Alex Simoes Follow
Boston north shore native. Coffee frappe lover.
Jun 15, 2017 · 4 min read

Uploading to Google Cloud Storage from Node.js

Recently here at Datawheel we've started hosting many of our sites on Google's Cloud Platform (their answer to AWS). On the whole we've been quite happy, though, as with any new service, there was a bit of a learning curve getting started. The most recent hurdle we had to jump was figuring out how to allow user uploads to Cloud Storage, Google's equivalent of Amazon's S3 buckets. In fact Google calls them buckets too.

So, if you're also trying to incorporate Google Cloud Storage into your Node app and don't know where to start, fret no more! This article will walk you through the basics and point out some gotchas along the way. BUT, if you'd rather just look at the code, have at!

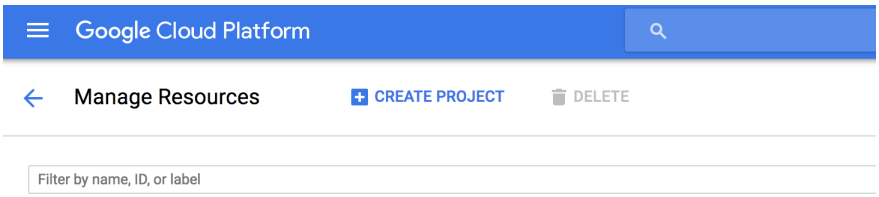
https://github.com/alexandersimoes/google-cloud-storage-node-boilerplate

. . .

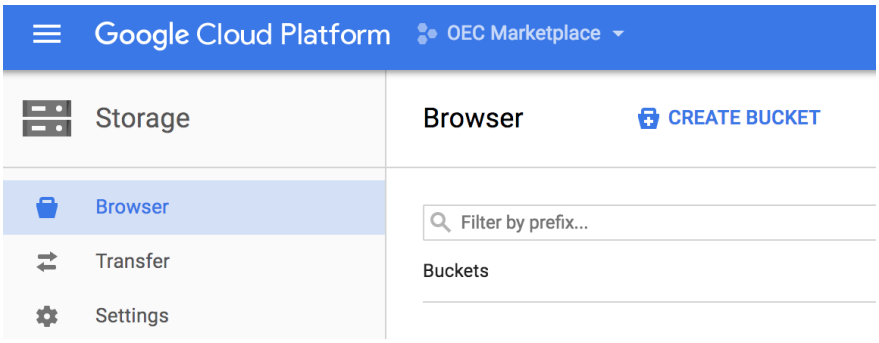
Getting Started

To get started using Google Cloud Storage in our Node.js evironment we'll need a project assigned to it along with a storage bucket and a permissions file. Below we'll outline how to do each of these steps.

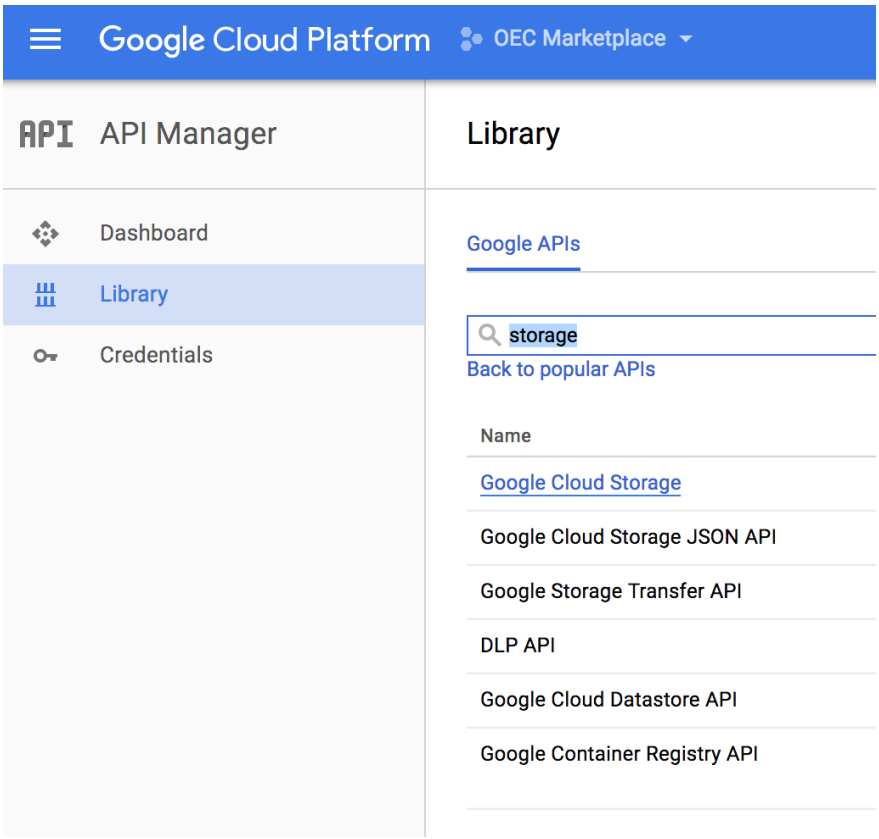
First off, let's create our project! We will need to have a project ID which will get assigned when we create a new instance. To create a new project go to the Cloud Platform Console.



Next, head over to your Storage browser and create a new bucket.



As a final step in our getting started phase we'll need to enable the Storage API for our Node app. We can do this through the web interface and download a JSON key file once the API has been enabled.



Spin Up Our Node App

Now that we've created our project and enabled the proper APIs we can actually get down to coding our site! We'll be using Express 4.x to handle all of our routes, Multer to read our file upload from our request and the google-cloud-storage node package as a wrapper around their API.

```
npm i @google-cloud/storage express multer --save
```

Since I'm a firm believer in ES6 being the future I prefer to also write my server-side Node code in ES6 as well. To do this we'll install a few additional packages. Though they are not absolutely necessary for this example app, to run all of the example you will need it.

```
npm i babel-cli babel-preset-es2015 babel-preset-stage-2 --
save-dev
```

Now for our server-side code we'll only have 2 routes: `/` and `/upload` which we'll tell express about.

```
// Display a form for uploading files.
app.get("/", (req, res) => {
  res.sendFile(path.join(`${__dirname}/index.html`));
});


// Process the file upload and upload to Google Cloud
Storage.
app.post("/upload", m.single("file"), (req, res, next) => {
  // Handle file upload here...
});
```

Now as for handling the file upload if you are testing the app code with postman, there is one gotcha that you'll need to know and that I discovered after pulling my hair out for an hour or so. Multer sends a cookie `connect-sid` which tells Postman not to re-request the server code. So anytime you make a change to your application, you will have

to either restart Postman or open a new tab with your request. Luckily I found this stackoverflow post which documents the issue in the second answer. THX INTERNET :-).

Handle File Uploads

The Google Cloud Storage node library exposes quite a few helper functions for dealing with file uploads. First off, instead of saving the user upload to a temporary file we'll just pipe the file stream straight to our bucket. This can be achieved with the `createWriteStream()` function. We can call this function on our `req.file` object and listen for the error and finish events that are fired asynchronously during the streaming process.

```
// Process the file upload and upload to Google Cloud
Storage.
app.post("/upload", m.single("file"), (req, res, next) => {
  const blob = bucket.file(req.file.originalname);
  const blobStream = blob.createWriteStream({
    metadata: {
      contentType: req.file.mimetype
    }
  });

  blobStream.on("error", err => {
  });

  blobStream.on("finish", () => {
  });
});
```

Once the file is finished uploading to our Google Cloud Storage bucket, we'll make it public (since the default behavior is for newly uploaded files to be private) and send the path of the file as a response back to the browser. This public path to the file is a unique identifier based on the bucket name and the filename.

```
blobStream.on("finish", () => {
  // The public URL can be used to access the file via HTTP.
  const publicUrl =
`https://storage.googleapis.com/${bucket.name}/${blob.name}`
;
```

```
    // Make the image public to the web
    blob.makePublic().then(() => {
     res.status(200).send(`Success!\n Image uploaded to
  ${publicUrl}`);
    });
  });
```

There we have it! Our `index.html` file is just a basic upload form but we could imagine how this could work entirely as AJAX requests and also how we could hook up links to store a reference to our new file path in a database for later retrieval.

Helpful links

alexandersimoes/google-cloud-storage-node-boilerplate

google-cloud-storage-node-boilerplate - Boilerplate Node.js app for handling user upload...

github.com

google-cloud

Google Cloud Platform's client library documentation

googlecloudplatform.github.io

Using Cloud Storage | App Engine flexible environment for Node.js docs | Google Cloud...

'use strict'; const format = require('util').format; const express = require('express'); const Multer ...

cloud.google.com

Node.js Bookshelf App | Node.js | Google Cloud Platform

The structured data part of the tutorial demonstrates how the sample app stores book...

cloud.google.com