## **Provisioning Auto-Import Google Cloud Resources Using Terraform**

### **Purpose**

This document provides step-by-step instructions for automating the provisioning of Google Cloud resources to create a pipeline that automatically imports files placed in a Google Bucket into a FHIR store.

### **Requirements**

A Google Cloud project

A Google Cloud service account with necessary credentials

An IDE (I used VSCode)

### **Procedure**

#### **Step 1: Acquire Service Account Credentials**

Before we can create a Terraform script to create the required resources, we have to obtain the credentials of a Service Account that has the required permissions. To do this, we will create a new access key.

1. First, navigate to the Service Accounts section of Google Cloud, then select the Service Account you would like to use.
2. Under the Keys section, click Add Key and select Create new key.
3. For this project, we will use the credentials in JSON format, so select that option and click Create
4. This will download the credentials to your local device.

#### **Step 2: Setup Environment and Install Terraform**

Create a folder wherever you want your workspace to exist. Subsequent files in this tutorial should be placed in this folder. In my code, I have the JSON file named **credentials.json**. Also in this folder, create files named **script.tf** for the Terraform script and **main.py** as the Cloud Function source code.

You will also need to [install Terraform on your machine](https://developer.hashicorp.com/terraform/tutorials/aws-get-started/install-cli), and ensure the **terraform** binary file is on your **PATH**. In this tutorial, I use Terraform v1.4.4.

#### **Step 3: Setup Cloud Function**

In your **main.py**, give the source code for your Cloud Function. In this tutorial, we will be creating a function (shown in more detail in **Google Cloud Auto-Import Cloud Function**)that triggers when a file is added to a certain bucket, and will then export the file from that bucket into a FHIR Store provisioned by our Terraform script.

import os

from googleapiclient import discovery

def hello\_gcs(event, context):

print(f"Processing file: {event['name']}.")

# Create client to interact with GC

client = discovery.build("healthcare", "v1")

# Get names of bucket and FHIR store

fhir\_store\_id = os.environ['FHIR\_STORE\_ID']

gcs\_uri = os.environ['GCS\_URI']

# Format of FHIR file

content\_format = "BUNDLE\_PRETTY"

body = {

"contentStructure": content\_format,

"gcsSource": {"uri": gcs\_uri},

}

request = (

client.projects()

.locations()

.datasets()

.fhirStores()

.import\_(name=fhir\_store\_id, body=body)

)

response = request.execute()

print("Imported FHIR resources: {}".format(gcs\_uri))

return response

#### **Step 4: Deploying Google Cloud Buckets in Terraform**

Here, we will define the resources we wish to be provisioned every time the script is run.

To start, we declare which platform (provider) we want the resources to be provisioned.

provider "google" {

project = "YOUR\_PROJECT\_ID"

region = "REGION"

credentials = "credentials.json"

}

Replace **YOUR\_PROJECT\_ID** with the ID of your project (i.e. involuted-woods-360619) and **REGION** with the region of your project (i.e. us-east1). The credentials variable is accessing the JSON we downloaded earlier.

Next, we’ll take care of the Cloud Function by deploying a bucket that contains its source code.

resource "google\_storage\_bucket" "auto\_export\_function\_source" {

location = "us-east1"

name = "auto\_export\_function\_source"

force\_destroy = true

}

Here, we’re creating a bucket with the name **auto-export-function-source**, and we’re telling Terraform to reprovision this bucket every time the script is run with **force\_destroy = true**.

Now, we’ll automatically zip our Cloud Function source code (this is the format a Cloud Function provisioned through a bucket is expecting), and add it as a resource to the **auto-export-function-source** bucket.

data "archive\_file" "main" {

type = "zip"

source\_file = "${path.module}/main.py"

output\_path = "${path.module}/main.zip"

}

resource "google\_storage\_bucket\_object" "function\_object" {

name = "function\_object"

bucket = google\_storage\_bucket.auto\_export\_function\_source.name

source = "main.zip"

}

Our zipped file is named **function\_object**.

Finally, we’ll deploy our last bucket. This bucket is the one that accepts files in the FHIR format and transfers them into our FHIR Store.

resource "google\_storage\_bucket" "auto\_export\_source\_bucket" {

location = "us-east1"

name = "auto\_export\_source\_bucket"

force\_destroy = true

}

#### **Step 5: Deploying Datasets, FHIR Stores and Cloud Functions in Terraform**

Now, we’ll ensure that we have a FHIR Store to place our files in, and finally deploy our Cloud Function connecting the whole pipeline.

resource "google\_healthcare\_dataset" "auto\_export\_dataset" {

name = "auto\_export\_dataset"

location = "us-east1"

}

resource "google\_healthcare\_fhir\_store" "auto\_export\_fhir\_store" {

dataset = google\_healthcare\_dataset.auto\_export\_dataset.id

name = "auto\_export\_fhir\_store"

version = "R4"

}

This creates a Dataset to store the FHIR Store, and finally the FHIR Store itself.

Lastly, the most complicated resource we provision in this tutorial: the Cloud Function itself.

resource "google\_cloudfunctions\_function" "auto\_export\_function" {

name = "auto\_export\_function"

runtime = "python37"

source\_archive\_bucket = google\_storage\_bucket.auto\_export\_function\_source.name

source\_archive\_object = google\_storage\_bucket\_object.function\_object.name

entry\_point = "hello\_gcs"

event\_trigger {

event\_type = "google.storage.object.finalize"

resource = google\_storage\_bucket.auto\_export\_source\_bucket.name

}

environment\_variables = {

FHIR\_STORE\_ID = google\_healthcare\_fhir\_store.auto\_export\_fhir\_store.id

GCS\_URI = google\_storage\_bucket.auto\_export\_source\_bucket.url

}

}

Here, we specify a few things:

* The language in which our Function is written (**Python 3.7**)
* Both the bucket in which the source code is located, and the actual zipped **.py** file in that bucket
* The entry point, or name of our Function (**hello\_gcs**)
* The way the Function is triggered. In our case, we want to run our Function when an **object** in the declared **resource** is finalized, but this can be changed depending on the use case
* The **environment\_variables**. Using these, we can reference resources we created early *within* the Cloud Function itself, as utilized with the **os.environ** calls. This is the way we actually link one resource to another (namely the FHIR Store and the bucket)

#### **Step 6: Testing a Terraform Script**

To test that everything is working properly, navigate to the folder containing the Terraform script, and run **terraform init**. This prepares the working directory for subsequent Terraform commands. Next you can use **terraform plan** to see exactly which resources would be provisioned with your current script, and finally **terraform apply** to actually attempt to deploy the resources. You should be able to see on the Cloud Console the different resources slowly popping up.

To make use of the function, simply upload a FHIR file in **BUNDLE\_PRETTY** format (this can be created using Synthea) and ensure that the allocated resources in the FHIR Store reflect the size of your files.

## **Conclusion**

Provisioning Google Cloud resources with Terraform is relatively straightforward, can save a lot of time, encourages scalability, and ensures functionality directly out of the box. Following this tutorial, you should now be able to provision a complete pipeline that accepts files in FHIR format and imports them into a dynamically deployed FHIR Store.