O'REILLY®

Google Cloud Cookbook

Practical Solutions for Building and Deploying Cloud Services

Early Release

RAW & UNEDITED

Rui Santos Costa
& Drew Hodun

Chapter 1. Cloud Functions

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at ruiscosta@google.com and dhodun@google.com.

Google Cloud Functions is a serverless compute solution that allows you to run event-based applications. There are two distinct types of Cloud Functions: HTTP functions and background functions. You invoke HTTP functions from standard HTTP requests. You use background functions when you want to have your Cloud Function invoked indirectly in response to an event, such as a message on a Pub/Sub topic, a change in a Cloud Storage bucket, or a Firebase event.

Google Cloud Functions allow you to write single-purpose functions which can be attached to events from your services. They also abstract the compute infrastructure and allow you to focus on your code. You don't have to worry about patching operating systems or provisioning resources. Cloud Functions scale automatically; they can scale from a single invocation to millions without intervention from the developer. In this chapter, we will present a range of recipes, from introductory recipes you can use to send emails or respond to SMS messages, to advanced

recipes that show you how to integrate CI/CD into your development workflow, and integrate with Cloud Endpoints for API management.

All code samples for this chapter are located at https://github.com/ruiscosta/google-cloud-cookbook/chapter-1. You can follow along and copy the code for each individual recipe by going to the folder with that recipe's number

1.1 Creating a public HTTP Google Cloud Function

Problem

You want to create your first HTTP cloud function to perform a simple Hello World.

Solution

Leverage Node.js to create an HTTP cloud function to use the response object to send a short Hello World response.

1. On your local workstation create a temporary folder to hold the files you will need to create the hello world http function.
2. In your terminal run the following command: npm init
3. Accept the defaults when prompted to set up a new npm package
4. In your favorite IDE create an index.js file in the root of the directory you created on step 1 and copy the code below to the file.

```
const escapeHtml = require('escape-html');
exports.helloHttp = (req, res) => {
  res.send(`Hello World!`);
};
```

5. Run the following command to install the escape-html package which will also add the package to your package.json dependencies.

```
npm install escape-html --save
```

6. To deploy the Cloud Function run the following command:

```
gcloud functions deploy hello-http-function --entry-poin
```

7. When presented with the choice to allow unauthenticated [helloHttp]? (y/N) **Select Y**. Selecting **Y** allows users to access your function without authentication.

Discussion

You have successfully deployed your first HTTP Google Cloud Function that allows all users to access it. It is a simple hello world but provides you some basic concepts on deploying functions. Let's further break down the deployment process and understand the arguments passed to the gcloud functions deploy command.

```
gcloud functions deploy NAME --entry-point NAME --runtime RUNT
```

NAME The registered name of the Cloud Function you are deploying. **NAME** can only contain letters, numbers, underscores, and hyphens. **NAME** can either be the name of a function in your source code, or it can be a

	custom string (for example, hello-http-function). If you use a custom string, you must also use the --entry-point flag to specify a function contained in your code, to tell the deploy command what function to execute.
--entry-point **ENTRY-POINT**	If you simply want to make your registered NAME different from the name of the executed function. If you don't use this flag, it means that NAME must specify the function in your source code to be executed during deployment. Your registered name will then have the same name as the function.
--runtime **RUNTIME**	The name of the runtime you are using. Example: • nodejs10 • python37 • go111 • java11
TRIGGER	When deploying a new function, you must specify one of --trigger-http, --trigger-topic, or --trigger-bucket, or specify both --trigger-event AND --trigger-resource. You use the flag --trigger-http to deploy HTTP functions. You use the other flags to deploy background functions.

1.2 Creating a Secure HTTP Google Cloud Function

Problem

You want to create your first Secure HTTP cloud function to perform a simple Hello World.

Solution

Configure the cloud function to deny unauthenticated requests.

The steps for creating a secure HTTP cloud function are the same for **Creating a public HTTP Google Cloud Function recipe** with one difference: When you are prompted to allow unauthenticated [helloHttp]? (y/N) **you would Select N.** Selecting **N** restricts only the users you have allowed to access your function.

Discussion

At this point you have successfully created a hello world Cloud Function that is restricted to the users who you add and provided the Cloud Functions Invoker role. To test, navigate to the Cloud Function HTTP URL. To find your URL in your Cloud Console navigate to Cloud Functions, click on the function you deployed. Click on the TRIGGER tab. If you click on the link it should present the following message:

Since you did not provide credentials the function denied you access. To securely access the function, you can test with your email address that you use for Google Cloud. Run the following command

```
curl -X GET "[YOUR_CLOUD_FUNCTION_URL]" -H "Authorization: Bea
```

Hello World” should print on the command line. By default your user account to access Google Cloud has the Cloud Functions Invoker role.

1.3 Accessing Environment Variables at runtime

Problem

You need a way for your code at runtime to access key/value pairs you define.

Solution

Create a cloud functions with environment variables to specify arbitrary key/value pairs at the time of deployment. Which will be surfaced as key/value pairs acessible by your code at runtime.

1. Create and deploy a new cloud function with the code below and allow unauthenticated access to your function.

```
const escapeHtml = require('escape-html');
exports.helloHttp = (req, res) => {
  res.send(process.env.MY_MESSAGE);
};
```

2. Run the following command to configure your environment variable listed as MY_MESSAGE in the command below which will hold a simple string value:

```
gcloud functions deploy sendGrid --set-env-vars MY_MESSA
```

3. Test your cloud function with a simple curl request. You will see the message Hello World which from your code retrieves the

environment variable MY_MESSAGE and returns the string value of the key.

Discussion

You have successfully created an environment variable that holds a value to your assigned key. Environment variables are key/value pairs deployed alongside a function. These variables are scoped to the function and are not visible to other functions in your project. You can add or remove environment variables using both the

gcloud

command-line tool and Cloud Console UI.

1.4 Sending Emails from Cloud Functions with SendGrid

Problem

You need the ability to send emails programmatically from your applications by calling a secure generic REST API.

Solution

Leverage the SendGrid SDK for Node.js to send emails from Google Cloud functions.

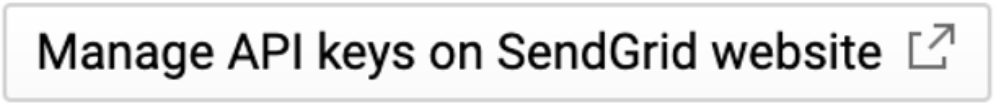
1. Create an empty folder, and run npm init. Get your code ready for the SenGrid function by creating the index.js file and copy the code

below. You will also need to install the @sendgrid/mail npm package in your package.json file.

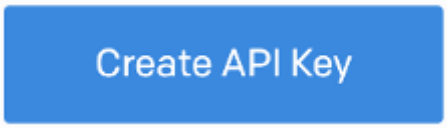
```
const sendgrid = require('@sendgrid/mail');
exports.sendGrid = async (req, res) => {
    console.log('running sendGrid Function')
    try {
        if (req.method !== 'POST') {
            const error = new Error('Only POST requests
            error.code = 405;
            throw error;
        }
        const msg = {
            to: req.body.to,
            from: req.body.from,
            subject: req.body.subject,
            text: req.body.text
        };
        sendgrid.setApiKey(process.env.SENDGRID_API_KEY)
        sendgrid.send(msg)
            .then((response) => {
                console.log(response)
                if (response.statusCode < 200 || respons
                    const error = Error(response.body);
                    error.code = response.statusCode;
                    throw error;
                }
                res.status(200).send(`\n\n Email Sent to
            })
        return Promise.resolve();
    } catch (err) {
        console.error(err);
        const code =
            err.code || (err.response ? err.response.sta
        res.status(code).send(err);
        return Promise.reject(err);
```

```
        }
    };
```

2. Deploy your Cloud Function and set unauthenticated to No.
3. To configure SendGrid you will need to enable the SendGrid API in the Google Cloud Marketplace as well as retrieve your SendGrid API key
4. Use the https://console.cloud.google.com/marketplace to sign up for the SendGrid email service.
5. Once the SendGrid service has been enabled you can retrieve API Keys on the sendGrid website. Click on **Manage API keys on SendGrid website**



6. Within the SendGrid website navigate to Settings > API Keys
7. Click Create API Key



8. Copy the API key created.
9. You will then set a Cloud Function environment variable value with the created API key.
10. Set a Cloud Function environment variable (see Recipe 1-3) which will hold your SendGrid API key, name the key SENDGRID_API_KEY and set the value to your SendGrid API key.

11. Set the Google Cloud Application Default Credentials locally. The below command obtains the user access credentials via a web flow and puts them on your local workstation. This command is useful when you are developing code that would normally use a service account but need to run the code in a local development environment where it's easier to provide user credentials.

```
gcloud auth application-default login
```

12. To test your newly created function run the following CURL command, replace the values within brackets with your settings. Example: replace "[TO_EMAIL]" with "someone@example.org".
13. To find your cloud function url go to Google Cloud Console > Compute > Cloud Functions, select the newly deployed function and select the trigger tab
14. **curl** is a tool to transfer data from or to a server, using a supported protocol. You will be using HTTPS.

```
curl -X POST "[YOUR_CLOUD_FUNCTION_URL]" -H "Authorizati
```

15. You will see the following message on terminal on a successful execution

```
Email Sent to [TO_EMAIL]
```

Discussion

At this point you have successfully created a Cloud Function to send emails with SendGrid, and set the parameters in the curl request to whom you want to send the email to plus the email contents. To allow users you

will need to give the respective user who needs to use your cloud function the Cloud Functions Invoker role, you can follow the recipe Authenticating End-users to add users.

1.5 Deploying Cloud Functions with a GitLab CI/CD Pipeline

Problem

You want an automated way of deploying cloud functions when commiting your code to a git repository.

Solution

Leverage GitLab's CI/CD pipeline to automate your deployment.

You will need the following prior to following the instructions:

1. You will need a GitLab Account
2. You will need to have an empty GitLab repository created and cloned locally to your workstation

To authorize GitLab access to our Google Cloud project we will need to create a Google Cloud service account and assign the required roles to the service account. This service account will authorize GitLab to deploy the Cloud Function to the defined project.

1. Open the IAM & Admin in the Google Cloud Console.

2. On the menu click **Service Accounts**

3. Click **CREATE SERVICE ACCOUNT**

4. Enter your Service account details:

5. 1. Service Account Name
   2. Service Account ID
   3. Service Account Description
   4. When completed it should look like Figure 1-1

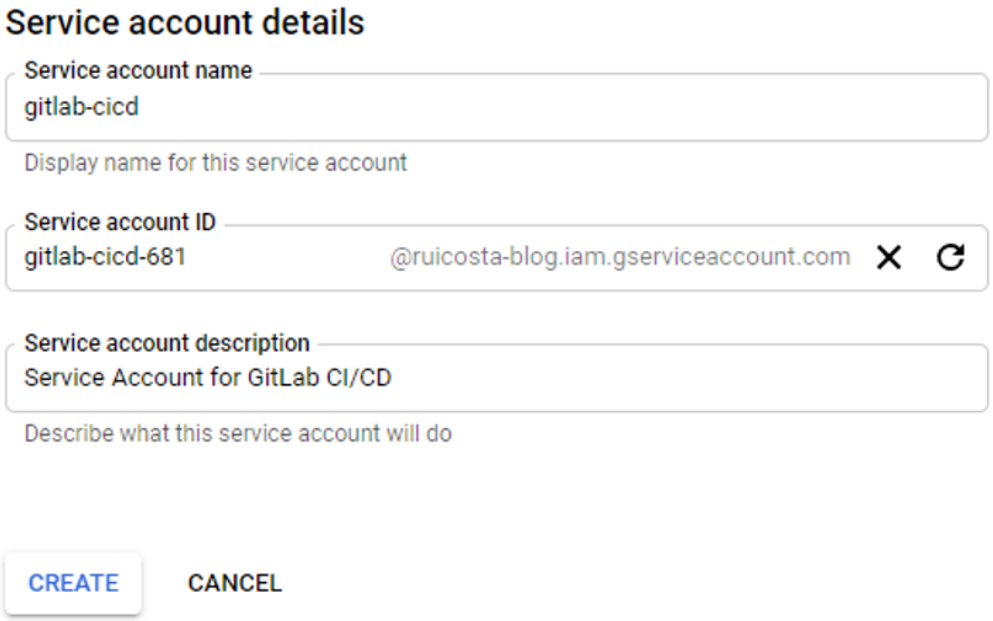
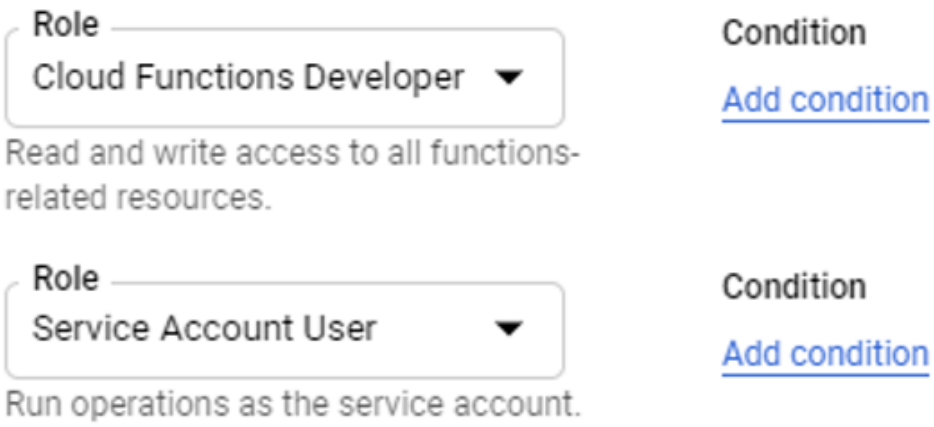


Figure 1-1. Service Account Details Screenshot

6. 1. Click **Create**.
   2. Assign the following roles to the Service Account
   3. Cloud Functions Developer
   4. Service Account User



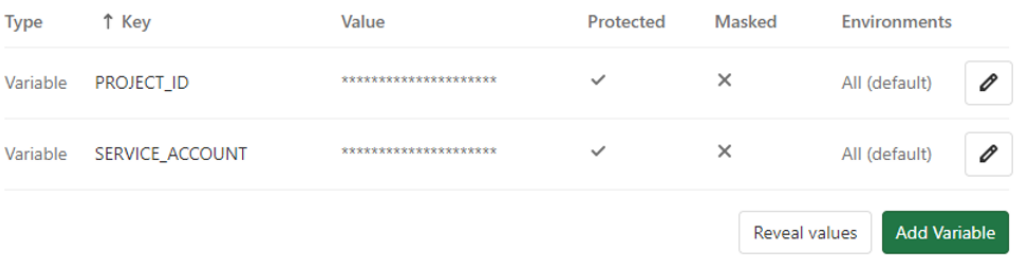
7. 1. Click Continue
   2. Click Done

3. Locate the newly created Service Account, click on the actions icon, and select create a key.

8. Choose JSON and click CREATE. This will download the JSON file to your local workstation.
9. Head over to your GitLab Project, and under settings go to CI/CD
10. Click Expand Variables
11. Create two new variables labeled as:

```
PROJECT_ID
SERVICE_ACCOUNT
```

12. Enter your Google Cloud Project ID
13. Open the JSON file for the service account you downloaded before, copy and paste it's content to the SERVICE_ACCOUNT key. It should look something like Figure :



At this point you have the authorization configured for GitLab to deploy Cloud Functions to your Google Cloud Project. The next step is to prepare your application code to be pushed to the master branch in your GitLab repository.

14. In your empty local GitLab repository run npm init, get your code ready by creating the index.js file and copy the code below.

```
const escapeHtml = require('escape-html');
exports.helloHttp = (req, res) => {
```

```
  res.send(`Hello World!`);
};
```

15. Commit your changes and push the code to your repository
16. If you head back to GitLab CI/CD you should see your job running

17. You should see the following message once the Pipeline has successfully executed

18. Open the Cloud Function in the Google Cloud Console.

19. You should see the Cloud Function successfully deployed.

Discussion

Congratulations you have successfully configured GitLab to Continuous Deploy to Google Cloud Functions when commits are performed on the master branch. **Continuous Delivery** (CD) ensures the delivery of **Continuous Integration (**CI) validated code to your application by means of a structured deployment pipeline. In recipe number 9 you will learn

how to perform **Continuous Integration (**CI) to validate your code prior to deployment of your code to a production environment.

1.6 Responding to SMS Messages with Twilio and Cloud Functions

Problem

You need a programmatic method to reply to SMS messages.

Solution

Create a Cloud Functions to reply to an SMS message using Twilio, a software to send and receive text messages globally.

1. Create an empty folder, run npm init, and get your code ready for the SMS function by creating the index.js file and copy the code below. You will also need to install the “"twilio” npm package in your package.json file.

```
const twilio = require('twilio');
const MessagingResponse = twilio.twiml.MessagingResponse
const projectId = process.env.GCLOUD_PROJECT;
const region = 'us-central1';
const TWILIO_AUTH_TOKEN = process.env.TWILIO_AUTH_TOKEN
exports.reply = (req, res) => {
  let isValid = true;
  if (process.env.NODE_ENV === 'production') {
    isValid = twilio.validateExpressRequest(req, TWILIO_
      url: `https://${region}-${projectId}.cloudfunction
    });
  }
```

```
  if (!isValid) {
    res
      .type('text/plain')
      .status(403)
      .send('Twilio Request Validation Failed.')
      .end();
    return;
  }
  const response = new MessagingResponse();
  response.message('Hello from the Google Cloud Cookbook
  res
    .status(200)
    .type('text/xml')
    .end(response.toString());
};
```

2. Deploy your HTTP Cloud Function
3. Create a Twilio account at the following location https://www.twilio.com/try-twilio
4. In your Twilio console, create a phone number
5. Once you have a phone number, click Manage Numbers and then click on your phone number
6. Under Messaging:
7. Set Configure with to Webhooks/TwiML.

    1. Set A Message Comes In to Webhook and enter the following URL:

        ```
        https://us-central1-[YOUR_PROJECT_ID].cloudfunction
        ```

    2. Click Save.

8. Return to your Twilio Account Settings and take note of the Auth Token for your Live Credentials. You will need it later in this recipe
9. Set a Cloud Function environment variable which will hold your Twilio Auth token, name the key TWILIO_AUTH_TOKEN and set the key to your Twilio Auth token.
10. Send an SMS message to your Twilio number
11. You should receive a response with a message that has been defined in the Cloud Function

Discussion

In this receipt you configured a Cloud Function that is triggered by a webhook request from Twilio when a SMS message is sent to your Twilio phone number. The webhook validates that the request came from Twilio and then sends a simple reply defined in the code. Twilio allows you to send and receive text messages globally and provide you with a pay-as-you-go pricing plan. Twilio also provides you with robust documentation to get you started using their service quickly.

1.7 Unit Testing with GitLab and Cloud Functions

Problem

You need a method to perform testing on your code before it's deployed to production.

Solution

Use Mocha and the GitLab CI/CD to perform unit testing on your code prior to deployment to a production environment.

You will need the following prior to following the instructions:

1. You will need a GitLab Account
2. You will need to have an empty GitLab repository created and cloned locally to your workstation
3. To authorize GitLab access to our Google Cloud project you will need to create a Google Cloud service account and assign the required roles to the service account. You can reference Receipe 1-5 on the steps to do this.
   At this point you have the authorization configured for GitLab to deploy Cloud Functions to your Google Cloud Project. The next step is to prepare your application code to be pushed to the master branch in your GitLab repository.
4. In your empty GitLab repository local folder, create the following:

5.
    1. Create a Folder called test in the root folder of your GitLab repository local folder
    2. Create index.test.js in the newly created test folder
    3. In the root folder of your GitLab repository local folder create a file index.js and .gitlab-ci.yml

6. Copy the code below to the respective files created in step 16

```
test/index.test.js
const assert = require('assert');
const sinon = require('sinon');
const uuid = require('uuid');
const {helloHttp} = require('..');
it('helloHttp: should print a name', () => {
  // Mock ExpressJS 'req' and 'res' parameters
  const name = uuid.v4();
  const req = {
    query: {},
    body: {
```

```
      name: name,
    },
  };
  const res = {send: sinon.stub()};
  // Call tested function
  helloHttp(req, res);
  // Verify behavior of tested function
  assert.ok(res.send.calledOnce);
  assert.deepStrictEqual(res.send.firstCall.args, [`Hell
});
.gitlab-ci.yml
image: google/cloud-sdk:latest
stages:
  - test
  - deploy_production
test:
  stage: test
  script:
    - npm install
    - npm run test
deploy_production:
  stage: deploy
  only:
    - master
  script:
    - echo $SERVICE_ACCOUNT > ${HOME}/gcloud-service-key
    - gcloud auth activate-service-account --key-file ${
    - gcloud --quiet --project $PROJECT_ID functions dep
index.js
const escapeHtml = require('escape-html');
exports.helloHttp = (req, res) => {
  res.send(`Hello ${escapeHtml(req.query.name || req.bod
};
```

7. You will also need to install the “mocha”, “sinon” and “uuid” npm packages in your package.json file.

8. Commit your changes and push the code to your repository
9. If you head back to GitLab CI/CD you should see your job running

10. You should see the following message once the Pipeline has successfully executed (Figure 1-3)

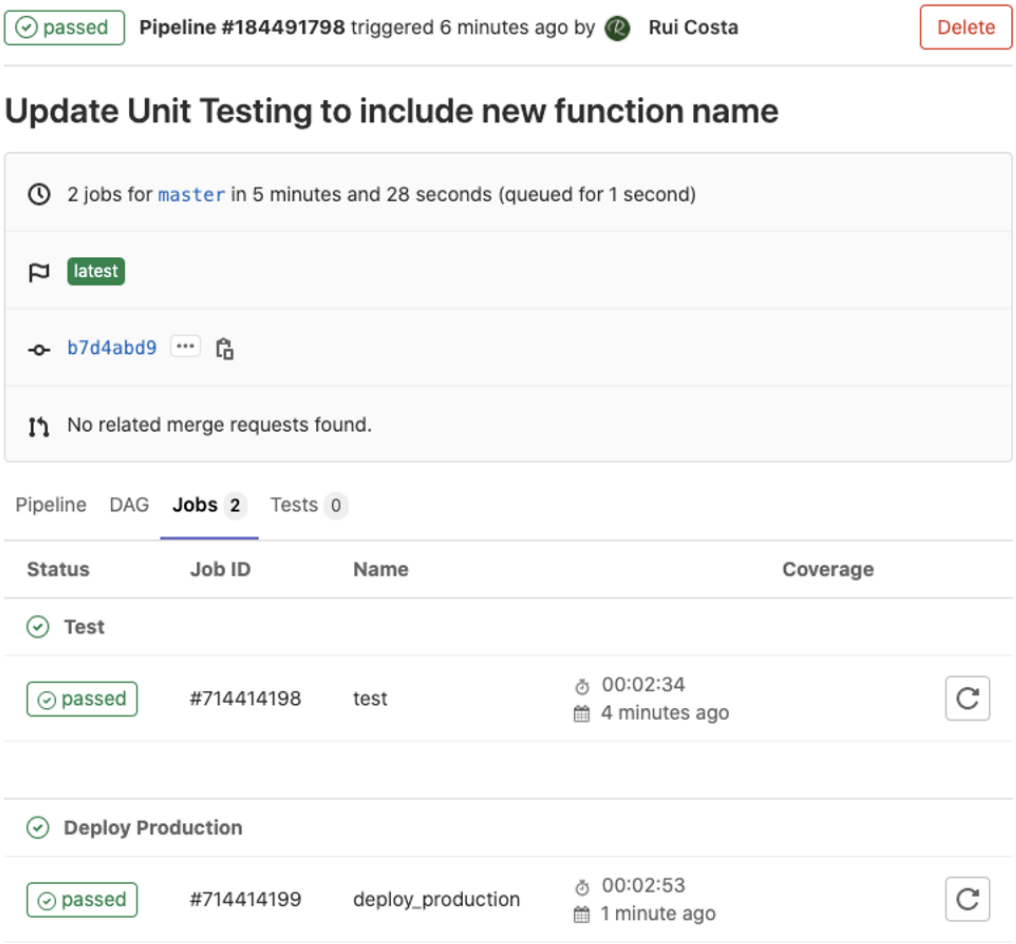
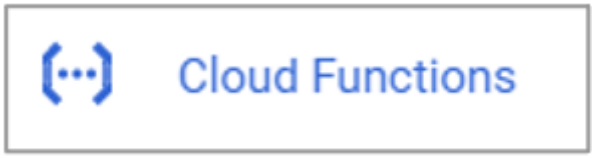


Figure 1-3. GitLab Completed Pipeline

11. Open the Cloud Function in the Google Cloud Console.



12. You should see the Cloud Function successfully deployed.

You have now successfully implemented unit testing, incorporating into the GitLab pipeline and automated the deployment.

Discussion: Understanding the code

In your repository you created a file called .gitlab-ci.yml. The stages section define the stages for your GitLab pipeline, here you have two stages: a test and deploy_production. On the test: stage, the following actions will be performed at runtime. They must succeeded before the next stage runs, if one of the actions fails the pipeline will fail and your code will not be deployed:

- apt update
- apt install -y nodejs npm
- npm install
- npm run test

The npm run test is where you are telling GitLab at runtime to run the test script defined in the package.json. In the sample code the test script executes the index.test.js.

```
"test": "mocha test/index.test.js --exit"
```

If the test fails the GitLab pipeline will fail, however if all tests pass GitLab will deploy the Cloud Function.

1.8 Building an API Gateway to Gather Telemetry Data

Problem

You need a method to gather telemetry data for your API service running on Cloud Functions.

Solution

With this recipe, you will build an Extensible Service Proxy V2 Beta (>ESPv2 Beta) as an API gateway.

1. Make a note of your project ID, the steps following **ESP_PROJECT_ID** as your project id
2. Make a note of the project number, the steps following **ESP_PROJECT_NUMBER** as your project number
3. The code is fairly long, please visit the following github repository to access the code samples.
4. Deploy your Cloud Function and set function to unauthenticated
5. Run the following command to deploy ESPv2 Beta to Cloud Run, change the CLOUD_RUN_SERVICE_NAME to name you want and se the ESP_PROJECT_ID to the one you noted above

```
gcloud run deploy CLOUD_RUN_SERVICE_NAME \
--image="gcr.io/endpoints-release/endpoints-runtime-serv
--allow-unauthenticated \
--platform managed \
--project=ESP_PROJECT_ID
```

6. On successful completion, the command displays a message similar to the following:

```
Service [esphello] revision [esphello-00001-zuf] has bee
```

7. In this example, https://esphello-6bc24kwh7a-uc.a.run.app is the **CLOUD_RUN_SERVICE_URL** and esphello-6bc24kwh7a-uc.a.run.app is the **CLOUD_RUN_HOSTNAME**.
8. Make a note of **CLOUD_RUN_HOSTNAME**. You will specify **CLOUD_RUN_HOSTNAME** in the host field of your OpenAPI yaml file.
9. You can verify that the initial version of ESPv2 Beta is deployed on Cloud Run by visiting the **CLOUD_RUN_SERVICE_URL** in your web browser. You should see a warning message about a missing environment variable. This warning message is expected.
10. Open openapi-functions.yaml file in your favorite IDE
11. In the address field in the x-google-backend section, replace **REGION** with the Google Cloud region where your function is located, **FUNCTIONS_PROJECT_ID** with your Google Cloud project ID and **FUNCTIONS_NAME** with your function name. For example:

```
x-google-backend:
  address: https://us-central1-ruicosta-blog.cloudfunctions.net/hello
  protocol: h2
```

Figure 1-4. Screenshot of openapi-functions.yaml

12. In the host field, specify **CLOUD_RUN_HOSTNAME**, the hostname portion of the URL that Cloud Run created when you deployed ESPv2 Beta. For example:

```
swagger: '2.0'
info:
  title: Cloud Endpoints + GCF
  description: Cloud Endpoints with a Google Cloud Functions
  version: 1.0.0
host: esphello-6bc24kwh7a-uc.a.run.app
```

Figure 1-5. Screenshot of openapi-functions.yaml

13. To deploy the Endpoints configuration, run the following command, replace the ESP_PROJECT_ID with yours.

14. ```
    gcloud endpoints services deploy openapi-functions.yaml
        --project ESP_PROJECT_ID
    ```

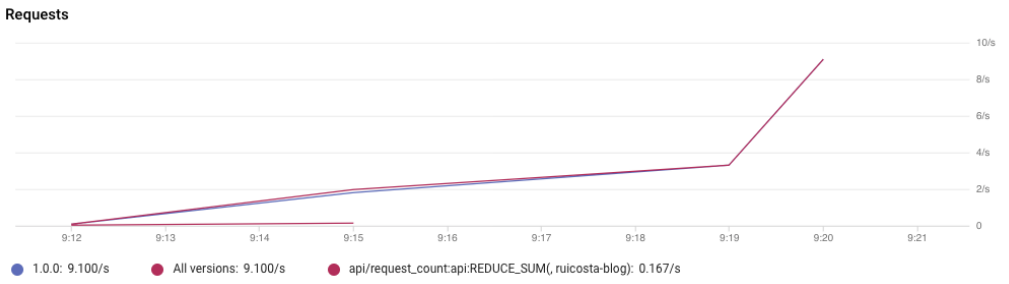
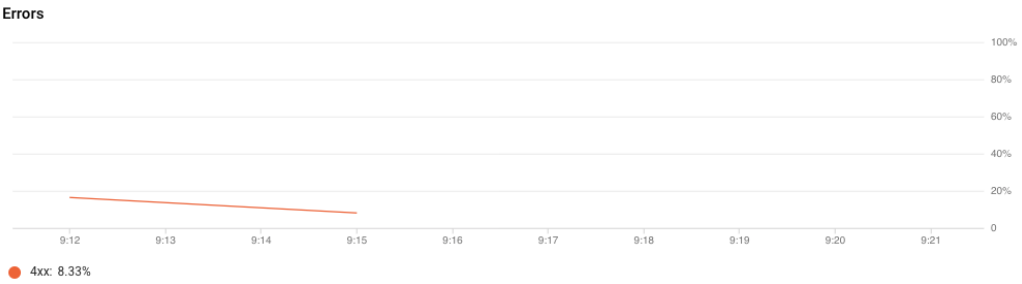
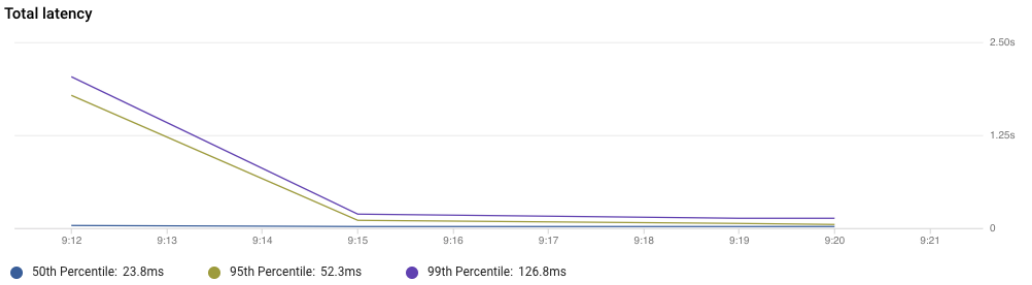
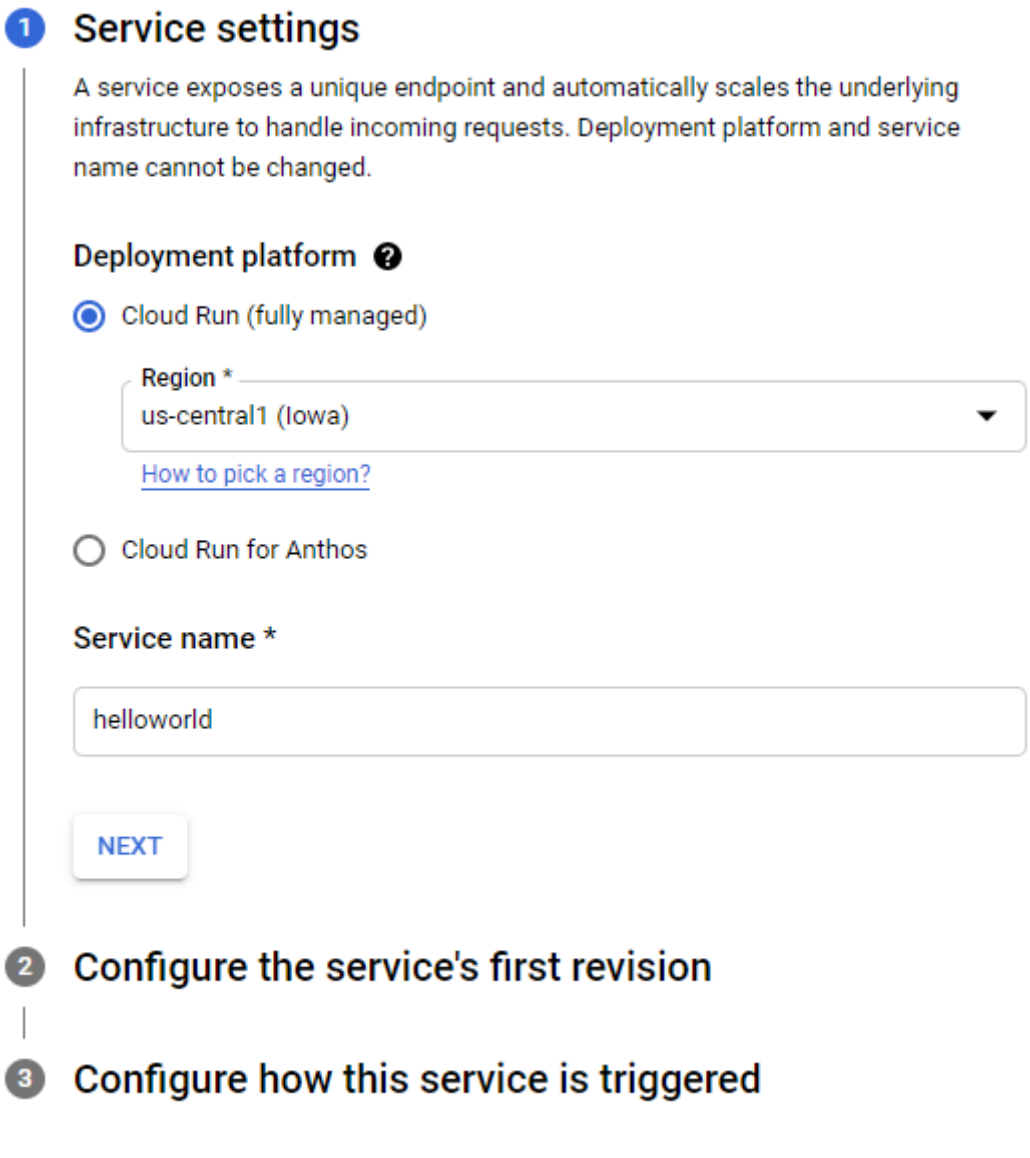
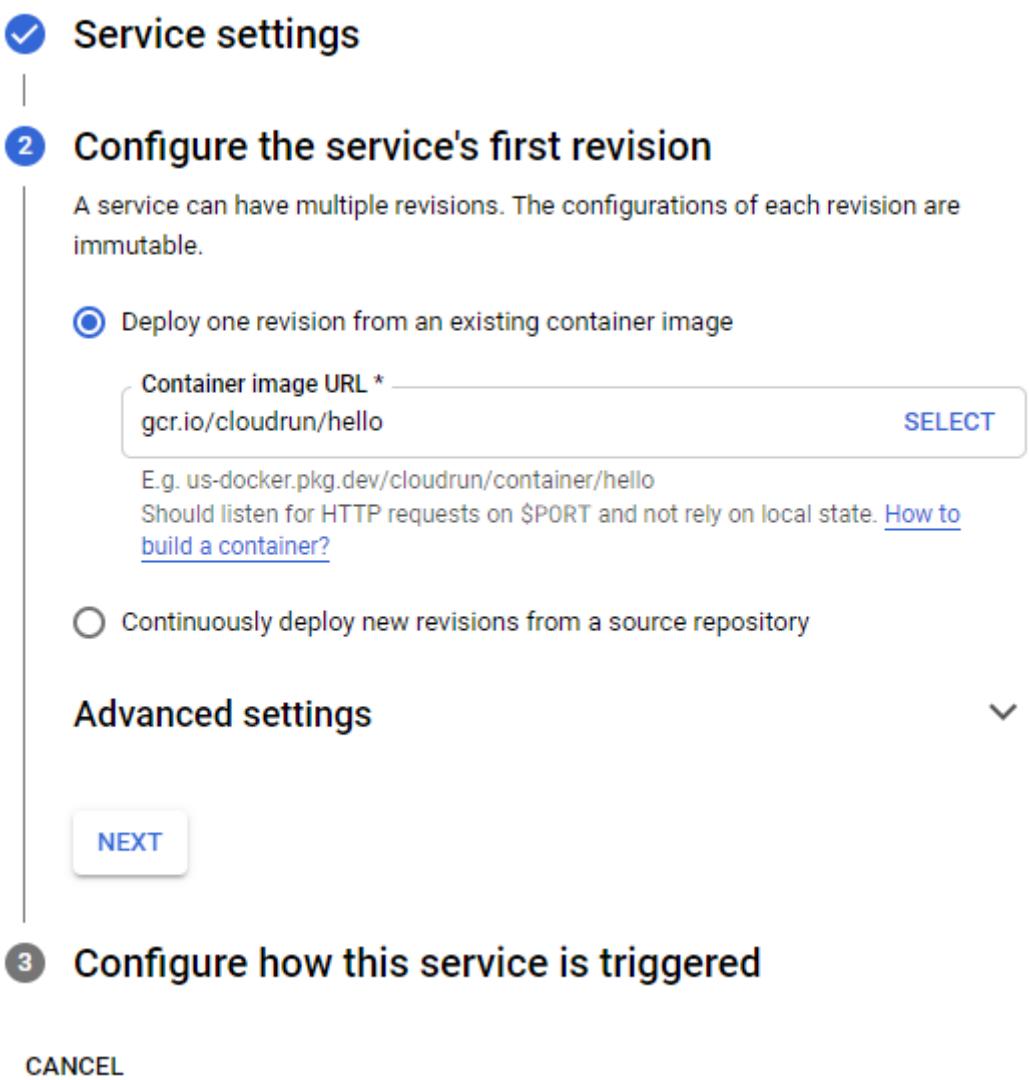
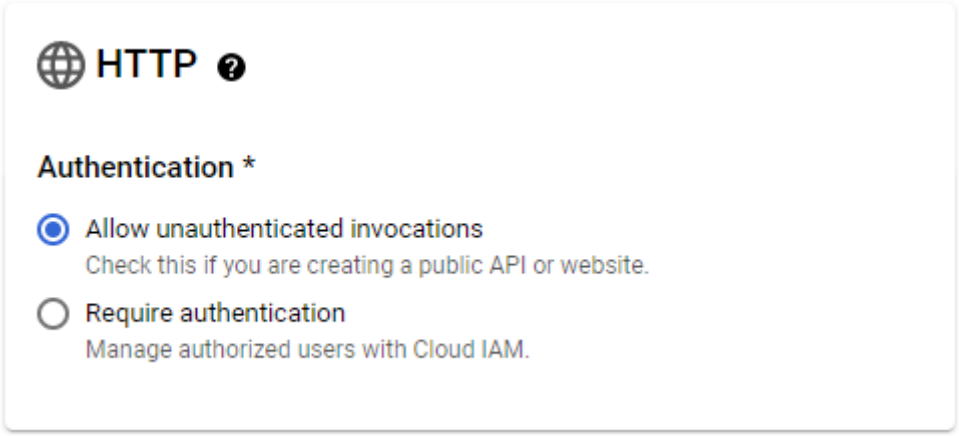
15. Keep a note of the **CONFIG_ID we will need this later**, For example the CONFIG_ID for the example below is 2020-09-05-r3

```
Service Configuration [2020-09-05r3] uploaded for service [esphello-6bc24kwh7a-uc.a.run.app]
```

Figure 1-6. Screenshot of gcloud endpoints services deploy output

16. To enable your Endpoints service run the following command:

    ```
    gcloud services enable ENDPOINTS_SERVICE_NAME
    ```

17. To determine the **ENDPOINTS_SERVICE_NAME** you can either go to the **Endpoints** page in the Cloud Console. The list of possible **ENDPOINTS_SERVICE_NAME** are shown under the **Service name** column.
18. To build the service config into a new ESPv2 Beta docker image run the following command:
    ```

```
chmod +x gcloud_build_image
./gcloud_build_image -s CLOUD_RUN_HOSTNAME \
    -c CONFIG_ID -p ESP_PROJECT_ID
```

19. For **CLOUD_RUN_HOSTNAME**, specify the hostname of the URL that Cloud Run created when you deployed ESPv2 Beta above, enter the CONFIG_ID you noted in step 14. Example:

```
chmod +x gcloud_build_image
./gcloud_build_image -s esphello-6bc24kwh7a-uc.a.ru
 -c 2020-09-05r3 -p ruicosta-blog
```

20. The output should look like:

Figure 1-7. Screenshot of gcloud_build_image bash script output

21. Keep a note of the the full URI after “serverless:”, you will use that as your **ESP_VERSION-CLOUD_RUN_HOSTNAME-CONFIG_ID in the next command**
22. Redeploy the ESPv2 Beta Cloud Run service with the new image. Replace **CLOUD_RUN_SERVICE_NAME** with the same Cloud Run service name you used when you originally deployed it above in Deploying ESPv2 Beta:

23.
```
gcloud run deploy esphello-6bc24kwh7a-uc.a.run.app \
--image="gcr.io/ESP_PROJECT_ID/endpoints-runtime-serverl
--allow-unauthenticated \
--platform managed \
--project=ESP_PROJECT_ID
Example:
gcloud run deploy esphello \
```

```
--image="gcr.io/ruicosta-blog/endpoints-runtime-serverle
--allow-unauthenticated \
--platform managed \
--project=ruicosta-blog
```

24. To test your deployed Cloud Function with Cloud Endpoints run the following command:

```
curl --request GET \
   --header "content-type:application/json" \
   "https://YOUR_ENDPOINT_HOST/hello"
```

25. You can view telemetry data for your function, go to Endpoints > Services page in the Google Cloud Console. Below are some of the telemetry data you will collect for your API.



Figure 1-8. Endpoint Requests



Figure 1-9. Endpoint Errors



Figure 1-10. Endpoint Latency

Method	Requests ↓	4xx	5xx	Latency 99%	Latency 95%	Latency median	Avg resp size	Avg req size
GET /hello	861	0	0	151 ms	61 ms	25 ms	508 B	1241 B

Figure 1-11. Endpoint Summary

Discussion

With this recipe Cloud Endpoint intercepts all requests to your function and performs any necessary checks (such as authentication) before invoking the function. When the function responds, ESPv2 Beta gathers and reports telemetry.

Chapter 2. Google Cloud Run

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at ruiscosta@google.com and dhodun@google.com.

Google Cloud Run is a serverless compute solution that allows you to run your containerized applications without having to manage the underlying infrastructure. Cloud Run is built on the same foundation as the open source community project Knative. Knative allows developers to focus on their application code without having to worry about the management of compute resources. Google Cloud Run also provides features such as autoscaling, redundancy, logging, and custom domains which enterprises require to run production workloads.

Google Cloud provides you with two ways to run Cloud Run: Cloud Run (fully managed); and Cloud Run for Anthos. Cloud Run (fully managed) allows you to deploy containers without having to worry about the underlying infrastructure. Cloud Run for Anthos allows you to run your containers on-premises or on a Google Cloud Kubernetes Engine cluster. Running on a GKE cluster does require the management of the cluster but does provide additional benefits such as custom machine types and

additional networking support. This chapter, however, will focus on Cloud Run (fully managed).

All code samples for this chapter are located at https://github.com/ruiscosta/google-cloud-cookbook/chapter-3. You can follow along and copy the code for each individual recipe by going to the folder with that recipe's number

2.1 Deploying a Prebuilt Hello World Container

Problem

You want to deploy your first Cloud Run prebuilt container application to perform a simple Hello World.

Solution

Leverage an existing container image running on Google Cloud Source Repository located at gcr.io/cloudrun/hello to deploy a new Cloud Run container that responds to incoming web requests.

1. Open the Cloud Run in the Google Cloud Console.

1. Click **Create service** to start the create service form process.
2. Select a region and enter a name. In the image below, the service name is **helloworld.**



1. Click **Next.**
2. For the container image url enter us.gcr.io/cloudrun/hello.



1. Click **Next**.
2. For authentication select **Allow unauthenticated invocations.**



1. Click **Create**
2. Click the URL displayed to launch your newly deploy Cloud Run container
3. You should see the following page in your browser

It's running!

Congratulations, you successfully deployed a container image to Cloud Run

Discussion

You have successfully deployed your first Cloud Run service from an existing container. It is a simple hello world application, but demonstrates some of the basic concepts in deploying Cloud Run services.

2.2 Building your own Hello World Container

Problem

You want to create a container for your application and deploy it to Google Cloud Run.

Solution

Build a container with your application code, deploy it to Google Cloud Container Registry and finally deploy your newly deployed container to Cloud Run.

1. Create a new directory on your local machine called **helloworld**
2. Initialize a **go.mod** file to declare the go module, type the code below to your **go.mod** file

3. ```
   module github.com/GoogleCloudPlatform/golang-samples/run
   go 1.13
   ```

4. Create a new file named **main.go** and type the following code into it:

   ```
   package main
   import (
           "fmt"
           "log"
           "net/http"
           "os"
   )
   ```
   ```

```
func main() {
        log.Print("starting server...")
        http.HandleFunc("/", handler)
        // Determine port for HTTP service.
        port := os.Getenv("PORT")
        if port == "" {
                port = "8080"
                log.Printf("defaulting to port %s", port
        }
        // Start HTTP server.
        log.Printf("listening on port %s", port)
        if err := http.ListenAndServe(":"+port, nil); er
                log.Fatal(err)
        }
}
func handler(w http.ResponseWriter, r *http.Request) {
        name := os.Getenv("NAME")
        if name == "" {
                name = "World"
        }
        fmt.Fprintf(w, "Hello %s!\n", name)
}
```

5. To containerize the application, create a new file named **Dockerfile** in the same directory as the source files, and type the following code:

```
FROM golang:1.15-buster as builder
WORKDIR /app
COPY go.* ./
RUN go mod download
COPY . ./
RUN go build -mod=readonly -v -o server
FROM debian:buster-slim
RUN set -x && apt-get update && DEBIAN_FRONTEND=noninter
    ca-certificates && \
```

```
    rm -rf /var/lib/apt/lists/*
COPY --from=builder /app/server /app/server
CMD ["/app/server"]
```

6. Create a .dockerignore file to exclude defined files from your container

```
# Exclude locally vendored dependencies.
vendor/
# Exclude "build-time" ignore files.
.dockerignore
.gcloudignore
# Exclude git history and configuration.
.gitignore
```

7. Build your container with the sample application image using Cloud Build. Run the following command from the directory containing the Dockerfile, replace **PROJECT-ID** with your Google Cloud Project ID

```
gcloud builds submit --tag gcr.io/PROJECT-ID/helloworld
```

8. Deploy the Cloud Run service using the newly deployed container image by running the following command

```
gcloud run deploy --image gcr.io/PROJECT-ID/helloworld -
```

9. Enter to accept the default name, helloworld.
10. Select the region of your choice, for example us-central1.

11. You will be prompted to **allow unauthenticated invocations**: respond y
12. Once the command completed successfully, the command will display the Cloud Run service URL. You can click or copy the URL and open in a Browser.

Discussion

You have successfully created a container image, deployed it to Google Cloud Container Registry and created a Cloud Run service from the deployed container image. In this recipe you also learned how to deploy a Cloud Run service with the gcloud command versus using the Google Cloud Console interface.

2.3 Using Cloud Run with a custom domain

Problem

You don't want to use the generated URL that Google Cloud provides for your Google Cloud service and prefer to use your own domain.

Solution

Use your own domain to map a domain, or subdomain to the Cloud Hello World created in **receipe 1**.

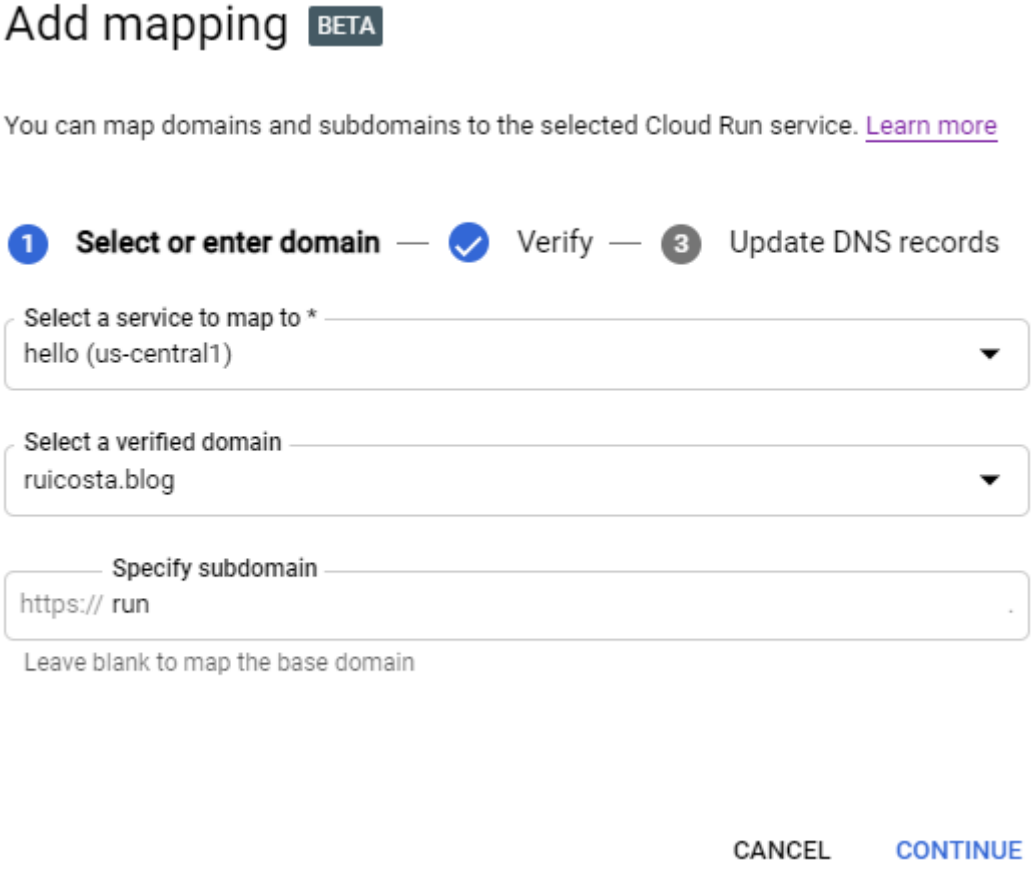
1. Verify ownership of your domain. Do list the currently verified domains run the following command:

```
gcloud domains list-user-verified
```

2. If the command does not list any output that means you do not have any verified domains.
3. Run the following command to verify your domain.

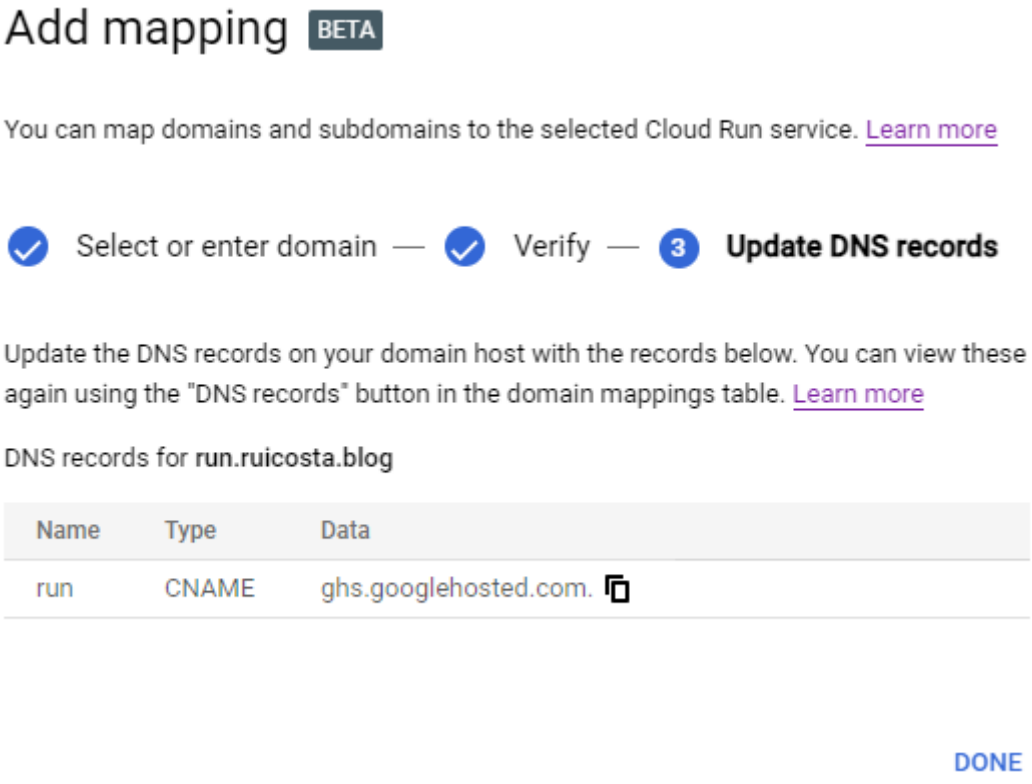
```
gcloud domains verify BASE-DOMAIN
Example: gcloud domains verify ruicosta.blog
```

4. **Note: If you want to map say run.ruicosta .blog , you will need to verify the root domain which is ruicosta.blog .**
5. The command will open a new Browser window that will take you to the Google Webmaster Central. Follow the instructions to verify your domain.
6. Open the **MANAGE CUSTOM DOMAINS** page in the Cloud Run services page within Google Cloud Console.
7. Click **ADD MAPPING.**
8. Choose your service to map to.
9. Select your verified domain.
10. Optionally: Choose a subdomain. You can choose to map to the root of the verified domain.

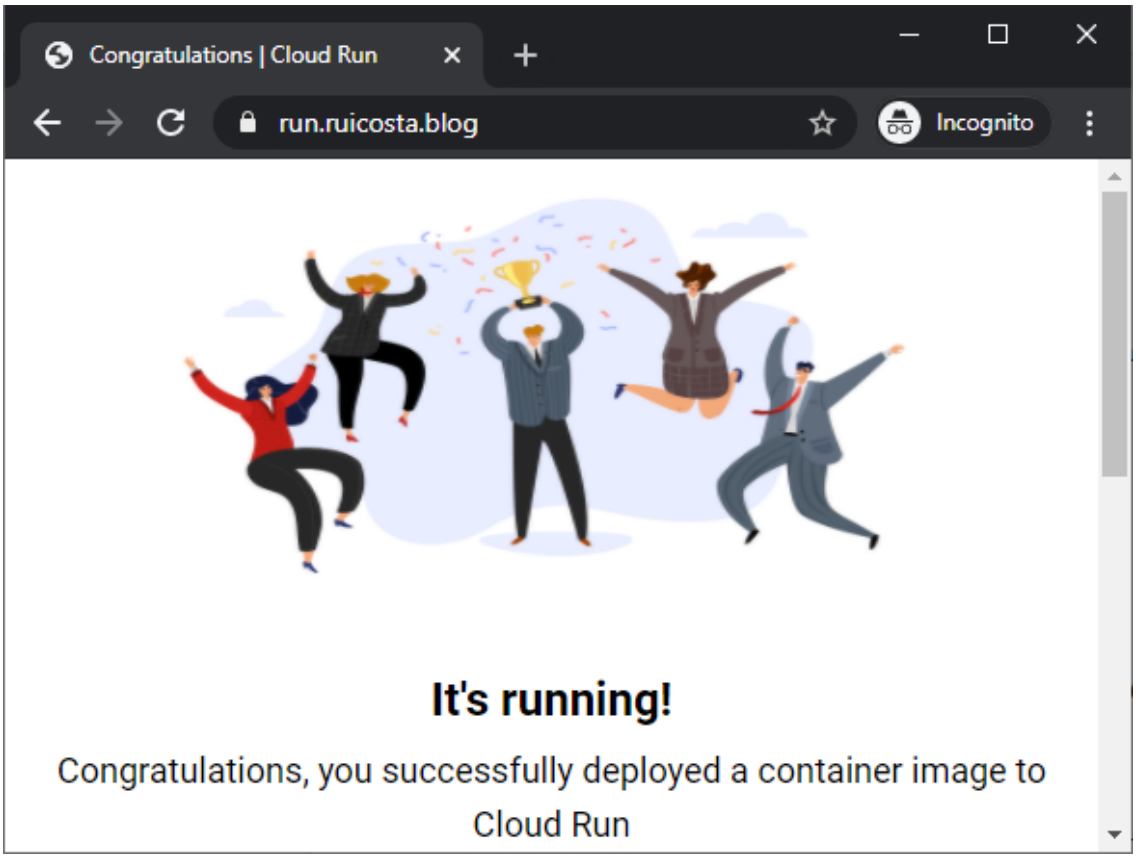


1. **Click Continue**

Update your DNS records at your domain registrar web site using the DNS records displayed in the last step.



1. Test your new custom domain mapping.
2. Example: Go to https://run.ruicosta.blog
3. Please note for your custom domain mapping it might take a few minutes for it to be updated with the newly verified domain.



Discussion

With custom domain mapping you can direct users to your Cloud Run service running with your domain name versus using the Google Cloud domain name. Benefits including; branding, easy to read and spell domain names. Please visit the Google Cloud Run documentation as custom domain names are not supported in all regions, check whether your region supports custom domains.

2.4 Triggering a Cloud Run from Cloud Pub/Sub

Problem

You want to trigger an operation when a message is published to a Cloud Pub/Sub topic. For example; A new message is received from your application by Cloud Pub/Sub and you want to execute an operation on the arrived message.

Solution

Set up Cloud Run to listen to arriving messages in Cloud Pub/Sub. On new message Cloud Run can run an operation on the newly arrived message.

1. Create a new Pub/Sub topic by running the following command:

```
gcloud pubsub topics create chapter-3-4-topic
```

2. Create a new directory on your local machine called **chapter-3-4**
3. Initialize a **go.mod** file to declare the go module, type the code below to your **go.mod** file

```
module github.com/GoogleCloudPlatform/golang-samples/run
go 1.14
```

4. Create a new file named **main.go** and type the following code into it:

```
package main
import (
        "encoding/json"
        "io/ioutil"
        "log"
        "net/http"
        "os"
)
```

```
func main() {
        http.HandleFunc("/", HelloPubSub)
        // Determine port for HTTP service.
        port := os.Getenv("PORT")
        if port == "" {
                port = "8080"
                log.Printf("Defaulting to port %s", port
        }
        // Start HTTP server.
        log.Printf("Listening on port %s", port)
        if err := http.ListenAndServe(":"+port, nil); er
                log.Fatal(err)
        }
}
type PubSubMessage struct {
        Message struct {
                Data []byte `json:"data,omitempty"`
                ID   string `json:"id"`
        } `json:"message"`
        Subscription string `json:"subscription"`
}
func HelloPubSub(w http.ResponseWriter, r *http.Request)
        var m PubSubMessage
        body, err := ioutil.ReadAll(r.Body)
        if err != nil {
                log.Printf("ioutil.ReadAll: %v", err)
                http.Error(w, "Bad Request", http.Status
                return
        }
        if err := json.Unmarshal(body, &m); err != nil {
                log.Printf("json.Unmarshal: %v", err)
                http.Error(w, "Bad Request", http.Status
                return
        }
        name := string(m.Message.Data)
        if name == "" {
                name = "World"
        }
```

```
        log.Printf("Hello %s!", name)
}
```

5. **Note:** You return a code from the service such as HTTP 200 or 204, acknowledge complete processing of the Pub/Sub message. Error codes, such as HTTP 400 or 500, indicate the message will be retried.
6. To containerize the application, create a new file named **Dockerfile** in the same directory as the source files, and type the following code:

```
FROM golang:1.15-buster as builder
# Create and change to the app directory.
WORKDIR /app
# Retrieve application dependencies.
COPY go.* ./
RUN go mod download
# Copy local code to the container image.
COPY . ./
# Build the binary.
RUN go build -mod=readonly -v -o server
FROM debian:buster-slim
RUN set -x && apt-get update && DEBIAN_FRONTEND=noninter
    ca-certificates && \
    rm -rf /var/lib/apt/lists/*
# Copy the binary to the production image from the build
COPY --from=builder /app/server /server
# Run the web service on container startup.
CMD ["/server"]
```

7. Build the container image and push it to Google Cloud Container Registry:

```
gcloud builds submit --tag gcr.io/PROJECT_ID/pubsub
```

8. Deploy the Cloud Run service from the newly deployed container image, replace **PROJECT_ID** with your Cloud project ID.

```
gcloud run deploy pubsub --image gcr.io/PROJECT_ID/pubsu
```

9. Enter to accept the default name, pubsub.
10. Select the region of your choice, for example us-central1.
11. You will be prompted to **allow unauthenticated invocations**: respond N. You will provide Pub/Sub the invoker service account permission so it can invoke the Cloud Run service.
12. Enable the Pub/Sub service to generate authentication tokens in your project:

```
gcloud projects add-iam-policy-binding PROJECT_ID \
--role=roles/iam.serviceAccountTokenCreator
```

13. Replace **PROJECT_ID** with your Google Cloud project ID.
14. Replace **PROJECT_NUMBER** with your Cloud project number.
15. Create a service account that the Pub/Sub subscription will use.

```
gcloud iam service-accounts create cloud-run-pubsub-invo
     --display-name "Cloud Run Pub/Sub Invoker"
```

16. Give the newly created service account permission to invoke your **pubsub** Cloud Run service:

```
gcloud run services add-iam-policy-binding pubsub \
--member=serviceAccount:cloud-run-pubsub-invoker@PROJECT
   --role=roles/run.invoker
```

17. Replace **PROJECT_ID** with your Google Cloud project ID.
18. Create a Pub/Sub subscription with the service account:

```
gcloud pubsub subscriptions create myRunSubscription --t
--push-endpoint=SERVICE-URL/ \
--push-auth-service-account=cloud-run-pubsub-invoker@PRO
```

19. Replace the **SERVICE-URL** with the Cloud Run service URL.
20. Replace the **PROJECT_ID** with your Google Cloud project ID.
21. To test the newly deployed Cloud Run service, we will trigger it with the newly deployed Cloud Pub/Sub subscription:

```
gcloud pubsub topics publish chapter-3-4-topic --message
```

22. Navigate to the Cloud Run service logs:
23. Click the **pubsub** service.
24. Select the **Logs** tab.
25. Look for the “Hello Test Runner!” message.

Discussion

1. By leveraging Cloud Pub/Sub you are able to invoke a Cloud Run service to perform operation. This process allows you to trigger an operation only when a message is received by Cloud Pub/Sub which becomes a powerful feature to perform background operations when certain events occur.

2.5 Deploy a Web application to Cloud Run

Problem

1. You need a method of deploying a static web application to Cloud Run.

Solution

1. Create a Cloud Run service that uses an nginx container to serve your web files.
2. Create a new directory on your local machine called **nginx**
3. Create the following directories in the root directory of of nginx; **html** and **nginx**
4. Create a new file named **index.html** in the **html** directory and type the following code into it:

```
<html>
<body>Hello from Google Cloud Run and NGINX</body>
</html>
```

1. Create a new file named **default.conf** in the **nginx** directory and type the following code into it:

```
server {
    listen       8080;
    server_name  localhost;
    location / {
        root   /usr/share/nginx/html;
        index  index.html index.htm;
```

```
        }
        # redirect server error pages to the static page /50
        error_page   500 502 503 504  /50x.html;
        location = /50x.html {
            root   /usr/share/nginx/html;
        }
    }
```

2. To containerize the application, create a new file named **Dockerfile** in the same directory as the source files, and type the following code:

```
FROM nginx
COPY html /usr/share/nginx/html
COPY nginx/default.conf /etc/nginx/conf.d/default.conf
```

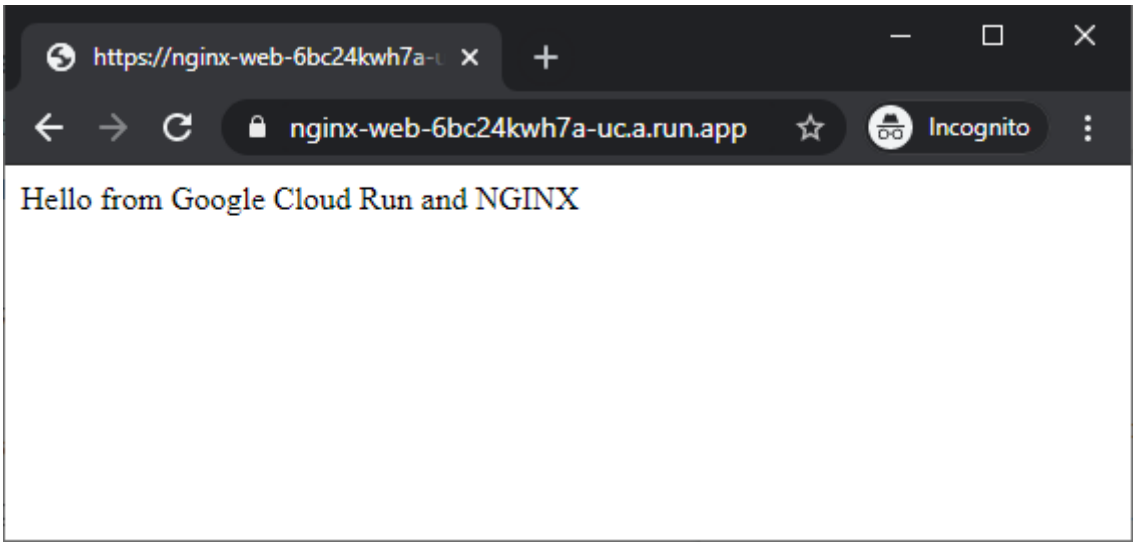
1. **Notes:** In this **Dockerfile** we are creating a new container based on the **nginx** container, adding the static html file plus setting the **nginx** configuration file.
2. Build your container by running the following command from the directory containing the Dockerfile, replace **PROJECT-ID** with your Google Cloud Project ID

3. `gcloud builds submit --tag gcr.io/PROJECT-ID/nginx-web`

4. Deploy the Cloud Run service using the newly deployed container image by running the following command

5. `gcloud run deploy --image gcr.io/PROJECT-ID/nginx-web --`

6. Enter to accept the default name, nginx-web.
7. Select the region of your choice, for example us-central1.
8. You will be prompted to **allow unauthenticated invocations**: respond Y
9. Once the command completed successfully, the command will display the Cloud Run service URL. You can click or copy the URL and open in a Browser.
10. You should now see the newly deployed container serving your html content with nginx



Discussion

Google Cloud provides other services to serve static and dynamic websites, but Cloud Run is a good option in cases where some customization is needed to the underlying runtime.

2.6 Rolling Back a Cloud Run Service Deployment

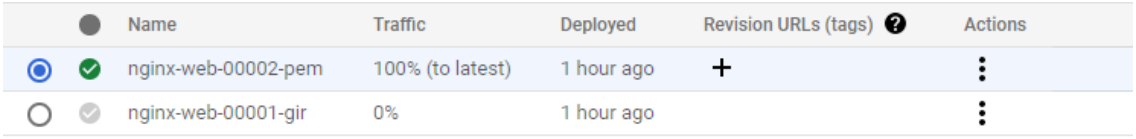
Problem

You need to rollback your Cloud Run service deployment due to a bug you found in your code.

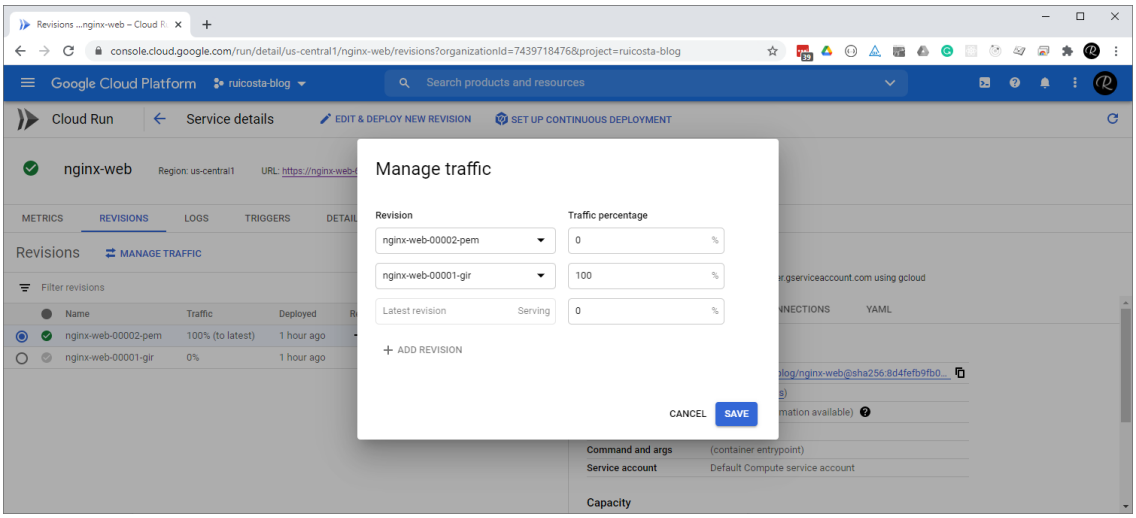
Solution

Cloud Run allows you to deploy multiple revisions of your application. With this feature Cloud Run provides you the ability to rollback to a previous revision.

1. To roll back to a previous revision go to the Google Cloud console and navigate to Cloud Run.
2. Locate the service you need to rollback.
3. The revisions tab will list all the available versions of your service.



1. In the list of revisions, select the revision you are rolling back:
2. Click Manage Traffic
3. Select the previous revision you want to roll back to in the dropdown list.
4. Set that previous revision's traffic percentage to 100.
5. Set the currently serving revision's percentage to 0.
6. Click **Save**.



1. Once you click save, the traffic will be transitioned to the version you selected allowing you to rollback to older versions.

Discussion

With the ability to rollback to previous Cloud Run versions it provides the capability to quickly fix errors in the current version being served.

2.7 Cloud Run Gradual Rollouts

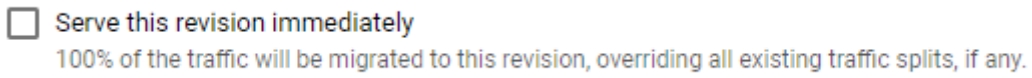
Problem

You need to deploy Cloud Run in a blue-green deployment.

Solution

Cloud Run allows you to split traffic between multiple versions of your Cloud Run service which gives you the capability to provide you with blue-green deployments.

1. To roll out a new Cloud Run service version gradually navigate to the Cloud Run services in the Google Cloud console.
2. Locate the service which you want to deploy a new revision gradually.
3. Click Deploy New Revision
4. Fill out the parameters as needed, make sure the checkbox labelled **Serve this revision immediately** is UNCHECKED.



1. Click Deploy
2. Click Manage Traffic
3. The new revision is listed but with a 0 percentage set: it is currently not serving any traffic since you unchecked Service this revision immediately.
4. In the Manage Traffic page:
   Set it to the desired percentage for the new revision, for example, 10.
   Click **Save**.
   Repeat these Manage Traffic steps but with changed percentages, increasing the percentage as needed for the new revision. You do not need to redeploy to change the traffic percentages.

Discussion

With the ability to gradually roll out with a blue-green deployment, you can validate your updated revision is working as expected. If something was to occur with the new revision, only the percentage of the served traffic would be impacted. If you need to, you can rollback 100 percent of the traffic by following receipe 6.

2.8 Cloud Run Configuration Parameters

Problem

You need to fine tune Cloud Run parameters as to how many maximum instances are running, setting CPU allocation and adjusting the request timeout.

Solution

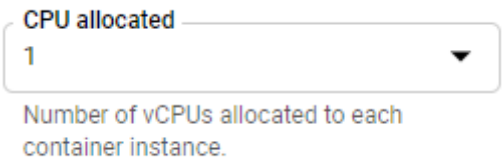
Cloud Run provides you with many options for tuning your service including setting request timeouts, how many instances to run and even updating CPU allocation all through the Google Cloud console.

CPU Allocation

Cloud Run by default sets 1 CPU is allocated for each container instance. You can adjust the CPU allocation using the Cloud Console.

1. Navigate to the Cloud Run services in the Google Cloud console.
2. Click on the service you want to adjust the CPU allocation for and click **Edit and Deploy New Revision**.

Under Advanced Settings, click **Container**.



1. Select the desired CPU allocation from the dropdown list.
2. Click **Deploy**.

Request Timeout The request timeout specifies the time which a response must be returned by Cloud Run. If a response isn't returned within the time specified, the request returns an error 504. The default timeout response for Cloud Run is 5 minutes.

1. Navigate to the Cloud Run services in the Google Cloud console.
2. Click on the service you want to adjust the request timeout for and click **Edit and Deploy New Revision**.

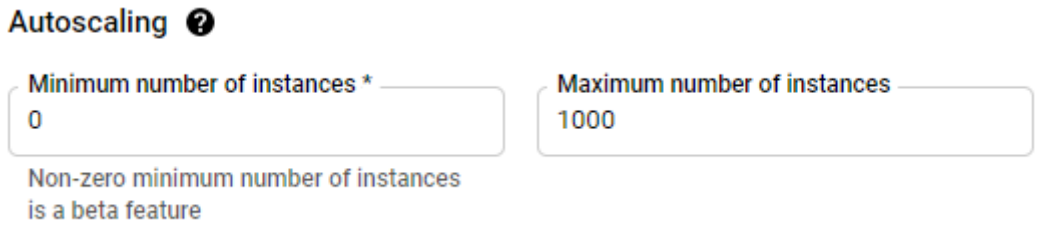
Under Advanced Settings, click **Container**.

1. In the **Request timeout** field, enter the timeout value that you want to use in seconds.
2. Click **Deploy**.

Maximum Number of Instances Maximum number of instances in Cloud Run allows you to limit the scaling of the service in response to incoming requests. You can use this setting to control your costs or control connections to say backend services.

1. Navigate to the Cloud Run services in the Google Cloud console.
2. Click on the service you want to adjust the maximum number of instances for and click **Edit and Deploy New Revision**.

Under Advanced Settings, click **Container**.

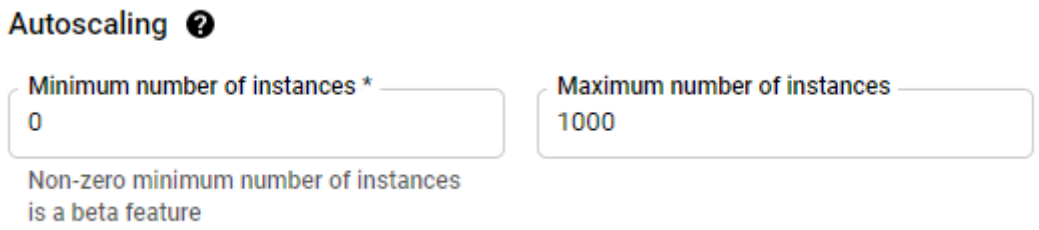


1. Enter the Maximum number of instances. You can use any value from 1 to 1000.
2. Click **Deploy**.

Minimum Number of Instances Cloud Run scales to the number of instances based on the number of incoming requests.If your service requires reduced latency and you need to limit the number of cold starts, you can set the minimum number of container instances to be kept warm and ready to serve requests.

1. Navigate to the Cloud Run services in the Google Cloud console.
2. Click on the service you want to adjust the minimum number of instances for and click **Edit and Deploy New Revision**.

Under Advanced Settings, click **Container**.



1. Enter a value for the Minimum number of instances. This will allow the value of instances listed to be kept warm and ready to receive requests.

2. Click **Deploy**.

Chapter 3. Google App Engine

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at ruiscosta@google.com and dhodun@google.com.

Google App Engine is a serverless compute solution that allows you to run your applications without having to manage the underlying infrastructure. App Engine supports Node.js, Java, Ruby, C#, Go, Python, PHP, or you can bring your own language runtime. App Engine has two editions: Standard and Flexible. With Flexible you can run your custom runtimes that allow you to bring any library and framework to App Engine.

App Engine provides you with enterprise-ready deployment features such as application versioning, traffic splitting, security, monitoring and debugging. With App Engine, all you need to focus on is your code; Google Cloud manages the underlying infrastructure.

All code samples for this chapter are located at https://github.com/ruiscosta/google-cloud-cookbook/04-appengine. You can follow along and copy the code for each individual recipe by going to the folder with that recipe's number.

3.1 Deploy a Hello World to App Engine Standard

Problem

You want to deploy your first App Engine application application to perform a simple Hello World.

Solution

Using Google Cloud command line and your favorite editor to build a simple Express.js application to run on App Engine.

1. On your local workstation create a temporary folder to hold the files you will need to create the hello world application.
2. In your favorite IDE create an app.js file in the root of the directory you created on step 1 and copy the code below to the file.

```
'use strict';
const express = require('express');
const app = express();
app.get('/', (req, res) => {
        res.status(200).send('Hello, world!').end();
});
const PORT = process.env.PORT || 8080;
app.listen(PORT, () => {
        console.log(`App listening on port ${PORT}`);
        console.log('Press Ctrl+C to quit.');
});
module.exports = app;
```

3. Now create an app.yaml file in the root of the same directory and copy the code below to the file. The app.yaml file defines the settings for your application including the runtime of your code.

```
runtime: nodejs14
```

4. Now create a package.json file in the root of the same directory and copy the code below to the file.

```
{
  "name": "appengine-hello-world",
  "engines": {
    "node": ">=14.0.0"
  },
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "express": "^4.17.1"
  }
}
```

5. In your terminal run the following command in the root directory you created in step 1: npm install
6. To deploy the application to App Engine Standard run the following command: gcloud app deploy
7. To view your deployed application, run the following command: gcloud app browse

Discussion

You have successfully deployed your first App Engine application using Node.js. It is a simple hello world application, but demonstrates some of

the basic concepts in deploying App Engine services.

3.2 Deploy a Hello World to App Engine Flexible

Problem

You want to deploy an App Engine application running as a container to perform a simple Hello World.

Solution

App Engine Flexible supports running a Docker container that can include custom runtimes or other source code written in a different programming language. Since App Engine Flexible supports running Docker containers, you will use the Flexible version of App Engine to deploy a simple hello world.

1. On your local workstation create a temporary folder to hold the files you will need to create the hello world application.
2. In your favorite IDE create a **Dockerfile** in the root of the directory you created in step 1 and copy the code below to the file.

```
FROM nginx
COPY nginx.conf /etc/nginx/nginx.conf
RUN mkdir -p /var/log/app_engine
RUN mkdir -p /usr/share/nginx/www/_ah && \
    echo "healthy" > /usr/share/nginx/www/_ah/health
ADD www/ /usr/share/nginx/www/
RUN chmod -R a+r /usr/share/nginx/www
```

3. **Note:** The FROM command builds a base image using the official nginx docker image.
4. Create a new file named **app.yaml** in the root of your temporary directory and type the following code into it:

```
runtime: custom
env: flex
```

5. Now create a new file named **nginx.conf** also in the root of your temporary directory you create and type the following code into it:

```
events {
    worker_connections 768;
}
http {
    sendfile on;
    tcp_nopush on;
    tcp_nodelay on;
    keepalive_timeout 65;
    types_hash_max_size 2048;
    include /etc/nginx/mime.types;
    default_type application/octet-stream;
    access_log /var/log/app_engine/app.log;
    error_log /var/log/app_engine/app.log;
    gzip on;
    gzip_disable "msie6";
    server {
        listen 8080;
        root /usr/share/nginx/www;
        index index.html index.htm;
    }
}
```

6. Create a new folder called **www** in the root of your temporary directory

7. Within the www folder create a new file called **index.html** and copy the code below to it:

```
<!doctype html>
<html>
  <head>
    <title>Hello World!</title>
  </head>
  <body>
    <h1>Welcome to nginx!</h1>
    <p>Brought to you by Google App Engine.</p>
  </body>
</html>
```

8. Run the following command to install the gcloud App Engine extension for Go: gcloud components install app-engine-go
9. Run the following command to deploy your application to App Engine: gcloud app deploy
10. To see your app running run the following command to launch your browser: gcloud app browse

Discussion

You have successfully deployed a static web application running on nginx web server as a custom runtime running on App Engine Flexible. App Engine Flexible is a perfect choice for applications that:

- Applications that run in a Docker container that includes a custom runtime
- Depends on frameworks that include native code.
- Accesses the resources or services of your Google Cloud project that reside in the Compute Engine network

The table below summarizes at a high level the differences between App Engine Standard and Flexible:

Feature	Standard environment	Flexible environment
Instance startup time	Seconds	Minutes
SSH debugging	No	Yes
Scaling	Manual, Basic, Automatic	Manual, Automatic
Scale to zero	Yes	No, minimum 1 instance
Modifying the runtime	No	Yes (through Dockerfile)
Deployment time	Seconds	Minutes
WebSockets	No	Yes
Supports installing third-party binaries	Yes	Yes

3.3 Securing your application with Identity-Aware Proxy (IAP)

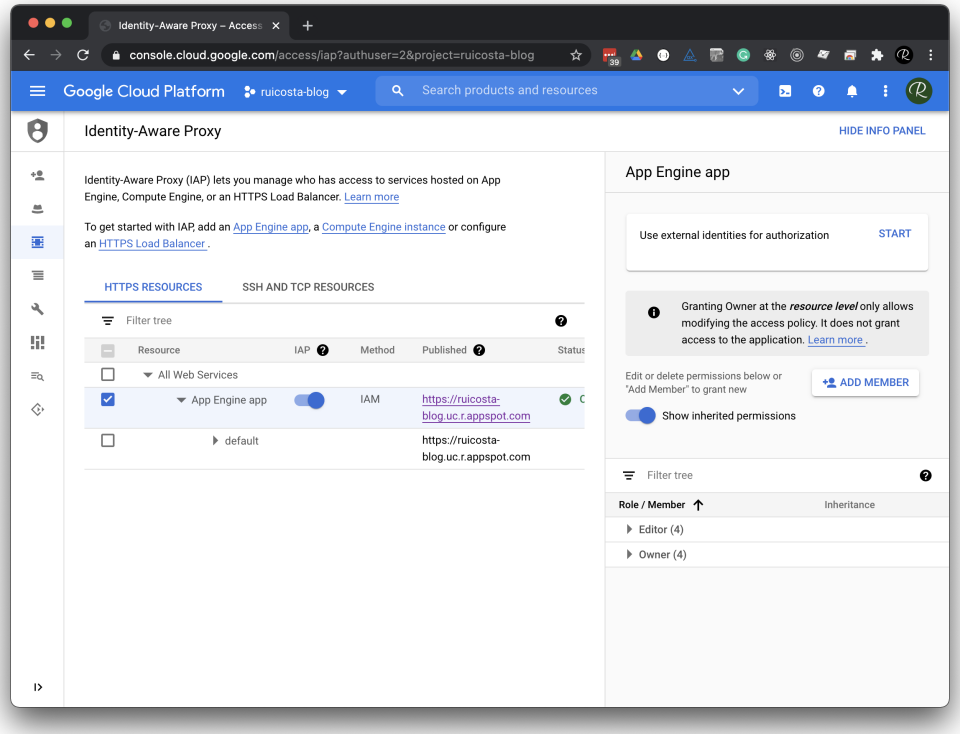
Problem

You've deployed your Hello World application running on App Engine Flexible and want only certain users to be able to access it.

Solution

You will use Google Cloud Identity-aware proxy (iap) to restrict access to only a set of predefined users. You will use the Cloud Hello World application created in **recipe 1** to secure with IAP.

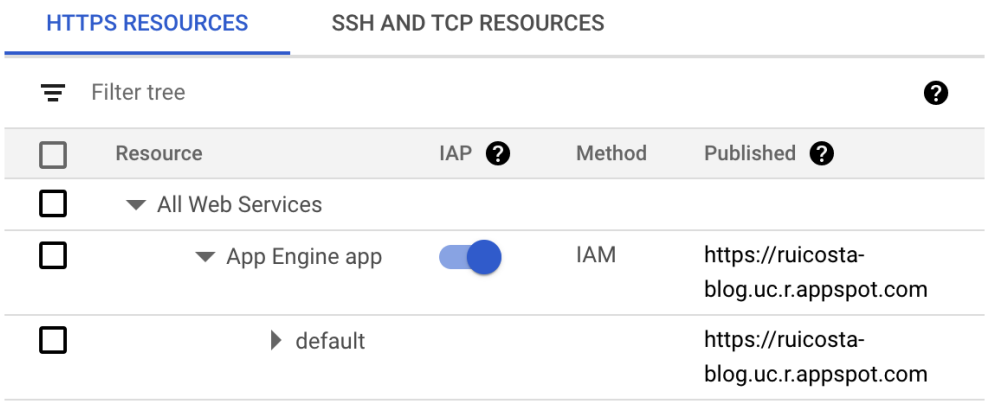
1. Go to the Identity-Aware Proxy page in the Google Cloud Console.
2. **Note:** If you haven't configured your OAuth consent screen, you'll be prompted to do so.
3. Select the resource you want to secure by checking the box to its left.
4. On the right side panel, click **Add Member**.



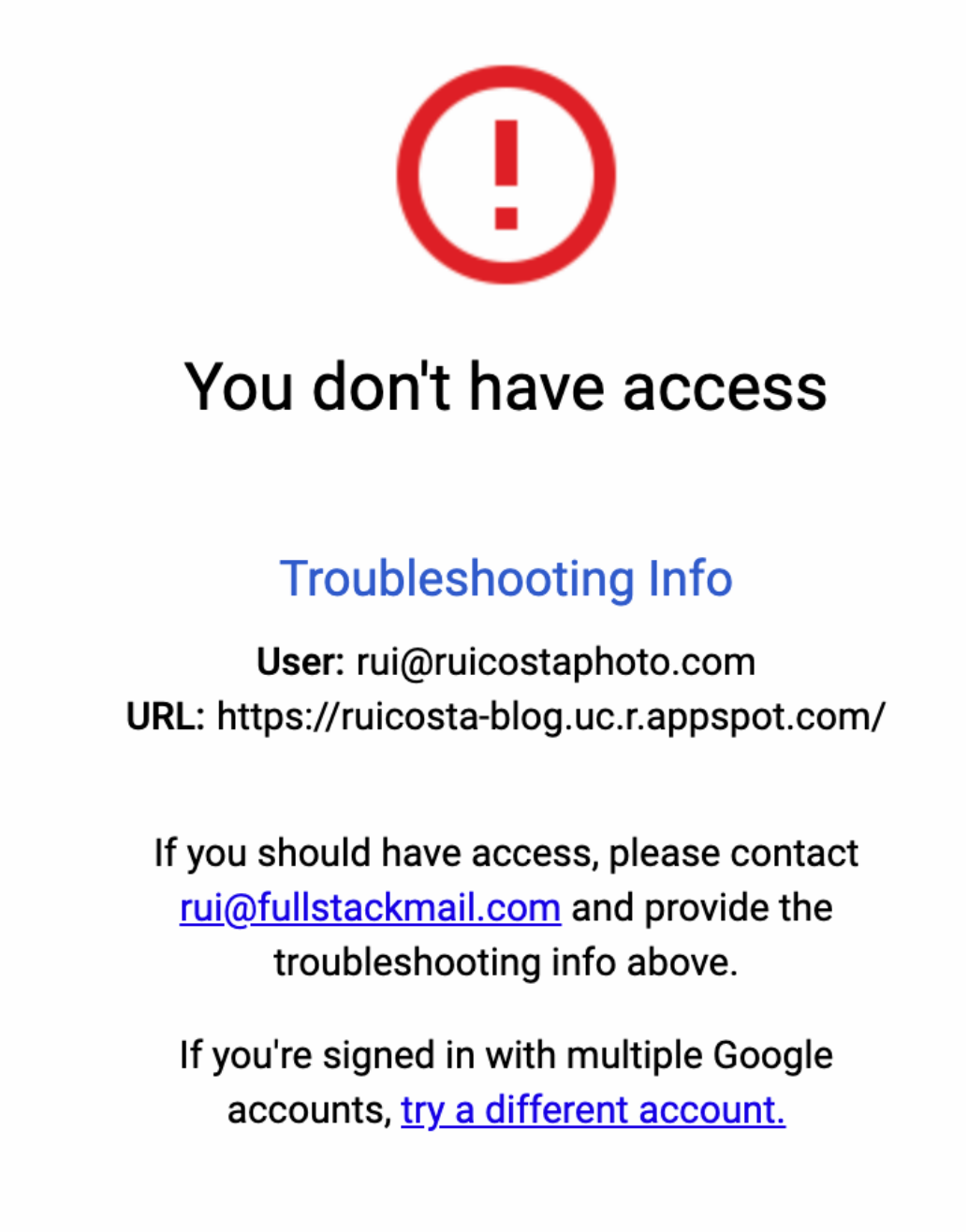
5. Add the email addresses of groups or individuals to whom you want to grant access to your App Engine application.
6. IAP IAM supports the following accounts:

    1. Google Account: user@gmail.com
    2. Google Group: admins@googlegroups.com
    3. Service account: server@example.gserviceaccount.com
    4. G Suite domain: example.com

7. Click Add when you completed adding the accounts to access your application.
8. On the IAP page, under HTTPS Resources, find the App Engine app you want to restrict access to and toggle the on/off switch in the IAP column.



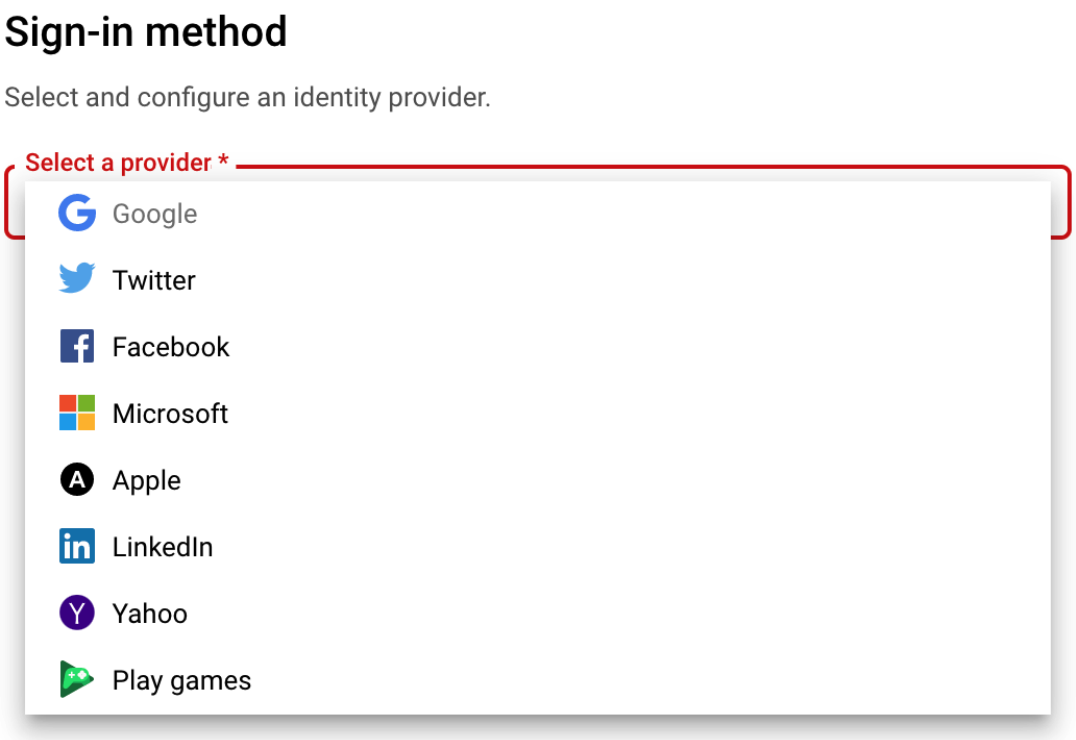
9. Access the URL for your App Engine application. You should be prompted to sign in, if you're not prompted try sign in with an Incognito window.
10. If you have not granted access to an account and they try to access your application they will receive the following message:



11. If you have authorized a user account and they sign in with the associated account they will have full access to your application running on App Engine.

Discussion

With Google Cloud Identity-aware proxy you can restrict access to your application running on App Engine, preventing unauthorized access to your resources. Identity-aware proxy also supports external identities such as Google, Microsoft, Email/Password and others which provide a robust set of sign-in options for your users to access your application.



3.4 Custom Domains with App Engine

Problem

You want to use your own custom domain rather than the default address that App Engine provides for you.

Solution

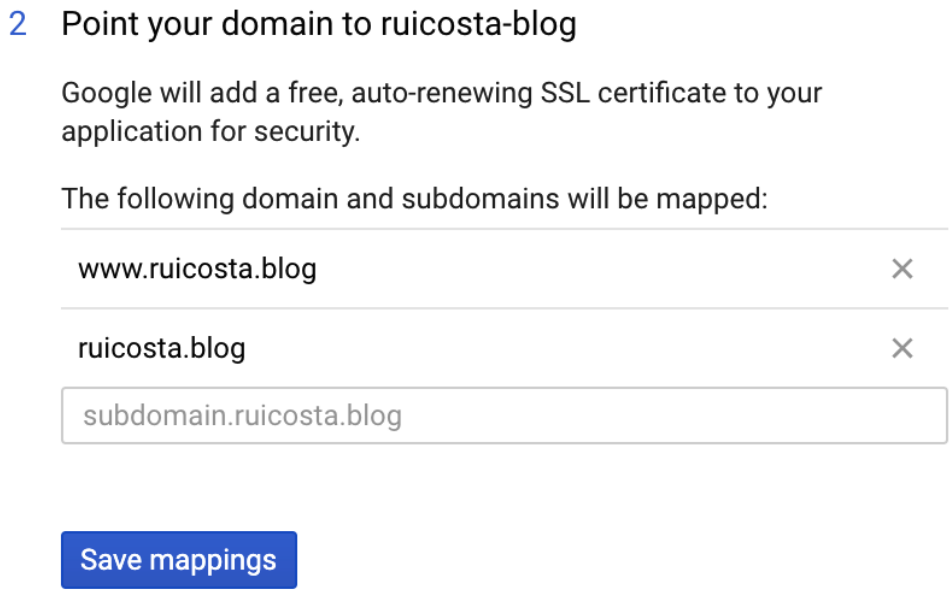
Google Cloud provides you the ability to map custom domains as well it issues a managed certificate for SSL for HTTPS connections. You will use the Cloud Hello World application created in recipe 1 to enable your custom domain.

Note: You will require a custom domain for this recipe.

1. In the Google Cloud Console, go to App Engine → Settings →Custom Domains.
2. Click Add a custom domain.
3. If your domain name has been verified, the domain name will appear in the dropdown. Select the domain from the drop-down menu and click Continue.
4. If you haven't verified your domain name follow these steps to verify:

    1. Select Verify a new domain from the drop-down menu.
    2. Enter your domain name and click Verify.
    3. Enter the required information in the Webmaster page.

5. After you complete the steps above in the Webmaster page, you will then return to the Add a new custom domain page in the Google Cloud Console.
6. In the Point your domain to section, specify the domain and or subdomains that you want to map. Click Save mappings when you completed adding all your mappings.



7. Click Continue to see your domain's DNS records.
8. Sign in to your domain registrar web site and update your DNS records with the records displayed.
9. Test by browsing to your new mapped domain, for example https://www.example.com.
10. **Note**: That it can take several minutes for the automatic SSL certificate to be issued.

Discussion

By mapping a custom domain you can enable your App Engine application to align with your branding as well as keep a secure site, since Google Cloud will provide an SSL certificate for your mapped domain.

3.5 Using Machine Learning APIs with App Engine

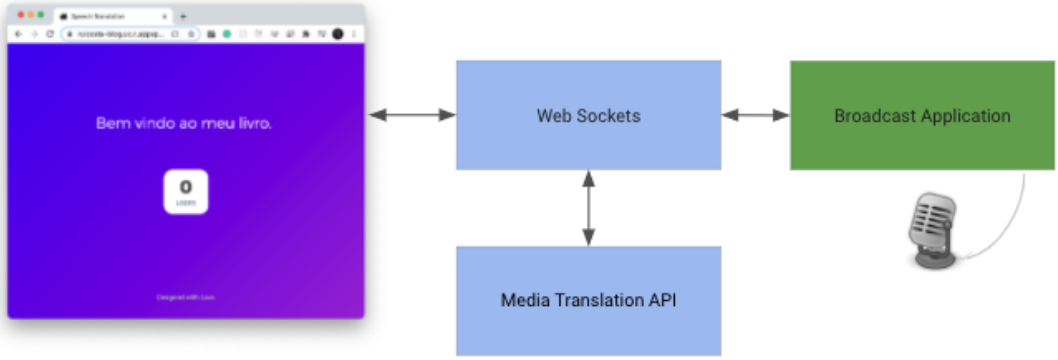
Problem

You need to build a real-time translation application for voice.

Solution

Google Cloud offers a Media Translation API that adds real-time audio translation to your applications. You will build two applications, a broadcast application and a client application in this recipe using App Engine to host your application.

Diagram



Notes

This recipe requires you to git clone the following repository: https://github.com/ruiscosta/google-cloud-cookbook/

1. In the cloned application go to 04-appengine/4-5-media
2. In your IDE edit the following file **client/client.js** and replace [PROJECT_ID] with your Google Cloud project:

```
const socket = io.connect('https://[YOUR_PROJECT_ID].uc.
```

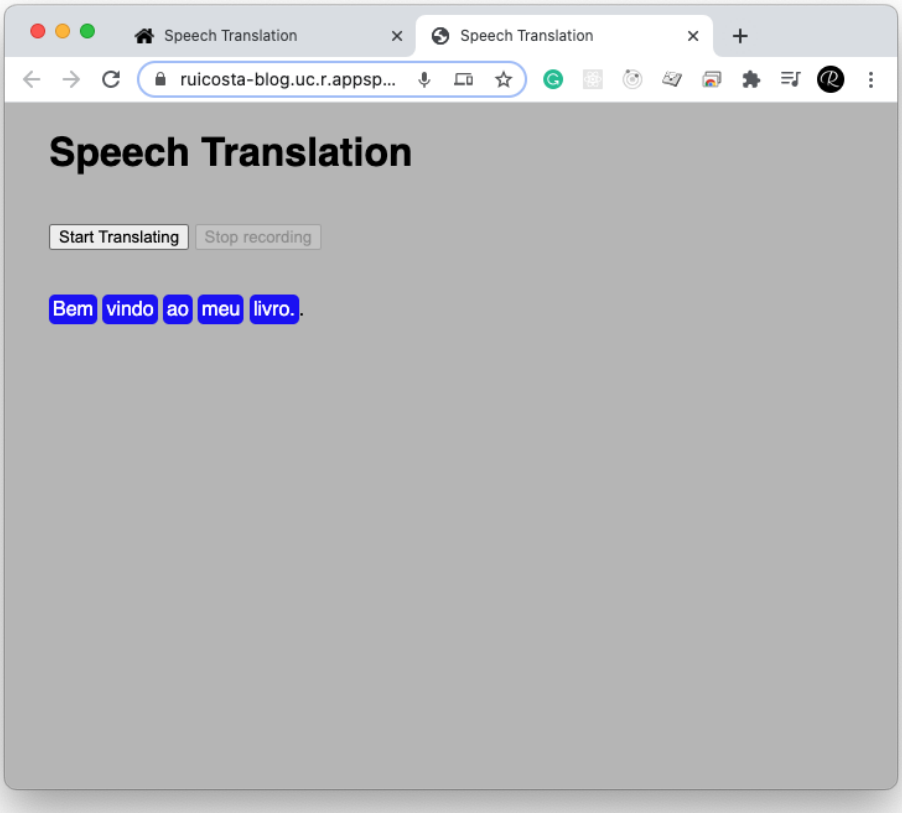
3. Repeat the process in step 2 but edit the broadcast/client.js file
4. Enable the Media Translation API in the Google Cloud Console
5. Deploy your App Engine application by running the following command: gcloud app deploy
6. In the app.js file in the root directory, you will notice the following Expres.js routes declared. The client path is for the users reading the translations from the persona broadcasting. The person broadcasting would visit the root for the App Engine application.

```
app.use('/', express.static('broadcast'))
app.use('/client', express.static('client'))
```

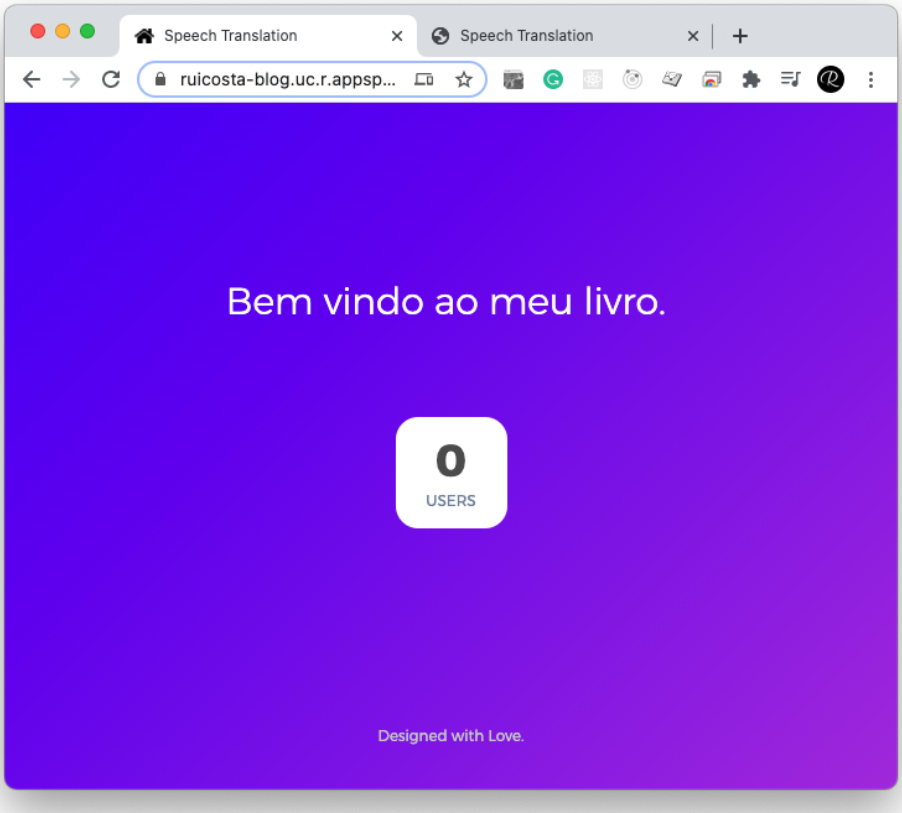
7. Once your application is deployed visit the root path and open a new tab with the /client path.
8. In the broadcast application click start translating and start speaking in English and watch the translation on the second tab. The translation is from English to Portugusse.
9. You can change the languages in app.js file, line 29 and 30

```
const sourceLanguage = 'en-US';
const targetLanguage = 'pt-BR';
```

10. Here is a sample of the Broadcast Application



11. Here is a sample of the Client Application



Discussion

In this recipe you used an existing repository to deploy a translation application to App Engine. This application uses Express.js, Websockets and the Media Translation API to allow realtime audio to be translated to the language you define in the code. Since we are using WebSockets we used App Engine Flexible since Standard does not support WebSockets. WebSockets allowed the real time communication of the broadcaster and users.

3.6 Cube.js and React Dashboards with App Engine

Problem

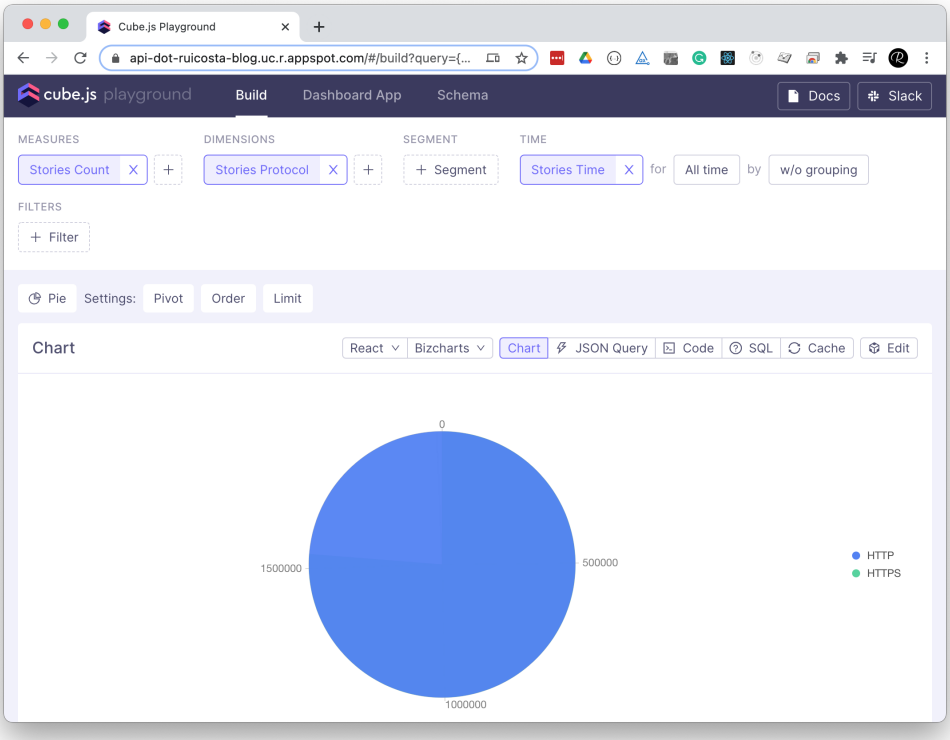
You want a secure method to display charts/graphs to users from aggregated BigQuery data.

Solution

App Engine provides for a great platform to run any of your applications. For this recipe you will use Cube.js, BigQuery and App Engine to build a user interface for viewing charts/graphs from data stored in your BigQuery dataset.

Note:

- You will be using Cube.js which is a Open Source analytical API platform, you can learn more about Cube.js at https://cube.dev/
- You will also learn how to deploy React.js to App Engine since the user dashboards will be running with the React.js framework.



1. On your local workstation create a temporary folder to hold the files you will need to create App Engine user dashboards.
2. In your temporary folder using the Cube.js CLI run the following command to create a new Cube.js application for BigQuery: npx cubejs-cli create real-time-dashboard -d bigquery
3. You will need credentials to access BigQuery, in the Google Cloud Console create a new service account. Add the BigQuery Data Viewer and BigQuery Job User roles to this service account and then generate a new JSON key file. Copy the JSON key to the root of the real-time-dashboard folder.
4. In your IDE edit the real-time-dashboard/.env file to include your Google Cloud Project as well as the location of your key file:

```
CUBEJS_DB_BQ_PROJECT_ID=example-google-project
CUBEJS_DB_BQ_KEY_FILE=./examples.json
```

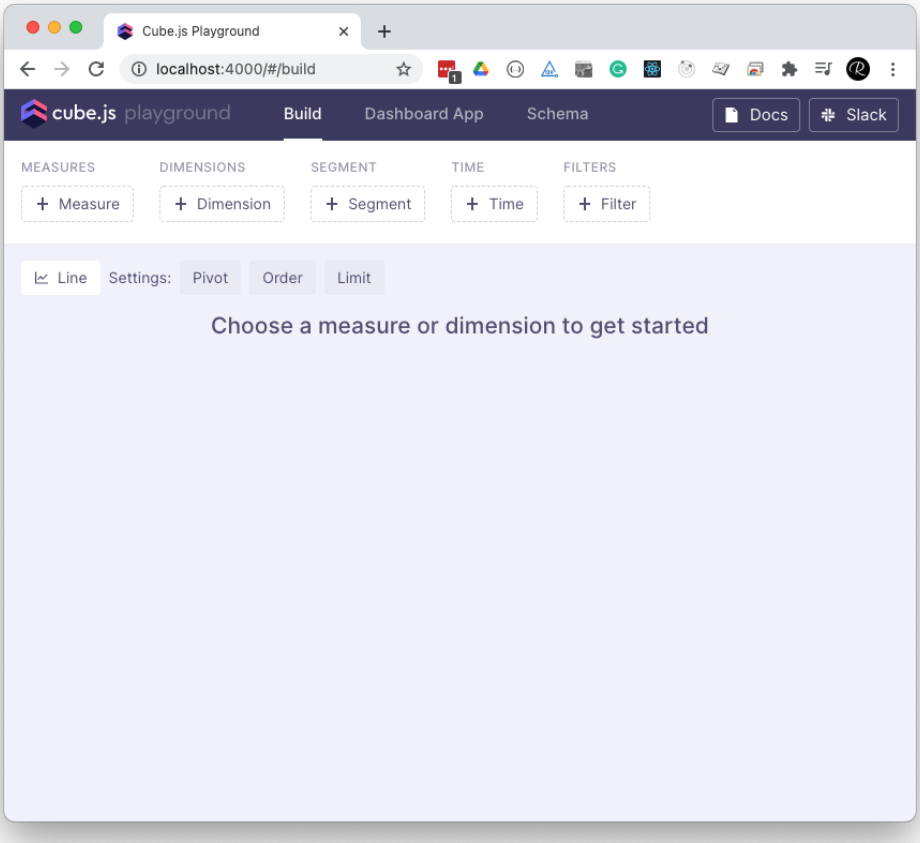
```
CUBEJS_DB_TYPE=bigquery
CUBEJS_API_SECRET=SECRET
```

5. Cube.js uses a data schema to generate SQL code, you will be working with a BigQuery public dataset, create a file called Stories.js in the real-time-dashboard/schema folder with the following code:

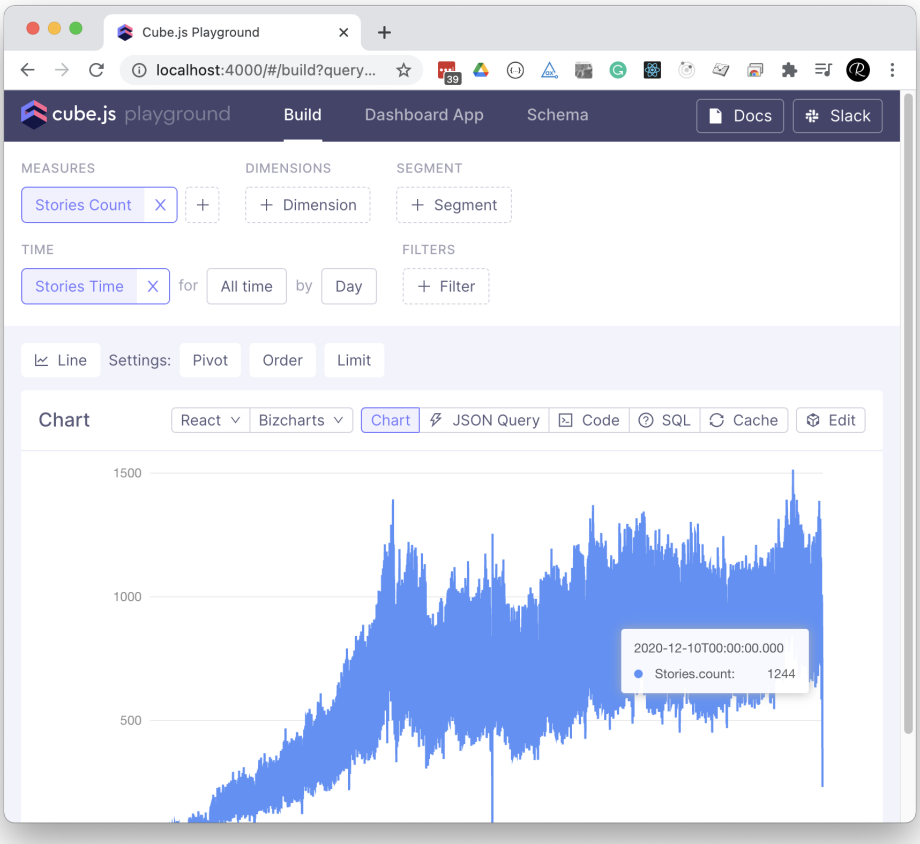
```
cube(`Stories`, {
    sql: `
      SELECT *
      FROM bigquery-public-data.hacker_news.full
      WHERE type = "story" AND STARTS_WITH(UPPER(url), "
    `,

    measures: {
      count: {
        type: `count`,
      },
    },

    dimensions: {
      protocol: {
        sql: `UPPER(REGEXP_EXTRACT(${CUBE}.url, r"^([a-z
        type: `string`,
      },

      time: {
        sql: `timestamp`,
        type: `time`,
      },
    },
  });
```

6. Now run real-time-dashboard locally to test and validate it's working as expected. Run the following command in the real-time-dashboard folder: npm run dev

In your browser, go to http://localhost:4000 which should launch Cube.js playground as seen below.



7. To test it's connecting to BigQuery, click on Measure and choose **Stories Count,** it should look like:



8. This service will become an API running on App Engine. The user dashboard will connect to the Cube.js API to fetch the required data to visualize to the user.
9. To build the dashboard, click on the Dashboard App in the Cube.js playground and choose .....
10. Once it's installed, locate the folder in your IDE, it will be under real-time-dashboard/dashboard-app
11. In your IDE edit the src/pages/DashboardPage.js to include the following:

```
-   const DashboardItems = []
+   const DashboardItems = [
  {
    id: 0,
```

```
    name: "Orders Status by Customers City",
    vizState: {
      query: {
        "measures": [
          "Stories.count"
        ],
        "timeDimensions": [],
        "order": {
          "Stories.count": "desc"
        },
        "dimensions": [
          "Stories.protocol"
        ]
      },
      chartType: "pie",
    }
  },
  {
    id: 1,
    name: "Orders Status by Customers City",
    vizState: {
      query:   {
        "measures": [
          "Stories.count"
        ],
        "timeDimensions": [
          {
            "dimension": "Stories.time",
            "granularity": "year"
          }
        ],
        "order": {},
        "dimensions": []
      },
      chartType: "line",
    }
  },
];
```

12. In your IDE edit the src/components/ChartRenderer.js to include the following:

```
 const ChartRenderer = ({
-   vizState
+   vizState, cubejsApi
-   const renderProps = useCubeQuery(query);
+   const renderProps = useCubeQuery(query, { subscribe:
```

13. Create an app.yaml file in the real-time-dashboard/dashboard-app with the following code:

```
runtime: nodejs14
handlers:
- url: /(.*\..+)$
  static_files: build/\1
  upload: build/(.*\..+)$
- url: /.*
  static_files: build/index.html
  upload: build/index.html
```

14. This configuration let's App Engine serve the optimized React.js build. When making changes you will always need to run npm run build before deploying your new version to App Engine.
15. Run the following commands to deploy the Dashboard to App Engine:

```
npm run build
gcloud app deploy
```

16. After this has been successfully deployed run the following command to view the application in your browser: gcloud app browse
17. Copy the URL of your deployed dashboard-app and edit the real-time-dashboard/dashboard-app/App.js file to replace the const API_URL variable with yours, it should look like this:

```
const API_URL = "https://ruicosta-blog.uc.r.appspot.com"
```

18. Go ahead and redeploy:

```
npm run build
        gcloud app deploy
```

19. You have at this point the Dashboard ready to connect to the Cube.js API which you just updated in the App.js file. Now it's time to deploy the API to App Engine.
20. Create a Dockerfile in the real-time-dashboard folder with the following code:

```
FROM cubejs/cube:latest
COPY . .
```

21. Create an app.yaml file in the real-time-dashboard folder with the following code:

```
runtime: custom
env: flex
service: api
```

22. Since Cube.js uses Websockets and App Engine standard does not support Websockets we need to use a custom runtime so you will use Flexible for the API.
23. Update the content of the cube.js file the following located in the root of the real-time-dashboard folder:

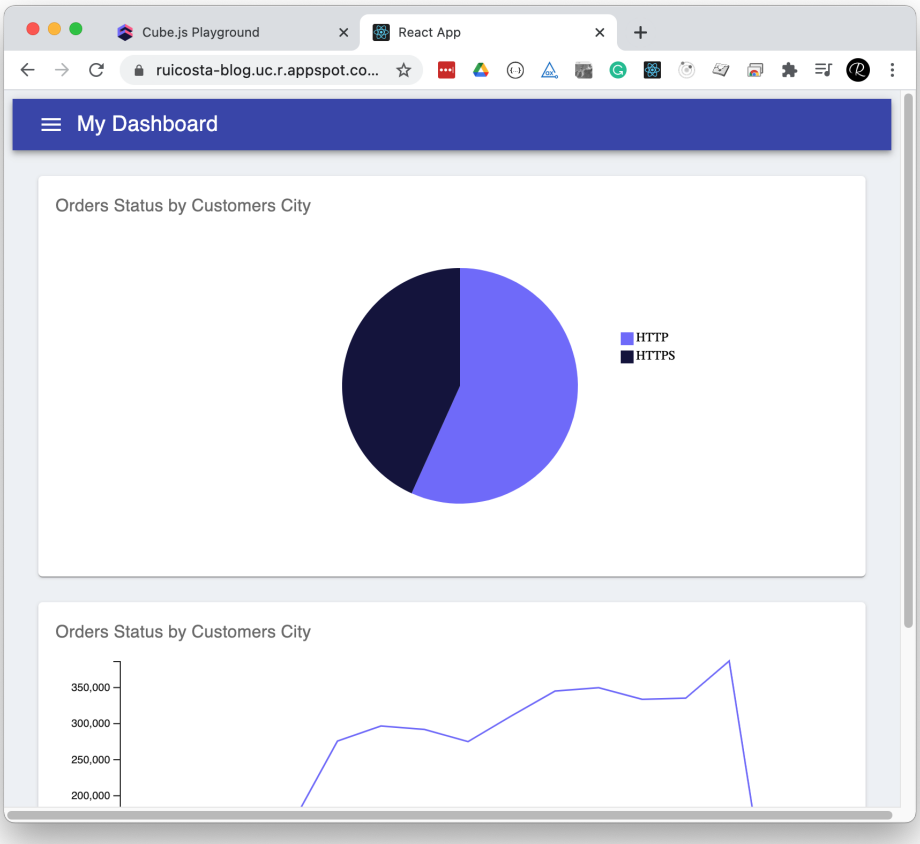
```
module.exports = {
    processSubscriptionsInterval: 1,
    orchestratorOptions: {
      queryCacheOptions: {
        refreshKeyRenewalThreshold: 1,
      }
    },
  };
```

24. Update the content of the .env file the following located in the root of the real-time-dashboard folder to include: CUBEJS_WEB_SOCKETS=true
25. Create a new file in the root of the real-time-dashboard folder called dispatch.yaml

```
- url: "*/cubejs-api*"
  service: api
```

26. The dispatch.yaml allows you to override routing rules, which allows your Dashboard application to access the API via the main URL of the Dashboard as not to cause issues with CORS.
27. You are now ready to deploy the API and have users access data via the dashboard.
28. In the root of the real-time-dashboard run the following command to deploy the API: gcloud app deploy
29. Once this has completed, deploy the dispatch rules by running the following: gcloud app deploy dispatch.yaml

30. If you now access the URL of the Dashboard you should be able to see the following:



31. Don't forget to secure your application by enabling IAP.

Discussion

In this recipe you deployed the Cube.js API, a user Dashboard running on the React.js Framework, toApp Engine and created routes with dispatch rules to build an interactive real-time dashboard for users. There are many moving parts in this recipe, but the key takeaways are:

- App Engine Standard does not support Websockets, so you used App Engine Flexible because the Cube.js API relies on Web Sockets.

- You can enable routes in App Engine, allowing you to redirect traffic to a different service based on the URL path. In this recipe you used it to avoid CORS issues.
- App Engine is very flexible; with custom runtimes, the possibilities of running your application on App Engine are endless.

3.7 Debugging an Instance

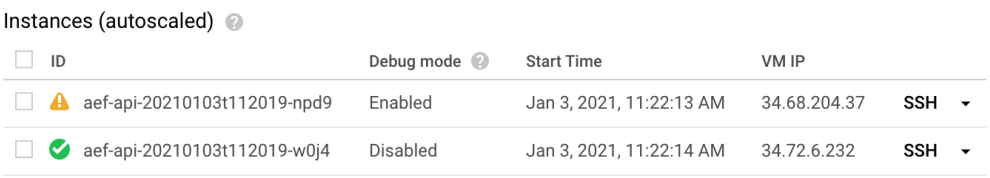
Problem

You notice an issue with your application, and you need a way to access the logs to debug.

Solution

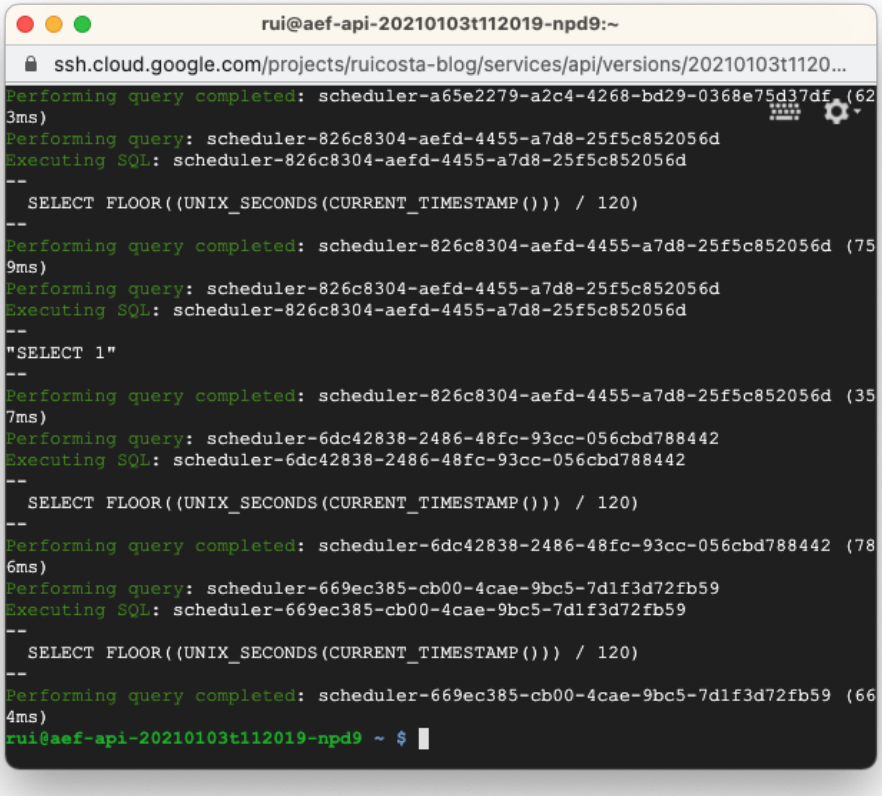
With App Engine Flexible you can enable debug mode and while debug is enabled you can access the VM to view the log files of your custom runtime.

1. To enable debug mode, run the following command: gcloud app --project [PROJECT-ID]
2. It will prompt you with the instances available to enable debugging. Choose one.
3. Go to the Google Cloud Console > App Engine > Instances
4. You should notice the instance you choose that **Debug mode** is now enabled



5. Click on the SSH button to connect to the instance
6. At this point you are connected to the instance host, which has several containers running in it.
7. Run the following command to list the containers running: sudo docker ps
8. The output of the sudo docker ps command lists each container by row; locate the row that contains your project ID and note the NAME of this container.

   To view the logs run the following command: sudo docker logs [CONTAINER-NAME]



9. This allows you to view the logs from your application for debugging purposes.

10. You can also connect to the instance by running the following: sudo docker exec -it [CONTAINER-NAME] /bin/bash
11. When completed don't forget to disable debugging by running the following command: gcloud app --project [PROJECT-ID] instances disable-debug

Discussion

The ability to connect to an instance and its containers allows you debug your application running on App Engine Flexible.

3.8 GitLab CI/CD and App Engine

Problem

You need a method to automate the deployment of your application to App Engine every time a change is made to the source code.

Solution

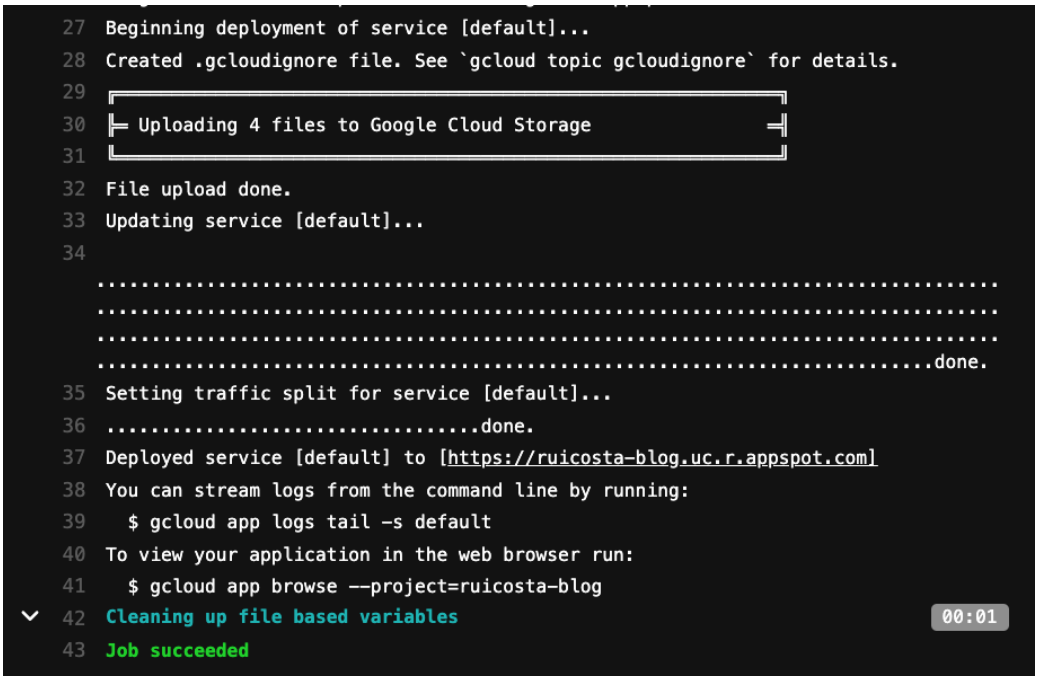
With this recipe you will use GitLab CI/CD which is a tool that allows you to apply all the continuous methods (Continuous Integration, Delivery, and Deployment) to your application.

1. Create a new GitLab project and clone the new repository to your local machine
2. Create a the Hello World application from **recipe 1** of this chapter but do not deploy it to App Engine
3. In the root of the directory create a GitLab CI/CD file named **.gitlab-ci.yml** with the following contents:

```
        image: google/cloud-sdk:slim
deploy:
    stage: deploy
    environment: Production
    only:
        -  master
    script:
        - gcloud auth activate-service-account --key-fil
        - gcloud app deploy app.yaml --quiet --project $
```

4. In the Google Cloud Console go to Identity > Service Accounts
5. Click **Create Service Account**
6. Enter name and description, then click on create
7. Select the role “Editor” and click on continue
8. Select the Service Account you just created and on the options click Create a key in JSON format, download the key to your local workstation
9. In the GitLab console within your project, go to the Setting > CI / CD
10. Expand the variables section
11. Create a new variable
12. Change the type of variable to “File”, the key will be named **GOOGLE_SERVICE_ACCOUNT_FILE** and in the value the content of the file that has been previously downloaded
13. Create another variable named **GOOGLE_PROJECT_ID** and the value will be the ID of the Google Cloud project
14. It’s time to commit your code and deploy your application to App Engine, commit your changes to your GitLab repository.
15. Head over to GitLab console and go to the GitLab CI/CD page, you will notice your pipeline running as an example:

16. If you click on the pipeline you will see the deployment steps, as seen below this completed successfully.



Discussion

The continuous methodologies of software development are based on automating the tests and deployments of your source code to minimize the chance of errors. GitLab CI/CD provides a set of tools, in this recipe you used the Continuous Deployment methodology: now, instead of deploying your application manually, it to be deployed automatically to Google Cloud App Engine.

Chapter 4. Google Cloud Compute Engine

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at ruiscosta@google.com and dhodun@google.com.

Google Compute Engine provides you the ability to run virtual machines on Google's infrastructure. You can run Windows and Linux virtual machines. You have the ability to customize the virtual machine to meet your needs, you can change the memory allocation, how many virtual CPUs are assigned and even automating patching of the operating system.

This chapter contains recipes for creating, and managing your virtual machines including unique methods to automate deployments, deploying containers to virtual machines and using Identity Aware Proxy to tunnel RDP traffic to securely connect to your Windows Virtual Machines.

4.1 Create a Windows Virtual Machine

Problem

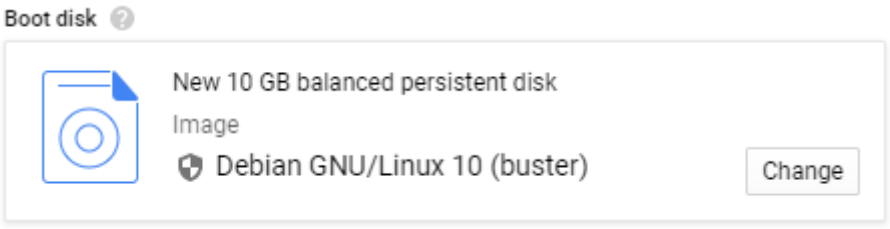
You have an application that needs to be installed on a Windows server. You also need access to the operating system to change configuration options required by the application.

Solution

Using the Google Cloud Console, you will create a Windows Server on Google Cloud Compute Engine. This will provide you with full access to the operating system to allow you to make any configuration changes required by the application.

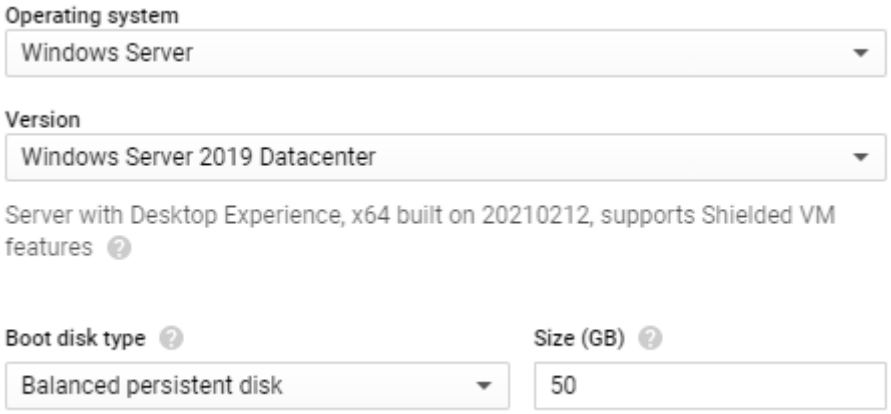
1. Sign in to Google Cloud Console.
2. In the main menu, navigate to Compute and click on Compute Engine

3. Select **VM instances** from the menu and click **Create**
4. Choose a name for your instance
5. Choose a Region and Zone for where this VM will be hosted in
6. Select a Machine configuration or customize based on your requirements
7. Click the Change button in the Boot disk section

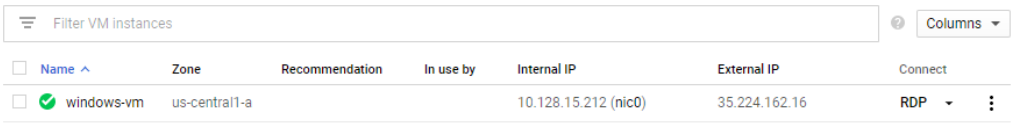


8. Choose **Windows Server** for the operating system

9. Chose the Windows Version required and select a disk size



10. Click **Select**
11. Leave other settings to default and click **Create**
12. It will take a few seconds for your Windows Virtual Machine to start up.



Discussion

In this recipe you used the Google Cloud Console to create a Windows Server. Creating a Windows Server on Compute Engine is a quick and painless way to run your Windows workloads on a managed infrastructure. When creating your virtual machine explore the options available as the ability to customize the memory and virtual CPU allocation for your virtual machine. You are never locked into a pre-configured template.

4.2 Create a Linux Virtual Machine and install NGNIX

Problem

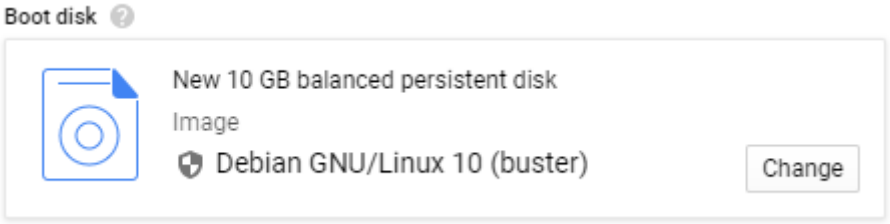
You want to install a NGINX web server in the Cloud and need full access to the operating system hosting NGINX.

Solution

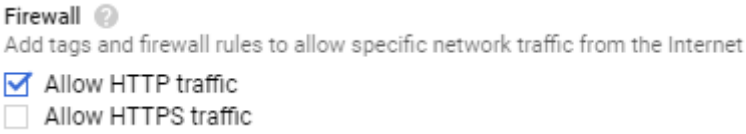
Using the Google Cloud Console, you will create a Linux Virtual Machine on Google Cloud Compute engine. This will provide you with full access to the operating system to allow you to make any configuration changes required by the application. You will also connect to the instance and install a NGINX web server.

1. Sign in to Google Cloud Console.
2. In the main menu, navigate to Compute and click on Compute Engine

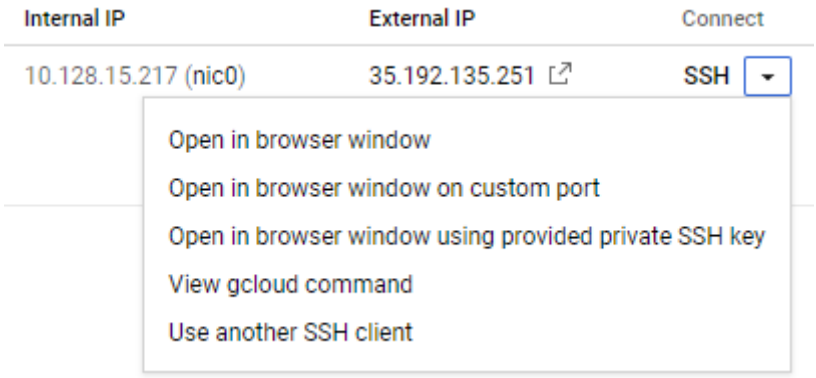
3. Select **VM instances** from the menu and click **Create**
4. Choose a name for your instance
5. Choose a Region and Zone for where this VM will be hosted in
6. Select a Machine configuration or customize based on your requirements
7. Leave the Boot disk set to Debian GNU/Linux 10 (buster)



8. Allow HTTP traffic to the instance



9. Click **Create**
10. Once the instance has been created and public IP address has been set, click on the SSH and select **Open in browser window**



11. In the instance terminal enter the following commands to install NGINX:
12. sudo su -
    apt-get update
    apt-get install -y nginx
    service nginx start

13. In the Cloud Console click on the Public IP Address of your server to open the URL in a new browser tab

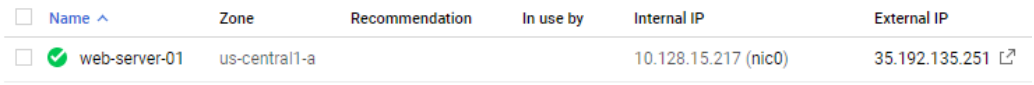
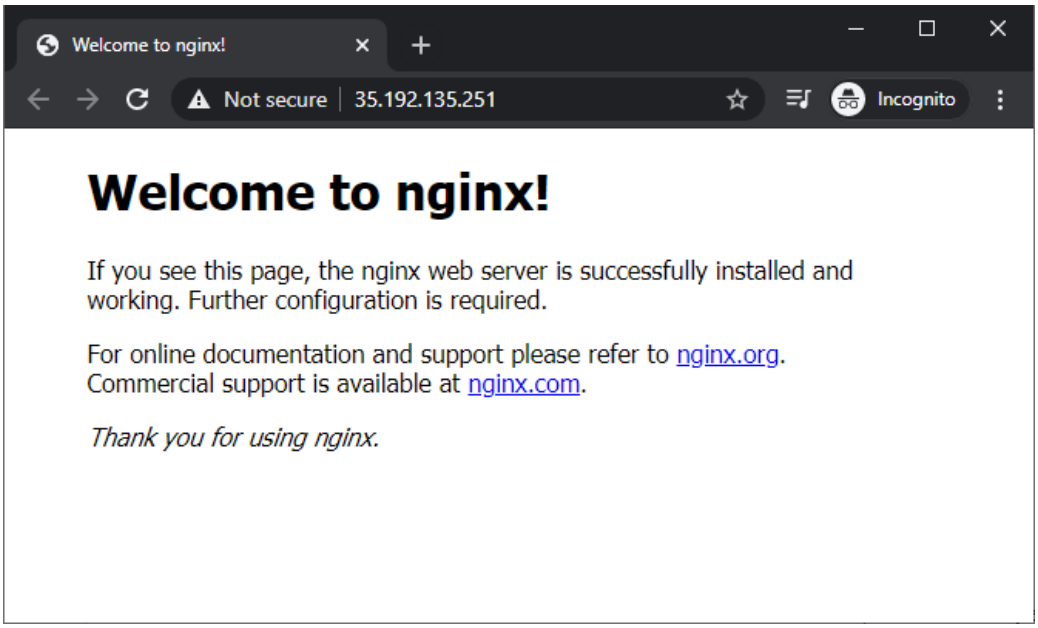


Figure 4-1. Cap tk

14. You should see the following in your browser



Discussion

In this recipe you used the Google Cloud Console to create a Linux Virtual Machine. You also installed NGINX via the Google Cloud SSH browser. The SSH browser is a quick and easy way to get access to your Linux terminal without having to manage local SSH keys which you can if you choose to. You also allowed HTTP traffic to the instance by selecting Allow HTTP Traffic in the firewall section of the creation screen. For this to work, you need to make sure you do not delete the default firewall rule that allows HTTP traffic. If you happen to delete the firewall rule, you can recreate it

or create a new one and associate the firewall tags with the instances you need HTTP traffic to be allowed.

4.3 Connecting to your Windows Virtual Machines with Identity-Aware Proxy TCP Forwarding

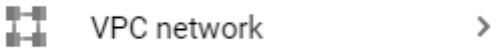
Problem

You want to install a NGINX web server in the Cloud and need full access to the operating system hosting NGINX. You also need the ability to replicate this process multiple times and require a quick method to deploy additional instances.

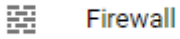
Solution

Using the Google Cloud Console, you will create a Linux Virtual Machine on Google Cloud Compute engine. This will provide you with full access to the operating system to allow you to make any configuration changes required by the application. You will also create a startup script to automate the installation of NGINX allowing you to quickly deploy additional instances.

1. You will need an existing Windows Virtual Machine ready and running on Compute Engine.
2. In the main menu of the Cloud Console, navigate to **Networking** and click on **VPC network**



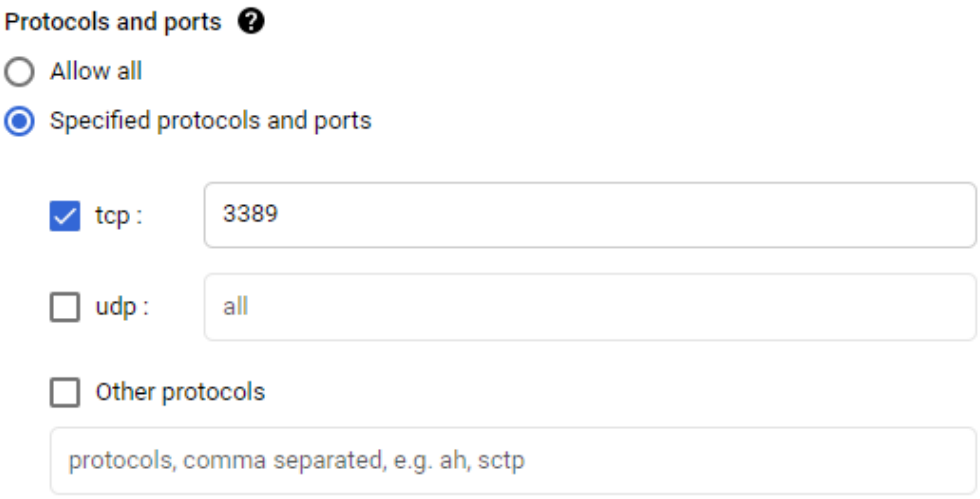
3. Click on **Firewall** in the menu



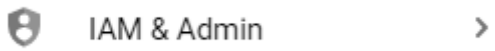
4. Click **Create a firewall rule**
5. Enter a name for the firewall rule as:
6. allow-remote-iap
7. For the **Targets** select All instances in the network

8. **For the Source IP ranges** enter the following CIDR range: 35.235.240.0/20

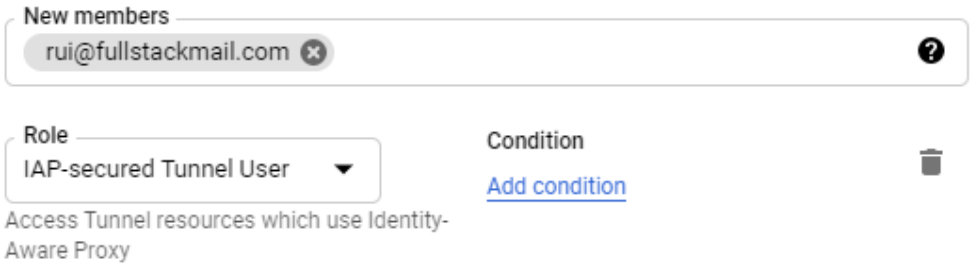
9. Note: The 35.235.240.0/20 CIDR range contains all IP addresses that Google Cloud IAP uses for TCP forwarding.
10. Select TCP and enter 3389 to allow RDP in the **Protocols and ports** section



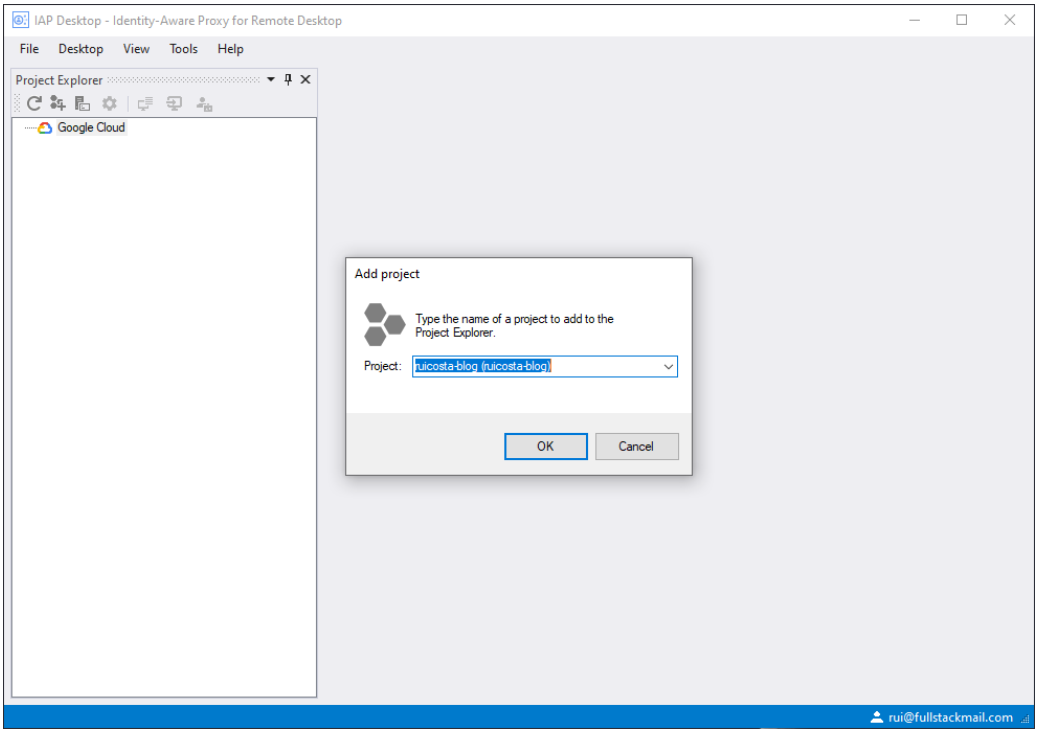
11. Click **Create**
12. Navigate to IAM & Admin in the Cloud Console menu



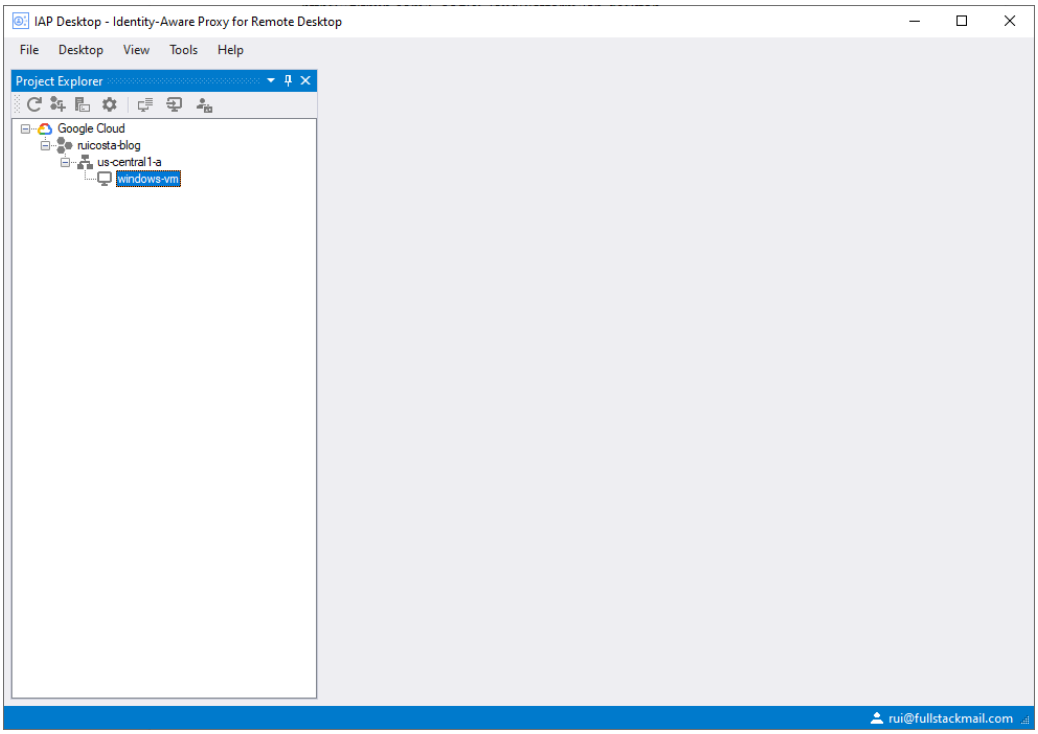
13. Click Add to a new permission to allow the groups or and users to access to the IAP TCP forwarding which will grant them access to connect to the instances in this project.
14. For the members, select the groups and or users for which you want to grant access
15. Select a role and choose **IAP-Secured Tunnel User**.



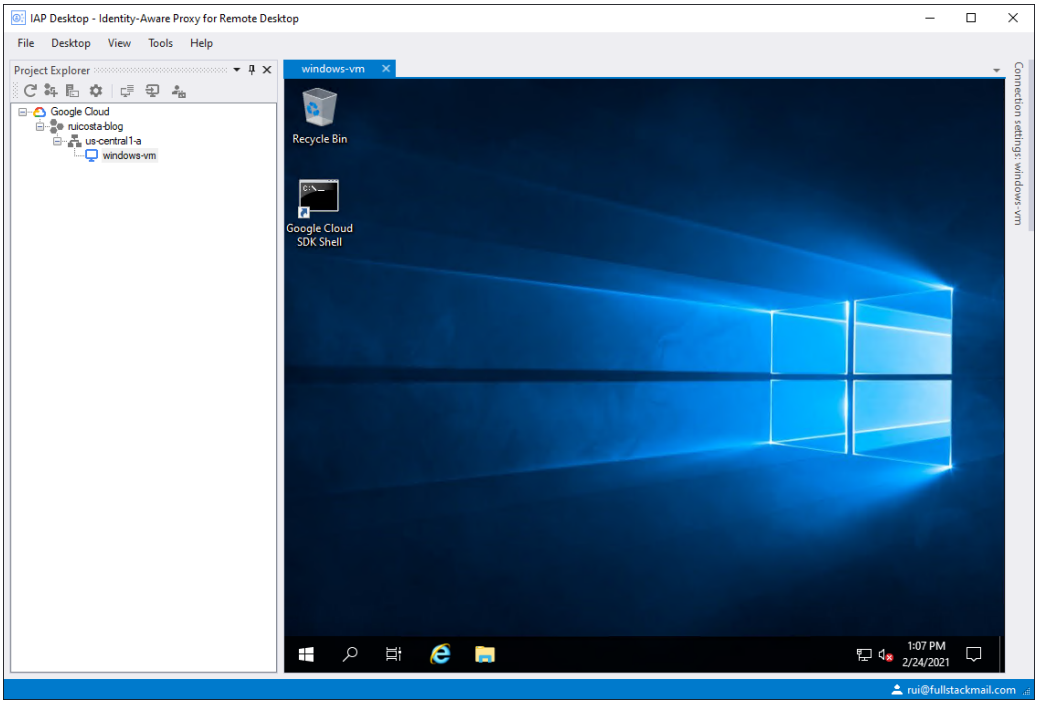
16. Click **Save**
17. Install IAP Desktop, you can find the download at the following location https://github.com/GoogleCloudPlatform/iap-desktop
18. Launch the IAP Desktop application and Sign-in with the account you granted access in step 13
19. Select your Google Cloud Project



20. Once successfully authenticated you should see your Windows Virtual Machines listed



21. Right click on the instance and select connect, after successfully authenticated you should see the Windows Server desktop



Discussion

Identity-Aware Proxy (IAP) allows you to create a centralized authorization layer for applications. In this recipe you used IAP to tunnel RDP traffic to your Windows Virtual Machines. Using the IAP Desktop client you established a secure connection to Google Cloud, then your account was authenticated and authorized. Once successfully authenticated and authorized you then had access to the instances you had been granted to in your Google Cloud Project. You can further secure this process by implementing 2 step verification, see recipe four for more information.

4.4 Virtual Machine OS Login with 2 Step Verification

Problem

You want to secure your virtual machine OS logins with a 2 step verification process.

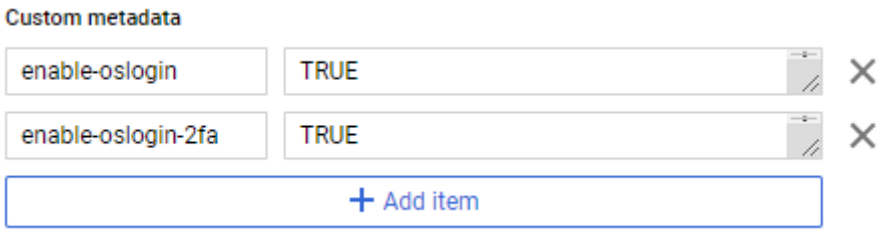
Solution

Compute Engine provides you the ability to secure your Virtual Machine logins with a 2 verification process. In this recipe you will enable 2 step verification on your Google Account as well as a Virtual Machine.

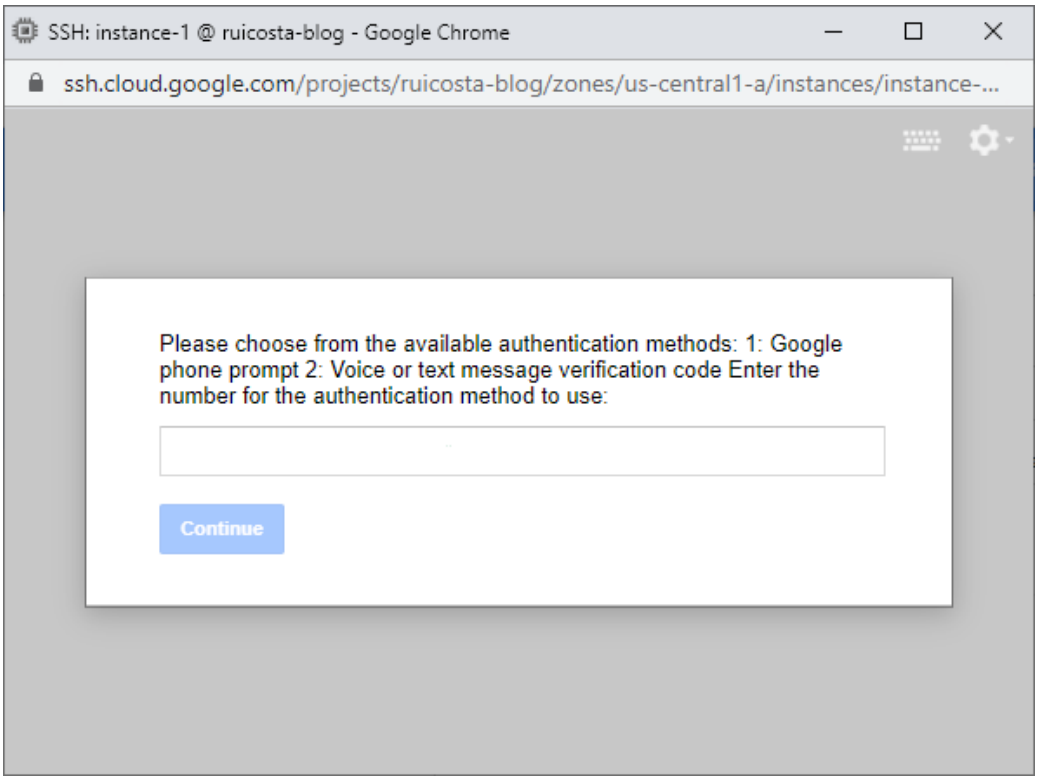
1. You will need at least a Linux virtual machine running on Compute Engine to continue with this recipe.
2. You will also need to enable 2 step verification on your Google Account. To enable this please visit the following link for instructions: https://support.google.com/a/answer/184711?hl=en

3. Sign in to Google Cloud Console.
4. In the main menu, navigate to Compute and click on Compute Engine

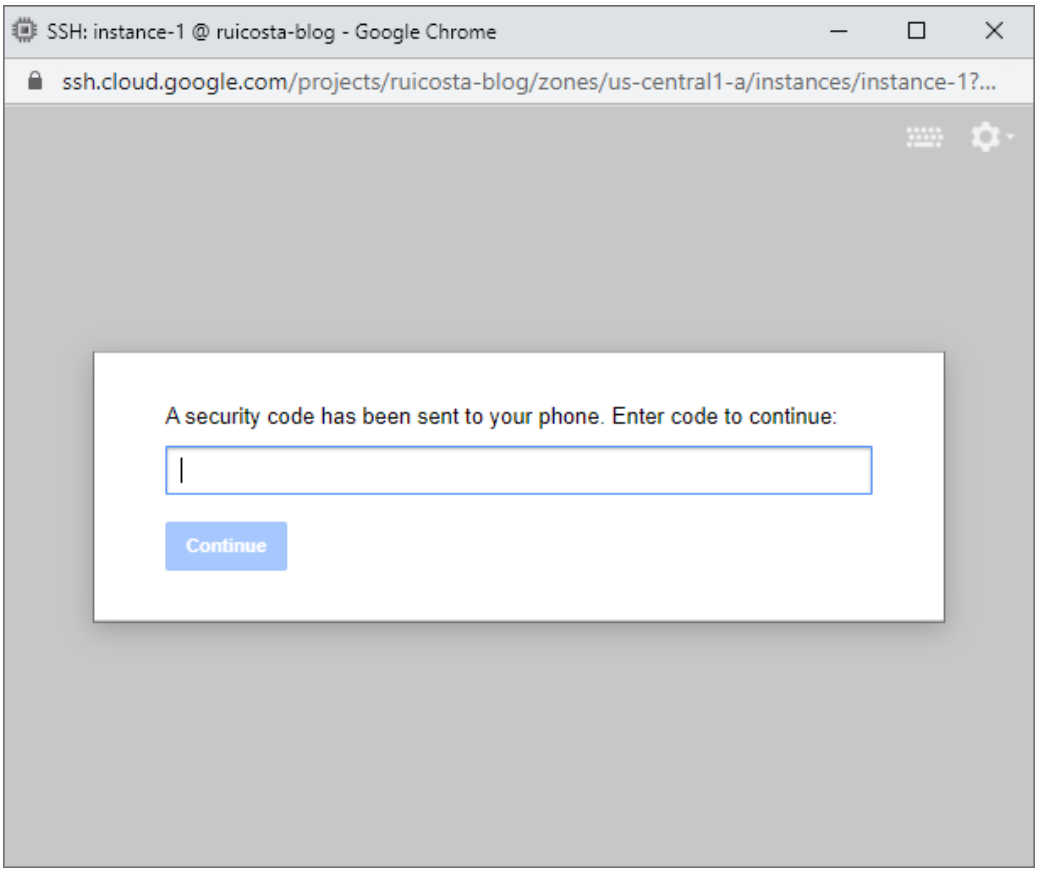
5. Select VM instances from the menu and select the instance you want to enable the 2 step verification for.
6. Click Edit
7. In the Custom metadata section add the following key/value pairs:
8. enable-oslogin: TRUE
9. enable-oslogin-2fa: TRUE



10. Click Save
11. Add the Compute OS Admin Login or Compute OS Login role to the user account who you wish to grant access to the Virtual Machine
12. Using the Cloud SSH Browser connect to the instance.
13. You will now be prompted for a 2 step verification.



14. After making your selection you should receive a code that you will enter into the 2 step verification.



15. Once verification is complete you will have access to Linux Virtual Machines

Discussion

By implementing a 2 step verification you further secured access to your virtual machines running in Google Cloud. Users that need access to instances running in Google Cloud with 2 step verification enabled will be required to enter a code or other methods as physical keys to access the instances. The 2 step verification process is set on your Google Account, if you choose to use a physical you will need to first configure this on your Google Account before you can use it for accessing your virtual machines.

4.5 Running Startup Scripts

Problem

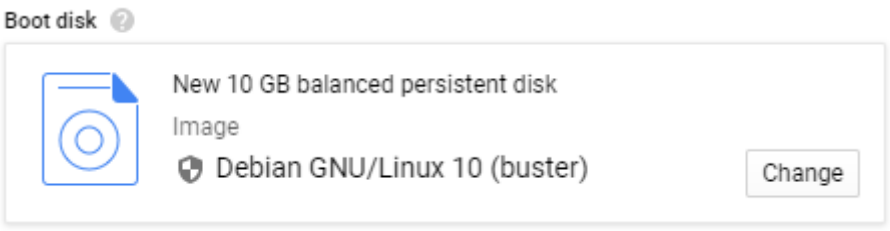
You want to install a NGINX web server in the Cloud and you need the ability to replicate this process multiple times to deploy additional instances.

Solution

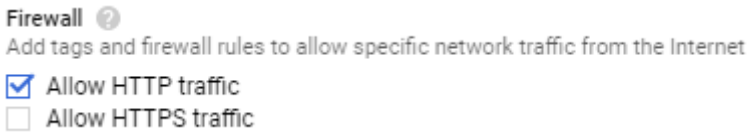
Using the Google Cloud Console, you will create a Linux Virtual Machine on Google Cloud Compute engine.You will also create a startup script to automate the installation of NGINX allowing you to quickly deploy additional instances.

1. Sign in to Google Cloud Console.
2. In the main menu, navigate to Compute and click on Compute Engine

3. Select **VM instances** from the menu and click **Create**
4. Choose a name for your instance
5. Choose a Region and Zone for where this VM will be hosted in
6. Select a Machine configuration or customize based on your requirements
7. Leave the Boot disk set to Debian GNU/Linux 10 (buster)

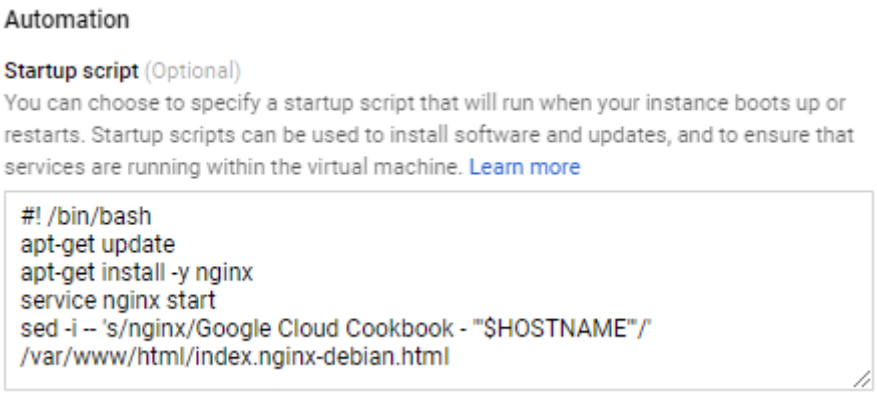


8. Allow HTTP traffic to the instance

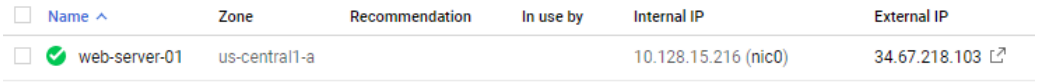


9. In the Startup script text input enter the following commands:

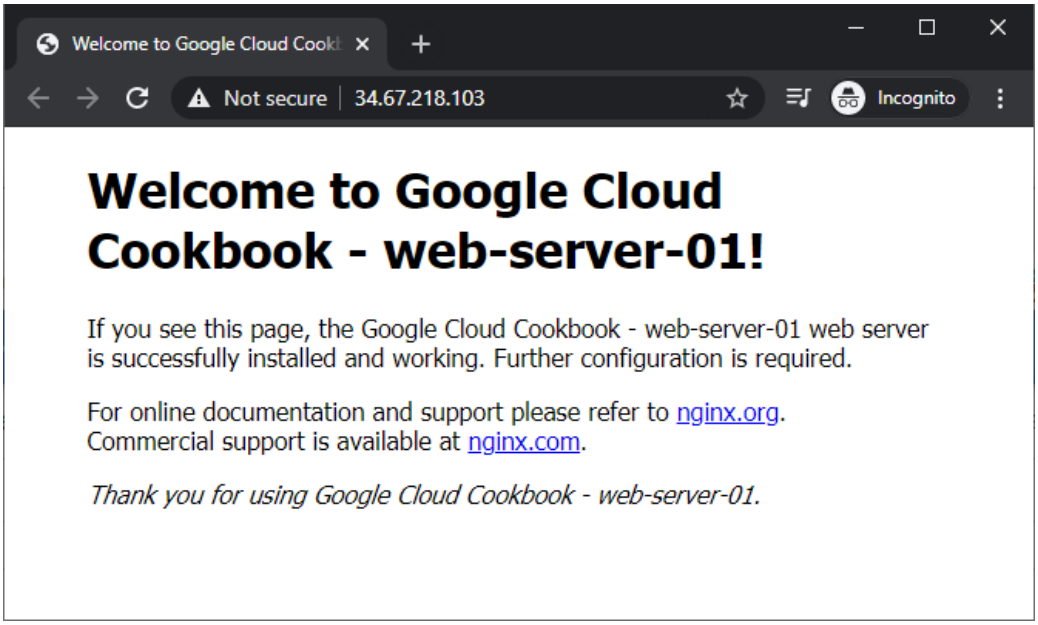
```
#! /bin/bash
apt-get update
apt-get install -y nginx
service nginx start
sed -i -- 's/nginx/Google Cloud Cookbook - "'$HOSTNAME'"/'
/var/www/html/index.nginx-debian.html
```



10. Click **Create**
11. Once the instance has been created and public IP address has been set, open your Internet Browser to the instances public IP Address



12. You should see the following in your browser



Discussion

Compute Engine allows you to create startup and shutdown scripts for your virtual machines. With this recipe you had the opportunity to use startup scripts to automate the installation of software. By using startup scripts it automated the deployment of say large fleet of web servers. In recipe six you will learn how to use the startup scripts to deploy a cluster of web servers which makes the process of installing software more efficient then connecting to each instance to install the software.

4.6 Create a Cluster of NGINX web server with an Instance Group

Problem

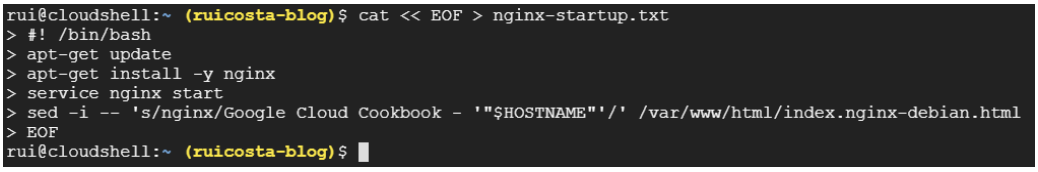
You need to host a highly available web server using NGINX. In the event one server fails you need for your web application to tolerate a host failure.

Solution

Using Instance Templates, you will create a cluster of Linux Virtual Machines on Google Cloud Compute engine.You will also create a Network Load Balancers to distribute traffic across the cluster of virtual machines.

1. Sign in to Google Cloud Console and launch Cloud Shell.
2. In your Cloud Shell create a startup script named **nginx-startup.txt** with the following commands:

```
#! /bin/bash
apt-get update
apt-get install -y nginx
service nginx start
sed -i -- 's/nginx/Google Cloud Cookbook - '"$HOSTNAME"'
```



3. Create a new Instance Template and define the startup script to file created in step 2

```
gcloud compute instance-templates create nginx-template
```

4. Create a target pool. A target pool will allow you have a single access point for load balancing. Run the following command:

```
gcloud compute target-pools create nginx-pool
```

5. Run the following command to create an Instance Group

```
gcloud compute instance-groups managed create nginx-grou
         --base-instance-name nginx \
         --size 2 \
         --template nginx-template \
            --target-pool nginx-pool
```

6. You should see the following output:

```
NAME         LOCATION    SCOPE  BASE_INSTANCE_NAME  SIZE  TARGET_SIZE  INSTANCE_TEMPLATE  AUTOSCALED
nginx-group  us-east1-c  zone   nginx               0     2            nginx-template     no
rui@cloudshell:~ (ruicosta-blog)$
```

7. Open the Cloud Console and navigate to VM instances, you should see two new instances created from the command in step 5.

8. Create a regional network load balancer for the new instance group created, run the following command:

```
gcloud compute forwarding-rules create nginx-lb \
         --ports 80 \
     --target-pool nginx-pool
```

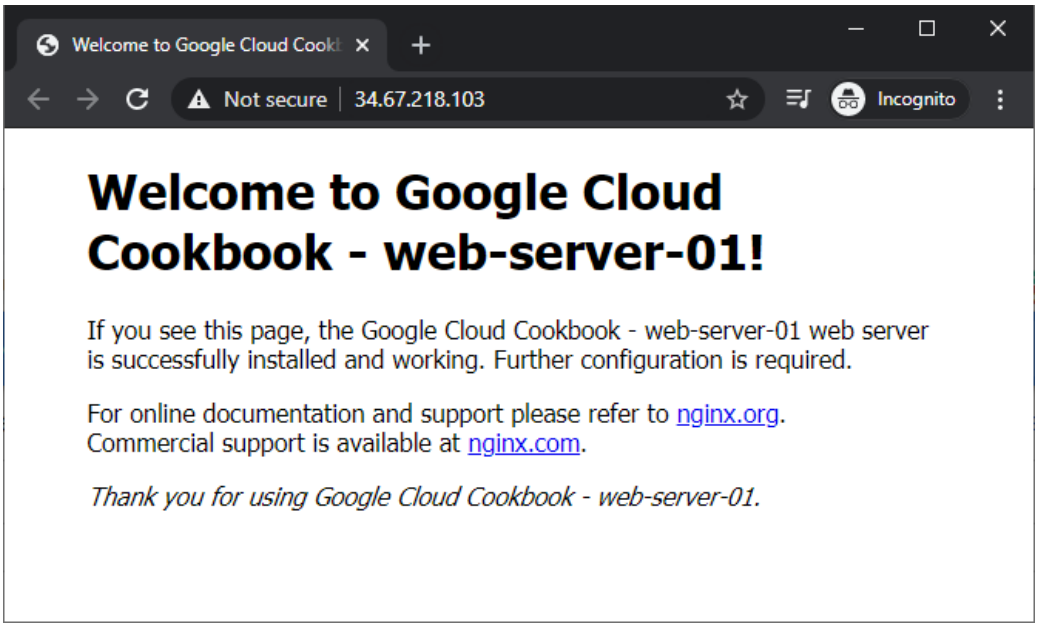
9. When prompted for a region choose the region your instance group is located in, for this example you would choose **us-east1**
10. Run the following command to allow HTTP access to your instance group:

```
gcloud compute firewall-rules create allow-80 --allow tc
```

11. To get the IP address of the regional load balancer run the following command:

```
gcloud compute forwarding-rules list
```

12. Visit the associated IP address in your browser. You should see a similar screen



Discussion

In this recipe you were introduced to Instance Templates plus Instance Groups. Instance templates are a great method to deploy instances that require identical settings. An example is creating web servers for a cluster, you want all the instances to have the same configuration as memory, virtual CPUs and the associated required software. You then used the instance template to create an instance group. An instance group is a grouping of virtual instances managed by a single entity. As a plus you deployed a load balancer to distribute traffic to the virtual instances in the instance group. In the event one instance failed your application would still be running as you had additional instances in the instance group.

4.7 Deploy Containers to Managed Instance Groups

Problem

You have a requirement to start running your applications as containers. You want to get started with Compute Engine and you want to run your NGINX as a container. You also want similar benefits that Kubernetes provides as autoscaling, autohealing, and rolling updates.

Solution

You will create a new Docker container. You will then deploy the Docker container to an Instance Group that will provide with the autoscaling, autohealing, and rolling updates requirements for your application.

1. Sign in to Google Cloud Console and launch Cloud Shell.

2. In this step you will create an new instance template and associate it with a publicly accessible container image. Run the following command in your Cloud Shell

```
gcloud compute instance-templates create-with-container
                --container-image gcr.io/cloud-marketpl
                --tags http-server
```

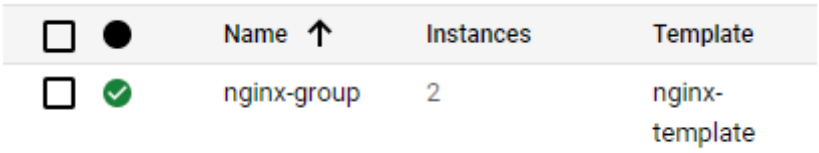
3. Create a target pool. A target pool will allow you have a single access point for load balancing. Run the following command:

```
gcloud compute target-pools create nginx-pool
```

4. Run the following command to create an Instance Group based on the newly created template:

```
gcloud compute instance-groups managed create nginx-grou
    --base-instance-name nginx-vm \
    --size 2 \
    --template nginx-template \
    --target-pool nginx-pool
```

5. In the Cloud Console, navigate to your instance groups. You should see your instance group listed.



6. Run the following command to create a regional load balancer for the newly created instance group:

```
gcloud compute forwarding-rules create nginx-lb \
         --ports 80 \
     --target-pool nginx-pool
```

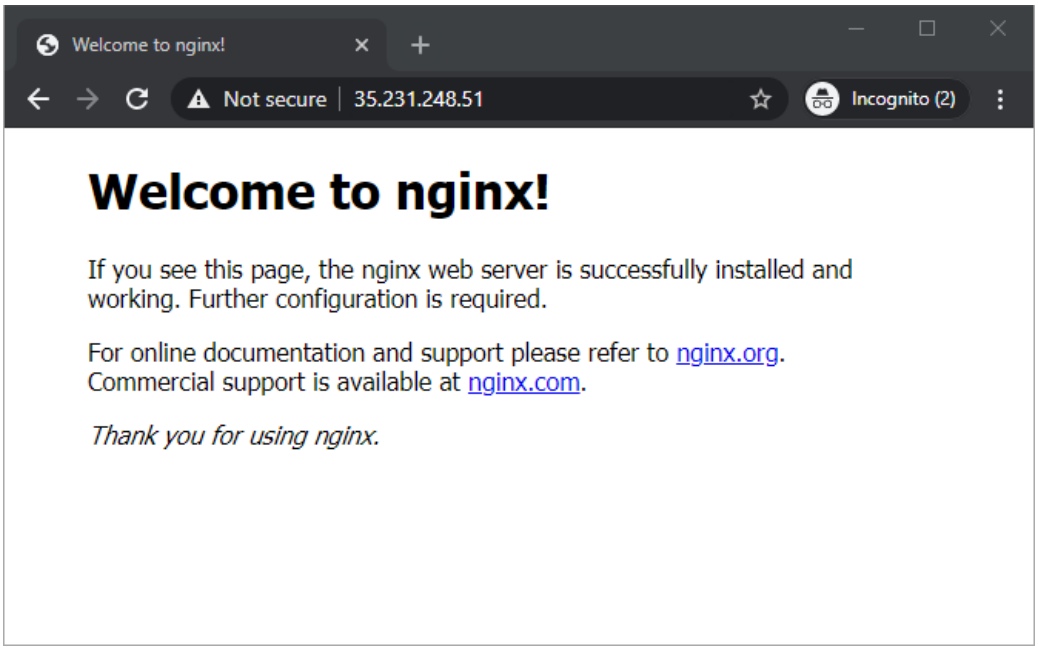
7. When prompted for a region choose the region your instance group is located in, for this example you would choose **us-east1**
8. To get the IP address of the regional load balancer run the following command:

```
gcloud compute forwarding-rules list
```

9. The output should look like this

```
rui@cloudshell:~ (ruicosta-blog)$ gcloud compute forwarding-rules list
NAME      REGION    IP_ADDRESS     IP_PROTOCOL  TARGET
nginx-lb  us-east1  35.231.248.51  TCP          us-east1/targetPools/nginx-pool
```

10. Visit the associated IP address in your browser
11. You should see a similar screen



Discussion

In this recipe you learned how to deploy a container to Compute Engine. This is a great way to get started with containers. It is recommended to run containers on Kubernetes Engine as the orchestration engine for your microservices. After deploying the containers you exposed them with the --tags http-server flag plus you created a load balancer to distribute the traffic across the containers running on multiple virtual machines.

4.8 Transferring Files to your Virtual Machine

Problem

You have a Linux virtual machine running on Compute Engine and you need to transfer files to the instance.

Solution

In this recipe you will use two methods of transferring files to your Linux virtual machine using the gcloud command line tool and SSH in your browser.

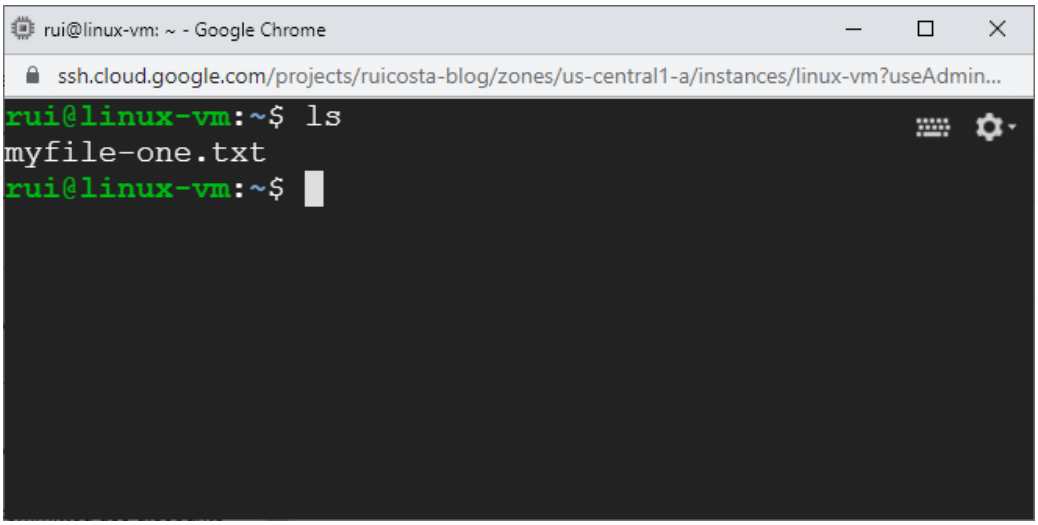
1. You will need a Linux virtual machine running on Compute Engine to continue with this recipe.
2. Sign in to Google Cloud Console and launch Cloud Shell.
3. In your cloud shell run the following command:

```
touch myfile-one.txt
```

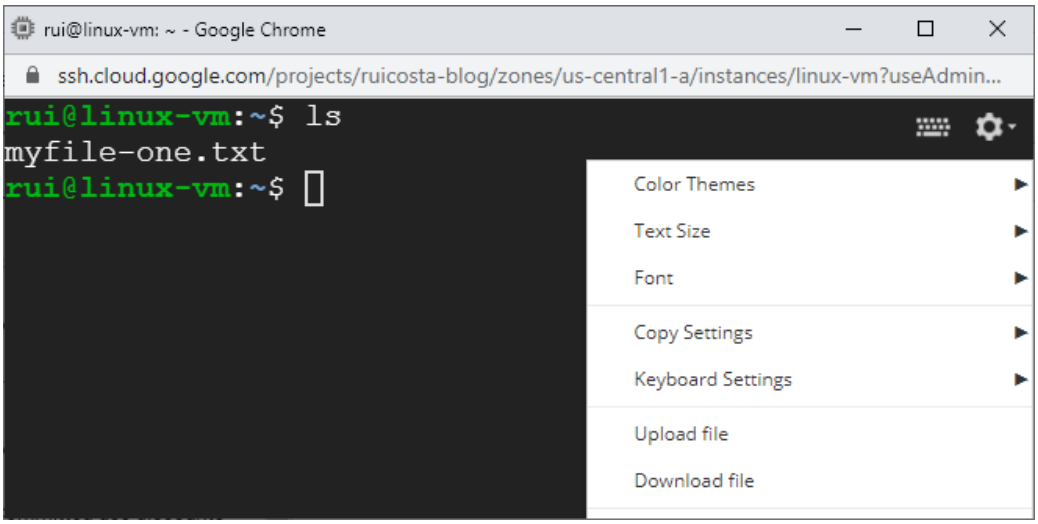
4. To copy the newly created file, run the following command and replace your-instance with your instance name:

```
gcloud compute scp myfile-one.txt instance-name:~
```

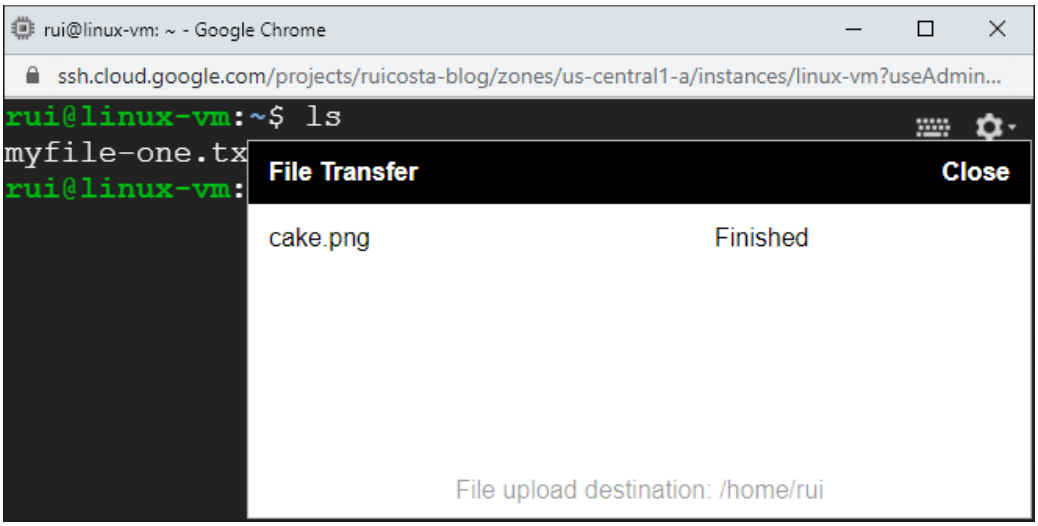
5. Connect to the Linux virtual machine with the Cloud Console SSH Browser
6. In the Linux terminal, validate the file copied



7. To upload a file with the SSH Browser, click on the settings icon and select Upload File



8. Upload a sample file from your local workstation



Discussion

Google Cloud provides you multiple ways to transfer files to your virtual machines. In this recipe you learned two methods to transfer files, one

via the gcloud command that can be run on your local machine and the other via a web browser.

4.9 Using VM Manager for Patch Management

Problem

You host your virtual machines on Compute Engine and you need a method patching all the operating systems at once.

Solution

Using OS patch management with VM Manager you will create a patch job to patch your fleet of Linux Virtual Machines.

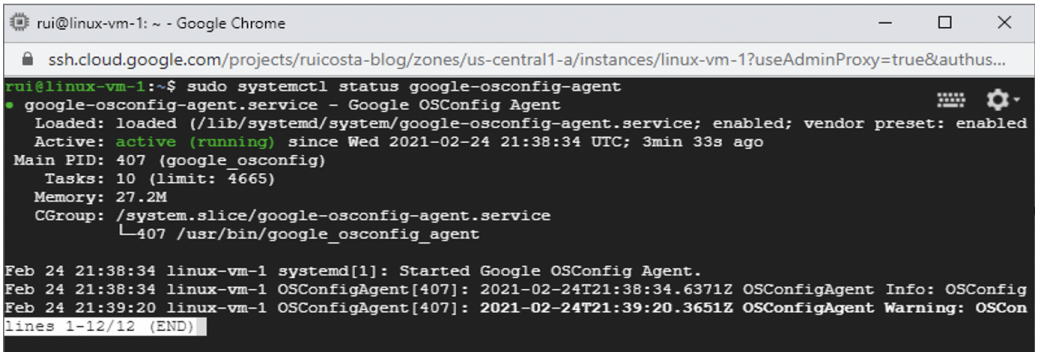
1. You will need at least one Linux virtual machine running on Compute Engine to continue with this recipe.
2. Sign in to Google Cloud Console and launch Cloud Shell.
3. To have Compute Engine manage your operating systems you will need to install configure and install VM manager, run the following commands to enable Operating system management for all your Virtual Machines:

```
gcloud compute project-info add-metadata \
  --project PROJECT_ID \
  --metadata=enable-osconfig=TRUE

gcloud compute project-info add-metadata \
  --project PROJECT_ID \
--metadata=enable-guest-attributes=TRUE,enable-osconfig=
```

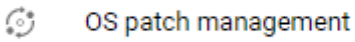
4. Replace the PROJECT_ID with your Google Cloud Project ID
5. Using the Cloud Shell SSH browser connect to one of your instances to validate the OS Config Agent is installed, run the following command:

```
sudo systemctl status google-osconfig-agent
```

6. If the agent is running you should see a similar output from the command:

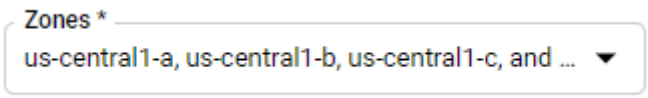


7. In the Cloud Console navigate to Compute Engine > VM Manager > OS patch management



8. Click **Enable VM Manager**
9. Click **New Patch Deployment**

10. Select the Target Zones for your Virtual Machines



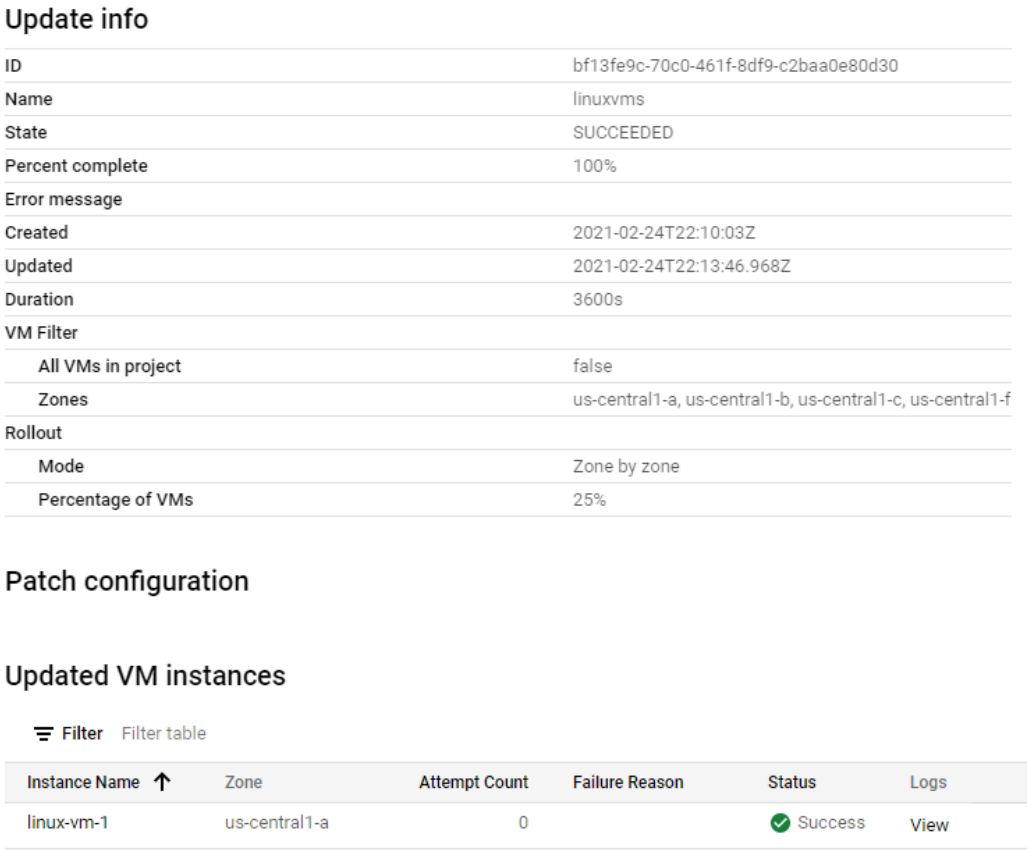
11. Click **Next**
12. Enter a **Deployment Name**

13. Click **Next**
14. Choose default options for Scheduling, Rollout Options and Advanced Options
15. Click **Deploy**
16. Your patch job will start automatically and provide you a status similar to the below

Update info

ID	bf13fe9c-70c0-461f-8df9-c2baa0e80d30
Name	linuxvms
State	PATCHING
Percent complete	50%
Error message	
Created	2021-02-24T22:10:03Z
Updated	2021-02-24T22:10:04.051Z
Duration	3600s
VM Filter	
All VMs in project	false
Zones	us-central1-a, us-central1-b, us-central1-c, us-central1-f
Rollout	
Mode	Zone by zone
Percentage of VMs	25%

17. Once the patch job is completed you should should see a status window similar to the below



Update info

ID	bf13fe9c-70c0-461f-8df9-c2baa0e80d30
Name	linuxvms
State	SUCCEEDED
Percent complete	100%
Error message	
Created	2021-02-24T22:10:03Z
Updated	2021-02-24T22:13:46.968Z
Duration	3600s
VM Filter	
All VMs in project	false
Zones	us-central1-a, us-central1-b, us-central1-c, us-central1-f
Rollout	
Mode	Zone by zone
Percentage of VMs	25%

Patch configuration

Updated VM instances

Filter Filter table

Instance Name	Zone	Attempt Count	Failure Reason	Status	Logs
linux-vm-1	us-central1-a	0		Success	View

Discussion

When managing one virtual machine it's fairly easy to perform patch updates. However, when running large fleets of virtual machines it becomes difficult to patch all your instances without some complicated scripting or 3rd party tool. With VM Manager and OS Patch management Google Cloud provides you the tools to distribute patch updates with multiple options as rolling updates, and restrict updates to certain zones.

4.10 Backuping your Virtual Machine

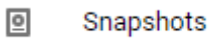
Problem

You have a Linux Virtual Machine running on Compute Engine. This is a critical application to your business and you want to perform backups of the persistent disks assigned to the Virtual Machine as a method to recover the instance if something goes wrong.

Solution

Using Compute Engine persistent disk snapshots you will create a snapshot of the virtual machine's persistent disk to have a recovery point of the disk in the event of mishap.

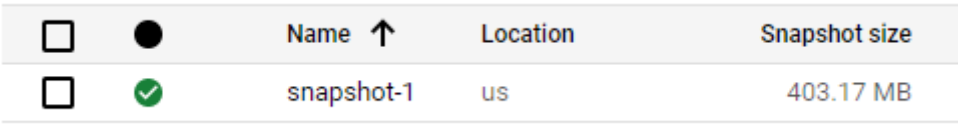
1. You will need a Linux virtual machine running on Compute Engine to continue with this recipe.
2. Sign in to Google Cloud Console.
3. Navigate to Compute > Compute Engine > Snapshots



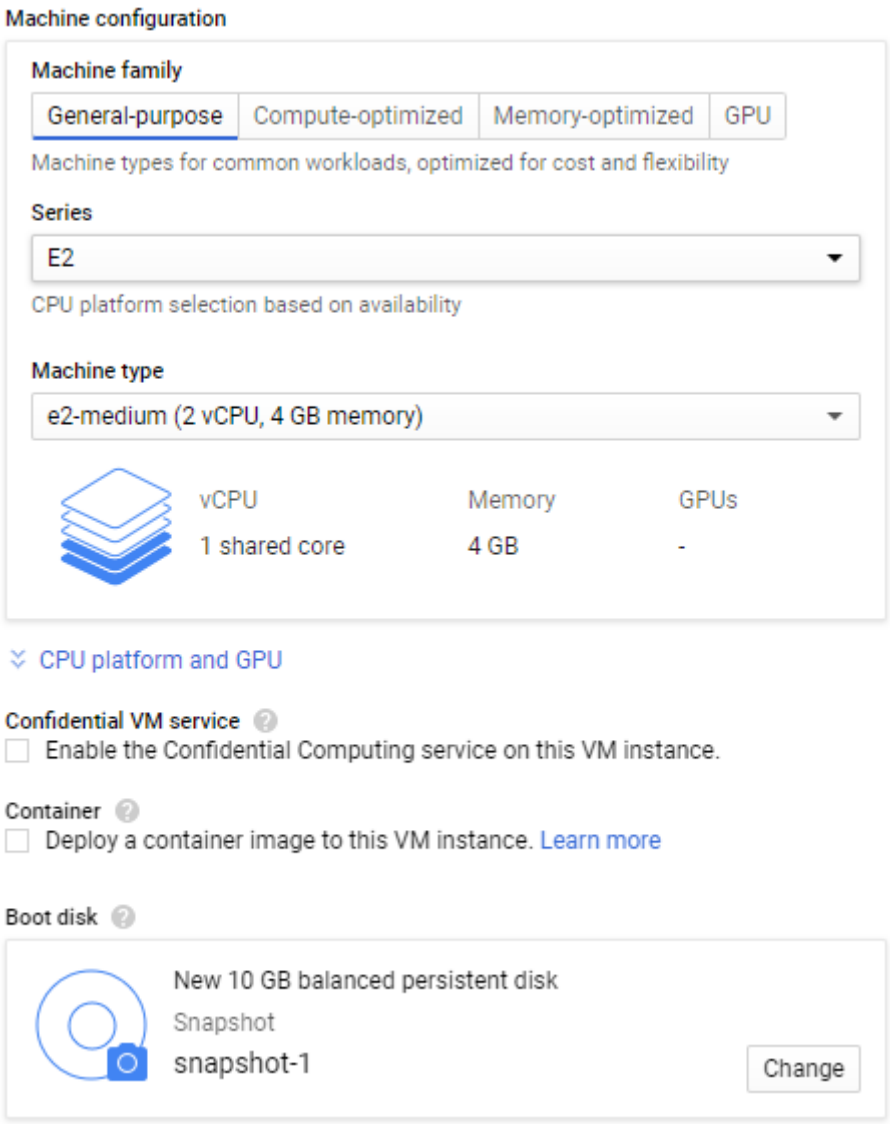
4. Click **Create Snapshot**

5. Enter a name for the snapshot and select the source disk of you virtual machine

6. Click **Create**
7. Once completed you should see your snapshot listed similar to the screenshot below



8. You can now create a new virtual machine based on this snapshot. Below is virtual machine creation window with the boot disk selected as the snapshot created.



Discussion

Using snapshots is a great method to protect your data in the event something goes wrong. It creates a point in time copy of your persistent disk. Besides being a recovery point, it also allows you to create a virtual machine based off the snapshot allowing you to perform file level recovery and testing changes in your application or operating system. You can create a virtual machine with the disk that was snapshotted to

maybe test how an operating system change would affect the state of the virtual machine.

Chapter 5. Google Cloud Kubernetes Engine

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at ruiscosta@google.com and dhodun@google.com.

Google Cloud Kubernetes Engine is a fully managed and secured platform that provides you the ability to run your constrained workloads. This chapter contains recipes for creating, and managing your containers including unique methods to automate deployments, plus deploying real-world applications as in MongoDB and Java applications.

All code for this recipe is located at: https://github.com/ruiscosta/google-cloud-cookbook/06-kubernetes

5.1 Create a Zonal Cluster

Problem

You want to run an application on Kubernetes, but want to run that application only within a single zone, within a region. You want to be able

to create and upgrade your kubernetes cluster quickly, and you are less concerned about things like availability and placing clusters closer to your disparate users.

Solution

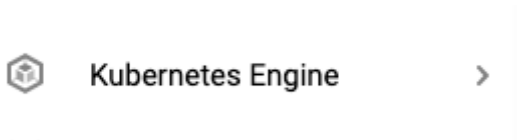
Run your application on a zonal Kubernetes cluster. With a single control plane managing your Kubernetes cluster, it's very easy to get started quickly.

Prerequisites

Ensure the following API is enabled:

- Kubernetes Engine API

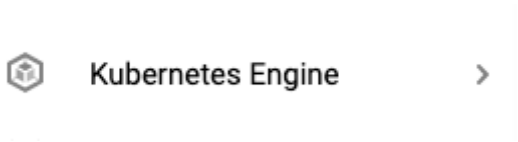
1. Sign in to Google Cloud Console.
2. In the main menu, navigate to Compute and click on **Kubernetes Engine**



3. Click the **Create** button at the top of the screen
4. Click **Configure** next to the **Standard** option.
5. In the **Cluster Basics** section:

    1. Choose a Name for your cluster
    2. In Location Type, select Zonal
    3. In Zone, select any zone of your choice
    4. Leave the remaining settings to default

6. In the left navigation pane there are a lot of other options that could be set, however we will leave them to default for the purposes of this recipe.
7. Click **Create** at the bottom of the screen
8. You will be navigated back to the **Clusters** screen where you will see your cluster spinning up. This process can take more than a minute to complete.
9. Once complete, you will see a green checkmark icon next to the name of your cluster. Your cluster is now ready for the deployment of applications.



Discussion

In this recipe, you used the Google Cloud Console to create a zonal Kubernetes cluster with GKE. Creating a zonal GKE cluster is a quick and easy way to get going with Kubernetes versus trying to launch a self managed Kubernetes cluster. The Kubernetes Control Plane is managed by GCP, so you don't need to worry about the operational overhead that comes with managing it. Beyond this recipe, you should look through the configuration options you have with GKE, which affords you tons of flexibility and additional configuration around **Nodepools**, **Automation**, **Networking**, **Security**, **Metadata**, and more.

5.2 Create a Regional Cluster

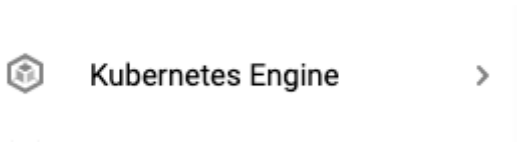
Problem

You want to run an application on Kubernetes, but want to run that application across two or more zones within a region. You value the availability of your application over the flexibility that may come with a zonal kubernetes cluster.

Solution

Run your application on a regional Kubernetes cluster. Regional clusters allow for higher availability, fault tolerance, no-downtime upgrades, making your application more resilient and spread across multiple zones within a single region.

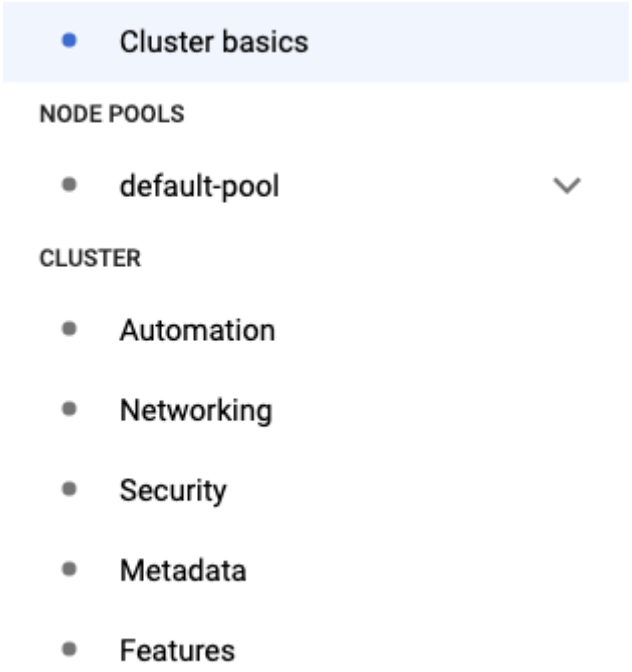
1. Sign in to Google Cloud Console.
2. In the main menu, navigate to Compute and click on **Kubernetes Engine**



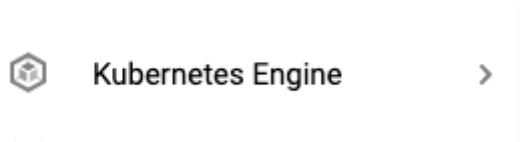
3. Click the **Create** button at the top of the screen
4. Click **Configure** next to the **Standard** option.
5. In the **Cluster Basics** section:

    1. Choose a **Name** for your cluster
    2. In **Location Type**, select **Regional**
    3. In **Region,** select any region of your choice
    4. Leave the remaining settings to default

6. In the left navigation pane there are a lot of other options that could be set, however we will leave them to default for the purposes of this recipe.



7. Click **Create** at the bottom of the screen
8. You will be navigated back to the **Clusters** screen where you will see your cluster spinning up. This process can take more than a minute to complete.



9. Once complete, you will see a green checkmark icon next to the name of your cluster.

Discussion

In this recipe, you used the Google Cloud Console to create a regional Kubernetes cluster with GKE. Creating a regional GKE cluster is a quick and easy way to get going with Kubernetes versus trying to self manage a

kubernetes cluster of your own. With a regional cluster, you have nodes deployed across the zones within that region, so expect that the **Number of Nodes**, **Total vCPUs**, and **Total Memory** are larger than your zonal GKE deployment with the same configuration. The Control Plane (managed by GCP) is also spread out across the zones within the region, so you don't need to worry about configuring it beyond deploying the cluster itself. Beyond this recipe, you should look through the configuration options you have with GKE, which afford you tons of flexibility and configurability around **Nodepools**, **Automation**, **Networking**, **Security**, **Metadata**, and more.

5.3 Deploy a MongoDB Database with StatefulSets

Problem

You want to run MongoDB on Google Cloud Kubernetes Engine which requires access to a persistent disk.

Solution

Since MongoDB requires persistent disk you deploy MongoDB as a stateful application. This will require you to use the StateFulSet controller to deploy MongoDB for a persistent identity for when pods need to be rescheduled or restarted. For MongoDB you will need to maintain access to the same volume if the pods are rescheduled or restarted.

1. Sign in to Google Cloud Console.
2. Launch the Google Cloud Shell.
3. Run the following command to instantiate a hello-world cluster:

```
                    gcloud container clusters create hello
--region us-central1
```

4. It will take a few minutes for the cluster to start up.
5. Use the following as the working directory which is the cloned repository from step 5 for the remaining steps below.

```
                  cd ./mongo-k8s-sidecar/example/StatefulS
```

6. In the cloned repository root fol run the following command to create your StorageClass which tells Kubernetes what disk type you want for your MongoDB database:

```
                    cat >> mongodb_storage_class_ssd.yaml
kind: StorageClass
apiVersion: storage.k8s.io/v1beta1
metadata:
  name: fast
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-ssd
        EOL
```

7. Run the following command to apply the StorageClass to Kubernetes:

```
                  kubectl apply -f mongodb_storage_class_s
```

8. In Kubernetes terms, a service describes policies or rules for accessing specific pods. In brief, a headless service is one that doesn't prescribe load balancing. When combined with StatefulSets, this will give you individual DNSs to access your pods, and in turn a way to connect to all of your MongoDB nodes individually.
9. Run the following command to create the mongo-statefulset.yaml file. You can also access the code for this recipe at: https://github.com/ruiscosta/google-cloud-cookbook/06-kubernetes

```
                    cat >> mongo-statefulset.yaml <<EOL
apiVersion: v1
kind: Service
metadata:
  name: mongo
  labels:
    name: mongo
spec:
  ports:
  - port: 27017
    targetPort: 27017
  clusterIP: None
  selector:
    role: mongo
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mongo
spec:
  serviceName: "mongo"
  replicas: 3
  selector:
    matchLabels:
      role: mongo
      environment: test
```

```
  template:
    metadata:
      labels:
        role: mongo
        environment: test
    spec:
      terminationGracePeriodSeconds: 10
      containers:
        - name: mongo
          image: mongo
          command:
            - mongod
            - "--replSet"
            - rs0
          ports:
            - containerPort: 27017
          volumeMounts:
            - name: mongo-persistent-storage
              mountPath: /data/db
        - name: mongo-sidecar
          image: cvallance/mongo-k8s-sidecar
          env:
            - name: MONGO_SIDECAR_POD_LABELS
              value: "role=mongo,environment=test"
  volumeClaimTemplates:
  - metadata:
      name: mongo-persistent-storage
      annotations:
        volume.beta.kubernetes.io/storage-class: "fast"
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 100Gi
EOL
```

10. To deploy the Headless Service and the StatefulSet run the following command:

```
kubectl apply -f mongo-statefulset.yaml
```

11. Before connecting to the MongoDB replica set, validate it's running by running the following command:

```
kubectl get statefulset
```

12. You should receive an output as:

```
NAME    READY   AGE
mongo   3/3     4m20s
```

13. To list the pods in your cluster run the following command:

```
kubectl get pods
```

14. Connect to the first replica set member:

```
kubectl exec -ti mongo-0 mongo
```

15. To instantiate the replica set run the following command:

```
rs.initiate()
exit
```

16. If you need to scale the replica set, run the following command to increase the replica set from 3 to 5:

```
kubectl scale --replicas=5 statefuls
```

17. If you need to scale down the replica set, run the following command to decrease the replica set from 5 to 3:

```
kubectl scale --replicas=3 statefulset mongo
```

18. You can now connect to your MongoDB replica set using the following URI formatting

19.
```
"mongodb://mongo-0.mongo,mongo-1.mongo
```

```
NAME    READY   AGE
mongo   3/3     4m20s
```

20. Each pod in a StatefulSet backed by a Headless Service will have a stable DNS name. The template follows this format: <**pod-name** >.< **service-name**>

Discussion

In this recipe, you deployed MongoDB as a stateful application. Using a **headless service** allows you to describe policies or rules for accessing specific pods. Simply a headless service is one that doesn't prescribe load

balancing. When combined with StatefulSets, this will give you individual DNSs to access your pods which allows you to connect to all of our MongoDB nodes individually. The **StatefulSet** configuration is the configuration for the workload that runs MongoDB and what orchestrates the Kubernetes resources. The next part is the **terminationGracePeriodSeconds** which is used to gracefully shutdown the pod when you scale down the number of replicas. Finally you have the volumeClaimTemplates which connects to the StorageClass we created before to provision the volume. With this you have successfully deployed MongoDB on Kubernetes.

5.4 Resizing a Cluster

Problem

You are running a Kubernetes cluster that has either too few nodes, and thereby is unable to meet spiky demand for your application; or your Kubernetes cluster has too many nodes and is over provisioned for the level of traffic it is receiving.

Solution

You should resize the number of nodes your Kubernetes cluster is running in order to ensure you have setup an optimal cluster based on your application traffic patterns.

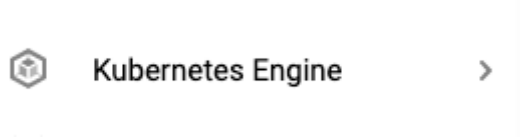
Prerequisites

Ensure the following APIs are enabled:

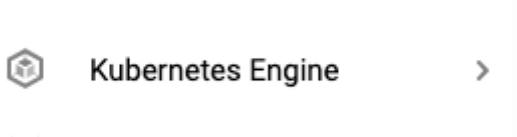
- Kubernetes Engine API

Ensure you have a Zonal or Regional cluster running that you can resize (see **Create a Zonal Cluster** or **Create a Regional Cluster** above)

1. Sign in to Google Cloud Console.
2. In the main menu, navigate to Compute and click on Kubernetes Engine



3. If you completed the recipes **Create a Zonal Cluster** or **Create a Regional Cluster**, you should have at least one Kubernetes cluster running. In the screenshot below, you will see we have two regional clusters running, one in **us-east1** and another in **us-west1**. For this recipe, we will increase the number of nodes in **cluster-1. (**Note: The process is the same for resizing a cluster's nodes, whether it's regional or zonal).
4. Click on the name of your cluster, here it is **cluster-1**
5. Now, click on Nodes, underneath the name of your cluster
6. You should now see one node pool, called **default-pool.** Click **default-pool** (or whatever the name of your particular node pool is).
7. Click **Edit** at the top of the node pool screen.
8. Now we are able to increase/decrease the default size of our node pool to any number of nodes that we prefer. In this example, we will increase the node pool size from 3 to 5. (Note: If you want your Kubernetes cluster to autoscale up based on node utilization, you can check the **Enable autoscaling** box. Checking the box will give you the option to set minimum and maximum node thresholds.).
9. Click the **Save** button at the bottom of the screen. You should see this screen:



10. This may take a minute or two, but once completed, you have effectively resized your default node pool from running 3 nodes to running 5 nodes in the **default-pool** nodepool.

Discussion

Resizing the number of nodes in a nodepool is a relatively simple process. The number of nodes you run, as well as the number of node pools you run, should be thought out carefully in order to meet the needs of your particular application. Enabling autoscaling in your node pools is a major value add that Kubernetes brings to the table, allowing your node pools to increase and decrease based on the utilization of that node and within a range you specify.

5.5 Load Testing with Locust

Problem

You want to run a load test on your kubernetes cluster(s). You want this load test to be executed on a single, or multiple, machines that are separate from your application. You want to write these load test scenarios in Python, and want the flexibility to be able to configure the test using a web based UI. This distributed load test should enable you to understand and validate how your application endures in real life scenarios of users using your website and/or web application.

Solution

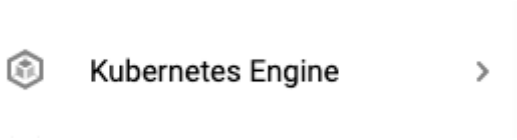
Use Locust to run a distributed load test against your application, and watch in real time how your deployment endures through the load test. Using Locust, you are able to test your application regardless of what infrastructure it runs on top of.

Prerequisites

Ensure the following APIs are enabled:

- Cloud Build API
- Kubernetes Engine
- Google App Engine Admin API
- Cloud Storage

1. Sign in to Google Cloud Console.
2. In the main menu, navigate to Compute and click on **Kubernetes Engine**



3. Click the **Create** button at the top of the screen
4. Click **Configure** next to the **Standard** option.
5. In the **Cluster Basics** section:

    1. In Name, set your cluster name to locust-cluster
    2. In Location Type, select Zonal
    3. In Zone, select us-central1-b

6. Now click **default-pool** under **Node Pools** in the left hand menu

    1. Check the box next to **Enable autoscaling**

2. Set the value of **Minimum number of nodes** to 3
3. Set the value of **Maximum number of nodes** to 10
4. The rest can remain on default settings

7. Click **Create** at the bottom of the screen
8. After a minute or so you should see this in the **Clusters** view in your console
9. Now let's get the credentials for the **locust-cluster** so we can execute kubectl commands on it from the cloud shell.
10. Open up the Cloud Shell by clicking this button in the top right corner of your screen
11. First, let's set a variable for our Project ID so we don't need to retype it every single time. Enter the following in your Cloud Shell and press enter:

```
PROJECT=$(gcloud info --format='value(co
```

12. Then, type the following into the cloud shell and press enter:

```
gcloud container clusters \
get-credentials locust-cluster --zone us
```

13. Now that we have the credentials to use this cluster from the command line, let's pull the git repository with the sample code for our web app, as well as our locust configuration. Enter the following to your cloud shell:

```
git clone \
https://github.com/GoogleCloudPlatform/distributed-load-
```

```
&& cd distributed-load-testing-using-kubernetes
```

14. Let's build the docker image for the Locust application, and store it in our local container repository. This could take a minute:

```
gcloud builds submit --tag \
gcr.io/$PROJECT/locust-tasks:latest docker-image
```

15. We will now launch the web application to AppEngine. To do this, run the following command in your cloud shell:

```
gcloud app deploy \
sample-webapp/app.yaml --project=$PROJECT
```

16. You may be prompted to select a region to deploy your App Engine application. You can feel free to select any region, but for the purposes of this tutorial you can simply choose **14** which is **us-central**
17. You may also be prompted with a summary of what you are deploying, and be given the **Y/n** option to proceed with deployment, in which you should enter **Y**. Note, the **target url** gives you the address of the AppEngine application's endpoint that you can open in a web browser.
18. Once deployed, enter the **target url**in a new browser tab and confirm you are able to load the website. It should, very simply, display **"Welcome to the "Distributed Load Testing Using Kubernetes" sample web app**.
19. Now that we've deployed our web application, let's configure and deploy our locust master and worker nodes to our **locust-cluster**.
20. First, let's add another variable to the cloud shell. Enter the following:

```
TARGET=“$PROJECT.appspot.com”
```

21. Now, we need to set the variables for our target web application and our project ID in the locust config files. You can do this quickly by running the following commands in your cloud shell:

```
sed -i -e "s/\[TARGET_HOST\]/$TARGET/g
        kubernetes-config/locust-master-controller.yaml
sed -i -e "s/\[TARGET_HOST\]/$TARGET/g"
        kubernetes-config/locust-worker-controller.yaml
sed -i -e "s/\[PROJECT_ID\]/$PROJECT/g"
        kubernetes-config/locust-master-controller.yaml
sed -i -e "s/\[PROJECT_ID\]/$PROJECT/g"
        kubernetes-config/locust-worker-controller.yaml
```

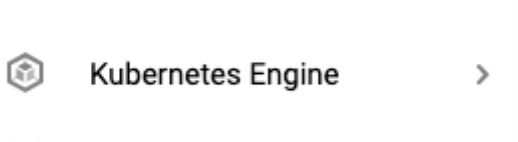
22. Now, we will deploy Locust to our **locust-cluster**. To do so, run the following in your cloud shell:

```
kubectl apply -f \
        kubernetes-config/locust-master-controller.yaml
kubectl apply -f \
        kubernetes-config/locust-master-service.yaml
kubectl apply -f \
        kubernetes-config/locust-worker-controller.yaml
```

23. Once completed, enter **kubectl get services** in the cloud shell and ensure your services are running. You should see something like this:
24. Open a new browser tab, and enter the **EXTERNAL-IP** for the **locust-master** service and append “:8089” into the URL bar (in our

example, that means 34.67.174.219:8089) and press enter. You should now see the Locust testing UI:

25. Ensure that the **HOST** field in the top right corner of the UI accurately has the web address of your web application that we deployed to AppEngine earlier. If it does not, please return to Step #20 and verify the commands were run correctly.
26. We can now run a simple load test by entering numbers for **Number of users to simulate**, and **Hatch rate**. For simple testing purposes, you can enter **50** for **Number of users to simulate** and **1** for the **Hatch rate**. Once entered, press **Start swarming**.



27. Congratulations! You are running a distributed load test on your sample web application. Once you're done observing how your application performs in the Locust UI, you can press **Stop** to end the test.

Discussion

In this example, you deployed a simple web application on AppEngine, and then launched a Locust deployment to your Kubernetes cluster. The Locust deployment was able to mimic real life traffic against your AppEngine application. Locust is a powerful tool and can be configured (using Python and the Web UI) to load and test any application. You can write your own test cases in Python to ensure all parts of your application are properly tested. More information about Locust can be found at www.locust.io.

5.6 Multi-Cluster Ingress

Problem

You have an application that runs on multiple Kubernetes clusters -- that are located in different regions -- and you want to be able to automatically route user traffic to the cluster that is nearest to them using a single HTTP(S) load balancer.

Solution

Use Multicluster Ingress for Anthos in order to run your application across as many Kubernetes clusters as you'd like, and route traffic to the nearest cluster based on the origin of the request.

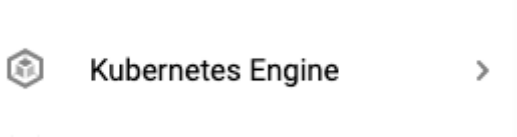
Prerequisites

Ensure the following APIs are enabled:

- Kubernetes Engine API
- GKE Hub
- Anthos
- Multi Cluster Ingress API

First we will create two regional clusters, in two different regions (us-east1 and us-west1)

1. Sign in to Google Cloud Console.
2. In the main menu, navigate to Compute and click on Kubernetes Engine



3. Click the Create button at the top of the screen
4. Click Configure next to the Standard option.
5. In the **Cluster Basics** section:

    1. In Name, set your cluster name to cluster-1
    2. In Location Type, select Regional
    3. In Region, select us-east1
    4. Leave the remaining settings to default

6. Click **Create** at the bottom of the screen

You will be navigated back to the **Clusters** screen where you will see your cluster spinning up. This process can take more than a minute to complete.

1. We will repeat this process and create another Regional cluster in a different region to the one we just created.
2. Click the **Create** button at the top of the screen
3. Click Configure next to the Standard option.
4. In the **Cluster Basics** section:

    1. In **Name**, set your cluster name to **cluster-2**
    2. In **Location Type**, select **Regional**
    3. In **Region,** select **us-west1**
    4. Leave the remaining settings to default

5. Click **Create** at the bottom of the screen
6. You will be navigated back to the Clusters screen where you will see your cluster spinning up. This process can take more than a

minute to complete.

7. Now we will register the clusters to the same **environment**.
8. In the main menu, navigate to Anthos and click on **Clusters** in the submenu
9. Now click **Register Existing Cluster**
10. You will now see both the clusters you created are ready to be registered
11. Next to **cluster-1**, hit **REGISTER**
12. You'll be asked for a Service Account for to register to the environ, choose Workload Identity
13. Click **Submit**
14. Repeat steps 16 - 18 for the second cluster, **cluster-2**
15. Now, we will set up **Ingress for Anthos**.
16. Next, click **Features** within the Anthos screen
17. Click Enable next to Ingress, then click Enable Ingress
18. In the Config Membership dropdown, select the first cluster you spun up (**cluster-1**) and click **Install**. After a minute or so, refresh the screen and you should see the following
19. Open up the cloud shell by clicking this button in the top right corner of your screen
20. Type the following into your cloud shell to make a directory that will hold the .yaml files we will need for the remainder of this tutorial:

```
mkdir multicluster-ingress-demo \
        && cd multicluster-ingress-demo
```

21. Before we can work with our clusters via kubectl in the cloud shell, we need to configure our cluster access by generating a kubeconfig entry. You can do this by running the following command for both of your clusters in the cloud shell:

```
gcloud container clusters \
        get-credentials cluster-1 --region us-east1
gcloud container clusters \
         get-credentials cluster-2 --region us-west1
```

22. Ensure you received a confirmation for each cluster:
23. Now we can work with the clusters from the cloud shell command line. Let's create the namespace for our application to run. You can do this by typing **nano namespace.yaml** into the cloud shell.
24. Paste this into the yaml file:

```
apiVersion: v1
kind: Namespace
metadata:
  name: zoneprinter
```

25. Save the file. Before we proceed, let's set the shell variable for our Project ID. Enter the following in the cloud shell:

```
PROJECT=$(gcloud info --format='value(config.project)')
```

26. Now let's apply **namespace.yaml** to both of our clusters, **cluster-1** and **cluster-2**. You can do this by running the following:

```
kubectl config use-context \
        gke_$(echo $PROJECT)_us-east1_cluster-1
kubectl apply -f namespace.yaml
kubectl config use-context \
        gke_$(echo $PROJECT)_us-west1_cluster-2
kubectl apply -f namespace.yaml
```

27. We will now deploy a sample app that shows the location of the datacenter you are reaching to both clusters, from an image called **zone-printer**.
28. Create a new yaml file by typing **nano app.yaml** in the gcloud terminal and paste the following into the yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: zone-ingress
  namespace: zoneprinter
  labels:
    app: zoneprinter
spec:
  selector:
    matchLabels:
      app: zoneprinter
  template:
    metadata:
      labels:
        app: zoneprinter
    spec:
      containers:
      - name: frontend
        image: gcr.io/google-samples/zone-printer:0.2
        ports:
        - containerPort: 8080
```

29. Save the file. Then let's apply **app.yaml** to both of our clusters, **cluster-1** and **cluster-2**. You can do this by running the following in the cloud shell:

```
kubectl config use-context \
gke_$(echo $PROJECT)_us-east1_cluster-1
kubectl apply -f app.yaml
```

```
kubectl config use-context \
gke_$(echo $PROJECT)_us-west1_cluster-2
kubectl apply -f app.yaml
```

30. Now that the app is running in both clusters, in the same namespace. Let's wrap up by creating the MultiClusterService and MultiClusterObject.
31. Let's first create the **MultiClusterService**. Create a new yaml file by typing **nano mcs.yaml** in the cloud shell, and paste the following into the yaml:

```
apiVersion: networking.gke.io/v1
kind: MultiClusterService
metadata:
  name: zone-mcs
  namespace: zoneprinter
spec:
  template:
    spec:
      selector:
        app: zoneprinter
      ports:
      - name: web
        protocol: TCP
        port: 8080
        targetPort: 8080
```

32. Save the file. Now, lets apply this file to **cluster-1**

```
kubectl config use-context \
                gke_$(echo $PROJECT)_us-east1_cluster-1
kubectl apply -f mcs.yaml
```

33. Lets now create the **MultiClusterIngress**. Create a new yaml file by typing **nano mci.yaml** in the cloud shell, and paste the following into the yaml:

```
apiVersion: networking.gke.io/v1
kind: MultiClusterIngress
metadata:
  name: zone-ingress
  namespace: zoneprinter
spec:
  template:
    spec:
      backend:
        serviceName: zone-mcs
        servicePort: 8080
```

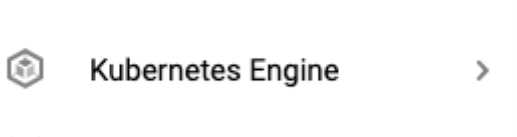
34. Save the file. Now, lets apply this file to **cluster-1**:

```
kubectl apply -f mci.yaml
```

35. Finally, let's pull the Virtual IP (VIP) to access our application from the MultiCluster Ingress. Run this command in your cloud shell:

```
kubectl describe mci zone-ingress -n zoneprinter
```

36. In the output, under the **Status** heading you will find an entry that says **VIP: <ip address>**. If you don't see **VIP: <ip address>** immediately, that's ok, the Ingress may take a few minutes to spin up. Keep running the describe command until you see the IP appear. The desired output should look similar to this:
37. Once you get the VIP, open a new tab, paste the VIP to the URL bar, and hit enter. You should see a webpage similar to this:



38. Congratulations! You've set up a Multicluster Ingress on your Kubernetes clusters running in different regions.

Discussion

In summary, and in order: we created two GKE clusters in **us-east1** and **us-west1**, then we registered the clusters to an **environ** and enabled the **Ingress for Anthos** feature, we then created the proper namespace and deployed the **zone-printer** application to both clusters, we then used **cluster-1** as our config cluster, and deployed a **Multicluster Service** and **Multicluster Ingress** to that cluster. The request now routes through a L7 HTTP Load Balancer and to the nearest cluster running the application from the location of the request. A **Multicluster Ingress** using **Ingress for Anthos** will allow you to route requests to your Kubernetes clusters running anywhere in the world

5.7 Continuous Delivery with Spinnaker and Kubernetes

Problem

With new application changes you need a way to automatically rebuild, retest, and redeploy the new application version to Kubernetes.

Solution

In this recipe you will use Google Cloud Source Repositories, Google Cloud Container Builder, and Spinnaker so when your application code changes it will trigger the continuous delivery pipeline to automatically rebuild, retest, and redeploy the new version.

1. Sign in to Google Cloud Console.
2. Launch the Google Cloud Shell.
3. Run the following command to instantiate a Kubernetes cluster:

```
gcloud container clusters create spinnaker-cluster \
        --machine-type=n1-standard-2
```

4. Create a Cloud Identity Access Management service account to delegate permissions to Spinnaker, allowing it to store data in Cloud Storage:

```
gcloud iam service-accounts create spinnaker-account \
        --display-name spinnaker-account
```

5. Store the service account email address and your current project ID in the Cloud Shell environment variables:

```
export SA_EMAIL=$(gcloud iam service-accounts list \
        --filter="displayName:spinnaker-account" \
        --format='value(email)')
export PROJECT=$(gcloud info \
        --format='value(config.project)')
```

6. Bind the **storage.admin** role to the newly created service account:

```
gcloud projects add-iam-policy-binding $PROJECT \
        --role roles/storage.admin \
```

```
        --member serviceAccount:$SA_EMAIL
```

7. Download the service account key, you will use this key when you install Spinnaker:

```
gcloud iam service-accounts keys create spinnaker-sa.jso
        --iam-account $SA_EMAIL
```

8. Create the Cloud Pub/Sub topic for notifications from Container Registry:

```
gcloud pubsub topics create projects/$PROJECT/topics/gcr
```

9. Create a subscription so Spinnaker can receive notifications of images being pushed:

```
gcloud pubsub subscriptions create gcr-triggers \
        --topic projects/${PROJECT}/topics/gcr
```

10. Provide the Spinnaker service account access the Pub/Sub subscription:

```
export SA_EMAIL=$(gcloud iam service-accounts list \
        --filter="displayName:spinnaker-account" \
        --format='value(email)')
gcloud beta pubsub subscriptions \
        add-iam-policy-binding gcr-triggers \
        --role roles/pubsub.subscriber --member \
        serviceAccount:$SA_EMAIL
```

11. To install Spinnaker you will be using Helm. Helm is a package manager you can use to deploy Kubernetes applications. Download and install Helm:

```
wget https://get.helm.sh/helm-v3.1.1-linux-amd64.tar.gzi
```

12. Extract the files from the downloaded archive file

```
tar zxfv helm-v3.1.1-linux-amd64.tar.gz
cp linux-amd64/helm .
```

13. Run the following command to grant Helm access to your cluster:

```
kubectl create clusterrolebinding user-admin-binding \
        --clusterrole=cluster-admin \
        --user=$(gcloud config get-value account)
```

14. Run the following command to grant Spinnaker the cluster-admin role so it can deploy resources across all namespaces:

```
kubectl create clusterrolebinding \
        --clusterrole=cluster-admin \
        --serviceaccount=default:default spinnaker-admin
```

15. Add the stable charts deployments to Helm's usable repositories

```
./helm repo add stable https://charts.helm.sh/stable
./helm repo update
```

16. Create a bucket for Spinnaker to store its pipeline configuration:

```
export PROJECT=$(gcloud info \
        --format='value(config.project)')
export BUCKET=$PROJECT-spinnaker-config
gsutil mb -c regional -l us-central1 gs://$BUCKET
```

17. Run the following command to create a spinnaker-config.yaml file, which describes how Helm should install Spinnaker:

```
export SA_JSON=$(cat spinnaker-sa.json)
export PROJECT=$(gcloud info --format='value(config.proj
export BUCKET=$PROJECT-spinnaker-config
cat > spinnaker-config.yaml <<EOF
gcs:
  enabled: true
  bucket: $BUCKET
  project: $PROJECT
  jsonKey: '$SA_JSON'
dockerRegistries:
- name: gcr
  address: https://gcr.io
  username: _json_key
  password: '$SA_JSON'
  email: 1234@5678.com
# Disable minio as the default storage backend
minio:
  enabled: false
# Configure Spinnaker to enable GCP services
halyard:
  spinnakerVersion: 1.19.4
  image:
    repository: us-docker.pkg.dev/spinnaker-community/do
    tag: 1.32.0
    pullSecrets: []
  additionalScripts:
```

```
    create: true
    data:
      enable_gcs_artifacts.sh: |-
        \$HAL_COMMAND config artifact gcs account add gc
        \$HAL_COMMAND config artifact gcs enable
      enable_pubsub_triggers.sh: |-
        \$HAL_COMMAND config pubsub google enable
        \$HAL_COMMAND config pubsub google subscription
          --subscription-name gcr-triggers \
          --json-path /opt/gcs/key.json \
          --project $PROJECT \
          --message-format GCR
EOF
```

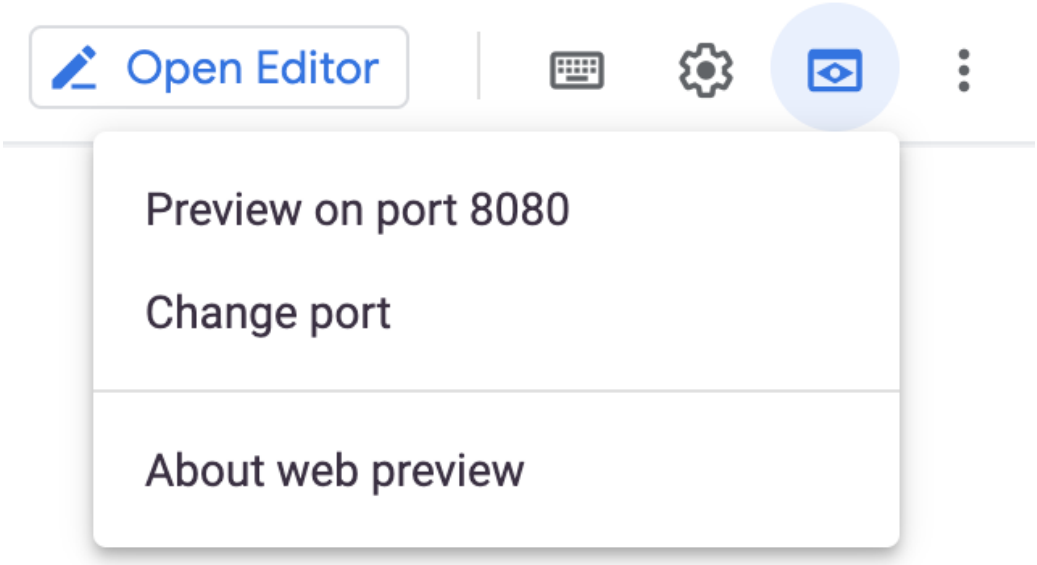
18. Use the Helm command-line interface run the following command to deploy the chart with your configuration set:

```
./helm install -n default cd stable/spinnaker \
        -f spinnaker-config.yaml \
        --version 2.0.0-rc9 --timeout 10m0s --wait
```

19. The command on step 18 will take a few minutes to complete. Once completed you can continue.
20. Run the following commands to set up port forwarding to Spinnaker:

```
        export DECK_POD=$(kubectl get pods --namespace \
                default -l "cluster=spin-deck" \
                -o jsonpath="{.items[0].metadata.name}")
kubectl port-forward --namespace default \
                $DECK_POD 8080:9000 >> /dev/null &
```

21. To open the newly deployed Spinnaker interface, click the Web Preview icon in the Cloud Shell and select Preview on Port 8080



22. You should see the Spinnaker interface
23. To allow Cloud Build to monitor changes to your source code, you will need to build a Docker image and push it to the Container registry.
24. In your Cloud Shell download the sample source code provided by Google:

```
cd ~
wget https://gke-spinnaker.storage.googleapis.com/sample
```

25. Unpack the source code:

```
tar xzfv sample-app-v4.tgz
```

26. Change directories to the source code:

```
cd sample-app
```

27. Make the first commit to your source code repository:

```
git init
git add .
git commit -m "First commit"
```

28. Create a source repository to store your source code:

```
gcloud source repos create sample-app
git config credential.helper gcloud.sh
```

29. Add your newly created repository as remote:

```
export PROJECT=$(gcloud info \
        --format='value(config.project)')
git remote \
        add origin\
        https://source.developers.google.com/p/$PROJECT/
```

30. Push your code to the new repository's main branch:

```
git push origin master
```

31. In the next steps you will configure Container Builder to build and push Docker images everytime you make changes to your source code.
32. In the Cloud Platform Console, click **Navigation menu** > **Cloud Build** > **Triggers**.
33. Click **Create trigger**.

34. Set the following trigger settings:

    1. **Name**: sample-app-tags
    2. **Event**: Push new tag

35. Select your newly created sample-app repository.

    1. **Tag**: v1.*
    2. **Build configuration**: Cloud Build configuration file (yaml or json)
    3. **Cloud Build configuration file location**: /cloudbuild.yaml

36. Click Create
37. Run the following commands to allow Spinnaker needs access to your Kubernetes manifests:

```
export PROJECT=$\
        (gcloud info --format='value(config.project)')
gsutil mb -l us-central1 gs://$PROJECT-kubernetes-manife
```

38. Enable versioning on the bucket so that you have a history of your manifests:

```
gsutil versioning set on gs://$PROJECT-kubernetes-manife
```

39. Set the your project ID in your kubernetes deployment manifests:

```
sed -i s/PROJECT/$PROJECT/g k8s/deployments/*
```

40. Commit the changes to the repository:

```
git commit -a -m "Set project ID"
```

41. Push your first image:

```
git tag v1.0.0
```

42. Push the tag:

```
git push --tags
```

43. Go to the Cloud Console. In Cloud Build, click **History** in the left pane to check that the build has been triggered.
44. You will now set up your container to deploy to the Kubernetes cluster automatically.
45. Download the 1.14.0 version of spin CLI:

```
curl -LO https://storage.googleapis.com/spinnaker-artifa
```

46. Make spin executable:

```
chmod +x spin
```

47. Use the spin CLI to create an app called sample in Spinnaker.

```
./spin application save --application-name sample \
--owner-email "$(gcloud config get-value core/account)"
        --cloud-providers kubernetes \
        --gate-endpoint http://localhost:8080/gate
```

48. From your sample-app source code directory, run the following command to upload an example pipeline to your Spinnaker instance:

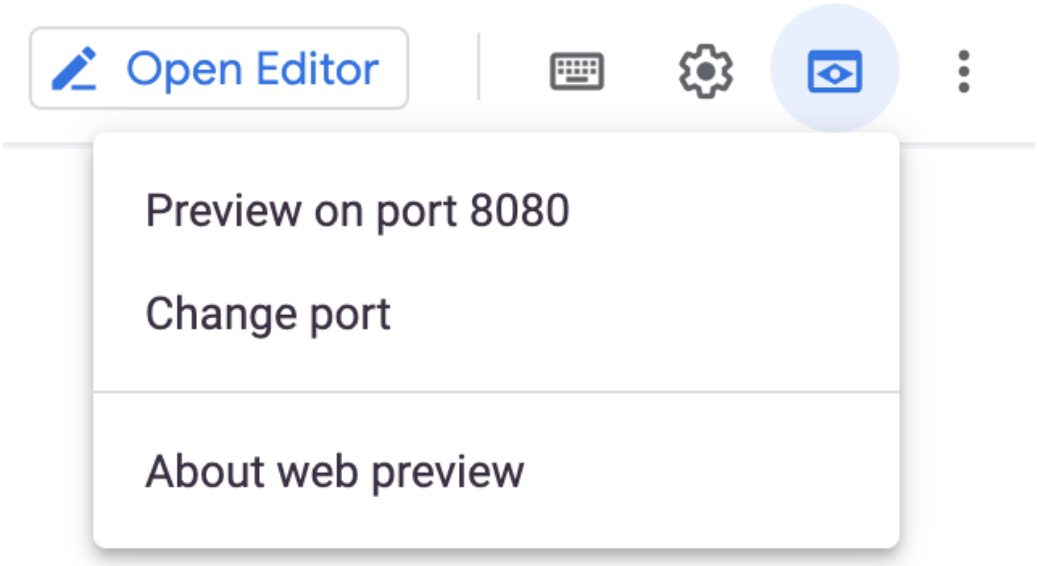
```
export PROJECT=$(gcloud info --format='value(config.proj
sed s/PROJECT/$PROJECT/g spinnaker/pipeline-deploy.json
./spin pipeline save --gate-endpoint http://localhost:80
```

49. By pushing a Git tag that starts with “v”, you trigger Container Builder to build a new Docker image and push it to Container Registry. Spinnaker detects that the new image tag begins with “v” and triggers a pipeline to deploy the image to all pods in the deployment.
50. From your sample-app directory, change the source code of your application:

```
sed -i 's/orange/blue/g' cmd/gke-info/common-service.go
```

51. Tag your change and push it to the source code repository:

```
git commit -a -m "Change color to blue"
git tag v1.0.1
git push --tags
```



52. Congratulations you have completed the auto deployment of your sample application to Kubernetes Engine.

5.8 Deploy a Spring Boot Java Application

Problem

You need to deploy a Java Spring Boot REST service to Kubernetes.

Solution

In this recipe you will use Google Cloud Source Repositories, Google Cloud Container Builder, and Jib to containerize and deploy the Spring Boot REST service to a Kubernetes cluster.

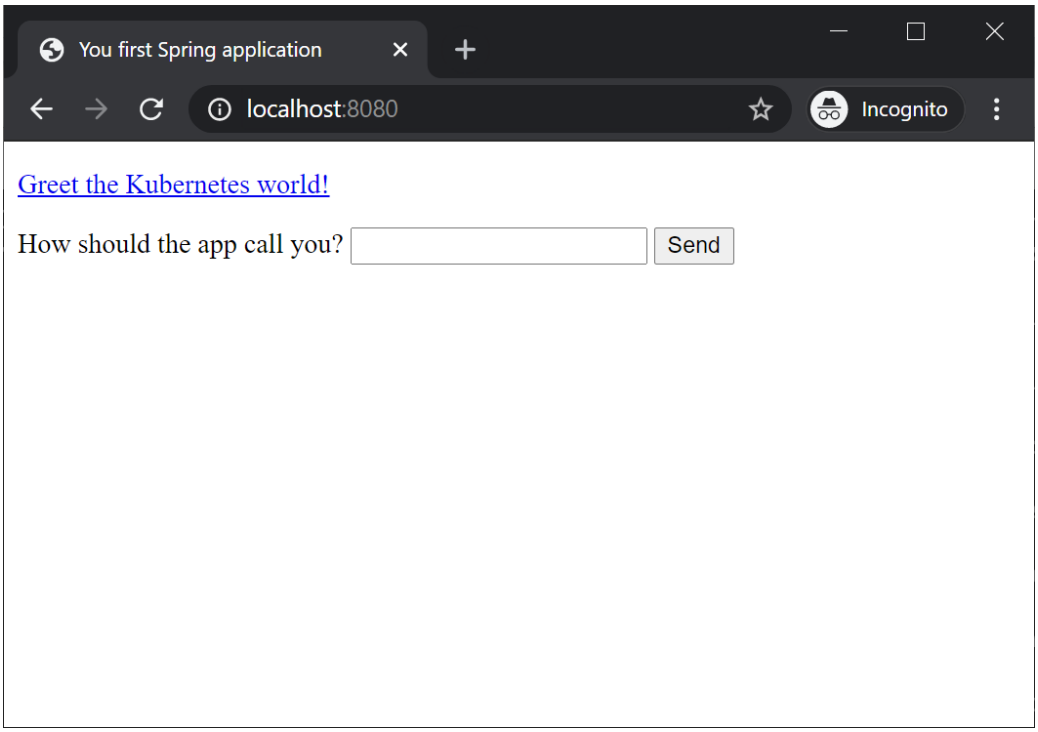
For this recipe you will need to git clone the following repository to your local workstation:https://github.com/ruiscosta/google-cloud-cookbook/06-

kubernetes

1. On your local workstation go to the working directory for this recipe from the cloned repository.
2. cd google-cloud-cookbook/06-kubernetes/6-8-java
3. Test the sample java application locally by running the following command:

```
./mvnw -DskipTests spring-boot:run
```

4. In your browser go to http://localhost:8080 you should see a similar screen:



5. Run the following command to enable Google Cloud Container Registry API to store the container image:
6. gcloud services enable containerregistry.googleapis.com

7. Use Jib to create the container image and push it to the the Container Registry, replace $GOOGLE_CLOUD_PROJECT with your Google Cloud Project ID:

```
mvn compile \
        com.google.cloud.tools:jib-maven-plugin:2.0.0:bu
        -Dimage=gcr.io/$GOOGLE_CLOUD_PROJECT/hello-java:
```

8. Build and push the image to a container registry:

```
mvn compile jib:build \
        -Dimage=gcr.io/ruicosta-blog/hello-java:v1
```

9. To test Docker installation run the following command:

```
mvn compile jib:dockerBuild \
        -Dimage=gcr.io/ruicosta-blog/hello-java:v1
```

10. To list the Docker images run the following command:

```
docker images
```

11. You should see an output similar to the following:
12. Run the following command to run the Docker container locally on your machine, replace the Image ID with yours from step 9:

```
docker run -p 8080:8080 -t IMAGE_ID
```

13. In your browser go to http://localhost:8080 and you should see a similar screen to step 3. You have now tested the Spring Boot

Application containerized application locally.

14. Create a Kubernetes two node cluster

```
gcloud container clusters create hello-java-cluster \
        --num-nodes 2 \
        --machine-type n1-standard-1 \
        --zone us-central1-c
```

15. A Kubernetes deployment can create, manage, and scale multiple instances of your application using the container image that you created. Deploy one instance of your application to Kubernetes cluster, replace the the GOOGLE_CLOUD_PROJECT with your Google Cloud Project ID:

```
kubectl create deployment hello-java \
        --image=gcr.io/$GOOGLE_CLOUD_PROJECT/hello-java:
```

16. In order to make the hello-java container accessible from outside the Kubernetes cluster, you have to expose the Pod as a Kubernetes service:

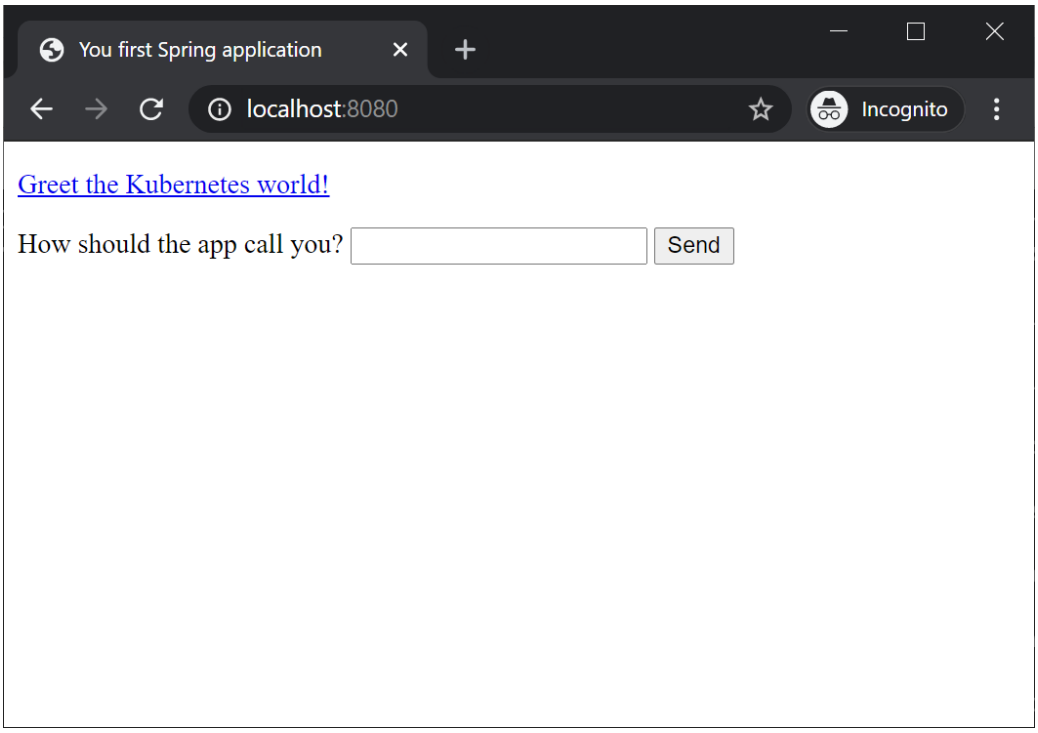
```
kubectl create service loadbalancer hello-java --tcp=808
```

17. To find the publicly accessible IP address of the service, run the following command:

```
kubectl get services
```

18. You should see a similar output,

19. Visit in your web browser http://EXTERNAL-IP:8080, you should see a similar page:
20. You have successfully deployed a Spring Boot Java application to a Kubernetes Cluster.



Discussion

In this recipe, you deployed a Java Spring Boot application to Google Cloud Kubernetes Engine. You leveraged Jib which builds containers without having to declare a Dockerfile. Jib was developed by Google to simplify the process of building Java containers, no need to create a Docker file, or wait for the build to complete. With Jib it handles all the steps required to build your Java container.

5.9 Skaffold

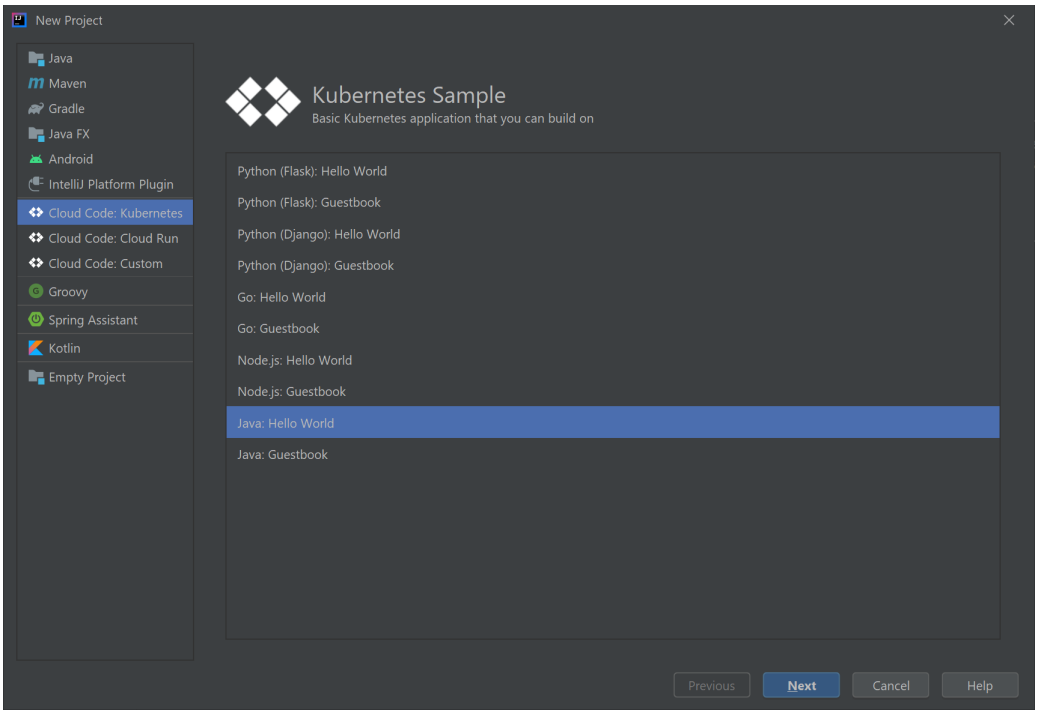
Problem

You need a method to develop, build, push and deploy your Java application quickly to Kubernetes.

Solution

In this recipe you will use Skaffold, and Cloud Code plugin for IntelliJ to develop, build, push and deploy your application to Kubernetes all from the IntelliJ IDE.

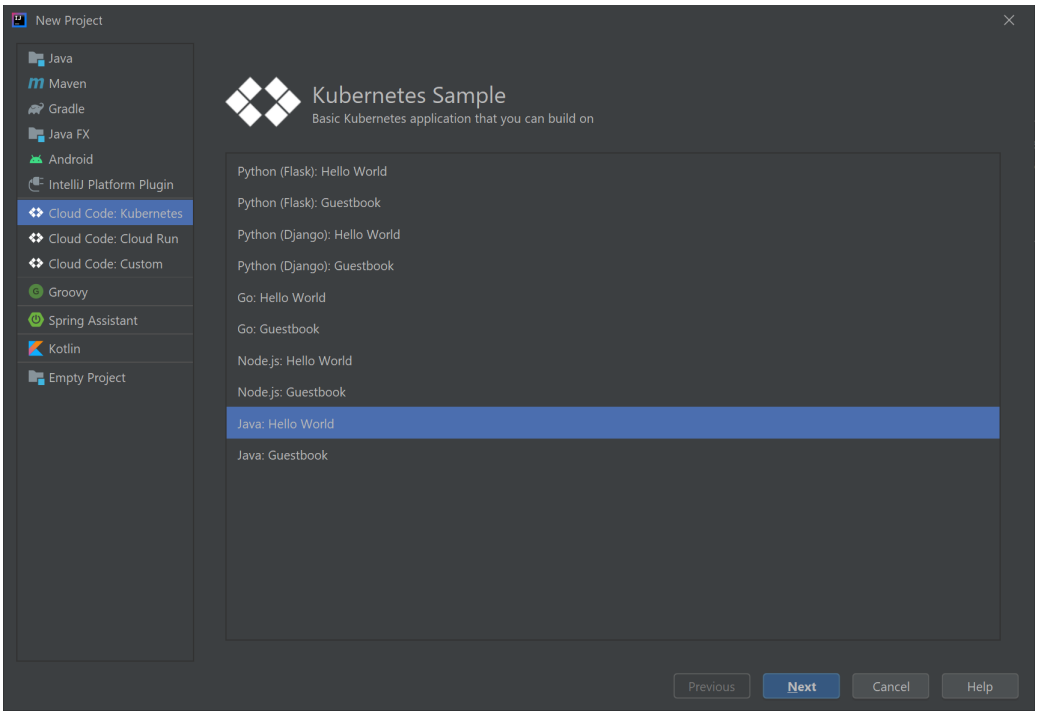
1. Install Cloud Code for IntelliJ by following this guide: https://cloud.google.com/code/docs/intellij/install
2. Create a new IntelliJ project.
3. Choose **Cloud Code: Kubnertes** > **Java: Hello World**



4. Enter the location of your Container repository.
5. Choose a project name and location for your project files.
6. Navigate to the Kubernetes Explorer from the right side panel, or by going to **Tools** > **Cloud Code** > **Kubernetes** > **View Cluster**

Explorer

7. Select Add a new GKE Cluster and click Create a New GKE Cluster
8. This will open the Google Cloud Console in a Web Browser to the cluster wizard page. In the Google Cloud Console create a new cluster.
9. Once you cluster is created your screen should update with the cluster name similar to the image below:
10. Click **OK**
11. Click **Run Develope on Kubernetes**
12. Once the process is complete, you should see the Workload created in the Google Cloud Console.
13. Besides making it easy to deploy your application from IntelliJ to Google Cloud, it also tunnels the traffic from your local workstation to Kubenrtes. In your web browser go to http://localhost and you should see a screen similar to the image below. The application is now running on a Google Cloud Kubernetes cluster in your Google Cloud project.



5.10 GKE Autopilot

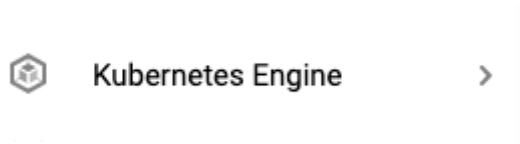
Problem

You want to run your application on Kubernetes, but don't want to have to manage nodes, nodepools, images, networking, and the other operational components of running a Kubernetes cluster. Effectively, you want to run your application(s) on Kubernetes while not having to worry about the management or operation of the cluster itself.

Solution

Run your application on GKE Autopilot. With GKE autopilot, many operational aspects of Kubernetes are abstracted away, and you are left with a Kubernetes infrastructure that is largely configured to Google GKE best practices.

1. Sign in to Google Cloud Console.
2. In the main menu, navigate to Compute and click on **Kubernetes Engine**



3. Click the **Create** button at the top of the screen
4. Click **Configure** next to the **Autopilot** option.
5. In the **Cluster Basics** section:

    1. In **Name,** give your autopilot cluster any name of your choice
    2. In **Region,** you can pick any region of your choice

6. The remaining options can be left as default. Click the **Create** button at the bottom of the screen
7. The cluster should take a minute or so to spin up and be ready for usage

Discussion

At the time of writing, GKE Autopilot is a new GCP offering that allows users to run Kubernetes clusters in a more managed format versus running GKE in Standard mode. The managed aspect of GKE Autopilot reduces the users full control over the cluster in exchange for an ease of operational overhead. GKE Autopilot is an exciting mode of operation that enables users to get started much more quickly with deploying their production workloads in GCP.

Chapter 6. Working with Data

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 7th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at ruiscosta@google.com and dhodun@google.com.

Without question, one of the greatest paradigm shifts when working with Cloud Computing is the near unlimited storage now available to users. Cheap, scalable blob storage, in the form of GCS allows administrators to start from a standpoint of "never delete data". Separation of compute and storage in the cases of BigQuery, Spark on Dataproc with persistence on GCS, etc extends this model and allows users to pay for only what they use of the expensive component (compute) while storing as much as possible, generally saving on engineering effort. These recipes show clever tips and tricks when working with the various data layers of Google Cloud, from how to move data round GCS buckets faster, to extending PubSub functionality, to automatically archiving long-term data.

6.1 Speeding up GCE Transfers - Multiprocessing

Problem

You want to increase the speed for a bulk transfer, either to/from GCS or within the GCS service.

Solution

You can leverage the '-m' flag to multiprocess your transfer.

1. Create a bucket and upload data to observe normal single-process transfer speed

```
BUCKET_NAME=my-bucket-4312
gsutil mb -l gs://$BUCKET_NAME
# can cancel once you see transfer speed, will move ~250
gsutil cp -r gs://gcp-public-data-landsat/LC08/01/044/03
```

2. Add '-m' to multi-process your transfer and observe greatly increased speed.

```
# can cancel once you see greatly increased transfer spe
gsutil -m cp -r gs://gcp-public-data-landsat/LC08/01/044
```

3. Delete test files

```
gsutil -m rm -r  gs://$BUCKET_NAME/034/*
```

6.2 Speeding up GCS Transfers - Parallel Composite Uploads for large

files

Problem

You want to increase the speed for a bulk transfer, particularly to/from GCS service for large files.

Solution

You can leverage parallel composite uploads, set at command line or in your .boto configuration file.

1. Create a bucket and download a single file to then perform test uploads

```
BUCKET_NAME=my-bucket-4312
gsutil mb -l gs://$BUCKET_NAME
# copy a largish file locally (~250MB)
gsutil cp gs://gcp-public-data-landsat/LC08/01/044/017/L
```

2. Upload the file as a single chunk and observe transfer speed.

```
gsutil cp LC08_L1GT_044017_20200809_20200809_01_RT_B8.TI
```

3. Upload the file as several simultaneous chunks and observe transfer speed. If your previous upload saturated your link, you may not see a performance increase.

```
gsutil -o "GSUtil:parallel_composite_upload_threshold=20
```

4. Delete the file

```
gsutil rm gs://$BUCKET_NAME/LC08_L1GT_044017_20200809_20
```

Discussion

You can leverage parallel composite uploads, set at command line or in your .boto configuration file. This is particularly helpful when uploading large files. However, this requires that both the source and destination environments have a CRC32C library installed for integrity checking. There are additional caveats, such as a maximum of 32 objects per composite.

6.3 Adding event timestamps to PubSub

Problem

The `timestamp` attribute on a PubSub message indicates the publish time, that is, when the message reached the PubSub service and potentially not when the event actually was created, particularly in offline scenarios. You want to process based on event time, not publish time.

Solution

You can leverage PubSub metadata, which can then be consumed by downstream applications.

1. Generate a timestamp as part of your payload dictionary

```
def generate_event():
    return {
        'user_id': random.randint(1, 100),
        'action': random.choices(['start', 'stop', 'rewi
        'timestamp': datetime.datetime.utcnow().strftime
    }
```

2. When you publish to PubSub, add a 'timestamp' field in addition to the encoded data

```
def publish_burst(publisher, topic_path, buffer):
    for message in buffer:
        json_str = json.dumps(message)
        data = json_str.encode('utf-8')
        publisher.publish(topic_path, data, timestamp=me
        print('Message for event {} published at {}'.for
```

3. Now your PubSub messages will have an event 'timestamp' attribute you can address in addition to the automatically set 'publishTime' metadata which you can use to order your data based on when it occurs. Dataflow, for example, allows you to use event_timestamp vs the publish time with the following (in Java)

```
pipeline.apply("ReadMessage", PubsubIO.readStrings()
                        .withTimestampAttribute("timesta
                        .fromTopic(options.getInputTopic
```

Discussion

You can set attributes in PubSub messages that can be intelligently consumed in downstream data, in this case an even-timestamp attribute.

Dataflow is smart enough to not only use this timestamp in place of publishing timestamp for time-based processing such as windowing, but it is able to efficiently use this metadata to update it's watermark - the internal mechanism by which it tracks how up-to-date data is. https://stackoverflow.com/questions/42169004/what-is-the-watermark-heuristic-for-pubsubio-running-on-gcd.

6.4 Mounting GCS as a File-System (sort of)

Problem

You want to use traditional filesystem based tools to interact with GCS blobs and 'directories'

Solution

You can use GCSFuse as a beta-quality convenience option to mount a GCS bucket to your VM. These are the Linux instructions.

1. Create a test VM:

```
gcloud compute --project=dhodun1 instances create gcs-fu
    --zone=us-central1-a --machine-type=e2-medium \
    --scopes=https://www.googleapis.com/auth/cloud-platf
    --image=ubuntu-2004-focal-v20210315 --image-project=
    --boot-disk-size=100GB
```

2. Create your bucket and populate some test data

```
BUCKET_NAME=my-bucket-4312
gsutil mb -l gs://$BUCKET_NAME
gsutil -m cp -r gs://gcp-public-data-landsat/LC08/01/044
```

3. SSH Onto the VM
4. Install FUSE (Ubunutu / Debian latest releases)

```
export GCSFUSE_REPO=gcsfuse-`lsb_release -c -s`
echo "deb http://packages.cloud.google.com/apt $GCSFUSE_
curl https://packages.cloud.google.com/apt/doc/apt-key.g
```

5. Check the current gcloud credentials to see what GCS fuse will use

```
gcloud auth list
```

6. Create mount point and mount:

```
BUCKET_NAME=my-bucket-4312
mkdir $BUCKET_NAME
gcsfuse --implicit-dirs $BUCKET_NAME $BUCKET_NAME
```

7. Add a file and test that it's indeed on GCS

```
ls $BUCKET_NAME
cd $BUCKET_NAME
echo "gcs fuse is cool!" > file.txt
cd
gsutil cat gs://$BUCKET_NAME/file.txt
```

8. Add entry to /etc/fstab to automatically mount

```
vim /etc/fstab
# add:
my-bucket-4312 /home/dhodun/my-bucket-4312 gcsfuse rw,im
```

9. Restart VM and test

```
sudo shutdown -r now
```

Discussion

GCS Fuse is a great beta-quality convenience tool for mounting and accessing GCS blobs. It is not a production replacement for a traditional file store (see Cloud Filestore), and lacks many features of a true POSIX filesystem, as well as some GCS features such as metadata. Also note that you will still incur GCS storage charges, particularly for nearline and coldline (archival) storage. See: https://cloud.google.com/storage/docs/gcs-fuse#notes.

6.5 Designing your Schema for Spanner with Interleaved Tables

Problem

You want to speed up certain queries in Spanner and child tables in your schema design container primary keys from the parent table.

Solution

You can leverage “Interleaved Tables” in Spanner to co-located on disk the child data with the parent data based on a primary key. This is an alternative to “Foreign Keys”.

1. Normally for basic, abbreviated star-schema setup you would create databases with similar primary keys as such.

```
-- Schema hierarchy:
-- + Customers (sibling table of Orders)
-- + Orders (sibling table of Customers)
CREATE TABLE Customers (
  CustomerId INT64 NOT NULL,
  FirstName  STRING(1024),
  LastName   STRING(1024),
  Address     STRING(1024),
) PRIMARY KEY (CustomerId);
CREATE TABLE Orders (
  OrderId       INT64 NOT NULL,
  OrderTotal    INT64 NOT NULL,
  QuantityItems INT64 NOT NULL,
  OrderItems   ARRAY(INT64),
) PRIMARY KEY (CustomerId, OrderId);
```

2. In Cloud Spanner, you can indicate that one table is a child of another, so long it contains a common Primary Key. In this case all the orders will be laid out on disk under a given CustomerId.

```
-- Schema hierarchy:
-- + Customers
--   + Orders (interleaved table, child table of Orders)
CREATE TABLE Customers (
  CustomerId INT64 NOT NULL,
  FirstName  STRING(1024),
  LastName   STRING(1024),
  Adress     STRING(1024),
```

```
) PRIMARY KEY (CustomerId);
CREATE TABLE Orders (
  OrderId       INT64 NOT NULL,
  OrderTotal    INT64 NOT NULL,
  QuantityItems INT64 NOT NULL,
  OrderItems   ARRAY(INT64),
) PRIMARY KEY (CustomerId, OrderId),
  INTERLEAVE IN PARENT Customers ON DELETE CASCADE;
```

Discussion

Interleaved Tables stores rows physically under its associated parent row, greatly speeding up some queries. In this case, if you wanted to perform general analytics on a given Customer, for example, what is their total spend for last year, or how many items have they ever bought, these analytic queries would complete much faster with Interleaved tables since all the necessary data is physically co-located. The alternative is the database engine has to perform a standard key join on the CustomerId to find all the orders from the order table, which might be a more expensive (slower) operation if the table is organized by timestamp, for example.

6.6 Automatically archiving and deleting objects on GCS

Problem

You want to automatically handle lifecycle management of GCS objects - namely changing the storage class to more archival-friendly classes as files get older, and delete the oldest files according to a policy.

Solution

You can leverage lifecycle management on a GCS bucket level with policies based on an object's age (or other attributes) to change storage class, delete, and other actions.

1. Create a bucket to be managed:

```
BUCKET_NAME=my-bucket-4312
gsutil mb -l gs://$BUCKET_NAME
```

2. Create a 'lifecycle_management.json' file with your bucket's policy. This policy will archive standard and DRA storage to nearline after 1 year, nearline to coldline after 3 years, and delete items after 7 years. It will not archive multi-regional objects - in this case they are assumed to be serving objects.

```
{
    "lifecycle": {
        "rule": [
            {
                "condition": {
                    "age:": 365,
                    "matchesStorageClass": ["STANDARD",
                },
                "action": {
                    "type:": "SetStorageClass",
                    "storageClass": "NEARLINE"
                }
            },
            {
                "condition": {
                    "age:": 1096,
                    "matchesStorageClass": ["NEARLINE"]
                },
                "action": {
                    "type:": "SetStorageClass",
```

```
                    "storageClass": "COLDLINE"
                }
            },
            {
                "condition": {
                    "age:": 2555,
                    "matchesStorageClass": ["COLDLINE"]
                },
                "action": {
                    "type:": "Delete"
                }
            }
        ]
    }
}
```

3. Apply the policy to the bucket:

```
gsutil lifecycle set lifecycle_config.json gs://$BUCKET_
```

4. Check the lifecycle

```
gsutil lifecycle get gs://$BUCKET_NAME
```

5. To remove lifecycle management, you apply a lifecycle config with no rules:

```
{
    "lifecycle": {
        "rule": []
    }
}
```

6. Apply:

```
gsutil lifecycle set no_lifecycle.json gs://$BUCKET_NAME
```

Discussion

Lifecycle management is a powerful, simple way to save storage costs and manage a large fleet of objects. Understanding different archival classes, in particular minimum costs for archival classes like NEARLINE and COLDLINE is important. For example, you pay for a minimum storage duration of 90 days for COLDLINE - if you delete an object 1 minute after creating it, you will still pay as if the object existed 90 days. For this reason, and the fact that some storage policies can permanently delete data, tight control and review of these policies is prudent.

6.7 Locking down Firestore Database so a User can edit only their data

Problem

You want to secure Firestore so that an authenticated user can create a document or edit their previous documents.

Solution

Firestore security rules can not only filter on whether a user is authenticated, but match their UserID to documents in the database, allowing for a simple per-user data authentication model.

1. Initiate Firebase in your main project directory

```
Firebase init
```

2. Edit the created ".rules" file, in our case firestore.rules. This config allows users to edit their User document (and only theirs) by matching the authenticated UserId with the UserId in the document database.

```
rules_version = '2';
service cloud.firestore {

    // Matches to the default database in the project -
    match /databases/{database}/documents {
        // Matches the userId on the autenticated reques
        // in the users collection
        // Otherwise, allows authenticated users to crea
        match /users/{userId} {
            allow read, update, delete: if request.auth
            allow create: if request.auth != null;
        }
    }
}
```

3. Apply the rules file (without redeploying the entire project)

```
firebase deploy --only firestore:rules
```

Discussion

Firestore rules and validation provides robust but concise functionality for controlling who can edit or read what in the Firestore database. This includes setting public access for certain fields, validating that newly written input data is formed correctly, and the above example. Note that

Firestore rules are not filters - i.e. in the above case, you could not run a query as this User to return all User documents and expect the security rule to only return this user's document. This query would fail since some (basically all other) documents would not be accessible to the user. You would need to query on for this user's document.

Recipe ideas:

- Gsutil rsync
- Pubsub dedupe?
- Pubsub snapshot
- Mysql something?