**Creating Services and Ingress Resources**

Toggle Lab Panel



**Creating Services and Ingress Resources**

1 hour Free

Rate Lab

**Overview**

In this lab, you will create two deployments of Pods and work with three different types of services, including the Ingress resource, and explore how Kubernetes services in GKE are associated with GCP Network Load Balancers.

**Objectives**

In this lab, you learn how to perform the following tasks:

* Observe Kubernetes DNS in action.
* Define various service types (ClusterIP, NodePort, LoadBalancer) in manifests along with label selectors to connect to existing labeled Pods and deployments, deploy those to a cluster, and test connectivity.
* Deploy an Ingress resource that connects clients to two different services based on the URL path entered.
* Verify Google Cloud Platform network load balancer creation for type=LoadBalancer services.

**Task 0. Lab Setup**

**Access Qwiklabs**

For each lab, you get a new GCP project and set of resources for a fixed time at no cost.

1. Make sure you signed into Qwiklabs using an **incognito window**.
2. Note the lab's access time (for example, img/time.pngand make sure you can finish in that time block.

There is no pause feature. You can restart if needed, but you have to start at the beginning.

1. When ready, click img/start_lab.png.
2. Note your lab credentials. You will use them to sign in to Cloud Platform Console. 
3. Click **Open Google Console**.
4. Click **Use another account** and copy/paste credentials for **this** lab into the prompts.

If you use other credentials, you'll get errors or **incur charges**.

1. Accept the terms and skip the recovery resource page.

Do not click **End Lab** unless you are finished with the lab or want to restart it. This clears your work and removes the project.

After you complete the initial sign-in steps, the project dashboard appears.

**Open Cloud Shell**

You will do most of the work for this lab in [Cloud Shell](https://cloud.google.com/developer-shell/#how_do_i_get_started).

1. On the GCP Console title bar, click **Activate Cloud Shell (** e92fcd01cbb5e0ff.png**)**.
2. Click **Start Cloud Shell**.

After a moment of provisioning, the Cloud Shell prompt appears:

**Task 1. Connect to the lab GKE cluster and test DNS**

In this task, you connect to the lab GKE cluster and create a deployment manifest for a set of Pods within the cluster that you will then use to test DNS resolution of Pod and service names.

**Connect to the lab GKE cluster**

1. In Cloud Shell, type the following command to create environment variables for the GCP zone and cluster name that will be used to create the cluster for this lab.

export my\_zone=us-central1-a

export my\_cluster=standard-cluster-1

1. Configure tab completion for the kubectl command-line tool.

source <(kubectl completion bash)

1. Configure access to your cluster for kubectl:

gcloud container clusters get-credentials $my\_cluster --zone $my\_zone

**Task 2. Create Pods and services to test DNS resolution**

You will create a service called dns-demo with two sample application Pods called dns-demo-1 and dns-demo-2.

**Clone the source file repository and deploy the sample application pods**

The pods that you will use to test network connectivity via Kubernetes services are deployed using the dns-demo.yaml manifest file that has been provided for you in the source repository. You will use these pods later to test connectivity via various services from systems inside your Kubernetes cluster. You will also test connectivity from external addresses using the Cloud Shell. You can use the Cloud Shell for this because it is on a network that is completely separate to your Kubernetes cluster networks.

apiVersion: v1

kind: Service

metadata:

name: dns-demo

spec:

selector:

name: dns-demo

clusterIP: None

ports:

- name: dns-demo

port: 1234

targetPort: 1234

---

apiVersion: v1

kind: Pod

metadata:

name: dns-demo-1

labels:

name: dns-demo

spec:

hostname: dns-demo-1

subdomain: dns-demo

containers:

- name: nginx

image: nginx

---

apiVersion: v1

kind: Pod

metadata:

name: dns-demo-2

labels:

name: dns-demo

spec:

hostname: dns-demo-2

subdomain: dns-demo

containers:

- name: nginx

image: nginx

1. In Cloud Shell enter the following command to clone the repository to the lab Cloud Shell.

git clone https://github.com/GoogleCloudPlatform/training-data-analyst

1. Create a soft link as a shortcut to the working directory.

ln -s ~/training-data-analyst/courses/ak8s/v1.1 ~/ak8s

1. Change to the directory that contains the sample files for this lab.

cd ~/ak8s/GKE\_Services/

1. Create the service and Pods.

kubectl apply -f dns-demo.yaml

1. Verify that your Pods are running.

kubectl get pods

The output should look like this example.

**Output (do not copy)**

NAME READY STATUS RESTARTS AGE

dns-demo-1 1/1 Running 0 8s

dns-demo-2 1/1 Running 0 8s

Click *Check my progress* to verify the objective. Create Pods and services to test DNS resolution

1. To get inside the cluster, open an interactive session to bash running from dns-demo-1.

kubectl exec -it dns-demo-1 /bin/bash

Now that you are inside a container in the cluster, subsequent commands will run from that context and you will use this to test network connectivity by pinging various targets in later tasks. However, you don't have a tool to ping in this container, so you need to install the ping command first.

1. Update apt-get and install a ping tool.

apt-get update

apt-get install -y iputils-ping

1. Ping dns-demo-2:

ping dns-demo-2.dns-demo.default.svc.cluster.local

This ping should succeed and report that the target has the ip-address you found earlier for the dns-demo-2 Pod.

1. Press Ctrl+C to abort the ping command.
2. Ping the dns-demo service's FQDN, instead of a specific Pod inside the service:

ping dns-demo.default.svc.cluster.local

This ping should also succeed but it will return a response from the FQDN of one of the two demo-dns Pods. This Pod might be either demo-dns-1 or demo-dns-2.

1. Press Ctrl+C to abort the ping command.

When you deploy applications, your application code runs inside a container in the cluster, and thus your code can access other services by using the FQDNs of those services from inside that cluster. This approach is simpler than using IP addresses or even Pod names, because those are more likely to change.

1. Leave the interactive shell on dns-demo-1 running.

**Task 3. Deploy a sample workload and a ClusterIP service**

In this task, you create a deployment manifest for a set of Pods within the cluster and then expose them using a ClusterIP service.

**Deploy a sample web application to your Kubernetes Engine cluster**

In this task you will deploy a sample web application container image that listens on an HTTP server on port 8080 using the following manifest that has already been created for you in the hello-v1.yaml manifest file.

apiVersion: extensions/v1beta1

kind: Deployment

metadata:

name: hello-v1

spec:

replicas: 3

selector:

matchLabels:

run: hello-v1

template:

metadata:

labels:

run: hello-v1

name: hello-v1

spec:

containers:

- image: gcr.io/google-samples/hello-app:1.0

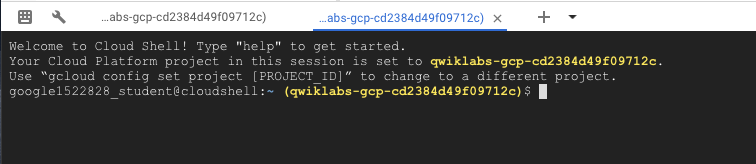
name: hello-v1

ports:

- containerPort: 8080

protocol: TCP

1. In the Cloud Shell menu bar, click the plus sign (+) icon to start a new Cloud Shell session.



A second Cloud Shell session appears in your Cloud Shell window. You can switch sessions by clicking them in the menu bar.

1. In the second Cloud Shell tab, change to the directory that contains the sample files for this lab.

cd ~/ak8s/GKE\_Services/

1. To create a deployment from the hello-v1.yaml file, execute the following command:

kubectl create -f hello-v1.yaml

1. To view a list of deployments, execute the following command:

kubectl get deployments

The output should look like this example.

**Output (do not copy)**

NAME READY UP-TO-DATE AVAILABLE AGE

hello-v1 3/3 3 3 14s

**Define service types in the manifest**

In this task you will deploy a Service using a ClusterIP using the hello-svc.yaml sample manifest that has already been created for you.

apiVersion: v1

kind: Service

metadata:

name: hello-svc

spec:

type: ClusterIP

selector:

name: hello-v1

ports:

- protocol: TCP

port: 80

targetPort: 8080

1. To deploy the ClusterIP service, execute the following command:

kubectl apply -f ./hello-svc.yaml

This manifest defines a ClusterIP service and applies it to Pods that correspond to the selector. In this case, the manifest is applied to the hello-v1 Pods that you deployed. This service will automatically be applied to any other deployments with the name: hello-v1 label.

1. Verify that the service was created and that a Cluster-IP was allocated:

kubectl get service hello-svc

Your IP address might be different from the example output.

**Output (do not copy)**

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

hello-svc ClusterIP 10.11.253.203 <none> 80/TCP 20s

No external IP is allocated for this service. Because the Kubernetes Cluster IP addresses are not externally accessible by default, creating this service does not make your application accessible outside of the cluster.

Click *Check my progress* to verify the objective. Deploy a sample workload and a ClusterIP service

**Test your application**

1. In the Cloud Shell, attempt to open an HTTP session to the new service using the following command:

curl hello-svc.default.svc.cluster.local

The connection should fail because that service is not exposed outside of the cluster.

**Output (do not copy)**

curl: (6) Could not resolve host: hello-svc.default.svc.cluster.local

Now you will test the service from inside the cluster using the interactive shell you have running on the dns-demo-1 Pod.

1. Return to your first Cloud Shell window, which is currently redirecting the stdin and stdout of the dns-demo-1 Pod.
2. Install curl so you can make calls to web services from the command line.

apt-get install -y curl

1. Use the following command to test the HTTP connection between the Pods.

curl hello-svc.default.svc.cluster.local

This connection should succeed and provide a response similar to the output below. Your hostname might be different from the example output.

**Output (do not copy)**

root@dns-demo-1:/# curl hello-svc.default.svc.cluster.local

Hello, world!

Version: 1.0.0

Hostname: hello-v1-85b99869f-mp4vd

root@dns-demo-1:/#

This connection works because the clusterIP can be resolved using the internal DNS within the Kubernetes Engine cluster.

**Task 4. Convert the service to use NodePort**

In this task, you will convert your existing ClusterIP service to a NodePort service and then retest access to the service from inside and outside the cluster.

A modified version of the hello-svc.yaml file called hello-nodeport-svc.yaml that changes the service type to NodePort has already been created for you.

apiVersion: v1

kind: Service

metadata:

name: hello-svc

spec:

type: NodePort

selector:

name: hello-v1

ports:

- protocol: TCP

port: 80

targetPort: 8080

nodePort: 30100

1. Return to your second Cloud Shell window. This is the window NOT connected to the stdin and stdout of the dns-test Pod.
2. To deploy the manifest that changes the service type for the hello-svc to NodePort , execute the following command:

kubectl apply -f ./hello-nodeport-svc.yaml

This manifest redefines hello-svc as a NodePort service and assigns the service port 30100 on each node of the cluster for that service.

1. Enter the following command to verify that the service type has changed to NodePort:

kubectl get service hello-svc

Your IP address might be different from the example output.

**Output (do not copy)**

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

hello-svc NodePort 10.11.253.203 <none> 80:30100/TCP 59m

Note that there is still no external IP allocated for this service.

Click *Check my progress* to verify the objective. Convert the service to use NodePort

**Test your application**

1. In the Cloud Shell, attempt to open an HTTP session to the new service using the following command:

curl hello-svc.default.svc.cluster.local

The connection should fail because that service is not exposed outside of the cluster.

**Output (do not copy)**

curl: (6) Could not resolve host: hello-svc.default.svc.cluster.local

Now you will test the service from another Pod.

1. Return to your first Cloud Shell window, which is currently redirecting the stdin and stdout of the dns-test Pod.
2. Use the following command to test the HTTP connection between the Pods.

curl hello-svc.default.svc.cluster.local

This connection should succeed and provide a response similar to the output below. Your hostname might be different from the example output.

**Output (do not copy)**

root@dns-demo-1:/# curl hello-svc.default.svc.cluster.local

Hello, world!

Version: 1.0.0

Hostname: hello-v1-85b99869f-mp4vd

root@dns-demo-1:/#

This connection works because the clusterIP can be resolved using the internal DNS within the GKE cluster.

**Task 5. Create static public IP addresses using GCP Networking**

**Reserve GCP Networking Static IP Addreesses**

1. In the GCP Console navigation menu, navigate to **Networking** > **VPC Network** > **External IP Addresses**.
2. Click **+ Reserve Static Address**.
3. Enter regional-loadbalancer for the **Name**.
4. Explore the options but leave the remaining settings at their defaults. Note that the default type is **Regional**.

Note that the default region must match the region where you have deployed your GKE cluster. If this is not the case then change the region here to match the region you used to create your cluster. If you accepted the lab defaults then that should be the us-central1 (Iowa) region.

1. Click **Reserve**.
2. Make a note of the external ip-address called regional-loadbalancer. You will use this in a later task.
3. Click **+ Reserve Static Address**.
4. Enter global-ingress for the **Name**.
5. Change the Type to **Global**.
6. Leave the remaining settings at their defaults.
7. Click **Reserve**.
8. Make a note of the external ip-address called global-ingress. You will use this in a later task.

Click *Check my progress* to verify the objective. Create static public IP addresses using GCP Networking

**Task 6. Deploy a new set of Pods and a LoadBalancer service**

You will now deploy a new set of Pods running a different version of the application so that you can easily differentiate the two services. You will then expose the new Pods as a LoadBalancer service and access the service from outside the cluster.

A second Deployment manifest called hello-v2.yaml has been created for you that creates a new deployment that runs version 2 of the sample hello application on port 8080.

apiVersion: extensions/v1beta1

kind: Deployment

metadata:

name: hello-v2

spec:

replicas: 3

selector:

matchLabels:

run: hello-v2

template:

metadata:

labels:

run: hello-v2

name: hello-v2

spec:

containers:

- image: gcr.io/google-samples/hello-app:2.0

name: hello-v2

ports:

- containerPort: 8080

protocol: TCP

1. Return to your second Cloud Shell window. This is the window NOT connected to the stdin and stdout of the dns-test Pod.
2. To deploy the manifest that creates the hello-v2 deployment execute the following command:

kubectl create -f hello-v2.yaml

1. To view a list of deployments, execute the following command:

kubectl get deployments

The output should look like this example.

**Output (do not copy)**

NAME READY UP-TO-DATE AVAILABLE AGE

hello-v1 3/3 3 3 18m

hello-v2 3/3 3 3 7s

**Define service types in the manifest**

In this task you will deploy a LoadBalancer Service using the hello-lb-svc.yaml sample manifest that has already been created for you.

You will use the sed command to replace the 10.10.10.10 placeholder address in the load balancer yaml file with the static address you reserved earlier for the load balancer.

apiVersion: v1

kind: Service

metadata:

name: hello-lb-svc

spec:

type: LoadBalancer

loadBalancerIP: 10.10.10.10

selector:

name: hello-v2

ports:

- protocol: TCP

port: 80

targetPort: 8080

1. Make sure you are still in the second Cloud Shell window. This is the window NOT connected to the stdin and stdout of the dns-test Pod.
2. Enter the following command in the Cloud Shell to save the regional static IP-address you created earlier into an environment variable.

export STATIC\_LB=$(gcloud compute addresses describe regional-loadbalancer --region us-central1 --format json | jq -r '.address')

1. Enter the following command in the Cloud Shell to replace the placeholder address with the regional static IP-address.

sed -i "s/10\.10\.10\.10/$STATIC\_LB/g" hello-lb-svc.yaml

1. To check that the external ip-address has been replaced correctly enter the following command in the Cloud shell.

cat hello-lb-svc.yaml

The loadBalancerIP should match the address you recorded earlier for the regional-loadbalancer reserved static ip-address.

1. To deploy your LoadBalancer service manifest, execute the following command:

kubectl apply -f ./hello-lb-svc.yaml

This manifest defines a LoadBalancer service, which deploys a GCP Network Load Balancer to provide external access to the service. This service is only applied to the Pods with the name: hello-v2 selector.

1. Verify that the service was created and that a node port was allocated:

kubectl get services

Your IP address might be different from the example output.

**Output (do not copy)**

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

dns-demo ClusterIP None <none> 1234/TCP 1h

hello-lb-svc LoadBalancer 10.11.253.48 35.184.45.240 80:30103/TCP 2m

hello-svc NodePort 10.11.253.203 <none> 80:30100/TCP 1h

kubernetes ClusterIP 10.11.240.1 <none> 443/TCP 2h

Notice that the new LoadBalancer service lists the regional-loadbalancer reserved address as its external IP address. In GKE environments the external load balancer for the LoadBalancer service type is implemented using a GCP load balancer and will take a few minutes to create. This external IP address makes the service accessible from outside the cluster.

**Confirm that a Regional GCP Load Balancer has been created**

1. In the GCP Console Navigate to **Networking** > **Network Services** > **Load Balancing**
2. You should see a TCP Load Balancer listed with 1 target pool backend and 2 instances. Click the name to open the details page.
3. The details should show that it is a regional TCP load balancer using the static IP address you called regional-loadbalancer when you reserved it earlier.

Click *Check my progress* to verify the objective. Deploy a new set of Pods and a LoadBalancer service

**Test your application**

1. In the Cloud Shell, attempt to open an HTTP session to the new service using the following command:

curl hello-lb-svc.default.svc.cluster.local

The connection should fail because that service name is not exposed outside of the cluster.

**Output (do not copy)**

curl: (6) Could not resolve host: hello-lb-svc.default.svc.cluster.local

This occurs because the external IP address is not registered with this hostname..

1. Try the connection again using the External IP address associated with the REgional Load Balancer service. Type the following command and substitute [external\_IP] with your service's external IP address.

curl [external\_IP]

**Example (do not copy)**

curl 35.184.45.240

This time the connection does not fail because the LoadBalancer's external ip-address can be reached from outside GCP.

**Note:** This may take up to 5 minutes to become available.

**Output (do not copy)**

Hello, world!

Version: 2.0.0

Hostname: hello-v2-59c99d8b65-jrx2d

1. Return to your first Cloud Shell window, which is currently redirecting the stdin and stdout of the dns-demo-1 Pod.
2. Use the following command to test the HTTP connection between the Pods.

curl hello-lb-svc.default.svc.cluster.local

This connection should succeed and provide a response similar to the output below. Your hostname will be different from the example output.

**Output (do not copy)**

root@dns-demo-1:/# curl hello-lb-svc.default.svc.cluster.local

Hello, world!

Version: 2.0.0

Hostname: hello-v2-59c99d8b65-78ggz

root@dns-demo-1:/#

The internal DNS name works within the Pod, and you can see that you are accessing the same v2 version of the application as you were from outside of the cluster using the external ip address.

1. Try the connection again within the Pod using the External IP address associated with the service. Type the following command and substitute [external\_IP] with your service's external IP address.

curl [external\_IP]

**Example (do not copy)**

curl 35.184.45.240

**Output (do not copy)**

root@dns-demo-1:/# curl 35.184.45.240

Hello, world!

Version: 2.0.0

Hostname: hello-v2-59c99d8b65-jrx2d

root@dns-demo-1:/#

The external IP also works from inside Pods running in the cluster, and returns a result from the same v2 version of the applications.

1. Exit the console redirection session to the Pod by typing exit.

exit

This will return you to the Cloud Shell command prompt.

**Task 7. Deploy an Ingress resource**

You have two services in your cluster for the hello application. One service is hosting version 1.0 via a NodePort service, while the other service is hosting version 2.0 via a LoadBalancer service. You will now deploy an Ingress resource that will direct traffic to both services based on the URL entered by the user.

**Create an ingress resource**

Ingress is a Kubernetes resource that encapsulates a collection of rules and configuration for routing external HTTP(S) traffic to internal services.

On GKE, Ingress is implemented using Cloud Load Balancing. When you create an ingress resource in your cluster, GKE creates an HTTP(S) load balancer and configures it to route traffic to your application.

A sample manifest called hello-ingress.yaml that will configure an ingress resource has been created for you.

apiVersion: extensions/v1beta1

kind: Ingress

metadata:

name: hello-ingress

annotations:

nginx.ingress.kubernetes.io/rewrite-target: /

kubernetes.io/ingress.global-static-ip-name: "global-ingress"

spec:

rules:

- http:

Paths:

- path: /v1

backend:

serviceName: hello-svc

servicePort: 80

- path: /v2

backend:

serviceName: hello-lb-svc

servicePort: 80

This configuration file defines an ingress resource that will attach to the global-ingress static ip-address you created in a previous step. This ingress resource will create a global HTTP(S) load balancer that directs traffic to your web services based on the path entered.

The kubernetes.io/ingress.global-static-ip-name annotation allows you to specify a named resxerved ip-address that the Ingress resource will use when creating the GCP Global HTTP(S) load balancer for the Ingress resource.

1. To deploy this ingress resource, execute the following command:

kubectl apply -f hello-ingress.yaml

When you deploy this manifest, Kubernetes creates an ingress resource on your cluster. The ingress controller running in your cluster is responsible for creating an HTTP(S) load balancer to route all external HTTP traffic (on port 80) to the web NodePort service and the LoadBalancer service that you exposed.

**Confirm that a Global HTTP(S) Load Balancer has been created**

1. In the GCP Console Navigate to **Networking** > **Network Services** > **Load Balancing**
2. You should now also see a HTTP(S) Load Balancer listed. Click the name to open the details page.
3. The details should show that it is a global HTTP(S) load balancer using the static IP address you called global-ingress when you reserved it earlier.

Click *Check my progress* to verify the objective. Deploy an Ingress resource

**Test your application**

1. To get the external IP address of the load balancer serving your application, execute the following command:

kubectl describe ingress hello-ingress

**Partial output (do not copy)**

Name: hello-ingress

Namespace: default

Address: 35.244.173.44

Default backend: default-http-backend:80 (10.8.2.5:8080)

Rules:

Host Path Backends

---- ---- --------

\*

/v1 hello-svc:80 (<none>)

/v2 hello-lb-svc:80 (<none>)

[...]

ingress.kubernetes.io/backends:{"k8s-[...]"HEALTHY"[...]}

[...]

Events:

Type Reason Age From Message

---- ------ --- ---- -------

Normal ADD 5m loadbalancer-controller default/hello-ingress

Normal CREATE 5m loadbalancer-controller ip: 35.244.173.44

**Note:** You need to wait for a few minutes for the load balancer to become active, and for the health checks to succeed, before the external address will be displayed.

Repeat the command every few minutes to check if the Ingress resource has finished initializing. When the GCP global HTTP(S) load balancer has been created and initialized the command will report the external ip-address which will match the global static ip-address you reserved earlier that you called global-ingress.

1. Use the External IP address associated with the Ingress resource, and type the following command, substituting [external\_IP] with the Ingress resource's external IP address. Be sure to include the /v1 in the URL path.

curl http://[external\_IP]/v1

**Example (do not copy)**

curl http://35.201.92.69/v1

**Output (do not copy)**

Hello, world!

Version: 1.0.0

Hostname: hello-v1-85b99869f-4rmqw

The v1 URL is configured in hello-ingress.yaml to point to the hello-svc NodePort service that directs traffic to the v1 application Pods. .

**Note:** GKE might take a few minutes to set up forwarding rules until the Global load balancer used for the Ingress resource is ready to serve your application. In the meantime, you might get errors such as HTTP 404 or HTTP 500 until the load balancer configuration is propagated across the globe.

1. Now test the v2 URL path from Cloud Shell. Use the External IP address associated with the Ingress resource, and type the following command, substituting [external\_IP] with the Ingress resource's external IP address. Be sure to include the /v2 in the URL path.

curl http://[external\_IP]/v2

The v2 URL is configured in hello-ingress.yaml to point to the hello-lb-svc LoadBalancer service that directs traffic to the v2 application Pods.

**Example (do not copy)**

curl http://35.201.92.69/v2

**Output (do not copy)**

Hello, world!

Version: 2.0.0

Hostname: hello-v2-59c99d8b65-jrx2d

**Inspect the changes to your networking resources in the GCP Console**

1. In GCP Console, on the **Navigation menu** , click **Networking** > **Network services**> **Load balancing**.

There are now two load balancers listed:

There is the initial regional load balancer for the hello-lb-svc service. This has a UID style name and is configured to load balance TCP port 80 traffic to the cluster nodes.

The second was created for the Ingress object and is a full HTTP(S) load balancer that includes host and path rules that match the Ingress configuration. This will have hello-ingress in its name.

1. Click the load balancer with hello-ingress in the name.

This will display the summary information about the protocols, ports, paths and backend services of the GCP Global HTTP(S) load balancer created for your Ingress resource.

**End your lab**