Article
Mikhail Khomenko · Jan 13, 2020 16m read

Open Exchange

Automating GKE creation on CircleCI builds

Last time we launched an IRIS application in the Google Cloud using its GKE service.

And, although creating a cluster manually (or through gcloud) is easy, the modern Infrastructure-as-Code (IaC) approach advises that the description of the Kubernetes cluster should be stored in the repository as code as well. How to write this code is determined by the tool that ' s used for IaC.

In the case of Google Cloud, there are several options, among them Deployment Manager and Terraform. Opinions are divided as to which is better: if you want to learn more, read this Reddit thread Opinions on Terraform vs. Deployment Manager? and the Medium article Comparing GCP Deployment Manager and Terraform.

For this article we ' ll choose Terraform, since it ' s less tied to a specific vendor and you can use your IaC with different cloud providers.

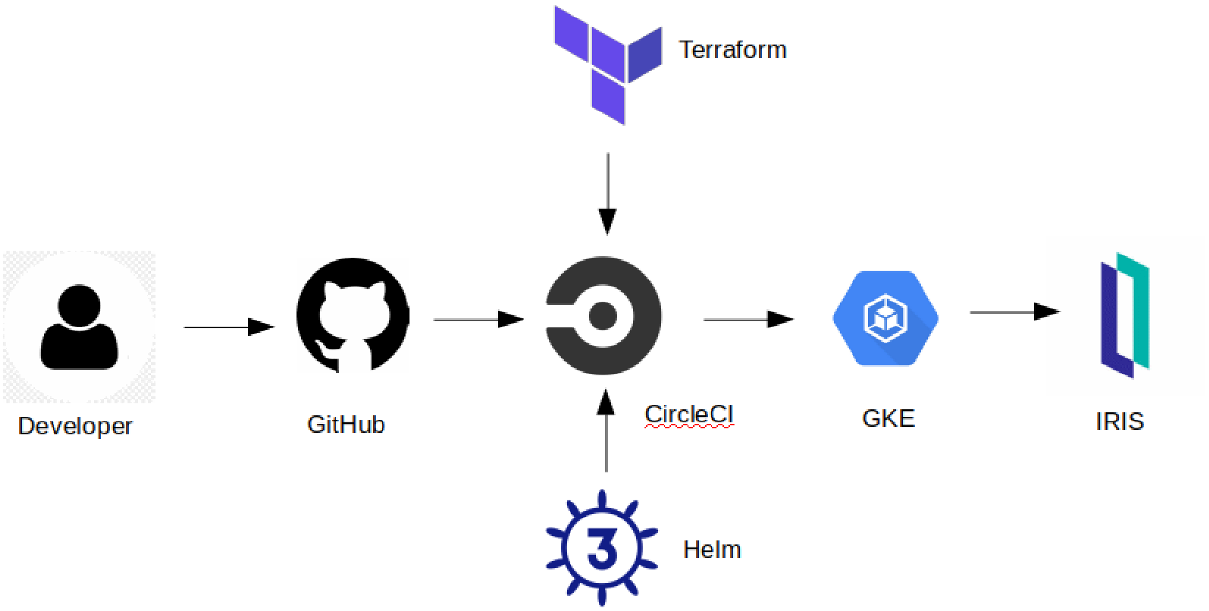
We ' ll assume you ' ve read the earlier article and already have a Google account, and that you ' ve created a project named " Development, " as in the previous article. In this article, its ID is shown as <PROJECT_ID>. In the examples below, change it to the ID of your own project.

Keep in mind that Google isn ' t free, although it has a free tier. Be sure to control your expenses.

We ' ll also assume that you ' ve already forked the original repository. We ' ll call this fork " my-objectscript-rest-docker-template " and refer to its root directory as " <root_repo_dir> " throughout this article.

All code samples are stored in this repo to simplify copying and pasting.

The following diagram depicts the whole deployment process in one picture:



So, let's install the latest version of Terraform at the time of writing:

```
$ terraform version
Terraform v0.12.17
```

The version is important here, because many examples on the Internet use earlier versions, and 0.12 brought many changes.

We want Terraform to perform certain actions (use certain APIs) in our GCP account. To enable this, create a Service Account with the name 'terraform', and enable the Kubernetes Engine API. Don ’ t worry about how we ’ re going to achieve this — just read further and your questions will be addressed.

Let's try an example with the gcloud utility, although we could also use the Web Console.

We're going to use a couple different commands in the following examples. See the following documentation topics for more details on these commands and features.

- gcloud iam service-accounts create
- Granting roles to a service account for specific resources
- gcloud iam service-accounts keys create
- Enabling an API in your Google Cloud project

Now let's walk through the example.

```
$ gcloud init
```

Because we worked with gcloud in the previous article, we won ’ t discuss all of the setup details here. For this example, run the following commands:

```
$ cd <root_repo_dir>
$ mkdir terraform; cd terraform
$ gcloud iam service-accounts create terraform --description "Terraform" --display-name "terraform"
```

Now let's add a few roles to the terraform service account besides “ Kubernetes Engine Admin ” (container.admin). These roles will be useful to us in the future.

```
$ gcloud projects add-iam-policy-binding <PROJECT_ID> \
  --member serviceAccount:terraform@<PROJECT_ID>.iam.gserviceaccount.com \
  --role roles/container.admin
$ gcloud projects add-iam-policy-binding <PROJECT_ID> \
  --member serviceAccount:terraform@<PROJECT_ID>.iam.gserviceaccount.com \
  --role roles/iam.serviceAccountUser

$ gcloud projects add-iam-policy-binding <PROJECT_ID> \
  --member serviceAccount:terraform@<PROJECT_ID>.iam.gserviceaccount.com \
  --role roles/compute.viewer

$ gcloud projects add-iam-policy-binding <PROJECTID> \
  --member serviceAccount:terraform@<PROJECTID>.iam.gserviceaccount.com
\
  --role roles/storage.admin

$ gcloud iam service-accounts keys create account.json \
--iam-account terraform@<PROJECTID>.iam.gserviceaccount.com
```

Note that the last entry creates your *account.json* file. Be sure to keep this file secret.

```
$ gcloud projects list
$ gcloud config set project <PROJECTID>
$ gcloud services list --available ¦ grep 'Kubernetes Engine'
$ gcloud services enable container.googleapis.com
$ gcloud services list --enabled ¦ grep 'Kubernetes Engine'
container.googleapis.com Kubernetes Engine API
```

Next, let’s describe the GKE cluster in Terraform HCL language. Note that we use several placeholders here; replace them with your values:

Placeholder	Meaning
<PROJECTID>	GCP project ID
<BUCKETNAME>	Storage for Terraform state/lock—should be uni
<REGION>	Region where resources will be created
<LOCATION>	Zone where resources will be created
<CLUSTERNAME>	GKE cluster name
<NODESPOOLNAME>	GKE worker nodes pool name

Here’s the HCL configuration for the cluster in practice:

```
$ cat main.tf
terraform {
  requiredversion = "⩾ 0.12"
  backend "gcs" {
  bucket = "<BUCKETNAME>"
  prefix = "terraform/state"
  credentials = "account.json"
  }
}
provider "google" {
  credentials = file("account.json")
  project = "<PROJECTID>"
  region = "<REGION>"
}

resource "googlecontainercluster" "gke-cluster" {
  name = "<CLUSTERNAME>"
  location = "<LOCATION>"
  removedefaultnodepool = true
  # In regional cluster (location is region, not zone)
  # this is a number of nodes per zone
  initialnodecount = 1
}

resource "googlecontainernodepool" "preemptiblenodepool" {
  name = "<NODESPOOLNAME>"
  location = "<LOCATION>"
  cluster = googlecontainercluster.gke-cluster.name
  # In regional cluster (location is region, not zone)
  # this is a number of nodes per zone
  nodecount = 1

  nodeconfig {
  preemptible = true
  machinetype = "n1-standard-1"
  oauthscopes = [
  "storage-ro",
  "logging-write",
  "monitoring"
  ]
  }
}
```

To make sure the HCL code is in the proper format, Terraform provides a handy formatting command you can use:

```
$ terraform fmt
```

The code snippet shown above indicates that the created resources will be provided by Google, and the resources themselves are googlecontainercluster and googlecontainernodepool, which we designate preemptible for costs savings. We also choose to create our own pool instead of using the default.

Let ’ s focus briefly on the following setting:

```
terraform {
  requiredversion = "⩾ 0.12"
  backend "gcs" {
  Bucket = "<BUCKETNAME>"
  Prefix = "terraform/state"
  credentials = "account.json"
  }
}
```

Terraform writes everything it's done into the status file and then uses this file for other work. For convenient sharing, it ’ s better to store this file somewhere in a remote place. A typical place is a Google Bucket.

Let's create this bucket. Use the name of your bucket instead of the placeholder <BUCKET_NAME>. Before bucket creation let ’ s check if <BUCKET_NAME> is available as it has to be unique across all GCP:

```
$ gsutil acl get gs://<BUCKET_NAME>
```

Good answer:

```
BucketNotFoundException: 404 gs://<BUCKET_NAME> bucket does not exist
```

"Busy" answer means you have to choose another name:

```
AccessDeniedException: 403 <YOUR_ACCOUNT> does not have
storage.buckets.get access to <BUCKET_NAME>
```

Let ’ s also enable versioning, as Terraform recommends.

```
$ gsutil mb -l EU gs://<BUCKET_NAME>
$ gsutil versioning get gs://<BUCKET_NAME>
gs://<BUCKET_NAME>: Suspended

$ gsutil versioning set on gs://<BUCKET_NAME>

$ gsutil versioning get gs://<BUCKET_NAME>
gs://<BUCKET_NAME>: Enabled
```

Terraform is modular and needs to add a Google provider plugin to create something in GCP. We use the following command to do this:

```
$ terraform init
```

Let's look at what Terraform is going to do to create a GKE cluster:

```
$ terraform plan -out dev-cluster.plan
```

The command output includes details of the plan. If you have no objections, let's implement this plan:

```
$ terraform apply dev-cluster.plan
```

By the way, to delete the resources created by Terraform, run this command from the <root_repo_dir>/terraform/ directory:

```
$ terraform destroy -auto-approve
```

Let’s leave the cluster as is for a while and move on. But first note that we don’t want to push everything into the repository, so we’ll add several files to the exceptions:

```
$ cat <root_repo_dir>/.gitignore
.DS_Store
terraform/.terraform/
terraform/*.plan
terraform/*.json
```

Using Helm

In the previous article, we stored Kubernetes manifests as yaml files in the <root_repo_dir>/k8s/ directory, which we then sent to the cluster using the "kubectl apply" command.

This time we'll try a different approach: using the Kubernetes package manager Helm, which has recently been updated to version 3. Please, use version 3 or later because version 2 had Kubernetes-side security issues (see Running Helm in production: Security best practices for details). First, we’ll pack the Kubernetes manifests from our k8s/ directory into a Helm package, which is known as a chart. A Helm chart installed in Kubernetes is called a release. In a minimal configuration, a chart will consist of several files:

```
$ mkdir <root_repo_dir>/helm; cd <root_repo_dir>/helm
$ tree <root_repo_dir>/helm/
helm/
├── Chart.yaml
├── templates
│   ├── deployment.yaml
│   ├── _helpers.tpl
│   └── service.yaml
└── values.yaml
```

Their purpose is well-described on the official site. The best practices for creating your own charts are described in the The Chart Best Practices Guide in the Helm documentation.

Here’s what the contents of our files look like:

```
$ cat Chart.yaml
apiVersion: v2
name: iris-rest
version: 0.1.0
appVersion: 1.0.3
description: Helm for ObjectScript-REST-Docker-template application
sources:
- https://github.com/intersystems-community/objectscript-rest-docker-template
- https://github.com/intersystems-community/gke-terraform-circleci-objects...
```

```
$ cat templates/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ template "iris-rest.name" . }}
  labels:
  app: {{ template "iris-rest.name" . }}
  chart: {{ template "iris-rest.chart" . }}
  release: {{ .Release.Name }}
  heritage: {{ .Release.Service }}
spec:
  replicas: {{ .Values.replicaCount }}
  strategy:
  {{- .Values.strategy | nindent 4 }}
  selector:
  matchLabels:
  app: {{ template "iris-rest.name" . }}
  release: {{ .Release.Name }}
  template:
  metadata:
  labels:
  app: {{ template "iris-rest.name" . }}
  release: {{ .Release.Name }}
  spec:
  containers:
  - image: {{ .Values.image.repository }}:{{ .Values.image.tag }}
  name: {{ template "iris-rest.name" . }}
  ports:
  - containerPort: {{ .Values.webPort.value }}
  name: {{ .Values.webPort.name }}

$ cat templates/service.yaml
{{- if .Values.service.enabled }}
apiVersion: v1
kind: Service
metadata:
  name: {{ .Values.service.name }}
  labels:
  app: {{ template "iris-rest.name" . }}
  chart: {{ template "iris-rest.chart" . }}
  release: {{ .Release.Name }}
  heritage: {{ .Release.Service }}
spec:
  selector:
  app: {{ template "iris-rest.name" . }}
  release: {{ .Release.Name }}
  ports:
  {{- range $key, $value := .Values.service.ports }}
  - name: {{ $key }}
{{ toYaml $value ¦ indent 6 }}
  {{- end }}
  type: {{ .Values.service.type }}
  {{- if ne .Values.service.loadBalancerIP "" }}
  loadBalancerIP: {{ .Values.service.loadBalancerIP }}
  {{- end }}
{{- end }}

$ cat templates/helpers.tpl
{{/* vim: set filetype=mustache: */}}
{{/*
Expand the name of the chart.
*/}}
{{- define "iris-rest.name" -}}
{{- default .Chart.Name .Values.nameOverride ¦ trunc 63 ¦ trimSuffix "-" -}}
{{- end -}}

{{/*
Create chart name and version as used by the chart label.
*/}}
{{- define "iris-rest.chart" -}}
{{- printf "%s-%s" .Chart.Name .Chart.Version ¦ replace "+" ""_¦ trunc 63 ¦
trimSuffix "-" -}}
{{- end -}}


$ cat values.yaml
namespaceOverride: iris-rest
replicaCount: 1

strategy: ¦
  type: Recreate

image:
  repository: eu.gcr.io/iris-rest
  tag: v1

webPort:
  name: web
  value: 52773

service:
  enabled: true
  name: iris-rest
  type: LoadBalancer
  loadBalancerIP: ""
  ports:
  web:
  port: 52773
  targetPort: 52773
  protocol: TCP
```

To create the Helm charts, install the Helm client and the kubectl command-line utility.

```
$ helm version
version.BuildInfo{Version:"v3.0.1",
GitCommit:"7c22ef9ce89e0ebeb7125ba2ebf7d421f3e82ffa", GitTreeState:"clean",
GoVersion:"go1.13.4"}
```

Create a namespace called iris. It would be nice if this were created during the deployment, but so far this is not the case.

First, add credentials for the cluster created by Terraform to kube-config:

```
$ gcloud container clusters get-credentials <CLUSTER_NAME> --zone
<LOCATION> --project <PROJECT_ID>
$ kubectl create ns iris
```

Confirm (without kicking off a real deploy) that Helm is going to create the following in Kubernetes:

```
$ cd <root_repo_dir>/helm
$ helm upgrade iris-rest \
  --install \
  . \
  --namespace iris \
  --debug \
  --dry-run
```

The output—the Kubernetes manifests—has been omitted for space here. If everything looks good, let ’ s deploy:

```
$ helm upgrade iris-rest --install . --namespace iris

$ helm list -n iris --all
Iris-rest iris 1 2019-12-14 15:24:19.292227564 +0200 EET deployed   iris-
rest-0.1.0 1.0.3
```

We see that Helm has deployed our application, but since we haven ’ t created the Docker image eu.gcr.io/iris-rest:v1 yet, Kubernetes can ’ t pull it (ImagePullBackOff):

```
$ kubectl -n iris get po
NAME  READY  STATUS  RESTARTS  AGE
iris-rest-59b748c577-6cnrt 0/1  ImagePullBackOff  0  10m
```

Let ’ s finish with it for now:

```
$ helm delete iris-rest -n iris
```

The CircleCI Side

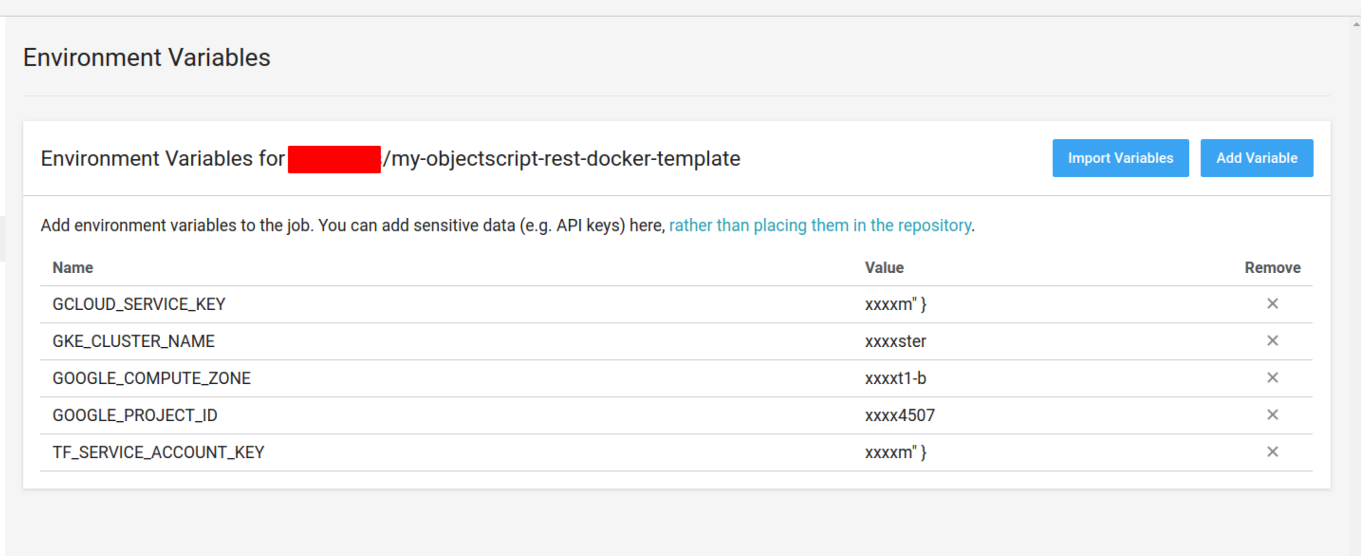
Now that we ’ ve tried out Terraform and the Helm client, let ’ s put them to use during the deployment process on the CircleCI side.

```
$ cat <rootrepodir>/.circleci/config.yml
version: 2.1
orbs:
  gcp-gcr: circleci/gcp-gcr@0.6.1

jobs:
  terraform:
  docker:
  # Terraform image version should be the same as when
  # you run terraform before from the local machine
  - image: hashicorp/terraform:0.12.17
  steps:
  - checkout
  - run:
  name: Create Service Account key file from environment variable
  workingdirectory: terraform
  command: echo ${TFSERVICEACCOUNTKEY} > account.json
  - run:
  name: Show Terraform version
  command: terraform version
  - run:
  name: Download required Terraform plugins
  workingdirectory: terraform
```

```
 command: terraform init
 - run:
 name: Validate Terraform configuration
 workingdirectory: terraform
 command: terraform validate
 - run:
 name: Create Terraform plan
 workingdirectory: terraform
 command: terraform plan -out /tmp/tf.plan
 - run:
 name: Run Terraform plan
 workingdirectory: terraform
 command: terraform apply /tmp/tf.plan
 k8sdeploy:
 docker:
 - image: kiwigrid/gcloud-kubectl-helm:3.0.1-272.0.0-218
 steps:
 - checkout
 - run:
 name: Authorize gcloud on GKE
 workingdirectory: helm
 command: ¦
 echo ${GCLOUDSERVICEKEY} > gcloud-service-key.json
 gcloud auth activate-service-account --key-file=gcloud-service-key.json
 gcloud container clusters get-credentials ${GKECLUSTERNAME} --zone
${GOOGLECOMPUTEZONE} --project ${GOOGLEPROJECTID}
 - run:
 name: Wait a little until k8s worker nodes up
 command: sleep 30 # It ’ s a place for improvement
 - run:
 name: Create IRIS namespace if it doesn't exist
 command: kubectl get ns iris ¦¦ kubectl create ns iris
 - run:
 name: Run Helm release deployment
 workingdirectory: helm
 command: ¦
 helm upgrade iris-rest \
 --install \
 . \
 --namespace iris \
 --wait \
 --timeout 300s \
 --atomic \
 --set image.repository=eu.gcr.io/${GOOGLEPROJECTID}/iris-rest \
 --set image.tag=${CIRCLESHA1}
 - run:
 name: Check Helm release status
 command: helm list --all-namespaces --all
 - run:
 name: Check Kubernetes resources status
 command: |
 kubectl -n iris get pods
 echo
 kubectl -n iris get services
workflows:
 main:
 jobs:
 - terraform
 - gcp-gcr/build-and-push-image:
 dockerfile: Dockerfile
 gcloud-service-key: GCLOUDSERVICEKEY
 google-compute-zone: GOOGLECOMPUTEZONE
 google-project-id: GOOGLEPROJECTID
 registry-url: eu.gcr.io
 image: iris-rest
 path: .
 tag: ${CIRCLESHA1}
 - k8sdeploy:
 requires:
 - terraform
 - gcp-gcr/build-and-push-image
```

You ’ ll need to add several environment variables to your project on CircleCI side:



Environment Variables

Environment Variables for [redacted]/my-objectscript-rest-docker-template

Add environment variables to the job. You can add sensitive data (e.g. API keys) here, rather than placing them in the repository.

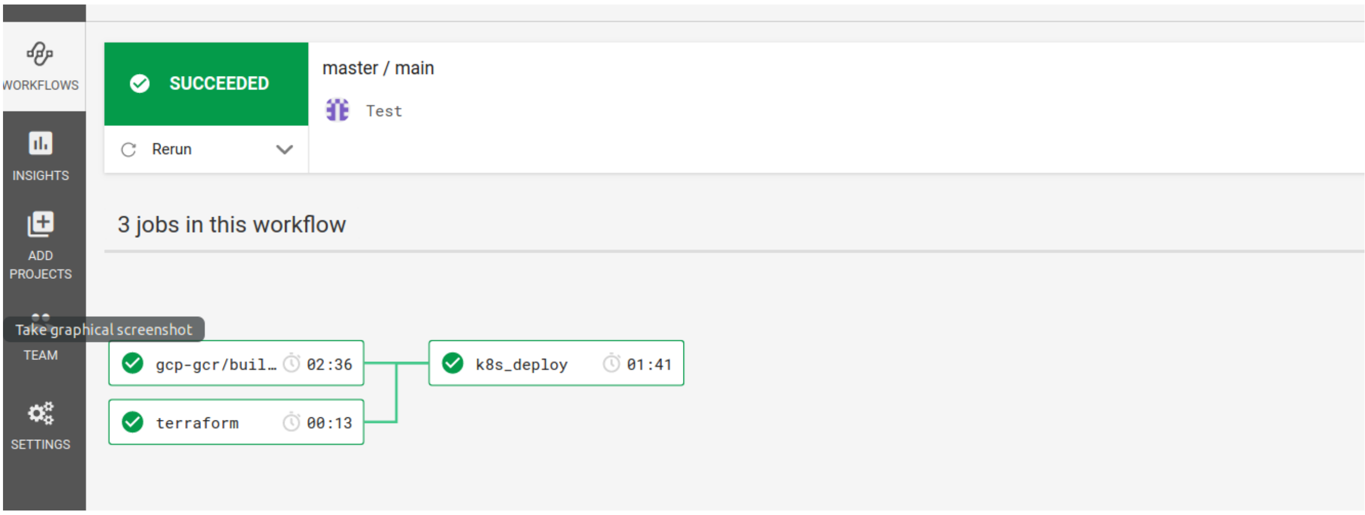
Name	Value	Remove
GCLOUD_SERVICE_KEY	xxxxm" }	×
GKE_CLUSTER_NAME	xxxxster	×
GOOGLE_COMPUTE_ZONE	xxxxt1-b	×
GOOGLE_PROJECT_ID	xxxx4507	×
TF_SERVICE_ACCOUNT_KEY	xxxxm" }	×

The GCLOUDSERVICEKEY is the CircleCI service account key, and TFSERVICEACCOUNTKEY is the Terraform service account key. Recall that the service account key is the whole content of *account.json* file.

Next, let ’ s push our changes to a repository:

```
$ cd <rootrepodir>
$ git add .circleci/ helm/ terraform/ .gitignore
$ git commit -m "Add Terraform and Helm"
$ git push
```

The CircleCI UI dashboard should show that everything is ok:



SUCCEEDED

master / main

Test

3 jobs in this workflow

gcp-gcr/buil… 02:36

terraform 00:13

k8s_deploy 01:41

Terraform is an idempotent tool and if the GKE cluster is present, the "terraform" job won ’ t do anything. If the cluster doesn ’ t exist, it will be created before Kubernetes deployment.

Finally, let ’ s check IRIS availability:

```
$ gcloud container clusters get-credentials <CLUSTERNAME> --zone <LOCATION> --project <PROJECTID>
$ kubectl -n iris get svc
NAME  TYPE  CLUSTER-IP  EXTERNAL-IP  PORT(S)  AGE
Iris-rest  LoadBalancer  10.23.249.42  34.76.130.11  52773:31603/TCP  53s

$ curl -XPOST -H "Content-Type: application/json" -u system:SYS
34.76.130.11:52773/person/ -d '{"Name":"John Dou"}'

$ curl -XGET -u system:SYS 34.76.130.11:52773/person/all
[{"Name":"John Dou"},]
```

Conclusion

Terraform and Helm are standard DevOps tools and should be fine integrated with IRIS deployment.

They do require some learning, but after some practice, they can really save you time and effort.

#Best Practices #Cloud #Containerization #DevOps #Docker #Google Cloud Platform (GCP) #Kubernetes
#InterSystems IRIS #Open Exchange
Check the related application on InterSystems Open Exchange

Source URL:https://community.intersystems.com/post/automating-gke-creation-circleci-builds