## 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.

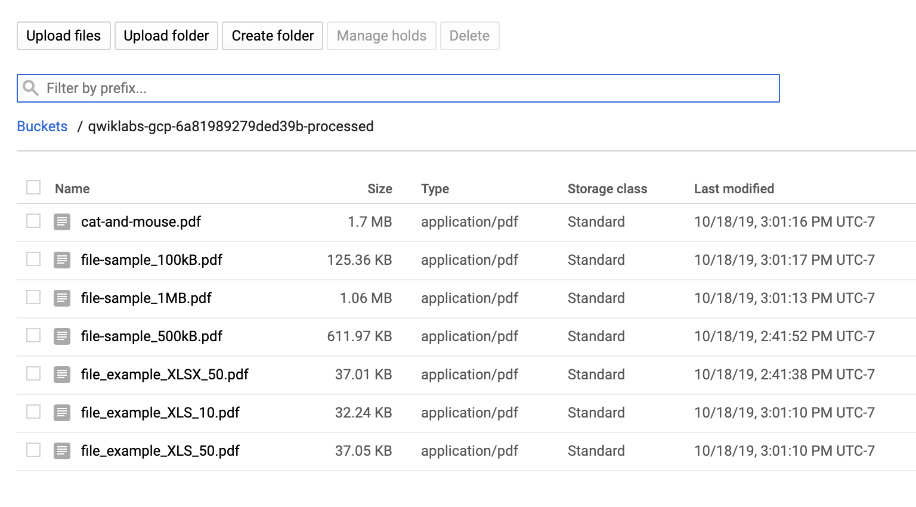
## 8.1 Build a Serverless App with Cloud Run that Creates PDF Files

* 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

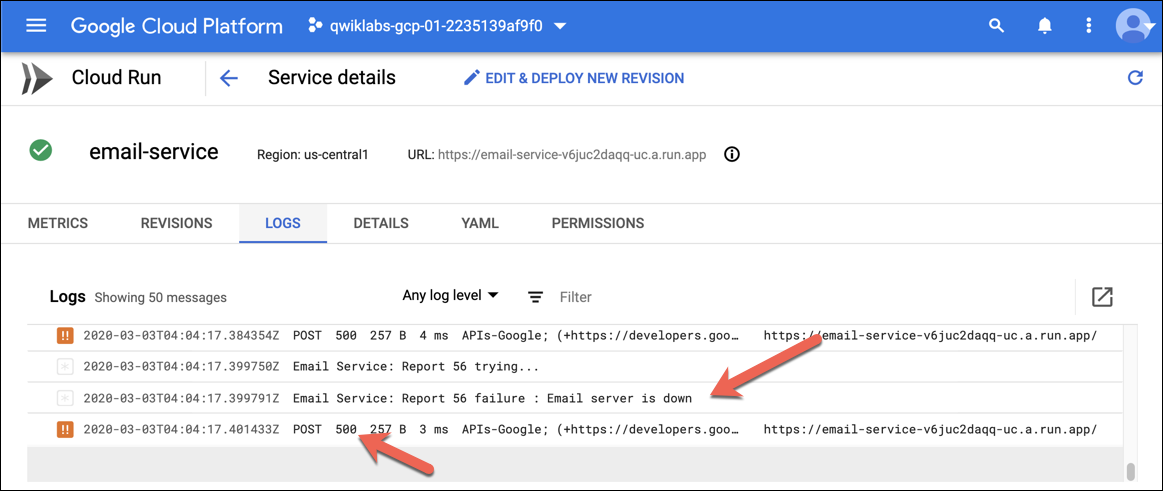
      Description automatically generated
  + This service will serve the purpose of prototyping, so it will only do two things:
    - Receive the lab report HTTPS POST containing the report data.
    - Publish a message on Pub/Sub.
* Add code for the Lab Report Service
  + Back in Cloud Shell, clone the repository 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