Setting up and deploying two simple microservices in **Express.js** using **Jenkins**, **Traefik**, **Kubernetes**, and **Google Cloud Platform (GCP)** with a load balancer is a complex process that involves multiple steps. Below is a tutorial that walks through the process, from coding the microservices to deploying them on GCP using Kubernetes.

**Prerequisites:**

* Basic knowledge of **Node.js**, **Express.js**, and **Docker**.
* An active **Google Cloud** account with billing enabled.
* **kubectl**, **gcloud** CLI, and **Docker** installed.
* A **Jenkins** instance (either local or hosted). Install from official web site and install jdk 21.
* **Traefik** configured as an ingress controller for Kubernetes. helm repo add traefik https://helm.traefik.io/traefik
* helm repo update

**1. Create Two Simple Express.js Microservices**

We will start by building two simple Express.js microservices (service1 and service2).

**Microservice 1 (service1)**

*Create a directory for service1 and initialize the project.*

mkdir service1

cd service1

npm init -y

*Install Express.js.*

npm install express

Create the index.js for the service:

javascript

Skopiuj kod

// service1/index.js

const express = require('express');

const app = express();

const port = 3000;

app.get('/', (req, res) => {

res.send('Hello from Service 1!');

});

app.listen(port, () => {

console.log(`Service 1 is running on port ${port}`);

});

**Microservice 2 (service2)**

Repeat the same steps for service2.

bash

Skopiuj kod

mkdir ../service2

cd ../service2

npm init -y

npm install express

Create the index.js for service2:

javascript

Skopiuj kod

// service2/index.js

const express = require('express');

const app = express();

const port = 3000;

app.get('/', (req, res) => {

res.send('Hello from Service 2!');

});

app.listen(port, () => {

console.log(`Service 2 is running on port ${port}`);

});

**2. Dockerize the Microservices**

For deployment on Kubernetes, we need to create Docker images for both services.

**Create Dockerfile for each service**

In both the service1 and service2 directories, create a Dockerfile with the following contents:

dockerfile

Skopiuj kod

# Dockerfile

FROM node:14-alpine

WORKDIR /app

COPY package\*.json ./

RUN npm install

COPY . .

EXPOSE 3000

CMD ["node", "index.js"]

**Build and Push Docker Images**

Use Docker to build and push the images to a container registry like **Google Container Registry (GCR)**.

1. **Authenticate with Google Cloud**:

bash

Code:

gcloud auth configure-docker

1. **Build and push images**:

bash

Code:

# In service1 directory

docker build -t gcr.io/clever-spirit-417020/service1:latest .

docker push gcr.io/clever-spirit-417020/service1:latest

# In service2 directory

docker build -t gcr.io/clever-spirit-417020/service2:latest .

docker push gcr.io/clever-spirit-417020/service2:latest

**3. Kubernetes Setup on GCP**

**3.1 Create a Kubernetes Cluster on GCP**

1. **Create a Kubernetes cluster** using Google Kubernetes Engine (GKE):

*Code:*

gcloud container clusters create microservices-cluster --zone <your-zone>

1. **Configure kubectl to connect to your cluster**:

*Code*:

gcloud container clusters get-credentials microservices-cluster --zone <your-zone>

**3.2 Create Kubernetes Deployment and Service Manifests**

Create a directory k8s-manifests/ where we will store the Kubernetes manifests.

**Deployment for service1:**

yaml

Code:

# k8s-manifests/service1-deployment.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: service1

spec:

replicas: 2

selector:

matchLabels:

app: service1

template:

metadata:

labels:

app: service1

spec:

containers:

- name: service1

image: gcr.io/<your-project-id>/service1:latest

ports:

- containerPort: 3000

**Deployment for service2:**

yaml

Code:

# k8s-manifests/service2-deployment.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: service2

spec:

replicas: 2

selector:

matchLabels:

app: service2

template:

metadata:

labels:

app: service2

spec:

containers:

- name: service2

image: gcr.io/<your-project-id>/service2:latest

ports:

- containerPort: 3000

**Services for both:**

yaml

Code:

# k8s-manifests/service1-service.yaml

apiVersion: v1

kind: Service

metadata:

name: service1

spec:

type: ClusterIP

selector:

app: service1

ports:

- protocol: TCP

port: 80

targetPort: 3000

yaml

Code:

# k8s-manifests/service2-service.yaml

apiVersion: v1

kind: Service

metadata:

name: service2

spec:

type: ClusterIP

selector:

app: service2

ports:

- protocol: TCP

port: 80

targetPort: 3000

**3.3 Deploy to Kubernetes**

Apply the deployments and services to your Kubernetes cluster:

bash

Code:

kubectl apply -f k8s-manifests/service1-deployment.yaml

kubectl apply -f k8s-manifests/service1-service.yaml

kubectl apply -f k8s-manifests/service2-deployment.yaml

kubectl apply -f k8s-manifests/service2-service.yaml

**3.4 Traefik Ingress Setup**

**Install Traefik as the Ingress Controller**

1. **Add Traefik Helm repo**:

scoop install helm

Code:

helm repo add traefik https://helm.traefik.io/traefik

helm repo update

1. **Install Traefik**:

*Code:*

helm install traefik traefik/traefik --set service.type=LoadBalancer

**Create an Ingress Resource**

Create an ingress.yaml file to expose both services.

***yaml***

***Code:***

# k8s-manifests/ingress.yaml

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

name: microservices-ingress

annotations:

traefik.ingress.kubernetes.io/router.entrypoints: web

spec:

rules:

- host: service1.example.com

http:

paths:

- path: /

pathType: Prefix

backend:

service:

name: service1

port:

number: 80

- host: service2.example.com

http:

paths:

- path: /

pathType: Prefix

backend:

service:

name: service2

port:

number: 80

Apply the ingress to the cluster:

bash

Code:

kubectl apply -f k8s-manifests/ingress.yaml

**3.5 Configure DNS (Optional)**

You can configure your DNS to point to the external IP of the Traefik load balancer. Obtain the external IP:

*Code:*

kubectl get svc traefik

Set up DNS records for service1.example.com and service2.example.com to point to this IP.

**4. Jenkins CI/CD Pipeline**

**4.1 Setup Jenkins Pipeline**

1. **Install Jenkins plugins**:
   * **Kubernetes** plugin.
   * **Docker** plugin.
   * **Pipeline** plugin.
   * Google Container Registry Auth
2. **Create a Jenkins pipeline** job with the following pipeline script:

pipeline {

agent any

stages {

stage('Build') {

steps {

script {

docker.build("gcr.io/<your-project-id>/service1:latest", "./service1")

docker.build("gcr.io/<your-project-id>/service2:latest", "./service2")

}

}

}

stage('Push to GCR') {

steps {

script {

docker.withRegistry('https://gcr.io', 'gcr:auth') {

docker.image("gcr.io/<your-project-id>/service1:latest").push()

docker.image("gcr.io/<your-project-id>/service2:latest").push()

}

}

}

}

stage('Deploy to Kubernetes') {

steps {

sh 'kubectl apply -f k8s-manifests/service1-deployment.yaml'

sh 'kubectl apply -f k8s-manifests/service2-deployment.yaml'

}

}

}

}

**5. Access Your Microservices**

After deployment, you can access your microservices via the domain names configured for Traefik:

* http://service1.example.com
* http://service2.example.com

**6. Load Balancer**

GCP’s Kubernetes Engine automatically provisions a load balancer for Traefik. This setup provides high availability and distributes traffic across the pods of each service.

To push Docker images to Google Container Registry (GCR) using Jenkins, the setup is quite similar to Artifact Registry. You'll still need the Google Cloud SDK plugin, Docker Pipeline plugin, and some other key configurations.

Here's a sample Jenkins Pipeline script for pushing Docker images to GCR:

**Required Jenkins Plugins**

1. **Google Cloud SDK Plugin**
2. **Docker Pipeline Plugin**
3. **Credentials Binding Plugin**
4. **Pipeline Plugin**

**Steps to Set Up**

1. **Service Account Creation and Key**:
   * In **Google Cloud Console**, create a service account with the **Storage Admin** role (for GCR).
   * Download the service account key (JSON format) and add it to **Jenkins > Manage Jenkins > Manage Credentials** under **Global credentials**.
2. **Pipeline Script**:

*groovy*

pipeline {

agent any

environment {

GOOGLE\_APPLICATION\_CREDENTIALS = credentials('your-google-credentials-id') // Jenkins ID for your Google Cloud credentials

PROJECT\_ID = 'your-gcp-project-id'

IMAGE\_NAME = 'your-image-name'

TAG = 'latest'

REGION = 'gcr.io' // You can also specify regions like us.gcr.io, eu.gcr.io, or asia.gcr.io if needed

}

stages {

stage('Authenticate to Google Cloud') {

steps {

sh 'gcloud auth activate-service-account --key-file=$GOOGLE\_APPLICATION\_CREDENTIALS'

sh 'gcloud config set project $PROJECT\_ID'

}

}

stage('Build Docker Image') {

steps {

script {

docker.build("$REGION/$PROJECT\_ID/$IMAGE\_NAME:$TAG")

}

}

}

stage('Push Docker Image to GCR') {

steps {

script {

docker.withRegistry("https://$REGION", 'gcloud') {

docker.image("$REGION/$PROJECT\_ID/$IMAGE\_NAME:$TAG").push()

}

}

}

}

}

}

**Explanation of the Script**

* **Authenticate to Google Cloud**: Uses the service account JSON key to authenticate.
* **Build Docker Image**: Builds the Docker image using the docker.build() method.
* **Push Docker Image to GCR**: Pushes the built Docker image to GCR using docker.withRegistry().

**Important Notes**

* Make sure Docker is set up on your Jenkins instance and configured to interact with GCR.
* Replace placeholders like your-google-credentials-id, your-gcp-project-id, your-image-name, and latest with your specific information.

This should allow you to push Docker images to GCR smoothly. Let me know if you have any questions on customizing it!

To push Docker images to Google Artifact Registry using Jenkins, you need to configure Jenkins with a few plugins and credentials for smooth integration with Google Cloud. Here’s a guide on the plugins and steps to set it up:

Required Plugins

Google Cloud SDK Plugin: This lets you interact with Google Cloud services, including Artifact Registry.

Docker Pipeline Plugin: Provides Docker support in Jenkins Pipeline (specifically, if you’re using declarative pipelines to build and push Docker images).

Credentials Binding Plugin: Facilitates securely binding credentials, like Google service account keys, in Jenkins pipelines.

Pipeline Plugin: If you’re working with scripted or declarative pipelines, this plugin helps to write pipeline scripts to automate the Docker build and push process.

Google OAuth Credentials Plugin (optional): If you’re using OAuth for Google service authentication, this plugin will help manage those credentials.

Setup Steps

Install the Plugins: In Jenkins, go to Manage Jenkins > Manage Plugins. Search for each plugin by name and install them.

Add Google Cloud Service Account Key:

In Google Cloud Console, create a service account with permissions for Artifact Registry (usually Artifact Registry Writer).

Generate a JSON key for this service account.

In Jenkins, go to Manage Jenkins > Manage Credentials, and add the JSON key under Global credentials.

Configure Docker and Google Cloud in the Pipeline: Use the following basic example pipeline script to log in, build, and push to Google Artifact Registry:

groovy

Skopiuj kod

pipeline {

agent any

environment {

GOOGLE\_APPLICATION\_CREDENTIALS = credentials('your-google-credentials-id') // Replace with your Jenkins credentials ID for Google Cloud

PROJECT\_ID = 'your-gcp-project-id'

REGION = 'your-artifact-registry-region' // e.g., us-central1

REPOSITORY = 'your-artifact-repository'

IMAGE\_NAME = 'your-image-name'

TAG = 'latest'

}

stages {

stage('Authenticate to Google Cloud') {

steps {

sh 'gcloud auth activate-service-account --key-file=$GOOGLE\_APPLICATION\_CREDENTIALS'

sh 'gcloud config set project $PROJECT\_ID'

}

}

stage('Build Docker Image') {

steps {

script {

docker.build("${REGION}-docker.pkg.dev/${PROJECT\_ID}/${REPOSITORY}/${IMAGE\_NAME}:${TAG}")

}

}

}

stage('Push Docker Image') {

steps {

script {

docker.withRegistry("https://${REGION}-docker.pkg.dev", 'gcloud') {

docker.image("${REGION}-docker.pkg.dev/${PROJECT\_ID}/${REPOSITORY}/${IMAGE\_NAME}:${TAG}").push()

}

}

}

}

}

}

This setup should get you started with pushing Docker images to Google Artifact Registry directly from Jenkins. Let me know if you’d like to dive into more details on any of these steps!

**no matches for kind "IngressRoute" in version "traefik.containo.us/v1alpha1"**

The error message indicates that the IngressRoute resource kind is not recognized by your Kubernetes cluster, which likely means the Traefik CRDs (Custom Resource Definitions) are not installed. To use IngressRoute with Traefik, you must first install the necessary CRDs that define Traefik-specific resources like IngressRoute.

**Step 1: Install Traefik CRDs**

If you installed Traefik using Helm, it should have created the CRDs automatically. However, if you installed it manually or are missing the CRDs, you can add them by applying the following YAML file directly from Traefik's GitHub repository.

Run the following command to install the Traefik CRDs:

bash

Skopiuj kod

kubectl apply -f <https://raw.githubusercontent.com/traefik/traefik/v2.10/docs/content/reference/dynamic-configuration/kubernetes-crd-definition-v1alpha1.yaml>

or

kubectl apply -f https://raw.githubusercontent.com/traefik/traefik/v3.2/docs/content/reference/dynamic-configuration/kubernetes-crd-definition-v1.yml

This file contains all necessary CRDs, including IngressRoute, Middleware, TraefikService, and more. Ensure the version of Traefik you’re using matches the URL version above (adjust the version if needed).

**Step 2: Verify CRDs**

After applying the CRDs, you can verify that the IngressRoute kind is available by running:

bash

Skopiuj kod

kubectl get crds | grep traefik

You should see output similar to:

plaintext

Skopiuj kod

ingressroutes.traefik.containo.us

ingressroutetcps.traefik.containo.us

ingressrouteudps.traefik.containo.us

middlewares.traefik.containo.us

traefikservices.traefik.containo.us

**Step 3: Apply the IngressRoute Configuration Again**

Now that the CRDs are installed, you should be able to apply your IngressRoute manifests:

bash

Skopiuj kod

kubectl apply -f k8s-manifests/ingress.yaml

If errors change the first line to apiVersion: traefik.io/v1alpha1

This should work without errors, allowing Traefik to manage your routes based on the IngressRoute configuration.