Serverless web performance monitoring using Cloud Functions

This tutorial describes how to create a web-performance-monitoring app using Google Cloud Platform (GCP) serverless technologies.

Performance plays a major role (https://developers.google.com/web/fundamentals/performance/why-performance-matters/) in the success of any web app. If your site performs poorly, you might experience fewer sign-ups and lower user retention, which will probably impact your business goals. Performance should be a key success criterion when designing, building, and testing your web app.

However, page performance can also change over time as your app evolves. Developers can add or update images and scripts, or the underlying app serving infrastructure itself can change. Therefore, it's important to regularly monitor page performance. Typically, you store the performance metrics to enable historical analysis. It's also common practice to generate alerts if page performance falls below some defined thresholds.

Objectives

- Create a Cloud Function (https://cloud.google.com/functions/) that uses headless Chrome to collect web page performance metrics.
- Store the metrics in Cloud Storage (https://cloud.google.com/storage/).
- Create another Cloud Function, triggered by the Cloud Storage creation event, to analyze the page metrics.
- Store the analysis results in Cloud Firestore (https://cloud.google.com/firestore/).

- Create another Cloud Function, triggered by the Cloud Firestore creation event, to publish an alert to Cloud Pub/Sub (https://cloud.google.com/pubsub/) if page performance is poor.
- Create a Cloud Scheduler (https://cloud.google.com/scheduler/) job to periodically trigger the first Cloud Function.
- Verify the outputs for success and for failure scenarios.

Costs

This tutorial uses billable components of Google Cloud Platform, including:

- Cloud Functions
- Cloud Scheduler
- Cloud Storage
- Cloud Firestore
- Cloud Pub/Sub

Use the pricing calculator (https://cloud.google.com/products/calculator/#id=d8864505-8045-462f-804e-4920af970f1d) to generate a cost estimate based on your projected usage.

Before you begin

1. Sign in (https://accounts.google.com/Login) to your Google Account.

   If you don't already have one, sign up for a new account (https://accounts.google.com/SignUp).

2. In the Cloud Console, on the project selector page, select or create a Cloud project.

   Note: If you don't plan to keep the resources that you create in this procedure, create a project instead of selecting an existing project. After you finish these steps, you can delete the project, removing all resources associated with the project.

   GO TO THE PROJECT SELECTOR PAGE (HTTPS://CONSOLE.CLOUD.GOOGLE.COM/PROJECTSELECT

3. Make sure that billing is enabled for your Google Cloud project. Learn how to confirm billing is enabled for your project (https://cloud.google.com/billing/docs/how-to/modify-project).
4. Enable the Cloud Functions, Cloud Scheduler, and Cloud Pub/Sub APIs.

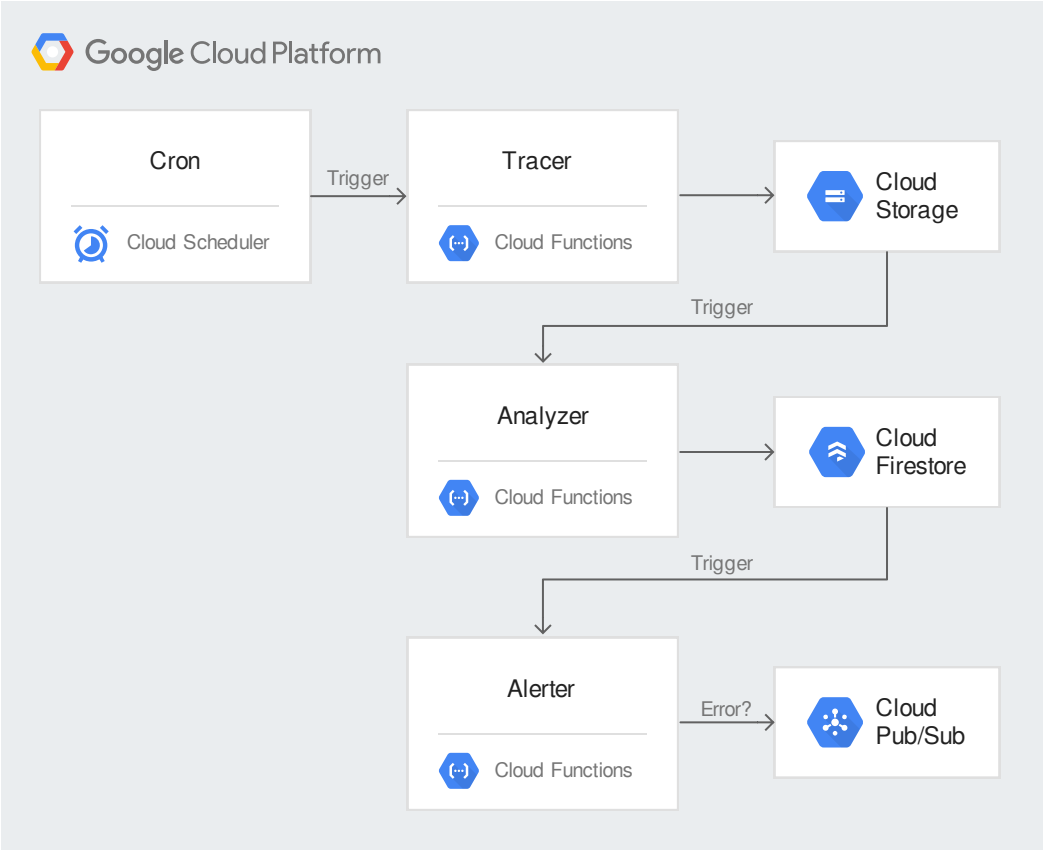
   ENABLE THE APIS (HTTPS://CONSOLE.CLOUD.GOOGLE.COM/FLOWS/ENABLEAPI?APIID=CLOUDFU

Architecture

Web performance monitoring operations are typically stateless and short-lived. They are also often event-driven, occurring either on a schedule or triggered as part of some other process, such as an automated testing pipeline. These characteristics make serverless architectures an appealing choice for implementing web analysis apps.

In this tutorial, you use various parts of the GCP serverless (https://cloud.google.com/serverless/) stack, including Cloud Functions, Cloud Firestore, Cloud Scheduler, and Cloud Pub/Sub. You don't have to manage the infrastructure for any of these services, and you pay only for what you use. The core of the app is implemented using Cloud Functions, which provides an event-driven and scalable serverless execution environment. Using Cloud Functions, you can create and connect apps using independent, loosely coupled pieces of logic.

The following diagram shows the architecture of the serverless solution that you create in this tutorial.



Preparing the environment

Before you create the serverless environment, you get the code from GitHub, set variables, and prepare resources you need later for analyzing and storing.

Get the code and set environment variables

1. In the GCP Console, open Cloud Shell.

   OPEN CLOUD SHELL (HTTPS://CONSOLE.CLOUD.GOOGLE.COM/?CLOUDSHELL=TRUE)

2. Clone the repository that contains the code for the Cloud Functions used in this tutorial:

```
git clone https://github.com/GoogleCloudPlatform/solutions-serverless-web-monit
```

3. Change to the functions directory:

```
cd solutions-serverless-web-monitoring/functions
```

4. Set the current project ID and project number as shell variables:

```
export PROJECT=$DEVSHELL_PROJECT_ID
export PROJECT_NUM=$(gcloud projects list \
    --filter="$PROJECT" \
    --format="value(PROJECT_NUMBER)")
```

5. Set the default deployment region for Cloud Functions. The following example sets the region to `us-east1`, but you can change this to any region where Cloud Functions is available (https://cloud.google.com/functions/docs/locations).

```
export REGION=us-east1
gcloud config set functions/region $REGION
```

Create Cloud Storage buckets

In this section, you create a Cloud Storage bucket to store the collected page performance data. You can choose any location or storage class (https://cloud.google.com/storage/docs/storage-classes), but it's a good practice to create buckets in the same location as the Cloud Functions that will use the buckets.

1. In Cloud Shell, export a shell variable for the names of the Cloud Storage buckets that will store the metrics. Bucket names must be globally unique, so the following command uses your GCP project number as a suffix on the bucket name.

```
export METRICS_BUCKET=page-metrics-$PROJECT_NUM
```

2. Use the `gsutil` tool to create the buckets:

```
gsutil mb -l $REGION gs://$METRICS_BUCKET
```

3. Update the `env-vars.yaml` file with the bucket names. This file contains environment variables that you will pass to the Cloud Functions later.

```
sed -i "s/\[YOUR_METRICS_BUCKET\]/$METRICS_BUCKET/" env-vars.yaml
```

Create a Cloud Firestore collection

In a later section, you analyze the page performance metrics. In this section, you create a Cloud Firestore collection (https://cloud.google.com/firestore/docs/data-model) to store the results of each analysis.

1. In the GCP Console, go to the Cloud Firestore page.

   GO TO THE CLOUD FIRESTORE PAGE (HTTPS://CONSOLE.CLOUD.GOOGLE.COM/FIRESTORE)

2. If you've never created a Cloud Firestore database before, perform the following steps:

   a. Click **Select Native mode** to activate Cloud Firestore.

   b. Select a regional location close to the region where your Cloud Functions will run.

   c. Click **Create Database**.

   It takes a few moments to complete the configuration.

3. Click **Start Collection** and set the collection ID to `page-metrics`.

4. Click **Save**.

Create a Cloud Pub/Sub topic and subscription

Typically you want to notify interested systems and parties if the analysis indicates that a page is performing poorly. In this section, you create Cloud Pub/Sub topics that will contain messages that describe any poor performance. we 1. In Cloud Shell, create a Cloud Pub/Sub topic named `performance-alerts`:

```none
gcloud pubsub topics create performance-alerts
```

1. Create a subscription to the topic. You use the subscription to verify that alert messages are being published to the topic.

```
gcloud pubsub subscriptions create performance-alerts-sub \
    --topic performance-alerts
```

Collecting page performance metrics

Many websites use JavaScript to dynamically render page content. This makes performance analysis more complicated, because the client needs to emulate a browser in order to fully load the web page. The Node.js 8 runtime for Cloud Functions has support for headless Chrome, which provides the functionality of a full web browser in a serverless environment.

Puppeteer (https://github.com/GoogleChrome/puppeteer) is a Node.js library built by the Chrome DevTools team that provides a high-level API to control headless Chrome. By default, Puppeteer installs a recent version of the browser alongside the library. Therefore, you can add Puppeteer as a dependency to the Cloud Function as an easy way to use headless Chrome within the function.

Measuring and analyzing web page performance is a large and complex field. For simplicity, in this tutorial you use Puppeteer to collect some top-level page performance metrics. However, you can also use Puppeteer and the Chrome DevTools Protocol (CDP) (https://chromedevtools.github.io/devtools-protocol/) to collect more detailed information, such as timeline traces. You can also better represent your end-user experience by emulating network congestion and performing CPU throttling. For a good introduction to analyzing web page performance, see the Chrome web developers site (https://developers.google.com/web/tools/chrome-devtools/evaluate-performance/).

Note that there are many factors that influence web page load times, including the performance characteristics of the client. It's important to establish baselines using the CPU and RAM configurations of the Cloud Function.

The following snippet from the `tracer/index.js` file shows how to use Puppeteer to load the web page:

functions/tracer/index.js (https://github.com/GoogleCloudPlatform/solutions-serverless-web-monitoring/blob/master/functions/tracer/index.js)

LATFORM/SOLUTIONS-SERVERLESS-WEB-MONITORING/BLOB/MASTER/FUNCTIONS/TRACER/INDEX.JS)

```
// launch Puppeteer and start a Chrome DevTools Protocol (CDP) session
// with performance tracking enabled.
browser = await puppeteer.launch({
  headless: true,
  args: ['--no-sandbox']
});
const page = await browser.newPage();
const client = await page.target().createCDPSession();
await client.send('Performance.enable');
```

```
// browse to the page, capture and write the performance metrics
console.log('Fetching url: '+url.href);
await page.goto(url.href, {
  'waitUntil' : 'networkidle0'
});
const performanceMetrics = await client.send('Performance.getMetrics');
options = createUploadOptions('application/json', page.url());
await writeToGcs(metricsBucket, filename, JSON.stringify(performanceMetrics), option
```

- In Cloud Shell, deploy the `trace` Cloud Function:

  ```
  gcloud functions deploy trace \
      --trigger-http \
      --runtime nodejs8 \
      --memory 1GB \
      --source tracer \
      --env-vars-file env-vars.yaml \
      --quiet
  ```

  It can take several minutes to deploy the Cloud Function.

  The deployment parameters specify that the function should have an HTTP trigger, should use the Node.js 8 runtime, and should have 1 GB memory. This amount of memory is required in order to run headless Chrome. Environment variables are supplied to the function by using the `env-vars.yaml` file

By default, HTTP-triggered Cloud Functions allow unauthenticated invocations. Therefore, you must secure (https://cloud.google.com/functions/docs/securing/) the trace function.

- Remove the `cloudfunctions.invoker` IAM role for `allUsers`:

  ```
  gcloud beta functions remove-iam-policy-binding trace \
      --member allUsers \
      --role roles/cloudfunctions.invoker
  ```

Analyzing the metrics

A typical goal of web-performance-monitoring exercises is to track performance against some defined benchmarks. If a particular metric exceeds an expected threshold, it can indicate a problem with a recent software release, or a problem with the underlying infrastructure.

In this section, you create a Cloud Function in Python to parse the page metrics and persist the results to a Cloud Firestore collection. The function evaluates the `FirstMeaningfulPaint` metric against an expected threshold, and marks the analysis result as problematic if the threshold is exceeded. `FirstMeaningfulPaint` is a user-centric metric (https://developers.google.com/web/fundamentals/performance/user-centric-performance-metrics) that broadly describes when a page becomes useful to the user. You use a Cloud Storage trigger to execute the analysis function whenever a new file is written to the bucket that contains the metrics.

The following snippet from the `analyzer/main.py` file shows the function logic:

functions/analyzer/main.py (https://github.com/GoogleCloudPlatform/solutions-serverless-web-monitoring/blob/master/functions/analyzer/main.py)

ATFORM/SOLUTIONS-SERVERLESS-WEB-MONITORING/BLOB/MASTER/FUNCTIONS/ANALYZER/MAIN.PY)

```
def analyze(data, context):
  """Function entry point, triggered by creation of an object in a GCS bucket.

  The function reads the content of the triggering file, analyses its contents,
  and persists the results of the analysis to a new Firestore document.

  Args:
    data (dict): The trigger event payload.
    context (google.cloud.functions.Context): Metadata for the event.
  """
  page_metrics = get_gcs_file_contents(data)
  max_time_meaningful_paint = int(os.environ.get('MAX_TIME_MEANINGFUL_PAINT'))
  analysis_result = analyze_metrics(data, page_metrics,
                                    max_time_meaningful_paint)
  docref = persist(analysis_result, data['name'])
  logging.info('Created new Firestore document %s/%s describing analysis of %s',
               docref.parent.id, docref.id, analysis_result['input_file'])
```

- Deploy the `analyze` Cloud Function:

```
gcloud functions deploy analyze \
    --trigger-resource gs://$METRICS_BUCKET \
    --trigger-event google.storage.object.finalize \
    --runtime python37 \
    --source analyzer \
    --env-vars-file env-vars.yaml
```

The function is triggered by a `finalize` event in the metrics bucket, which is sent every time an object is created in the bucket. The function uses the Python 3.7 runtime.

Alerting on failures

Typically, you want to take action if the metrics analysis indicates a poorly performing page.

In this section, you create a Cloud Function to send a message to a Cloud Pub/Sub topic if page performance is unsatisfactory. The function is triggered every time a document is created in the Cloud Firestore collection. Interested parties can subscribe to the Cloud Pub/Sub topic and take appropriate action. For example, a support app could subscribe to the Cloud Pub/Sub messages and send an email, trigger a support pager, or open a bug.

The following snippet from the `alerter/main.py` file shows the function logic:

functions/alerter/main.py
(https://github.com/GoogleCloudPlatform/solutions-serverless-web-monitoring/blob/master/functions/alerter/main.py)

_ATFORM/SOLUTIONS-SERVERLESS-WEB-MONITORING/BLOB/MASTER/FUNCTIONS/ALERTER/MAIN.PY)

```
def generate_alert(data, context):
  """Cloud Function entry point, triggered by a change to a Firestore document.

  If the triggering document indicates a Failed status, send the document to
  configured PubSub topic.

  Args:
    data (dict): The event payload.
    context (google.cloud.functions.Context): Metadata for the event.
  """
  doc_fields = data['value']['fields']
  status = doc_fields['status']['stringValue']
  if 'FAIL' in status:
    global publish_client
    if not publish_client:
      publish_client = pubsub.PublisherClient()

    logging.info('Sending alert in response to %s status in document %s',
                 status, context.resource)
    project = os.environ.get('GCP_PROJECT')
    topic = os.environ.get('ALERT_TOPIC')
```

```
    fqtn = 'projects/{}/topics/{}'.format(project, topic)
    msg = json.dumps(data['value']).encode('utf-8')
    publish_client.publish(fqtn, msg)
```

Notice that the alert is sent only if the status field indicates a failure.

- Deploy the `alert` Cloud Function:

  ```
  gcloud functions deploy alert \
      --trigger-event providers/cloud.firestore/eventTypes/document.create \
      --trigger-resource "projects/$PROJECT/databases/(default)/documents/page-me
      --runtime python37 \
      --source alerter \
      --env-vars-file env-vars.yaml \
      --entry-point generate_alert
  ```

  The function is triggered by a `document.create` event in the `page-metrics` Cloud Firestore collection. The `{any}` suffix is a wildcard indicating that the function should be triggered any time a document is created in the collection.

Scheduling the analysis

It's good practice to regularly monitor page performance. For example, you might want to analyze a certain page every hour, every day, or every week. In this section, you create a Cloud Scheduler job to periodically run the the analysis pipeline by triggering the `trace` function.

The Cloud Scheduler job is executed using a service account that's been granted the required `cloudfunctions.invoker` IAM role for the `trace` function.

Sometimes web pages don't respond correctly, or requests time out, so retries are unavoidable with web analysis apps. It's therefore important to have retry logic in your app. Cloud Functions supports retries for background functions (https://cloud.google.com/functions/docs/writing/background).

Retries are not available for HTTP-triggered Cloud Functions, so you can't use Cloud Functions to retry the `trace` function. However, Cloud Scheduler supports retries. For more information on configuring retry parameters, see the RetryConfig (https://cloud.google.com/scheduler/docs/reference/rest/v1/projects.locations.jobs#RetryConfig) documentation.

1. Verify that the three Cloud Functions have been correctly deployed and are showing `ACTIVE` status:

```
gcloud functions list
```

2. Create a new service account that will be used as the identity for executing the Cloud Scheduler job:

```
gcloud iam service-accounts create tracer-job-sa
```

3. Grant the new service account the `cloudfunctions.invoker` IAM role for the `trace` function:

```
gcloud beta functions add-iam-policy-binding trace \
    --role roles/cloudfunctions.invoker \
    --member "serviceAccount:tracer-job-sa@$PROJECT.iam.gserviceaccount.com"
```

4. Create a Cloud Scheduler job:

```
gcloud scheduler jobs create http traceWithRetry \
    --uri="https://$REGION-$PROJECT.cloudfunctions.net/trace" \
    --http-method=POST \
    --message-body="{\"url\":\"http://www.example.com\"}" \
    --headers="Content-Type=application/json" \
    --oidc-service-account-email="tracer-job-sa@$PROJECT.iam.gserviceaccount.co
    --schedule="0 3 * * *" \
    --time-zone="UTC" \
    --max-retry-attempts=3 \
    --min-backoff=30s
```

Because the job will call the HTTP-triggered `trace` function, the command specifies the job type as `http`, and supplies the function trigger URL as the `uri` value. The page to analyze, in this case `www.example.com`, is provided in the `message-body` flag. The `oidc-service-account-email` flag defines the service account to use for authentication. The command indicates the number of retries to attempt using the `max-retry-attempts` flag, and the value passed with the `schedule` flag sets the run schedule to 3:00 AM UTC every day.

Verifying results

In this section, you verify that you get the expected behavior for both success and failure conditions.

Verify success

The Cloud Scheduler job won't run until the next scheduled time, which in this case is 3:00 AM UTC. To see the results immediately, you can manually trigger a run.

1. Wait 90 seconds for the scheduler job to finish initializing.
2. Run the Cloud Scheduler job manually:

```
gcloud scheduler jobs run traceWithRetry
```

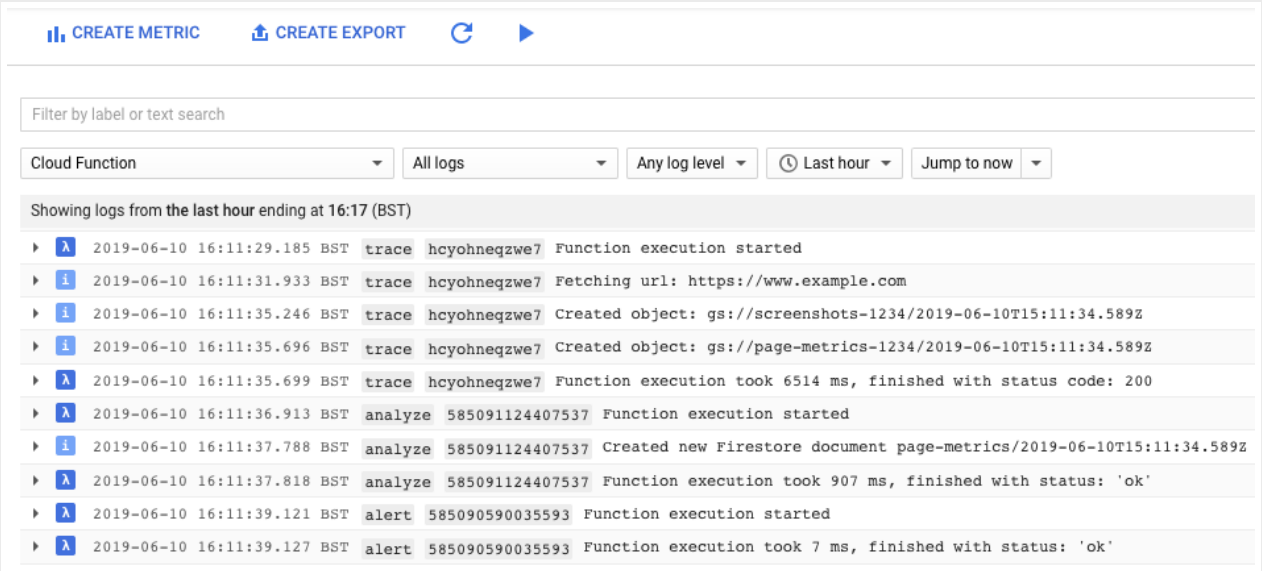
3. Wait about 30 seconds for the function pipeline to complete.
4. List the contents of the metrics bucket to show that page metrics have been collected:

```
gsutil ls -l gs://$METRICS_BUCKET
```

5. In the GCP Console, open the Stackdriver Logging viewer page:

   GO TO THE LOGGING PAGE (HTTPS://CONSOLE.CLOUD.GOOGLE.COM/LOGS/VIEWER?RESOURCE=(

   You see log messages from each of the three Cloud Functions: `trace`, `analyze`, and `alert`. It can take a few moments for the logs to flow through, so you might need to refresh to the logs pane.



6. Make a note of the Cloud Firestore document ID, which is listed following the text `Created new Firestore document page-metrics/`
7. In the GCP Console, go to the Cloud Firestore page:

   GO TO THE CLOUD FIRESTORE PAGE (HTTPS://CONSOLE.CLOUD.GOOGLE.COM/FIRESTORE/DATA/)
8. Inspect the document that contains results of the analysis. The document values indicate a `PASS` status and contain the latest page performance metrics.
9. In Cloud Shell, verify that no alert messages have been sent to the Cloud Pub/Sub topic by trying to pull a message off the subscription:

```
gcloud pubsub subscriptions pull \
    projects/$PROJECT/subscriptions/performance-alerts-sub \
    --auto-ack
```

   You see no items listed.

Verify failure

1. Manually trigger the trace function. This time, you provide the GCP Tutorials (https://cloud.google.com/docs/tutorials) page as the URL. This page has a lot of dynamic content that increases the page load time over the expected maximum threshold.

```
gcloud functions call trace \
    --data='{"url":"https://cloud.google.com/docs/tutorials"}'
```

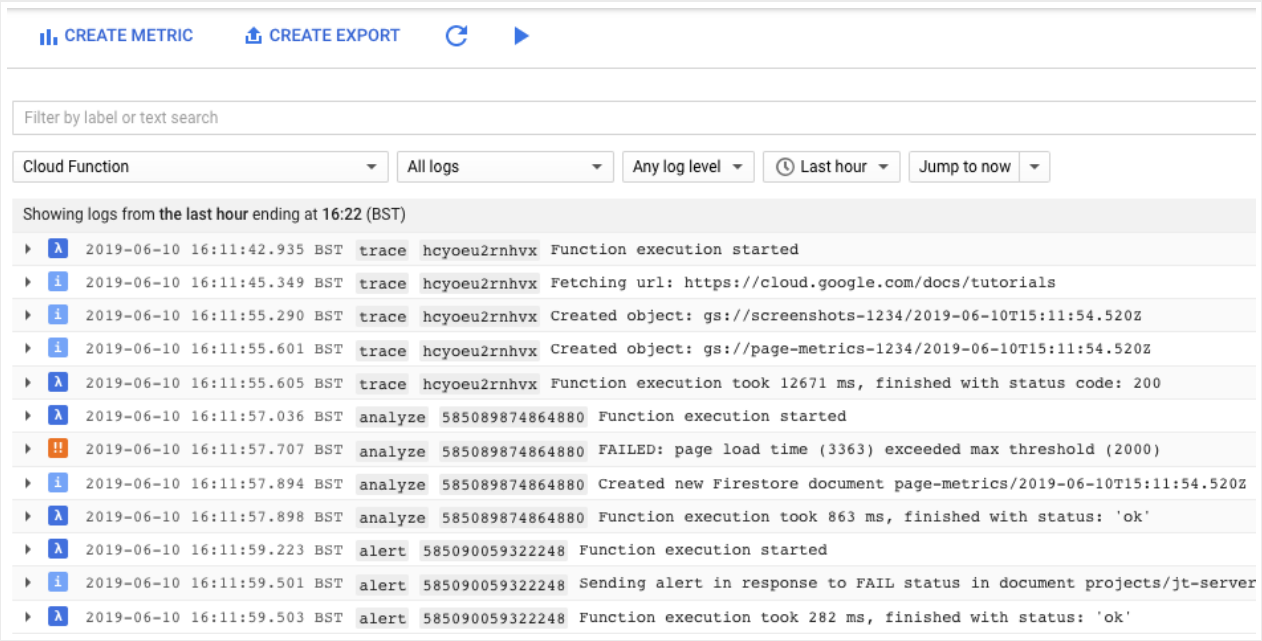
   Because you have project the `Owner` or `Editor` IAM role, you have sufficient permissions to invoke the function.
2. Wait about 30 seconds for the function pipeline to complete.
3. List the contents of the metrics bucket to verify that additional metrics have been collected:

```
gsutil ls -l gs://$METRICS_BUCKET
```

   You now see two items in each bucket.
4. In the GCP Console, go to the Stackdriver Logging viewer page and filter for the Cloud Function logs:

GO TO THE LOGGING PAGE (HTTPS://CONSOLE.CLOUD.GOOGLE.COM/LOGS/VIEWER?RESOURCE=C

You see an error from the `analyze` function indicating that the page exceeded the maximum allowed load time. Again, you might need to refresh the logs pane to see the latest messages.



5. Make a note of the Cloud Firestore document ID.
6. In the GCP Console, go to the Cloud Firestore page:

   GO TO THE CLOUD FIRESTORE PAGE (HTTPS://CONSOLE.CLOUD.GOOGLE.COM/FIRESTORE/DATA/)
7. Find the document that describes the failed analysis.

   The status field is marked as `FAIL`.
8. In Cloud Shell, verify that an alert message was sent to the Cloud Pub/Sub topic by pulling a message off the subscription.

```
gcloud pubsub subscriptions pull \
    projects/$PROJECT/subscriptions/performance-alerts-sub \
    --auto-ack
```

   This time, you see the contents of the message.

Cleaning up

Delete the project

Caution: Deleting a project has the following effects:

- **Everything in the project is deleted.** If you used an existing project for this tutorial, when you delete it, you also delete any other work you've done in the project.
- **Custom project IDs are lost.** When you created this project, you might have created a custom project ID that you want to use in the future. To preserve the URLs that use the project ID, such as an `appspot.com` URL, delete selected resources inside the project instead of deleting the whole project.

1. In the Cloud Console, go to the **Manage resources** page.

   GO TO THE MANAGE RESOURCES PAGE (HTTPS://CONSOLE.CLOUD.GOOGLE.COM/IAM-ADMIN/PRC

2. In the project list, select the project you want to delete and click **Delete** .
3. In the dialog, type the project ID, and then click **Shut down** to delete the project.

What's next

- Learn more about GCP serverless (https://cloud.google.com/serverless/) technologies.
- Explore other Cloud Functions tutorials (https://cloud.google.com/functions/docs/tutorials/).
- Watch the video (https://youtu.be/lhZOFUY1weo) from Google I/O '18 that describes other uses for Puppeteer and headless Chrome.
- Try out other Google Cloud Platform features for yourself. Have a look at our tutorials (https://cloud.google.com/docs/tutorials).

Last updated November 12, 2019.