## 8. Serverless Cloud Run Development

* learn how to do the following using Cloud Run by connecting and leveraging data stored in Cloud Storage, building a resilient, asynchronous system with Cloud Run and Pub/Sub, building a REST API gateway using Cloud Run, building and exposing service using Cloud Run.

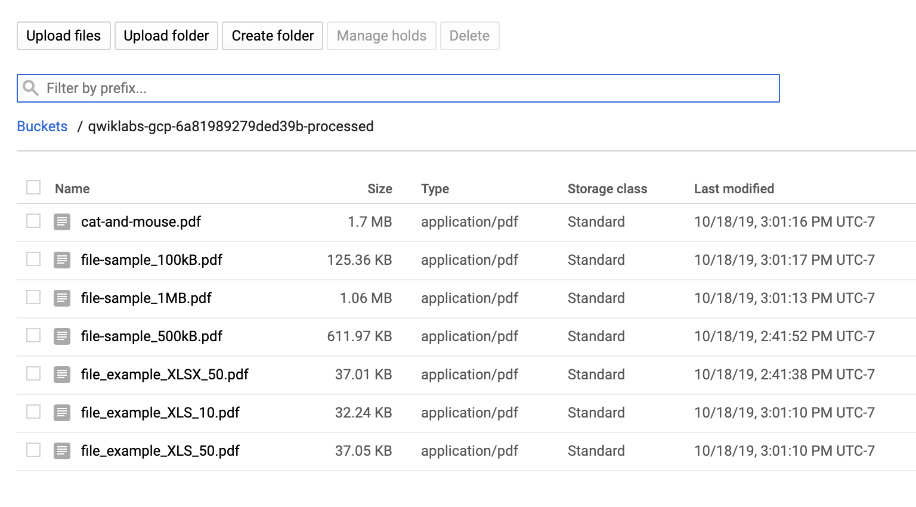
## 8.1 Build a Serverless App with Cloud Run that Creates PDF Files

* Overview
  + For the labs in the Google Cloud Serverless Workshop: Pet Theory Quest, you will read through a fictitious business scenario and assist the characters with their serverless migration plan.
  + Twelve years ago, Lily started the Pet Theory chain of veterinary clinics. Pet Theory currently sends invoices in DOCX format to clients, but many clients have complained that they are unable to open them. To improve customer satisfaction, Lily has asked Patrick in IT to investigate an alternative to improve the current situation.
  + Pet Theory's Ops team is a single person, so they are keen to invest in a cost efficient solution that doesn't require a lot of ongoing maintenance. After analyzing the various processing options, Patrick decides to use Cloud Run.
  + Cloud Run is serverless, so it abstracts away all infrastructure management and lets you focus on building your application instead of worrying about overhead. As a Google serverless product, it is able to scale to zero, meaning it won't incur cost when not used. It also lets you use custom binary packages based on containers, which means building consistent isolated artifacts is now feasible.
  + In this lab you will build a PDF converter web app on Cloud Run that automatically converts files stored in Cloud Storage into PDFs stored in seperated folders.
* Architecture
  + This diagram gives you an overview of the services you will be using and how they connect to one another:
    - Graphical user interface, application

      Description automatically generated
* What you will learn
  + Convert a Node JS application to a container.
  + Build containers with Google Cloud Build.
  + Create a Cloud Run service that converts files to PDF files in the cloud.
  + Use event processing with Cloud Storage
* Understanding the task
  + Pet theory would like to convert their invoices into PDFs so that customers can open them reliably. The team wants to accomplish this conversion automatically to minimize the workload for Lisa, the office manager.
  + Ruby, Pet Theory's computer consultant, gets a message from Patrick in IT...
    - Patrick, IT Administrator
      * Hi Ruby,
        + I've done some research and found that LibreOffice is good at converting many different file formats to PDF.
        + Would it be possible to run LibreOffice in the cloud without having to maintain the servers?
      * Patrick
    - Software Consultant
      * Hey Patrick,
        + I think I have just the thing for this type of situation.
        + It looks like we can run LibreOffice in a serverless environment with Cloud Run. No server maintenance is needed!
        + I'll send over some resources that will help you get set up.
      * Ruby
  + Help Patrick set up and deploy Cloud Run.
* Enable the Cloud Run API
  + navigation > APIs & Services > Library > enable Cloud Run API
* Deploy a simple Cloud Run service
  + Ruby has developed a Cloud Run prototype and would like Patrick to deploy it onto Google Cloud. Now help Patrick establish the PDF Cloud Run service for Pet Theory.
  + Open a new Cloud Shell session and run the following command to clone the Pet Theory repository:
    - git clone https://github.com/rosera/pet-theory.git
  + Then change your current working directory to lab03:
    - cd pet-theory/lab03
  + Edit package.json
    - ...
    - "scripts": {
    - "start": "node index.js",
    - "test": "echo \"Error: no test specified\" && exit 1"
    - },
    - ...
  + Now run the following cmds to install the packages that your conversion script will be using:
    - npm install express
    - npm install body-parser
    - npm install child\_process
    - npm install @google-cloud/storage
  + Now open the lab03/index.js file and review the code.
    - The application will be deployed as a Cloud Run service that accepts HTTP POSTs. If the POST request is a Pub/Sub notification about an uploaded file, the service writes the file details to the log. If not, the service simply returns the string "OK".
  + Review the file named lab03/Dockerfile.
    - The above file is called a manifest and provides a recipe for the Docker command to build an image. Each line begins with a command that tells Docker how to process the following information:
    - The first list indicates the base image should use node v12 as the template for the image to be created.
    - The last line indicates the command to be performed, which in this instance refers to "npm start".
  + To build and deploy the REST API, use Google Cloud Build. Run this command to start the build process:
    - gcloud builds submit \
    - --tag gcr.io/$GOOGLE\_CLOUD\_PROJECT/pdf-converter
  + The command builds a container with your code and puts it in the Container Registry of your project.
  + navigation menu, and select Container Registry > Images. You should see your container hosted:
    - Graphical user interface, text, application

      Description automatically generated
  + Return to your code editor tab and in Cloud Shell run the following command to deploy your application:
    - gcloud run deploy pdf-converter \
    - --image gcr.io/$GOOGLE\_CLOUD\_PROJECT/pdf-converter \
    - --platform managed \
    - --region us-central1 \
    - --no-allow-unauthenticated \
    - --max-instances=1
  + When the deployment is complete, you will see a message like this:
    - Service [pdf-converter] revision [pdf-converter-00001] has been deployed and is serving 100 percent of traffic at https://pdf-converter-[hash].a.run.app
  + Create the environment variable $SERVICE\_URL for the app so you can easily access it:
    - SERVICE\_URL=$(gcloud beta run services describe pdf-converter --platform managed --region us-central1 --format="value(status.url)")
    - echo $SERVICE\_URL
  + Make an anonymous POST request to your new service:
    - curl -X POST $SERVICE\_URL
  + This will result in an error message saying "Your client does not have permission to get the URL". This is good; you don't want the service to be callable by anonymous users.
  + Now try invoking the service as an authorized user:
    - curl -X POST -H "Authorization: Bearer $(gcloud auth print-identity-token)" $SERVICE\_URL
  + If you get the response "OK" you have successfully deployed a Cloud Run service. Well done!
* Trigger your Cloud Run service when a new file is uploaded
  + Now that the Cloud Run service has been successfully deployed, Ruby would like Patrick to create a staging area for the data to be converted. The Cloud Storage bucket will use an event trigger to notify the application when a file has been uploaded and needs to be processed.
  + Run the following command to create a bucket in Cloud Storage for the uploaded docs:
    - gsutil mb gs://$GOOGLE\_CLOUD\_PROJECT-upload
  + And another bucker for the processed PDFs:
    - gsutil mb gs://$GOOGLE\_CLOUD\_PROJECT-processed
  + Now return to your Cloud Console tab, open the Navigation menu and select Cloud Storage. Verify that the buckets have been created (there will be other buckets there as well that are used by the platform.)
  + In Cloud Shell run the following command to tell Cloud Storage to send a Pub/Sub notification whenever a new file has finished uploading to the docs bucket:
    - gsutil notification create -t new-doc -f json -e OBJECT\_FINALIZE gs://$GOOGLE\_CLOUD\_PROJECT-upload
  + The notifications will be labeled with the topic "new-doc".
  + Then create a new service account which Pub/Sub will use to trigger the Cloud Run services:
    - gcloud iam service-accounts create pubsub-cloud-run-invoker --display-name "PubSub Cloud Run Invoker"
  + Give the new service account permission to invoke the PDF converter service:
    - gcloud beta run services add-iam-policy-binding pdf-converter --member=serviceAccount:pubsub-cloud-run-invoker@$GOOGLE\_CLOUD\_PROJECT.iam.gserviceaccount.com --role=roles/run.invoker --platform managed --region us-central1
  + Find your project number by running this command:
    - gcloud projects list
  + Look for the project whose name starts with "qwiklabs-gcp-". You will be using the value of the Project Number in the next command.
  + Create a PROJECT\_NUMBER environment variable, replacing [project number] with the Project Number from the last command:
    - PROJECT\_NUMBER=[project number]
  + Then enable your project to create Cloud Pub/Sub authentication tokens:
    - gcloud projects add-iam-policy-binding $GOOGLE\_CLOUD\_PROJECT --member=serviceAccount:service-$PROJECT\_NUMBER@gcp-sa-pubsub.iam.gserviceaccount.com --role=roles/iam.serviceAccountTokenCreator
  + Finally, create a Pub/Sub subscription so that the PDF converter can run whenever a message is published on the topic "new-doc".
    - gcloud beta pubsub subscriptions create pdf-conv-sub --topic new-doc --push-endpoint=$SERVICE\_URL --push-auth-service-account=pubsub-cloud-run-invoker@$GOOGLE\_CLOUD\_PROJECT.iam.gserviceaccount.com
* See if the Cloud Run service is triggered when files are uploaded to Cloud Storage
  + To verify the application is working as expected, Ruby asks Patrick to upload some test data to the named storage bucket and then check Cloud Logging.
  + Copy some test files into your upload bucket:
    - gsutil -m cp gs://spls/gsp644/\* gs://$GOOGLE\_CLOUD\_PROJECT-upload
  + Once the upload is done, return to your Cloud Console tab, open the navigation menu, and select Logging from under the Operations section.
  + In the first dropdown, filter your results to Cloud Run Revision and click Add. Then click Run Query.
  + In the Query results, look for a log entry that starts with file: and click it. It shows a dump of the file data that Pub/Sub sends to your Cloud Run service when a new file is uploaded.
  + Can you find the name of the file you uploaded in this object?
    - Graphical user interface, text, application, email

      Description automatically generated
  + Note: If you do not see any log entries that begin with "file", try clicking on the "load newer logs" button near the bottom of the page.
  + Now return to the code editor tab and run the following command in Cloud Shell to clean up your upload directory by deleting the files in it:
    - gsutil -m rm gs://$GOOGLE\_CLOUD\_PROJECT-upload/\*
* Docker containers
  + Patrick needs to convert a backlog of invoices to PDFs so all customers can open them. He emails Ruby for some help...
  + Patrick, IT Administrator
    - Hi Ruby
      * Based on your findings, I think we can automate this process and also move to using PDF as the invoice format.
      * I spent a bit of time yesterday coding a solution and built a Node.js script to do what we need. Could you take a look?
    - Patrick
  + Patrick sends Ruby the code fragment he wrote to produce a PDF from a file:
    - const {promisify} = require('util');
    - const exec = promisify(require('child\_process').exec);
    - const cmd = 'libreoffice --headless --convert-to pdf --outdir ' +
    - `/tmp "/tmp/${fileName}"`;
    - const { stdout, stderr } = await exec(cmd);
    - if (stderr) {
    - throw stderr;
    - }
  + Ruby responds back to Patrick...
  + Ruby, Software Consultant
    - Hi Patrick
      * Cloud Run uses containers, so we need to provide your application in this format. For the next step we need to create a Dockerfile manifest for the application.
      * Your code uses LibreOffice. Can you send me the command for installing that software? I will need to include it in the container.
    - Ruby
  + Patrick, IT Administrator
    - Hi Ruby
      * Awesome, here is how I usually install LibreOffice on servers in the office:
        + apt-get update -y && apt-get install -y libreoffice && apt-get clean
      * Let me know if you need any more information.
    - Patrick
  + Building the container will require the integration of a number of components:
    - Diagram

      Description automatically generated with medium confidence
  + Update the Docker container
    - With all the files identified, the Dockerfile can now be created. Help Ruby set up and deploy the container.
    - The package for LibreOffice was not included in the container before, which means it now needs to be added. Patrick has previously provided the commands he uses to build his application, Ruby will add these as a RUN command within the Dockerfile.
    - Open the Dockerfile manifest and add the command RUN apt-get update -y && apt-get install -y libreoffice && apt-get clean line as shown below:
      * FROM node:12
      * RUN apt-get update -y \
      * && apt-get install -y libreoffice \
      * && apt-get clean
      * WORKDIR /usr/src/app
      * COPY package.json package\*.json ./
      * RUN npm install --only=production
      * COPY . .
      * CMD [ "npm", "start" ]
  + Deploy the new version of the pdf-conversion service
  + Ensure your index.js file looks like the following:
* const {promisify} = require('util');
* const {Storage} = require('@google-cloud/storage');
* const exec = promisify(require('child\_process').exec);
* const storage = new Storage();
* const express = require('express');
* const bodyParser = require('body-parser');
* const app = express();
* app.use(bodyParser.json());
* const port = process.env.PORT || 8080;
* app.listen(port, () => {
* console.log('Listening on port', port);
* });
* app.post('/', async (req, res) => {
* try {
* const file = decodeBase64Json(req.body.message.data);
* await downloadFile(file.bucket, file.name);
* const pdfFileName = await convertFile(file.name);
* await uploadFile(process.env.PDF\_BUCKET, pdfFileName);
* await deleteFile(file.bucket, file.name);
* }
* catch (ex) {
* console.log(`Error: ${ex}`);
* }
* res.set('Content-Type', 'text/plain');
* res.send('\n\nOK\n\n');
* })
* function decodeBase64Json(data) {
* return JSON.parse(Buffer.from(data, 'base64').toString());
* }
* async function downloadFile(bucketName, fileName) {
* const options = {destination: `/tmp/${fileName}`};
* await storage.bucket(bucketName).file(fileName).download(options);
* }
* async function convertFile(fileName) {
* const cmd = 'libreoffice --headless --convert-to pdf --outdir /tmp ' +
* `"/tmp/${fileName}"`;
* console.log(cmd);
* const { stdout, stderr } = await exec(cmd);
* if (stderr) {
* throw stderr;
* }
* console.log(stdout);
* pdfFileName = fileName.replace(/\.\w+$/, '.pdf');
* return pdfFileName;
* }
* async function deleteFile(bucketName, fileName) {
* await storage.bucket(bucketName).file(fileName).delete();
* }
* async function uploadFile(bucketName, fileName) {
* await storage.bucket(bucketName).upload(`/tmp/${fileName}`);
* }
  + The main logic is housed in these functions:
    - const file = decodeBase64Json(req.body.message.data);
    - await downloadFile(file.bucket, file.name);
    - const pdfFileName = await convertFile(file.name);
    - await uploadFile(process.env.PDF\_BUCKET, pdfFileName);
    - await deleteFile(file.bucket, file.name);
  + Whenever a file has been uploaded, this service gets triggered. It performs these tasks, one per line above:
    - Extracts the file details from the Pub/Sub notification.
    - Downloads the file from Cloud Storage to the local hard drive. This is actually not a physical disk, but a section of virtual memory that behaves like a disk.
    - Converts the downloaded file to PDF.
    - Uploads the PDF file to Cloud Storage. The environment variable process.env.PDF\_BUCKET contains the name of the Cloud Storage bucket to write PDFs to. You will assign a value to this variable when you deploy the service below.
    - removes the original file from Cloud Storage.
  + The rest of index.js implements the functions called by this top-level code.
  + It's time to deploy the service, and to set the PDF\_BUCKET environment variable. It's also a good idea to give LibreOffice 2 GB of RAM to work with (see the line with the --memory option).
  + Run the following command to build the container:
    - gcloud builds submit \
    - --tag gcr.io/$GOOGLE\_CLOUD\_PROJECT/pdf-converter
  + Now deploy the latest version of your application:
    - gcloud run deploy pdf-converter \
    - --image gcr.io/$GOOGLE\_CLOUD\_PROJECT/pdf-converter \
    - --platform managed \
    - --region us-central1 \
    - --memory=2Gi \
    - --no-allow-unauthenticated \
    - --max-instances=1 \
    - --set-env-vars PDF\_BUCKET=$GOOGLE\_CLOUD\_PROJECT-processed
  + With LibreOffice part of the container, this build will take longer than the previous one. This is a good time to get up and stretch for a few minutes.
* Testing the pdf-conversion service
  + Once the deployment commands finish, make sure that the service was deployed correctly by running:
    - curl -X POST -H "Authorization: Bearer $(gcloud auth print-identity-token)" $SERVICE\_URL
  + If you get the response "OK" you have successfully deployed the updated Cloud Run service. LibreOffice can convert many file types to PDF: DOCX, XLSX, JPG, PNG, GIF, etc.
  + Run the following command to upload some example files:
    - gsutil -m cp gs://spls/gsp644/\* gs://$GOOGLE\_CLOUD\_PROJECT-upload
  + Navigation > Cloud Storage > Open the -upload bucket, Refresh button a couple of times to see how the files are deleted, one by one, as they are converted to PDFs.
  + Then click on the bucket ends in "-processed". It should contain PDF versions of all files.
  + 

## 8.2 Build a Resilient, Asynchronous System with Cloud Run and Pub/Sub

* Overview
  + For the labs in the Google Cloud Serverless Workshop: Pet Theory Quest, you will read through a fictitious business scenario and assist the characters with their serverless migration plan.
  + Twelve years ago, Lily started the Pet Theory chain of veterinary clinics. Over the years, the number of clinics has grown, and so has the need for automation. The way Pet Theory handles the results of medical tests when they come back from the lab is too slow and error-prone, and Lily wants to improve this.
  + Currently, Patrick, Pet Theory's IT administrator, handles test results manually. Whenever a test result comes back, he composes and sends an email to the client whose pet was tested, then he taps out a text message on his phone and sends the results as a text to the client.
  + Patrick is working with Ruby, a software consultant, to design a more scalable system. They want to build a solution that doesn't require a lot of ongoing maintenance. Patrick and Ruby have decided to go with serverless technology.
* Requirements
  + Pet Theory would like to automate the process of sharing client test results. They have experienced a tough time keeping up with an increased volume of appointments, so Lily decides to ask Ruby for some assistance...
  + Lily, Founder of Pet Theory
    - Hi Ruby,
    - Thanks for sorting out the insurance portal.
    - I was wondering if something could be done about the medical test results? We need a more efficient way of sending results to our clients.
    - Lily
  + Ruby, Software Consultant
    - Hey Lily,
    - Sure - let me see what I can do. I have a few ideas that may improve the situation.
    - Ruby
* Architecture
  + Pet Theory uses an external company for medical tests. Once the lab company completes a medical test, they send the results back to Pet Theory.
  + The lab company use a HTTP(s) POST to Pet Theory's web endpoint for medical lab results. The illustration below outlines the general architecture.
    - Diagram

      Description automatically generated
  + After looking at the general process followed, Ruby believes that a system can be designed in which Pet Theory is able to:
    - Receive the HTTP POST request and confirm receipt to the medical lab.
    - Email the test result to the client.
    - Send a text message (SMS) and an email to the client with the test result.
  + Ruby's design isolates each of the above activities and requires:
    - A service to perform the request and response for the medical result(s)
    - A service to email test results to the client
    - A service to send a text message (SMS) to the client
    - Pub/Sub to be used for inter-service communication
    - Serverless infrastructure to be used for the application architecture
  + Through the use of single use functions, Ruby is looking to develop code that is easier to write and contains fewer bugs.
  + Ruby, Software Consultant
    - Hi Patrick,
    - Lily would like me to build a prototype to help with the processing of medical records.
    - To get started, could you set up a Pub/Sub Topic called new-lab-report.
      * Ruby
  + Patrick, IT Administrator
    - Hey Ruby,
    - That sounds like a cool project. I can get that finished for you this morning, both activities are really quick to setup on Google Cloud.
    - Patrick
  + Create a Pub/Sub topic
    - Help Patrick to create a Pub/Sub topic called new-lab-report.
      * Diagram

        Description automatically generated
    - When a service publishes a Pub/Sub message, that message must be tagged with a topic. The Lab Report is consumed via the service to be created and publish a message for each report found.
    - First you need to create a topic that can be used for this task.
    - Run the following command to create a Pub/Sub topic:
      * gcloud pubsub topics create new-lab-report
    - Any service subscribed to the topic "new-lab-report" will be able to consume the message published by the Lab Report Service. In the above diagram you can see two such consumers, Email Service and SMS Service.
    - Then enable Cloud Run, which will run your code in the cloud:
      * gcloud services enable run.googleapis.com
    - Don't forget to update Ruby to let her know that the Pub/Sub topic is ready for her!
    - Patrick, IT Administrator
      * Hey Ruby,
      * All done.
      * If you have time, I would like to see how this prototype is put together. Could we work on this together?
      * Patrick
    - Ruby, Software Consultant
      * Hi Patrick,
      * That's great, thanks for getting to this so quickly. I'll set up a time and we'll start building.
      * Ruby
* Build the Lab Report Service
  + Help Ruby to set up the new Lab Report Service.
    - Diagram

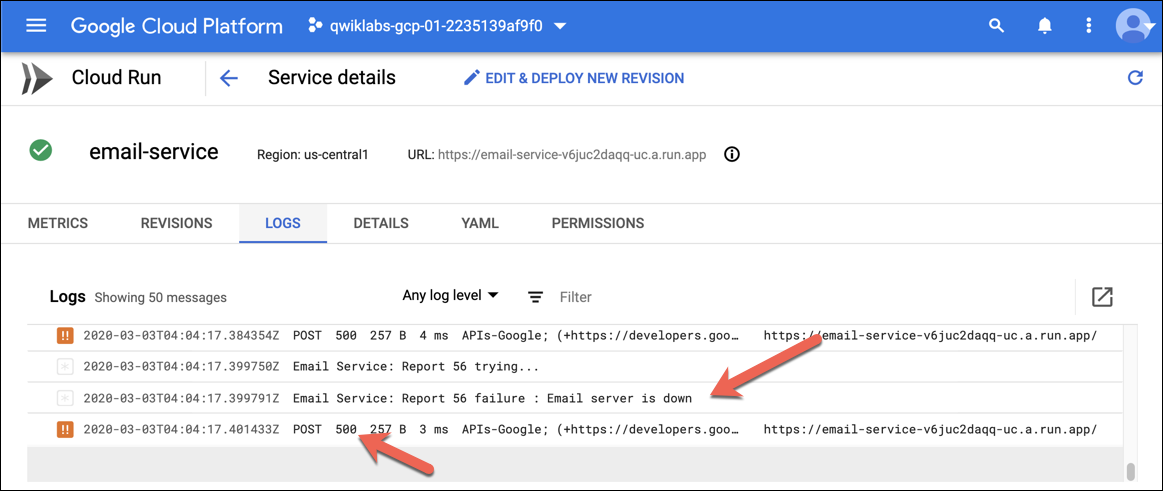
      Description automatically generated
  + This service will serve the purpose of prototyping, so it will only do two things:
    - Receive the lab report HTTPS POST containing the report data.
    - Publish a message on Pub/Sub.
* Add code for the Lab Report Service
  + Back in Cloud Shell, clone the repository needed for this lab:
    - git clone https://github.com/rosera/pet-theory.git
    - cd pet-theory/lab05/lab-service
    - npm install express
    - npm install body-parser
    - npm install @google-cloud/pubsub
  + Open the package.json file.
    - "scripts": {
    - "start": "node index.js",
    - "test": "echo \"Error: no test specified\" && exit 1"
    - },
  + Create a new file named index.js and add this code to it:
    - const {PubSub} = require('@google-cloud/pubsub');
    - const pubsub = new PubSub();
    - const express = require('express');
    - const app = express();
    - const bodyParser = require('body-parser');
    - app.use(bodyParser.json());
    - const port = process.env.PORT || 8080;
    - app.listen(port, () => {
    - console.log('Listening on port', port);
    - });
    - app.post('/', async (req, res) => {
    - try {
    - const labReport = req.body;
    - await publishPubSubMessage(labReport);
    - res.status(204).send();
    - }
    - catch (ex) {
    - console.log(ex);
    - res.status(500).send(ex);
    - }
    - })
    - async function publishPubSubMessage(labReport) {
    - const buffer = Buffer.from(JSON.stringify(labReport));
    - await pubsub.topic('new-lab-report').publish(buffer);
    - }
  + The heart of the code is this section:
    - const labReport = req.body;
    - await publishPubSubMessage(labReport);
  + These two lines do the main work of the service:
    - Extract the lab report from the POST request.
    - Publish a PubSub message containing the newly posted lab report.
  + Now create a file named Dockerfile and add the code below into it:
    - FROM node:10
    - WORKDIR /usr/src/app
    - COPY package.json package\*.json ./
    - RUN npm install --only=production
    - COPY . .
    - CMD [ "npm", "start" ]
  + This file defines how to package up the Cloud Run service into a container.
  + Deploy the lab-report-service
    - Create a script named deploy.sh and paste these commands into it:
      * gcloud builds submit \
      * --tag gcr.io/$GOOGLE\_CLOUD\_PROJECT/lab-report-service
      * gcloud run deploy lab-report-service \
      * --image gcr.io/$GOOGLE\_CLOUD\_PROJECT/lab-report-service \
      * --platform managed \
      * --region us-central1 \
      * --allow-unauthenticated \
      * --max-instances=1
    - Run the following to make this file executable:
      * chmod u+x deploy.sh
      * ./deploy.sh
    - Due to timing issues, you may get an error the first time you run this command. If you do, simply rerun deploy.sh.
    - When the deployment has successfully completed, you will see a message similar to this:
      * Service [lab-report-service] revision [lab-report-service-00001] has been deployed and is serving traffic at https://lab-report-service-[hash].a.run.app
    - Nice work, the Lab Report Service has been deployed and will consume medical lab results over HTTP. You can now test if the new service is up and running.
  + Test the Lab Report Service
    - To validate the Lab Report Service, simulate three HTTPS POSTs made by the lab company, each containing one lab report. For the purpose of testing, the lab reports created will only contain an ID.
    - First, put the URL to the report in an environment variable, to make it easier to work with.
      * export LAB\_REPORT\_SERVICE\_URL=$(gcloud run services describe lab-report-service --platform managed --region us-central1 --format="value(status.address.url)")
    - Confirm the LAB\_REPORT\_SERVICE\_URL has been captured:
      * echo $LAB\_REPORT\_SERVICE\_URL
    - Create a new file named post-reports.sh and add the code below into it:
      * curl -X POST \
      * -H "Content-Type: application/json" \
      * -d "{\"id\": 12}" \
      * $LAB\_REPORT\_SERVICE\_URL &
      * curl -X POST \
      * -H "Content-Type: application/json" \
      * -d "{\"id\": 34}" \
      * $LAB\_REPORT\_SERVICE\_URL &
      * curl -X POST \
      * -H "Content-Type: application/json" \
      * -d "{\"id\": 56}" \
      * $LAB\_REPORT\_SERVICE\_URL &
    - The above script will use the curl command to post three distinct ID's to the Lab Service URL. Each command will be run individually in the background.
    - Make the post-reports.sh script executable:
      * chmod u+x post-reports.sh
      * ./post-reports.sh
    - This script posted three lab reports to your Lab Report Service. Check the logs to see the results!
    - From the Navigation menu click Cloud Run > newly deployed lab-report-service. Click it > Logs.
    - On the Logs page are the results of the three test reports that you just posted with the script. Hopefully the returned HTTP codes are 204, meaning OK - not content, shown below. If you don’t see any entries, try scrolling up and down using the scrollbar to the right. This reloads the log.
      * Graphical user interface, text, application, email

        Description automatically generated
    - The next task is to write the SMS and Email services. These services will be triggered when the Lab Report Service publishes a Pub/Sub message on the "new-lab-report" topic.
* The Email Service
  + Help Ruby to set up the new Email Service.
    - Diagram

      Description automatically generated
  + Add code for the Email Service
    - cd ~/pet-theory/lab05/email-service
    - npm install express
    - npm install body-parser
  + Open the package.json file.
    - "scripts": {
    - "start": "node index.js",
    - "test": "echo \"Error: no test specified\" && exit 1"
    - },
  + Create a new file called index.js and add the following to it:
    - const express = require('express');
    - const app = express();
    - const bodyParser = require('body-parser');
    - app.use(bodyParser.json());
    - const port = process.env.PORT || 8080;
    - app.listen(port, () => {
    - console.log('Listening on port', port);
    - });
    - app.post('/', async (req, res) => {
    - const labReport = decodeBase64Json(req.body.message.data);
    - try {
    - console.log(`Email Service: Report ${labReport.id} trying...`);
    - sendEmail();
    - console.log(`Email Service: Report ${labReport.id} success :-)`);
    - res.status(204).send();
    - }
    - catch (ex) {
    - console.log(`Email Service: Report ${labReport.id} failure: ${ex}`);
    - res.status(500).send();
    - }
    - })
    - function decodeBase64Json(data) {
    - return JSON.parse(Buffer.from(data, 'base64').toString());
    - }
    - function sendEmail() {
    - console.log('Sending email');
    - }
  + This code will run when Pub/Sub posts a message to the service. This is what it does:
    - It decodes the Pub/Sub message and then tries to call the sendEmail() function.
    - If that succeeds and no exception is thrown, it will return status code 204 so Pub/Sub knows that the message was processed.
    - If there is an exception, the service will return status code 500 so that Pub/Sub knows the message was not processed and it should re-post it to the service later.
  + Once the communication between services is working, Ruby will add code to the sendEmail() function to actually send the email.
  + Now create a file named Dockerfile and add the code below into it:
    - FROM node:10
    - WORKDIR /usr/src/app
    - COPY package.json package\*.json ./
    - RUN npm install --only=production
    - COPY . .
    - CMD [ "npm", "start" ]
  + This file defines how to package up the Cloud Run service into a container.
  + Deploy the Email Service
    - Create a new file called deploy.sh and add the following to it:
      * gcloud builds submit \
      * --tag gcr.io/$GOOGLE\_CLOUD\_PROJECT/email-service
      * gcloud run deploy email-service \
      * --image gcr.io/$GOOGLE\_CLOUD\_PROJECT/email-service \
      * --platform managed \
      * --region us-central1 \
      * --no-allow-unauthenticated \
      * --max-instances=1
    - Make deploy.sh executable:
      * chmod u+x deploy.sh
      * ./deploy.sh
    - When the deployment is complete, you will see a message similar to this:
      * Service [email-service] revision [email-service-00001] has been deployed and is serving traffic at https://email-service-[hash].a.run.app
    - The service has been successfully deployed. You now need to ensure the Email Service is triggered when a Pub/Sub message is available.
  + Configure Pub/Sub to trigger the Email Service
    - Whenever a new Pub/Sub message is published using the "new-lab-report" topic, it should trigger the Email Service. To achieve this task, configure a service account to automatically handle the associated requests for this service.
      * Diagram

        Description automatically generated
    - Create a new service account that will be used to trigger the services responding to Pub/Sub messages:
      * gcloud iam service-accounts create pubsub-cloud-run-invoker --display-name "PubSub Cloud Run Invoker"
    - Give the new service account permission to invoke the Email Service:
      * gcloud run services add-iam-policy-binding email-service --member=serviceAccount:pubsub-cloud-run-invoker@$GOOGLE\_CLOUD\_PROJECT.iam.gserviceaccount.com --role=roles/run.invoker --region us-central1 --platform managed
    - Next, tell Pub/Sub to invoke the SMS Service when a "new-lab-report" message is published.
    - Put the project number in an environment variable for easy access:
      * PROJECT\_NUMBER=$(gcloud projects list --filter="qwiklabs-gcp" --format='value(PROJECT\_NUMBER)')
    - Next, enable the project to create Pub/Sub authentication tokens. Run the code below:
      * gcloud projects add-iam-policy-binding $GOOGLE\_CLOUD\_PROJECT --member=serviceAccount:service-$PROJECT\_NUMBER@gcp-sa-pubsub.iam.gserviceaccount.com --role=roles/iam.serviceAccountTokenCreator
    - Put the URL of the Email Service in another environment variable:
      * EMAIL\_SERVICE\_URL=$(gcloud run services describe email-service --platform managed --region us-central1 --format="value(status.address.url)")
    - Confirm the EMAIL\_SERVICE\_URL has been captured:
      * echo $EMAIL\_SERVICE\_URL
    - Create a Pub/Sub subscription for the Email Service.
      * gcloud pubsub subscriptions create email-service-sub --topic new-lab-report --push-endpoint=$EMAIL\_SERVICE\_URL --push-auth-service-account=pubsub-cloud-run-invoker@$GOOGLE\_CLOUD\_PROJECT.iam.gserviceaccount.com
    - Nice work, the service is now set up to respond to Cloud Pub/Sub messages, as a next step validate the code to confirm it meets requirements.
  + Test the Lab Report Service and the Email Service together
    - Using the script created earlier, post to the lab reports again:
      * ~/pet-theory/lab05/lab-service/post-reports.sh
    - (Navigation menu > Cloud Run). You will see the two Cloud Run services in your account.
    - Click email-service and then click Logs. You will see the result of this service being triggered by Pub/Sub. If you don’t see the messages you expect, you may need to scroll up and down with the scrollbar to get the log to refresh.
    - Great job! The Email service is now able to write information to the log whenever a message is processed from the Cloud Pub/Sub topic queue! The last task is to write the SMS Service.
* The SMS Service
  + Help Ruby to set up the new SMS Service.
    - Diagram

      Description automatically generated
  + Add code for the SMS Service
    - cd ~/pet-theory/lab05/sms-service
    - npm install express
    - npm install body-parser
  + Open the package.json file.
    - "scripts": {
    - "start": "node index.js",
    - "test": "echo \"Error: no test specified\" && exit 1"
    - },
  + Create a new file called index.js and add the following to it:
    - const express = require('express');
    - const app = express();
    - const bodyParser = require('body-parser');
    - app.use(bodyParser.json());
    - const port = process.env.PORT || 8080;
    - app.listen(port, () => {
    - console.log('Listening on port', port);
    - });
    - app.post('/', async (req, res) => {
    - const labReport = decodeBase64Json(req.body.message.data);
    - try {
    - console.log(`SMS Service: Report ${labReport.id} trying...`);
    - sendSms();
    - console.log(`SMS Service: Report ${labReport.id} success :-)`);
    - res.status(204).send();
    - }
    - catch (ex) {
    - console.log(`SMS Service: Report ${labReport.id} failure: ${ex}`);
    - res.status(500).send();
    - }
    - })
    - function decodeBase64Json(data) {
    - return JSON.parse(Buffer.from(data, 'base64').toString());
    - }
    - function sendSms() {
    - console.log('Sending SMS');
    - }
  + Now create a file named Dockerfile and add the code below into it:
    - FROM node:10
    - WORKDIR /usr/src/app
    - COPY package.json package\*.json ./
    - RUN npm install --only=production
    - COPY . .
    - CMD [ "npm", "start" ]
  + This file defines how to package up the Cloud Run service into a container. Now the code has been created, the next step is to deploy the service.
  + Deploy the SMS Service
    - Create a file named deploy.sh and add this code into it:
      * gcloud builds submit \
      * --tag gcr.io/$GOOGLE\_CLOUD\_PROJECT/sms-service
      * gcloud run deploy sms-service \
      * --image gcr.io/$GOOGLE\_CLOUD\_PROJECT/sms-service \
      * --platform managed \
      * --region us-central1 \
      * --no-allow-unauthenticated \
      * --max-instances=1
    - Make deploy.sh executable:
      * chmod u+x deploy.sh
      * ./deploy.sh
    - When the deployment is complete, a message similar to this is displayed:
      * Service [sms-service] revision [sms-service-00001] has been deployed and is serving traffic at https://sms-service-[hash].a.run.app
    - The SMS Service is successfully deployed, but it isn't linked to the Cloud Pub/Sub service. Correct that in the next section.
  + Configure Cloud Pub/Sub to trigger the SMS Service
    - As with the Email Service, the link between Cloud Pub/Sub and the SMS service needs to be configured so that messages can be consumed.
      * Diagram

        Description automatically generated
    - Set the permissions to allow Pub/Sub to trigger the SMS Service:
      * gcloud run services add-iam-policy-binding sms-service --member=serviceAccount:pubsub-cloud-run-invoker@$GOOGLE\_CLOUD\_PROJECT.iam.gserviceaccount.com --role=roles/run.invoker --region us-central1 --platform managed
    - Next, tell Pub/Sub to invoke the SMS Service when a “new-lab-report” message is published.
    - The first step is to put the URL address of the SMS Service in an environment variable:
      * SMS\_SERVICE\_URL=$(gcloud run services describe sms-service --platform managed --region us-central1 --format="value(status.address.url)")
    - Confirm the SMS\_SERVICE\_URL has been captured:
      * echo $SMS\_SERVICE\_URL
    - Then create the Pub/Sub subscription:
      * gcloud pubsub subscriptions create sms-service-sub --topic new-lab-report --push-endpoint=$SMS\_SERVICE\_URL --push-auth-service-account=pubsub-cloud-run-invoker@$GOOGLE\_CLOUD\_PROJECT.iam.gserviceaccount.com
    - Run the test script again to post three lab reports to the Lab Report Service:
      * ~/pet-theory/lab05/lab-service/post-reports.sh
    - Click sms-service > Logs. You will see the result of this service being triggered by Pub/Sub.
    - The prototype system has been created and successfully tested. However, Patrick is concerned that resilience, as part of the initial validation process, hasn't been tested.
* Test the resiliency of the system
  + What happens if one of the services goes down? Patrick has run into this before, as it is a common situation.
  + Help Ruby investigate how to ensure the system can handle this scenario. She wants to test what happens when a service fails by deploying a bad version of the Email Service.
  + Go back to the email-service directory:
    - cd ~/pet-theory/lab05/email-service
  + Add some invalid text to the Email Service application to cause an error.
  + Edit index.js, This will throw an exception, as if the email server was down:
    - function sendEmail() {
    - throw 'Email server is down';
    - console.log('Sending email');
    - }
  + The addition of this code will crash the service when it is invoked.
  + Deploy this bad version of the Email Service:
    - ./deploy.sh
  + When the Email Service deployment has successfully completed, post data to the lab reports again, then go and watch the email-service log status closely:
    - ~/pet-theory/lab05/lab-service/post-reports.sh
  + Navigation menu > Cloud Run > email-service.
  + The Email Service is being invoked, but it will keep crashing. If you scroll back a bit in the logs you will find the root cause: “Email server is down”. You can also see that the service returns status code 500, and that Pub/Sub keeps retrying calling the service.
    - 
  + If you look at the logs from the SMS service, you will see that it operates successfully.
  + Now fix the error in the Email Service to restore the application!
  + Takeaways
    - If services communicate asynchronously with each other via Pub/Sub instead of calling each other directly, the system can be more resilient.
    - The Lab Report Service trigger is independent of other services, thanks to the use of Pub/Sub. For example, if customers should also want to receive lab results via another messaging service, it can be added without needing to update the Lab Report Service.
    - Cloud Pub/Sub handled the retries, the services didn't have to. Services are only required to return a status code: success or failure.
    - If a service goes down, the system is capable of automatically "healing" itself when the service comes back online, thanks to Pub/Sub retries.

## 8.3 Developing a REST API with Go and Cloud Run

* Overview
  + In previous labs in this series, moved Pet Theory's customer database to a serverless Firestore database in the cloud, and then opened up access so customers can make appointments online. Since Pet Theory's Ops team is a single person, they need a serverless solution that doesn't require a lot of ongoing maintenance.
  + In this lab, you'll help Ruby and Patrick to give insurance companies access to customer data without exposing Personal Identifiable Information (PII). You will build a secure Representational State Transfer (REST) API gateway using Cloud Run, which is serverless. This will let the insurance companies see the total cost of treatments without seeing customers' PII.
* Requirements
  + Lily, Founder of Pet Theory
    - Hi Ruby,
    - Remember our conversation last week when I expressed how swamped I am with paperwork and phone calls from the insurance company? If only there was a way to allow the representatives to access customer records in an efficient, secure way.
    - This current level of workload isn't sustainable. Can you help?
    - Lily
  + Ruby, Software Consultant
    - Hi Lily,
    - Yesterday I had lunch with Patrick and we drew up a plan to make it easier for authorized 3rd parties to securely access Pet Theory's digital records.
    - We will build this in four steps:
    - Build a simple REST API.
    - Import customer test data.
    - Connect the REST API to the customer database.
    - Add authentication to the REST API.
    - Patrick and I already have the skillset for steps 1 + 2, so we are off to a good start. We plan to have a working prototype by the end of the week.
    - Ruby
  + Help Ruby manage the activities necessary to build the REST API for Pet Theory.
* Enable Google APIs
  + Cloud Build cloudbuild.googleapis.com
  + Cloud Run run.googleapis.com
* Developing the REST API
  + Activate your project:
    - gcloud config set project $(gcloud projects list --format='value(PROJECT\_ID)' --filter='qwiklabs-gcp')
  + Clone the pet-theory repository and access the source code:
    - git clone https://github.com/rosera/pet-theory.git && cd pet-theory/lab08
  + view the go.mod and go.sum files
  + Create the file main.go and add the below contents to the file:
    - package main
    - import (
    - "fmt"
    - "log"
    - "net/http"
    - "os"
    - )
    - func main() {
    - port := os.Getenv("PORT")
    - if port == "" {
    - port = "8080"
    - }
    - http.HandleFunc("/v1/", func(w http.ResponseWriter, r \*http.Request) {
    - fmt.Fprintf(w, "{status: 'running'}")
    - })
    - log.Println("Pets REST API listening on port", port)
    - if err := http.ListenAndServe(":"+port, nil); err != nil {
    - log.Fatalf("Error launching Pets REST API server: %v", err)
    - }
    - }
  + Now create a file named Dockerfile and add the following to it:
    - FROM gcr.io/distroless/base-debian10
    - WORKDIR /usr/src/app
    - COPY server .
    - CMD [ "/usr/src/app/server" ]
  + The file server is the execution binary built from main.go.
  + Run the following command to build the binary:
    - go build -o server
  + For most Cloud Run Go based apps, a template Dockerfile like the one above can typically be used without modifying it.
  + Deploy your simple REST API by running:
    - gcloud builds submit \
    - --tag gcr.io/$GOOGLE\_CLOUD\_PROJECT/rest-api:0.1
  + Cmd builds a container with your code and puts in the Navigation > Container Registry > rest-api.
  + Once the container has been built, deploy it:
    - gcloud run deploy rest-api \
    - --image gcr.io/$GOOGLE\_CLOUD\_PROJECT/rest-api:0.1 \
    - --platform managed \
    - --region us-central1 \
    - --allow-unauthenticated \
    - --max-instances=2
  + Click on the Service URL, Append /v1/ to see this message: {“status”:”running”}
  + The REST API is up and running. With the prototype service available, in the next section the API will be used to retrieve "customer" information from a Firestore database.
* Import test customer data
  + Ruby, Software Consultant
    - Hey Patrick,
    - Do you still have the pseudo customer data we created a while back? We will need that for testing.
    - Do you remember how to set up a Firestore database and import data?
    - Ruby
  + Patrick, IT Administrator
    - Hi Ruby,
    - Yes, I still have the test data. I will migrate it to Firestore today so you can use it for testing.
    - Patrick
  + Ruby and Patrick have previously created a test database of 10 customers, with some proposed treatments for one customer's cat.
  + Help Patrick configure the Firestore database and import the customer test data. First, enable Firestore in your project.
  + Navigation > Firestore > Select Native Mode button > location "nam5" (United States) multi-region near.
  + Click the Create Database button. Wait for the database to be created.
  + Migrate the import files into a Cloud Storage bucket that has been created for you:
    - gsutil cp -r gs://spls/gsp645/2019-10-06T20:10:37\_43617 gs://$GOOGLE\_CLOUD\_PROJECT-customer
  + Now import this data into Firebase:
    - gcloud beta firestore import gs://$GOOGLE\_CLOUD\_PROJECT-customer/2019-10-06T20:10:37\_43617/
  + In Firestore, click customers under "Root". You should see the imported pet data.
    - Graphical user interface, application, Teams

      Description automatically generated
  + Nice work, the Firestore database has been successfully created and populated with test data!
* Connect the REST API to the Firestore database
  + Ruby, Software Consultant
    - Hi Lily,
    - Just a quick update: Patrick and I have completed the first two tasks on the list.
    - Now I'm moving on to structuring the REST API so it can access the customer data in Firestore.
    - Ruby
  + Lily, Founder of Pet Theory
    - Hi Ruby,
    - Great work, Ruby! Looking forward to seeing the next stage in action.
    - Lily
  + In this section you'll help Ruby create another end-point in the REST API that will look like this:
    - https://rest-api-[hash].a.run.app/v1/customer/22530
  + For example, that URL should return the total amounts for all proposed, accepted, and rejected treatments for the customer with id 22530, if they exist in the Firestore database:
    - {
    - "status": "success",
    - "data": {
    - "proposed": 1602,
    - "approved": 585,
    - "rejected": 489
    - }
    - }
  + If the customer doesn't exist in the database, status code 404 (not found) and an error message should be returned instead.
  + This new functionality requires a package to access the Firestore database and another one to handle cross-origin resource sharing (CORS).
  + Get the value of the $GOOGLE\_CLOUD\_PROJECT environment variable
    - echo $GOOGLE\_CLOUD\_PROJECT
  + Open the existing main.go file in the pet-theory/lab08 directory.
  + Replace the content of the file with the code below, and then replace the PROJECT\_ID:
    - package main
    - import (
    - "context"
    - "encoding/json"
    - "fmt"
    - "log"
    - "net/http"
    - "os"
    - "cloud.google.com/go/firestore"
    - "github.com/gorilla/handlers"
    - "github.com/gorilla/mux"
    - "google.golang.org/api/iterator"
    - )
    - var client \*firestore.Client
    - func main() {
    - var err error
    - ctx := context.Background()
    - client, err = firestore.NewClient(ctx, "PROJECT\_ID")
    - if err != nil {
    - log.Fatalf("Error initializing Cloud Firestore client: %v", err)
    - }
    - port := os.Getenv("PORT")
    - if port == "" {
    - port = "8080"
    - }
    - r := mux.NewRouter()
    - r.HandleFunc("/v1/", rootHandler)
    - r.HandleFunc("/v1/customer/{id}", customerHandler)
    - log.Println("Pets REST API listening on port", port)
    - cors := handlers.CORS(
    - handlers.AllowedHeaders([]string{"X-Requested-With", "Authorization", "Origin"}),
    - handlers.AllowedOrigins([]string{"https://storage.googleapis.com"}),
    - handlers.AllowedMethods([]string{"GET", "HEAD", "POST", "OPTIONS", "PATCH", "CONNECT"}),
    - )
    - if err := http.ListenAndServe(":"+port, cors(r)); err != nil {
    - log.Fatalf("Error launching Pets REST API server: %v", err)
    - }
    - }
  + Add handler support at the bottom of the file:
    - func rootHandler(w http.ResponseWriter, r \*http.Request) {
    - fmt.Fprintf(w, "{status: 'running'}")
    - }
    - func customerHandler(w http.ResponseWriter, r \*http.Request) {
    - id := mux.Vars(r)["id"]
    - ctx := context.Background()
    - customer, err := getCustomer(ctx, id)
    - if err != nil {
    - w.WriteHeader(http.StatusInternalServerError)
    - fmt.Fprintf(w, `{"status": "fail", "data": '%s'}`, err)
    - return
    - }
    - if customer == nil {
    - w.WriteHeader(http.StatusNotFound)
    - msg := fmt.Sprintf("`Customer \"%s\" not found`", id)
    - fmt.Fprintf(w, fmt.Sprintf(`{"status": "fail", "data": {"title": %s}}`, msg))
    - return
    - }
    - amount, err := getAmounts(ctx, customer)
    - if err != nil {
    - w.WriteHeader(http.StatusInternalServerError)
    - fmt.Fprintf(w, `{"status": "fail", "data": "Unable to fetch amounts: %s"}`, err)
    - return
    - }
    - data, err := json.Marshal(amount)
    - if err != nil {
    - w.WriteHeader(http.StatusInternalServerError)
    - fmt.Fprintf(w, `{"status": "fail", "data": "Unable to fetch amounts: %s"}`, err)
    - return
    - }
    - fmt.Fprintf(w, fmt.Sprintf(`{"status": "success", "data": %s}`, data))
    - }
  + Add Customer support to the bottom of the file:
* type Customer struct {
* Email string `firestore:"email"`
* ID string `firestore:"id"`
* Name string `firestore:"name"`
* Phone string `firestore:"phone"`
* }
* func getCustomer(ctx context.Context, id string) (\*Customer, error) {
* query := client.Collection("customers").Where("id", "==", id)
* iter := query.Documents(ctx)
* var c Customer
* for {
* doc, err := iter.Next()
* if err == iterator.Done {
* break
* }
* if err != nil {
* return nil, err
* }
* err = doc.DataTo(&c)
* if err != nil {
* return nil, err
* }
* }
* return &c, nil
* }
* func getAmounts(ctx context.Context, c \*Customer) (map[string]int64, error) {
* if c == nil {
* return map[string]int64{}, fmt.Errorf("Customer should be non-nil: %v", c)
* }
* result := map[string]int64{
* "proposed": 0,
* "approved": 0,
* "rejected": 0,
* }
* query := client.Collection(fmt.Sprintf("customers/%s/treatments", c.Email))
* if query == nil {
* return map[string]int64{}, fmt.Errorf("Query is nil: %v", c)
* }
* iter := query.Documents(ctx)
* for {
* doc, err := iter.Next()
* if err == iterator.Done {
* break
* }
* if err != nil {
* return nil, err
* }
* treatment := doc.Data()
* result[treatment["status"].(string)] += treatment["cost"].(int64)
* }
* return result, nil
* }
* Pop Quiz
  + Which function responds to URLs with the pattern `/v1/customer/`
    - customerHandler
  + Which statement returns success to the client
    - fmt.Fprintf(w, fmt.Sprintf(`{"status": "success", "data": %s}
  + Which functions read from the Firestore database
    - getCustomer and getAmounts
* Deploying a new Revision
  + Rebuild the source code:
    - go build -o server
  + Build a new image for the REST API:
    - gcloud builds submit \
    - --tag gcr.io/$GOOGLE\_CLOUD\_PROJECT/rest-api:0.2
  + Deploy the updated image:
    - gcloud run deploy rest-api \
    - --image gcr.io/$GOOGLE\_CLOUD\_PROJECT/rest-api:0.2 \
    - --platform managed \
    - --region us-central1 \
    - --allow-unauthenticated \
    - --max-instances=2
  + When the deployment is complete, you will see a similar message to before. The URL for your REST API did not change when you deployed the new version:
  + Try /customer/22530 to get this JSON response, listing the sum total of the customer's proposed, approved and rejected treatments:
    - Graphical user interface, text, application

      Description automatically generated
  + Here are some additional client IDs you can put in the URL instead of 22530:
    - 34216
    - 70156 (all amounts should be zero)
    - 12345 (client/pet doesn't exist, should return an error e.g. Query is nil)
  + You have built a scalable, low-maintenance, serverless REST API that reads from a database. Demonstrate how to build a REST API with Go and Cloud Run

## 8.4 Creating PDFs with Go and Cloud Run

* Overview
  + In this lab you will build a PDF converter web app on Cloud Run, which is a serverless service, that automatically converts files stored in Google Drive into PDFs stored in segregated Google Drive folders.
* Objectives
  + Convert a Go application to a container
  + Learn how to build containers with Google Cloud Build
  + Create a Cloud Run service that converts files to PDF files in the cloud.
  + Understand how to create Service Accounts and add permissions
  + Use event processing with Cloud Storage
* Architecture
  + In this lab you will assist the Pet Theory Veterinary practice to automatically convert their invoices into PDFs so that customers can open them reliably.
    - Graphical user interface, application, Teams

      Description automatically generated
* Using Googleapis
  + During this lab you will use Google APIs. The following APIs have been enabled for you:
  + Name API
  + Cloud Build cloudbuild.googleapis.com
  + Cloud Storage storage-component.googleapis.com
  + Cloud Run run.googleapis.com
* Get the source code & Activate your lab account:
  + gcloud auth list --filter=status:ACTIVE --format="value(account)"
  + git clone https://github.com/Deleplace/pet-theory.git
  + cd pet-theory/lab03
* Creating an invoice microservice
  + In this section you will create a Go application to process requests. As outlined in the architecture diagram, you will integrate Cloud Storage as part of the solution.
  + Navigate to pet-theory > lab03 > server.go
* package main
* import (
* "fmt"
* "io/ioutil"
* "log"
* "net/http"
* "os"
* "os/exec"
* "regexp"
* "strings"
* )
* func main() {
* http.HandleFunc("/", process)
* port := os.Getenv("PORT")
* if port == "" {
* port = "8080"
* log.Printf("Defaulting to port %s", port)
* }
* log.Printf("Listening on port %s", port)
* err := http.ListenAndServe(fmt.Sprintf(":%s", port), nil)
* log.Fatal(err)
* }
* func process(w http.ResponseWriter, r \*http.Request) {
* log.Println("Serving request")
* if r.Method == "GET" {
* fmt.Fprintln(w, "Ready to process POST requests from Cloud Storage trigger")
* return
* }
* //
* // Read request body containing Cloud Storage object metadata
* //
* gcsInputFile, err1 := readBody(r)
* if err1 != nil {
* log.Printf("Error reading POST data: %v", err1)
* w.WriteHeader(http.StatusBadRequest)
* fmt.Fprintf(w, "Problem with POST data: %v \n", err1)
* return
* }
* //
* // Working directory (concurrency-safe)
* //
* localDir, errDir := ioutil.TempDir("", "")
* if errDir != nil {
* log.Printf("Error creating local temp dir: %v", errDir)
* w.WriteHeader(http.StatusInternalServerError)
* fmt.Fprintf(w, "Could not create a temp directory on server. \n")
* return
* }
* defer os.RemoveAll(localDir)
* //
* // Download input file from Cloud Storage
* //
* localInputFile, err2 := download(gcsInputFile, localDir)
* if err2 != nil {
* log.Printf("Error downloading Cloud Storage file [%s] from bucket [%s]: %v",
* gcsInputFile.Name, gcsInputFile.Bucket, err2)
* w.WriteHeader(http.StatusInternalServerError)
* fmt.Fprintf(w, "Error downloading Cloud Storage file [%s] from bucket [%s]",
* gcsInputFile.Name, gcsInputFile.Bucket)
* return
* }
* //
* // Use LibreOffice to convert local input file to local PDF file.
* //
* localPDFFilePath, err3 := convertToPDF(localInputFile.Name(), localDir)
* if err3 != nil {
* log.Printf("Error converting to PDF: %v", err3)
* w.WriteHeader(http.StatusInternalServerError)
* fmt.Fprintf(w, "Error converting to PDF.")
* return
* }
* //
* // Upload the freshly generated PDF to Cloud Storage
* //
* targetBucket := os.Getenv("PDF\_BUCKET")
* err4 := upload(localPDFFilePath, targetBucket)
* if err4 != nil {
* log.Printf("Error uploading PDF file to bucket [%s]: %v", targetBucket, err4)
* w.WriteHeader(http.StatusInternalServerError)
* fmt.Fprintf(w, "Error downloading Cloud Storage file [%s] from bucket [%s]",
* gcsInputFile.Name, gcsInputFile.Bucket)
* return
* }
* //
* // Delete the original input file from Cloud Storage.
* //
* err5 := deleteGCSFile(gcsInputFile.Bucket, gcsInputFile.Name)
* if err5 != nil {
* log.Printf("Error deleting file [%s] from bucket [%s]: %v", gcsInputFile.Name,
* gcsInputFile.Bucket, err5)
* // This is not a blocking error.
* // The PDF was successfully generated and uploaded.
* }
* log.Println("Successfully produced PDF")
* fmt.Fprintln(w, "Successfully produced PDF")
* }
* func convertToPDF(localFilePath string, localDir string) (resultFilePath string, err error) {
* log.Printf("Converting [%s] to PDF", localFilePath)
* cmd := exec.Command("libreoffice", "--headless", "--convert-to", "pdf",
* "--outdir", localDir,
* localFilePath)
* cmd.Stdout, cmd.Stderr = os.Stdout, os.Stderr
* log.Println(cmd)
* err = cmd.Run()
* if err != nil {
* return "", err
* }
* pdfFilePath := regexp.MustCompile(`\.\w+$`).ReplaceAllString(localFilePath, ".pdf")
* if !strings.HasSuffix(pdfFilePath, ".pdf") {
* pdfFilePath += ".pdf"
* }
* log.Printf("Converted %s to %s", localFilePath, pdfFilePath)
* return pdfFilePath, nil
* }
  + Now run the following to build the application:
    - go build -o server
  + The functions called by this top-level code are in source files:
    - server.go
    - notification.go
    - gcs.go
  + With the application has been successfully built, you can create the pdf-conversion service.
* Create a pdf-conversion service
  + The PDF service will use Cloud Run and Cloud Storage to initate a process each time a file is uploaded to the designated storage.
  + To achieve this you will use a common pattern of event notifications together with Cloud Pub/Sub. Doing this enables the application to concentrate only on processing information. Transporting and passing information is performed by other services, which allows you to keep the application simple.
  + Building the invoice module requires the integration of two components:
    - Graphical user interface, application

      Description automatically generated
  + Adding the LibreOffice package means it can be used in your application.
  + In the Open editor, Open the existing Dockerfile manifest and update the file as shown below:
    - FROM debian:buster
    - RUN apt-get update -y \
    - && apt-get install -y libreoffice \
    - && apt-get clean
    - WORKDIR /usr/src/app
    - COPY server .
    - CMD [ "./server" ]
  + Initiate a rebuild of the pdf-converter image using Cloud Build:
    - gcloud builds submit \
    - --tag gcr.io/$GOOGLE\_CLOUD\_PROJECT/pdf-converter
  + Deploy the updated pdf-converter service.
  + NOTE: It's a good idea to give LibreOffice 2GB of RAM to work with, see the line with the --memory option.
  + Run these commands to build the container and to deploy it:
    - gcloud run deploy pdf-converter \
    - --image gcr.io/$GOOGLE\_CLOUD\_PROJECT/pdf-converter \
    - --platform managed \
    - --region us-central1 \
    - --memory=2Gi \
    - --no-allow-unauthenticated \
    - --set-env-vars PDF\_BUCKET=$GOOGLE\_CLOUD\_PROJECT-processed \
    - --max-instances=3
  + The Cloud Run service has now been successfully deployed. However we deployed an application that requires the correct permissions to access it.
* Create a Service Account
  + A Service Account is a special type of account with access to Google APIs.
  + In this lab uses a Service Account to access Cloud Run when a Cloud Storage event is processed. Cloud Storage supports a rich set of notifications that can be used to trigger events.
  + Next, update the code to notify the application when a file has been uploaded.
  + Click the Navigation menu > Storage, and verify that two buckets have been created. You should see:
    - PROJECT\_ID-processed
    - PROJECT\_ID-upload
  + Create a Pub/Sub notification to indicate a new file has been uploaded to the docs bucket ("uploaded"). The notifications will be labeled with the topic "new-doc".
    - gsutil notification create -t new-doc -f json -e OBJECT\_FINALIZE gs://$GOOGLE\_CLOUD\_PROJECT-upload
  + Create a new service account to trigger the Cloud Run services:
    - gcloud iam service-accounts create pubsub-cloud-run-invoker --display-name "PubSub Cloud Run Invoker"
  + Give the service account permission to invoke the PDF converter service:
    - gcloud run services add-iam-policy-binding pdf-converter \
    - --member=serviceAccount:pubsub-cloud-run-invoker@$GOOGLE\_CLOUD\_PROJECT.iam.gserviceaccount.com \
    - --role=roles/run.invoker \
    - --region us-central1 \
    - --platform managed
  + Find your project number by running this command:
    - PROJECT\_NUMBER=$(gcloud projects list \
    - --format="value(PROJECT\_NUMBER)" \
    - --filter="$GOOGLE\_CLOUD\_PROJECT")
  + Enable your project to create Cloud Pub/Sub authentication tokens:
    - gcloud projects add-iam-policy-binding $GOOGLE\_CLOUD\_PROJECT \
    - --member=serviceAccount:service-$PROJECT\_NUMBER@gcp-sa-pubsub.iam.gserviceaccount.com \
    - --role=roles/iam.serviceAccountTokenCreator
  + With the Service Account created it can be used to invoke the Cloud Run Service.
* Testing the Cloud Run service
  + Before progressing further, test the deployed service. Remember the service requires authentication, so test that to ensure it is actually private.
  + Save the URL of your service in the environment variable $SERVICE\_URL:
    - SERVICE\_URL=$(gcloud run services describe pdf-converter \
    - --platform managed \
    - --region us-central1 \
    - --format "value(status.url)")
  + Display the SERVICE URL:
    - echo $SERVICE\_URL
  + Make an anonymous GET request to your new service:
    - curl -X GET $SERVICE\_URL
  + The anonymous GET request will result in an error message "Your client does not have permission to get URL". This is good; you don't want the service to be callable by anonymous users.
  + Now try invoking the service as an authorized user:
    - curl -X GET -H "Authorization: Bearer $(gcloud auth print-identity-token)" $SERVICE\_URL
  + You will get the response "Ready to process POST requests from Cloud Storage trigger"
  + Great work, you have successfully deployed an authenticated Cloud Run service.
* Cloud Storage trigger
  + To initiate a notification when new content is uploaded to Cloud Storage, add a subscription to your existing Pub/Sub Topic.
  + Remember: Cloud Storage notifications will automatically push a message to your Topic queue when new content is uploaded. Using notifications allows you to create powerful applications that respond to events without needing to write additional code.
  + Create a Pub/Sub subscription so that the PDF converter will be run whenever a message is published to the topic new-doc:
    - gcloud pubsub subscriptions create pdf-conv-sub \
    - --topic new-doc \
    - --push-endpoint=$SERVICE\_URL \
    - --push-auth-service-account=pubsub-cloud-run-invoker@$GOOGLE\_CLOUD\_PROJECT.iam.gserviceaccount.com
  + Now whenever a file is uploaded the Pub/Sub subscription will interact with your Service Account. The Service Account will then initiate your PDF Converter Cloud Run service.
* Testing Cloud Storage Notification
  + To test the Cloud Run service, use the example files available.
  + Copy the test files into your upload bucket:
    - gsutil -m cp -r gs://spls/gsp762/\* gs://$GOOGLE\_CLOUD\_PROJECT-upload
  + Cloud Storage upload bucket > see the files are deleted, one by one, as they are converted to PDFs. Then click "-processed". It should contain PDF versions of all files.
  + Once the upload is done, Navigation > Cloud Run > pdf-converter > LOGS > a filter of "Converting" to see the converted files.
  + From the Navigation > Storage > "upload" folder and confirm all files uploaded have been processed.

## 8.5 Serverless Cloud Run Development: Challenge Lab

* Situational Overview
  + Pet Theory is a veterinary practice who are keen to utilize serverless architecture to update their existing systems.
  + In this challenge lab, you are part of the development team and have been assigned the task of migrating a service to serverless. Pay close attention to the provided instructions to successfully complete the exercise.
* Architecture
  + Pet Theory has nominated the existing monolithic Billing application to be reimagined using serverless.
  + Over the course of this lab, you will be expected to implement this design update.
    - Graphical user interface, application, Teams

      Description automatically generated
  + The development team will provide either the code or an image to be deployed as part of the solution.
* Developing a minimal viable product (MVP)
  + You will build a prototype solution for Pet Theory that meets the following high level requirement specification.

|  |  |
| --- | --- |
| **Ref** | **Definition of Done** |
| 1 | Deploy Staging Architecture |
| 2 | Deploy Prod Architecture |
| 3 | Secure Access between components in the Prod Architecture |

* Provision the Qwiklabs environment
  + gcloud config set project \
  + $(gcloud projects list --format='value(PROJECT\_ID)' \
  + --filter='qwiklabs-gcp')
  + gcloud config set run/region us-central1
  + gcloud config set run/platform managed
  + git clone https://github.com/rosera/pet-theory.git && cd pet-theory/lab07
* Task 1: Enable a Public Service
  + Overview, Set up a Rest API for the billing service. Use the information in the table below.

|  |  |
| --- | --- |
| **FIELD** | **VALUE** |
| Billing Image | billing-staging-api:0.1 |
| Billing Service | public-billing-service-188 |
| Authentication | unauthenticated |
| Code | pet-theory/lab07/unit-api-billing |

* + Architecture
    - Graphical user interface, application, Word

      Description automatically generated
  + Using the available code deploy the Billing Service.
  + Assessment, To complete this task successfully, you are required to implement the following:
    - Build an image using Cloud Build
    - Deploy a Cloud Run service as an unauthenticated service
    - Test service responds when the endpoint is accessed
  + Solution
    - cd ~/pet-theory/lab07/unit-api-billing
    - gcloud builds submit \
    - --tag gcr.io/$GOOGLE\_CLOUD\_PROJECT/billing-staging-api:0.1
    - gcloud run deploy public-billing-service-188 \
    - --image gcr.io/$GOOGLE\_CLOUD\_PROJECT/billing-staging-api:0.1 \
    - --platform managed \
    - --region us-central1 \
    - --allow-unauthenticated \
    - --max-instances=1
    - BILLING\_URL=$(gcloud beta run services describe public-billing-service-188 --platform managed --region us-central1 --format="value(status.url)")
    - echo $BILLING\_URL
    - curl -X POST $BILLING\_URL
* Task 2: Deploy a Frontend Service
  + Overview, Set up a Frontend Service. Use the information in the table below.

|  |  |
| --- | --- |
| **FIELD** | **VALUE** |
| Image Name | frontend-staging:0.1 |
| Service Name | frontend-staging-service-102 |
| Authentication | unauthenticated |
| Code | pet-theory/lab07/staging-frontend-billing |

* + Architecture
  + Graphical user interface, application, Word

    Description automatically generated
  + Assessment, To complete this section successfully, you are required to implement the following tasks:
    - Build an image using Cloud Build
    - Deploy the image to Cloud Run as unauthenticated service
    - Service should respond when the endpoint is accessed
  + Solution
    - cd ~/pet-theory/lab07/staging-frontend-billing
    - gcloud builds submit \
    - --tag gcr.io/$GOOGLE\_CLOUD\_PROJECT/frontend-staging:0.1
    - gcloud run deploy frontend-staging-service-102 \
    - --image gcr.io/$GOOGLE\_CLOUD\_PROJECT/frontend-staging:0.1 \
    - --platform managed \
    - --region us-central1 \
    - --allow-unauthenticated \
    - --max-instances=1
    - FrontEnd\_URL=$(gcloud beta run services describe frontend-staging-service-102 --platform managed --region us-central1 --format="value(status.url)")
    - echo $FrontEnd\_URL
    - curl -X POST $FrontEnd\_URL
* Task 3: Deploy a Private Service
  + Overview, The dev team updated their application and would like this deployed to the staging env.

|  |  |
| --- | --- |
| **FIELD** | **VALUE** |
| Image Name | billing-staging-api:0.2 |
| Service Name | private-billing-service-521 |
| Repository | gcr.io |
| Authentication | authenticated |
| Code | pet-theory/lab07/staging-api-billing |

* + Architecture
    - Graphical user interface, application, Word

      Description automatically generated
  + Assessment: Cloud Run Development, to implement the following tasks:
    - Delete the existing Billing Service
    - Build an image using Cloud Build
    - Deploy the image to Cloud Run requiring authentication
    - Assign the SERVICE\_URL to a environment variable
  + Note: Replace PRIVATE\_BILLING\_SERVICE inside the code-block with private-billing-service-521
    - BILLING\_URL=$(gcloud run services describe PRIVATE\_BILLING\_SERVICE \
    - --platform managed \
    - --region us-central1 \
    - --format "value(status.url)")
  + Service should respond when the endpoint is accessed
    - curl -X get -H "Authorization: Bearer $(gcloud auth print-identity-token)" $BILLING\_URL
  + Solution
    - task 3
    - cd ~/pet-theory/lab07/staging-api-billing
    - gcloud beta run services delete public-billing-service-188
    - gcloud builds submit \
    - --tag gcr.io/$GOOGLE\_CLOUD\_PROJECT/billing-staging-api:0.2
    - gcloud run deploy private-billing-service-521 \
    - --image gcr.io/$GOOGLE\_CLOUD\_PROJECT/billing-staging-api:0.2 \
    - --platform managed \
    - --region us-central1 \
    - --no-allow-unauthenticated \
    - --max-instances=1
    - BILLING\_URL=$(gcloud run services describe private-billing-service-521 --platform managed --region us-central1 --format="value(status.url)")
    - echo $BILLING\_URL
    - curl -X POST $BILLING\_URL
    - curl -X POST -H "Authorization: Bearer $(gcloud auth print-identity-token)" $BILLING\_URL
* Task 4: Create a Billing Service Account
  + Overview, In preparation for the deployment to prod, you will need to create a Service Account for the Billing Service.

|  |  |
| --- | --- |
| **FIELD** | **VALUE** |
| Service Account | billing-service-sa-399 |
| Display Name | Billing Service Cloud Run |
| Service Name | billing-service |
| Role | N/A |

* + Architecture
    - Graphical user interface, application

      Description automatically generated
  + Assessment: Service Account, , you are required to implement the following tasks:
    - Create a Service Account
  + Solution
    - gcloud iam service-accounts create billing-service-sa-399 --display-name "Billing Service Cloud Run"
* Task 5: Deploy the Billing Service
  + Overview, Associate the new Billing Service Account with Billing Service.

|  |  |
| --- | --- |
| **FIELD** | **VALUE** |
| Image Name | billing-prod-api:0.1 |
| Service Name | billing-prod-service-818 |
| Repository | gcr.io |
| Authentication | authenticated |
| Code | pet-theory/lab07/prod-api-billing |
| Service Account | billing-service-sa-399 |

* + Architecture
    - Graphical user interface, application, Word

      Description automatically generated
  + Assessment: Cloud Run Development, you are required to implement the following tasks:
    - Deploy the image to Cloud Run
    - Enable Authentication
    - Enable Service Account
    - Service should respond when the endpoint is accessed
  + Get the URL of the Billing Service
  + Note: Replace PRIVATE\_BILLING\_SERVICE inside the code-block with private-billing-service-521
    - PROD\_BILLING\_URL=$(gcloud run services \
    - describe PRIVATE\_BILLING\_SERVICE \
    - --platform managed \
    - --region us-central1 \
    - --format "value(status.url)")
  + Access the deployed endpoint
    - curl -X get -H "Authorization: Bearer \
    - $(gcloud auth print-identity-token)" \
    - $PROD\_BILLING\_URL
  + Solution
    - cd ~/pet-theory/lab07/prod-api-billing
    - gcloud builds submit \
    - --tag gcr.io/$GOOGLE\_CLOUD\_PROJECT/billing-prod-api:0.1
    - gcloud run deploy billing-prod-service-818 \
    - --image gcr.io/$GOOGLE\_CLOUD\_PROJECT/billing-prod-api:0.1 \
    - --platform managed \
    - --region us-central1 \
    - --no-allow-unauthenticated \
    - --max-instances=1
    - PROD\_BILLING\_URL=$(gcloud run services describe PRIVATE\_BILLING\_SERVICE private-billing-service-521 --platform managed --region us-central1 --format="value(status.url)")
    - echo $PROD\_BILLING\_URL
    - curl -X POST $PROD\_BILLING\_URL
    - curl -X POST -H "Authorization: Bearer $(gcloud auth print-identity-token)" $PROD\_BILLING\_URL
    - gcloud iam service-accounts create billing-service-sa-399 --display-name "Billing Service Cloud Run"
    - gcloud run services add-iam-policy-binding billing-prod-service-818 --member=serviceAccount:billing-service-sa-399@$GOOGLE\_CLOUD\_PROJECT.iam.gserviceaccount.com --role=roles/run.invoker --region us-central1 --platform managed
* Task 6: Frontend Service Account
  + Overview, Create a new Service Account for the Frontend capable of invoking the Billing Service.

|  |  |
| --- | --- |
| **FIELD** | **VALUE** |
| Service Account | frontend-service-sa-497 |
| Display Name | Billing Service Cloud Run Invoker |
| Service Name | frontend-prod-service |
| Role | run.invoker |

* + Architecture
    - Graphical user interface, application

      Description automatically generated
  + Assessment, you are required to implement the following tasks:
    - Create Service Account
    - Apply Service Account for Frontend Service
    - Give Service Account run.invoker permission
    - Bind Account to Service
  + Solution
    - gcloud iam service-accounts create frontend-service-sa-497 --display-name "Billing Service Cloud Run Invoker"
* Task 7: Redeploy the Frontend Service
  + Overview, Use the new Service Account and redeploy the Frontend Service.

|  |  |
| --- | --- |
| **FIELD** | **VALUE** |
| Image Name | frontend-prod:0.1 |
| Service Name | frontend-prod-service-491 |
| Repository | gcr.io |
| Authentication | unauthenticated |
| Code | pet-theory/lab07/prod-frontend-billing |
| Service Account | frontend-service-sa-497 |

* + Architecture
    - Graphical user interface, application, Word

      Description automatically generated
  + Assessment: Cloud Run Development, you are required to implement the following tasks:
    - Deploy the image to Cloud Run
    - Enable Authentication
    - Enable Service Account
    - Service should respond when the endpoint is accessed
  + Solution
    - cd ~/pet-theory/lab07/prod-frontend-billing
    - gcloud builds submit \
    - --tag gcr.io/$GOOGLE\_CLOUD\_PROJECT/frontend-prod:0.1
    - gcloud run deploy frontend-prod-service-491 \
    - --image gcr.io/$GOOGLE\_CLOUD\_PROJECT/frontend-prod:0.1 \
    - --platform managed \
    - --region us-central1 \
    - --no-allow-unauthenticated \
    - --max-instances=1
    - PROD\_BILLING\_URL=$(gcloud run services describe frontend-prod-service-491 --platform managed --region us-central1 --format="value(status.url)")
    - echo $PROD\_BILLING\_URL
    - curl -X POST $PROD\_BILLING\_URL
    - curl -X POST -H "Authorization: Bearer $(gcloud auth print-identity-token)" $PROD\_BILLING\_URL
    - gcloud iam service-accounts create frontend-service-sa-497 --display-name "Billing Service Cloud Run"
    - gcloud run services add-iam-policy-binding frontend-prod-service-491 --member=serviceAccount:frontend-service-sa-497@$GOOGLE\_CLOUD\_PROJECT.iam.gserviceaccount.com --role=roles/run.invoker --region us-central1 --platform managed
  + Now access the production frontend service to display the user interface. Information on screen is consumed from the private billing service using the service account.
* Graphical user interface, table

  Description automatically generated

## 9. Serverless Firebase Development

* learn how to do the following using Firebase: 1. Build serverless web apps. 2. Import data into a Firestore database. 3. Create a build pipeline with Cloud Build to automate deployments. 4. Build a Google Assistant application.

## 9.1 Importing Data to a Firestore Database

* Overview
  + For the labs in the Google Cloud Serverless Workshop(<https://www.cloudskillsboost.google/quests/98>

): Pet Theory Quest, you will read through a fictitious business scenario and assist the characters with their serverless migration plan.

* + Twelve years ago, Lily started the Pet Theory chain of veterinary clinics. The Pet Theory chain has expanded rapidly over the last few years. However, their old appointment scheduling system is not able to handle the increased load, so Lily is asking you to build a cloud-based system that scales better than the legacy solution.
  + Pet Theory's Ops team is a single person, Patrick, so they need a solution that doesn't require lots of ongoing maintenance. The team has decided to go with serverless technology.
  + Ruby has been hired as a consultant to help Pet Theory make the transition to serverless. After comparing serverless database options, the team decides to go with Cloud Firestore. Since Firestore is serverless, capacity doesn't have to be provisioned ahead of time which means that there is no risk of running into storage or operations limits. Firestore keeps your data in sync across client apps through real-time listeners and offers offline support for mobile and web, so a responsive app can be built that works regardless of network latency or Internet connectivity.
  + In this lab you will help Patrick upload Pet Theory's existing data to a Cloud Firestore database. He will work closely with Ruby to accomplish this.
* Architecture
  + This diagram gives you an overview of the services you will be using and how they connect to one another:
    - Graphical user interface, application

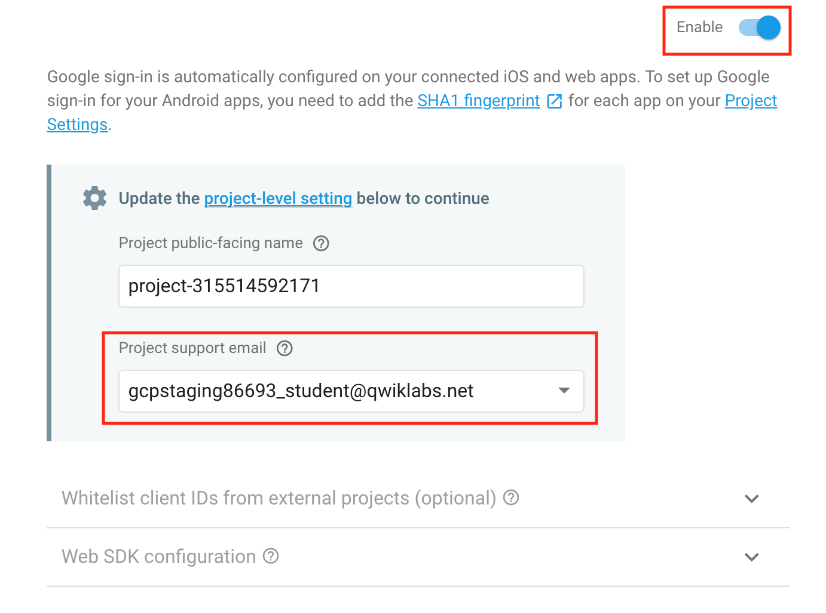
      Description automatically generated
  + What you will learn
    - Set up Firestore in Google Cloud.
    - Write database import code.
    - Generate a collection of customer data for testing.
    - Import the test customer data into Firestore.
    - Manipulate data in Firestore through the Console.
    - Add a developer to a Google Cloud project without giving them Firestore access.
* Set up Firestore in Google Cloud
  + Patrick's task is to upload Pet Theory's existing data to a Cloud Firestore database. He will work closely with Ruby to accomplish this goal. Ruby receives a message from Patrick in IT...
  + Patrick, IT Administrator
    - Hi Ruby,
    - Our first step in going serverless is creating a Firestore database with Google Cloud. Can you help with this task? I am not very familiar with setting this up.
    - Patrick
  + Ruby, Software Consultant
    - Hey Patrick,
    - Sure, I would be happy to help with that. I'll send you some resources to get started, let's get in touch once you're done creating the database.
    - Ruby
  + Help Patrick set up a Firestore database through the Cloud Console.
  + Navigation > Firestore > Click the Select Native Mode button > Create Database
    - Both modes are high performing with strong consistency, but they look different and are optimized for different use cases.
      * Native Mode is good for letting lots of users access the same data at the same time (plus, it has features like real-time updates and direct connection between your database and a web/mobile client
      * Datastore Mode puts an emphasis on high throughput (lots of reads and writes).
  + On completion of the task, Ruby emails Patrick...
  + Ruby, Software Consultant
    - Hey Patrick,
    - Great work setting up the Firestore database! To manage database access, we will use a Service Account that has been automatically created with the necessary privileges.
    - We are now ready to migrate from the old database to Firestore.
    - Ruby
  + Patrick, IT Administrator
    - Hey Ruby,
    - Thanks for the help, setting up the Firestore database was straightforward.
    - I hope the database import process will be easier than it is with the legacy database, which is quite complex and requires a lot of steps.
    - Patrick
* Write database import code
  + The new Cloud Firestore database is in place, but it's empty. The customer data for Pet Theory still only exists in the old database.
  + Patrick sends a message to Ruby...
  + Patrick, IT Administrator
    - Hi Ruby,
    - My manager would like to begin migrating the customer data to the new Firestore database.
    - I have exported a CSV file from our legacy database, but it's not clear to me how to import this data into Firestore.
    - Any chance you can lend me a hand?
    - Patrick
  + Ruby, Software Consultant
    - Hey Patrick,
    - Sure, let's set up a meeting to discuss what needs to be done.
    - Ruby
  + As Patrick said, the customer data will be available in a CSV file. Help Patrick create an app that reads customer records from a CSV file and writes them to Firestore. Since Patrick is familiar with Javascript, build this application with the Node.js JavaScript runtime.
  + In Cloud Shell, run the following command to clone the Pet Theory repository:
    - git clone https://github.com/rosera/pet-theory
    - cd pet-theory/lab01
  + In the directory you can see Patrick's package.json. This file lists the packages that your Node.js project depends on and makes your build reproducible, and therefore easier to share with others.
  + An example package.json is shown below:
    - {
    - "name": "lab01",
    - "version": "1.0.0",
    - "description": "This is lab01 of the Pet Theory labs",
    - "main": "index.js",
    - "scripts": {
    - "test": "echo \"Error: no test specified\" && exit 1"
    - },
    - "keywords": [],
    - "author": "Patrick - IT",
    - "license": "MIT",
    - "dependencies": {
    - "csv-parse": "^4.4.5"
    - }
    - }
  + Now that Patrick has his source code imported, he gets in touch with Ruby to see what packages he needs to make the migration work.
  + Patrick, IT Administrator
    - Hi Ruby,
    - The code I use for the legacy database is pretty basic, it just creates a CSV ready for the import process. Anything I need to download before I get started?
    - Patrick
  + Ruby, Software Consultant
    - Hi Patrick,
    - I would suggest using one of the many @google-cloud Node packages to interact with Firestore.
    - We should then only need to make small changes to the existing code since the heavy lifting has been taken care of.
    - Ruby
  + To allow Patrick's code to write to the Firestore database, you need to install some additional peer dependancies.
  + Run the following command to do so:
    - npm install @google-cloud/firestore
  + To enable the app to write logs to Cloud Logging, install an additional module:
    - npm install @google-cloud/logging
  + After successful completion of the command, the package.json will be automatically updated to include the new peer dependencies, and will look like this.
    - ...
    - "dependencies": {
    - "@google-cloud/firestore": "^2.4.0",
    - "@google-cloud/logging": "^5.4.1",
    - "csv-parse": "^4.4.5"
    - }
  + Now it's time to take a look at the script that reads the CSV file of customers and writes one record in Firestore for each line in the CSV file. Patrick's original application is shown below:
    - const {promisify} = require('util');
    - const parse = promisify(require('csv-parse'));
    - const {readFile} = require('fs').promises;
    - if (process.argv.length < 3) {
    - console.error('Please include a path to a csv file');
    - process.exit(1);
    - }
    - function writeToDatabase(records) {
    - records.forEach((record, i) => {
    - console.log(`ID: ${record.id} Email: ${record.email} Name: ${record.name} Phone: ${record.phone}`);
    - });
    - return ;
    - }
    - async function importCsv(csvFileName) {
    - const fileContents = await readFile(csvFileName, 'utf8');
    - const records = await parse(fileContents, { columns: true });
    - try {
    - await writeToDatabase(records);
    - }
    - catch (e) {
    - console.error(e);
    - process.exit(1);
    - }
    - console.log(`Wrote ${records.length} records`);
    - }
    - importCsv(process.argv[2]).catch(e => console.error(e));
  + It takes the output from the input CSV file and imports it into the legacy database. Next, update this code to write to Firestore.
  + Open the file pet-theory/lab01/importTestData.js.
  + To reference the Firestore API via the application, you need to add the peer dependency to the existing codebase.
  + Add the following Firestore dependency on line 4 of the file:
    - const {Firestore} = require('@google-cloud/firestore');
  + Ensure that your code looks like the following:
    - const {promisify} = require('util');
    - const parse = promisify(require('csv-parse'));
    - const {readFile} = require('fs').promises;
    - const {Firestore} = require('@google-cloud/firestore'); // add this
  + Integrating with the Firestore database can be achieved with a couple of lines of code. Ruby has shared some template code with you and Patrick for exactly that purpose.
  + Add the following code underneath line 9, or the if (process.argv.length < 3) conditional:
    - const db = new Firestore();
    - function writeToFirestore(records) {
    - const batchCommits = [];
    - let batch = db.batch();
    - records.forEach((record, i) => {
    - var docRef = db.collection('customers').doc(record.email);
    - batch.set(docRef, record);
    - if ((i + 1) % 500 === 0) {
    - console.log(`Writing record ${i + 1}`);
    - batchCommits.push(batch.commit());
    - batch = db.batch();
    - }
    - });
    - batchCommits.push(batch.commit());
    - return Promise.all(batchCommits);
    - }
  + The above code snippet declares a new database object, which references the database created earlier in the lab. The function uses a batch process in which each of the records is processed in turn and sets a document reference based on the identifier added. At the end of the function, the batch content is written to the database.
  + Finally, you need to add a call to the new function. Update the importCsv function to add the function call to writeToFirestore and remove the call to writeToDatabase. It should look like this:
    - async function importCsv(csvFileName) {
    - const fileContents = await readFile(csvFileName, 'utf8');
    - const records = await parse(fileContents, { columns: true });
    - try {
    - await writeToFirestore(records);
    - // await writeToDatabase(records);
    - }
    - catch (e) {
    - console.error(e);
    - process.exit(1);
    - }
    - console.log(`Wrote ${records.length} records`);
    - }
  + Next, add logging for the application. To reference the Logging API via the application, add the peer dependency to the existing codebase. Add the line const {Logging} = require('@google-cloud/logging'); just below the other require statements at the top of the file:
    - const {promisify} = require('util');
    - const parse = promisify(require('csv-parse'));
    - const {readFile} = require('fs').promises;
    - const {Firestore} = require('@google-cloud/firestore');
    - const {Logging} = require('@google-cloud/logging'); //add this
  + Add few constant variables and initialize the Logging client. Add those just below the above lines in the file (~line 5), like this:
    - const logName = 'pet-theory-logs-importTestData';
    - // Creates a Logging client
    - const logging = new Logging();
    - const log = logging.log(logName);
    - const resource = {
    - type: 'global',
    - };
  + Add code to write the logs in importCsv function just below the line console.log(Wrote ${records.length} records); which should look like this:
    - // A text log entry
    - success\_message = `Success: importTestData - Wrote ${records.length} records`
    - const entry = log.entry({resource: resource}, {message: `${success\_message}`});
    - log.write([entry]);
  + After these updates, your importCsv function code block should look like the following:
    - async function importCsv(csvFileName) {
    - const fileContents = await readFile(csvFileName, 'utf8');
    - const records = await parse(fileContents, { columns: true });
    - try {
    - await writeToFirestore(records);
    - //await writeToDatabase(records);
    - }
    - catch (e) {
    - console.error(e);
    - process.exit(1);
    - }
    - console.log(`Wrote ${records.length} records`);
    - // A text log entry
    - success\_message = `Success: importTestData - Wrote ${records.length} records`
    - const entry = log.entry({resource: resource}, {message: `${success\_message}`});
    - log.write([entry]);
    - }
  + Now when the application code is running, the Firestore database will be updated with the contents of the CSV file. The function importCsv takes a filename and parses the content on a line by line basis. Each line processed is now sent to the Firestore function writeToFirestore, where each new record is written to the "customer" database.
  + Note: In a production environment, you will write your own version of the import script.
* Create test data
  + Time to import some data! Patrick contacts Ruby about a concern he has about running a test with real customer data...
  + Patrick, IT Administrator
    - Hi Ruby,
    - I think it would be better if we don't use customer data for testing. We need to maintain customer privacy, but also need to have some confidence that our data import script works correctly.
    - Can you think of an alternative way to test?
    - Patrick
  + Ruby, Software Consultant
    - Hey Patrick,
    - Fair point, Patrick. This is a tricky area, as customer data may include personal identifiable information, also called PII.
    - I'll share some starter code with you to create pseudo customer data. We can then use this data to test the import script.
    - Ruby
  + Help Patrick get this pseudo-random data generator up and running.
  + First, install the "faker" library, which will be used by the script that generates the fake customer data. Run the following command to update the dependency in package.json:
    - npm install faker@4.1.0
  + Now open the file named createTestData.js with the code editor and inspect the code. Ensure it looks like the following:
    - const fs = require('fs');
    - const faker = require('faker');
    - function getRandomCustomerEmail(firstName, lastName) {
    - const provider = faker.internet.domainName();
    - const email = faker.internet.email(firstName, lastName, provider);
    - return email.toLowerCase();
    - }
    - async function createTestData(recordCount) {
    - const fileName = `customers\_${recordCount}.csv`;
    - var f = fs.createWriteStream(fileName);
    - f.write('id,name,email,phone\n')
    - for (let i=0; i<recordCount; i++) {
    - const id = faker.random.number();
    - const firstName = faker.name.firstName();
    - const lastName = faker.name.lastName();
    - const name = `${firstName} ${lastName}`;
    - const email = getRandomCustomerEmail(firstName, lastName);
    - const phone = faker.phone.phoneNumber();
    - f.write(`${id},${name},${email},${phone}\n`);
    - }
    - console.log(`Created file ${fileName} containing ${recordCount} records.`);
    - }
    - recordCount = parseInt(process.argv[2]);
    - if (process.argv.length != 3 || recordCount < 1 || isNaN(recordCount)) {
    - console.error('Include the number of test data records to create. Example:');
    - console.error(' node createTestData.js 100');
    - process.exit(1);
    - }
    - createTestData(recordCount);
  + Add Logging for the codebase. Reference the Logging API module from the application code with the following:
    - const fs = require('fs');
    - const faker = require('faker');
    - const {Logging} = require('@google-cloud/logging'); //add this
  + Now add a few constant variables and initialize the Logging client. Add those just below the const statements:
    - const logName = 'pet-theory-logs-createTestData';
    - // Creates a Logging client
    - const logging = new Logging();
    - const log = logging.log(logName);
    - const resource = {
    - // This example targets the "global" resource for simplicity
    - type: 'global',
    - };
  + Add code to write the logs in the createTestData function just below the line console.log(Created file ${fileName} containing ${recordCount} records.); which will look like this:
    - // A text log entry
    - const success\_message = `Success: createTestData - Created file ${fileName} containing ${recordCount} records.`
    - const entry = log.entry({resource: resource}, {name: `${fileName}`, recordCount: `${recordCount}`, message: `${success\_message}`});
    - log.write([entry]);
  + After updating, the createTestData function code block should look like this:
    - async function createTestData(recordCount) {
    - const fileName = `customers\_${recordCount}.csv`;
    - var f = fs.createWriteStream(fileName);
    - f.write('id,name,email,phone\n')
    - for (let i=0; i<recordCount; i++) {
    - const id = faker.random.number();
    - const firstName = faker.name.firstName();
    - const lastName = faker.name.lastName();
    - const name = `${firstName} ${lastName}`;
    - const email = getRandomCustomerEmail(firstName, lastName);
    - const phone = faker.phone.phoneNumber();
    - f.write(`${id},${name},${email},${phone}\n`);
    - }
    - console.log(`Created file ${fileName} containing ${recordCount} records.`);
    - // A text log entry
    - const success\_message = `Success: createTestData - Created file ${fileName} containing ${recordCount} records.`
    - const entry = log.entry({resource: resource}, {name: `${fileName}`, recordCount: `${recordCount}`, message: `${success\_message}`});
    - log.write([entry]);
    - }
  + Run the following command to configure your Project ID in Cloud Shell, replacing PROJECT\_ID with your Qwiklabs Project ID:
    - gcloud config set project PROJECT\_ID
  + Now set the project ID as an environment variable:
    - PROJECT\_ID=$(gcloud config get-value project)
  + Run the following command in Cloud Shell to create the file customers\_1000.csv, which will contain 1000 records of test data:
    - node createTestData 1000
  + You should receive a similar output:
    - Created file customers\_1000.csv containing 1000 records.
  + Open the file customers\_1000.csv and verify that the test data has been created.
* Import the test customer data
  + To test the import capability, use both the import script and the test data created earlier:
    - node importTestData customers\_1000.csv
  + If you get an error that resembles the following:
    - Error: Cannot find module 'csv-parse'
    - Run the following command to add the csv-parse package to your environment:
      * npm install csv-parse
  + Then run the command again. You should receive the following output:
    - Writing record 500
    - Writing record 1000
    - Wrote 1000 records
  + At this point, if you are feeling adventurous, feel free to create a larger test data file and import it into the Firestore database!
    - node createTestData 20000
    - node importTestData customers\_20000.csv
  + Over the past couple of sections you have seen how Patrick and Ruby have created test data and a script to import data into Firestore. Patrick now feels more confident about loading customer data into the Firestore database.
* Inspect the data in Firestore
  + With a little help from you and Ruby, Patrick has now successfully migrated the test data to the Firestore database. Open up Firestore and see the results!
  + Navigation > Firestore > customers > see the list of customers, EDIT, DELETE, DELETE DB
    - Select a customer's phone number to edit.
    - Now hover over the customer's email and select Delete field. Confirm by clicking Delete.
    - Click the three vertically placed dots next to customers to delete all records.
* Add a developer to the project without giving them Firestore access
  + Now that customer data has been pulled into Firestore, Patrick gets in touch with Ruby to plan the final phase of the database migration...
  + Patrick, IT Administrator
    - Hi Ruby,
    - I had a meeting with the Lily today, she is really pleased with the database migration to Firestore!
    - As part of that meeting, I was tasked with ensuring security permissions are set correctly.
    - Our developers should only be able to read the system log and to check-in source code. They should not be able to read or modify the data in Firestore. Can you help me set that up?
    - Patrick
  + Ruby, Software Consultant
    - Hi Patrick,
    - I'd be happy to!
    - I'll forward you some resources that will help you get set up.
    - Ruby
  + Help Patrick search a list of pre-defined roles for the roles that need to be allocated to Pet Theory developers.
  + Open the understanding roles page and search for "view logs" on the page. You'll find a role called roles/logging.viewer that lets a member read the logs.
  + Patrick also wants developers to be able to check code into source control. Search the page for the word "repository". There is a role called roles/source.writer that lets a member read and write to the source control repository.
  + Now add these two roles to the developer, replacing [EMAIL] with the user 2nd User ID you have for this lab:
    - gcloud projects add-iam-policy-binding $PROJECT\_ID \
    - --member=user:[EMAIL] --role=roles/logging.viewer
    - gcloud projects add-iam-policy-binding $PROJECT\_ID \
    - --member=user:[EMAIL] --role roles/source.writer
  + For this lab you're using the email you logged in with, but in a production environment you use the user's email to assign the role.
  + Log out, log in with 2nd user, you can see the Firestore database, but no edit or delete permission
  + Ruby, Software Consultant
    - Hi Patrick,
    - It was great working with you today. We all made a lot of good progress!
    - Ruby
  + Patrick, IT Administrator
    - Hi Ruby,
    - Thanks for all the pointers! I'm really impressed with Firestore's ease of set up and flexibility.
    - Pretty amazing that I was able to take my legacy JavaScript code and repurpose it to populate the Firestore database with a couple of changes.
    - Thanks for taking the time to walk me through all of this, it is very much appreciated!
    - Patrick

## 9.2 Build a Serverless Web App with Firebase

* Overview
  + For the labs in the Google Cloud Run Serverless Workshop, you will read through a fictitious business scenario and assist the characters with their serverless migration plan.
  + In the previous lab, Migrating Data to a Firestore Database, you learned how to leverage Firestore to host customer data. In this lab you will build a fully fledged Firebase web app that allows users to log information and schedule appointments in real time.
* Architecture
  + This diagram gives you an overview of the services you will be using and how they connect to one another:
    - Graphical user interface, application, Word, Teams

      Description automatically generated
  + What you will learn
    - Enable the Firebase API in your Google Cloud project.
    - Create and configure a Firebase project.
    - Configure Firestore Security to automate server-side authentication and authorization.
    - Add Google sign-in to your web app.
    - Configure your database so users can add their contact information.
    - Explore and deploy code that allows users to schedule appointments.
    - Explore Firebase's real time updates in your web app.
* Provisioning the Firebase environment
  + Following a successful database migration project, Ruby and Patrick have been tasked with assisting the Pet Theory web development team with Firebase hosting.
  + Ruby sends Patrick an email:
  + Ruby, Software Consultant
    - Hi Patrick,
    - Awesome work last week. Great to see that the clinic's data has been migrated to Firestore!
    - It looks like the next task is to use Firebase to host the Pet Theory website.
    - Ruby
  + Patrick, IT Administrator
    - Hi Ruby,
    - I haven't heard of Firebase hosting before, what are the benefits? Where would I get started?
    - Patrick
  + Ruby, Software Consultant
    - Hi Patrick
    - The primary benefit of Firebase hosting is that it is serverless, so there is no infrastructure to manage. Security rules are also embedded within the application, so permissions can be restricted to minimize issues when handling customer data.
    - It also has a "pay as you use" model, which means Firebase is a comprehensive mobile development platform for our use case.
    - Ruby
  + Patrick, IT Administrator
    - Hi Ruby
    - Sounds like Firebase will make security and infrastructure management (a big part of my job) a whole lot easier. I'm excited to not be billed for idle servers either!
    - Patrick
  + Ruby sends Patrick some background information in an email, and they hold a meeting to work out the key activities. From this meeting they determine that he needs to:
    - Enable the Google Cloud Firestore API.
    - Add the Firestore CLI to the Google Cloud project.
    - Create a Firebase project.
    - Establish security policies.
  + Next, help Patrick accomplish these tasks.
* Enable the Google Cloud Firestore API and Setup a Firestore Database
  + Navigation >APIs & Services > Library > Enable Google Cloud Firestore API.
  + Navigation > Firestore > Select Native Mode > Select nam5 location > Create Database.
* Create a Firebase project
  + Open the Firebase console(https://console.firebase.google.com/) in a new tab of your browser .
  + Click the account icon in the upper right-hand corner of the page and ensure that it is the Qwiklabs student account that you have been provisioned for this lab.
  + Then click Add project. When prompted for a project name, select your Qwiklabs Project ID from the drop-down menu.
  + Accept the Firebase terms and click Continue. Confirm the "Pay as you go" billing plan.
  + Click Continue and on the following page de-select the Google Analytics for your Firebase Project.
  + Then click Add Firebase. Click Continue when you see a prompt that says your new project is ready:
* Register your app
  + After completing the last step, you should be in the Firebase Console. If you close that page, you can find the link here.
  + Select the web icon (highlighted below) from the list of "Get started by adding Firebase to your app" icons:
    - Graphical user interface, text, application

      Description automatically generated
  + When prompted for an "App nickname", type in Pet Theory.
  + Then check the box next to "Also set up Firebase hosting for this app.
  + Click on the Register app button.
  + Click Next > Next > Continue to console. You should now be on the following page:
    - Graphical user interface, text

      Description automatically generated
  + You have now configured Firebase for the project. Next, you will initialize the project to reference your Firebase host.
* Install the Firebase CLI and deploy to Firebase Hosting
  + Return to the Cloud Console for this step.
  + In Cloud Shell, run the following command to clone the Pet Theory repository:
    - git clone https://github.com/rosera/pet-theory.git
    - cd pet-theory/lab02
  + In Cloud Shell run the following npm command to create a new Node app and install the Firebase CLI, a command-line tool:
    - npm init --yes
    - npm install -g firebase-tools@7.3.0
  + Now run the following command to link your Google account with Firebase:
    - firebase login --no-localhost
  + Enter in Y if asked if Firebase can collect error reporting information and press Enter.
  + Copy and paste the URL generated in a new browser tab and press Enter (directly clicking on the link results in an error).
  + Select your Qwiklabs Google account and then click Allow. You will then be given an access code:
  + Copy the access code, paste it in the Cloud Shell prompt, and press Enter. You should receive output similar to the following response:
  + Run the following command to set your project region to us-central1 so that it matches up with your Firebase project's resource location:
    - gcloud config set compute/region us-central1
  + Now initialize a new Firebase project in your current working directory:
    - firebase init
  + Running this command will step you through setting up your project directory and Firebase products.
  + You will be asked to select the Firebase CLI features you want set up in this folder. Use the arrow keys and the spacebar to select Firestore and Hosting. Ensure your shell matches the following and then hit Enter:
    - ? Which Firebase CLI features do you want to set up for this folder? Press Space to select features, then Enter to confirm your choices.
      * ◯ Database: Deploy Firebase Realtime Database Rules
      * ◉ Firestore: Deploy rules and create indexes for Firestore
      * ◯ Functions: Configure and deploy Cloud Functions
      * ❯◉ Hosting: Configure and deploy Firebase Hosting sites
      * ◯ Storage: Deploy Cloud Storage security rules
  + Then run through the rest of the steps to configure Firebase:
    - Key down to Use an existing project and press Enter.
    - Select your Qwiklabs Project ID from the list (it's the one that starts with "qwiklabs-gcp-") then Enter.
    - Press Enter and then N to keep your firestore.rules file.
    - Press Enter and then Y to keep your firestore.indexes.json file.
    - Press Enter to keep your public directory and then N to disallow rewrites to your /index.html file.
    - Press Enter to Set up automatic builds and deploys with GitHub? and press N.
    - Enter in N when prompted to overwrite your 404.html file.
    - Enter in N when prompted to overwrite your index.html file.
  + You should receive the following output:
    - ✔ Wrote public/404.html
    - ✔ Wrote public/index.html
    - i Writing configuration info to firebase.json...
    - i Writing project information to .firebaserc...
    - i Writing gitignore file to .gitignore...
    - ✔ Firebase initialization complete!
  + The local configuration is now complete. Now the database authentication step is required to provide access between the services.
* Set up authentication and a database
  + Return to the Firebase Console for this step.
  + Click on the Project Overview button in the left-hand navigation menu.
  + Select Authentication tile and then click on Get Started:
  + From the list of Sign-in Providers, click on the Google item.
  + Click the enable toggle in the top right corner and for the Project support email select your Qwiklabs Google account from the drop down list. Your page should now resemble the following:
    - 
  + Once you have verified the above, click on the Save button.
  + You have now set up Firestore authentication. In the following step you will work on hosting the application with Firebase.
* Scenario: set up the application
  + You have assisted Patrick in setting up a working Firebase hosting environment where a web developer can deploy their code. However, Patrick has never enabled Firebase authentication nor has he deployed code to Firebase, so he emails Ruby for some help...
  + Patrick, IT Administrator
    - Hey Ruby,
    - Thanks for all the tips! The Firebase environment looks like it is all set up. My next task is to deploy the webs developers' code.
    - Can you help me understand what this entails and what I need to do next?
    - Patrick
  + Ruby, Consultant
    - Hi Patrick,
    - That's great to hear! I'll send you the instructions on how to run the application and add the following features:
    - Set up web authentication for logging in.
    - Enable customer details to be logged on the Profile page.
    - Create a self service portal for appointments.
    - Ruby
  + Patrick, DevOps Engineer
    - Hi Ruby
    - That sounds like it will be quite a bit of work.
    - Does that mean I'll have to make structural changes every time I want to add something new? Not to mention the time it will take to see those updates...
    - Patrick
  + Ruby, Consultant
    - Hey Patrick
    - You can do most of the heavy lifting with Firebase libraries.
    - Seeing your updates in your browser is as simple as calling firebase deploy from the command line.
    - Make changes, deploy, and see your changes on the website!
    - Ruby
  + Patrick, IT Administrator
    - Hi Ruby
    - Wow, that's very comforting! Firebase hosting just gets better and better :-)
    - Patrick
  + Now that Patrick has a better understanding of what is required, you will help him deploy Pet Theory as a Firebase application.
* Configure Firestore authentication and add sign-in to your web app
  + open the pet-theory/lab02/firestore.rules file in the editor. Review the contents of the file:
    - service cloud.firestore {
    - match /databases/{database}/documents {
    - match /customers/{email} {
    - allow read, write: if request.auth.token.email == email;
    - }
    - match /customers/{email}/{document=\*\*} {
    - allow read, write: if request.auth.token.email == email;
    - }
    - }
    - }
  + This will configure the Firestore database so that each user can only access their own data.
  + Now open the public folder and select the index.html to view it in the code editor.
* Deploy your application
  + Run the following command to ensure that you are in the pet-theory/lab02/ directory:
    - cd ~/pet-theory/lab02/
    - firebase deploy
  + You should receive the following output:
    - ✔ Deploy complete!
    - Project Console: https://console.firebase.google.com/project/qwiklabs-gcp-7d652f8cf1f91cce/overview
    - Hosting URL: https://qwiklabs-gcp-01-8be196f95006.firebaseapp.com
  + Copy the hosting URL (should resemble [PROJECT-ID].firebaseapp.com) and open it in a new tab. Now click on the Sign in with Google button:
    - If you see a browser is not supported or 3rd party cookies and data may be disabled error, make sure to enable cookies in your browser. This can be done in Chrome by clicking the eye icon at the far right of the url tab and following the blue links in the popup. App\_02\_enable\_cookies.png
  + Select your Qwiklabs Google account. You will be brought to the following page:
    - Graphical user interface, application

      Description automatically generated
  + You have now deployed code to let users use Google authentication to access the appointments app.
    - Note: Managing passwords is a difficult task and could expose your company to additional risk. Also, users don't want to create yet another user id and password. A small company like Pet Theory doesn't have the resources or requisite skill set to do this. In this instance it is much better to let the application users log in with their existing Google account (or any other identity providers)!
* Add a customer page to your web app
  + Return to the Code Editor tab. You will next review three files in the public directory.
  + From the side menu, open the customer.html file and review the code.
  + Now open the customer.js file and copy and paste the following code:
    - let user;
    - firebase.auth().onAuthStateChanged(function(newUser) {
    - user = newUser;
    - if (user) {
    - const db = firebase.firestore();
    - db.collection("customers").doc(user.email).onSnapshot(function(doc) {
    - const cust = doc.data();
    - if (cust) {
    - document.getElementById('customerName').setAttribute('value', cust.name);
    - document.getElementById('customerPhone').setAttribute('value', cust.phone);
    - }
    - document.getElementById('customerEmail').innerText = user.email;
    - });
    - }
    - });
    - document.getElementById('saveProfile').addEventListener('click', function(ev) {
    - const db = firebase.firestore();
    - var docRef = db.collection('customers').doc(user.email);
    - docRef.set({
    - name: document.getElementById('customerName').value,
    - email: user.email,
    - phone: document.getElementById('customerPhone').value,
    - })
    - })
  + Now open the styles.css file and paste in the following code:
    - body { background: #ECEFF1; color: rgba(0,0,0,0.87); font-family: Roboto, Helvetica, Arial, sans-serif; margin: 0; padding: 0; }
    - #message { background: white; max-width: 360px; margin: 100px auto 16px; padding: 32px 24px 16px; border-radius: 3px; }
    - #message h3 { color: #888; font-weight: normal; font-size: 16px; margin: 16px 0 12px; }
    - #message h2 { color: #ffa100; font-weight: bold; font-size: 16px; margin: 0 0 8px; }
    - #message h1 { font-size: 22px; font-weight: 300; color: rgba(0,0,0,0.6); margin: 0 0 16px;}
    - #message p { line-height: 140%; margin: 16px 0 24px; font-size: 14px; }
    - #message a { display: block; text-align: center; background: #039be5; text-transform: uppercase; text-decoration: none; color: white; padding: 16px; border-radius: 4px; }
    - #message, #message a { box-shadow: 0 1px 3px rgba(0,0,0,0.12), 0 1px 2px rgba(0,0,0,0.24); }
    - #load { color: rgba(0,0,0,0.4); text-align: center; font-size: 13px; }
    - @media (max-width: 600px) {
    - body, #message { margin-top: 0; background: white; box-shadow: none; }
    - body { border-top: 16px solid #ffa100; }
    - }
  + Now that your customer page has been updated, return to the Console and your Cloud Shell session and run the following command to deploy the application:
    - firebase deploy
  + You should receive the following output:
    - ✔ Deploy complete!
    - Project Console: https://console.firebase.google.com/project/qwiklabs-gcp-7d652f8cf1f91cce/overview
    - Hosting URL: https://qwiklabs-gcp-01-8be196f95006.firebaseapp.com
  + Go to your app tab and hard refresh the page with CMND+SHIFT+R (Mac) or CTRL+SHIFT+R (Windows). Simple refreshing will not display the need updates. Enter some customer info—make up a name and phone number and click Save profile.
  + Go to the Firebase Console and click Firestore Database to view the profile information saved:
    - Graphical user interface, text, application, email

      Description automatically generated
  + Return to the web app page and click on the Appointments link. You will see a blank page since haven't deployed the appointments code yet.
* Let users schedule appointments
  + From the side menu, select appointments.html and paste in the following code:
    - <!DOCTYPE html>
    - <html>
    - <head>
    - <meta charset="utf-8">
    - <meta name="viewport" content="width=device-width, initial-scale=1">
    - <title>Pet Theory appointments</title>
    - <script src="/\_\_/firebase/6.4.2/firebase-app.js"></script>
    - <script src="/\_\_/firebase/6.4.2/firebase-auth.js"></script>
    - <script src="/\_\_/firebase/6.4.2/firebase-firestore.js"></script>
    - <script src="/\_\_/firebase/init.js"></script>
    - <link type="text/css" rel="stylesheet" href="styles.css" />
    - </head>
    - <body>
    - <div id="message">
    - <h2>Scheduled appointments</h2>
    - <div id="appointments"></div>
    - <hr/>
    - <h2>Schedule a new appointment</h2>
    - <select id="timeslots">
    - <option value="0">Choose time</option>
    - </select>
    - <br><br>
    - <button id="makeAppointment">Schedule</button>
    - </div>
    - <script src="appointments.js"></script>
    - </body>
    - </html>
  + Now open the appointments.js file and paste in the following code:
* let user;
* firebase.auth().onAuthStateChanged(function(newUser) {
* user = newUser;
* if (user) {
* const db = firebase.firestore();
* const appColl = db.collection('customers').doc(user.email).collection('appointments');
* appColl.orderBy('time').onSnapshot(function(snapshot) {
* const div = document.getElementById('appointments');
* div.innerHTML = '';
* snapshot.docs.forEach(appointment => {
* div.innerHTML += formatDate(appointment.data().time) + '<br/>';
* })
* if (div.innerHTML == '') {
* div.innerHTML = 'No appointments scheduled';
* }
* });
* }
* });
* const timeslots = document.getElementById('timeslots');
* getOpenTimes().forEach(time => {
* timeslots.add(new Option(formatDate(time), time));
* });
* document.getElementById('makeAppointment').addEventListener('click', function(ev) {
* const millis = parseInt(timeslots.selectedOptions[0].value);
* if (millis > 0) {
* const db = firebase.firestore();
* db.collection('customers').doc(user.email).collection('appointments').add({
* time: millis
* })
* timeslots.remove(timeslots.selectedIndex);
* }
* })
* function getOpenTimes() {
* const retVal = [];
* let startDate = new Date();
* startDate.setMinutes(0);
* startDate.setSeconds(0);
* startDate.setMilliseconds(0);
* let millis = startDate.getTime();
* while (retVal.length < 5) {
* const hours = Math.floor(Math.random() \* 5) + 1;
* millis += hours \* 3600 \* 1000;
* if (isDuringOfficeHours(millis)) {
* retVal.push(millis);
* }
* }
* return retVal;
* }
* function isDuringOfficeHours(millis) {
* const aDate = new Date(millis);
* return aDate.getDay() != 0 && aDate.getDay() != 6 &&
* aDate.getHours() >= 9 && aDate.getHours() <= 17;
* }
* function formatDate(millis) {
* const aDate = new Date(millis);
* const days = ['Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat'];
* const months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
* 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];
* return days[aDate.getDay()] + ' ' + aDate.getDate() + ' ' +
* months[aDate.getMonth()] + ', ' + aDate.getHours() + ':' +
* (aDate.getMinutes() < 10? '0'+aDate.getMinutes(): aDate.getMinutes());
* }
  + Now that your files have been built, view your changes. Run the following command in Cloud Shell to deploy your application:
    - firebase deploy
  + Refresh your web app tab, where you previously got the blank page. Go ahead and schedule a couple of appointments:
    - Graphical user interface, text, application

      Description automatically generated
  + Now go to the Firebase Console, click Firestore Database and select the appointments collection that you just created under your user.
  + You should see a similar collection of appointments codes:
    - Graphical user interface, text, application

      Description automatically generated
* See Firestore Real-Time updates
  + Open a new browser tab and point it to [PROJECT ID].firebaseapp.com, then log in using the Google button and click Appointments.
  + Arrange the two browser tabs side-by-side. In the first browser window, schedule a new appointment.
  + Now look at your other browser tab—you should see the appointment automatically added without having to refresh:
    - Graphical user interface, text, application

      Description automatically generated
  + Pretty cool! Firestore updates clients (web apps and native mobile apps) in real time, without the user having to refresh or reload.
  + Go to the Firebase Console, click Data tab in Cloud Firestore, and edit the data. You can even delete the appointments collection under your user record. Both browser windows keep updating in real time.

## 9.3 Deploy a Hugo Website with Cloud Build and Firebase Pipeline

* Overview
  + In this lab you will create a pipeline for deploying websites based on Hugo, a static website builder. You will store the website content in Cloud Source Repositories and deploy the website with Firebase, then use Cloud Build to create a pipeline to automatically deploy new content that is committed to the repository.
* Objectives
  + An overview of static websites
  + Setting up a website with Hugo
  + Storing the website content in Cloud Source Repositories
  + Deploying the website with Firebase
  + Creating a build pipeline with Cloud Build to automate the deployment
* Process overview
  + Here's a diagram of what you are going to build:
    - Diagram

      Description automatically generated
  + The goal is to be able to commit code and have it trigger the pipeline which will in turn deploy the website. Your journey will be divided into two parts. First, you will build the website locally and deploy it to Firebase manually so you can gain an understanding of the entire process. Second, you will automate the process by building a pipeline with Cloud Build.
* Manual deployment
  + First build the website manually on a Linux instance to learn the end-to-end process. You will also use the Linux instance to perform some of the one-time tasks that are needed to get Firebase up and running.
  + navigation > Compute Engine > VM Instances > see one instance that has been built > SSH.
  + Install Hugo locally
    - Now install Hugo locally in the Linux instance so that you can test the website locally before deploying it with Firebase. A shell script has been provided to make this easier.
    - In the Linux instance shell, examine the file cat /tmp/installhugo.sh.
    - Output (do not copy)
      * #!/bin/bash
      * # Copyright 2020 Google Inc. All rights reserved.
      * #
      * # Licensed under the Apache License, Version 2.0 (the "License");
      * # you may not use this file except in compliance with the License.
      * # You may obtain a copy of the License at
      * #
      * # http://www.apache.org/licenses/LICENSE-2.0
      * #
      * # Unless required by applicable law or agreed to in writing, software
      * # distributed under the License is distributed on an "AS IS" BASIS,
      * # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
      * # See the License for the specific language governing permissions and
      * # limitations under the License.
      * \_HUGO\_VERSION=0.69.2
      * echo Downloading Hugo version $\_HUGO\_VERSION...
      * wget \
      * --quiet \
      * -O hugo.tar.gz \
      * https://github.com/gohugoio/hugo/releases/download/v${\_HUGO\_VERSION}/hugo\_extended\_${\_HUGO\_VERSION}\_Linux-64bit.tar.gz
      * echo Extracting Hugo files into /tmp...
      * mv hugo.tar.gz /tmp
      * tar -C /tmp -xzf /tmp/hugo.tar.gz
      * echo The Hugo binary is now at /tmp/hugo.
    - Note the use of the wget command to download Hugo and the tar command to unpack the Hugo archive. You will see similar commands later in this lab when you create the pipeline.
    - Enter the commands below to run the script and install Hugo:
      * cd ~
      * /tmp/installhugo.sh
    - You will receive a message saying that Hugo has been installed into the /tmp directory as shown below. You are ready to build the website infrastructure.
      * Text

        Description automatically generated
  + Create a repository and the initial web site
    - Now create a Cloud Source Repository to hold the web site and then clone the repository to the Linux instance. Cloning a repository creates a mirror of it in the shell. This allows you to implement the web site while in the shell and later commit your changes to the file system. Later in this lab, you will set up a pipeline that responds to these commits to the repository.
    - Enter the following commands in the Linux instance shell:
      * cd ~
      * gcloud source repos create my\_hugo\_site
      * gcloud source repos clone my\_hugo\_site
    - You will receive confirmations about the creation of the repository and the cloning of the repository as shown in the figure below. You can ignore the two warning messages about the charge for the repository and that the repository is empty.
      * Graphical user interface, text

        Description automatically generated
    - Now you are ready to create the site structure. Enter the commands below in the Linux shell.
      * cd ~
      * /tmp/hugo new site my\_hugo\_site --force
    - Normally the hugo command creates the directory. The --force option will create the site in the repository directory, which already exists. This allows you to keep the Git-related information in the directory that you just cloned. You will see messages indicating that the site has been created as shown in the figure below.
      * Graphical user interface, text

        Description automatically generated
    - Now install the Ananke theme to provide a layout for your site. Enter the following commands in the Linux instance shell:
      * cd ~/my\_hugo\_site
      * git submodule add \
      * https://github.com/budparr/gohugo-theme-ananke.git \
      * themes/ananke
      * echo 'theme = "ananke"' >> config.toml
    - You will see messages indicating that the theme has been cloned, as shown below.
      * Text

        Description automatically generated
    - With the structure of the web site set up, you can now preview it. Enter the command below to launch the site at TCP port 8080:
      * cd ~/my\_hugo\_site
      * /tmp/hugo server -D --bind 0.0.0.0 --port 8080
    - Hugo will build the site and serve it for access on TCP port 8080 as shown in the figure below. The server will run until it is stopped by pressing Ctrl+C.
      * Text

        Description automatically generated
    - Open a browser tab and browse to the external IP address at port 8080. Use the following URL, replacing [EXTERNAL IP] with the external IP address of your instance:
      * http://[EXTERNAL IP]:8080
    - The web site should look like this.
      * Graphical user interface, text, application

        Description automatically generated
    - Go back to the Linux shell and press Ctrl+C to stop the Hugo server.
  + Add Firebase to your project
    - Now that you know what the website looks like, it's time to deploy it to Firebase. First enable Firebase within your existing project.
    - Open a new tab in your browser then open this link (https://console.firebase.google.com/) in it to go to the Firebase console.
    - Now click Add project. You will be asked to select a name for your project. Click inside of the name field and select your existing Google Cloud project that starts with "qwiklabs-gcp-..." as shown in the figure below:
      * Graphical user interface, text, application

        Description automatically generated
    - Accept the Firebase terms, then click Continue.
    - You may be asked to confirm the Firebase billing plan. The Firebase costs are included with the lab. If you are prompted, click Confirm plan.
    - You will be asked to acknowledge some of the criteria when adding Firebase to a project. Click Continue.
    - You will be asked to confirm the use of Google Analytics for this Firebase project. Since this is a lab environment, use the toggle to disable Google Analytics and click Add Firebase. It will take about one minute for Firebase to be added to the project.
    - Click Continue if prompted after Firebase is added.
  + Deploy the site to Firebase
    - Install Fireball CLI in the Linux instance shell:
      * curl -sL https://firebase.tools | bash
    - Now you need to initialize Firebase. Enter the command below into the shell:
      * cd ~/my\_hugo\_site
      * firebase init
    - Select Hosting using the arrow keys and spacebar. When asked for a project option, select Use an existing project, then use the arrow keys, spacebar, and the Enter key to select the Project ID provided on the lab instruction page. For the public directory, select the default value public. For configuring as a single page application, select the default value of N. For setting up automatic builds and deploys with GitHub, select N.
    - If asked to overwrite any existing files, select Y.
    - You are ready to deploy the application. Enter the commands below into the Linux instance shell to rebuild the site with Hugo and to deploy it with Firebase:
      * /tmp/hugo && firebase deploy
    - After the application has been deployed, you will receive a hosting URL. Click on it and you will see the same website being served from the Firebase CDN (content delivery network). If you receive a generic "welcome" message, wait a few minutes for the CDN to be initialized and refresh the browser window. Save this hosting URL for later use.
    - You have now performed the entire deployment locally. Next, automate the process from end to end using Cloud Build.
* Automate the deployment
  + Perform the initial commit
    - The goal of building the pipeline is to be able to trigger builds when changes are made to the repository. You will start by performing an initial commit to the repository so that you can validate your ability to make future changes.
    - Configure the git commands global parameters by entering the commands below into the Linux shell. Use your name (or any name you wish) in place of GIT\_NAME. Use the username/e-mail address you were given for this lab for the GIT\_EMAIL value. Make sure to include the quotation marks.
      * git config --global user.name "[GIT\_NAME]"
      * git config --global user.email "[GIT\_EMAIL]"
    - Enter the commands below in the Linux shell to create a .gitignore file to exclude certain directories from the repository:
      * cd ~/my\_hugo\_site
      * echo "resources" >> .gitignore
    - Perform the initial commit to the repository by entering the commands below:
      * git add .
      * git commit -m "Add app to Cloud Source Repositories"
      * git push -u origin master
    - You have now committed (uploaded) the initial version of the website to Google Cloud.
  + Configure the build
    - Cloud Build uses a file named cloudbuild.yaml in the root directory of the repository to perform the build. The file is in YAML format. Spacing and indentation are important, so it has already been placed on the Linux instance for you.
    - Enter the command below in the Linux shell. Note the final period (".") at the end of the cp command:
      * cd ~/my\_hugo\_site
      * cp /tmp/cloudbuild.yaml .
    - Run the following to see what the cloudbuild.yaml file looks like. Some of the lines have wrapped because of their length.
      * cat cloudbuild.yaml
    - Output (do not copy)
* # Copyright 2020 Google Inc. All rights reserved.
* #
* # Licensed under the Apache License, Version 2.0 (the "License");
* # you may not use this file except in compliance with the License.
* # You may obtain a copy of the License at
* #
* # http://www.apache.org/licenses/LICENSE-2.0
* #
* # Unless required by applicable law or agreed to in writing, software
* # distributed under the License is distributed on an "AS IS" BASIS,
* # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* # See the License for the specific language governing permissions and
* # limitations under the License.
* steps:
* - name: 'gcr.io/cloud-builders/wget'
* args:
* - '--quiet'
* - '-O'
* # Unless required by applicable law or agreed to in writing, software
* # distributed under the License is distributed on an "AS IS" BASIS,
* # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* # See the License for the specific language governing permissions and
* # limitations under the License.
* steps:
* - name: 'gcr.io/cloud-builders/wget'
* args:
* - '--quiet'
* - '-O'
* - 'firebase'
* - 'https://firebase.tools/bin/linux/latest'
* - name: 'gcr.io/cloud-builders/wget'
* args:
* - '--quiet'
* - '-O'
* - 'hugo.tar.gz'
* - 'https://github.com/gohugoio/hugo/releases/download/v${\_HUGO\_VERSION}/hugo\_extended\_${\_HUGO\_VERSION}\_Linux-64bit.tar.gz'
* waitFor: ['-']
* - name: 'ubuntu:18.04'
* args:
* - 'bash'
* - '-c'
* - |
* mv hugo.tar.gz /tmp
* tar -C /tmp -xzf /tmp/hugo.tar.gz
* mv firebase /tmp
* chmod 755 /tmp/firebase
* /tmp/hugo
* /tmp/firebase deploy --project ${PROJECT\_ID} --non-interactive --only hosting -m "Build ${BUILD\_ID}"
* substitutions:
* \_HUGO\_VERSION: 0.69.2
  + - Here are some observations about the cloudbuild.yaml file:
      * There are three named steps in this file each of which is performed by a container image. The first two steps use a Google-supported builder to use wget to download the Hugo and Firebase tools. These two steps run in parallel. Using the wget builder is faster than installing wget manually.
      * The third step uses a standard Ubuntu container to install Hugo and Firebase after which the site is built and deployed. Installing Hugo and Firebase for each deployment allows you to change the version of Hugo whenever you desire while also using the latest version of Firebase.
      * The tar and wget commands are nearly identical to those used earlier in the installhugo.sh script.
      * The file also uses a custom substitution variable (\_HUGO\_VERSION) and a Google-provided substitution variable (PROJECT\_ID) to allow for this template to be used in different environments.
      * The Hugo and Firebase binaries are created and installed in a temporary directory so that they do not inadvertently get deployed to the website itself.
  + Create the Cloud Build trigger
    - Now create a trigger that will respond to commits to the master branch of the repository.
    - Navigation Menu > Cloud Build > Triggers > Click CREATE TRIGGER.
    - For the trigger configuration, enter the following details:
      * Filed Value
      * Name commit-to-master-branch
      * Description Push to master
      * Event Push to a branch
      * Repository my\_hugo\_site
      * Branch (regex) ^master$ (be sure Invert Regex is unchecked)
      * Build Configuration Cloud Build configuration file (yaml or json)
      * Cloud Build Configuration file location / cloudbuild.yaml
    - Click Create.
  + Update the Cloud Build service account
    - The Cloud Build Service account needs to have permissions to use Firebase to deploy the website.
    - From the Cloud Console Navigation menu, select IAM & Admin > IAM.
    - Locate the entry containing cloudbuild.gserviceaccount.com. Note that there is another service account that contains cloudbuild. Make sure you pick the service account cloudbuild.gserviceaccount.com. Click the Edit principal icon, then click ADD ANOTHER ROLE and add the role Firebase Products > Firebase Hosting Admin to it. Click SAVE.
  + Test the pipeline
    - Now that you have created the pipeline, you can make a change to the site then commit it to see if the change propagates.
    - In the Linux shell enter the command below to move to the repository directory:
      * cd ~/my\_hugo\_site
    - Edit the file config.toml and change the title to something different, such as My Cool New Hugo Site and save the changed file.
    - In the Linux shell, enter the commands below to commit the changes to the repository and trigger the Cloud Build pipeline:
      * git add .
      * git commit -m "I updated the site title"
      * git push -u origin master
    - Go to the Cloud Build console and check the build history. You should see a successful deployment as shown in the figure below. If not, consult the build details to identify the problem. Browse to the hosting URL you had received before. If you do not have it, you can go to the Firebase console and examine the project to find the domain name. It may take a few minutes for the CDN to update. Note that the site has an SSL certificate and is accessed using the https (Hypertext Transfer Protocol Secure) protocol.
      * Table

        Description automatically generated

## 9.4 Google Assistant: Build an Application with Dialogflow and Cloud Functions

* Overview
  + Google Assistant <https://assistant.google.com/#?modal_active=none> is a personal voice assistant that offers a host of actions and integrations. From making appointments and setting reminders, to ordering coffee and playing music, the 1 million+ actions available suit a wide range of voice command tasks. Google Assistant is offered on Android and iOS, but it can even be integrated with other devices like smartwatches, Google Homes, and Android TVs.
  + Actions <https://developers.google.com/actions/extending-the-assistant> is the central platform for developing Google Assistant applications. The Actions platform integrates with human-computer interaction suites, which simplifies conversational app development. The most widely used suite is Dialogflow <https://dialogflow.com/> , which uses an underlying machine learning (ML) and natural language understanding (NLU) schema to build rich Assistant applications. The Actions platform also integrates with Cloud Functions, which lets you run backend fulfillment code in response to events triggered by Dialogflow requests.
  + In this lab, you will get hands-on practice with the Actions platform, the Dialogflow suite, and Cloud Functions by building a "Silly Name Maker" application, which returns a user with a silly name after they have entered in a lucky number and favorite color. You will build a Dialogflow agent that intelligently parses user input for specific information. The agent will be supplemented with a webhook, which will trigger a Cloud Function that handles fulfillment logic and returns your user with their silly name.
  + What you will learn
* Create an Actions project and build an Action.
  + Create a Dialogflow agent and configure the default welcome intent.
  + Build a custom intent with entities.
  + Initialize a Cloud Function.
  + Add fulfillment logic and packages to your Cloud Function.
  + Add a webhook to your Action.
  + Test your Assistant application with the Actions simulator on expected and unexpected conversational paths.
  + Optional: test your Assistant application on a Google Home device.
* Create an Actions project
  + Regardless of the Assistant application you're building, you will always have to create an Actions project so your app has an underlying organizational unit.
  + Open the Actions on Google Developer Console <http://console.actions.google.com/> in a new tab. Sign in with your Qwiklabs credentials if prompted. You should be looking at a clean Actions console that resembles the following:
    - Graphical user interface, text, application

      Description automatically generated
  + Click New Project and agree to Actions on Google's terms of service when prompted by clicking Agree and continue.
  + Click into the Project Name field and select your Qwiklabs Google Cloud project ID from the dropdown. Then click Import project:
    - Graphical user interface, application

      Description automatically generated
  + Soon after you will be presented with a welcome page that resembles the following:
    - Graphical user interface, application

      Description automatically generated
  + Now click Actions Console in the top left corner to return to the homepage. Then click on the project you just created (title has your Project ID as the name.)
* Build an Action
  + An action is an interaction you build for the Google Assistant. An action supports a specific intent (a goal or task that users want to accomplish), which is carried out by a corresponding fulfillment (logic that handles an intent and carries out the corresponding Action.) You will now build an Action that supports silly name generation.
  + Click on your project name. Then from the center menu click Build your Action > Add Action(s) > Get Started. Then select Custom Intent > BUILD:
    - Graphical user interface, application

      Description automatically generated
  + This will take you to the Dialogflow console. Select your Qwiklabs account and click Allow when Dialogflow prompts you for permission to access your Google Account.
  + When you land on the Dialogflow account settings page, check the box next to Yes, I have read and accept the agreement and click ACCEPT.
  + If you are brought to the following Dialogflow agent creation page, click CREATE:
    - Graphical user interface, text, application, email

      Description automatically generated
  + If you are brought to this page instead:
    - Graphical user interface, website

      Description automatically generated
  + Close the Dialogflow agent creation tab. You will return to the Actions Console.
  + Click Get Started > Custom Intent > BUILD.
  + Select your Qwiklabs account and click Allow when Dialogflow prompts you for permission to access your Google Account.
  + Now click CREATE:
    - Graphical user interface, text, application, email

      Description automatically generated
  + An agent is an organizational unit that collects information needed to complete a user's request, which it then forwards to a service that provides fulfillment logic.
  + You will now build the basic framework for fulfillment logic. This will be handled (later) by a Cloud Function, which will return a response with a user's silly name.
  + Click Fulfillment from the left-hand menu. Move the slider for Webhook to the right, setting it to Enabled.
  + Now enter the temporary URL https://google.com in for the URL field. You will update this URL when you build your Cloud Function. Your page should resemble the following:
    - Graphical user interface, text, application, email

      Description automatically generated
  + croll down and click Save in the bottom right corner. Then click Intents from the left hand menu and select Default Welcome Intent:
    - Graphical user interface, text, application, email

      Description automatically generated
  + You will now build the main entry point into your application by configuring the default welcome intent.
* Configure the default welcome intent
  + When you created the Dialogflow agent, a Default Welcome intent is automatically created. This intent represents the main entry point into your conversation and the main action of your app. Your app must have a Default Welcome Intent defined, so that Actions on Google knows how to invoke your app. See Invocation and Discovery for more information on how these invocation models work.
  + Make sure that you are in the Default Welcome Intent. Here are some things to notice about the default settings:
    - The Events section of the intent specifies a Welcome event, which signifies that this intent is the default entry point into your app. The Google Assistant uses this event to trigger your app when users invoke your app by name, such as with "Ok Google, talk to Silly Name Maker".
    - In the Responses section, there are default, static text responses. The Default Welcome Intent contains pre-populated responses that you should remove to add your own.
  + Now that you know the Default Welcome Intent's purpose, you will modify some fields. Scroll down to the Responses section and click the trash icon to scrap all of the default text responses:
    - Graphical user interface, application

      Description automatically generated
  + Now click on ADD RESPONSES > Text response, and type in the following: Hi! Welcome to Silly Name Maker! Let's get started. What is your lucky number?
    - Graphical user interface, application

      Description automatically generated
  + Now scroll up and click Save in the top-right corner. Now when users invoke your app, they know that they are entering your app's experience and what they should say next.
  + In general, the app's responses should guide users on what to say next and should stay within your conversations grammar. This will be mapped out further in the following section.
* Design the conversation
  + Before you begin building out your Assistant application's language path, it's always a good idea to map out a conversation by writing sample dialogs. For the Silly Name Maker app, we've provided a couple of simple dialogs, which are described below:
  + Expected path
    - App: Hi! Welcome to Silly Name Maker! Let's get started. What is your lucky number?
    - User: 5
    - App: What is your favorite color?
    - User: Yellow
    - App: Alright, your silly name is Yellow 5! I hope you like it. See you next time.
  + Unexpected path
    - App: Hi! Welcome to Silly Name Maker! Let's get started. What is your lucky number?
    - User: Dog
    - App: Hmm that doesn't sound like a number to me. What's your lucky number again?
    - User: 22
    - App: What is your favorite color?
    - User: Blue
    - App: Alright, your silly name is Blue 22! I hope you like it. See you next time.
  + Conversational design tips: This example shows only a couple of the dialogs that you could write for your own apps. See this Design Walkthrough for more examples of dialogs.
  + Now that you have a better understanding of the type of application you'll be building, you will develop your custom intent to parse and extract colors and numbers from user input.
* Configure a custom intent
  + Dialogflow intents define your conversation's grammar and what tasks to carry out when users make specific requests. Since Dialogflow includes a natural language understanding (NLU) engine, you don't need to be exhaustive when defining phrases. Dialogflow automatically expands on the phrases you provide and understands many more variations of them.
  + You will now create an intent that defines how users need to provide their lucky number and favorite color to generate a silly name.
  + From the left hand menu, click on the + icon by the Intents menu item. For the Intent name field, enter make\_name:
    - Application

      Description automatically generated with medium confidence
  + In the Training phrases field click Add Training Phrases. Then in the Add user expression field enter in My lucky number is 23. Highlight the number 23 and assign it the @sys.number "entity" and hit Enter.
  + Your actions and parameters section should resemble the following:
    - Graphical user interface, text, application

      Description automatically generated
  + Entities extract specific data from user expressions and store them in accessible variables. By assigning this portion of the user says to be an entity, Dialogflow can extract parameters from the user input, validate the parameter and provide it to your fulfillment as a variable.
  + Without assigning entities, you would have to parse the input yourself and find the parameters you need.
  + You will now add some more user expressions in the Training phrases section, and assign any number within those phrases the @sys.number entity (if it isn't automatically set.) Add the following training phrases by entering them in the Add user expression field:
    - 23
    - The luckiest number I have is 12
    - My lucky number is 18
  + Your Training phrases section should now resemble the following — note the highlighted numbers, which align with the @sys.number entity:
    - Graphical user interface, text, application, email

      Description automatically generated
  + Now that your training phrases have been added, scroll down and expand the Actions and parameters > MANAGE PARAMETERS AND ACTION section.
  + Check the required checkbox for the number parameter. This tells Dialogflow to not trigger the intent until the parameter is properly provided by the user.
  + Now click on Define prompts for the number parameter (right-hand side) and provide a re-prompt phrase. Type in What's your lucky number? in the prompt field and then click Close:
    - Graphical user interface, application

      Description automatically generated
  + This phrase will be spoken to the user repeatedly until Dialogflow detects a number in the user input.
  + In the Action and parameters section, add a parameter with the following information:
    - Required - Select the checkbox
    - Parameter name - color
    - Entity - @sys.color
    - Value - $color
    - Prompts - What's your favorite color?
  + This additional parameter uses Dialogflow's built-in "slot-filling" feature, which allows you to obtain additional input parameters from the user without having to create an intent for each one and without having to make the user speak all the required input in one phrase.
  + Slot filling also lets you set parameters as being required so that your agent doesn't process the input until the user provides all required parameters. Your actions and parameters should now resemble the following:
    - Graphical user interface

      Description automatically generated with medium confidence
  + Now scroll to the top of the page and click Save.
  + Now scroll down and expand the Fulfillment section and click Enable fulfillment. Then click the Enable webhook call for this intent slider:
    - Graphical user interface, text, application, email

      Description automatically generated
  + This tells Dialogflow to call your fulfillment to generate a response to the user instead of using Dialogflow's response feature.
  + In the Responses section right above Fulfillment, click on + icon and select Google Assistant.
  + Move the toggle for Set this intent as end of conversation. This tells Dialogflow to relinquish control back to the Google Assistant after your fulfillment returns a response to the user.
    - Graphical user interface, text, application

      Description automatically generated
  + Then scroll to the top of the intent and click SAVE once more to save the entire intent.
  + You have now successfully declared your conversation's grammar with Dialogflow intents. You have also used Dialogflow's built-in response feature to return a static response to the user when they invoke your app. You also created an intent that uses fulfillment to return a response, which you will now implement with a Cloud Function.
* Initialize and configure a Cloud Function
  + You will now build a Cloud Function to handle your fulfillment logic. Your fulfillment takes the extracted user input that Dialogflow parsed and responds with the user's silly name. Return to the Cloud Console for this step.
  + Open the Navigation menu and select Cloud Functions, which is located under the serverless header. Then click CREATE FUNCTION.
  + This will open a template to create a new Cloud Function. Your page will resemble the following:
    - Graphical user interface, text, application, email

      Description automatically generated
  + For the Cloud Function Name field, enter in silly-name-maker.
  + Then scroll down to the authentication section and check the box next to Allow unauthenticated invocations:
    - Graphical user interface, text, application, email

      Description automatically generated
  + Click Save.
  + Forgetting to do the above will cause your simulation test to fail at the end!
  + Click Next.
  + Now scroll down and find the inline editor for index.js and package.json. Make sure that the index.js tab is open. This file defines your fulfillment logic and is used to create and deploy a Cloud Function. Here are some specifics on its basic functioning:
    - When Dialogflow intents are triggered, the intent's action name (declared in the action area of the intent) is provided to you in the request to your fulfillment. You use this action name to determine what logic to carry out.
    - Within every request to your fulfillment, if Dialogflow parsed parameters from the user input, you can access the parameter by name. Here, you declare the names of the parameters so you can access them later.
    - This function fulfills the action by generating a silly name. It's called whenever the make\_name intent is triggered. It uses the parameters from the user input to generate the name.
  + Now that you have a better understanding of index.js, you will build out the function's fulfillment logic. Remove the boilerplate code from the file. Then copy and paste the following code into index.js:
    - 'use strict';
    - const {dialogflow} = require('actions-on-google');
    - const functions = require('firebase-functions');
    - const app = dialogflow({debug: true});
    - app.intent('make\_name', (conv, {color, number}) => {
    - conv.close(`Alright, your silly name is ${color} ${number}! ` +
    - `I hope you like it. See you next time.`);
    - });
    - exports.sillyNameMaker = functions.https.onRequest(app);
    - Copied!
    - Now open the package.json tab. This file declares package dependencies for your fulfillment, including the Actions client library. Replace the contents of the file with the following:
    - {
    - "name": "silly-name-maker",
    - "description": "Find out your silly name!",
    - "version": "0.0.1",
    - "author": "Google Inc.",
    - "dependencies": {
    - "actions-on-google": "^3.0.0",
    - "firebase-admin": "^10.0.1",
    - "firebase-functions": "3.16.0"
    - }
    - }
  + Once you have those files configured, find the Entry point field. Enter sillyNameMaker for the value.
  + Now click the Deploy button below. It will take about a minute for your function to be built. When the creation completes, your overview page will resemble the following:
    - Graphical user interface, text, application

      Description automatically generated
  + Now click on the silly-name-maker function to get more details about it's configuration. Then click on the Trigger tab. You will see a trigger URL that resembles the following:
    - Graphical user interface, application

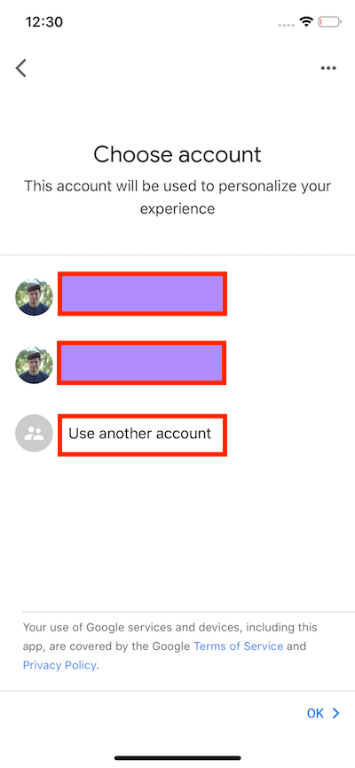
      Description automatically generated
  + Copy the trigger URL. You will use it as the URL for the Dialogflow webhook, which is configured in the next section.
* Configure the webhook
  + Return to the Dialogflow console and click on the Fulfillment menu item from the left hand navigation menu.
  + In the URL field, replace https://google.com with the trigger URL you generated in the previous step.
  + Your webhook should now resemble the following:
    - Graphical user interface, text, application, email

      Description automatically generated
  + Scroll down to the bottom of the page and click Save in the lower right corner.
* Test your Assistant application with the Actions simulator
  + Now that you your Cloud Function has been deployed and your webhook has been properly set up, you can preview the app in the Actions simulator.
  + Check your Google permission settings
    - In order to test the Silly Name Maker, you need to enable the necessary permissions.
    - Open a new tab in your browser and visit the Activity Controls page. Sign in with your Qwiklabs credentials if requested.
    - Ensure that the following permissions are enabled by sliding the toggles to TURN ON the following cards (confirm by clicking on Turn on > Got it):
      * Web & App Activity
    - Now close the Activity Controls page.
  + Test the application with the simulator
    - Return to the Dialogflow console. Then from the left-hand menu, click Integrations. Then click on integration link inside the Try Actions Builder section.
    - Once you land on the following page, click TEST:
      * Graphical user interface, text, application

        Description automatically generated
  + To invoke the Action, hit the enter key in the Talk to my test app box of the simulator console. You should be presented with a similar response:
    - Graphical user interface, text, application

      Description automatically generated
  + Now enter in a number. When prompted to enter in a favorite color, enter in a color. You should be returned with a silly name built from the number and color you entered in:
    - Graphical user interface, text, application, email

      Description automatically generated
  + Now try entering in some input that follows the unexpected path. Enter Talk to my test app in the input area once more. Then provide a number when asked to enter in your lucky number.
  + When asked for your favorite color, try entering in a word that isn't a color. The reprompt phrase should kick in, asking you for your favorite color again:
    - Graphical user interface, text, application

      Description automatically generated
  + Once you provide the correct input, your silly name will be generated and the conversation will end.
* Optional: test your application on a Google Home device
  + In this optional section, you will learn how to test an application on a Google Home device. At the risk of stating the obvious, you must have a Google Home device to complete this step. You will also need a smart phone (Android or iOS) to configure your Google Home. The following walkthrough uses an iPhone. If you use Android the steps may be slightly different.
  + Make sure that you have a new device or one that is factory reset before you begin this step. This guide will teach you how to reset all Google Home devices.
  + Configure your Google Home
    - Ensure that your device is plugged into an outlet and download the Google Home application for android or iOS. Ensure that your smartphone's Wi-Fi and Bluetooth are turned on.
    - Once the application has downloaded, open it and click GET STARTED. Then for the Choose account section, select Use another account > OK:
    - 
    - Then sign in using your Qwiklabs username and password. You will then be prompted to Choose a Google Home device. Select Create another home > Next:
      * Graphical user interface, application

        Description automatically generated
  + Now give your Google Home a nickname (lab instructions use Qwiklabs.) Your page should now resemble the following:
    - * Diagram

        Description automatically generated with medium confidence
    - Now, open your Wi-Fi settings on your smart phone. You should see a connection like GoogleHomeXXXX.x. Connect to this Wi-Fi network:
      * Graphical user interface, text, application

        Description automatically generated
    - Note: the above screenshot comes from an iPhone. Android will differ.
    - Now return to the Google Home application. Your application should automatically detect the Google Home device and will attempt to connect to it. Once connected, you will be asked if you heard the test sound:
      * Graphical user interface, text, application, chat or text message

        Description automatically generated
    - If so, select YES. Decide whether or not you want to help improve your Google Home by sending crash reports. When prompted, select where your device is located (e.g. the kitchen) and click NEXT.
    - Now you will be asked to select a Wi-Fi network. Choose your local network. You will then see the following "Connecting to Wi-Fi" screen:
      * Chart, bubble chart

        Description automatically generated
    - Once the device is connected, you will be asked to "Set up Google Assistant". Click NEXT.
    - Note: if this step fails and you get an "operation couldn't be completed" error, try clicking NEXT once more. If this doesn't work, you may have to run through the steps in this section once more.
    - Decide whether or not you want Google Assistant to recognize your voice. Decide whether you want to get personal results. Specify whether or not you want to use Voice 1 or Voice 2 for your Assistant. Enter your address.
    - Say NOT NOW when prompted to add media services. Say NO THANKS when prompted to receive email updates. When you land on the "Almost done!" page, click NEXT. You should now receive a prompt that says your device is ready:
      * A picture containing chart

        Description automatically generated
    - Click CONTINUE. Follow any extra set up prompts.
    - Now that you have set up your Google Home with your Qwiklabs account, you're all ready to test your application.
  + Test the application on your Google Home
    - Now that your device is configured with your Qwiklabs account, you can test your application using voice commands.
    - Say the following to your Google Home:
    - "Okay Google, talk to my test app".
    - You will then hear "Sure, here's the version of of my test app." This will be followed by the welcome intent: "Hi, welcome to Silly Name Maker! Let's get started. What is your favorite number?"
    - You can now follow the prompts to generate a silly name.

## 9.4 Serverless Firebase Development: Challenge Lab

* Prerequisites
  + Firestore
  + Cloud Run
  + Cloud Build
  + Container Registry
* Provision the environment
  + Link to the project:
    - gcloud config set project $(gcloud projects list --format='value(PROJECT\_ID)' --filter='qwiklabs-gcp')
  + Clone the repo:
    - git clone https://github.com/rosera/pet-theory.git
* Challenge scenario
  + In this lab you will create a frontend solution using a Rest API and Firestore database. Cloud Firestore is a NoSQL document database that is part of the Firebase platform where you can store, sync, and query data for your mobile and web apps at scale. Lab content is based on resolving a real world scenario through the use of Google Cloud serverless infrastructure.
  + You will build the following architecture:
    - Graphical user interface, application, Word

      Description automatically generated
* Task 1: Create a Firestore database
  + In this scenario you create a Firestore Database in Google Cloud. The high level architecture diagram below summarizes the general architecture.
    - Graphical user interface, diagram

      Description automatically generated
  + Requirements:

|  |  |
| --- | --- |
| **Field** | **Value** |
| Cloud Firestore | Native Mode |
| Location | Nam5 (United States) |

* + Create a Firestore database
    - To complete this section successfully, you are required to implement the following:
    - Cloud Firestore Database
    - Use Firestore Native Mode
    - Add location Nam5 (United States)
  + Solution
    - Navigation > Firestore > Select Native Mode > location "nam5" (United States) multi-region near.
    - Click the Create Database button. Wait for the database to be created.
* Task 2: Populate the Database
  + In this scenario, populate the database using test data.
  + A high level architecture diagram below summarizes the general architecture.
    - Graphical user interface, application, Teams

      Description automatically generated
  + Populate the Database
    - Example Firestore schema

|  |  |  |
| --- | --- | --- |
| **Collection** | **Document** | **Field** |
| data | 70234439 | [dataset] |

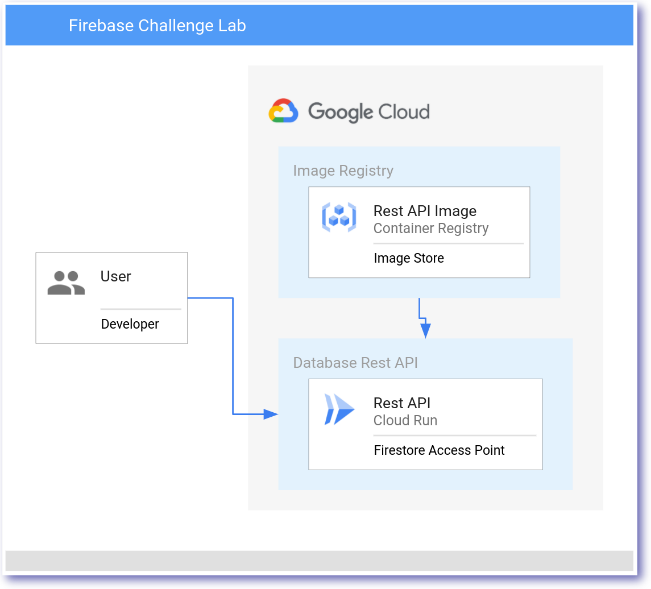
* + - Netflix Shows Dataset includes the following information <https://www.kaggle.com/shivamb/netflix-shows>

|  |  |
| --- | --- |
| **Field** | **Description** |
| show\_id: | Unique ID for every Movie / Tv Show |
| type: | Identifier - A Movie or TV Show |
| title: | Title of the Movie / Tv Show |
| director: | Director of the Movie |
| cast: | Actors involved in the movie / show |
| country: | Country where the movie / show was produced |
| date\_added: | Date it was added on Netflix |
| release\_year: | Actual Release year of the move / show |
| rating: | TV Rating of the movie / show |
| duration: | Total Duration - in minutes or number of seasons |

* + - To complete this section successfully, you are required to implement the following tasks:
    - Use the sample code from pet-theory/lab06/firebase-import-csv/solution
    - npm install
    - To import CSV use the node pet-theory/lab06/firebase-import-csv/solution/index.js
    - node index.js netflix\_titles\_original.csv
    - Verify the Firestore Database has been updated by viewing the data in the Firestore UI.
  + Solution
    - cd ~/pet-theory/lab06/firebase-import-csv/solution
    - npm install
    - node index.js netflix\_titles\_original.csv
* Task 3: Create a REST API
  + In this scenario, create an example REST API.
  + A high level architecture diagram below summarizes the general architecture.
    - Graphical user interface, application

      Description automatically generated
  + Cloud Run Development

|  |  |
| --- | --- |
| **Field** | **Value** |
| Container Registry Image | rest-api:0.1 |
| Cloud Run Service | Dataset Service Name |
| Permission | --allow-unauthenticated |

* + To complete this section successfully, you are required to implement the following tasks:
  + Access pet-theory/lab06/firebase-rest-api/solution-01
  + Build and Deploy the code to Google Container Registry
  + Deploy the image as a Cloud Run Service
  + NOTE: Deploy your service with 1 max instance to ensure you do not exceed max limit for Cloud Run instances.
  + Goto cloud run and click netflix-dataset-service-684 then copy the service URL
    - SERVICE\_URL=copy url from your netflix-dataset-service-684
    - curl -X GET $SERVICE\_URL should respond with:
  + {"status":"Netflix Dataset! Make a query."}
  + Solution
    - cd ~/pet-theory/lab06/firebase-rest-api/solution-01
    - npm install
    - gcloud builds submit --tag gcr.io/$GOOGLE\_CLOUD\_PROJECT/rest-api:0.1
    - gcloud beta run deploy netflix-dataset-service-684 --image gcr.io/$GOOGLE\_CLOUD\_PROJECT/rest-api:0.1 --platform managed --region us-central1 --allow-unauthenticated --max-instances=1
    - SERVICE\_URL=$(gcloud beta run services describe netflix-dataset-service-684 --platform managed --region us-central1 --format "value(status.url)")
    - curl -X GET $SERVICE\_URL
* Task 4: Firestore API access
  + In this scenario, deploy an updated revision of the code to access the Firestore DB.
  + A high level architecture diagram below summarizes the general architecture.
    - 
  + Deploy Cloud Run revision 0.2

|  |  |
| --- | --- |
| **Field** | **Value** |
| Container Registry Image | rest-api:0.2 |
| Cloud Run Service | netflix-dataset-service-684 |
| Permission | --allow-unauthenticated |

* + To complete this section successfully, you are required to implement the following tasks:
  + Access pet-theory/lab06/firebase-rest-api/solution-02
  + Build the updated application
  + Use Cloud Build to tag and deploy image revision to Container Registry
  + Deploy the new image as Cloud Run service
  + NOTE: Deploy your service with 1 max instance to ensure you do not exceed max limit for Cloud Run instances.
  + Goto cloud run and click netflix-dataset-service-684 then copy the service URL
    - SERVICE\_URL=copy url from your netflix-dataset-service-684
    - curl -X GET $SERVICE\_URL/2019 should respond with json dataset
  + Solution
    - cd ~/pet-theory/lab06/firebase-rest-api/solution-02
    - npm install
    - gcloud builds submit --tag gcr.io/$GOOGLE\_CLOUD\_PROJECT/rest-api:0.2
    - gcloud beta run deploy netflix-dataset-service-684 --image gcr.io/$GOOGLE\_CLOUD\_PROJECT/rest-api:0.2 --platform managed --region us-central1 --allow-unauthenticated --max-instances=1
    - SERVICE\_URL=$(gcloud beta run services describe netflix-dataset-service-684 --platform managed --region us-central1 --format "value(status.url)")
    - curl -X GET $SERVICE\_URL/2019
* Task 5: Deploy the Staging Frontend
  + In this scenario, deploy the Staging Frontend.
  + A high level architecture diagram below summarizes the general architecture.
    - Graphical user interface, application, Word, PowerPoint

      Description automatically generated
  + Deploy Frontend

|  |  |
| --- | --- |
| **Field** | **Value** |
| REST\_API\_SERVICE | REST API SERVICE URL |
| Container Registry Image | frontend-staging:0.1 |
| Cloud Run Service | Frontend Staging Service Name |

* + To complete this section successfully, you are required to implement the following tasks:
    - Access pet-theory/lab06/firebase-frontend
    - Build the frontend staging application
    - Use Cloud Build to tag and deploy image revision to Container Registry
    - Deploy the new image as Cloud Run service
    - NOTE: Deploy your service with 1 max instance to ensure you do not exceed max limit for Cloud Run instances.
    - Frontend access to Rest API and Firestore Database
  + Access the Frontend Service URL.
  + Note: It's using a demo dataset to provide the onscreen entries
    - Graphical user interface, application

      Description automatically generated
  + Solution
    - cd ~/pet-theory/lab06/firebase-frontend
    - npm install
    - gcloud builds submit --tag gcr.io/$GOOGLE\_CLOUD\_PROJECT/frontend-staging:0.1
    - gcloud beta run deploy frontend-staging-service-774 --image gcr.io/$GOOGLE\_CLOUD\_PROJECT/frontend-staging:0.1 --platform managed --region us-central1 --allow-unauthenticated --max-instances=1
    - SERVICE\_URL=$(gcloud beta run services describe frontend-staging-service-774 --platform managed --region us-central1 --format "value(status.url)")
    - curl -X GET $SERVICE\_URL
    - click the $SERVICE\_URL link to see web site
* Task 6: Deploy the Production Frontend
  + In this scenario, update the Staging Frontend to use the Firestore database.
  + A high level architecture diagram below summarizes the general architecture.
    - Graphical user interface, application, Word

      Description automatically generated
  + Deploy Frontend

|  |  |
| --- | --- |
| **Field** | **Value** |
| REST\_API\_SERVICE | REST API SERVICE URL |
| Container Registry Image | frontend-production:0.1 |
| Cloud Run Service | Frontend Production Service Name |

* + To complete this section successfully, you are required to implement the following tasks:
    - Access pet-theory/lab06/firebase-frontend/public
    - Update the frontend application i.e. app.js to use the REST API
    - Don't forget to append the year to the SERVICE\_URL
    - Use Cloud Build to tag and deploy image revision to Container Registry
    - Deploy the new image as Cloud Run service
    - NOTE: Deploy your service with 1 max instance to ensure you do not exceed max limit for Cloud Run instances.
    - Frontend access to Rest API and Firestore Database
  + Now that the services have been deployed you will be able to see the contents of the Firestore database using the frontend service.
    - Table

      Description automatically generated
  + Solution
    - edit app.js add REST API SERVICE URL un comment
    - pet-theory/lab06/firebase-frontend/public/app.js
    - const REST\_API\_SERVICE = "https://XXXX-SERVICE.run.app/2020"
    - change to url from task 4
    - const REST\_API\_SERVICE = "https://netflix-dataset-service-684-4hkezjuzlq-uc.a.run.app/2020"
    - gcloud beta builds submit --tag gcr.io/$GOOGLE\_CLOUD\_PROJECT/frontend-production:0.1
    - gcloud beta run deploy frontend-production-service-237 --image gcr.io/$GOOGLE\_CLOUD\_PROJECT/frontend-production:0.1 --platform managed --region us-central1 --allow-unauthenticated --max-instances=1
    - SERVICE\_URL=$(gcloud beta run services describe frontend-production-service-237 --platform managed --region us-central1 --format "value(status.url)")
    - curl -X GET $SERVICE\_URL

## 10. Deploy to Kubernetes in Google Cloud

* learn about Google Kubernetes Engine and multiple deployment approaches including how to: configure and build images to run and debug Docker containers, build Kubernetes Engine clusters, and manage them with kubectl, deploy Kubernetes applications using deployments and continuous delivery techniques.

## 10.1. Introduction to Docker

* Overview
  + Docker is an open platform for developing, shipping, and running applications. With Docker, you can separate your applications from your infrastructure and treat your infrastructure like a managed application. Docker helps you ship code faster, test faster, deploy faster, and shorten the cycle between writing code and running code.
  + Docker does this by combining kernel containerization features with workflows and tooling that helps you manage and deploy your applications.
  + Docker containers can be directly used in Kubernetes, which allows them to be run in the Kubernetes Engine with ease. After learning the essentials of Docker, you will have the skillset to start developing Kubernetes and containerized applications.
* What you'll learn
  + How to build, run, and debug Docker containers.
  + How to pull Docker images from Docker Hub and Google Container Registry.
  + How to push Docker images to Google Container Registry.
* Hello World
  + run a hello world container to get started:
    - docker run hello-world
  + (Command Output)
    - Unable to find image 'hello-world:latest' locally
    - latest: Pulling from library/hello-world
    - …
    - Status: Downloaded newer image for hello-world:latest
    - Hello from Docker!
  + This simple container returns Hello from Docker! to your screen. While the command is simple, notice in the output the number of steps it performed. The docker daemon searched for the hello-world image, didn't find the image locally, pulled the image from a public registry called Docker Hub, created a container from that image, and ran the container for you.
  + Run the following command to take a look at the container image it pulled from Docker Hub:
    - docker images
      * REPOSITORY TAG IMAGE ID CREATED SIZE
      * hello-world latest 1815c82652c0 6 days ago 1.84 kB
  + This is the image pulled from the Docker Hub public registry. The Image ID is in SHA256 hash format—this field specifies the Docker image that's been provisioned. When the docker daemon can't find an image locally, it will by default search the public registry for the image. Let's run the container again:
    - docker run hello-world
  + Notice the second time you run this, the docker daemon finds the image in your local registry and runs the container from that image. It doesn't have to pull the image from Docker Hub.
  + Finally, look at the running containers by running the following command:
    - docker ps
      * CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
  + There are no running containers. The hello-world containers you ran previously already exited. In order to see all containers, including ones that have finished executing, run docker ps -a:
    - docker ps -a
      * CONTAINER ID IMAGE COMMAND ... NAMES
      * 6027ecba1c39 hello-world "/hello" ... elated\_knuth
      * 358d709b8341 hello-world "/hello" ... epic\_lewin
  + This shows you the Container ID, a UUID generated by Docker to identify the container, and more metadata about the run. The container Names are also randomly generated but can be specified with docker run --name [container-name] hello-world.
* Build
  + Let's build a Docker image that's based on a simple node application. Execute the following command to create and switch into a folder named test.
    - mkdir test && cd test
  + Create a Dockerfile:
    - cat > Dockerfile <<EOF
    - # Use an official Node runtime as the parent image
    - FROM node:6
    - # Set the working directory in the container to /app
    - WORKDIR /app
    - # Copy the current directory contents into the container at /app
    - ADD . /app
    - # Make the container's port 80 available to the outside world
    - EXPOSE 80
    - # Run app.js using node when the container launches
    - CMD ["node", "app.js"]
    - EOF
  + This file instructs the Docker daemon on how to build your image.
    - The initial line specifies the base parent image, which in this case is the official Docker image for node version 6.
    - In the second, we set the working (current) directory of the container.
    - In the third, we add the current directory's contents (indicated by the "." ) into the container.
    - Then we expose the container's port so it can accept connections on that port and finally run the node command to start the application.
  + Spend some time reviewing the Dockerfile command references to understand each line of the Dockerfile.
  + Now you'll write the node application, and after that you'll build the image.
  + Run the following to create the node application:
    - cat > app.js <<EOF
    - const http = require('http');
    - const hostname = '0.0.0.0';
    - const port = 80;
    - const server = http.createServer((req, res) => {
    - res.statusCode = 200;
    - res.setHeader('Content-Type', 'text/plain');
    - res.end('Hello World\n');
    - });
    - server.listen(port, hostname, () => {
    - console.log('Server running at http://%s:%s/', hostname, port);
    - });
    - process.on('SIGINT', function() {
    - console.log('Caught interrupt signal and will exit');
    - process.exit();
    - });
    - EOF
  + This is a simple HTTP server that listens on port 80 and returns "Hello World".
  + Now let's build the image.
  + Note again the ".", which means current directory so you need to run this command from within the directory that has the Dockerfile:
    - docker build -t node-app:0.1 .
  + The -t is to name and tag an image with the name:tag syntax. The name of the image is node-app and the tag is 0.1. The tag is highly recommended when building Docker images. If you don't specify a tag, the tag will default to latest and it becomes more difficult to distinguish newer images from older ones. Also notice how each line in the Dockerfile above results in intermediate container layers as the image is built.
  + Now, run the following command to look at the images you built:
    - docker images
      * REPOSITORY TAG IMAGE ID CREATED SIZE
      * node-app 0.1 f166cd2a9f10 25 seconds ago 656.2 MB
      * node 6 5a767079e3df 15 hours ago 656.2 MB
      * hello-world latest 1815c82652c0 6 days ago 1.84 kB
  + Notice node is the base image and node-app is the image you built. You can't remove node without removing node-app first. The size of the image is relatively small compared to VMs. Other versions of the node image such as node:slim and node:alpine can give you even smaller images for easier portability. The topic of slimming down container sizes is further explored in Advanced Topics. You can view all versions in the official repository here.
* Run
  + In this module, use this code to run containers based on the image you built:
    - docker run -p 4000:80 --name my-app node-app:0.1
  + The --name flag allows you to name the container if you like. The -p instructs Docker to map the host's port 4000 to the container's port 80. Now you can reach the server at http://localhost:4000. Without port mapping, you would not be able to reach the container at localhost.
  + Open another terminal (in Cloud Shell, click the + icon), and test the server:
    - curl http://localhost:4000
      * Hello World
  + The container will run as long as the initial terminal is running. If you want the container to run in the background (not tied to the terminal's session), you need to specify the -d flag.
  + Close the initial terminal and then run the following command to stop and remove the container:
    - docker stop my-app && docker rm my-app
  + Now run the following command to start the container in the background:
    - docker run -p 4000:80 --name my-app -d node-app:0.1
    - docker ps
      * CONTAINER ID IMAGE COMMAND CREATED ... NAMES
      * xxxxxxxxxxxx node-app:0.1 "node app.js" 16 seconds ago ... my-app
  + Notice the container is running in the output of docker ps. You can look at the logs by executing docker logs [container\_id].
  + Tip: You don't have to write the entire container ID, as long as the initial characters uniquely identify the container. For example, you can execute docker logs 17b if the container ID is 17bcaca6f....
    - docker logs [container\_id]
      * Server running at http://0.0.0.0:80/
  + Let's modify the application. In your Cloud Shell, open the test directory you created earlier in the lab:
    - cd test
  + Edit app.js with a text editor of your choice (for example nano or vim) and replace "Hello World" with another string:
    - ....
    - const server = http.createServer((req, res) => {
    - res.statusCode = 200;
    - res.setHeader('Content-Type', 'text/plain');
    - res.end('Welcome to Cloud\n');
    - });
    - ....
  + Build this new image and tag it with 0.2:
    - docker build -t node-app:0.2 .
  + Notice in Step 2 we are using an existing cache layer. From Step 3 and on, the layers are modified because we made a change in app.js.
  + Run another container with the new image version. Notice how we map the host's port 8080 instead of 80. We can't use host port 4000 because it's already in use.
    - docker run -p 8080:80 --name my-app-2 -d node-app:0.2
    - docker ps
      * CONTAINER ID IMAGE COMMAND CREATED
      * xxxxxxxxxxxx node-app:0.2 "node app.js" 53 seconds ago ...
      * xxxxxxxxxxxx node-app:0.1 "node app.js" About an hour ago ...
  + Test the containers:
    - curl http://localhost:8080
      * Welcome to Cloud
  + And now test the first container you made:
    - curl http://localhost:4000
* Debug
  + Now that we're familiar with building and running containers, let's go over some debugging practices.
  + You can look at the logs of a container using docker logs [container\_id]. If you want to follow the log's output as the container is running, use the -f option.
    - docker logs -f [container\_id]
      * Server running at http://0.0.0.0:80/
  + Sometimes you will want to start an interactive Bash session inside the running container. You can use docker exec to do this. Open another terminal (in Cloud Shell, click the + icon) and enter the following command:
    - docker exec -it [container\_id] bash
  + The -it flags let you interact with a container by allocating a pseudo-tty and keeping stdin open. Notice bash ran in the WORKDIR directory (/app) specified in the Dockerfile. From here, you have an interactive shell session inside the container to debug.
  + (Command Output)
    - root@xxxxxxxxxxxx:/app#
  + Look at the directory
    - ls
      * Dockerfile app.js
  + Exit the Bash session:
    - exit
  + You can examine a container's metadata in Docker by using Docker inspect:
    - docker inspect [container\_id]
  + (Command Output)
    - [
    - {
    - "Id": "xxxxxxxxxxxx....",
    - "Created": "2017-08-07T22:57:49.261726726Z",
    - "Path": "node",
    - "Args": [
    - "app.js"
    - ],
    - ...
  + Use --format to inspect specific fields from the returned JSON. For example:
    - docker inspect --format='{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' [container\_id]
  + (Example Output)
    - 192.168.9.3
  + Be sure to check out the following resources for more information on debugging:
    - Docker inspect reference
  + Docker exec reference
* Publish
  + Now you're going to push your image to the Google Container Registry (gcr). After that you'll remove all containers and images to simulate a fresh environment, and then pull and run your containers. This will demonstrate the portability of Docker containers.
  + To push images to your private registry hosted by gcr, you need to tag the images with a registry name. The format is [hostname]/[project-id]/[image]:[tag].
  + For gcr:
    - [hostname]= gcr.io
    - [project-id]= your project's ID
    - [image]= your image name
    - [tag]= any string tag of your choice. If unspecified, it defaults to "latest".
  + You can find your project ID by running:
    - gcloud config list project
  + Tag node-app:0.2. Replace [project-id] with your configuration..
    - docker tag node-app:0.2 gcr.io/[project-id]/node-app:0.2
    - docker images
  + (Command Output)
    - REPOSITORY TAG IMAGE ID CREATED
    - node-app 0.2 76b3beef845e 22 hours ago
    - gcr.io/[project-id]/node-app 0.2 76b3beef845e 22 hours ago
    - node-app 0.1 f166cd2a9f10 26 hours ago
    - node 6 5a767079e3df 7 days ago
    - hello-world latest 1815c82652c0 7 weeks ago
  + Push this image to gcr. Remember to replace [project-id].
    - docker push gcr.io/[project-id]/node-app:0.2
  + Check that the image exists in gcr, Navigation > Container Registry > node-app or visit: http://gcr.io/[project-id]/node-app.:
  + Let's test this image. You could start a new VM, ssh into that VM, and install gcloud. For simplicity, we'll just remove all containers and images to simulate a fresh environment.
  + Stop and remove all containers:
    - docker stop $(docker ps -q)
    - docker rm $(docker ps -aq)
  + You have to remove the child images (of node:6) before you remove the node image. Replace [project-id].
    - docker rmi node-app:0.2 gcr.io/[project-id]/node-app node-app:0.1
    - docker rmi node:6
    - docker rmi $(docker images -aq) # remove remaining images
    - docker images
  + (Command Output)
    - REPOSITORY TAG IMAGE ID CREATED SIZE
  + At this point you should have a pseudo-fresh environment. Pull the image and run it.
    - docker pull gcr.io/[project-id]/node-app:0.2
    - docker run -p 4000:80 -d gcr.io/[project-id]/node-app:0.2
    - curl http://localhost:4000
  + (Command Output)
    - Welcome to Cloud

## 10.2. Kubernetes Engine: Qwik Start – Refer Cloud Architect

* Overview
  + Google Kubernetes Engine (GKE) provides a managed environment for deploying, managing, and scaling your containerized applications using Google infrastructure. The Kubernetes Engine environment consists of multiple machines (specifically Compute Engine instances) grouped to form a container cluster. In this lab, you get hands-on practice with container creation and application deployment with GKE.
* Cluster orchestration with Google Kubernetes Engine
* Google Kubernetes Engine (GKE) clusters are powered by the Kubernetes open source cluster management system. Kubernetes provides the mechanisms through which you interact with your container cluster. You use Kubernetes commands and resources to deploy and manage your applications, perform administrative tasks, set policies, and monitor the health of your deployed workloads.
* Kubernetes draws on the same design principles that run popular Google services and provides the same benefits: automatic management, monitoring and liveness probes for application containers, automatic scaling, rolling updates, and more. When you run your applications on a container cluster, you're using technology based on Google's 10+ years of experience with running production workloads in containers.
* Kubernetes on Google Cloud
* When you run a GKE cluster, you also gain the benefit of advanced cluster management features that Google Cloud provides. These include:
* Load balancing for Compute Engine instances
* Node pools to designate subsets of nodes within a cluster for additional flexibility
* Automatic scaling of your cluster's node instance count
* Automatic upgrades for your cluster's node software
* Node auto-repair to maintain node health and availability
* Logging and Monitoring with Cloud Monitoring for visibility into your cluster
* Now that you have a basic understanding of Kubernetes, you will learn how to deploy a containerized application with GKE in less than 30 minutes. Follow the steps below to set up your lab environment.
* Setup and requirements
* Before you click the Start Lab button
* Read these instructions. Labs are timed and you cannot pause them. The timer, which starts when you click Start Lab, shows how long Google Cloud resources will be made available to you.
* This hands-on lab lets you do the lab activities yourself in a real cloud environment, not in a simulation or demo environment. It does so by giving you new, temporary credentials that you use to sign in and access Google Cloud for the duration of the lab.
* What you need
* To complete this lab, you need:
* Access to a standard internet browser (Chrome browser recommended).
* Time to complete the lab.
* Note: If you already have your own personal Google Cloud account or project, do not use it for this lab.
* Note: If you are using a Chrome OS device, open an Incognito window to run this lab.
* How to start your lab and sign in to the Google Cloud Console
* Click the Start Lab button. If you need to pay for the lab, a pop-up opens for you to select your payment method. On the left is a panel populated with the temporary credentials that you must use for this lab.
* Open Google Console
* Copy the username, and then click Open Google Console. The lab spins up resources, and then opens another tab that shows the Sign in page.
* Sign in
* Tip: Open the tabs in separate windows, side-by-side.
* If you see the Choose an account page, click Use Another Account. Choose an account
* In the Sign in page, paste the username that you copied from the left panel. Then copy and paste the password.
* Important: You must use the credentials from the left panel. Do not use your Google Cloud Training credentials. If you have your own Google Cloud account, do not use it for this lab (avoids incurring charges).
* Click through the subsequent pages:
* Accept the terms and conditions.
* Do not add recovery options or two-factor authentication (because this is a temporary account).
* Do not sign up for free trials.
* After a few moments, the Cloud Console opens in this tab.
* Note: You can view the menu with a list of Google Cloud Products and Services by clicking the Navigation menu at the top-left. Cloud Console Menu
* Activate Cloud Shell
* Cloud Shell is a virtual machine that is loaded with development tools. It offers a persistent 5GB home directory and runs on the Google Cloud. Cloud Shell provides command-line access to your Google Cloud resources.
* In the Cloud Console, in the top right toolbar, click the Activate Cloud Shell button.
* Cloud Shell icon
* Click Continue.
* cloudshell\_continue.png
* It takes a few moments to provision and connect to the environment. When you are connected, you are already authenticated, and the project is set to your PROJECT\_ID. For example:
* Cloud Shell Terminal
* gcloud is the command-line tool for Google Cloud. It comes pre-installed on Cloud Shell and supports tab-completion.
* You can list the active account name with this command:
* gcloud auth list
* Copied!
* (Output)
* ACTIVE: \*
* ACCOUNT: student-01-xxxxxxxxxxxx@qwiklabs.net
* To set the active account, run:
* $ gcloud config set account `ACCOUNT`
* You can list the project ID with this command:
* gcloud config list project
* Copied!
* (Output)
* [core]
* project = <project\_ID>
* (Example output)
* [core]
* project = qwiklabs-gcp-44776a13dea667a6
* For full documentation of gcloud see the gcloud command-line tool overview.
* Task 1: Set a default compute zone
* Your compute zone is an approximate regional location in which your clusters and their resources live. For example, us-central1-a is a zone in the us-central1 region.
* To set your default compute zone to us-central1-a, start a new session in Cloud Shell, and run the following command:
* gcloud config set compute/zone us-central1-a
* Copied!
* Expected output (Do not copy):
* Updated property [compute/zone].
* Task 2: Create a GKE cluster
* A cluster consists of at least one cluster master machine and multiple worker machines called nodes. Nodes are Compute Engine virtual machine (VM) instances that run the Kubernetes processes necessary to make them part of the cluster.
* Note: Cluster names must start with a letter and end with an alphanumeric, and cannot be longer than 40 characters.
* To create a cluster, run the following command, replacing [CLUSTER-NAME] with the name you choose for the cluster (for example:my-cluster).
* gcloud container clusters create [CLUSTER-NAME]
* Copied!
* You can ignore any warnings in the output. It might take several minutes to finish creating the cluster.
* Expected output (Do not copy):
* NAME: my-cluster
* LOCATION: us-central1-a
* MASTER\_VERSION: 1.21.5-gke.1302
* MASTER\_IP: 34.69.232.119
* MACHINE\_TYPE: e2-medium
* NODE\_VERSION: 1.21.5-gke.1302
* NUM\_NODES: 3
* STATUS: RUNNING
* Click Check my progress to verify the objective.
* Assessment Completed!
* Create a GKE cluster
* Assessment Completed!
* Task 3: Get authentication credentials for the cluster
* After creating your cluster, you need authentication credentials to interact with it.
* To authenticate the cluster, run the following command, replacing [CLUSTER-NAME] with the name of your cluster:
* gcloud container clusters get-credentials [CLUSTER-NAME]
* Copied!
* Expected output (Do not copy):
* Fetching cluster endpoint and auth data.
* kubeconfig entry generated for my-cluster.
* Task 4: Deploy an application to the cluster
* You can now deploy a containerized application to the cluster. For this lab, you'll run hello-app in your cluster.
* GKE uses Kubernetes objects to create and manage your cluster's resources. Kubernetes provides the Deployment object for deploying stateless applications like web servers. Service objects define rules and load balancing for accessing your application from the internet.
* To create a new Deployment hello-server from the hello-app container image, run the following kubectl create command:
* kubectl create deployment hello-server --image=gcr.io/google-samples/hello-app:1.0
* Copied!
* Expected output (Do not copy):
* deployment.apps/hello-server created
* This Kubernetes command creates a Deployment object that represents hello-server. In this case, --image specifies a container image to deploy. The command pulls the example image from a Container Registry bucket. gcr.io/google-samples/hello-app:1.0 indicates the specific image version to pull. If a version is not specified, the latest version is used.
* Click Check my progress to verify the objective.
* Assessment Completed!
* Create a new Deployment: hello-server
* Assessment Completed!
* To create a Kubernetes Service, which is a Kubernetes resource that lets you expose your application to external traffic, run the following kubectl expose command:
* kubectl expose deployment hello-server --type=LoadBalancer --port 8080
* Copied!
* In this command:
* --port specifies the port that the container exposes.
* type="LoadBalancer" creates a Compute Engine load balancer for your container.
* Expected output (Do not copy):
* service/hello-server exposed
* To inspect the hello-server Service, run kubectl get:
* kubectl get service
* Copied!
* Expected output (Do not copy):
* NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
* hello-server loadBalancer 10.39.244.36 35.202.234.26 8080:31991/TCP 65s
* kubernetes ClusterIP 10.39.240.1 <none> 433/TCP 5m13s
* Note: It might take a minute for an external IP address to be generated. Run the previous command again if the EXTERNAL-IP column status is pending.
* To view the application from your web browser, open a new tab and enter the following address, replacing [EXTERNAL IP] with the EXTERNAL-IP for hello-server.
* http://[EXTERNAL-IP]:8080
* Copied!
* Expected output:
* output.png
* Click Check my progress to verify the objective.
* Assessment Completed!
* Create a Kubernetes Service
* Assessment Completed!
* Task 5: Deleting the cluster
* To delete the cluster, run the following command:
* gcloud container clusters delete [CLUSTER-NAME]
* Copied!
* When prompted, type Y to confirm.
* Deleting the cluster can take a few minutes. For more information on deleted GKE clusters, view the documentation.

## 10.3. Orchestrating the Cloud with Kubernetes - Refer Cloud Architect

## 10.4. Managing Deployments Using Kubernetes Engine – Refer Cloud Architect

## 10.5. Continuous Delivery with Jenkins in Kubernetes Engine

* Lab

## 10.6. Deploy to Kubernetes in Google Cloud: Challenge Lab

* This challenge lab tests your skills and knowledge from the labs in the Kubernetes in Google Cloud quest. You should be familiar with the content of the labs before attempting this lab.