HELLO NODE KUBERNETES:

Create the backend and run using node servername

Create Docker Container Image:
.. write a docker file
.. build the image
.. run the container and see the backend running.
.. push the working image to docker hub/ in this lab
.. its google artifact registry..

This thing need attention? *gcloud artifacts repositories create my-docker-repo* \
--repository-format=docker \
--location=us-west1 \
--project=qwiklabs-gcp-02-404ec7e1e03f..
How can this be used locally?
Then this..
gcloud auth configure-docker
Then ..
docker tag hello-node:v1
us-west1-docker.pkg.dev/project_Id/my-docker-repo/hello-node:v1
Followed by 😣
docker push
us-west1-docker.pkg.dev/project_id/my-docker-repo/hello-node:v1

Now you have a project-wide Docker image available which Kubernetes can access and orchestrate.

1. Create Your Kubernetes Cluster 🏗️

A Kubernetes cluster provides the environment to run your containerized applications. It consists of a **control plane** (the "master") managed by Google and **worker nodes**, which are the virtual machines that actually run your application containers.

First, you need to tell the gcloud command-line tool which Google Cloud project you're working with.

Bash

Set your active project

gcloud config set project YOUR_PROJECT_ID

Next, create the cluster. This command provisions the necessary compute resources.

Bash

Create a GKE cluster named 'hello-world'

gcloud container clusters create hello-world \

--num-nodes 2 \

--machine-type e2-medium \

--zone "us-west1-b"

- --num-nodes 2: Creates a cluster with **two worker nodes**.
- --machine-type e2-medium: Specifies the size (CPU/RAM) of each worker node.
- --zone "us-west1-b": Defines the physical location where your cluster will be created.

2. Create Your Pods via a Deployment 🚀

You don't usually create Pods directly. Instead, you create a **Deployment**, which is a Kubernetes object that manages a set of identical Pods. It ensures that a specified number of Pods are always running and handles updates and rollbacks.

Bash

Create a deployment named 'hello-node' from a container image

kubectl create deployment hello-node \

--image=us-west1-docker.pkg.dev/YOUR_PROJECT_ID/my-docker-repo/hello-node:v1

- This command tells Kubernetes to pull the specified container image and run it in a Pod managed by the hello-node deployment.

3. Inspect and Troubleshoot Your Application 🔍

Once you've created resources, you need to verify they are running correctly.

Key Inspection Commands:

Bash

Check the status of your deployments

kubectl get deployments

See the individual pods created and managed by the deployment

kubectl get pods

Get a high-level overview of your cluster's health and endpoints

kubectl cluster-info

View your current kubectl configuration (e.g., which cluster you're connected to)

kubectl config view

Key Troubleshooting Commands:

Bash

See all events in the cluster (useful if a pod is stuck or failing)

kubectl get events

View the logs from a specific pod to debug application errors

kubectl logs <pod-name>

4. Allow External Traffic 🌐

By default, your application is only accessible from within the cluster. To expose it to the internet, you create a **Service**. A Service provides a stable network endpoint (IP address and port) for your set of Pods.

Bash

Expose the 'hello-node' deployment using a LoadBalancer Service

kubectl expose deployment hello-node --type="LoadBalancer" --port=8080

- --type="LoadBalancer": This is crucial. It tells the cloud provider (GCP) to provision an external network load balancer, which will get a public IP address and route external traffic to your Pods.

To find the public IP address, run:

Bash

Get information about your services, including the external IP

kubectl get services

Look for the EXTERNAL-IP address. It might take a few minutes to appear. Once it does, you can access your app at http://<EXTERNAL_IP>:8080.

5. Scale Up Your Service 📈

To handle more traffic or ensure high availability, you can easily increase the number of Pods running your application.

Bash

Scale the number of replicas in the deployment to 4

kubectl scale deployment hello-node --replicas=4

Verify that 4 pods are now running

kubectl get pods

The Deployment and the LoadBalancer Service will automatically handle distributing traffic across all four pods.

6. Roll Out an Upgrade 🔄

To update your application to a new version, you first need to build and push your new container image (v2) to the registry.

Bash

1. Tag the new version of your image

docker tag hello-node:v2 us-west1-docker.pkg.dev/YOUR_PROJECT_ID/my-docker-repo/hello-node:v2

2. Push the new image to the Artifact Registry

docker push us-west1-docker.pkg.dev/YOUR_PROJECT_ID/my-docker-repo/hello-node:v2

Now, you can tell Kubernetes to perform a **rolling update**. This updates the Pods one by one without any downtime for your users.

Bash

3. Update the deployment to use the new image version

kubectl set image deployment/hello-node hello-node=us-west1-docker.pkg.dev/YOUR_PROJECT_ID/my-docker-repo/hello-node:v2

Kubernetes will terminate old Pods and create new ones with the v2 image until the entire deployment is updated.

🖥️ Local Solution: Running This on Your Linux Terminal

You asked what can be done without gcloud. You can replicate this entire workflow on your local machine! The kubectl commands are universal, but you need a local Kubernetes cluster instead of GKE.

Popular tools for local Kubernetes clusters include **Minikube**, **Kind**, and **Docker Desktop's built-in Kubernetes**. Here's the workflow using **Minikube**.

Step 1: Install a Local Cluster

First, install kubectl, Docker, and Minikube. Then, start your local cluster:

Bash

Start a single-node Kubernetes cluster on your machine

minikube start

Step 2: Build & Load Your Docker Image

Instead of pushing your image to a remote registry like GCR, you can build it locally and load it directly into your Minikube cluster's Docker environment.

Bash

This command points your terminal's Docker daemon to the one inside Minikube

eval $(minikube docker-env)

Now, build your image. It will be immediately available to the cluster.

docker build -t hello-node:v1 .

To switch back to your host's Docker daemon when you're done:

eval $(minikube docker-env -u)

Step 3: Run kubectl Commands (They're the Same!)

All the kubectl commands you used before will work exactly the same way against your local cluster.

Bash

Create the deployment using the locally built image

kubectl create deployment hello-node --image=hello-node:v1

Inspect it

kubectl get pods

Scale it

kubectl scale deployment hello-node --replicas=4

Step 4: Expose The Service (The Main Difference)

A local environment doesn't have a cloud load balancer. So, type="LoadBalancer" behaves differently. The two common ways to access your service locally are:

Using minikube service (Easiest): This command automatically creates a tunnel and opens the service URL in your browser. It's the recommended way for Minikube.
Bash
Expose the service. Minikube will handle the LoadBalancer type.

kubectl expose deployment hello-node --type="LoadBalancer" --port=8080

Let Minikube open the service for you

minikube service hello-node

1.

Using NodePort: This exposes the service on a specific port on the cluster's node IP.
Bash
Expose using a NodePort instead of LoadBalancer

kubectl expose deployment hello-node --type=NodePort --port=8080

Find the URL for the service

minikube service hello-node --url

2.

This local setup is perfect for development, testing, and learning Kubernetes fundamentals without incurring any cloud costs.