Show Submission Credentials

P2. Kubernetes and Container Orchestration Concepts and Hands-on Practice with Kubernetes.

- ✔ Introduction to Kubernetes (k8s)
- ✔ YAML Overview
- ✔ Kubernetes on Azure
- ✔ Kubernetes on GCP
- ✔ Introduction to Ingress
- ✔ Introduction to Helm
- ✔ How to Develop Helm Charts
- ✔ Introduction to Volumes
- ✔ Debugging Kubernetes
- ✔ Conclusion

Introduction to Kubernetes (k8s)

What is Kubernetes?

Information

Before proceeding with this primer, please make sure that you have finished the **Intro to Containers and Docker** primer.

> Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications.

In the **Intro to Containers and Docker** primer, you've already learned about the power of containers.

The traditional way of hosting an application is to deploy the application to VMs, however, this approach is heavy-weight and it often entangles the application with the VMs. To host applications on a VM, you often need to configure the VM beforehand to set up the environment for the application. As the VMs are mutable, unless there is a rigid and professional DevOps process to automate the VM configuration and application deployment while preventing developers from performing manual operations on the VMs, the VMs will often deteriorate into black boxes that become hard to reproduce over time because of the accumulated manual operations made to the VMs.

The new way is to containerize the applications. With Dockerfiles, the environment is fully reproducible and the deployment is automated. Also, as a container only runs one application, the environments are isolated from other applications.

Containers work best for service-based architectures. You are encouraged to break down monolithic architectures into separate components. Componentization allows you to develop smaller codebases in a faster and more reliable manner, and each service is easier to test and maintain.

However, there is a challenge to resolve for containerized solutions: you need to orchestrate, assemble, and manage containers to achieve the same functionality as you can get with a monolithic application. Kubernetes (K8s) is the solution to automating application deployment, scaling, and management as containerized services.

Kubernetes builds upon years of experience of running production workloads at Google (https://queue.acm.org/detail.cfm?id=2898444) and since its release, has quickly become the dominant platform for container orchestration. Some of its best features include:

- **Managing the whole cluster as if it is a single logical unit**. This means that all the underlying hardware, software and network protocol are hidden, and the user is able to interact with the Kubernetes cluster using a few commands or programming interfaces.
- Automatically placing containers across machines to distribute the workload and optimize utilization.

- Easily deploying and horizontally scaling on the fly.
- Automatically handling rollouts and rollbacks of any change to the configuration or system settings of the containers. A mistake in the new configuration that would kill the containers would not happen at once, and Kubernetes is able to roll back to the previous version.
- Easy deployment of Kubernetes on major cloud vendors such as AWS, Azure and GCP, with support to services like Load Balancing.
- Automated health check and container replacement on failure.
- Security control with easy-to-manage secret configuration.

Conceptual Overview

Below we will elaborate on the core concepts of Kubernetes.

Kubernetes Cluster

A Kubernetes Cluster is a collection of physical or virtual machines. It consists of a master node and multiple worker nodes. In short, the master node is coordinating the cluster, while the worker nodes run applications.

Master Node

The master node is the point of user interaction with the cluster. It runs Kubernetes cluster services which manage the worker nodes and distributes containers across the workers. The master node runs three main infrastructure services:

1. The **API Server (`kube-apiserver`)** is the central point of management for the cluster. It also assigns pods (which will be discussed later) to worker nodes.
2. The **Controller Manager (`kube-controller-manager`)** handles the replication process. Whenever a change is required, the controller manager implements the appropriate procedure, whether it involves scaling resources up or scaling them down.
3. The **Scheduler (`kube-scheduler`)** assigns workloads to specific nodes in the cluster.

Worker Nodes

The worker nodes run the various pods that perform application-specific processing. Each node additionally runs these services:

1. The **Kubelet Daemon (`kubelet`)** which communicates with the master server API and ensures that the containers it is assigned to are in the desired state.
2. The **Kube-Proxy (`kube-proxy`)** which routes traffic to the appropriate container based on IP and port number of the incoming request.

Kubernetes API Objects

Kubernetes facilitates declarative configuration and automation. The user defines their desired cluster state in a YAML configuration file and sends it to Kubernetes, which manages the containers to ensure the state of the cluster continually matches the desired state of the users.

Kubernetes is based on "desired state management". *Kubernetes Objects* are used to represent the state of your cluster. These objects can be expressed in the YAML format.

Kubernetes objects can describe:

- What containerized applications are running and on which nodes
- The resources available to the applications
- The policies around how those applications behave, such as fault-tolerance

Defining these objects informs Kubernetes of the **desired** state of the cluster. Kubernetes will constantly ensure that the appropriate resources are launched or updated to meet the desired state.

Every Kubernetes object has a set of common fields that specify what resource is being requested and describes the characteristics of that resource. These common fields include:

- ***spec*** - Describes the **desired** state for the object; the characteristics that you want the object to have. This is provided by the author of the YAML configuration file.
- ***status*** - Describes the **actual** state of the object.
- ***apiVersion*** - Which version of the Kubernetes API you're using to create this object.
- ***kind*** - What kind of object you want to create
- ***metadata*** - Data that helps uniquely identify the object, including a `name` string, UID, and optional `namespace`.

To create a Kubernetes object, the required fields you need to provide are `apiVersion`, `kind`, `metadata`, and `spec`. `status` is supplied and updated by the Kubernetes system. At any given time, the Kubernetes Control Plane actively manages an object's actual state to match the desired state you supplied.

This field is supplied and updated by the Kubernetes system. At any given time, Kubernetes constantly manages an object's actual state to match the desired state you supplied.

Below we will review a sample Kubernetes Deployment (https://kubernetes.io/docs/concepts/workloads/controllers/deployment/) object provided by GCP in the GoogleCloudPlatform/kubernetes-engine-samples (https://github.com/GoogleCloudPlatform/kubernetes-engine-samples/tree/master/hello-app) repository.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: helloweb
  labels:
    app: hello
spec:
  selector:
    matchLabels:
      app: hello
      tier: web
  template:
    metadata:
      labels:
        app: hello
        tier: web
    spec:
      containers:
      - name: hello-app
        image: gcr.io/google-samples/hello-app:1.0
        ports:
        - containerPort: 8080
```

In this Kubernetes Deployment object we specify various common fields:

- ***apiVersion*** - `apps/v1` ; the API version for each object is defined by Kubernetes and can be found in the API documentation.
- ***kind*** - This object is of the `Deployment` kind.
- ***metadata*** - The deployment is given a `name` as `helloweb` and a `label` as `app: hello` , this object will be identified and referenced by the metadata.
- ***spec*** - You will notice that there are two `spec` fields in this YAML document. The outer `spec` belongs to the Deployment object, and the inner `spec` within `template` is an inline reference to a Container object.
    - ***spec.template.spec.containers*** - `containers` is an array of Container (https://kubernetes.io/docs/concepts/containers/) objects. For the container defined under `spec.template.spec.containers` , a `name` is specified to identify the container, a Docker `image` are specified to direct Kubernetes of which image to launch, and `ports` describes the ports that are requested by the container.

Pods

A Pod is a group of containers in Kubernetes. A Pod is the smallest deployable unit of deployment in the Kubernetes object model. The containers in a Pod share the same storage and are under the same IP address. It is common to have multiple containers work together to produce a single artifact. Pods provide a clean way to package containers together. You may think of peas in a pod.

Putting the concepts above together, the diagram illustrates a simplified Kubernetes cluster.

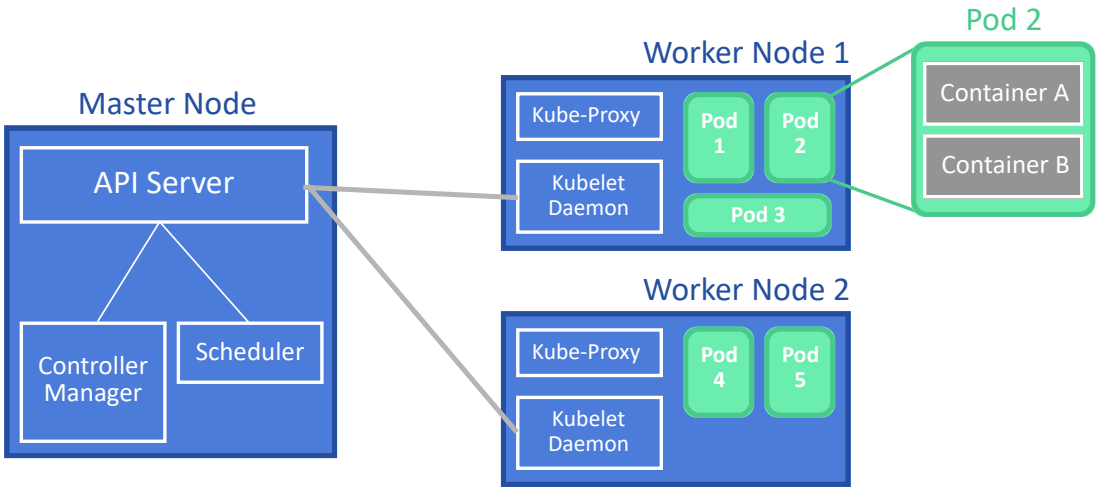


Figure 1: A Simple Kubernetes Cluster

Additional Resources

- The team at Deis has released a short video that goes into more depth about how the objects in Kubernetes are related to one another.

Video 1: The Illustrated Children's Guide to Kubernetes

- Kubernetes API Overview (https://kubernetes.io/docs/concepts/overview/kubernetes-api/). Each version of Kubernetes has its published API documentation. You can refer to the API reference (https://kubernetes.io/docs/reference/#api-reference) to learn the object structure and review sample YAML files.
- For a more advanced overview of the Kubernetes architecture, the kelseyhightower/kubernetes-the-hard-way (https://github.com/kelseyhightower/kubernetes-the-hard-way) repository explores the in-depth concepts of using Kubernetes.

References:

1. Standardized Glossary - Kubernetes (https://kubernetes.io/docs/reference/glossary/?all=true)
2. Pods - Kubernetes (https://kubernetes.io/docs/user-guide/pods/)
3. Kubernetes in 5 mins - VMware Cloud-Native Apps (https://www.youtube.com/watch?v=PH-2FfFD2PU)
4. Kubernetes Webinar Series - Getting Started with Kubernetes (https://www.youtube.com/watch?v=_vHTaIJm9uY)

YAML Overview

YAML: YAML Ain't Markup Language

> What It Is: YAML is a human-friendly data serialization standard for all programming languages. [1]

Kubernetes objects can be expressed in the YAML format. This section will cover the basic usage of YAML.

For more advanced usage of YAML, you may refer to the YAML 1.2 specification (http://www.yaml.org/spec/1.2/spec.html).

Indentation

Indentation is a fundamental concept in YAML, serving a similar purpose to the indentation in Python; as such, an incorrect indentation may result in incorrectly formatted YAML documents or documents with an incorrect structure.

As specified in the Indentation Spaces (http://www.yaml.org/spec/1.2/spec.html#id2777534) section, the main concepts of indentation in YAML are:

1. ***Indentation is defined as zero or more space characters at the start of a line.***
2. ***Tab characters must NOT be used in the indentation.***
3. ***Each node must be indented further than its parent node.***
4. ***All sibling nodes must use the exact same indentation level.***

Collections

One common structure used in YAML documents is block collections (http://www.yaml.org/spec/1.2/spec.html#id2759963).

A sequence of scalars

The following YAML file defines a sequence of scalar values (a list type structure). Additionally, you will note that YAML comments (http://www.yaml.org/spec/1.2/spec.html#id2780069) begin with the **#** character and that a sequence of three dashes (**---**) is used to indicate the start of a document (http://www.yaml.org/spec/1.2/spec.html#id2760395).

```
# This is a comment in a YAML document

# YAML uses three dashes ("---") to signify the start of a
# document and may appear multiple times in a file

# A list of major cloud service providers
---
- AWS
- Azure
- GCP
```

Mapping scalars to scalars

In addition to constructing lists of scalars, you may map scalars to scalars (i.e., a key-value mapping). The YAML file below shows a mapping of cloud service providers to their full names.

```
# Mapping of a cloud service provider to its full name
---
AWS: Amazon Web Services
Azure: Microsoft Azure Cloud Computing Platform & Services
GCP: Google Cloud Platform
```

Mapping scalars to sequences

Now that we have covered sequences of scalars and mapping of scalars, it's possible to envision mapping scalars to sequences (i.e., named lists). In this example, we will map a cloud service provider to a subset of their cloud services.

```
# Mapping from cloud service provider to a subset of its cloud services
---
AWS:
    - EC2
    - S3
    - ELB
Azure:
    - Azure Container Service
    - HDInsight
    - Blob Storage
GCP:
    - Compute Engine
    - BigQuery
    - Cloud Functions
```

Mapping of mappings

In this example, we demonstrate how to define a structure that has nested mappings. It may help to think of this structure similar to a name object. In the example below, the **t2.nano** object's **vCPU** field is set to **1**.

```
# Mapping EC2 instance types to their descriptions
---
t2.nano:
    vCPU: 1
    ECU: Variable
    Memory: 0.5
    Instance Storage: EBS Only
    Usage Price Per Hour: 0.0058

t2.micro:
    vCPU: 1
    ECU: Variable
    Memory: 1
    Instance Storage: EBS Only
    Usage Price Per Hour: 0.0116
```

YAML Linters

Debugging YAML syntax issues manually can be frustrating.

Fortunately, there are IDE plugins such as the JetBrains Kubernetes plugin (https://plugins.jetbrains.com/plugin/10485-kubernetes) and online applications that validate YAML syntax.

There are also command-line YAML linters:

- adrienverge/yamllint (https://github.com/adrienverge/yamllint)
- Pryz/yaml-lint (https://github.com/Pryz/yaml-lint)
- rasshofer/yaml-lint (https://github.com/rasshofer/yaml-lint)

References:

1. YAML (http://yaml.org/)
2. YAML Ain't Markup Language (YAML™) Version 1.2 (http://www.yaml.org/spec/1.2/spec.html)

Kubernetes on Azure

Azure Kubernetes Service

Azure Kubernetes Service (AKS) manages your hosted Kubernetes environment, making it quick and easy to deploy and manage containerized applications without container orchestration expertise.

This tutorial shows you how to package a web application in a Docker container image, and run that container image on an AKS cluster.

Setup

We need to install Docker on a VM to get started.

First, provision an `Ubuntu 18.04 LTS` VM on `Azure`, and **open the inbound HTTP port 80 and SSH port 22.**

Please remember to tag your resources using `project:containers`.

Note: Please make sure that the VM Image is the Ubuntu LTS, not the student workspace image. In the following steps, you will need run commands with `sudo`, and you will not have the `sudo` permission if you use the student workspace image.

Install Docker

Run the following official convenience script (https://docs.docker.com/install/linux/docker-ce/ubuntu/#install-using-the-convenience-script) to install Docker Engine.

```
curl -fsSL https://get.docker.com -o get-docker.sh && sudo sh get-docker.sh
```

By default, Docker must be run as a root user. If you would like to use Docker as a non-root user, you should now consider adding your user to the "docker" group with something like:

```
# whoami: returns the current Linux user
sudo usermod -aG docker $(whoami)
```

Log out and back into the VM for the command to take effect, or you will get errors such as `docker: Got permission denied while trying to connect to the Docker daemon socket.`

Verify that Docker Engine is installed correctly by running the hello-world image.

```
docker run hello-world

# Expected output:
# ...
# Hello from Docker!
# This message shows that your installation appears to be working correctly.
# ...
```

Install kubectl

```
sudo snap install kubectl --classic
```

Install Azure CLI

Install Azure CLI.

```
curl -sL https://aka.ms/InstallAzureCLIDeb | sudo bash
```

Verify the Azure CLI version. **Note that you need Azure CLI version 2.0.73 or later to get the AKS and ACR integration feature, which is required to complete this tutorial.**

```
az --version
```

Log in to Azure.

```
az login
```

List the subscriptions of your Azure account.

```
az account list --output table
```

Set a subscription to be the current active subscription that you want to provision a container registry with.

```
az account set --subscription <SUBSCRIPTION_ID>
```

Resource providers registration

Registering a resource provider enables your subscription to work with a category of Azure services, for example, `Microsoft.Compute` for virtual machines and `Microsoft.CognitiveServices` for Azure Cognitive Services. By default, many resource providers are automatically registered. However, you may need to manually register some resource providers. The scope for registration is always the subscription.

To work with Azure AKS Cluster with a **new** subscription, you need to first register the following resource provider:

```
az provider register --namespace Microsoft.ContainerService
```

Registering resource provider may take time. Once you execute the command above, we suggest that you move on reading the writeup.

Create an Azure Container Registry

Create a resource group.

```
RESOURCE_GROUP_NAME=aksDemoResourceGroup
az group create --name ${RESOURCE_GROUP_NAME} --location eastus
```

Create a container registry. The registry name must be globally unique (case insensitive) within Azure and contain 5-50 alphanumeric characters. We suggest that you only use lowercase letters and digits and avoid uppercase letters. You can get a globally unique string that consists of lowercase letters and digits at the Online UUID Generator (https://www.uuidgenerator.net/), but you will need to make some modifications such as to remove the dashes.

```
ACR_NAME=<unique_acr_name>
az acr create --name ${ACR_NAME} --resource-group ${RESOURCE_GROUP_NAME} --locat
ion eastus --sku Basic
```

Create a New AKS Cluster with ACR Integration

Create an AKS cluster and set up the integration between the AKS cluster and the ACR instance.

```
CLUSTER_NAME=myAKSCluster
az aks create --name ${CLUSTER_NAME} --resource-group ${RESOURCE_GROUP_NAME} --a
ttach-acr ${ACR_NAME} --generate-ssh-keys
```

If you see an error message that says `--attach-acr` is an unrecognized argument, you need to upgrade the Azure CLI to the latest version.

Build a Container Image

In this tutorial, you will deploy a sample web application called `hello-app` that responds to all requests with the message "Hello, World!" on port 80.

To download the `hello-app` source code including the Dockerfile, run the following commands:

```
git clone https://github.com/GoogleCloudPlatform/kubernetes-engine-samples
cd kubernetes-engine-samples/hello-app
```

Build and tag the container image for uploading.

```
docker build -t ${ACR_NAME}.azurecr.io/clouduser/hello-app:v1 .
```

Upload the container image to ACR

Before pushing and pulling container images, you must log in to the container registry with the `az acr login` command.

```
az acr login --name ${ACR_NAME}
# Expected output:
# Login Succeeded
```

You can now verify the fully qualified name of your ACR login server with the following command.

```
az acr list --resource-group ${RESOURCE_GROUP_NAME} --query "[].{acrLoginServer:
loginServer}" --output table

# Expected output:
AcrLoginServer
----------------------------------------
<lowercased_acr_name>.azurecr.io
```

The login server name is in the format `<lowercased_acr_name>.azurecr.io` (all lowercase). We will use `${ACR_NAME}.azurecr.io` in the following bash commands, with the assumption that your ACR name only contains lowercase letters and digits.

Push the image to the Azure Container Registry.

```
docker push ${ACR_NAME}.azurecr.io/clouduser/hello-app:v1
```

Managing an AKS Cluster

Run the following commands to set up access to the AKS cluster. Note that you should always remember to run this `az aks get-credentials` command once before you can use `kubectl` to manage an AKS cluster.

```
az aks get-credentials --resource-group=${RESOURCE_GROUP_NAME} --name=${CLUSTER_
NAME}
# Expected output:
# Merged "${CLUSTER_NAME}" as current context in .../.kube/config
```

Deploy the Application to the AKS Cluster

Run the `kubectl` command below to create a Deployment named `hello-web` on the AKS cluster.

```
kubectl create deployment hello-web --image=${ACR_NAME}.azurecr.io/clouduser/hel
lo-app:v1
# Expected output:
# deployment.apps/hello-web created
```

After the command finishes, check that the deployment is working correctly:

```
kubectl get deployments
# Expected output:
# NAME        READY   UP-TO-DATE   AVAILABLE   AGE
# hello-web   1/1     1            1           ...
```

Kubernetes represents applications as Pods. In this tutorial, each Pod contains only your `hello-app` container. To see the Pod created by the Deployment, run the following command:

```
kubectl get pods
#
# NAME                          READY   STATUS    RESTARTS   AGE
# <auto-generated-pod-name-1>   1/1     Running   0          66s
```

Expose Applications to the Internet

By default, the containers you run are not accessible from the Internet. You must explicitly expose your application to traffic from the Internet, run the following command:

```
# --port: specifies the port number configured on the Load Balancer
# --target-port: specifies the port number that the hello-app container is liste
ning on.
kubectl expose deployment hello-web --type=LoadBalancer --port 80 --target-port
8080
# Expected output:
# service/hello-web exposed
```

The command above creates a Service resource, which provides networking and IP support to your application's Pods.

AKS assigns the external IP address to the Service resource (not the Deployment). To retrieve the external IP that AKS provisioned, inspect the `EXTERNAL-IP` value of the Service with the `kubectl get service` command. Retry later if the `EXTERNAL-IP` value is `<pending>`.

```
kubectl get service
```

Once you have determined the external IP address for your application. Visit `http://<EXTERNAL-IP>` to verify that the web server returns a plain text response as follows:

```
Hello, world!
Version: 1.0.0
Hostname: <auto-generated-pod-name-1>
```

Scalability and Rolling Updates

Scale up the Application

Run the following command to add two additional replicas to your Deployment for a total of three:

```
kubectl scale deployment hello-web --replicas=3
# Expected output:
# deployment.extensions/hello-web scaled
```

View the new replicas running on your cluster with the following commands:

```
kubectl get deployments
# Expected output:
# NAME        READY   UP-TO-DATE   AVAILABLE   AGE
# hello-web   3/3     3            3           ...

kubectl get pods
# Expected output:
# NAME                          READY   STATUS    RESTARTS   AGE
# <auto-generated-pod-name-3>   1/1     Running   0          ...
# <auto-generated-pod-name-2>   1/1     Running   0          ...
# <auto-generated-pod-name-1>   1/1     Running   0          ...
```

Now, simulate a traffic load by sending 100 web requests to the external IP address for your application.

```
EXTERNAL_IP=<external_ip>
for i in {1..100}; do curl http://${EXTERNAL_IP}/; done
```

You will observe that the `Hostname` part of the text response will differ between these web requests. For each web request, the load balancer you provisioned routes the traffic to one of the replicas.

Performing Rolling Updates for an Application

First, make a change to the application source code. Use the following command to find and replace `Hello, world!` with `Hello, cloud!`.

```
# sed, a stream editor
# Sample usage:
# sed -i 's/old-text/new-text/g' input_file
# Alternative usage for Mac users:
# sed -i '' 's/old-text/new-text/g' input_file
sed -i 's/Hello, world!/Hello, cloud!/g' main.go
```

Build a new version of your application tagged as `v2` and push the image to the registry.

```
docker build -t ${ACR_NAME}.azurecr.io/clouduser/hello-app:v2 .
docker push ${ACR_NAME}.azurecr.io/clouduser/hello-app:v2
```

Apply a rolling update to the existing deployment by updating the image to the newly built image:

```
kubectl set image deployment/hello-web hello-app=${ACR_NAME}.azurecr.io/clouduse
r/hello-app:v2
# Expected output:
# deployment.extensions/hello-web image updated
```

The old pods will incrementally replaced by the new pods. The rolling update may take some time to finish. During the rolling update, you may observe the transition process of the rolling update. For example, you may observe the old pods being terminated (with the `STATUS` as `Terminating`) and/or the new pods being created (with the `STATUS` as `Pending`).

```
kubectl get pods
# Sample output during the rolling update:
# NAME                          READY   STATUS        RESTARTS   AGE
# hello-web-556d989c58-lfvkw    0/1     Terminating   0          111s
# hello-web-556d989c58-wf54p    0/1     Terminating   0          111s
# hello-web-556d989c58-z5fmp    1/1     Terminating   0          3m26s
# hello-web-6b44c9b79c-mwljk    1/1     Running       0          6s
# hello-web-6b44c9b79c-pl2v8    1/1     Running       0          4s
# hello-web-6b44c9b79c-z7wxl    1/1     Running       0          2s
#
# Sample output after the rolling update completes:
# NAME                          READY   STATUS    RESTARTS   AGE
# hello-web-6b44c9b79c-mwljk    1/1     Running   0          2m1s
# hello-web-6b44c9b79c-pl2v8    1/1     Running   0          119s
# hello-web-6b44c9b79c-z7wxl    1/1     Running   0          117s
```

You can inspect the status of a rollout using the `kubectl rollout status` command.

```
kubectl rollout status deployment hello-web
# Sample output during the rolling update:
# Waiting for rollout to finish: 2 out of 3 new replicas have been updated...
#
# Sample output after the rolling update completes:
# deployment "hello-web" successfully rolled out
```

After the rolling update completes, visit `http://<EXTERNAL-IP>` to verify that the web server returns a plain text response as follows:

```
Hello, cloud!
Version: 1.0.0
Hostname: <auto-generated-pod-name-v2-[1|2|3]>
```

The welcome text will be `Hello, cloud!` and the hostname will be one of the newly created pods.

Clean Up Resources

After you finish the tutorial, please clean up the resources by deleting the resource group you used.

References: - Authenticate with Azure Container Registry from Azure Kubernetes Service (https://docs.microsoft.com/en-us/azure/aks/cluster-container-registry-integration)

Kubernetes on GCP

Google Kubernetes Engine

Google Kubernetes Engine (GKE) is a managed, production-ready environment for deploying containerized applications.

This tutorial shows you how to package a web application in a Docker container image, and run that container image on a Google Kubernetes Engine cluster.

Before you start

1. Stop any docker processes from Azure to prevent port clashes.
2. Create a GCP project and enable billing (https://cloud.google.com/billing/docs/how-to/modify-project#enable_billing_for_a_project).
3. Enable the APIs for the GCP project: Google Container Registry API (https://console.cloud.google.com/apis/library/containerregistry.googleapis.com) and Google Kubernetes Engine API (https://console.cloud.google.com/apis/library/container.googleapis.com).

You will finish the GKE tutorial in this section using Google Cloud Shell, which has the `gcloud`, `docker`, and `kubectl` command-line tools pre-installed.

Go to the Google Cloud Platform Console (https://console.cloud.google.com/) and open the Cloud Shell with the button at the top of console. **You will use Google Cloud Shell to complete this section, and there is no need to provision any VM.**

Figure: Google Activate Cloud Shell.

Tutorial Summary

In this tutorial, you will package and deploy your application on GKE by finishing the following steps:

1. Package your application into a Docker image
2. Upload the image to Google Container Registry (GCR)
3. Create a container cluster
4. Deploy your app to the cluster

Besides, you will perform the most common operations for containerized deployment:

1. Scale up an application with a load balancer
2. Perform a rolling update for an application without downtime

Build a Container Image

In this tutorial, you will deploy a sample web application called `hello-app` that responds to all requests with the message "Hello, World!" on port 80.

To download the `hello-app` source code including the Dockerfile, run the following commands:

```
git clone https://github.com/GoogleCloudPlatform/kubernetes-engine-samples
cd kubernetes-engine-samples/hello-app
```

Set the PROJECT_ID environment variable to your GCP project ID. Build and tag the container image for uploading. Here we will use `us.gcr.io` , which hosts the image in the United States.

```
PROJECT_ID=<your_gcp_project_id>
docker build -t us.gcr.io/${PROJECT_ID}/hello-app:v1 .
```

Run the Container Locally

To test your container image using your local Docker engine, run the following command:

```
docker run -d -p 8080:8080 us.gcr.io/${PROJECT_ID}/hello-app:v1
```

Verify that the container works and responds to requests with "Hello, World!":

```
curl http://localhost:8080
# Expected output:
# Hello, world!
# Version: 1.0.0
# Hostname: <container_id>
```

Stop the local running container. Use the following commands to list and stop the containers:

```
docker ps

# stop a container by ID
docker stop CONTAINER_ID
```

Upload the container image to GCR

Authenticate to the Google Container Registry.

```
gcloud auth configure-docker
```

Push the image to the Google Container Registry.

```
docker push us.gcr.io/${PROJECT_ID}/hello-app:v1
```

Create a GKE Cluster

Now that you have built the container image and pushed the image to the GCR, you need to create a GKE cluster to run the container image. A GKE cluster consists of a pool of Compute Engine VM instances running Kubernetes.

Create a Cluster with the CLI

In the Cloud Shell, run the following command to create a k8s cluster:

```
# Set your project ID and Compute Engine zone options for the gcloud tool:
gcloud config set project ${PROJECT_ID}
gcloud config set compute/zone us-east1-d

# The name may contain only lowercase alphanumerics and '-',
# and it must start with a letter and end with an alphanumeric,
# and it must be no longer than 40 characters.
CLUSTER_NAME=gke-demo-cluster

gcloud container clusters create ${CLUSTER_NAME} --num-nodes=3
# Expected output:
# ...
# Creating cluster gke-demo-cluster in us-east1-d... Cluster is being health-che
cked (master is healthy)...done.
# Created [https://container.googleapis.com/v1/projects/<project_id>/zones/us-ea
st1-d/clusters/gke-demo-cluster].
# To inspect the contents of your cluster, go to: https://console.cloud.google.c
om/kubernetes/workload_/gcloud/us-east1-d/gke-demo-cluster?project=<project_id>
# kubeconfig entry generated for gke-demo-cluster.
# NAME              LOCATION    MASTER_VERSION  MASTER_IP      MACHINE_TYPE   NO
DE_VERSION  NUM_NODES  STATUS
# gke-demo-cluster  us-east1-d  1.15.12-gke.17    .............  n1-standard-1
1.15.12-gke.17  3         RUNNING
```

(Alternatively) Create a Cluster from the Web Console

You can also create a cluster from the GCP console (https://console.cloud.google.com/kubernetes/add).

Deploy the Application to the GKE Cluster

To deploy and manage applications on a GKE cluster, you must communicate with the Kubernetes cluster management system. You typically do this by using the `kubectl` command-line tool. You can use `kubectl` to deploy applications, inspect and manage cluster resources, and view logs.

Warning

Authenticate to a GKE Cluster

If you created the cluster from the web console or on a different machine from the one you are currently working on, you need to run the following command to grant the `kubectl` tool the access to your cluster.

```
$ gcloud container clusters get-credentials ${CLUSTER_NAME}
```

If you created the cluster with your current machine with the `gcloud container clusters create` command, the authentication to the cluster is already set up.

Run the command below to create a Deployment named `hello-web` on the GKE cluster.

```
kubectl create deployment hello-web --image=us.gcr.io/${PROJECT_ID}/hello-app:v1
# Expected output:
# deployment.apps/hello-web created
```

After the command finishes, check that the deployment is working correctly:

```
kubectl get deployments
# Expected output:
# NAME        READY   UP-TO-DATE   AVAILABLE   AGE
# hello-web   1/1     1            1           ...
```

Kubernetes represents applications as Pods. In this tutorial, each Pod contains only your `hello-app` container. To see the Pod created by the Deployment, run the following command:

```
kubectl get pods
#
# NAME                          READY   STATUS    RESTARTS   AGE
# <auto-generated-pod-name-1>   1/1     Running   0          66s
```

Expose Applications to the Internet

By default, the containers you run on GKE are not accessible from the Internet. You must explicitly expose your application to traffic from the Internet, run the following command:

```
# --port: specifies the port number configured on the Load Balancer
# --target-port: specifies the port number that the hello-app container is liste
ning on.
kubectl expose deployment hello-web --type=LoadBalancer --port 80 --target-port
8080
# Expected output:
# service/hello-web exposed
```

The command above creates a Service resource, which provides networking and IP support to your application's Pods. GKE creates an external IP and a Load Balancer for your application. Note that a Load Balancer is subject to billing. You can view the load balancers in the web console (https://console.cloud.google.com/net-services/loadbalancing/loadBalancers/list).

GKE assigns the external IP address to the Service resource (not the Deployment). To retrieve the external IP that GKE provisioned, inspect the `EXTERNAL-IP` value of the Service with the `kubectl get service` command. Retry later if the `EXTERNAL-IP` value is `<pending>`.

```
kubectl get service
```

Alternatively, instead of retrying the command manually to inspect the output, you can use the Linux tool `watch`. It runs the specified command repeatedly and displays the results on standard output so you can watch it change over time. You can press "Control+C" to stop it.

```
watch kubectl get service
```

Once you have determined the external IP address for your application. Visit `http://<EXTERNAL-IP>` to verify that the web server returns a plain text response as follows:

```
Hello, world!
Version: 1.0.0
Hostname: <auto-generated-pod-name-1>
```

Scalability and Rolling Updates

Nowadays, applications have to be designed upfront for **scalability** and **change without downtime**. Now you will practice how to scale your application and apply a rolling update to the existing deployment.

Scale up the Application

Run the following command to add two additional replicas to your Deployment for a total of three:

```
kubectl scale deployment hello-web --replicas=3
# Expected output:
# deployment.extensions/hello-web scaled
```

View the new replicas running on your cluster with the following commands:

```
kubectl get deployments
# Expected output:
# NAME        READY   UP-TO-DATE   AVAILABLE   AGE
# hello-web   3/3     3            3           ...

kubectl get pods
# Expected output:
# NAME                          READY   STATUS    RESTARTS   AGE
# <auto-generated-pod-name-3>   1/1     Running   0          ...
# <auto-generated-pod-name-2>   1/1     Running   0          ...
# <auto-generated-pod-name-1>   1/1     Running   0          ...
```

Now, simulate a traffic load by sending 100 web requests to the external IP address for your application.

```
EXTERNAL_IP=<external_ip>
for i in {1..100}; do curl http://${EXTERNAL_IP}/; done
```

You will observe that the `Hostname` part of the text response will differ between these web requests. For each web request, the load balancer you provisioned routes the traffic to one of the replicas.

Performing Rolling Updates for an Application

Rolling updates *incrementally* replace the Pods with new ones. Rolling updates are designed to update your applications *without downtime*.

First, make a change to the application source code. Use the following command to find and replace `Hello, world!` with `Hello, cloud!`.

```
# sed, a stream editor
# Sample usage:
# sed -i 's/old-text/new-text/g' input_file
# Alternative usage for Mac users:
# sed -i '' 's/old-text/new-text/g' input_file
sed -i 's/Hello, world!/Hello, cloud!/g' main.go
```

Build a new version of your application tagged as `v2` and push the image to the registry.

```
docker build -t us.gcr.io/${PROJECT_ID}/hello-app:v2 .
docker push us.gcr.io/${PROJECT_ID}/hello-app:v2
```

Apply a rolling update to the existing deployment by updating the image to the newly built image:

```
kubectl set image deployment/hello-web hello-app=us.gcr.io/${PROJECT_ID}/hello-a
pp:v2
# Expected output:
# deployment.extensions/hello-web image updated
```

The old pods will incrementally replaced by the new pods. The rolling update may take some time to finish. During the rolling update, you may observe the transition process of the rolling update. For example, you may observe the old pods being terminated (with the `STATUS` as `Terminating`) and/or the new pods being created (with the `STATUS` as `Pending`). You can use `watch kubectl get pods` to better observe the changes.

```
kubectl get pods
# Sample output during the rolling update:
# NAME                          READY   STATUS        RESTARTS   AGE
# hello-web-556d989c58-lfvkw    0/1     Terminating   0          111s
# hello-web-556d989c58-wf54p    0/1     Terminating   0          111s
# hello-web-556d989c58-z5fmp    1/1     Terminating   0          3m26s
# hello-web-6b44c9b79c-mwljk    1/1     Running       0          6s
# hello-web-6b44c9b79c-pl2v8    1/1     Running       0          4s
# hello-web-6b44c9b79c-z7wxl    1/1     Running       0          2s
#
# Sample output after the rolling update completes:
# NAME                          READY   STATUS    RESTARTS   AGE
# hello-web-6b44c9b79c-mwljk    1/1     Running   0          2m1s
# hello-web-6b44c9b79c-pl2v8    1/1     Running   0          119s
# hello-web-6b44c9b79c-z7wxl    1/1     Running   0          117s
```

You can inspect the status of a rollout using the `kubectl rollout status` command.

```
kubectl rollout status deployment hello-web
# Sample output during the rolling update:
# Waiting for rollout to finish: 2 out of 3 new replicas have been updated...
#
# Sample output after the rolling update completes:
# deployment "hello-web" successfully rolled out
```

After the rolling update completes, visit `http://<EXTERNAL-IP>` to verify that the web server returns a plain text response as follows:

```
Hello, cloud!
Version: 1.0.0
Hostname: <auto-generated-pod-name-v2-[1|2|3]>
```

The welcome text will be `Hello, cloud!` and the hostname will be one of the newly created pods.

After you finish the tutorial in this section, please delete the Cloud Load Balancer created for your Service:

```
kubectl delete service hello-web
# Expected output:
# service "hello-web" deleted
```

For more information about performing, monitoring and managing rolling updates on GKE clusters, please refer to this GCP doc (https://cloud.google.com/kubernetes-engine/docs/how-to/updating-apps).

Further Reading

You can refer to the official documentation pages for more commands and usages:

1. https://kubernetes.io/docs/user-guide/kubectl-overview/ (https://kubernetes.io/docs/user-guide/kubectl-overview/)
2. https://kubernetes.io/docs/user-guide/kubectl/ (https://kubernetes.io/docs/user-guide/kubectl/)

References: Deploying a containerized web application - Google Kubernetes Engine (https://cloud.google.com/kubernetes-engine/docs/tutorials/hello-app)

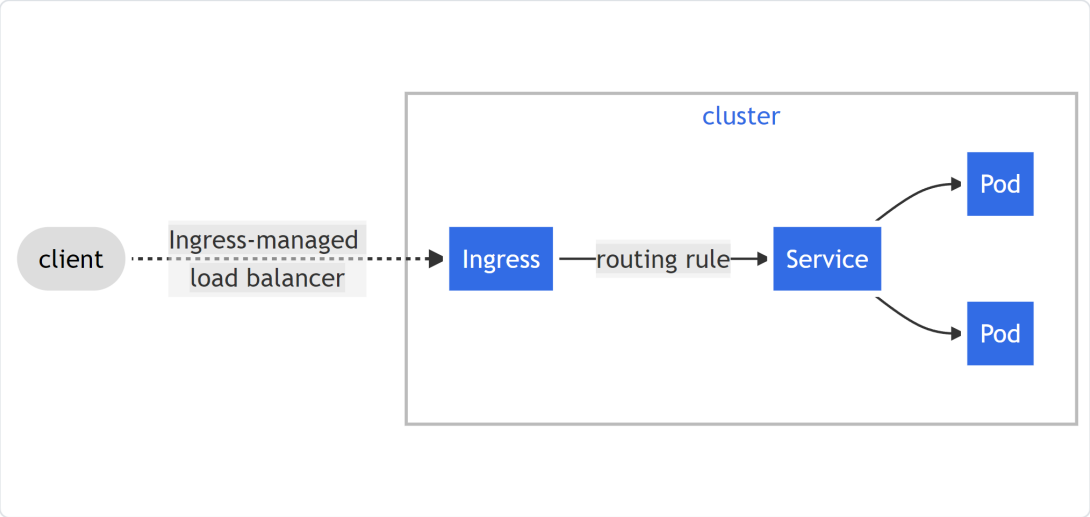
> Warning
>
> You will need the GKE cluster created in this section to complete the hands-on practices in the next sections, please DO NOT terminate the resources for now.

Introduction to Ingress

What is Ingress

Ingress (https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.22/#ingress-v1-networking-k8s-io) exposes HTTP and HTTPS routes from outside the cluster to services (https://kubernetes.io/docs/concepts/services-networking/service/) within the cluster. Traffic routing is controlled by rules defined on the Ingress resource.

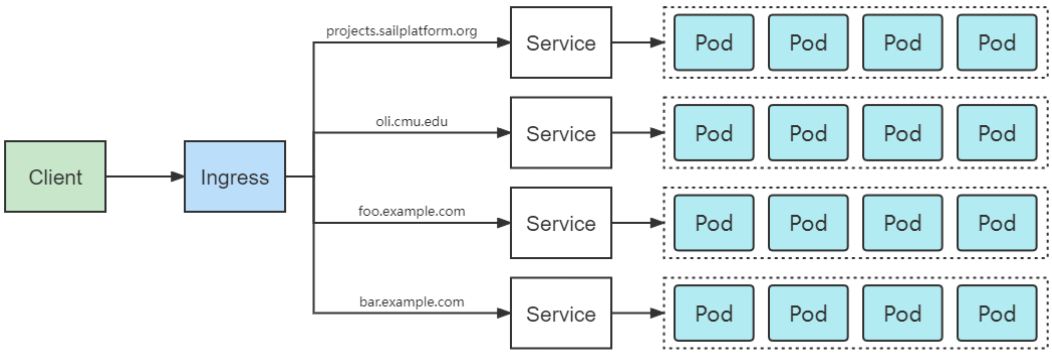
Here is a simple example where an Ingress sends all its traffic to one Service:



Ingress controller (https://kubernetes.io/docs/concepts/services-networking/ingress-controllers) is responsible for fulfilling the Ingress, usually with a load balancer, though it may also configure your edge router or additional frontends to help handle the traffic.

Why Ingress

One important reason is that each LoadBalancer (https://kubernetes.io/docs/concepts/services-networking/service/#loadbalancer) service requires its own load balancer with its own public IP address, whereas an Ingress only requires one, even when providing access to dozens of services. When a client sends an HTTP request to the Ingress, the host and path in the request determine which service the request is forwarded to. Here is a simple example where an Ingress sends all its traffic to multiple Service:



To make Ingress resources work, an **Ingress controller** needs to be running in the cluster. Different Kubernetes environments use different implementations of the controller. For example, Google Kubernetes Engine(GKE) uses Google Cloud Platform's own HTTP load-balancing features to provide the Ingress functionality.

Installing NGINX Ingress Controller

Here, we choose to use the NGINX Ingress controller. In addition, there are many kinds of Ingress controllers to choose from.

1. Adding the Helm Repository:

   $ helm repo add nginx-stable https://helm.nginx.com/stable $ helm repo update

2. Installing the Chart

   $ helm install example-ingress-controller nginx-stable/nginx-ingress

Creating an Ingress resource

Once you've confirmed there's an Ingress controller running in your cluster, you can now create an Ingress resource. The following listing shows what the YAML manifest for the Ingress looks like.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: example-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: "projects.sailplatform.org"
    http:
    paths:
    - path: /
      pathType: Prefix
      backend:
        service:
          name: test
          port:
            number: 80
```

This defines an Ingress with a single rule, which makes sure all HTTP requests received by the Ingress controller, in which the host `projects.sailplatform.org` is requested, will be sent to the `test` service on port `80`.

As with all other Kubernetes resources, an Ingress needs `apiVersion`, `kind`, and `metadata` fields. The name of an Ingress object must be a valid DNS subdomain name (https://kubernetes.io/docs/concepts/overview/working-with-objects/names#dns-subdomain-names).

Ingress frequently uses `annotations` to configure some options depending on the Ingress controller, an example of which is the rewrite-target annotation (https://github.com/kubernetes/ingress-nginx/blob/master/docs/examples/rewrite/README.md). Different Ingress controller (https://kubernetes.io/docs/concepts/services-networking/ingress-controllers) support different annotations. Review the documentation for your choice of Ingress controller to learn which annotations are supported.

The Ingress spec (https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status) has all the information needed to configure a load balancer or proxy server. Most importantly, it contains a list of rules matched against all

incoming requests. Ingress resource only supports rules for directing HTTP(S) traffic.

Create an Ingress resource by running the following command:

```
$ kubectl create -f <YOUR_INGRESS_YAML_MANIFEST_PATH>
```

After creating the Ingress above, you can view it with the following command:

```
$ kubectl describe ingress example-ingress
```

Ingress rules

Each HTTP rule contains the following information:

- An optional `host`. If no host is specified, the rule applies to all inbound HTTP traffic through the IP address specified. If a host is provided (In our example, `projects.sailplatform.org`), the rules apply to that host.
- A list of `paths` (for example, `/testpath`), each of which has an associated backend defined with a `service.name` and a `service.port.name` or `service.port.number`. Both the host and path must match the content of an incoming request before the load balancer directs traffic to the referenced Service.
- A `backend` is a combination of Service and port names as described in the Service doc (https://kubernetes.io/docs/concepts/services-networking/service/). HTTP (and HTTPS) requests to the Ingress that matches the host and path of the rule are sent to the listed backend.

A `defaultBackend` is often configured in an Ingress controller to service any requests that do not match a path in the spec.

Further Reading

You can refer to the official documentation pages for more information about Ingress and Ingress controller:

1. https://kubernetes.io/docs/concepts/services-networking/ingress/ (https://kubernetes.io/docs/concepts/services-networking/ingress/)
2. https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/ (https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/)

Introduction to Helm

Helm - The Kubernetes Package Manager

> Helm (https://helm.sh/) helps you manage Kubernetes applications — Helm Charts (https://helm.sh/docs/topics/charts/) helps you define, install, and upgrade even the most complex Kubernetes application.

Up to this point, we have been deploying standalone applications that do not have many dependencies or configuration. However, as the complexity of your software projects increases, it is advantageous to use tools that help manage this complexity. Helm is one such tool that can be used to maintain the relationship of Kubernetes objects within a cluster. Helm Charts are the primary means of organizing these relationships. Helm Charts are composed of template files that can be parameterized to deploy uniquely configured versions of an application. The following sections will introduce Helm, Helm Charts v.s. Helm releases, and the steps for releasing applications.

Application Management

In the previous section, you used `kubectl` to manually create Kubernetes objects, specifically deployments and services. For simple architectures, such as a single web server, this may be an acceptable method of deployment. However, as more Kubernetes objects are added to the architecture, the overhead of manually managing all the objects will grow.

You may be familiar with the `apt-get` or `yum` tools used to install packages in Linux based systems. Analogously, `helm` is like a package tool that allows you to install applications into a Kubernetes cluster. Refer to the Helm documentation (https://helm.sh/), and you will find that Helm provides benefits similar to package managers in Linux:

- **Manage Complexity**: Helm Charts allow developers to describe complex architectures and easily deploy, update and remove these applications to Kubernetes clusters.
- **Easy Updates**: Applications and resources in Kubernetes can be upgraded seamlessly using the Helm client.
- **Simple Sharing**: Application Charts are versioned and can be distributed for deployment by other users. There are many stable packages that you can choose from, such as MySQL, MongoDB, WordPress, etc. Once you become more familiar with Helm, you are encouraged to explore the Helm Hub (https://hub.helm.sh/).
- **Rollbacks** - In case of a failed application deployment, you can use helm to revert to a known worked state.

Helm Charts

In the terminology of Helm, a package is referred to as a Chart (https://helm.sh/docs/glossary/#chart). Charts serve a similar purpose to `.deb` or `.rpm` packages in Linux systems. Refer to the Helm Glossary (https://helm.sh/docs/glossary/) if you come across an unfamiliar term regarding Helm. Directly from the Helm Glossary, the definition of a Chart is as follows:

Chart Definition

1. A Helm package that contains information sufficient for installing a set of Kubernetes resources into a Kubernetes cluster.
2. Charts contain a `Chart.yaml` file as well as templates, default values (`values.yaml`), and dependencies.
3. Charts are developed in a well-defined directory structure and then packaged into an archive format called a chart archive.

Chart Structure

The Chart File Structure (https://helm.sh/docs/topics/charts/#the-chart-file-structure) documentation provides a brief description of each of the files that are included in a chart directory.

A sample Chart file structure is as follows:

```
wordpress/
  Chart.yaml          # A YAML file containing information about the chart
  values.yaml         # The default configuration values for this chart
  charts/             # A directory containing any charts upon which this chart
depends.
  templates/          # A directory of templates that, when combined with value
s,
                      # will generate valid Kubernetes manifest files.
  ... (Optional components ommited)
```

For this discussion, we will ignore the optional components and focus on the core components:

1. **Chart.yaml** - A YAML file that contains chart information (e.g., name, version, description, etc.). The Chart.yaml File (https://helm.sh/docs/topics/charts/#the-chart-yaml-file) documentation provides descriptions of each field in this file.
2. **values.yaml** - The default configuration of this chart. The values listed in this file will be substituted in the files under the `templates/` directory. Template files may contain Go templates (https://golang.org/pkg/text/template/). For example, if `imageRegistry` is defined in `values.yaml` , it will replace `{{.Values.imageRegistry}}` in template files.
3. **templates/** - A directory of template files that will be combined with the values defined in `values.yaml` . The files under this directory will be used to define all of the Kubernetes objects required to deploy the application.

Helm Release

A Helm release is an instance of a chart running in a Kubernetes cluster. A Chart is a Helm package which contains all of the resource definitions necessary to run an application, tool, or service inside of a Kubernetes cluster. The relation of a Helm chart and a Helm release is

akin to the relation of Docker image and Docker container, i.e., a Helm chart is a blueprint and a Helm release is an instance of the blueprint.

Helm Architecture

The Kubernetes Helm Architecture (https://helm.sh/docs/architecture/) documentation describes the components required to deploy applications using Helm.

1. **Helm client**: A command-line tool that enables users to develop charts, manage repositories, and install, describe or upgrade a release.
2. **The Helm Library**: Provides the logic for executing all Helm operations. It interfaces with the Kubernetes API server and provides the following capability.

Installation

At this time you should have a running Kubernetes cluster that you can access via `kubectl`. If you do not have a running cluster, return to the previous section and launch a cluster.

Installing Helm

Install the Helm 3 client (`helm`) to the same "client" machine where `kubectl` is installed with the commands below. The commands are for Linux environments (e.g., Ubuntu 18.04 LTS, the OS of Google Cloud Shell, etc.)

```
curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/master/scr
ipts/get-helm-3
chmod 700 get_helm.sh
./get_helm.sh
```

Note: If you are using the Google Cloud Shell, a different version of Helm may be pre-installed (e.g., Helm 2). You **MUST** re-install Helm with the commands above to ensure that you have Helm 3 installed.

To install Helm 3 on MacOS and Windows, refer to Installing Helm (https://helm.sh/docs/using_helm/#installing-helm).

Verify the versioning

Verify the Helm installation and its version. Note that the version must be `v3.x.x`. A major release consists of major new features and/or large architectural changes. Some existing features can be deprecated or no longer supported. The internal implementation of Helm 3 has changed considerably from Helm 2. The following hands-on practice is all designed for Helm 3.

```
helm version
# Example Output:
# version.BuildInfo{Version:"v3.8.0", GitCommit:"b29d20baf09943e134c2fa5e1e1cab3
bf93315fa", GitTreeState:"clean", GoVersion:"go1.17.5"}
```

Install a Helm Release Using a Prepared Chart

In this section, you will practice how to install a Helm release using a prepared Helm chart. We will use MySQL as an example, and you will install MySQL to a Kubernetes cluster using the Helm chart of MySQL (https://github.com/helm/charts/tree/master/stable/mysql).

Before you install any chart, you need to run `helm repo update` to get the latest Chart information from chart repositories, similar to the `apt-get update` command in Linux.

```
helm repo add stable https://charts.helm.sh/stable

helm repo update
```

Next, install the `stable/mysql` chart in the cluster.

To install a chart as a Helm release, the syntax of the command is

```
helm install [RELEASE_NAME] [CHART] [flags]
```

Below is the concrete command you will use to install the `stable/mysql` chart as a Helm release named `mysql-instance`. You may get a warning stating that the mysql helm chart is deprecated, but it shoudn't cause any problems with deploying the helm chart.

```
# Set environment variables for the MySQL properties
# Please avoid using digits only in these variables or the value will be recogni
zed as integer instead of string which may cause error.
export MYSQL_ROOT_PWD="REPLACE_THIS_VALUE"
export MYSQL_PWD="REPLACE_THIS_VALUE"
export MYSQL_USERNAME="REPLACE_THIS_VALUE"

helm install \
    mysql-instance \
    --set mysqlRootPassword=$MYSQL_ROOT_PWD,mysqlUser=$MYSQL_USERNAME,mysqlPassw
ord=$MYSQL_PWD,mysqlDatabase=test \
    stable/mysql
# Sample output:
# NAME:   mysql-instance
# LAST DEPLOYED: ...
# NAMESPACE: default
# STATUS: DEPLOYED
#
# RESOURCES:
# ...
#
# NOTES:
# MySQL can be accessed via port 3306 on the following DNS name from within your
cluster:
# mysql-instance.default.svc.cluster.local
...
```

If MySQL is successfully released, the `NOTES` section will contain the information about connecting to the database.

You can list the release(s) in the cluster using `helm list`.

```
helm list
# Expected output:
# NAME            NAMESPACE       REVISION        UPDATED   STATUS          CHAR
T           APP VERSION
# mysql-instance  default         1               .......   deployed        mysq
l-1.6.2     5.7.28
```

You can retrieve the status of a release using `helm status`.

```
helm status mysql-instance
```

Verify the connection to the MySQL database running in the cluster. You will do so by launching an Ubuntu pod and connecting to the database via the `mysql` client.

```
# Launch an Ubuntu pod in the cluster, and connect to the pod (with -i --tty)
kubectl run -i --tty ubuntu --image=ubuntu:16.04 --restart=Never -- bash -il

# In the pod, install the mysql client in the container
apt-get update && apt-get install mysql-client -y

# Provide your root password to connect to the new mysql database
mysql -h mysql-instance -p
```

Type `exit` to exit the MySQL prompt, and type `exit` to log out the connection to the Ubuntu pod.

Now, you have successfully installed a Helm release from a provided Helm chart and verified that the release is functioning. Congratulations!

You can now delete the Helm release. To delete a release, use `helm uninstall` (or any of its equivalent commands `helm delete` and `helm del`):

```
helm uninstall mysql-instance
# Expected output:
# release "mysql-instance" uninstalled
```

References

- https://helm.sh/ (https://helm.sh/)
- https://helm.sh/docs/architecture/ (https://helm.sh/docs/architecture/)
- https://github.com/helm/helm (https://github.com/helm/helm)
- https://github.com/helm/charts (https://github.com/helm/charts)
- https://github.com/helm/charts/tree/master/stable/mysql (https://github.com/helm/charts/tree/master/stable/mysql)

> **Warning**
>
> You will need the GKE cluster created in this section to complete the hands-on practices in the next sections, please DO NOT terminate the resources for now.

How to Develop Helm Charts

In the previous section, you have practiced the basic Helm commands and experimented with a provided Helm chart to create a Helm release. You are now prepared to develop your own Helm chart to define your own Helm releases.

Kubernetes YAML files are the starting point of developing Helm Charts. In the following sections, you will first develop a worked application with Kubernetes YAML files and deploy the application with `kubectl`, and then transform the Kubernetes YAML files into a Helm chart.

You should continue using **the Google Cloud Shell** as your workspace to work with **the same GKE cluster** as the one you used in the previous two sections.

Develop Kubernetes YAML Files and Use Kubernetes CLI to Deploy Web Services

How to Develop Helm Charts

service.yaml: In Kubernetes, a Service can be defined in an abstract way to expose an application running on a set of Pods as a network service. service.yaml file defines such specification.

```
apiVersion: v1  # defines the particular functionalities of Kubernetes to be use
d
kind: Service       # ServiceTypes allow you to specify what kind of Service you
want
metadata:              # Attributes related to  your config
  name: my-service  # create a named service where name is valid DNS label name
spec:
  selector:     # set of pods that a service targets is defined with a label sel
ector
    app: MyApp
  ports:
    - protocol: TCP   # protocol selected for the ports below
      port: 80            # incoming request port
      targetPort: 9376 # port that the pod listens to
```

deployment.yaml: In Kubernetes, Deployment provides declarative updates for Pods and ReplicaSets. You describe a desired state in a Deployment, and the Deployment Controller changes the actual state to the desired state at a controlled rate. deployment.yaml defines such desired state

```
apiVersion: apps/v1   # defines the particular functionalities of Kubernetes to
be used
kind: Deployment   # ServiceTypes allow you to specify what kind of Service you
want
metadata:  # Attributes related to  your deployment
  name: nginx-deployment        # creates a deployment with the given name
  labels:
    app: nginx
spec:
  replicas: 3       # specify number of replicated Pods
  selector:     # defines how the Deployment finds which Pods to manage
    matchLabels:
      app: nginx
  template:           # template of the configuration to be deployed on pods
    metadata:
      labels:
        app: nginx       # label for the pods
    spec:             # specifications defined of the pods
      containers:
      - name: nginx #  name of app that pods run
        image: nginx:1.14.2     # docker image: tag to be run on pod
        ports:
        - containerPort: 80 # listening port of the container
```

configMap.yaml: (Optional) A ConfigMap is an API object for setting configuration data separately from application code. Multiple Pods can reference the same ConfigMap.

This is an example of a Pod that mounts a ConfigMap in a volume:

```
apiVersion: v1  # defines the particular functionalities of Kubernetes to be use
d
kind: Pod   # ServiceTypes allow you to specify what kind of Service you want
metadata:
  name: mypod
spec:             # specifications defined of the pods
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
      readOnly: true
  volumes:
  - name: foo
    configMap:      # field set to reference your ConfigMap object
      name: myconfigmap
```

Note: configMap name needs to referenced in deployment.yaml

In the example below, you will first develop and deploy a application on a Kubernetes cluster. We will use the Hello Application example of the Google Kubernetes Engine tutorials (https://github.com/GoogleCloudPlatform/kubernetes-engine-samples/tree/master/hello-app) the starting point.

Create a new folder and change the current working directory to it.

```
mkdir -p ~/hello-app
cd ~/hello-app
```

Download the example **deployment.yaml**. A **deployment** Kubernetes Object is used as the template of application instances.

```
wget https://raw.githubusercontent.com/GoogleCloudPlatform/kubernetes-engine-samples/master/hello-app/manifests/helloweb-deployment.yaml -O deployment.yaml
```

Download the example **service.yaml**. Add the following snippet into service.yaml. A service here works as a load balancer, to expose a public IP address and receive HTTP request from outside of the server and redirect to the application.

```
wget https://clouddeveloper.blob.core.windows.net/primers/container/service.yaml
-O service.yaml
```

Use `kubectl` to deploy the service to the cluster with the YAML definitions:

```
kubectl apply -f .
```

Use the following commands to check the deployment results, including viewing the deployed Kubernetes objects and pod logs.

```
# List all services in the namespace
kubectl get services

# List all pods in the namespace
kubectl get pods

# Retrieve the logs for a specific pod
kubectl logs POD_NAME
```

You can send HTTP request to the web service with the external IP address of the load balancer. You can find the external IP address when doing `kubectl get services`. Use curl or browser to send the request.

```
curl http://LOAD_BALANCER_EXTERNAL_IP_ADDRESS
# Expected response"
# Hello, world!
# Version: 1.0.0
# Hostname: helloweb-xxxxx
```

Delete the deployment and the load balancer service from the GKE cluster.

```
kubectl delete -f .
```

Transform a Kubernetes YAML definition into a Helm Chart

Now that you have a worked Kubernetes YAML definition, you will learn how to build a Helm Chart from it.

Before you move on, please make sure that you have deleted the `helloweb` service you installed if you have not done so.

Download and extract the Helm chart template.

```
cd ~
wget https://clouddeveloper.blob.core.windows.net/primers/container/helloweb.tar
tar -xvf helloweb.tar
cd ~/helloweb
```

All template files are stored in a chart's `templates/` folder. In the simplest case, you can reuse the Kubernetes YAML files as they are as the template files. Copy the **deployment.yaml** and **service.yaml** into the template folder.

```
cp ~/hello-app/*.yaml ~/helloweb/templates

ls ~/helloweb/templates
# Expected output:
# deployment.yaml  _helpers.tpl  service.yaml
```

Now you developed a Helm chart. You can use `helm` instead of `kubectl` to install, manage, and delete the web application on the Kubernetes cluster. Go to the `helloweb` folder and create the helm chart.

```
cd ~/helloweb
helm install helloweb .
# Expected output:
# NAME: helloweb
# LAST DEPLOYED: ...
# NAMESPACE: default
# STATUS: deployed
# REVISION: 1
# TEST SUITE: None
```

Use the following commands to check the deployment results, including viewing the deployed Kubernetes objects and pod logs.

```
# List all services in the namespace
kubectl get services

# List all pods in the namespace
kubectl get pods

# Retrieve the logs for a specific pod
kubectl logs POD_NAME
```

Use curl or browser to send the request, and verify it is working.

```
curl http://LOAD_BALANCER_EXTERNAL_IP_ADDRESS
# Expected response
# Hello, world!
# Version: 1.0.0
# Hostname: helloweb-xxxxx
```

Delete the Helm release with the following command. Remember always use Helm instead of `kubectl` to delete the Kubernetes Objects managed by Helm.

```
helm uninstall helloweb
```

Introduction to Volumes

Introduction to Volumes

Docker has a concept of volumes (https://docs.docker.com/storage/), though it is somewhat looser and less managed. A Docker volume is a directory on disk or in another container. Docker provides volume drivers, but the functionality is somewhat limited.

Kubernetes supports many types of volumes. A Pod can use any number of volume types simultaneously. We can use the analogy of a pod being a small computer which runs a single application. This application can consist of one or more containers that run the application processes. These processes share computing resources such as CPU, memory, network interfaces and so on. However, each container has its own isolated filesystem provided by the container image.

When a container starts, the files in its filesystem are those that were added to its container image during build time. The process running in the container can then modify those files or create new files. However, the changes to the files will be lost if the container crashes. Similar to a pod, when a pod ceases to exist, Kubernetes destroys ephemeral volumes; however, Kubernetes does not destroy persistent volumes. For any kind of volume in a given pod, data is preserved across container restarts.

How volumes fit into pods

A volume is defined at the pod level and then mounted at the desired location in the container. Thus, the lifecycle of a volume is tied to the lifecycle of the pod and is independent of the lifecycle of the containers in the pod. In addition, a pod can have multiple volumes and each container can mount arbitrary volumes in different locations.

Since the volume is tied to the lifecycle of the pod and only exists for as long as the pod exists, how can we persist data across pod instances? The answer is choosing a different volume type. A pod volume can map to persistent storage outside the pod. Therefore, the file directory representing the volume could be an existing network-attached storage volume(eg. AWS EBS) whose lifecycle isn't tied to any pod. The data store in the volume can be used by the application with a new pod running on a different worker node.

Types of Volumes

There is a wide range of volume types is available. Here's a non-exhaustive list of the supported volume types:

- `emptyDir` : First created when a Pod is assigned to a node, and exists as long as that Pod is running on that node. All containers in the Pod can read and write the same files in the `emptyDir` volume. When a Pod is removed from a node for any reason, the data in the `emptyDir` is deleted permanently.

- `nfs` : Allows an existing NFS (Network File System) share to be mounted into a Pod. Unlike `emptyDir` , which is erased when a Pod is removed, the contents of an `nfs` volume are preserved and the volume is merely unmounted.
- `gcePersistentDisk` (Google Compute Engine Persistent Disk), `awsElasticBlockStore` (Amazon Web Services Elastic Block Store), `azureFile` (Microsoft Azure File Service), `azureDisk` (Microsoft Azure Data Disk): Used for mounting cloud provider-specific storage.
- `configMap` , `secret` , `downwardAPI` , `projected` : Special types of volumes used to expose information about the pod and other Kubernetes objects through files. They are typically used to configure the application running in the pod.
- `persistentVolumeClaim` : Used to mount a PersistentVolume (https://kubernetes.io/docs/concepts/storage/persistent-volumes/) into a Pod. A portable way to integrate external storage into pods. Instead of pointing directly to an external storage volume, this volume type points to a `PersistentVolumeClaim` object that points to a `PersistentVolume` object that finally references the actual storage.
- `csi` : A pluggable way of adding storage via the Container Storage Interface. This volume type allows anyone to implement their own storage driver that is then referenced in the `csi` volume definition. During pod setup, the CSI driver is called to attach the volume to the pod.

Using external storage in pods

Here we choose to use Google Compute Engine Persistent Disk as a Volume. Before we create Persistent Disk, we may confirm that we have a Kubernetes cluster. Then we should create the Persistent Disk in the same zone as Kubernetes cluster.

```
$ gcloud container clusters list
NAME: gke-volume-demo-cluster
LOCATION: us-east1-d
MASTER_VERSION: 1.21.5-gke.1302
MASTER_IP: 104.196.8.235
MACHINE_TYPE: e2-medium
NODE_VERSION: 1.21.5-gke.1302
NUM_NODES: 3
STATUS: RUNNING
$ gcloud compute disks create --size=10GiB --zone=us-east1-d db-data
NAME: db-data
ZONE: us-east1-d
SIZE_GB: 10
TYPE: pd-standard
STATUS: READY
```

The output indicates that we creates a GCE Persistent Disk named `db-data` with 10 GB space.

The MySQL official image is available at Docker hub (https://hub.docker.com/_/mysql). Here is the pod manifest `mysql-db.yaml` .

```
# mysql-db.yaml
apiVersion: v1
kind: Pod
metadata:
  name: mysql-db
spec:
  volumes:
  - name: db-data
    gcePersistentDisk:
      pdName: db-data
      fsType: ext4
  containers:
  - name: mysql
    image: mysql
    env:
    - name: MYSQL_ROOT_PASSWORD
      value: "123456"
    volumeMounts:
    - name: db-data
      mountPath: /var/lib/mysql
```

After creating a pod with a gcePersistentDisk volume, we are going to verify that the GCE Persistent Disk persistsdata.

First, we should deploy the pod in the cluster and insert a record into the database.

```
# Create the database pod
$ kubectl apply -f mysql-db.yaml
pod "mysql-db" deleted
# Check the status of the pod
$ kubectl get pods
NAME       READY   STATUS    RESTARTS   AGE
mysql-db   1/1     Running   0          26s
# open a mysql shell in the running mysql container
$ kubectl exec -it mysql-db -c mysql -- mysql -uroot -p123456

# Check original database status
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
4 rows in set (0.00 sec)

# Create a test database
mysql> CREATE DATABASE demo_db;
Query OK, 1 row affected (0.06 sec)

mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| demo_db            |
| information_schema |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
5 rows in set (0.00 sec)

mysql> USE demo_db;
Database changed

mysql> SHOW TABLES;
Empty set (0.00 sec)

# Create a test table in the new database
mysql> CREATE TABLE test_persistent(id INT, name VARCHAR(20));
Query OK, 0 rows affected (0.07 sec)

mysql> SHOW TABLES;
+------------------+
| Tables_in_demo_db |
+------------------+
| test_persistent   |
+------------------+
1 row in set (0.00 sec)
```

```
# Insert a record
mysql> INSERT INTO test_persistent (id, name)
    -> VALUES (100, "majd");
Query OK, 1 row affected (0.03 sec)

mysql> SELECT * FROM test_persistent;
+------+------+
| id   | name |
+------+------+
|  100 | majd |
+------+------+
1 row in set (0.00 sec)
```

After setting up the database, we are going to delete the pod and recreate it to check if data is missing.

```
# Delete the database pod
$ kubectl delete -f mysql-db.yaml
pod "mysql-db" deleted
# Recreate the database pod
$ kubectl apply -f mysql-db.yaml
pod/mysql-db created
# Check the status of the pod
$ kubectl get pods
NAME       READY   STATUS    RESTARTS   AGE
mysql-db   1/1     Running   0          15s
# open a mysql shell in the running mysql container
$ kubectl exec -it mysql-db -c mysql -- mysql -uroot -p123456

# Check the number of databases
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| demo_db            |
| information_schema |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
5 rows in set (0.01 sec)

mysql> USE demo_db;
Database changed

# Check the existence of the test record
mysql> SHOW TABLES;
+-------------------+
| Tables_in_demo_db |
+-------------------+
| test_persistent   |
+-------------------+
1 row in set (0.00 sec)

mysql> SELECT * FROM test_persistent;
+------+------+
| id   | name |
+------+------+
|  100 | majd |
+------+------+
1 row in set (0.00 sec)
```

As expected, the data still exists even though you deleted and recreated the pod. This confirms that you can use a GCE Persistent Disk to persist data across multiple instantiations of the same pod. To be perfectly precise, it isn't the same pod. These are two pods whose volumes point to the same underlying persistent storage volume.

Further Reading

You can refer to the official documentation pages for more information about volumes and persistent volumes.

1. https://kubernetes.io/docs/concepts/storage/volumes/ (https://kubernetes.io/docs/concepts/storage/volumes/)
2. https://kubernetes.io/docs/concepts/storage/persistent-volumes/ (https://kubernetes.io/docs/concepts/storage/persistent-volumes/)

References:

1. Kubernetes Documentation - Volumes (https://kubernetes.io/docs/concepts/storage/volumes/)
2. Kubernetes in Action, Second Edition (https://livebook.manning.com/book/kubernetes-in-action-second-edition/)

Danger

Clean Up Resources

You have finished all the tutorial in this primer, please clean up the cloud resources.

Delete the container cluster:

```
CLUSTER_NAME=gke-demo-cluster
gcloud container clusters delete ${CLUSTER_NAME}
```

Debugging Kubernetes

Debugging Kubernetes

To give you a better grasp of how to use kubernetes, we have provided a deployment configuration full of bugs, please use the following tools to try to get the right message.

Useful commands:

```
helm install <APPLICATION_NAME> <HELM_CHART_PATH>
helm list
helm uninstall <APPLICATION_NAME>
kubectl get deployment
kubectl get svc
kubectl get pods
kubectl describe pod <POD_ID>
kubectl logs <POD_ID>
kubectl describe configmaps <CONFIGMAP_NAME>
kubectl get configmaps <CONFIGMAP_NAME> -o yaml
docker images
docker container ls -a
docker logs <CONTAINER_NAME>
```

Tutorial Summary

In this tutorial, you are going to:

1. Use correct image registry url.
2. Use correct port mapping configuration.
3. Use correct labels.

Deploy the application

You can use the existing GCP project or create an empty project in GCP using the steps shared above.

You will finish the GKE tutorial in this section using Google Cloud Shell, which has the `gcloud`, `docker`, and `kubectl` command-line tools pre-installed.

Go to the Google Cloud Platform Console (https://console.cloud.google.com/) and open the Cloud Shell with the button at the top of the console. **You will use Google Cloud Shell to complete this section, and there is no need to provision any VM.**

Set work directory

```
# Create work directory
$ mkdir debug-example && cd debug-example
# Download compiled application
$ wget https://clouddeveloper.blob.core.windows.net/primers/container/buggy-app.
pyc
# Download helm chart
$ wget https://clouddeveloper.blob.core.windows.net/primers/container/helm.zip
$ unzip helm.zip
# You may use the Dockerfile in the docker primer
$ touch Dockerfile
# Copy the below buggy Dockerfile into the Dockerfile we just created
# You may use Vim or other editor to paste the code
# Download requirements.txt
$ wget https://clouddeveloper.blob.core.windows.net/primers/container/requiremen
ts.txt
```

Here is the "buggy" `Dockerfile` we are going to use.

```
# Dockerfile
FROM python:3.8.10

EXPOSE 8080

WORKDIR /code

COPY requirements.txt /code/requirements.txt
COPY buggy-app.pyc /code/buggy-app.pyc

RUN pip install --no-cache-dir --upgrade -r /code/requirements.txt

CMD ["python3", "buggy-app.pyc"]
```

Build docker images and push to GCR

```
# Authenticate to the Google Container Registry.
$ gcloud auth configure-docker
# Set your project ID and Compute Engine zone options for the gcloud tool
gcloud config set project ${PROJECT_ID}
gcloud config set compute/zone us-east1-d
# The name may contain only lowercase alphanumerics and '-',
# and it must start with a letter and end with an alphanumeric,
# and it must be no longer than 40 characters.
$ export CLUSTER_NAME=gke-demo-cluster
$ gcloud container clusters create ${CLUSTER_NAME} --num-nodes=3
...
Creating cluster gke-demo-cluster in us-east1-d...done.
Created [https://container.googleapis.com/v1/projects/debug-example-k8s/zones/us
-east1-d/clusters/gke-demo-cluster].
To inspect the contents of your cluster, go to: https://console.cloud.google.co
m/kubernetes/workload_/gcloud/us-east1-d/gke-demo-cluster?project=debug-example-
k8s
kubeconfig entry generated for gke-demo-cluster.
NAME: gke-demo-cluster
LOCATION: us-east1-d
MASTER_VERSION: 1.20.10-gke.1600
MASTER_IP: 35.231.100.162
MACHINE_TYPE: e2-medium
NODE_VERSION: 1.20.10-gke.1600
NUM_NODES: 3
STATUS: RUNNING

$ docker build -t debug-example:0.1.0 .
$ docker images
REPOSITORY      TAG       IMAGE ID       CREATED          SIZE
debug-example   0.1.0     33aa8adf40f9   22 seconds ago   996MB
python          3.8.10    a369814a9797   4 months ago     883MB
$ docker tag debug-example:0.1.0 us.gcr.io/debug-example-k8s/k8s-debug:v001
$ docker push us.gcr.io/debug-example-k8s/k8s-debug:v001
$ helm install debug-example helm/debug-example
```

Debugging in kubernetes

```
$ helm list
NAME            NAMESPACE       REVISION        UPDATED
STATUS          CHART                   APP VERSION
debug-example   default         1               2021-11-08 03:39:11.625376245 +0
000 UTC deployed        debug-sample-0.1.0      1.0
$ kubectl get deployment
NAME                   READY   UP-TO-DATE   AVAILABLE   AGE
k8s-debug-deployment   0/1     1            0           70s
$ kubectl get pods
NAME                                   READY   STATUS             RESTARTS   AG
E
k8s-debug-deployment-5c84d754c4-ls9xn   0/1     ImagePullBackOff   0          89
s
$ kubectl logs k8s-debug-deployment-5c84d754c4-ls9xn
Error from server (BadRequest): container "k8s-debug" in pod "k8s-debug-deployme
nt-5c84d754c4-ls9xn" is waiting to start: trying and failing to pull image
```

The error message indicates that we cannot pull image from the current url. Thus, we may change the url to `image: us.gcr.io/debug-example/k8s-debug:v001` in `deployment.yaml`.

```
$ helm uninstall debug-example
# After changing the image url
$ helm install debug-example helm/debug-example
$ kubectl get pods
NAME                                   READY   STATUS    RESTARTS   AGE
k8s-debug-deployment-7667cb7b8c-xr7fz   1/1     Running   0          30s
$ kubectl get svc
NAME               TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)        AGE
k8s-debug-service   NodePort    10.3.247.76   <none>        80:30861/TCP   85s
kubernetes          ClusterIP   10.3.240.1    <none>        443/TCP        13m
```

As we can see, there is no external ip for us to query. Thus, we need to change `NodePort` to `LoadBalancer`

```
# After changing the service.yaml
$ helm upgrade debug-example helm/debug-example
$ kubectl get svc
NAME               TYPE           CLUSTER-IP    EXTERNAL-IP    PORT(S)        A
GE
k8s-debug-service   LoadBalancer   10.3.247.76   35.227.22.64   80:30861/TCP   8
m23s
kubernetes          ClusterIP      10.3.240.1    <none>         443/TCP        2
0m
$ curl http://35.227.22.64:80/k8s
curl: (7) Failed to connect to 35.227.22.64 port 80: Connection refused
```

We got such an error message because we have a wrong port mapping configuration. Since the application only listens to port 8000, we should map 8000 to 80.

```
# After changing the port mapping configuration
$ helm upgrade debug-example helm/debug-example
$ curl http://35.227.22.64:80/k8s
curl: (7) Failed to connect to 35.227.22.64 port 80: Connection refused
```

Notice that we also have mismatch labels between service and deployment. Thus, we need to change the selector label in `service.yaml`.

```
# After changing the label
$ helm upgrade debug-example helm/debug-example
"Aoh, something going wrong."
```

This response message indicates that there is something going wrong in our pods, we should check the logs in the pods.

```
$ kubectl get pods
NAME                                    READY   STATUS    RESTARTS   AGE
k8s-debug-deployment-6d7d545f6c-mpbkt   1/1     Running   0          4m52s
$ kubectl logs k8s-debug-deployment-6d7d545f6c-mpbkt
INFO:     Started server process [1]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
Welcome to the K8S Debug Tutorial!
It looks like we are missing the environment variable CC_K8S that the program ne
eds.
Try to set CC_K8S="I LOVE CLOUD COMPUTING" in the configMap.
You can refer to https://kubernetes.io/docs/concepts/configuration/configmap/

INFO:     10.142.0.4:52446 - "GET /k8s HTTP/1.1" 200 OK
INFO:     10.0.1.1:3866 - "GET / HTTP/1.1" 200 OK
```

According to the error message, we should create a configMap.yaml and set the environment variable `CC_K8S` to `"I LOVE CLOUD COMPUTING"`.

```
# After adding the configMap.yaml
$ helm upgrade debug-example helm/debug-example
$ curl http://35.227.22.64:80/k8s
"Great! You pass the test!"
```

By now, we will pass all the debugging tests. In addition, we could use `kubectl describe configmaps <CONFIGMAP_NAME>` or `kubectl get configmaps <CONFIGMAP_NAME> -o yaml` to see the configuration.

```
$ kubectl describe configmaps debug-config
Name:         debug-config
Namespace:    default
Labels:       app.kubernetes.io/managed-by=Helm
Annotations:  meta.helm.sh/release-name: debug-example
              meta.helm.sh/release-namespace: default

Data
====
CC_K8S:
----
I LOVE CLOUD COMPUTING

BinaryData
====

Events:  <none>
$ kubectl get configmaps debug-config -o yaml
apiVersion: v1
data:
  CC_K8S: I LOVE CLOUD COMPUTING
kind: ConfigMap
metadata:
  annotations:
    meta.helm.sh/release-name: debug-example
    meta.helm.sh/release-namespace: default
  creationTimestamp: "2021-11-08T04:09:56Z"
  labels:
    app.kubernetes.io/managed-by: Helm
  name: debug-config
  namespace: default
  resourceVersion: "11448"
  uid: d1606bff-2ea7-426a-a913-2ce0f53004f0
```

Tips

If you got `curl: (7) Failed to connect to localhost port 80: Connection refused`, please check your port mapping policy and labels name and value.

Conclusion

Conclusion

You have gained the hands-on experience on Kubernetes and Helm. Congratulations!

Danger

Please remember to delete all resources in this primer as you may incur charges, which include:

1. The GCP project which includes the GKS cluster

2. The Azure resource group which includes the AKS cluster
3. The Ubuntu 18.04 VM if any