as shown below in Figure 20. For the **url** field, specify a URL link to an online image of your own choosing. Leave the Document ID field blank so that a random ID will be generated as the image document is saved.



Field	Type	Value
url	string	https://d2sofvaw

Figure 20 - Creating Images Collection and Document

Upon saving, you will see the following. Copy the ID of the newly created image document.



fit2095-2019s2	images	jIWSz9KDGyKLS6kwzm45
+ Start collection	+ Add document	+ Start collection
images	jIWSz9KDGyKLS6kwzm45	+ Add field

url: "https://d2sofvawe08yqg.cloudfront.net/angularfirebase/hero?1549497065"

Figure 21 - Images Collection and Document Created

3. Link image document to product document.

Currently, it is not too useful if we do not have any relationships between products and images. We would like to allow products to have multiple images. To establish a relationship between documents, we can add references to a document, which will then point to another document.

Go back to the previously created product and click “Add field”. Name the new field as images and set its type to be of array. Then, add the first array element as a reference type, and type “images/” in the Document path field. Paste the copied value of the image ID from the previous step right after “images/” and click “Add”. References are simply the path of the document in Firestore. Be careful not to save reference paths as a string data type.

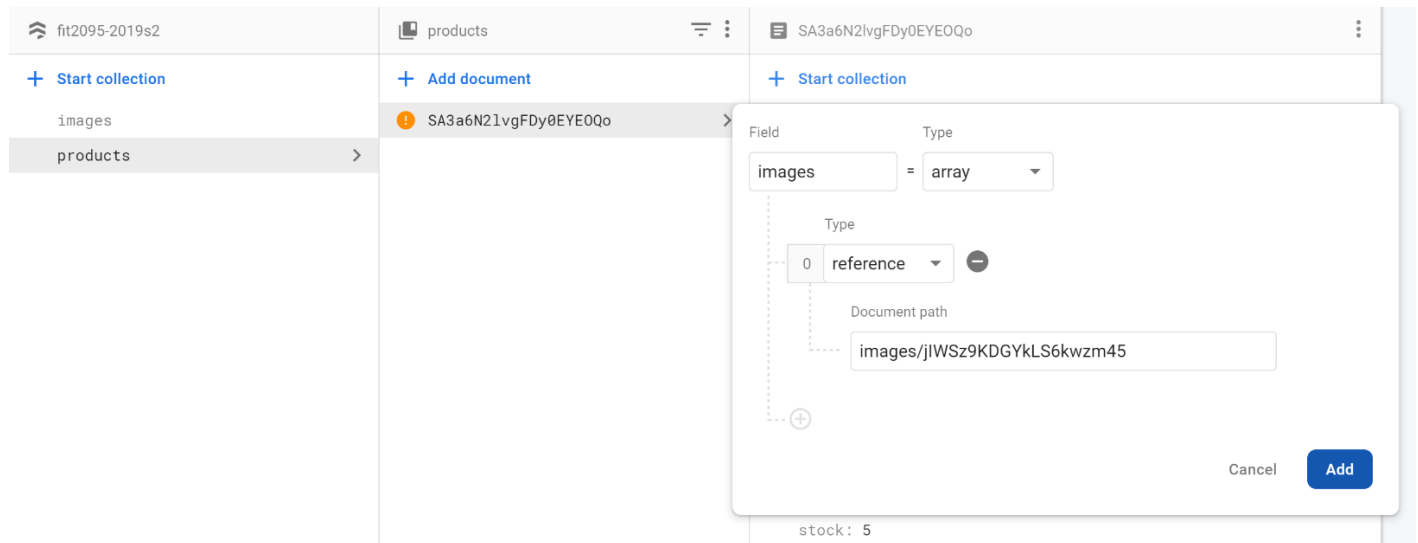


Figure 22 - Referencing Image Document from Product Document

Integrating Firebase Firestore into an Angular application

Angular has official support for Firebase. In the Angular project directory, run the following command:

```
npm install --save firebase @angular/fire
```

1. Obtain project configuration (access keys and ID strings) from the Firebase console.

Navigate to <https://console.firebase.google.com/>, and go to the project console page (see Figure 14) for the project we have created earlier. Click the “Project Overview” button in the navigation menu on the left side of the project console page. Under the heading for our project name, click “Add App”, and then select the “Web” option when prompted to select a platform. In the screen that appears, add an app nickname (for example, “ProductsGallery”), then click “Register app”.

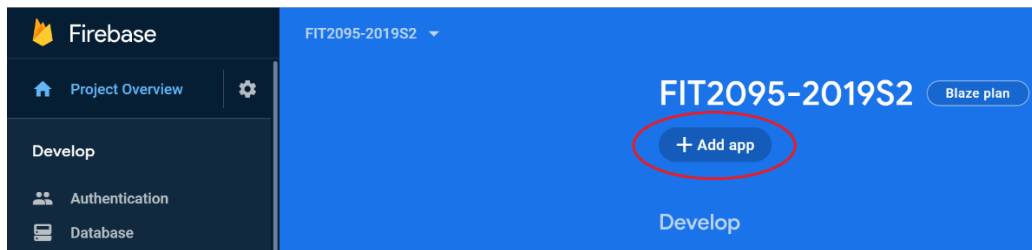


Figure 23 - Adding an App to Firebase Project

You will then be given credentials that can be used for our web application. Copy the following parts of the output which are highlighted in blue below:

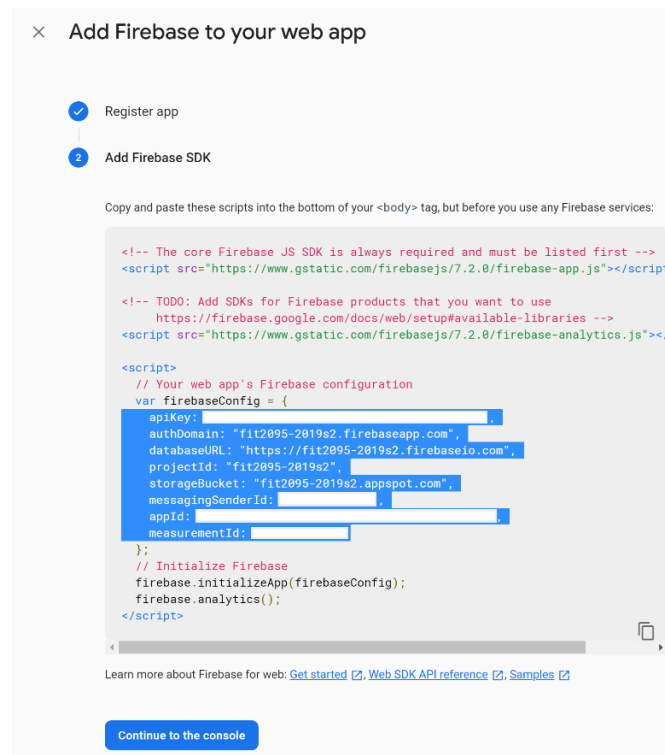


Figure 24 - Firebase Credentials for Web App

2. Paste the copied Firebase credentials into *src/environments/environments.ts*.

You need to add a new key called “firebase”, create an empty object as its value and then past the copied credentials inside that empty object as shown below:

```
export const environment = {  
  production: false,  
  firebase: {  
    apiKey: '<apiKey-value>',  
    authDomain: '<authDomain-value>',  
    databaseURL: '<databaseURL-value>',  
    projectId: '<projectId-value>',  
    storageBucket: '<storageBucket-value>',  
    messagingSenderId: '<messagingSenderId-value>',  
    appId: "<appId-value>",  
    measurementId: "<measurementId-value>"  
  }  
};
```

3. Import the AngularFireModule in *app.module.ts*.

Add the following import statements:

```
import { AngularFireModule } from '@angular/fire';  
import { AngularFirestoreModule } from '@angular/fire/firestore';  
import { environment } from '../environments/environment';
```

Add the following line in the *imports* array under the *@NgModule* decorator:

```
AngularFireModule.initializeApp(environment.firebase),  
AngularFirestoreModule.enablePersistence({ synchronizeTabs: true }) // Database stays available even when offline
```

1. Import AngularFirestore into `<your-component>.component.ts`, where `<your-component>` is the component name.

Add the following import statements:

```
import { AngularFirestore, DocumentChangeAction, DocumentReference, QuerySnapshot, QueryDocumentSnapshot } from '@angular/fire/firestore';
import { Observable } from 'rxjs';
import { map } from 'rxjs/operators';
```

2. Add the following TypeScript code snippet to the exported class of `<your-component>.component.ts`, where `<your-component>` is the component name. Delete the existing constructor before you paste. Make sure both HttpClient and AngularFirestore are injected into the constructor.

```
displayProducts: boolean = true;
products: any[];
images: any[];
constructor(private http: HttpClient, private database: AngularFirestore) {
  this.getAllProducts();
  this.getAllImages();
}
/* onSelectFile() AND onSubmitForm() CODE FROM PREVIOUS INTEGRATING GOOGLE CLOUD FUNCTIONS INTO AN ANGULAR APPLICATION SECTION GOES HERE. */
getAllProducts() {
  this.getCollectionObservable("products")
    .subscribe((products: any[]) => {
      this.products = products;

      // Note that in the Products collection, we defined the images attribute of a product as an array of references to image documents.
      // We need to resolve those image references in order to get the URL and ID of each image. Lets call these images as resolvedImages.
      // I.e. myProduct.images: [DocumentReference(), DocumentReference()], myProduct.resolvedImages: [Promise<{id, url}>, Promise<{id, url}>]
      // To resolve each image reference, we need to query the database and this will be an asynchronous operation.
      // We can get the value once it is ready by using Promises.
      // https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise
      // https://codeburst.io/javascript-map-vs-foreach-f38111822c0f
      this.products = products.map(product => {
        // Iterate through each image reference of the current product document.
        // A DocumentReference is simply a reference to a Firestore document.
        product.resolvedImages = product.images.map(async (image: DocumentReference) => {
          // Calling get() on a DocumentReference returns a promise for a DocumentSnapshot.
          // A DocumentSnapshot contains data read from a Firestore document.
          // Currently, if we have n images, we make n HTTP requests.
          // If we do this in production environments, we will very quickly exceed Firestore free usage quotas. Can you think of a better way?
          const documentSnapshot = await image.get();

          return {
            id: documentSnapshot.id,
            url: documentSnapshot.data().url
          };
        });
        return product;
      });
    });
}

getAllImages() {
  this.getCollectionObservable("images")
    .subscribe((images: any[]) => {
      this.images = images;
    });
}

// Helper function to promote code reuse.
getCollectionObservable(collectionName: string): Observable<any> {
  // When documents are loaded, changes are made to the internal state of our local Firestore database (in this case, "added").
  // These change actions are represented as DocumentChangeAction objects and contain info about what data has been added for a document.
  // Includes information like document ID, DocumentReferences to other documents and other important metadata.
  // Using snapshotChanges() allows us to subscribe to real-time updates of our data. With get(), data does not update once we have fetched them.
  return this.database
    .collection(collectionName)
    .snapshotChanges()
    .pipe(
      map((actions: DocumentChangeAction<any>[]) => actions.map(action => {
        const id = action.payload.doc.id;
        const data = action.payload.doc.data();
        return { id, ...data };
      }))
    );
}

onToggle() {
  this.displayProducts = !this.displayProducts;
}
```

3. Add the following HTML code snippet to `<your-component>.component.html`, where `<your-component>` is the component name.

```
<hr />
<button (click)="onToggle()">Toggle Product/Image View</button>
<div *ngIf="displayProducts === true">
  <h1>Products:</h1>
  <button (click)="getAllProducts()">Refresh Table</button>
  <table>
    <tr>
      <th>ID</th>
      <th>Name</th>
      <th>Price</th>
      <th>Stock</th>
      <th>Description</th>
      <th>Images</th>
    </tr>
    <tr *ngFor="let product of products">
      <td>{{product.id}}</td>
      <td>{{product.name}}</td>
      <td>{{product.price | currency:"AUD"}}</td>
      <td>{{product.stock}}</td>
      <td>{{product.description}}</td>
      <td>
        <!-- As each resolved image's ID and URL are obtained asynchronously, use the async pipe to wait for the URL value to arrive. -->
        <!-- To ensure we wait before accessing the ID and URL value, use the "?" operator before ".id" and ".url" -->
        <span *ngFor="let resolvedImage of product.resolvedImages">
           &nbsp;
          <span class="image-tooltip">{{ (resolvedImage | async)?.id }}</span>
        </span>
      </td>
    </tr>
  </table>
</div>

<div *ngIf="displayProducts === false">
  <h1>Images:</h1>
  <button (click)="getAllImages()">Refresh Table</button>
  <table>
    <tr>
      <th>ID</th>
      <th>URL</th>
      <th>Preview</th>
    </tr>
    <tr *ngFor="let image of images">
      <td>{{image.id}}</td>
      <td>{{image.url}}</td>
      <td>
        
      </td>
    </tr>
  </table>
</div>
<hr />
```

4. Add the following CSS code snippet to `<your-component>.component.css`, where `<your-component>` is the component name.

```
/* https://www.w3schools.com/css/css_table.asp */
table {
  border-collapse: collapse;
}

table, th, td {
  border: 1px solid black;
}

tr:nth-child(even) {
  background-color: #f2f2f2;
}

/* https://www.w3schools.com/howto/howto_css_two_columns.asp */
/* Clear floats after the columns */
.row:after {
  content: "";
  display: table;
  clear: both;
}

.column {
  float: left;
  width: 50%;
}

form { display: table; }
.form-row { display: table-row; }
label { display: table-cell; padding-right: 5px; }
input { display: table-cell; margin-bottom: 5px; }
select { display: table-cell; margin-bottom: 5px; }
button { display: table-cell; }
```

Create

1. Add the following TypeScript code snippet to the exported class of `<your-component>.component.ts`, where `<your-component>` is the component name.

```
operationMode: string = "create";

newProductName: string = "";
newProductDescription: string = "";
newProductPrice: number = null;
newProductStock: number = null;

createProduct() {
  this.database.collection("products")
    .add({
      name: this.newProductName,
      description: this.newProductDescription,
      price: this.newProductPrice,
      stock: this.newProductStock,
      images: []
    })
    .catch(error => {
      console.error("Error creating product: ", error);
    });
}

newImageUrl: string = "";

createImage() {
  this.database.collection("images")
    .add({
      url: this.newImageUrl
    })
    .catch(error => {
      console.error("Error creating image: ", error);
    });
}
```

2. Add the following HTML code snippet to `<your-component>.component.html`, where `<your-component>` is the component name.

```
<select id="modeSelector" [(ngModel)]="operationMode">
  <option value="create">Create Documents</option>
</select>

<div *ngIf="operationMode === 'create'" class="row">
  <div class="column container">
    <h2>Create Product</h2>
    <form>
      <div class="form-row">
        <label>Name</label>
        <input type="text" name="newProductName" [(ngModel)]="newProductName" required><br />
      </div>
      <div class="form-row">
        <label>Description</label>
        <input type="text" name="newProductDescription" [(ngModel)]="newProductDescription" required><br />
      </div>
      <div class="form-row">
        <label>Price</label>
        <input type="number" name="newProductPrice" [(ngModel)]="newProductPrice" required><br />
      </div>
      <div class="form-row">
        <label>Stock</label>
        <input type="number" name="newProductStock" [(ngModel)]="newProductStock" min="0" step="1" required><br />
      </div>
      <div class="form-row">
        <label></label>
        <button (click)="createProduct()">Create Product</button><br />
      </div>
    </form>
  </div>

  <div class="column container">
    <h2>Create Image</h2>
    <form>
      <div class="form-row">
        <label>URL</label>
        <input type="text" name="newImageUrl" [(ngModel)]="newImageUrl" required><br />
      </div>
      <div class="form-row">
        <label></label>
        <button (click)="createImage()">Create Image</button><br />
      </div>
    </form>
  </div>
</div>
```


Update

1. Add the following TypeScript code snippet to the exported class of `<your-component>.component.ts`, where `<your-component>` is the component name.

```
productIdToUpdate: string = "";
nameOfProductToUpdate: string = "";
descriptionOfProductToUpdate: string = "";
priceOfProductToUpdate: string = "";
stockOfProductToUpdate: string = "";
selectedImageIds: string[] = [];

updateProduct() {
  this.database
    .collection("products")
    .doc(this.productIdToUpdate)
    .update({
      name: this.nameOfProductToUpdate,
      description: this.descriptionOfProductToUpdate,
      price: this.priceOfProductToUpdate,
      stock: this.stockOfProductToUpdate,
      images: this.selectedImageIds.map(imageId => {
        return this.database.collection("images").doc(imageId).ref;
      }) // Convert each of the string IDs provided by the selection box into DocumentReferences.
    })
    .catch(error => {
      console.error("Error updating product: ", error);
    });
}

imageIdToUpdate: string = "";
urlOfImageToUpdate: string = "";

updateImage() {
  this.database
    .collection("images")
    .doc(this.imageIdToUpdate)
    .update({
      url: this.urlOfImageToUpdate
    })
    .catch(error => {
      console.error("Error updating image: ", error);
    });
}

onSelectProductToUpdate() {
  let matchingProducts = this.products.filter(product => {
    return product.id === this.productIdToUpdate;
  }); // The matchingProducts array will contain at most 1 element as product IDs are unique.

  this.nameOfProductToUpdate = matchingProducts[0].name;
  this.descriptionOfProductToUpdate = matchingProducts[0].description;
  this.priceOfProductToUpdate = matchingProducts[0].price;
  this.stockOfProductToUpdate = matchingProducts[0].stock;
  this.selectedImageIds = matchingProducts[0].images.map(image => {
    return image.id;
  }); // Convert each of the DocumentReferences from Firestore into string IDs the selection box can understand.
}

onSelectImageToUpdate() {
  let matchingImages = this.images.filter(image => {
    return image.id === this.imageIdToUpdate
  }); // The matchingImages array will contain at most 1 element as image IDs are unique.

  this.urlOfImageToUpdate = matchingImages[0].url;
}
```

2. Add the following HTML code snippet to `<your-component>.component.html`, where `<your-component>` is the component name.

```
<div *ngIf="operationMode === 'update'" class="row">
  <div class="column container">
    <h2>Update Product</h2>
    <form>
      <div class="form-row">
        <label>Select Product To Update</label>
        <select name="productUpdateSelector" [(ngModel)]="productIdToUpdate"
          (change)="onSelectProductToUpdate($event.target.value)">
          <option *ngFor="let product of products" value="{{product.id}}">{{product.id}}</option>
        </select>
      </div>
      <div class="form-row">
        <label>Name</label>
        <input type="text" name="nameOfProductToUpdate" [(ngModel)]="nameOfProductToUpdate" required<br />
      </div>
      <div class="form-row">
        <label>Description</label>
        <input type="text" name="descriptionOfProductToUpdate" [(ngModel)]="descriptionOfProductToUpdate"
          required<br />
      </div>
      <div class="form-row">
        <label>Price</label>
        <input type="number" name="priceOfProductToUpdate" [(ngModel)]="priceOfProductToUpdate" required<br />
      </div>
      <div class="form-row">
        <label>Stock</label>
        <input type="number" name="stockOfProductToUpdate" [(ngModel)]="stockOfProductToUpdate" min="0" step="1"
          required<br />
      </div>
      <div class="form-row">
        <label>Images (Ctrl/Cmd + Click to select multiple)</label>
        <select multiple name="productImagesUpdateSelector" [(ngModel)]="selectedImageIds">
          <option *ngFor="let image of images" value="{{image.id}}">{{image.id}}
        </option>
        </select><br />
      </div>
      <div class="form-row">
        <label></label>
        <button (click)="updateProduct()">Update Product</button><br />
      </div>
    </form>
  </div>

  <div class="column container">
    <h2>Update Image</h2>
    <form>
      <div class="form-row">
        <label>Select Image To Update</label>
        <select name="imageSelector" [(ngModel)]="imageIdToUpdate"
          (change)="onSelectImageToUpdate($event.target.value)">
          <option *ngFor="let image of images" value="{{image.id}}">{{image.id}}</option>
        </select>
      </div>
      <div class="form-row">
        <label>URL</label>
        <input type="text" name="urlOfImageToUpdate" [(ngModel)]="urlOfImageToUpdate" required<br />
      </div>
      <div class="form-row">
        <label></label>
        <button (click)="updateImage()">Update Image</button><br />
      </div>
    </form>
  </div>
</div>
```

Also, add the following `<option>` element as a child element of the `<select>` element with id `modeSelector`.

```
<option value="update">Update Documents</option>
```

Delete

1. Add the following TypeScript code snippet to the exported class of `<your-component>.component.ts`, where `<your-component>` is the component name.

```
productIdToDelete: string = "";

deleteProduct() {
  this.database.collection("products")
    .doc(this.productIdToDelete)
    .delete()
    .catch(function (error) {
      console.error("Error deleting product: ", error);
    });
}

imageIdToDelete: string = "";

deleteImage() {
  let imageRef = this.database.collection("images")
    .doc(this.imageIdToDelete)
    .ref;

  // Remove the reference to the image from the product that refers to it.
  this.database.collection("products", ref => ref.where('images', 'array-contains', imageRef))
    .get() // This is the second way to perform GET requests with Firestore, other than with snapshotChanges().
    .subscribe((querySnapshot: QuerySnapshot<any>) => {
      let numProductsToUpdate: number = querySnapshot.size;

      if (numProductsToUpdate === 0) {
        // If we have no affected products, delete the image document right away and return.
        imageRef
          .delete()
          .catch(function (error) {
            console.error("Error deleting image: ", error);
          });

        return;
      }

      // A QuerySnapshot contains QueryDocumentSnapshots from a Firestore query.
      // QueryDocumentSnapshots are similar to DocumentSnapshots.
      querySnapshot.forEach((matchingProduct: QueryDocumentSnapshot<any>) => {
        matchingProduct.ref
          .update({
            images: matchingProduct.data().images.filter((ref: DocumentReference) => {
              return ref.id !== imageRef.id;
            }) // Remove a document's image reference if its ID is the same as the ID of the image being deleted.
          })
          .then(() => {
            numProductsToUpdate--;
            // After we updated all product documents, delete the image.
            if (numProductsToUpdate == 0) {
              imageRef
                .delete()
                .catch(function (error) {
                  console.error("Error deleting image: ", error);
                });
            }
          })
          .catch(error => {
            console.log("Error deleting product image reference: ", error);
          });
      });
    });
}
```

2. Add the following HTML code snippet to `<your-component>.component.html`, where `<your-component>` is the component name.

```
<div *ngIf="operationMode === 'delete'" class="row">
  <div class="column container">
    <h2>Delete Product</h2>
    <form>
      <div class="form-row">
        <label>Select Product To Delete</label>
        <select name="productDeleteSelector" [(ngModel)]="productIdToDelete">
          <option *ngFor="let product of products" value="{{product.id}}">{{product.id}}</option>
        </select>
      </div>

      <div class="form-row">
        <label></label>
        <button (click)="deleteProduct()">Delete Product</button>
        <br />
      </div>
    </form>
  </div>

  <div class="column container">
    <h2>Delete Image</h2>
    <form>
      <div class="form-row">
        <label>Select Image To Delete</label>
        <select name="imageDeleteSelector" [(ngModel)]="imageIdToDelete">
          <option *ngFor="let image of images" value="{{image.id}}">{{image.id}}</option>
        </select>
      </div>

      <div class="form-row">
        <label></label>
        <button (click)="deleteImage()">Delete Image</button>
        <br />
      </div>
    </form>
  </div>
</div>
```

Also, add the following `<option>` element as a child element of the `<select>` element with id `modeSelector`.

```
<option value="delete">Delete Documents</option>
```

Firestore Storage

In this section, we will add the ability to allow users to upload their own product images to be displayed by the image gallery application by utilising Firestore Storage to store the uploaded images.

Using Firestore Storage

Creating a bucket

Under the “Development” tab in the navigation menu on the left side of the project console/overview page, select “Storage”, and then click “Get started”. You will see a pop-up screen regarding secure rules for Cloud Storage, and the data centre location where our files will be stored at. Proceed and create the bucket.

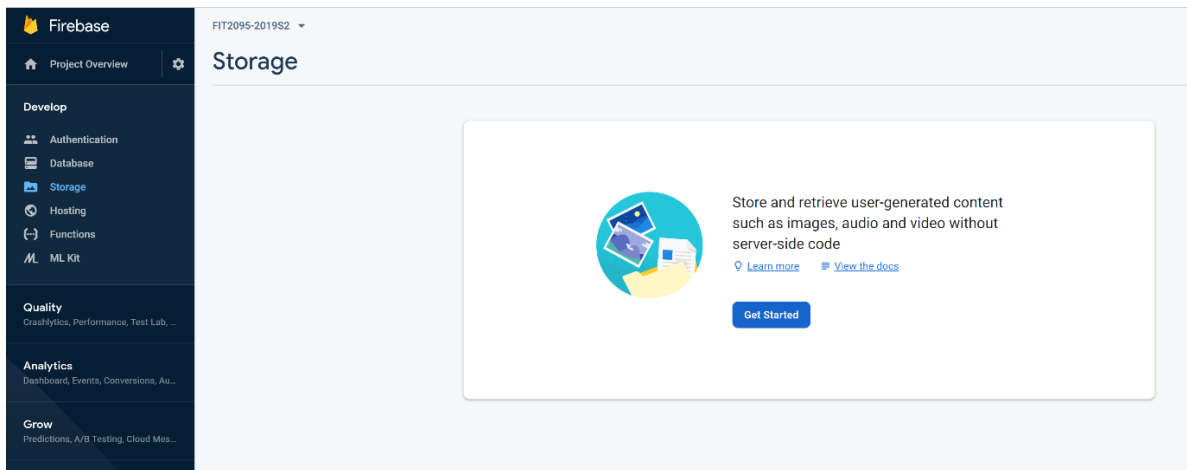


Figure 25 - Create Firestore Storage

You will see the following screen once the bucket is created. Click on the “Rules” tab.

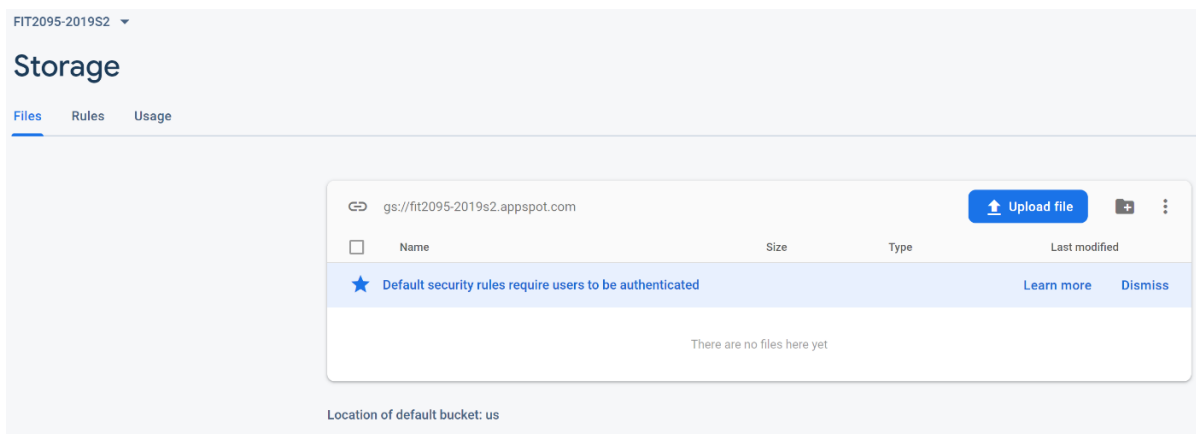


Figure 26 - Firestore Storage Created

Change the rule on line 5 from *allow read, write: if request.auth != null;* to *allow read, write;* and click “Publish”.

Warning: Allowing reads and writes without authentication is not suitable for production use! Usually, you will need to restrict Firebase Storage access only to authenticated users when in production by change the rule on line 5 back to:

allow read, write: if request.auth != null;

For more information on security rules: <https://firebase.google.com/docs/rules/rules-language?authuser=0>

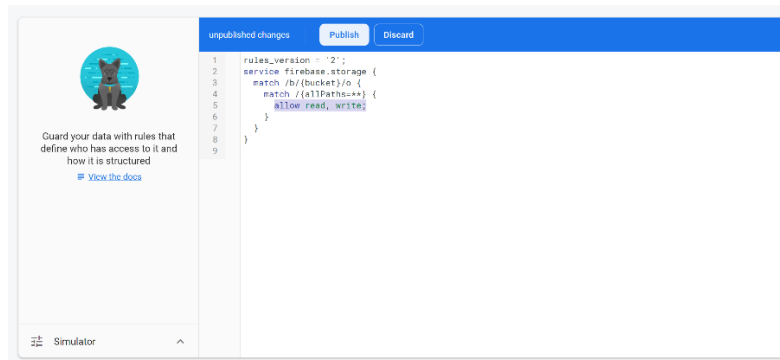
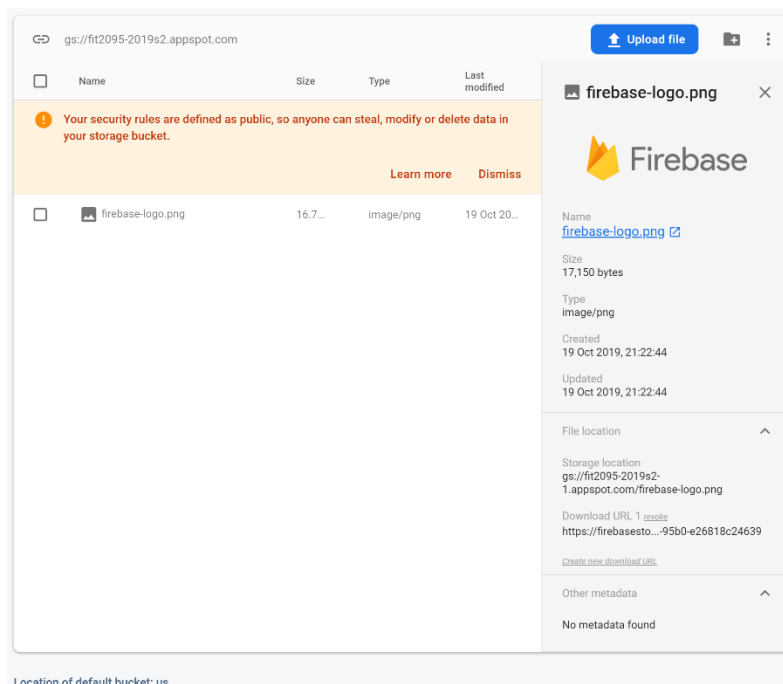


Figure 27 - Changing Storage Security Rules to Public

Adding files via the Firebase web interface

In the Firebase Storage viewer (see Figure 26), click on “Upload File”. The file selector menu will appear; select an image file to upload. Once the upload is finished, click on the new file entry that appears in the viewer. You will see the following screen, and most importantly, the download URL for the file we just uploaded.



Integrating Firebase Storage into an Angular application

1. Import the AngularFireStorageModule in *app.module.ts*.

Add the following import statements:

```
import { AngularFireStorageModule } from '@angular/fire/storage';
```

Add the following line in the *imports* array under the *@NgModule* decorator:

```
AngularFireStorageModule
```

2. Import AngularFireStorage into *<your-component>.component.ts*, where *<your-component>* is the component name.

Add the following import statements:

```
import { AngularFireStorage } from '@angular/fire/storage';
```

3. Delete the *createImage()*, *updateImage()* and *deleteImage()* functions from *<your-component>.component.ts*, where *<your-component>* is the component name. Do not delete any existing variables.

4. Add the following TypeScript code snippets to the exported class of *<your-component>.component.ts*, where *<your-component>* is the component name. Make sure HttpClient, AngularFirestore and Angular FireStorage are injected into the constructor as shown in the snippet immediately below. Do not delete any existing variables.

```
constructor(private http: HttpClient, private database: AngularFirestore, private storage: AngularFireStorage) {  
  this.getAllProducts();  
  this.getAllImages();  
}
```

createImage()

```
async createImage() {  
  // We need to ensure that the image ID in Firestore, and the actual image file ID in Storage are the same.  
  // We shall be storing thumbnails in a "thumbnails" folder.  
  let newImageId = this.database.createId();  
  let fileRef = this.storage.ref(`thumbnails/${newImageId}`);  
  
  // With await, we make sure the file gets uploaded first before we proceed to obtaining its download URL.  
  await fileRef.put(this.imageFileToCreate);  
  
  // The image document in Firestore then gets created with the new URL.  
  // The set() function will update an existing document, but will create it if it does not (like upsert in MongoDB).  
  fileRef.getDownloadURL().subscribe(downloadUrl => {  
    this.database.collection("images").doc(newImageId)  
      .set({  
        url: downloadUrl,  
      })  
      .catch(error => {  
        console.error("Error creating image: ", error);  
      });  
  });  
}  
  
imageFileToCreate: File;  
onUploadImageForCreation(files: FileList) {  
  this.imageFileToCreate = files[0];  
}
```

updateImage()

```
async updateImage() {
  // An update is essentially a delete and a create done together on the same file.
  // We need to wait (using await) to ensure we create only after the delete has finished.
  let fileRef = this.storage.ref(`thumbnails/${this.imageIdToUpdate}`);
  await fileRef.delete().toPromise();
  await fileRef.put(this.imageFileToUpdate);

  fileRef.getDownloadURL().subscribe(downloadUrl => {
    this.database.collection("images").doc(this.imageIdToUpdate)
      .set({
        url: downloadUrl
      })
      .catch(error => {
        console.error("Error updating image: ", error);
      });
  });
}

imageFileToUpdate: File;
onUploadImageForUpdate(files: FileList) {
  this.imageFileToUpdate = files[0];
}
```

deleteImage()

```
deleteImage() {
  let imageRef = this.database.collection("images")
    .doc(this.imageIdToDelete)
    .ref;

  // Remove the reference to the image from the product that refers to it.
  this.database.collection("products", ref => ref.where('images', 'array-contains', imageRef))
    .get() // This is the second way to perform GEI requests with Firestore, other than with snapshotChanges().
    .subscribe((querySnapshot: QuerySnapshot<any>) => {
      let numProductsToUpdate: number = querySnapshot.size;

      if (numProductsToUpdate === 0) {
        // If we have no affected products, delete the image document right away and return.
        imageRef
          .delete()
          .then(() => {
            let fileRef = this.storage.ref(`thumbnails/${this.imageIdToDelete}`);
            fileRef.delete().toPromise();
          })
          .catch(function (error) {
            console.error("Error deleting image: ", error);
          });

        return;
      }

      // A QuerySnapshot contains QueryDocumentSnapshots from a Firestore query.
      // QueryDocumentSnapshots are similar to DocumentSnapshots.
      querySnapshot.forEach((matchingProduct: QueryDocumentSnapshot<any>) => {
        matchingProduct.ref
          .update({
            images: matchingProduct.data().images.filter((ref: DocumentReference) => {
              return ref.id !== imageRef.id;
            }) // Remove a document's image reference if its ID is the same as the ID of the image being deleted.
          })
          .then(() => {
            numProductsToUpdate--;
            // After we updated all product documents, delete the image.
            if (numProductsToUpdate == 0) {
              imageRef
                .delete()
                .then(() => {
                  let fileRef = this.storage.ref(`thumbnails/${this.imageIdToDelete}`);
                  fileRef.delete().toPromise();
                })
                .catch(function (error) {
                  console.error("Error deleting image: ", error);
                });
            }
          })
          .catch(error => {
            console.log("Error deleting product image reference: ", error);
          });
      });
    });
}
```


5. Replace the following pieces of HTML code in *<your-component>.component.html*:

These lines in the Create Image form:

```
<label>URL</label>  
<input type="text" name="newImageUrl" [(ngModel)]="newImageUrl" required><br />
```

with:

```
<input type="file" name="newImageToUpload" (change)="onUploadImageForCreation($event.target.files)" required><br />
```

And this line in the Update Image form:

```
<label>URL</label>  
<input type="text" name="urlOfImageToUpdate" [(ngModel)]="urlOfImageToUpdate" required><br />
```

with:

```
<input type="file" name="newImageToUpdate" (change)="onUploadImageForUpdate($event.target.files)" required><br />
```

Connecting Google Cloud Functions, Firebase Firestore and Firebase Storage

We have just seen how we can use Firebase Firestore and Firebase Storage together, and we now have a fully functional product image gallery application that allows users to add, update and delete products and images. For the keen-eyed amongst you, you may have noticed throughout the previous sections that it takes quite some time for the preview of a new image to appear in the images table when we created or updated an image, especially when the image file is quite large. How do we fix this issue?

Fortunately, we can use the image resizing cloud function that we created from earlier (**Note: as mentioned, the current image resizing cloud function implementation only accepts images of any size up to a limit of 10MB**). This function will be invoked before we upload the user's selected image to Firebase Storage. Once a resized image is returned from the cloud function call, that resized image is then uploaded to Firebase Storage instead. This ensures that we only store and display preview images as 64px by 64px thumbnails instead of at full resolution.

1. Add the following TypeScript code snippets to the exported class of `<your-component>.component.ts`, where `<your-component>` is the component name.

This snippet is a modified version of the `onSubmitForm()` function from the “Integrating Google Cloud Functions into an Angular application” section. Replace the value of the `url` variable with the value you have obtained earlier from the “Testing our cloud function” section.

```
async resizeImage(file: File) {
  const formData = new FormData();
  formData.append('file', file, file.name);
  formData.append('newWidth', "64");
  formData.append('newHeight', "64");

  const url = "<REPLACE ME WITH THE CLOUD FUNCTION ENDPOINT URL>";

  // Observables can be converted to Promises and be used like a regular Promise.
  // The resized image will be returned as a Binary Large Object (blob).
  // The put() method we are using to upload files to Firebase Storage accepts the following upload formats:
  // Files selected from the from input field (of type file), blobs, and images encoded as Base64 strings.
  return await this.http.post<any>(url, formData, { responseType: 'blob' as 'json' }).toPromise();
}
```

Add this snippet to the very start of the `createImage()` function, before all the existing lines of code in that function.

```
async createImage() {
  let imageBlob = await this.resizeImage(this.imageFileToCreate);
  this.imageFileToCreate = imageBlob;
  /* EXISTING createImage() CODE GOES HERE */
}
```

Add this snippet to the very start of the `updateImage()` function, before all the existing lines of code in that function.

```
async updateImage() {
  let imageBlob = await this.resizeImage(this.imageFileToUpdate);
  this.imageFileToCreate = imageBlob;
  /* EXISTING updateImage() CODE GOES HERE */
}
```

...And that is it!

Further Exploration

1. Add Firebase Authentication, login system and enable authorisation for Firestore and Firebase Storage

<https://firebase.google.com/docs/auth/web/firebaseui>

<https://firebase.google.com/docs/firestore/solutions/role-based-access>

2. Use Firebase Hosting to serve Angular apps

<https://firebase.google.com/docs/hosting>

<https://medium.com/@longboardcreator/deploying-angular-6-applications-to-firebase-hosting-b5dacde9c772>

3. There are many ways to represent relationships between documents and collections in Firebase (and generally, in NoSQL-based databases). Explore the various ways.

<https://firebase.google.com/docs/firestore/manage-data/structure-data>

<https://angularfirebase.com/lessons/firestore-nosql-data-modeling-by-example/>

<https://proandroiddev.com/working-with-firestore-building-a-simple-database-model-79a5ce2692cb>

Further Challenges

1. At the end of the guide, the Create Image and Update Image forms have misaligned labels, input fields and buttons. Fix it so that labels are on the left column, and the input fields and submit buttons are on the right column.
2. Currently, the size of the preview images is hardcoded to be 64px by 64px. Modify the given code so that the thumbnail size can be set throughout the application by just using a single variable. You can start by modifying the *resizeImage()* function and the ** tags in the provided HTML code.
3. There are duplicated pieces of code in the *createImage()*, *updateImage()* and *deleteImage()* functions. Can you refactor the code and remove the duplication?
4. At the end of the guide, the user can only upload their own images and can no longer specify an existing URL. Modify the given code to allow users to choose between these two options when creating and updating an image.
5. When obtaining the array of *n* products from Firestore, we will be making a further *n* requests when resolving the products' image references to obtain their respective image URLs. Can you think of a way and modify the code so that we don't make further Firestore *get()* calls after we have initially obtained the array of products? *Hint: Images array*

References

<https://martinfowler.com/bliki/Serverless.html> (Figure 1)

<https://www.cloudflare.com/learning/serverless/what-is-serverless/>

<https://www.cloudflare.com/learning/serverless/why-use-serverless/>

<https://serverless.com/blog/when-why-not-use-serverless/>

<https://cloud.google.com/functions/docs/writing/http>

<https://angularfirebase.com/lessons/image-thumbnail-resizer-cloud-function/>

<https://firebase.google.com/docs/>

<https://github.com/angular/angularfire/>

The source code for this project can be found on GitHub at: <https://bit.ly/35K5ypD>