

First Project:

DevOps exercise GCP: Infrastructure-As-Code

Mala Shrestha
Junior Cloud Engineer

Deploying Docker React Image using Google Kubernetes Engine:

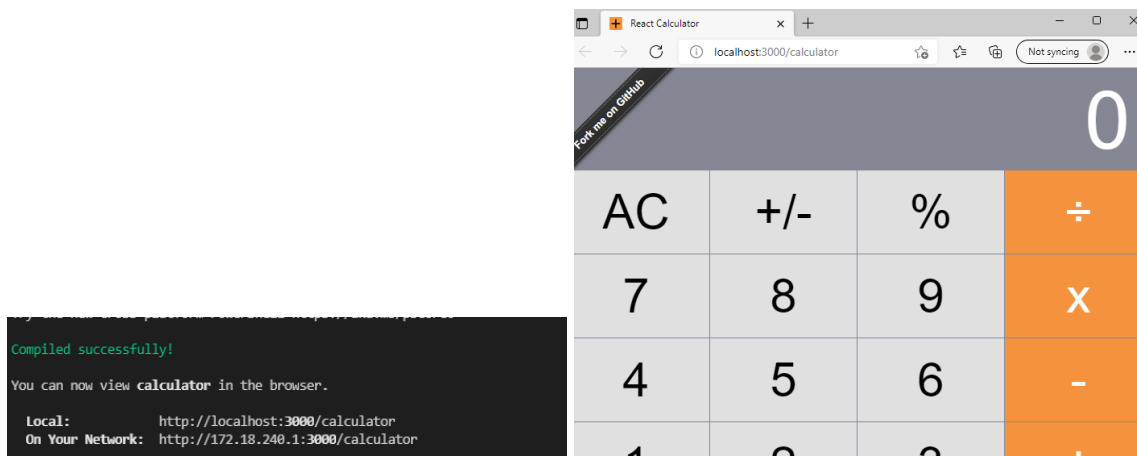
Step 1:

- The first step is cloning the react application from the given GitHub repository into a working directory in the local machine.

```
C:\Users\Mala Shrestha\Documents\Project1>git clone https://github.com/ahfarmer/calculator.git
```

Step 2:

- After cloning, change the directory to the directory of the cloned folder and the application can be tested in a code editor (Visual Studio Code in this case) to check if it is running correct locally with the command **npm start**. The successful compilation displays a message as follow opening your application in the browser at port 3000.



Step 3:

- Since, the application is running perfect, it can be converted to docker image now which will be used to deploy later in Google Cloud Platform using Kubernetes Engine. In order to do so, it should be ensured that 'Docker Desktop for Windows' is successfully installed first. The installation link can be found here:

<https://docs.docker.com/desktop/windows/install/>

After the successful installation of docker, the docker version can be checked using `docker --version`. Then, the docker image of the application can be built using the command `docker build -t image-name` in the same working directory as mentioned above.

```
PS C:\Users\Mala Shrestha\Documents\Project1\calculator> docker build -t my-image .
[+] Building 7.8s (2/3)
=> [internal] load build definition from Dockerfile                                0.1s
=> => transferring dockerfile: 31B                                              0.1s
=> [internal] load .dockerignore                                                 0.1s
=> => transferring context: 34B                                                0.0s
=> [internal] load metadata for docker.io/library/node:16                      7.5s
```

And can be checked with the command `docker images` that displays the images built.

```
PS C:\Users\Mala Shrestha\Documents\Project1\calculator> docker images
REPOSITORY          TAG         IMAGE ID      CREATED      SIZE
new-calc            latest      aa9f30894421  4 days ago  1.24GB
react-calc-image    latest      aa9f30894421  4 days ago  1.24GB
gcr.io/first-project-336610/new-calc    react      aa9f30894421  4 days ago  1.24GB
gcr.io/project-terraform-337113/react-calc-image    final      aa9f30894421  4 days ago  1.24GB
my-image            latest      aa9f30894421  4 days ago  1.24GB
```

The built image is then run using `docker run -p 3000:3000 my-image` command that displays the docker image of react application.

Starting the development server...
Compiled successfully!
You can now view calculator in the browser.

Local: <http://localhost:3000/calculator>
On Your Network: <http://172.17.0.2:3000/calculator>

localhost:3000/calculator

| | | |
|----|-----|---|
| AC | +/- | % |
| 7 | 8 | 9 |
| 4 | 5 | 6 |

Step 4:

- The next step is to take the application image to GCP. First, a new project is created in GCP. All the required APIs (Container Registry, Compute engine and Google Kubernetes Engine API) are enabled. The image built in step 3 is then pushed to Google Container Registry following the commands below:

```
docker build -t gcr.io/PROJECT_ID/IMAGE_NAME:TAG .
```

```
docker push gcr.io/PROJECT_ID/IMAGE_NAME:TAG
```


While pushing the image, if an authentication error with container registry is encountered, it can be solved by generating a short-lived access token to connect to the container registry which should be used within 60 minutes.


Username is `oauth2accesstoken`

Password is the output of `gcloud auth print-access-token`

After generating the password with the command, a login is required using `docker login -u oauth2accesstoken -p "ya29.8QEQLfY_..." https://gcr.io`

After a successful authentication, the push command is given again which will copy the image to the container registry of GCP.

| Name ↑ | Hostname ? | Visibility ? |
|--|------------|--------------|
|  react-calc-image | gcr.io | Private |

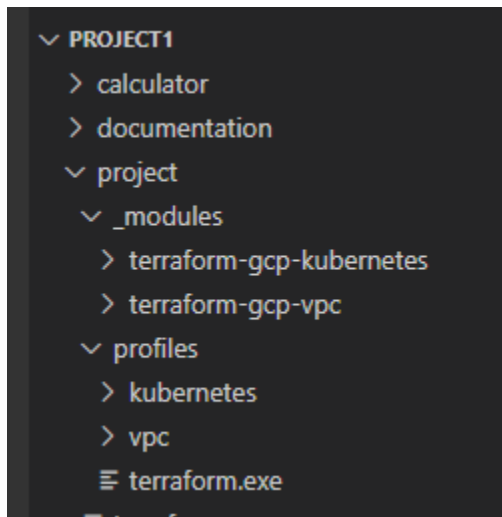
| <input type="checkbox"/> | Name | Tags | Virtual Size ? | Created | Uploaded ↓ |
|--------------------------|--|-------|----------------|------------|------------|
| <input type="checkbox"/> |  35be3dfb94f2 | final | 440.4 MB | 4 days ago | 4 days ago |

Step 5:

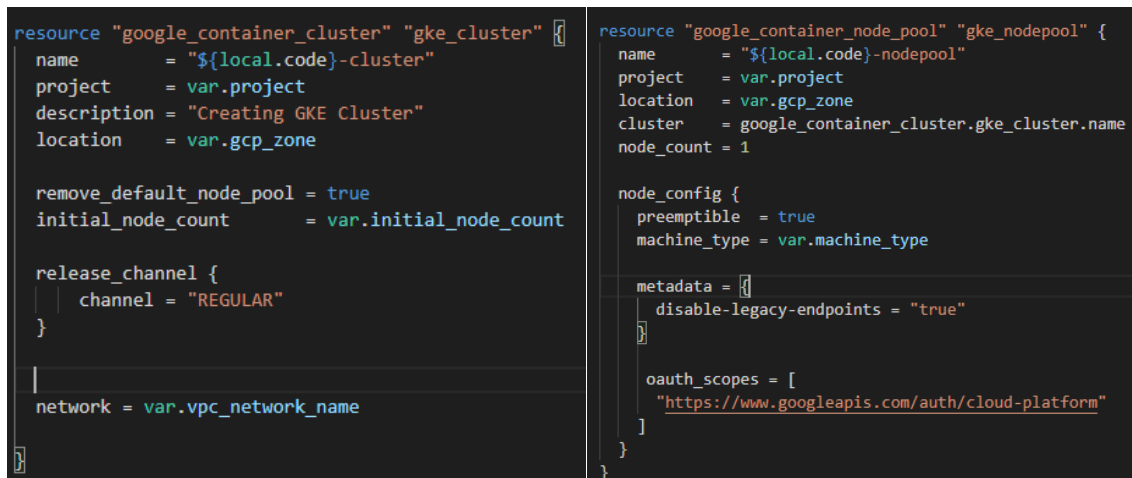
- When the image is all set in GCR, the necessary resources for Google Kubernetes Engine (GKE) are configured using Terraform as a configuration management system. Before creating resources with terraform, it is necessary to install terraform beforehand. The installation link can be found here: <https://www.terraform.io/downloads>

The installed file 'terraform.exe' should be kept in the working directory in order to run terraform commands. In the same root folder containing the application folder, a new folder 'project' is created. Inside the project folder, two separate folders '_modules' and 'profiles' are created. In _modules folder, two more folders are created: one for Kubernetes resources and the other for VPC network resources. Similarly, in profiles

folder, Kubernetes and VPC folders are made to call the modules. The structure of the folders looks like below:



In terraform-gcp-kubernetes and terraform-gcp-vpc folders, three .tf files are created; main.tf, variables.tf and output.tf that plan the resources to be created in GCP. For Kubernetes main.tf, two resources are created; gke_cluster and gke_nodepool. The resources are planned as below:



The variables.tf and output.tf are as follow:

```

variable "project" {
  type = string
  description = "Name of the project"
}

variable "gcp_zone" {
  type = string
  description = "GCP Zone"
}

variable "initial_node_count" {
  default = 1
}

variable "machine_type" {
  default = "n1-standard-1"
}

variable "environment" {
  description = "Working environment"
  default = "dev"
}

locals {
  region_code = jsondecode(file("${path.module}/../../_modules/terraform-gcp-kubernetes/region.json"))[var.region]
}

variable "app" {
  description = "Application"
  default = "calc"
}

variable "region" {
  description = "Region code"
  default = "us-central1"
}

variable "vpc_network_name" {
  description = "VPC network name"
  default = "dev-io-yfvaevogikh-calc-vpc"
}

```

```

output "endpoint" {
  value = google_container_cluster.gke_cluster.endpoint
}

output "master_version" {
  value = google_container_cluster.gke_cluster.master_version
}

```

Similarly, for VPC network, the resource `vpc_network` is created in `main.tf` along with `variables.tf` and `output.tf` as below:

```

resource "google_compute_network" "vpc_network" {
  project = var.project

  name                = "${local.code}-vpc"
  auto_create_subnetworks = var.enable_auto_create_subnetworks
  routing_mode        = var.routing_mode
}

```

```

variable "project" {
  type = string
  description = "Name of the project"
}

variable "enable_auto_create_subnetworks" {
  type = bool
  description = "It will create a subnet for each region automatically"
  default = true
}

variable "routing_mode" {
  type = string
  description = "The network-wide routing mode to use"
  default = "REGIONAL"
}

variable "environment" {
  description = "Working environment"
  default = "dev"
}

locals {
  region_code = jsondecode(file("${path.module}/../../_modules/terraform-gcp-kubernetes/region.json"))[var.region]
}

variable "region" {
  description = "Region code"
  default = "us-central1"
}

```

```

output "id" {
  value = google_compute_network.vpc_network.id
}

output "name" {
  value = google_compute_network.vpc_network.name
}

output "self_link" {
  value = google_compute_network.vpc_network.self_link
}

```

After making the modules ready, they are called in kubernetes and vpc folders of profiles folder in main.tf files as below:

```
t > profiles > kubernetes > main.tf

provider "google" {
  region = var.gcp_region
  zone   = var.gcp_zone
}

module "kubernetes" {
  source = "../../_modules/terraform-gcp-kubernetes"

  gcp_zone = var.gcp_zone
  project  = var.gcp_project
  initial_node_count = var.initial_node_count
  machine_type = var.machine_type
  vpc_network_name = var.vpc_network_name
}
```

```
ect > profiles > vpc > main.tf

1 provider "google" {
2   region = var.gcp_region
3   zone   = var.gcp_zone
4 }
5
6 module "vpc" {
7   source = "../../_modules/terraform-gcp-vpc"
8
9   project = var.gcp_project
10 }
11
12
```

The variables.tf files are added like in modules with the necessary inclusions and .tfvars files are also added to input the values of required variables as shown below:

```
> profiles > kubernetes > backend.tfvars

gcp_region = "us-central1"

gcp_zone = "us-central1-c"

gcp_project = "project-terraform-337113"
```

```
> profiles > vpc > vpc.tfvars

gcp_region = "us-central1"

gcp_zone = "us-central1-c"

gcp_project = "project-terraform-337113"
```

In order to secure the application, firewall rule is also added to VPC network by creating a separate resource file 'firewall.tf' file. This allows the traffic from the given ports only as seen below:

```
resource "google_compute_firewall" "default" {
  name        = "${local.code}-firewall"
  project     = var.gcp_project
  description = "Setting firewall rules"
  priority    = "1000"
  direction   = "INGRESS"
  network     = "dev-io-yfvaevoiggkh-calc-vpc"
  source_ranges = ["10.128.0.0/20"]
  allow {
    protocol = "tcp"
    ports    = ["80", "8080", "1000-3000"]
  }
}
```

Naming Convention:

The naming of the resources is done following the sequence as below:

{Environment}-{RegionCode}-{RandomCode}-{AppName}-{Free String}

where the 'RegionCode' is given a dynamic value based on the list of the zones and region codes provided in a table format.

The naming is done by creating a random string resource in main.tf and setting a local variable to reuse in the resources names.

```
resource "random_string" "code" {
  length  = 5
  upper   = false
  number  = false
  lower   = true
  special = false
}

locals {
  code = "${var.environment}-${local.region_code}-${random_string.code.result}-${var.app}"
}

resource "google_container_cluster" "gke_cluster" {
  name = "${local.code}-cluster"
```

For region code, region.json file is created in terraform-gcp-kubernetes folder providing the list of all available zones with their region codes. With the help of local variable in variables.tf, it is reused in the namings.

```

> _modules > terraform-gcp-kubernetes > region.json > ...
{
  "asia-east1": "TW",
  "asia-east2": "HK",
  "asia-northeast1": "TK",
  "asia-northeast2": "OS",
  "asia-northeast3": "SL",
  "asia-south1": "MU",
  "asia-south2": "DE",
  "asia-southeast1": "SI",
  "asia-southeast2": "JA",
  "australia-southeast1": "SY",
  "australia-southeast2": "ME",
  "europe-central2": "WR",
  "europe-north1": "HA",
  "europe-west1": "BE",
  "europe-west2": "UK",
  "europe-west3": "FR",
  "europe-west4": "NH",
  "europe-west6": "SW",
  "northamerica-northeast1": "QU",
  "northamerica-northeast2": "IT",
  "southamerica-east1": "SP",
  "southamerica-west1": "SC",
  "us-central1": "io",
  "us-east1": "NA",
  "us-east4": "VG",
  "us-west1": "OG",
  "us-west2": "LA",
  "us-west3": "LK",
  "us-west4": "LV"
}

locals {
  region_code = jsondecode(file("${path.module}/../../_modules/terraform-gcp-kubernetes/region.json"))[var.region]
}

variable "region" {
  description = "Region code"
  default = "us-central1"
}

```

Step 6:

- The first resource created after setting up in terraform is VPC network as the created VPC network is used as default network to create the cluster. In order to create the VPC network in GCP, first the navigation is done to vpc folder of profiles folder. Then, the following terraform commands are applied to format the configuration files, initialize the working directory, and apply the plan respectively as shown below:

```

terraform fmt
terraform init
terraform apply -var-file=vpc.tfvars

```

After the successful compilation, the newly created VPC network can be seen in GCP VPC networks list below default as seen below:

| VPC networks | CREATE VPC NETWORK | REFRESH |
|-------------------------------|-------------------------|---------------------|
| | europa-central2 | default |
| | northamerica-northeast2 | default |
| | asia-south2 | default |
| | australia-southeast2 | default |
| | southamerica-west1 | default |
| ▼ dev-io-yfvaevogghk-calc-vpc | 29 | 1460 |
| | us-central1 | dev-io-vfvaevniakh- |

The same VPC network is used as a default network while creating cluster and applying firewall rule.


```
variable "vpc_network_name" {
  description = "VPC network name"
  default = "dev-io-yfvaevoiggkh-calc-vpc"
}
```

Step 7:

- The remaining resources: cluster, nodepool and firewall are created in GCP following the same procedure in step 6. First, the working directory is changed to kubernetes folder of profiles folder and the commands are given as follow:

```
terraform fmt
terraform init
terraform apply -var-files=backend.tfvars
```

The successful compilation results the resources created in GCP as shown below:

Kubernetes clusters

+

CREATE

+

DEPLOY

↺

REFRESH

OVERVIEW

COST OPTIMIZATION

PREVIEW

Filter

Enter property name or value

| <input type="checkbox"/> Status | Name ↑ | Location | Number of nodes | Total vCPUs | Total memory | Notifications |
|-------------------------------------|----------------------------|---------------|-----------------|---------------------|-------------------------|----------------------------|
| <input checked="" type="checkbox"/> | dev-io-piqbo-calc-cluster | us-central1-c | 1 | 1 | 3.75 GB | |
| <input type="checkbox"/> | dev-io-isqfk-calc-firewall | Ingress | Apply to all | IP ranges: 10.128.0 | tcp:80, 1000-3000, 8080 | Allow |
| | | | | | 1000 | dev-io-yfvaevoggl-calc-vpc |

Step 7:

- When all the resources are ready, the deployment of the application can be done by creating a deployment.yaml file in deployment folder inside a kubernetes folder. This yaml file creates a deployment server and pulls the image from the container registry to deploy. In this case, the replica is set to one.

```

t > profiles > kubernetes > deployment > ! deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: calc-server
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      run: calc-server
  template:
    metadata:
      labels:
        run: calc-server
    spec:
      containers:
        - image: gcr.io/project-terraform-337113/react-calc-image:final
          name: calc-server
          ports:
            - containerPort: 80
              protocol: TCP

```

In order to create this server, the folder is changed to deployment folder and the connection to the cluster is made to run the kubectl commands:

```
gcloud container clusters get-credentials dev-io-piqbo-calc-cluster --zone us-central1-c --project project-terraform-337113
```

After creating the connection, the following command is executed to run the deployment.yaml file to create the deployment server in GCP.

```
$kubectl apply -f deployment.yaml
```

The created server can be checked from Kubernetes Engine workloads as shown below:

Workloads
REFRESH
DEPLOY
DELETE

Cluster
Namespace
RESET
SAVE

Workloads are deployable units of computing that can be created and managed in a cluster.

OVERVIEW
COST OPTIMIZATION
PREVIEW

Filter
Is system object: False
Filter workloads

| <input type="checkbox"/> | Name ↑ | Status | Type | Pods | Namespace | Cluster |
|--------------------------|-------------|--------|------------|------|-----------|---------------------------|
| <input type="checkbox"/> | calc-server | OK | Deployment | 1/1 | default | dev-io-piqbo-calc-cluster |

Since, the deployment is not yet public, a services.yaml file is created of type load balancer to get the access publicly in the same deployment folder with the target port of

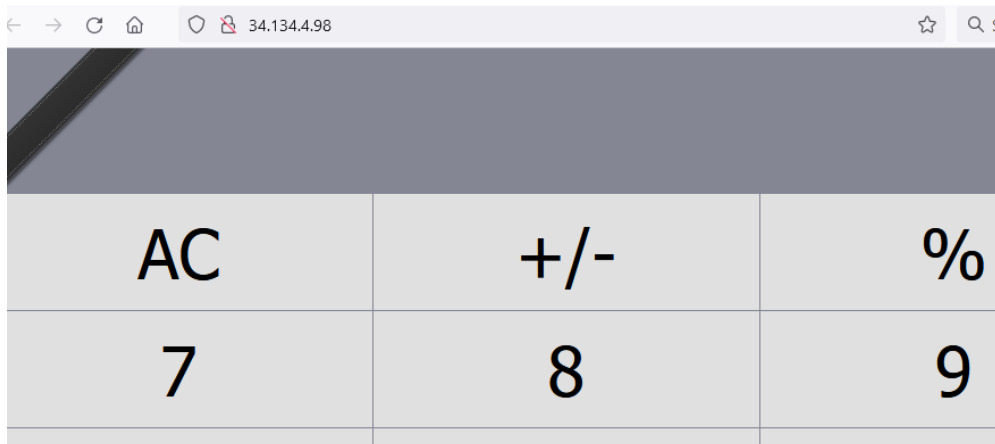
3000 as the docker image port is exposed to 3000 in Step 3. This file is also run with the command `$kubectl apply -f services.yaml`

```
> profiles > kubernetes > deployment > ! services.yaml
apiVersion: v1
kind: Service
metadata:
  name: calc-server
  namespace: default
spec:
  ports:
    - port: 80
      protocol: TCP
      targetPort: 3000
  selector:
    run: calc-server
  type: LoadBalancer
```

The successful compilation results the output in Kubernetes Engine Services & Ingress section as below:

| Filter Is system object: False Filter services and ingresses | | | | | | | |
|---|---------------------|--------|------------------------|----------------|------|-----------|---------------------------|
| <input type="checkbox"/> | Name ↑ | Status | Type | Endpoints | Pods | Namespace | Clusters |
| <input type="checkbox"/> | calc-server | OK | External load balancer | 34.134.4.98:80 | 1/1 | default | dev-io-piqbo-calc-cluster |

The endpoints address of the load balancer opens the application deployed in the new browser as seen below:



Troubleshooting deployment error:

While creating the deployment server, 'Crashloopbackoff' error is encountered in starting the container.

```
C:\Users\Mala Shrestha\Documents\Firstproject\project\profiles\kubernetes\deployment>kubect1 get pod
NAME                                READY   STATUS             RESTARTS   AGE
calc-server-67bc69bb88-xt55x        0/1     CrashLoopBackOff    13          43m
```

The log file of the pod results something like this:

```
C:\Users\Mala Shrestha\Documents\Firstproject\project\profiles\kubernetes\deployment>kubect1 logs c
-xt55x

> calculator@0.1.0 start
> react-scripts start

←[34m@←[39m ←[90m@wds@←[39m: Project is running at http://10.28.1.10/
←[34m@←[39m ←[90m@wds@←[39m: webpack output is served from /calculator
←[34m@←[39m ←[90m@wds@←[39m: Content not from webpack is served from /app/public
←[34m@←[39m ←[90m@wds@←[39m: 404s will fallback to /calculator/
Starting the development server...
```

This error is resolved by changing the version of “react-scripts” to 5.0.0 in package.json file of the application. The docker image is then created again and pushed to the container registry which gets deployed without throwing the same error again.

```
"devDependencies": {
  "chai": "^4.2.0",
  "gh-pages": "^2.0.1",
  "prettier": "^1.17.1",
  "react-scripts": "^5.0.0"
}
```