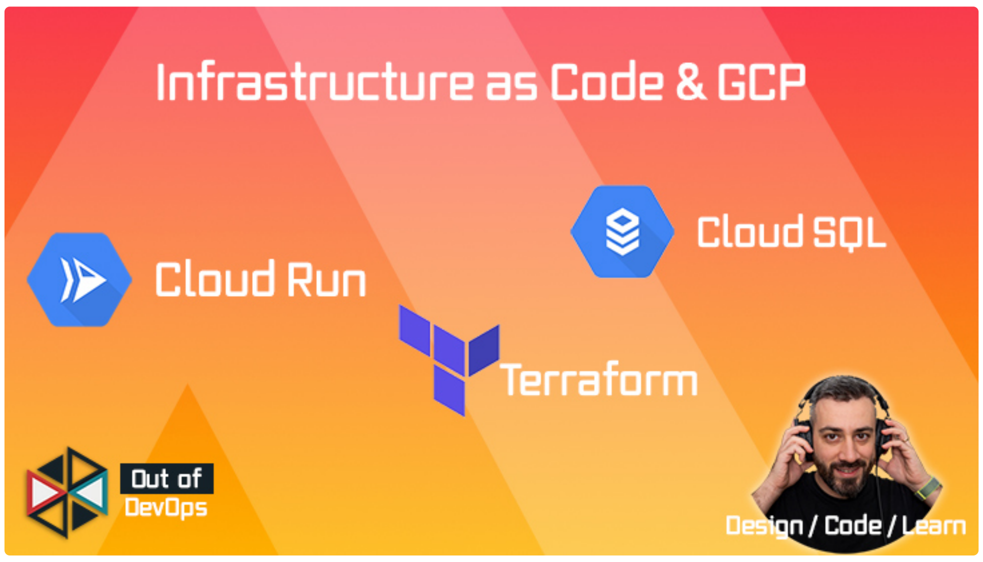
Cloud Run and IaC with Terraform

Post by Anto

Feb 03, 2021



Provision cloud run with terraform

Couple of posts back we learnt how to run containers in GCP using Cloud Run, today we are going to do exactly the same thing but this time using IaC with the help of Terraform.

Without further ado let's get started!!!

Pre requisite:

- Terraform installed (I am going to use the latest version 0.14.5)
- Admin access to a GCP project
- Gcloud installed

We are going to split the terraform code in 2 parts, and Admin Part and an App Part.

```
mkdir admin
mkdir app
```

The `admin` folder is going to contain resources that require higher privileges or not related to the app. Instead the `app` folder will contain the cloud-run resources for the deployment of the app.

Admin Implementation

Let's open our preferred editor, in my case visual studio code:

```
code .
```

In the admin folder, the resources we have to create are:

- Apis to enable (`api.tf`)
- Docker repository in artifact registry (`artifact_registry.tf`)
- A CloudSQL instance (`db.tf`)
- A DB for our app (`db.tf`)

Enabling Google Services

Now we can create 2 terraform files, let's call them vars.tf and apis.tf, respectively one for variables and another for the apis we want to enable for a specific project:

- `Artifact Registry API`
- `Cloud SQL Admin API`
- `Cloud Run Admin API`
- `Compute Engine API`
- `Cloud Resource Manager API` (needed for terraform)

So we are going to create a variable with the list of APIs in the vars.tf:

```
//admin/vars.tf

variable "apis" {
  description = "List of apis to enable"
  type        = list(string)
  default = [
    "artifactregistry.googleapis.com",
    "sqladmin.googleapis.com",
    "run.googleapis.com",
    "compute.googleapis.com",
    "cloudresourcemanager.googleapis.com" //needed by terraform
  ]
}
```

In the apis.tf instead we define the api resource:

```
// admin/apis.tf

resource "google_project_service" "apis" {
  for_each = toset(var.apis)
  service  = each.value

  disable_dependent_services = true
}
```

Here is a useful command to list all the available apis: `gcloud services list --available`

Docker Artifact Registry

Now we need to create the resource for the docker repository:

In a file called `artifact_registry.tf` we create a `google_artifact_registry_repository`

```
// admin/artifact_registry.tf

resource "google_artifact_registry_repository" "docker_ar" {
  provider = google-beta

  location      = var.region
  repository_id = "docker-ar"
  description   = "Docker repository"
  format        = "DOCKER"

  depends_on = [
    google_project_service.apis["artifactregistry.googleapis.com"],
  ]
}
```

Note the `depends_on` meta-argument, it explicitly defines a dependency between the resource and its API. Even though most of the time terraform is able to infer the dependencies between resources during creation sometimes it cannot do the same during destruction, so I prefer to be explicit.

The last set of resources instead are related to the database, in the db.tf we are going to create the database instance, the database and a user.

```
// admin/db.tf

resource "random_id" "db_name_suffix" {
  byte_length = 4
}

resource "google_sql_database_instance" "sgs" {
  name                = "sgs-${random_id.db_name_suffix.hex}"
  database_version    = "POSTGRES_12"
  deletion_protection = false #Used for demonstration purposes remove this in real scenarios
  region              = var.region

  settings {
    tier              = "db-f1-micro"
    availability_type = "ZONAL"

    backup_configuration {
      enabled            = false
      binary_log_enabled = false
    }

    ip_configuration {
      ipv4_enabled = true
    }

  }

  depends_on = [
    google_project_service.apis["sqladmin.googleapis.com"],
  ]
}

resource "google_sql_database" "sgs" {
  name     = "sgs"
  instance = google_sql_database_instance.sgs.name

  depends_on = [
    google_sql_database_instance.sgs,
  ]
}

resource "google_sql_user" "users" {
  name            = "sgs"
  instance        = google_sql_database_instance.sgs.name
  password        = var.db_password
  deletion_policy = "ABANDON"
}
```

Things to notice here are:

- the `random_id` resource
- the deletion_policy for the `google_sql_user`

When a database is created and then deleted its name cannot be reused right away (https://cloud.google.com/sql/faq?hl=en#reuse) so the `random_id` resource helps with the testing of our terraform code since we probably have to `terraform apply` and `terraform destroy` a number of time.

Last thing left is to configure our terraform providers and define the output, for that we will use a file called `[config.tf](http://config.tf)` and a file called `output.tf`

```
// admin/config.tf

terraform {
  required_providers {
    google = {
      version = "3.53.0"
      source  = "hashicorp/google"
    }
    google-beta = {
      version = "3.53.0"
      source  = "hashicorp/google-beta"
    }
    random = {
      version = "3.0.1"
      source  = "hashicorp/random"
    }
  }
}

provider "google" {
  project = var.project_id
  region  = var.region
}

provider "google-beta" {
  project = var.project_id
  region  = var.region
}
```

```
// admin/output.tf

output "db_name" {
  value = google_sql_database_instance.sgs.name
}
```

We also need to add the variables for:

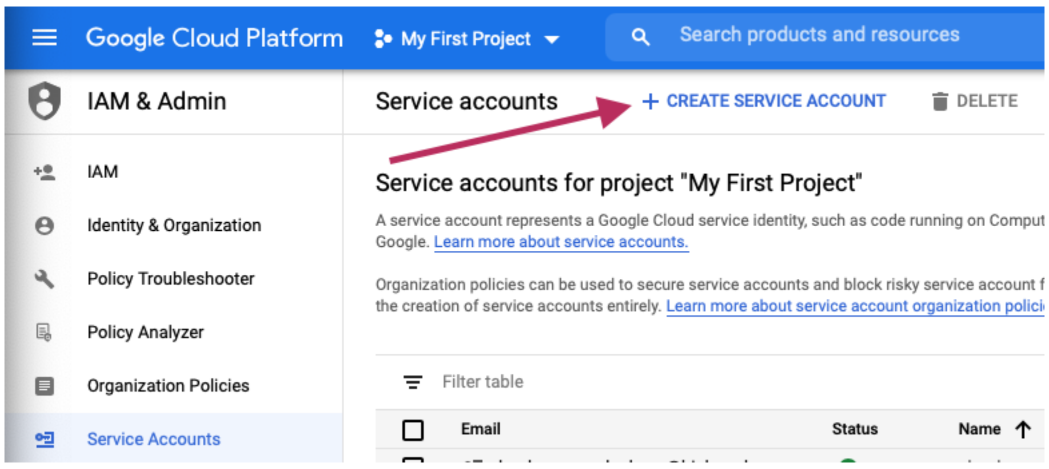
- `project_id`
- `region`
- `db_password`

```
// admin/vars.tf

variable "project_id" {
  type        = string
  description = "The GCP Project ID"
}

variable "region" {
  type        = string
  description = "The default region to use"
  default     = "europe-west1"
}

variable "db_password" {
  type        = string
  description = "The password for the sgs db instance"
}

.
.
.
```

Service Account and IAM Roles(admin provisioner)

Now that the admin part of our tutorial is done we can proceed with the provisioning of the resources. To do that we need a service account key with the right permissions so `terraform` can call the APIs.

console.cloud.google.com → IAM & Admin → Service Accounts

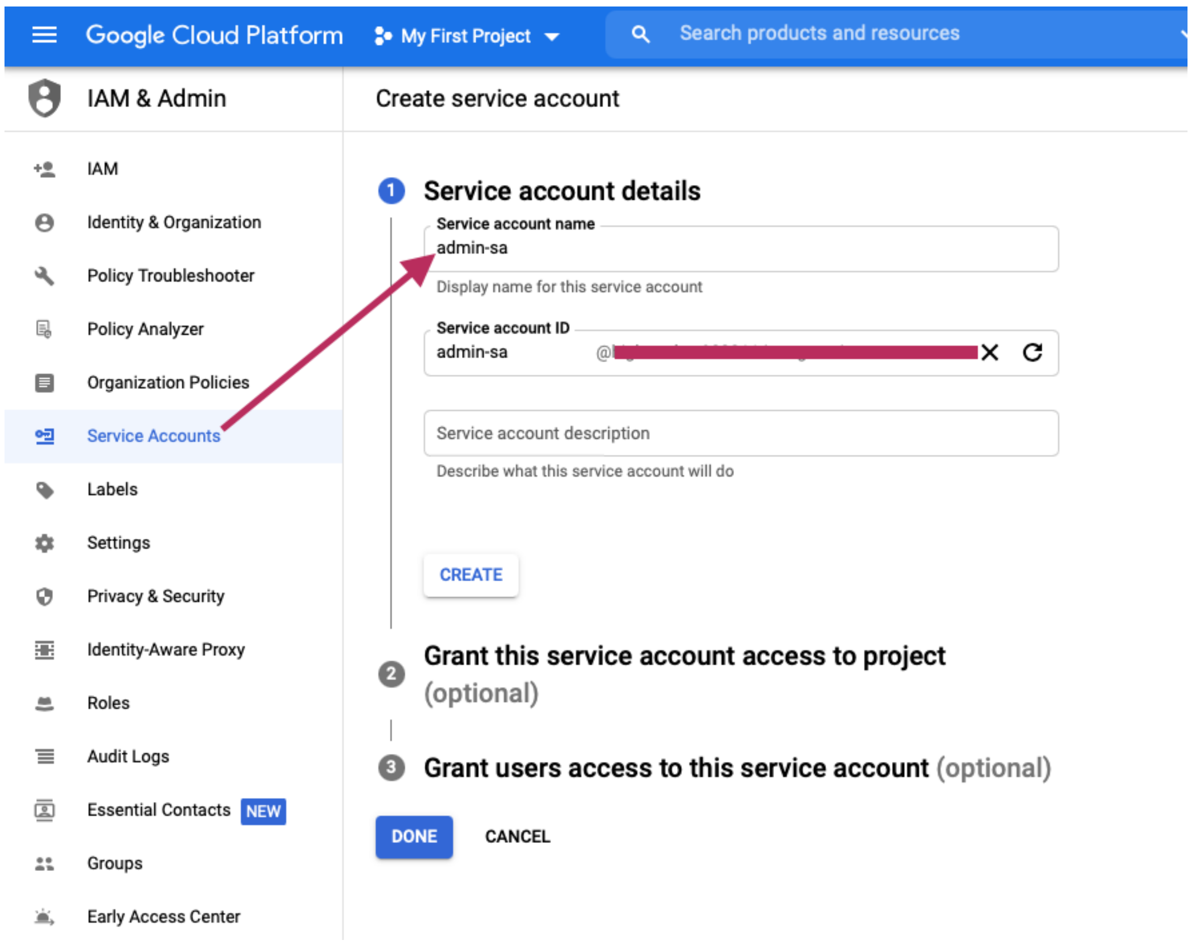


Service accounts for project "My First Project"

A service account represents a Google Cloud service identity, such as code running on Comput Google. Learn more about service accounts.

Organization policies can be used to secure service accounts and block risky service account f the creation of service accounts entirely. Learn more about service account organization polici

I called it `admin-sa`



Create service account

1 Service account details

Service account name: admin-sa

Display name for this service account

Service account ID: admin-sa @[redacted]

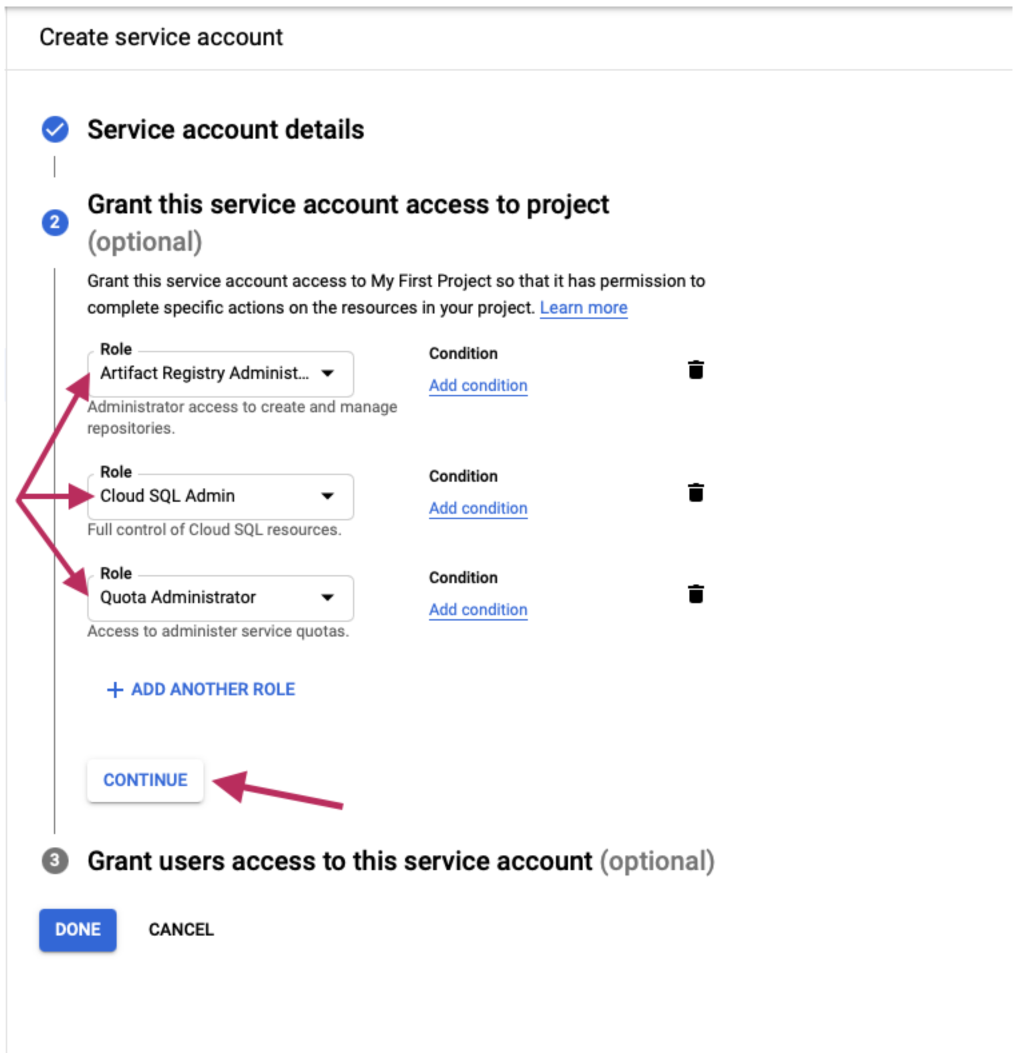
Service account description

Describe what this service account will do

2 Grant this service account access to project (optional)

3 Grant users access to this service account (optional)

Assign roles to the SA:


Create service account

Service account details

2 Grant this service account access to project (optional)

Grant this service account access to My First Project so that it has permission to complete specific actions on the resources in your project. Learn more

Role: Artifact Registry Administ... — Condition: Add condition
Administrator access to create and manage repositories.

Role: Cloud SQL Admin — Condition: Add condition
Full control of Cloud SQL resources.

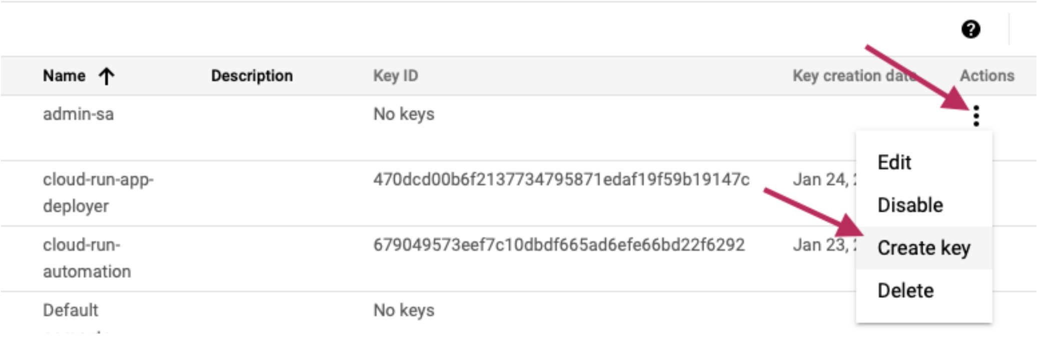
Role: Quota Administrator — Condition: Add condition
Access to administer service quotas.

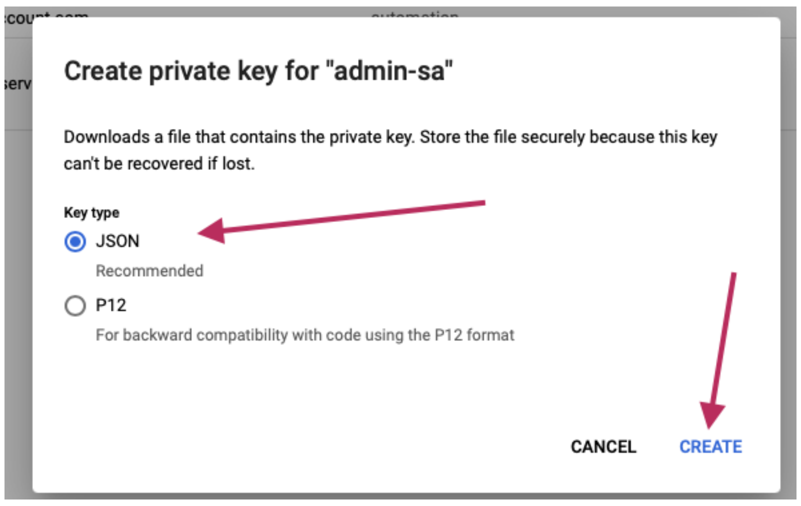
3 Grant users access to this service account (optional)

The roles assigned are:

- Artifact Registry Administrator
- Cloud SQL Admin
- Quota Administrator

Now create the service account key of type json by clicking on `Create key`


Name	Description	Key ID	Key creation date	Actions
admin-sa		No keys		⋮
cloud-run-app-deployer		470dcd00b6f2137734795871edaf19f59b19147c	Jan 24,	
cloud-run-automation		679049573eef7c10dbdf665ad6efe66bd22f6292	Jan 23,	
Default		No keys		



Create private key for "admin-sa"

Downloads a file that contains the private key. Store the file securely because this key can't be recovered if lost.

Key type

● JSON
Recommended

○ P12
For backward compatibility with code using the P12 format

export the path to the key as

```
export GOOGLE_APPLICATION_CREDENTIALS=/path/to/key_file.json
```

The admin part is done we just need to execute the following commands:

```
terraform init #To download all the dependencies (providers and modules)
```

```
export PROJECT_ID=you-project-id
```

```
export DB_PASSWORD=SuperSecretPassword
```

```
terraform plan -var="project_id=$PROJECT_ID" -var="db_password=$DB_PASSWORD" -out=plan.out
```

```
terraform apply plan.out
```

The output returns the database instance name, we are going to need it so let's export it

```
export DB_INSTANCE_NAME=your-instance-name
```

Docker registry setup

Now we can populate the registry with our image:

```
docker pull outofdevops/simple-go-service:cloud-run
```

```
gcloud auth configure-docker \
    europe-west1-docker.pkg.dev
```

```
docker tag outofdevops/simple-go-service:cloud-run europe-west1-docker.pkg.dev/${PROJECT_ID}/docker-ar/simple-go-service:cloud-run
```

```
docker push europe-west1-docker.pkg.dev/${PROJECT_ID}/docker-ar/simple-go-service:cloud-run
```

We can list all the images in the repository with:

```
gcloud artifacts docker images list europe-west1-docker.pkg.dev/${PROJECT_ID}/docker-ar
```

App Implementation

Now we can move to the other app directory and start creating the resources we need do deploy our container on cloud-run. Similarly to the admin part we create the following files:

- `cloud-run.tf`
- `config.tf`
- `output.tf`
- `vars.tf`

```
//app/cloud-run.tf

locals {
  image_name = "europe-west1-docker.pkg.dev/${var.project_id}/docker-ar/simple-go-service:cloud-run"
}

resource "google_cloud_run_service" "sgs" {
  name     = "cloudrun-sgs"
  location = var.region
  template {
    spec {
      containers {
        image = local.image_name
        env {
          name  = "APP_DB_USERNAME"
          value = "sgs"
        }
        env {
          name  = "APP_DB_PASSWORD"
          value = var.db_password
        }
        env {
          name  = "APP_DB_NAME"
          value = "sgs"
        }
        env {
          name  = "APP_DB_HOST"
          value = "/cloudsql/${data.google_sql_database_instance.sgs.connection_name}"
        }
      }
    }

    metadata {
      annotations = {
        "autoscaling.knative.dev/maxScale"      = "1"
        "run.googleapis.com/cloudsql-instances" = data.google_sql_database_instance.sgs.connection_name
        "run.googleapis.com/client-name"        = "terraform"
      }
    }
  }
  autogenerate_revision_name = true
}

data "google_sql_database_instance" "sgs" {
  name = var.db_instance_name
}

data "google_iam_policy" "public" {
  binding {
    role = "roles/run.invoker"
    members = [
      "allUsers",
    ]
  }
}
# Enable public access on Cloud Run service
resource "google_cloud_run_service_iam_policy" "public" {
  location    = google_cloud_run_service.sgs.location
  project     = google_cloud_run_service.sgs.project
  service     = google_cloud_run_service.sgs.name
  policy_data = data.google_iam_policy.public.policy_data
}
```

Then we move to the config file

```
//app/config.tf

terraform {
  required_providers {
    google = {
      version = "3.53.0"
      source  = "hashicorp/google"
    }
  }
}

provider "google" {
  project = var.project_id
  region  = var.region
}
```

The last two files are `vars.tf` and `output.tf`

```
//app/vars.tf

variable "project_id" {
  type        = string
  description = "The GCP Project ID"
}

variable "db_password" {
  type        = string
  description = "The password for the sgs db instance"
}

variable "region" {
  type        = string
  description = "The default region to use"
  default     = "europe-west1"
}

variable "db_instance_name" {
  description = "The name of the database instance"
  type        = string
}
```

```
//app/output.tf

output "service_url" {
  value = google_cloud_run_service.sgs.status[0].url
}
```

We need to create a service account for the app resources, similarly to the `admin-sa` assign the following roles:

- `Cloud Run Admin`
- `Cloud SQL Viewer`
- `Service Account User`

Now download the key and export its path as `GOOGLE_APPLICATION_CREDENTIALS`:

```
export GOOGLE_APPLICATION_CREDENTIALS=/path/to/key_file.json
terraform plan -var="project_id=$PROJECT_ID" -var="db_password=$DB_PASSWORD" -
var="db_instance_name=$DB_INSTANCE_NAME" -out=plan.out
terraform apply plan.out
```

Export the output of `service_url` as bash variable and test:

```
> export SERVICE_URL=the-value-returned-in-the-output
> curl $SERVICE_URL/events | jq
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   164  100   164    0     0    351      0 --:--:-- --:--:-- --:--:--   350
[]
> curl --header "Content-Type: application/json" \
  --request POST \
  --data '{"name": "video", "state": "recording"}' \
  ${SERVICE_URL}/events
curl $SERVICE_URL/events | jq
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   164  100   164    0     0    351      0 --:--:-- --:--:-- --:--:--   350
[
  {
    "id": "4e5fe630-a36c-4455-b145-9874eb2353f8",
    "name": "video",
    "state": "recording"
  }
]
```

Destroy all the resources

Don't forget to delete all the resources created to avoid surprises on the GCloud bill.

```
cd app
terraform destroy -var="project_id=$PROJECT_ID" -var="db_password=$DB_PASSWORD" -
var="db_instance_name=$DB_INSTANCE_NAME" -auto-approve
cd ../admin
terraform destroy -var="project_id=$PROJECT_ID" -var="db_password=$DB_PASSWORD" -auto-approve
```

Conclusions

Provisioning infrastructure as code it's a cumbersome process at the beginning but once done the flexibility is guaranteed. With this script you can recreate the resources in seconds and you can also reuse the code with multiple applications. To favour code reuse terraform provides even a better feature: terraform modules but I will introduce them in another post.

For more articles like this follow me on socials (links down in the footer).

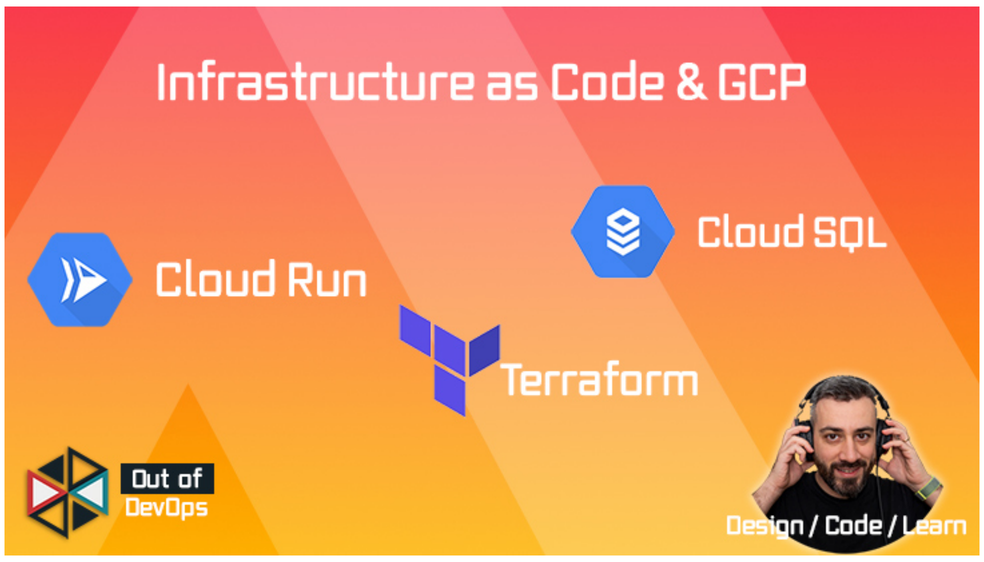
References

Links:

- `google_project_service` https://registry.terraform.io/providers/hashicorp/google/latest/docs/resources/google_project_service
- `google_artifact_registry_repository` https://registry.terraform.io/providers/hashicorp/google/latest/docs/resources/artifact_registry_repository
- `google_sql_user` https://registry.terraform.io/providers/hashicorp/google/latest/docs/resources/sql_user
- `google_sql_database` https://registry.terraform.io/providers/hashicorp/google/latest/docs/resources/sql_database
- `google_sql_database_instance` https://registry.terraform.io/providers/hashicorp/google/latest/docs/resources/sql_database_instance
- `google_cloud_run_service` https://registry.terraform.io/providers/hashicorp/google/latest/docs/resources/cloud_run_service
- `google_cloud_run_service_iam_policy` https://registry.terraform.io/providers/hashicorp/google/latest/docs/resources/cloud_run_service_iam#google_cloud_run_service_iam_policy
- `depends_on` https://www.terraform.io/docs/language/meta-arguments/depends_on.html
- https://cloud.google.com/sql/docs/postgres/create-manage-users
- List with all the roles: https://cloud.google.com/iam/docs/understanding-roles

Search

LATEST POST



Post By Anto Feb 03, 2021

[Cloud Run and IaC with Terraform]

Post By Anto Jan 10, 2021

[Infrastructure as Code (IaC): What is Terraform?]

TAG

Artifact registry Cloudrun Cloudsql Devops Gcp Iac Software engineering Terraform

CATEGORIES

Devops Software engineering

Out Of DevOps

Follow