Full Stack Development

19 / 22

Storage

Google Cloud

17 - Add Videos to Firestore

14:29

Mark Lesson Complete

Add Videos to Firestore

We now are able to upload videos end-to-end, but we are not saving any information about the videos in our database. So we have no easy way to serve the videos to our users, or to display them in our UI.

We should only save the video metadata to our database, and not the actual video file. And we should only do so if the video was successfully processed.

Firebase

Therefore, the best place to put this logic is in our `video-processing-service`.

And this will also allow us to fix a *very subtle bug* that we have in our current implementation.

1. The Bug

Currently, Pub/Sub will redeliver a message if the subscriber does not respond with a `200` status code within 600 seconds.

But the max timeout for a Cloud Run request is 3600 seconds.

So what if a video takes longer than 600 seconds to process?

Pub/Sub will redeliver the same message, and we will end up processing the same video multiple times.

So how can we fix it?

Along with the video metadata, we should store a `status` field in Firestore. When we receive a message from Pub/Sub, we should check the status of the video in Firestore. If it does not exist we should set it to `processing`. If the field already exists, we should ignore the request and *not* process the video again.

> This is called **idempotency**. So it's safe for Pub/Sub to repeat the same request multiple times without any side effects.

After processing is complete, we can update the status to `processed` (or `complete`).

This way we know the video is ready to be served to our users within the UI.

2. Add Video to Firestore from Video Processing Service

Install `firebase-admin` in the video processing service:

```
npm install firebase-admin
```

Create a file called `firestore.ts` within the `src` directory.

```
import { credential } from "firebase-admin";
import {initializeApp} from "firebase-admin/app";
import {Firestore} from "firebase-admin/firestore";

initializeApp({credential:
credential.applicationDefault()});

const firestore = new Firestore();

// Note: This requires setting an env variable in
Cloud Run
/** if (process.env.NODE_ENV !== 'production') {
  firestore.settings({
      host: "localhost:8080", // Default port for
Firestore emulator
      ssl: false
  });
} */


const videoCollectionId = 'videos';

export interface Video {
  id?: string,
  uid?: string,
  filename?: string,
  status?: 'processing' | 'processed',
  title?: string,
  description?: string
}

async function getVideo(videoId: string) {
  const snapshot = await
firestore.collection(videoCollectionId).doc(
videoId).get();
  return (snapshot.data() as Video) ?? {};
}

export function setVideo(videoId: string, video:
Video) {
  return firestore
    .collection(videoCollectionId)
    .doc(videoId)
    .set(video, { merge: true })
}

export async function isVideoNew(videoId: string) {
  const video = await getVideo(videoId);
  return video?.status === undefined;
}
```

> Since there can only be one Firestore instance per GCP project, we don't have to specify which Firestore instance to use.

We are using the `applicationDefault()` credential, which means that the service account will be automatically inferred from the environment. This is because we are running the video processing service within Cloud Run, in the same project as our Firestore instance.

It's still possible to test this locally, but we will have to set up the Firebase emulator to start a local Firestore instance. We will skip this, and deploy straight to Cloud Run.

We will also update the `/process-video` endpoint in `index.ts` file to look like this:

```
import { isVideoNew, setVideo } from "./firestore";

app.post("/process-video", async (req, res) => {
  ...

  const inputFileName = data.name; // In format of
<UID>-<DATE>.<EXTENSION>
  const outputFileName =
`processed-${inputFileName}`;
  const videoId = inputFileName.split('.')[0];

  if (!isVideoNew(videoId)) {
    return res.status(400).send('Bad Request: video
already processing or processed.');
  } else {
    await setVideo(videoId, {
      id: videoId,
      uid: videoId.split('-')[0],
      status: 'processing'
    });
  }

  // Download the raw video from Cloud Storage
  await downloadRawVideo(inputFileName);

  ...

  // Upload the processed video to Cloud Storage
  await uploadProcessedVideo(outputFileName);

  await setVideo(videoId, {
    status: 'processed',
    filename: outputFileName
  });

  await Promise.all([
    deleteRawVideo(inputFileName),
    deleteProcessedVideo(outputFileName)
  ]);

  return res.status(200).send('Processing finished
successfully');
});
```

> Some of the unchanged code has been omitted for brevity.

3. Redeploy Video Processing Service

Rebuild the Docker image:

```
docker build -t us-central1-docker.pkg.dev/<PROJECT_ID>/video-processing-repo/video-processing-service .
```

> Important: If you are using mac, add `--platform linux/amd64` to the above command.

Then, push the Docker image to Google Artifact Registry:

```
docker push us-central1-docker.pkg.dev/<PROJECT_ID>/video-processing-repo/video-processing-service
```

Redeploy the container to Cloud Run via the CLI:

```
# Deploy container to cloud run
gcloud run deploy video-processing-service --image us-central1-docker.pkg.dev/PROJECT_ID/video-processing-repo/video-processing-service \
  --region=us-central1 \
  --platform managed \
  --timeout=3600 \
  --memory=2Gi \
  --cpu=1 \
  --min-instances=0 \
  --max-instances=1 \
  --ingress=internal
```

Or via the console: **https://console.cloud.google.com/run**

4. Test Upload Video from Next.js

Run the Next.js app locally and upload a video.

To further

5. Inspect Firestore in Firebase Console

https://console.firebase.google.com/project//firestore/data/