**Hello Minikube**

This tutorial shows you how to run a sample app on Kubernetes using minikube. The tutorial provides a container image that uses NGINX to echo back all the requests.

Objectives

* Deploy a sample application to minikube.
* Run the app.
* View application logs.

Before you begin

This tutorial assumes that you have already set up minikube. See **Step 1** in [minikube start](https://minikube.sigs.k8s.io/docs/start/) for installation instructions.

**Note:**

Only execute the instructions in **Step 1, Installation**. The rest is covered on this page.

You also need to install kubectl. See [Install tools](https://kubernetes.io/docs/tasks/tools/#kubectl) for installation instructions.

Create a minikube cluster

minikube start

Open the Dashboard

Open the Kubernetes dashboard. You can do this two different ways:

Open a **new** terminal, and run:

*# Start a new terminal, and leave this running.*

minikube dashboard

Now, switch back to the terminal where you ran minikube start.

**Note:**

The dashboard command enables the dashboard add-on and opens the proxy in the default web browser. You can create Kubernetes resources on the dashboard such as Deployment and Service.

To find out how to avoid directly invoking the browser from the terminal and get a URL for the web dashboard, see the "URL copy and paste" tab.

By default, the dashboard is only accessible from within the internal Kubernetes virtual network. The dashboard command creates a temporary proxy to make the dashboard accessible from outside the Kubernetes virtual network.

To stop the proxy, run Ctrl+C to exit the process. After the command exits, the dashboard remains running in the Kubernetes cluster. You can run the dashboard command again to create another proxy to access the dashboard.

Create a Deployment

A Kubernetes [*Pod*](https://kubernetes.io/docs/concepts/workloads/pods/) is a group of one or more Containers, tied together for the purposes of administration and networking. The Pod in this tutorial has only one Container. A Kubernetes [*Deployment*](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/) checks on the health of your Pod and restarts the Pod's Container if it terminates. Deployments are the recommended way to manage the creation and scaling of Pods.

1. Use the kubectl create command to create a Deployment that manages a Pod. The Pod runs a Container based on the provided Docker image.
2. *# Run a test container image that includes a webserver*
3. kubectl create deployment hello-node --image=registry.k8s.io/e2e-test-images/agnhost:2.39 -- /agnhost netexec --http-port=8080
4. View the Deployment:
5. kubectl get deployments

The output is similar to:

NAME READY UP-TO-DATE AVAILABLE AGE

hello-node 1/1 1 1 1m

(It may take some time for the pod to become available. If you see "0/1", try again in a few seconds.)

1. View the Pod:
2. kubectl get pods

The output is similar to:

NAME READY STATUS RESTARTS AGE

hello-node-5f76cf6ccf-br9b5 1/1 Running 0 1m

1. View cluster events:
2. kubectl get events
3. View the kubectl configuration:
4. kubectl config view
5. View application logs for a container in a pod (replace pod name with the one you got from kubectl get pods).

**Note:**

Replace hello-node-5f76cf6ccf-br9b5 in the kubectl logs command with the name of the pod from the kubectl get pods command output.

kubectl logs hello-node-5f76cf6ccf-br9b5

The output is similar to:

I0911 09:19:26.677397 1 log.go:195] Started HTTP server on port 8080

I0911 09:19:26.677586 1 log.go:195] Started UDP server on port 8081

**Note:**

For more information about kubectl commands, see the [kubectl overview](https://kubernetes.io/docs/reference/kubectl/).

Create a Service

By default, the Pod is only accessible by its internal IP address within the Kubernetes cluster. To make the hello-node Container accessible from outside the Kubernetes virtual network, you have to expose the Pod as a Kubernetes [*Service*](https://kubernetes.io/docs/concepts/services-networking/service/).

**Warning:**

The agnhost container has a /shell endpoint, which is useful for debugging, but dangerous to expose to the public internet. Do not run this on an internet-facing cluster, or a production cluster.

1. Expose the Pod to the public internet using the kubectl expose command:
2. kubectl expose deployment hello-node --type=LoadBalancer --port=8080

The --type=LoadBalancer flag indicates that you want to expose your Service outside of the cluster.

The application code inside the test image only listens on TCP port 8080. If you used kubectl expose to expose a different port, clients could not connect to that other port.

1. View the Service you created:
2. kubectl get services

The output is similar to:

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

hello-node LoadBalancer 10.108.144.78 <pending> 8080:30369/TCP 21s

kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 23m

On cloud providers that support load balancers, an external IP address would be provisioned to access the Service. On minikube, the LoadBalancer type makes the Service accessible through the minikube service command.

1. Run the following command:
2. minikube service hello-node

This opens up a browser window that serves your app and shows the app's response.

Enable addons

The minikube tool includes a set of built-in [addons](https://kubernetes.io/docs/concepts/cluster-administration/addons/" \o "" \t "_blank) that can be enabled, disabled and opened in the local Kubernetes environment.

1. List the currently supported addons:
2. minikube addons list

The output is similar to:

addon-manager: enabled

dashboard: enabled

default-storageclass: enabled

efk: disabled

freshpod: disabled

gvisor: disabled

helm-tiller: disabled

ingress: disabled

ingress-dns: disabled

logviewer: disabled

metrics-server: disabled

nvidia-driver-installer: disabled

nvidia-gpu-device-plugin: disabled

registry: disabled

registry-creds: disabled

storage-provisioner: enabled

storage-provisioner-gluster: disabled

1. Enable an addon, for example, metrics-server:
2. minikube addons enable metrics-server

The output is similar to:

The 'metrics-server' addon is enabled

1. View the Pod and Service you created by installing that addon:
2. kubectl get pod,svc -n kube-system

The output is similar to:

NAME READY STATUS RESTARTS AGE

pod/coredns-5644d7b6d9-mh9ll 1/1 Running 0 34m

pod/coredns-5644d7b6d9-pqd2t 1/1 Running 0 34m

pod/metrics-server-67fb648c5 1/1 Running 0 26s

pod/etcd-minikube 1/1 Running 0 34m

pod/influxdb-grafana-b29w8 2/2 Running 0 26s

pod/kube-addon-manager-minikube 1/1 Running 0 34m

pod/kube-apiserver-minikube 1/1 Running 0 34m

pod/kube-controller-manager-minikube 1/1 Running 0 34m

pod/kube-proxy-rnlps 1/1 Running 0 34m

pod/kube-scheduler-minikube 1/1 Running 0 34m

pod/storage-provisioner 1/1 Running 0 34m

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

service/metrics-server ClusterIP 10.96.241.45 <none> 80/TCP 26s

service/kube-dns ClusterIP 10.96.0.10 <none> 53/UDP,53/TCP 34m

service/monitoring-grafana NodePort 10.99.24.54 <none> 80:30002/TCP 26s

service/monitoring-influxdb ClusterIP 10.111.169.94 <none> 8083/TCP,8086/TCP 26s

1. Check the output from metrics-server:
2. kubectl top pods

The output is similar to:

NAME CPU(cores) MEMORY(bytes)

hello-node-ccf4b9788-4jn97 1m 6Mi

If you see the following message, wait, and try again:

error: Metrics API not available

1. Disable metrics-server:
2. minikube addons disable metrics-server

The output is similar to:

metrics-server was successfully disabled

Clean up

Now you can clean up the resources you created in your cluster:

kubectl delete service hello-node

kubectl delete deployment hello-node

Stop the Minikube cluster

minikube stop

Optionally, delete the Minikube VM:

*# Optional*

minikube delete

If you want to use minikube again to learn more about Kubernetes, you don't need to delete it.

# Using Minikube to Create a Cluster

Learn what a Kubernetes cluster is. Learn what Minikube is. Start a Kubernetes cluster.

### **Objectives**

* Learn what a Kubernetes cluster is.
* Learn what Minikube is.
* Start a Kubernetes cluster on your computer.

### **Kubernetes Clusters**

**Kubernetes coordinates a highly available cluster of computers that are connected to work as a single unit.** The abstractions in Kubernetes allow you to deploy containerized applications to a cluster without tying them specifically to individual machines. To make use of this new model of deployment, applications need to be packaged in a way that decouples them from individual hosts: they need to be containerized. Containerized applications are more flexible and available than in past deployment models, where applications were installed directly onto specific machines as packages deeply integrated into the host. **Kubernetes automates the distribution and scheduling of application containers across a cluster in a more efficient way.** Kubernetes is an open-source platform and is production-ready.

A Kubernetes cluster consists of two types of resources:

* The **Control Plane** coordinates the cluster
* **Nodes** are the workers that run applications

## Cluster Diagram

**The Control Plane is responsible for managing the cluster.** The Control Plane coordinates all activities in your cluster, such as scheduling applications, maintaining applications' desired state, scaling applications, and rolling out new updates.

**A node is a VM or a physical computer that serves as a worker machine in a Kubernetes cluster.** Each node has a Kubelet, which is an agent for managing the node and communicating with the Kubernetes control plane. The node should also have tools for handling container operations, such as [containerd](https://containerd.io/docs/" \o "" \t "_blank) or [CRI-O](https://cri-o.io/#what-is-cri-o). A Kubernetes cluster that handles production traffic should have a minimum of three nodes because if one node goes down, both an [etcd](https://kubernetes.io/docs/concepts/architecture/" \l "etcd) member and a control plane instance are lost, and redundancy is compromised. You can mitigate this risk by adding more control plane nodes.

When you deploy applications on Kubernetes, you tell the control plane to start the application containers. The control plane schedules the containers to run on the cluster's nodes. **Node-level components, such as the kubelet, communicate with the control plane using the**[**Kubernetes API**](https://kubernetes.io/docs/concepts/overview/kubernetes-api/), which the control plane exposes. End users can also use the Kubernetes API directly to interact with the cluster.

A Kubernetes cluster can be deployed on either physical or virtual machines. To get started with Kubernetes development, you can use Minikube. Minikube is a lightweight Kubernetes implementation that creates a VM on your local machine and deploys a simple cluster containing only one node. Minikube is available for Linux, macOS, and Windows systems. The Minikube CLI provides basic bootstrapping operations for working with your cluster, including start, stop, status, and delete.

# Using kubectl to Create a Deployment

Learn about application Deployments. Deploy your first app on Kubernetes with kubectl.

### **Objectives**

* Learn about application Deployments.
* Deploy your first app on Kubernetes with kubectl.

### **Kubernetes Deployments**

#### Note:

This tutorial uses a container that requires the AMD64 architecture. If you are using minikube on a computer with a different CPU architecture, you could try using minikube with a driver that can emulate AMD64. For example, the Docker Desktop driver can do this.

Once you have a [running Kubernetes cluster](https://kubernetes.io/docs/tutorials/kubernetes-basics/create-cluster/cluster-intro/), you can deploy your containerized applications on top of it. To do so, you create a Kubernetes **Deployment**. The Deployment instructs Kubernetes how to create and update instances of your application. Once you've created a Deployment, the Kubernetes control plane schedules the application instances included in that Deployment to run on individual Nodes in the cluster.

Once the application instances are created, a Kubernetes Deployment controller continuously monitors those instances. If the Node hosting an instance goes down or is deleted, the Deployment controller replaces the instance with an instance on another Node in the cluster. **This provides a self-healing mechanism to address machine failure or maintenance.**

In a pre-orchestration world, installation scripts would often be used to start applications, but they did not allow recovery from machine failure. By both creating your application instances and keeping them running across Nodes, Kubernetes Deployments provide a fundamentally different approach to application management.

### **Summary:**

* Deployments
* Kubectl

## Deploying your first app on Kubernetes

## You can create and manage a Deployment by using the Kubernetes command line interface, kubectl. Kubectl uses the Kubernetes API to interact with the cluster. In this module, you'll learn the most common kubectl commands needed to create Deployments that run your applications on a Kubernetes cluster.

When you create a Deployment, you'll need to specify the container image for your application and the number of replicas that you want to run. You can change that information later by updating your Deployment; Modules [5](https://kubernetes.io/docs/tutorials/kubernetes-basics/scale/scale-intro/) and [6](https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro/) of the bootcamp discuss how you can scale and update your Deployments.

For your first Deployment, you'll use a hello-node application packaged in a Docker container that uses NGINX to echo back all the requests. (If you didn't already try creating a hello-node application and deploying it using a container, you can do that first by following the instructions from the [Hello Minikube tutorial](https://kubernetes.io/docs/tutorials/hello-minikube/)).

You will need to have installed kubectl as well. If you need to install it, visit [install tools](https://kubernetes.io/docs/tasks/tools/#kubectl).

Now that you know what Deployments are, let's deploy our first app!

### **kubectl basics**

This performs the specified action (like create, describe or delete) on the specified resource (like node or deployment). You can use --help after the subcommand to get additional info about possible parameters (for example: kubectl get nodes --help).

Check that kubectl is configured to talk to your cluster, by running the **kubectl version** command.

Check that kubectl is installed and you can see both the client and the server versions.

To view the nodes in the cluster, run the **kubectl get nodes** command.

You see the available nodes. Later, Kubernetes will choose where to deploy our application based on Node available resources.

### **Deploy an app**

Let’s deploy our first app on Kubernetes with the kubectl create deployment command. We need to provide the deployment name and app image location (include the full repository url for images hosted outside Docker Hub).

**kubectl create deployment kubernetes-bootcamp --image=gcr.io/google-samples/kubernetes-bootcamp:v1**

Great! You just deployed your first application by creating a deployment. This performed a few things for you:

* searched for a suitable node where an instance of the application could be run (we have only 1 available node)
* scheduled the application to run on that Node
* configured the cluster to reschedule the instance on a new Node when needed

To list your deployments use the kubectl get deployments command:

**kubectl get deployments**

We see that there is 1 deployment running a single instance of your app. The instance is running inside a container on your node.

### **View the app**

[Pods](https://kubernetes.io/docs/concepts/workloads/pods/) that are running inside Kubernetes are running on a private, isolated network. By default they are visible from other pods and services within the same Kubernetes cluster, but not outside that network. When we use kubectl, we're interacting through an API endpoint to communicate with our application. We will cover other options on how to expose your application outside the Kubernetes cluster later, in [Module 4](https://kubernetes.io/docs/tutorials/kubernetes-basics/expose/). Also as a basic tutorial, we're not explaining what Pods are in any detail here, it will be covered in later topics. The kubectl proxy command can create a proxy that will forward communications into the cluster-wide, private network. The proxy can be terminated by pressing control-C and won't show any output while it's running.

**You need to open a second terminal window to run the proxy.**

**kubectl proxy**.

You can see all those APIs hosted through the proxy endpoint. For example, we can query the version directly through the API using the curl command:

**curl http://localhost:8001/version**

The API server will automatically create an endpoint for each pod, based on the pod name, that is also accessible through the proxy.

First we need to get the Pod name, and we'll store it in the environment variable POD\_NAME:

**export POD\_NAME=$(kubectl get pods -o go-template --template '{{range .items}}{{.metadata.name}}{{"\n"}}{{end}}')**  
**echo Name of the Pod: $POD\_NAME**

You can access the Pod through the proxied API, by running:

**curl** [**http://localhost:8001/api/v1/namespaces/default/pods/$POD\_NAME:8080/proxy/**](http://localhost:8001/api/v1/namespaces/default/pods/$POD_NAME:8080/proxy/)

# Viewing Pods and Nodes

## Kubernetes Pods

When you created a Deployment in Module [2](https://kubernetes.io/docs/tutorials/kubernetes-basics/deploy-app/deploy-intro/), Kubernetes created a **Pod** to host your application instance. A Pod is a Kubernetes abstraction that represents a group of one or more application containers (such as Docker), and some shared resources for those containers. Those resources include:

* Shared storage, as Volumes
* Networking, as a unique cluster IP address
* Information about how to run each container, such as the container image version or specific ports to use

A Pod models an application-specific "logical host" and can contain different application containers which are relatively tightly coupled. For example, a Pod might include both the container with your Node.js app as well as a different container that feeds the data to be published by the Node.js webserver. The containers in a Pod share an IP Address and port space, are always co-located and co-scheduled, and run in a shared context on the same Node.

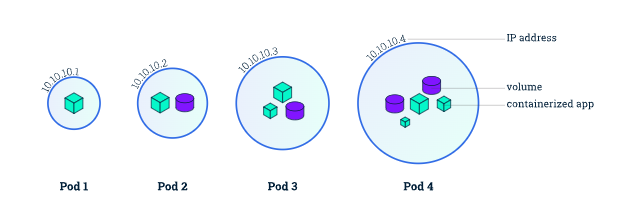
Pods are the atomic unit on the Kubernetes platform. When we create a Deployment on Kubernetes, that Deployment creates Pods with containers inside them (as opposed to creating containers directly). Each Pod is tied to the Node where it is scheduled, and remains there until termination (according to restart policy) or deletion. In case of a Node failure, identical Pods are scheduled on other available Nodes in the cluster.

**Pods overview**

**Nodes**

A Pod always runs on a **Node**. A Node is a worker machine in Kubernetes and may be either a virtual or a physical machine, depending on the cluster. Each Node is managed by the control plane. A Node can have multiple pods, and the Kubernetes control plane automatically handles scheduling the pods across the Nodes in the cluster. The control plane's automatic scheduling takes into account the available resources on each Node.

Every Kubernetes Node runs at least:

* Kubelet, a process responsible for communication between the Kubernetes control plane and the Node; it manages the Pods and the containers running on a machine.
* A container runtime (like Docker) responsible for pulling the container image from a registry, unpacking the container, and running the application.

## Node overview

## Troubleshooting with kubectl

In Module [2](https://kubernetes.io/docs/tutorials/kubernetes-basics/deploy-app/deploy-intro/), you used the kubectl command-line interface. You'll continue to use it in Module 3 to get information about deployed applications and their environments. The most common operations can be done with the following kubectl subcommands:

* **kubectl get** - list resources
* **kubectl describe** - show detailed information about a resource
* **kubectl logs** - print the logs from a container in a pod
* **kubectl exec** - execute a command on a container in a pod

You can use these commands to see when applications were deployed, what their current statuses are, where they are running and what their configurations are.

Now that we know more about our cluster components and the command line, let's explore our application.

### **Check application configuration**

Let's verify that the application we deployed in the previous scenario is running. We'll use the kubectl get command and look for existing Pods:

**kubectl get pods**

If no pods are running, please wait a couple of seconds and list the Pods again. You can continue once you see one Pod running.

Next, to view what containers are inside that Pod and what images are used to build those containers we run the kubectl describe pods command:

**kubectl describe pods**

We see here details about the Pod’s container: IP address, the ports used and a list of events related to the lifecycle of the Pod.

The output of the describe subcommand is extensive and covers some concepts that we didn’t explain yet, but don’t worry, they will become familiar by the end of this bootcamp.

***Note:*** the *describe* subcommand can be used to get detailed information about most of the Kubernetes primitives, including Nodes, Pods, and Deployments. The describe output is designed to be human readable, not to be scripted against.

### **Show the app in the terminal**

Recall that Pods are running in an isolated, private network - so we need to proxy access to them so we can debug and interact with them. To do this, we'll use the kubectl proxy command to run a proxy in a **second terminal**. Open a new terminal window, and in that new terminal, run:

**kubectl proxy**

Now again, we'll get the Pod name and query that pod directly through the proxy. To get the Pod name and store it in the POD\_NAME environment variable:

**export POD\_NAME="$(kubectl get pods -o go-template --template '{{range .items}}{{.metadata.name}}{{"\n"}}{{end}}')"**  
**echo Name of the Pod: $POD\_NAME**

To see the output of our application, run a curl request:

**curl http://localhost:8001/api/v1/namespaces/default/pods/$POD\_NAME:8080/proxy/**

The URL is the route to the API of the Pod.

### **View the container logs**

Anything that the application would normally send to standard output becomes logs for the container within the Pod. We can retrieve these logs using the kubectl logs command:

**kubectl logs "$POD\_NAME"**

***Note:*** We don't need to specify the container name, because we only have one container inside the pod.

### **Executing command on the container**

We can execute commands directly on the container once the Pod is up and running. For this, we use the exec subcommand and use the name of the Pod as a parameter. Let’s list the environment variables:

**kubectl exec "$POD\_NAME" -- env**

Again, it's worth mentioning that the name of the container itself can be omitted since we only have a single container in the Pod.

Next let’s start a bash session in the Pod’s container:

**kubectl exec -ti $POD\_NAME -- bash**

We have now an open console on the container where we run our NodeJS application. The source code of the app is in the server.js file:

**cat server.js**

You can check that the application is up by running a curl command:

**curl http://localhost:8080**

# Using a Service to Expose Your App

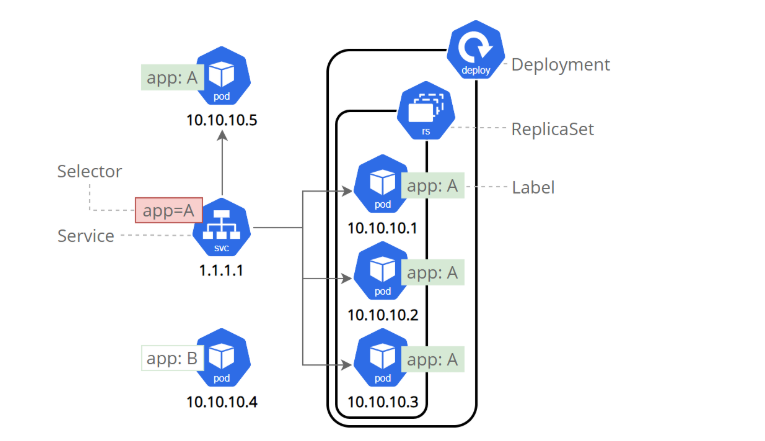
Although each Pod has a unique IP address, those IPs are not exposed outside the cluster without a Service. Services allow your applications to receive traffic. Services can be exposed in different ways by specifying a type in the spec of the Service:

* *ClusterIP* (default) - Exposes the Service on an internal IP in the cluster. This type makes the Service only reachable from within the cluster.
* *NodePort* - Exposes the Service on the same port of each selected Node in the cluster using NAT. Makes a Service accessible from outside the cluster using <NodeIP>:<NodePort>. Superset of ClusterIP.
* *LoadBalancer* - Creates an external load balancer in the current cloud (if supported) and assigns a fixed, external IP to the Service. Superset of NodePort.
* *ExternalName* - Maps the Service to the contents of the externalName field (e.g. foo.bar.example.com), by returning a CNAME record with its value. No proxying of any kind is set up. This type requires v1.7 or higher of kube-dns, or CoreDNS version 0.0.8 or higher.

### **Services and Labels**

A Service routes traffic across a set of Pods. Services are the abstraction that allows pods to die and replicate in Kubernetes without impacting your application. Discovery and routing among dependent Pods (such as the frontend and backend components in an application) are handled by Kubernetes Services.

Services match a set of Pods using [labels and selectors](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels), a grouping primitive that allows logical operation on objects in Kubernetes. Labels are key/value pairs attached to objects and can be used in any number of ways:

* Designate objects for development, test, and production
* Embed version tags
* Classify an object using tags

Labels can be attached to objects at creation time or later on. They can be modified at any time. Let's expose our application now using a Service and apply some labels.

### **Step 1: Creating a new Service**

Let’s verify that our application is running. We’ll use the kubectl get command and look for existing Pods:

**kubectl get pods**

If no Pods are running then it means the objects from the previous tutorials were cleaned up. In this case, go back and recreate the deployment from the [Using kubectl to create a Deployment](https://kubernetes.io/docs/tutorials/kubernetes-basics/deploy-app/deploy-intro#deploy-an-app) tutorial. Please wait a couple of seconds and list the Pods again. You can continue once you see the one Pod running.

Next, let’s list the current Services from our cluster:

**kubectl get services**

We have a Service called kubernetes that is created by default when minikube starts the cluster. To create a new service and expose it to external traffic we'll use the expose command with NodePort as parameter.

**kubectl expose deployment/kubernetes-bootcamp --type="NodePort" --port 8080**

**kubectl get services**

We have now a running Service called kubernetes-bootcamp. Here we see that the Service received a unique cluster-IP, an internal port and an external-IP (the IP of the Node).

To find out what port was opened externally (for the type: NodePort Service) we’ll run the describe service subcommand:

**kubectl describe services/kubernetes-bootcamp**

Create an environment variable called NODE\_PORT that has the value of the Node port assigned:

**export NODE\_PORT="$(kubectl get services/kubernetes-bootcamp -o go-template='{{(index .spec.ports 0).nodePort}}')"**  
**echo "NODE\_PORT=$NODE\_PORT"**

Now we can test that the app is exposed outside of the cluster using curl, the IP address of the Node and the externally exposed port:

**curl http://"$(minikube ip):$NODE\_PORT"**

In a separate terminal window, execute:  
**minikube service kubernetes-bootcamp --url**

The output looks like this:

**http://127.0.0.1:51082  
! Because you are using a Docker driver on darwin, the terminal needs to be open to run it.**

Then use the given URL to access the app:  
**curl 127.0.0.1:51082**

And we get a response from the server. The Service is exposed.

### **Step 2: Using labels**

The Deployment created automatically a label for our Pod. With the describe deployment subcommand you can see the name (the key) of that label:

**kubectl describe deployment**

Let’s use this label to query our list of Pods. We’ll use the kubectl get pods command with -l as a parameter, followed by the label values:

**kubectl get pods -l app=kubernetes-bootcamp**

You can do the same to list the existing Services:

**kubectl get services -l app=kubernetes-bootcamp**

Get the name of the Pod and store it in the POD\_NAME environment variable:

**export POD\_NAME="$(kubectl get pods -o go-template --template '{{range .items}}{{.metadata.name}}{{"\n"}}{{end}}')"**  
**echo "Name of the Pod: $POD\_NAME"**

To apply a new label we use the label subcommand followed by the object type, object name and the new label:

**kubectl label pods "$POD\_NAME" version=v1**

This will apply a new label to our Pod (we pinned the application version to the Pod), and we can check it with the describe pod command:

**kubectl describe pods "$POD\_NAME"**

We see here that the label is attached now to our Pod. And we can query now the list of pods using the new label:

**kubectl get pods -l version=v1**

And we see the Pod.

### **Step 3: Deleting a service**

To delete Services you can use the delete service subcommand. Labels can be used also here:

**kubectl delete service -l app=kubernetes-bootcamp**

Confirm that the Service is gone:

**kubectl get services**

This confirms that our Service was removed. To confirm that route is not exposed anymore you can curl the previously exposed IP and port:

**curl http://"$(minikube ip):$NODE\_PORT"**

This proves that the application is not reachable anymore from outside of the cluster. You can confirm that the app is still running with a curl from inside the pod:

**kubectl exec -ti $POD\_NAME -- curl http://localhost:8080**

### **Kubernetes Scaling & Load Balancing Cheatsheet:**

#### ****1. Check Existing Deployments****

kubectl get deployments

* Lists all deployments with their current state.

#### ****2. Check ReplicaSets****

kubectl get rs

* Shows ReplicaSets, with desired and current replicas.

#### ****3. Expose Deployment (LoadBalancer Service)****

kubectl expose deployment/kubernetes-bootcamp --type="LoadBalancer" --port=8080

* Creates a LoadBalancer service for external access.

#### ****4. Scale Up Deployment****

kubectl scale deployments/kubernetes-bootcamp --replicas=4

* Increases replicas to 4 for scaling up.

#### ****5. Verify Scaling Changes****

kubectl get deployments

kubectl get pods -o wide

kubectl describe deployments/kubernetes-bootcamp

* Confirms the number of replicas and checks deployment events.

#### ****6. Check Load Balancer Details****

kubectl describe services/kubernetes-bootcamp

* Shows service details, including load-balancer IP and ports.

#### ****7. Test Load Balancing****

export NODE\_PORT="$(kubectl get services/kubernetes-bootcamp -o go-template='{{(index .spec.ports 0).nodePort}}')"

echo NODE\_PORT=$NODE\_PORT

curl http://"$(minikube ip):$NODE\_PORT"

* Sends multiple requests to check if traffic is distributed among pods.

#### ****8. Minikube Tunnel (If Needed)****

minikube service kubernetes-bootcamp --url

curl 127.0.0.1:<port>

* Used if running Minikube with Docker Desktop.

#### ****9. Scale Down Deployment****

kubectl scale deployments/kubernetes-bootcamp --replicas=2

kubectl get deployments

kubectl get pods -o wide

* Decreases replicas and verifies changes.

# Performing a Rolling Update

Perform a rolling update using kubectl.

### **Objectives**

* Perform a rolling update using kubectl.

### **Updating an application**

Users expect applications to be available all the time, and developers are expected to deploy new versions of them several times a day. In Kubernetes this is done with rolling updates. A **rolling update** allows a Deployment update to take place with zero downtime. It does this by incrementally replacing the current Pods with new ones. The new Pods are scheduled on Nodes with available resources, and Kubernetes waits for those new Pods to start before removing the old Pods.

## Rolling updates overview

## 

Similar to application Scaling, if a Deployment is exposed publicly, the Service will load-balance the traffic only to available Pods during the update. An available Pod is an instance that is available to the users of the application.

Rolling updates allow the following actions:

* Promote an application from one environment to another (via container image updates)
* Rollback to previous versions
* Continuous Integration and Continuous Delivery of applications with zero downtime

In the following interactive tutorial, we'll update our application to a new version, and also perform a rollback.

### **Update the version of the app**

To list your Deployments, run the get deployments subcommand: **kubectl get deployments**

To list the running Pods, run the get pods subcommand:

**kubectl get pods**

To view the current image version of the app, run the describe pods subcommand and look for the Image field:

**kubectl describe pods**

To update the image of the application to version 2, use the set image subcommand, followed by the deployment name and the new image version:

**kubectl set image deployments/kubernetes-bootcamp kubernetes-bootcamp=docker.io/jocatalin/kubernetes-bootcamp:v2**

The command notified the Deployment to use a different image for your app and initiated a rolling update. Check the status of the new Pods, and view the old one terminating with the get pods subcommand:

**kubectl get pods**

### **Verify an update**

First, check that the service is running, as you might have deleted it in previous tutorial step, run describe services/kubernetes-bootcamp. If it's missing, you can create it again with:

**kubectl expose deployment/kubernetes-bootcamp --type="NodePort" --port 8080**

Create an environment variable called NODE\_PORT that has the value of the Node port assigned:

**export NODE\_PORT="$(kubectl get services/kubernetes-bootcamp -o go-template='{{(index .spec.ports 0).nodePort}}')"**  
**echo "NODE\_PORT=$NODE\_PORT"**

Next, do a curl to the exposed IP and port:

**curl http://"$(minikube ip):$NODE\_PORT"**

Every time you run the curl command, you will hit a different Pod. Notice that all Pods are now running the latest version (v2).

You can also confirm the update by running the rollout status subcommand:

**kubectl rollout status deployments/kubernetes-bootcamp**

To view the current image version of the app, run the describe pods subcommand:

**kubectl describe pods**

In the Image field of the output, verify that you are running the latest image version (v2).

### **Roll back an update**

Let’s perform another update, and try to deploy an image tagged with v10:

**kubectl set image deployments/kubernetes-bootcamp kubernetes-bootcamp=gcr.io/google-samples/kubernetes-bootcamp:v10**

Use get deployments to see the status of the deployment:

**kubectl get deployments**

Notice that the output doesn't list the desired number of available Pods. Run the get pods subcommand to list all Pods:

**kubectl get pods**

Notice that some of the Pods have a status of ImagePullBackOff.

To get more insight into the problem, run the describe pods subcommand:

**kubectl describe pods**

In the Events section of the output for the affected Pods, notice that the v10 image version did not exist in the repository.

To roll back the deployment to your last working version, use the rollout undo subcommand:

**kubectl rollout undo deployments/kubernetes-bootcamp**

The rollout undo command reverts the deployment to the previous known state (v2 of the image). Updates are versioned and you can revert to any previously known state of a Deployment.

Use the get pods subcommand to list the Pods again:

**kubectl get pods**

To check the image deployed on the running Pods, use the describe pods subcommand:

**kubectl describe pods**

The Deployment is once again using a stable version of the app (v2). The rollback was successful.

Remember to clean up your local cluster

**kubectl delete deployments/kubernetes-bootcamp services/kubernetes-bootcamp**