Managing Deployments Using Kubernetes Engine

1 hour 7 Credits
Rate Lab

GSP053

Overview

Dev Ops practices will regularly make use of multiple deployments to manage application deployment scenarios such as "Continuous Deployment", "Blue-Green Deployments", "Canary Deployments" and more. This lab is to provide practice in scaling and managing containers so you can accomplish these common scenarios where multiple heterogeneous deployments are being used.

What you'll do

- Practice with kubectl tool
- Create deployment yaml files
- Launch, update, and scale deployments
- Practice with updating deployments and deployment styles

Prerequisites

- Before taking this lab, you should have taken at least the labs Introduction to Docker and Hello Node Kubernetes
- Linux System Administration skills
- Dev Ops theory: concepts of continuous deployment

Introduction to deployments

Heterogeneous deployments typically involve connecting two or more distinct infrastructure environments or regions to address a specific technical or operational need. Heterogeneous deployments are called "hybrid", "multi-cloud", or "public-private", depending upon the specifics of the deployment. For the purposes of this lab, heterogeneous deployments include those that span regions within a single cloud environment, multiple public cloud environments (multi-cloud), or a combination of on-premises and public cloud environments (hybrid or public-private).

Various business and technical challenges can arise in deployments that are limited to a single environment or region:

- **Maxed out resources**: In any single environment, particularly in on-premises environments, you might not have the compute, networking, and storage resources to meet your production needs.
- **Limited geographic reach**: Deployments in a single environment require people who are geographically distant from one another to access one deployment. Their traffic might travel around the world to a central location.
- **Limited availability**: Web-scale traffic patterns challenge applications to remain fault-tolerant and resilient.
- **Vendor lock-in**: Vendor-level platform and infrastructure abstractions can prevent you from porting applications.
- **Inflexible resources**: Your resources might be limited to a particular set of compute, storage, or networking offerings.

Heterogeneous deployments can help address these challenges, but they must be architected using programmatic and deterministic processes and procedures. One-off or ad-hoc deployment procedures can cause deployments or processes to be brittle and intolerant of failures. Ad-hoc processes can lose data or drop traffic. Good deployment processes must be repeatable and use proven approaches for managing provisioning, configuration, and maintenance.

Three common scenarios for heterogeneous deployment are multi-cloud deployments, fronting on-premises data, and continuous integration/continuous delivery (CI/CD) processes.

The following exercises practice some common use cases for heterogeneous deployments, along with well-architected approaches using Kubernetes and other infrastructure resources to accomplish them.

Setup

Before you click the Start Lab button

Read these instructions. Labs are timed and you cannot pause them. The timer, which starts when you click **Start Lab**, shows how long Google Cloud resources will be made available to you.

This Qwiklabs hands-on lab lets you do the lab activities yourself in a real cloud environment, not in a simulation or demo environment. It does so by giving you new, temporary credentials that you use to sign in and access Google Cloud for the duration of the lab.

What you need

To complete this lab, you need:

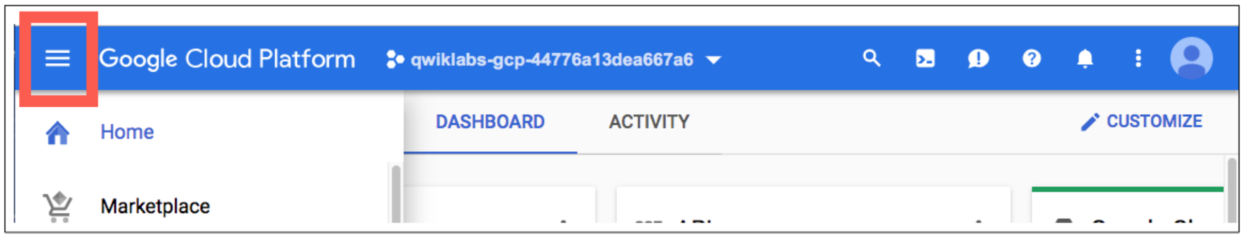
- Access to a standard internet browser (Chrome browser recommended).
- Time to complete the lab.

Note: If you already have your own personal Google Cloud account or project, do not use it for this lab.

Note: If you are using a Pixelbook, open an Incognito window to run this lab.

After a few moments, the Cloud Console opens in this tab.

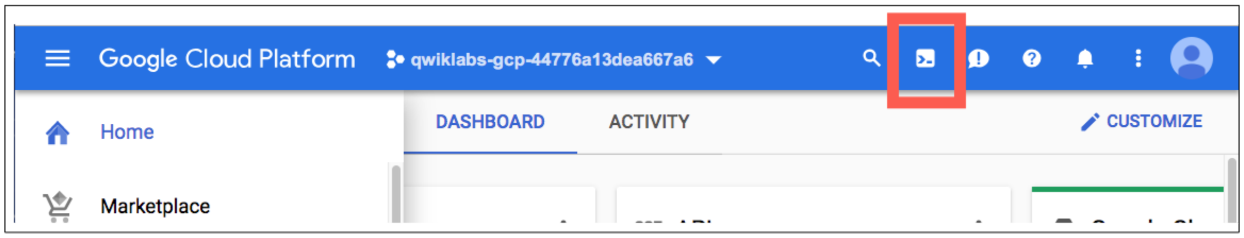
Note: You can view the menu with a list of Google Cloud Products and Services by clicking the **Navigation menu** at the top-left.



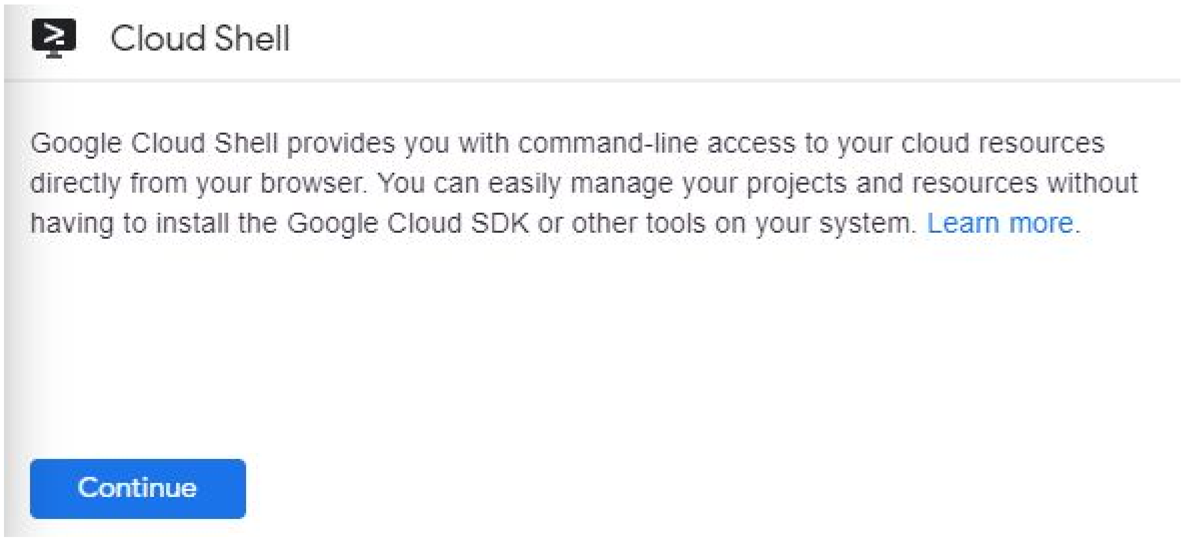
Activate Cloud Shell

Cloud Shell is a virtual machine that is loaded with development tools. It offers a persistent 5GB home directory and runs on the Google Cloud. Cloud Shell provides command-line access to your Google Cloud resources.

In the Cloud Console, in the top right toolbar, click the **Activate Cloud Shell** button.

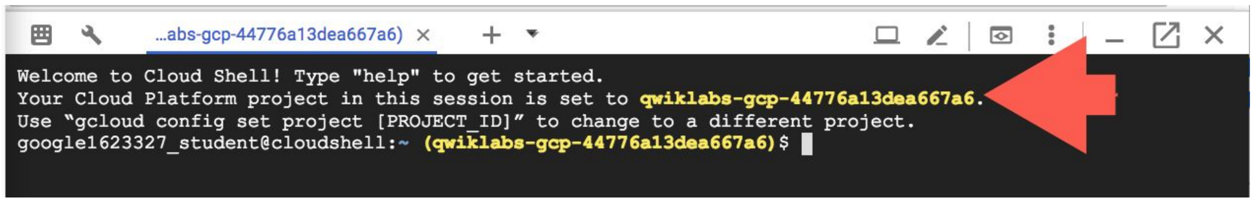


Click **Continue**.


Cloud Shell

Google Cloud Shell provides you with command-line access to your cloud resources directly from your browser. You can easily manage your projects and resources without having to install the Google Cloud SDK or other tools on your system. Learn more.

It takes a few moments to provision and connect to the environment. When you are connected, you are already authenticated, and the project is set to your *PROJECT_ID*. For example:



```
Welcome to Cloud Shell! Type "help" to get started.
Your Cloud Platform project in this session is set to qwiklabs-gcp-44776a13dea667a6.
Use "gcloud config set project [PROJECT_ID]" to change to a different project.
google1623327_student@cloudshell:~ (qwiklabs-gcp-44776a13dea667a6)$
```

gcloud is the command-line tool for Google Cloud. It comes pre-installed on Cloud Shell and supports tab-completion.

You can list the active account name with this command:

gcloud auth list

(Output)

Credentialed accounts:
 - <myaccount>@<mydomain>.com (active)

(Example output)

Credentialed accounts:
 - google1623327_student@qwiklabs.net

You can list the project ID with this command:

gcloud config list project

(Output)

```
[core]
project = <project_ID>
```

(Example output)

```
[core]
project = qwiklabs-gcp-44776a13dea667a6
```

For full documentation of gcloud see the gcloud command-line tool overview.

Set zone

Set your working Google Cloud zone by running the following command, substituting the local zone as us-central1-a:

```
gcloud config set compute/zone us-central1-a
```

Get sample code for this lab

Get the sample code for creating and running containers and deployments:

```
git clone https://github.com/googlecodelabs/orchestrate-with-kubernetes.git
cd orchestrate-with-kubernetes/kubernetes
```

Create a cluster with five n1-standard-1 nodes (this will take a few minutes to complete):

```
gcloud container clusters create bootcamp --num-nodes 5 --scopes
"https://www.googleapis.com/auth/projecthosting,storage-rw"
```

Learn about the deployment object

Let's get started with Deployments. First let's take a look at the Deployment object. The explain command in kubectl can tell us about the Deployment object.

```
kubectl explain deployment
```

We can also see all of the fields using the --recursive option.

```
kubectl explain deployment --recursive
```

You can use the explain command as you go through the lab to help you understand the structure of a Deployment object and understand what the individual fields do.

```
kubectl explain deployment.metadata.name
```

Create a deployment

Update the deployments/auth.yaml configuration file:

```
vi deployments/auth.yaml
```

Start the editor:

```
i
```

Change the image in the containers section of the Deployment to the following:

```
...
containers:
- name: auth
  image: kelseyhightower/auth:1.0.0
...
```

Save the auth.yaml file: press <Esc> then:

```
:wq
```

Now let's create a simple deployment. Examine the deployment configuration file:

```
cat deployments/auth.yaml
```

(Output)

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: auth
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: auth
        track: stable
    spec:
      containers:
        - name: auth
          image: "kelseyhightower/auth:1.0.0"
          ports:
            - name: http
              containerPort: 80
            - name: health
              containerPort: 81
...
```

Notice how the Deployment is creating one replica and it's using version 1.0.0 of the auth container.

When you run the kubectl create command to create the auth deployment, it will make one pod that conforms to the data in the Deployment manifest. This means we can scale the number of Pods by changing the number specified in the replicas field.

Go ahead and create your deployment object using kubectl create:

```
kubectl create -f deployments/auth.yaml
```

Once you have created the Deployment, you can verify that it was created.

```
kubectl get deployments
```

Once the deployment is created, Kubernetes will create a ReplicaSet for the Deployment. We can verify that a ReplicaSet was created for our Deployment:

```
kubectl get replicasets
```

We should see a ReplicaSet with a name like auth-xxxxxxx

Finally, we can view the Pods that were created as part of our Deployment. The single Pod is created by the Kubernetes when the ReplicaSet is created.

```
kubectl get pods
```

It's time to create a service for our auth deployment. You've already seen service manifest files, so we won't go into the details here. Use the kubectl create command to create the auth service.

```
kubectl create -f services/auth.yaml
```

Now, do the same thing to create and expose the hello Deployment.

```
kubectl create -f deployments/hello.yaml
kubectl create -f services/hello.yaml
```

And one more time to create and expose the frontend Deployment.

```
kubectl create secret generic tls-certs --from-file tls/
kubectl create configmap nginx-frontend-conf --from-file=nginx/frontend.conf
kubectl create -f deployments/frontend.yaml
kubectl create -f services/frontend.yaml
```

Note: You created a ConfigMap for the frontend.

Interact with the frontend by grabbing its external IP and then curling to it.

```
kubectl get services frontend
```

It may take a few seconds before the External-IP field is populated for your service. This is normal. Just re-run the above command every few seconds until the field is populated.

```
curl -ks https://<EXTERNAL-IP>
```

And you get the hello response back.

You can also use the output templating feature of kubectl to use curl as a one-liner:

```
curl -ks https://`kubectl get svc frontend -o=jsonpath="{.status.loadBalancer.ingress[0].ip}"`
```

Test Completed Task

Click **Check my progress** below to check your lab progress. If you successfully created Kubernetes cluster and Auth, Hello and Frontend deployments, you'll see an assessment score.

Create a Kubernetes cluster and deployments (Auth, Hello, and Frontend)

Scale a Deployment

Now that we have a Deployment created, we can scale it. Do this by updating the spec.replicas field. You can look at an explanation of this field using the kubectl explain command again.

```
kubectl explain deployment.spec.replicas
```

The replicas field can be most easily updated using the kubectl scale command:

```
kubectl scale deployment hello --replicas=5
```

Note: It may take a minute or so for all the new pods to start up.

After the Deployment is updated, Kubernetes will automatically update the associated ReplicaSet and start new Pods to make the total number of Pods equal 5.

Verify that there are now 5 hello Pods running:

```
kubectl get pods | grep hello- | wc -l
```

Now scale back the application:

```
kubectl scale deployment hello --replicas=3
```

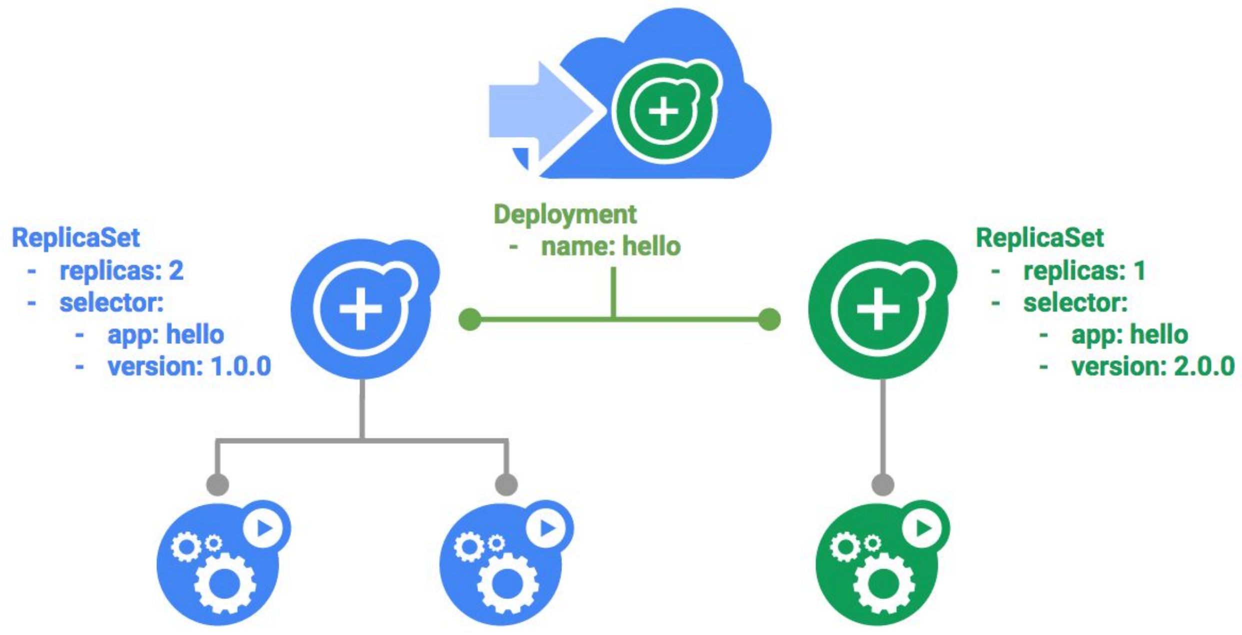
Again, verify that you have the correct number of Pods.

```
kubectl get pods | grep hello- | wc -l
```

You learned about Kubernetes deployments and how to manage & scale a group of Pods.

Rolling update

Deployments support updating images to a new version through a rolling update mechanism. When a Deployment is updated with a new version, it creates a new ReplicaSet and slowly increases the number of replicas in the new ReplicaSet as it decreases the replicas in the old ReplicaSet.



Trigger a rolling update

To update your Deployment, run the following command:

```
kubectl edit deployment hello
```

Change the image in the containers section of the Deployment to the following:

```
...
containers:
- name: hello
  image: kelseyhightower/hello:2.0.0
...
```

Save and exit.

Once you save out of the editor, the updated Deployment will be saved to your cluster and Kubernetes will begin a rolling update.

See the new ReplicaSet that Kubernetes creates.:

```
kubectl get replicaset
```

You can also see a new entry in the rollout history:

```
kubectl rollout history deployment/hello
```

Pause a rolling update

If you detect problems with a running rollout, pause it to stop the update. Give that a try now:

```
kubectl rollout pause deployment/hello
```

Verify the current state of the rollout:

```
kubectl rollout status deployment/hello
```

You can also verify this on the Pods directly:

```
kubectl get pods -o jsonpath --template='{range
.items[*]}{.metadata.name}{"\t"}{"\t"}{.spec.containers[0].image}{"\n"}{end}'
```

Resume a rolling update

The rollout is paused which means that some pods are at the new version and some pods are at the older version. We can continue the rollout using the resume command.

```
kubectl rollout resume deployment/hello
```

When the rollout is complete, you should see the following when running the status command.

```
kubectl rollout status deployment/hello
```

(Output)

```
deployment "hello" successfully rolled out
```

Rollback an update

Assume that a bug was detected in your new version. Since the new version is presumed to have problems, any users connected to the new Pods will experience those issues.

You will want to roll back to the previous version so you can investigate and then release a version that is fixed properly.

Use the rollout command to roll back to the previous version:

```
kubectl rollout undo deployment/hello
```

Verify the roll back in the history:

```
kubectl rollout history deployment/hello
```

Finally, verify that all the Pods have rolled back to their previous versions:

```
kubectl get pods -o jsonpath --template='{range
.items[*]}{.metadata.name}{"\t"}{"\t"}{.spec.containers[0].image}{"\n"}{end}'
```

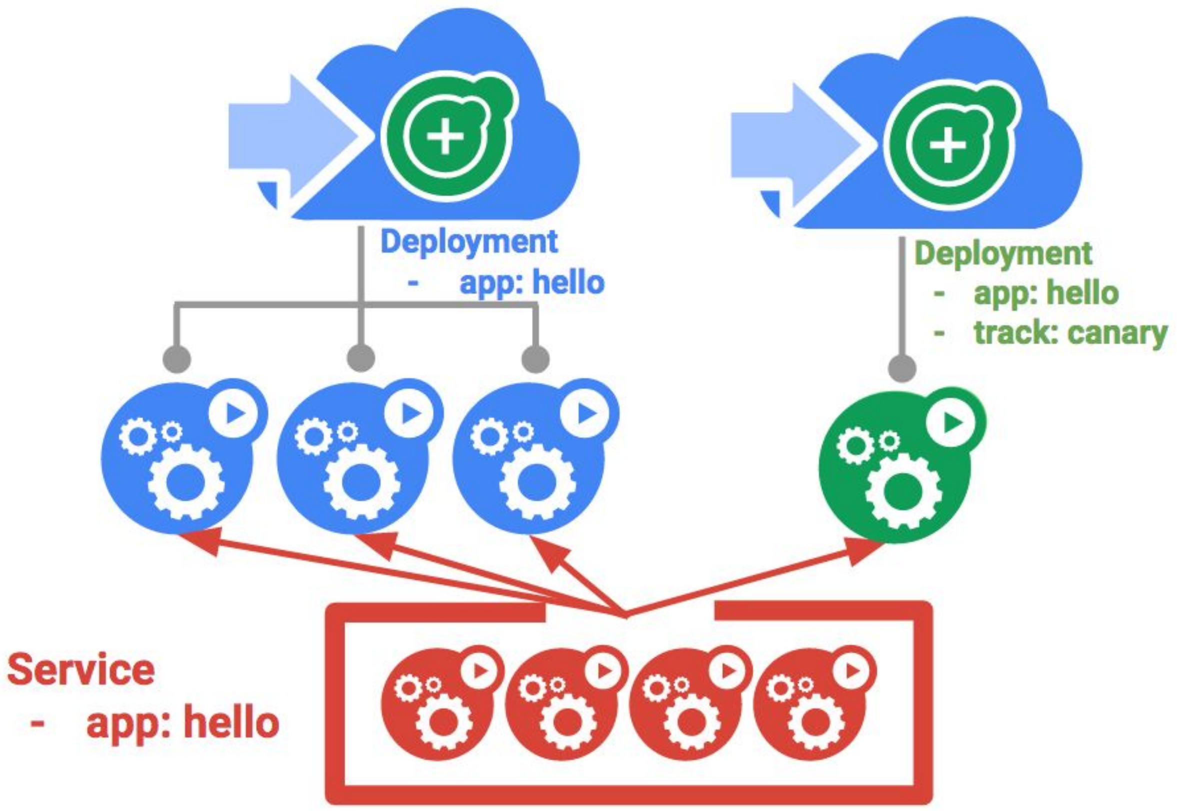
Great! You learned about rolling updates for Kubernetes deployments and how to update applications without downtime.

Canary deployments

When you want to test a new deployment in production with a subset of your users, use a canary deployment. Canary deployments allow you to release a change to a small subset of your users to mitigate risk associated with new releases.

Create a canary deployment

A canary deployment consists of a separate deployment with your new version and a service that targets both your normal, stable deployment as well as your canary deployment.



First, create a new canary deployment for the new version:

```
cat deployments/hello-canary.yaml
```

(Output)

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: hello-canary
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: hello
        track: canary
        # Use ver 1.0.0 so it matches version on service selector
        version: 1.0.0
    spec:
      containers:
        - name: hello
          image: kelseyhightower/hello:2.0.0
          ports:
            - name: http
              containerPort: 80
            - name: health
              containerPort: 81
...
```

Make sure to update version to 1.0.0 (if your version is pointing to any other)

Now create the canary deployment:

```
kubectl create -f deployments/hello-canary.yaml
```

After the canary deployment is created, you should have two deployments, hello and hello-canary. Verify it with this kubectl command:

```
kubectl get deployments
```

On the hello service, the selector uses the app:hello selector which will match pods in **both** the prod deployment and canary deployment. However, because the canary deployment has a fewer number of pods, it will be visible to fewer users.

Verify the canary deployment

You can verify the hello version being served by the request:

```
curl -ks https://`kubectl get svc frontend
-o=jsonpath="{.status.loadBalancer.ingress[0].ip}"`/version
```

Run this several times and you should see that some of the requests are served by hello 1.0.0 and a small subset (1/4 = 25%) are served by 2.0.0.

Test Completed Task

Click **Check my progress** below to check your lab progress. If you successfully created Canary deployment, you'll see an assessment score.

Canary deployments in production - session affinity

In this lab, each request sent to the Nginx service had a chance to be served by the canary deployment. But what if you wanted to ensure that a user didn't get served by the Canary deployment? A use case could be that the UI for an application changed, and you don't want to confuse the user. In a case like this, you want the user to "stick" to one deployment or the other.

You can do this by creating a service with session affinity. This way the same user will always be served from the same version. In the example below the service is the same as before, but a new sessionAffinity field has been added, and set to ClientIP. All clients with the same IP address will have their requests sent to the same version of the hello application.

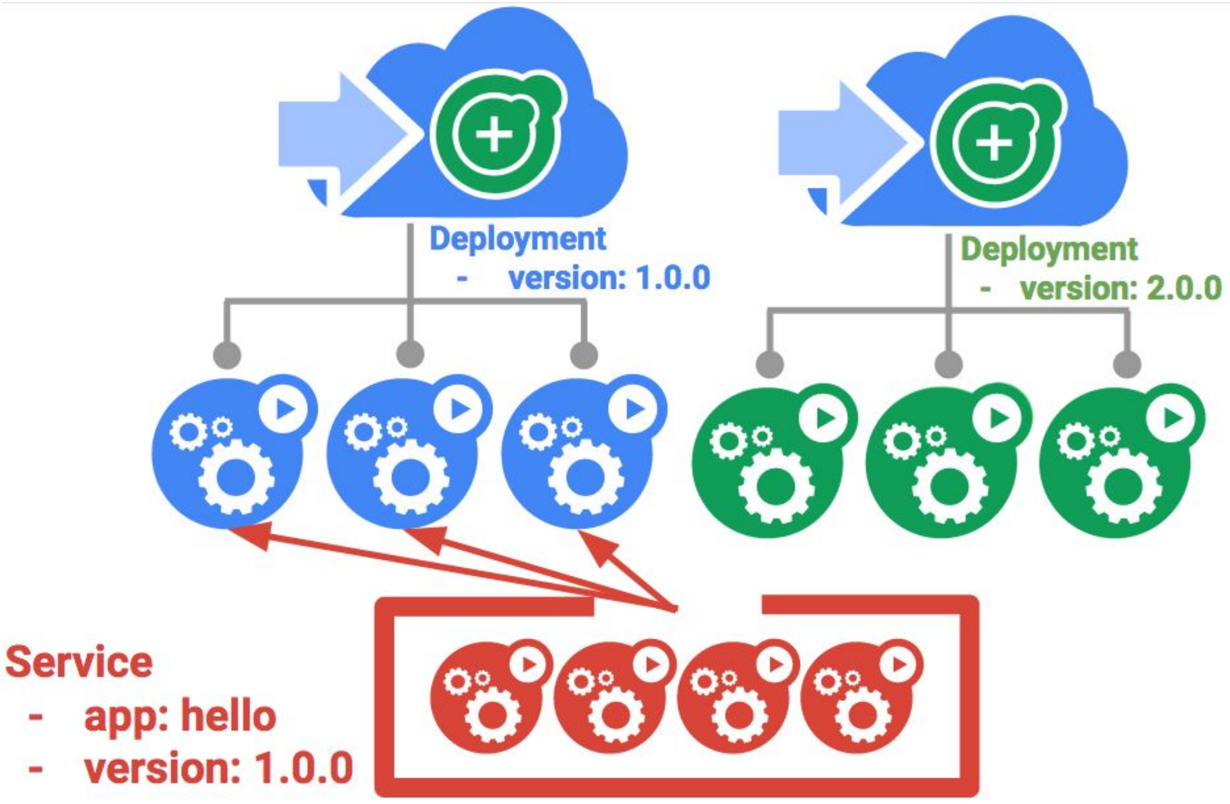
```
kind: Service
apiVersion: v1
metadata:
  name: "hello"
spec:
  sessionAffinity: ClientIP
  selector:
    app: "hello"
  ports:
    - protocol: "TCP"
      port: 80
      targetPort: 80
```

Due to it being difficult to set up an environment to test this, you don't need to here, but you may want to use sessionAffinity for canary deployments in production.

Blue-green deployments

Rolling updates are ideal because they allow you to deploy an application slowly with minimal overhead, minimal performance impact, and minimal downtime. There are instances where it is beneficial to modify the load balancers to point to that new version only after it has been fully deployed. In this case, blue-green deployments are the way to go.

Kubernetes achieves this by creating two separate deployments; one for the old "blue" version and one for the new "green" version. Use your existing hello deployment for the "blue" version. The deployments will be accessed via a Service which will act as the router. Once the new "green" version is up and running, you'll switch over to using that version by updating the Service.



A major downside of blue-green deployments is that you will need to have at least 2x the resources in your cluster necessary to host your application. Make sure you have enough resources in your cluster before deploying both versions of the application at once.

The service

Use the existing hello service, but update it so that it has a selector app:hello, version: 1.0.0. The selector will match the existing "blue" deployment. But it will not match the "green" deployment because it will use a different version.

First update the service:

```
kubectl apply -f services/hello-blue.yaml
```

Updating using Blue-Green Deployment

In order to support a blue-green deployment style, we will create a new "green" deployment for our new version. The green deployment updates the version label and the image path.

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: hello-green
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: hello
        track: stable
        version: 2.0.0
    spec:
      containers:
        - name: hello
          image: kelseyhightower/hello:2.0.0
          ports:
            - name: http
              containerPort: 80
            - name: health
              containerPort: 81
          resources:
            limits:
              cpu: 0.2
              memory: 10Mi
          livenessProbe:
            httpGet:
              path: /healthz
              port: 81
              scheme: HTTP
            initialDelaySeconds: 5
            periodSeconds: 15
            timeoutSeconds: 5
          readinessProbe:
            httpGet:
              path: /readiness
              port: 81
              scheme: HTTP
            initialDelaySeconds: 5
            timeoutSeconds: 1
```

Create the green deployment:

```
kubectl create -f deployments/hello-green.yaml
```

Once you have a green deployment and it has started up properly, verify that the current version of 1.0.0 is still being used:

```
curl -ks https://`kubectl get svc frontend
-o=jsonpath="{.status.loadBalancer.ingress[0].ip}"`/version
```

Now, update the service to point to the new version:

```
kubectl apply -f services/hello-green.yaml
```

With the service is updated, the "green" deployment will be used immediately. You can now verify that the new version is always being used.

```
curl -ks https://`kubectl get svc frontend
-o=jsonpath="{.status.loadBalancer.ingress[0].ip}"`/version
```

Blue-Green Rollback

If necessary, you can roll back to the old version in the same way. While the "blue" deployment is still running, just update the service back to the old version.

```
kubectl apply -f services/hello-blue.yaml
```

Once you have updated the service, your rollback will have been successful. Again, verify that the right version is now being used:

```
curl -ks https://`kubectl get svc frontend
-o=jsonpath="{.status.loadBalancer.ingress[0].ip}"`/version
```

You did it! You learned about blue-green deployments and how to deploy updates to applications that need to switch versions all at once.

Congratulations!

History
=============
```
        1  gcloud auth list
          2  gcloud config list project
          3  gcloud config set project gcloud config list project
          4  gcloud config set project  qwiklabs-gcp-00-ceb9b42244cb
          5  gcloud config set compute/zone us-central1-a
          6  git clone https://github.com/googlecodelabs/orchestrate-with-kubernetes.git
          7  cd orchestrate-with-kubernetes/kubernetes
          8  gcloud container clusters create bootcamp --num-nodes 5 --scopes
"https://www.googleapis.com/auth/projecthosting,storage-rw"
          9  clear
   10  kubectl explain deployment
   11  kubectl explain deployment --recursive
   12  kubectl explain deployment.metadata.name
   13  vi deployments/auth.yaml
   14  at deployments/auth.yaml
   15  cat deployments/auth.yaml
   16  kubectl create -f deployments/auth.yaml
   17  kubectl get deployments
   18  kubectl get pods
   19  kubectl get replicasets
   20  kubectl get pods
   21  kubectl create -f services/auth.yaml
   22  kubectl create -f deployments/hello.yaml
   23  kubectl create -f services/hello.yaml
   24  kubectl create secret generic tls-certs --from-file tls/
   25  kubectl create configmap nginx-frontend-conf --from-file=nginx/frontend.conf
   26  kubectl create -f deployments/frontend.yaml
   27  kubectl create -f services/frontend.yaml
   28  kubectl get services frontend
   29  curl -k https://35.239.204.38
   30  curl -ks https://`kubectl get svc frontend -o=jsonpath="{.status.loadBalancer.ingress[0].ip}"`
   31  kubectl get svc
   32  kubectl get svc frontend
   33  kubectl get svc frontend -o=jsonpath="{.status.loadbalancer.ingress[0]/ip}"
   34  kubectl get svc frontend -o=jsonpath="{.status.loadbalancer.ingress[0].ip}"
   35  https://kubectl get svc frontend -o=jsonpath="{.status.loadbalancer.ingress[0].ip}"
   36  kubectl get svc frontend -o=jsonpath="{.status.loadBalancer.ingress[0].ip}”
   37  kubectl get svc frontend -o=jsonpath="{.status.loadBalancer.ingress[0].ip}"
   38  kubectl explain deployment.spec.replicas
   39  kubectl explain deployments
   40  kubectl explain deployments --recursive
   41  kubectl explain deployments.spec
   42  kubectl explain deployments.spec.replicas
   43  kubectl get pods | grep hello- | wc -l
   44  kubect get pods
   45  kubectl get pods
   46  kubectl deployments hello --replicas=5
   47  kubectl scale deployments hello --replicas=5
   48  kubectl get pods
   49  kubectl get pods hello
   50*
   51  kubectl get pods | grep hello- | wc -l
   52  kubectl scale deployments hello --replicas=3
   53  kubectl get pods | grep hello- | wc -l
   54  kubectl edit deployments hello
   55  kubectl get rs
   56  kubectl rollout history deployment/hello
   57  kubectl rollout pause deployment/hello
   58  kubectl get rs
   59  kubectl rollout status deployment/hello
   60  kubectl get rs
   61  kubectl get pods -o jsonpath --template='{range
.items[*]}{.metadata.name}{"\t"}{"\t"}{.spec.containers[0].image}{"\n"}{end}'
   62  history
   63  kubectl rollout resume deployment/hello
   64  kubectl rollout status deployment/hello
   65  kubectl get pods -o jsonpath --template='{range
.items[*]}{.metadata.name}{"\t"}{"\t"}{.spec.containers[0].image}{"\n"}{end}'
   66  kubectl rollout undo deployment/hello
   67  kubectl get pods -o jsonpath --template='{range
.items[*]}{.metadata.name}{"\t"}{"\t"}{.spec.containers[0].image}{"\n"}{end}'
   68  kubectl rollout history deployment/hello
   69  kubectl get pods -o jsonpath --template='{range
.items[*]}{.metadata.name}{"\t"}{"\t"}{.spec.containers[0].image}{"\n"}{end}'
   70  cat deployments/hello-canary.yaml
71  kubectl create -f deployments/hello-canary.yaml
   72  kubectl get deployments
   73  curl -ks https://`kubectl get svc frontend
-o=jsonpath="{.status.loadBalancer.ingress[0].ip}"`/version
   74  kubectl get deployments
   75  curl -ks https://`kubectl get svc frontend
-o=jsonpath="{.status.loadBalancer.ingress[0].ip}"`/version
   76  kubectl apply -f services/hello-blue.yaml
   77  kubectl get svc
   78  kubectl create -f deployments/hello-green.yaml
   79  curl -ks https://`kubectl get svc frontend
-o=jsonpath="{.status.loadBalancer.ingress[0].ip}"`/version
   80  kubectl apply -f services/hello-green.yaml
   81  curl -ks https://`kubectl get svc frontend
-o=jsonpath="{.status.loadBalancer.ingress[0].ip}"`/version
   82  kubectl apply -f services/hello-blue.yaml
   83  curl -ks https://`kubectl get svc frontend
-o=jsonpath="{.status.loadBalancer.ingress[0].ip}"`/version
```