Kubernetes

* Kubernetes is a portable, extensible open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation.

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
* **Kubernetes automates the distribution and scheduling of application containers across a cluster in a more efficient way.**
* A Kubernetes cluster consists of two types of resources:

The **Master** coordinates the cluster

Nodes are workers that run application

* **The Master is responsible for managing the cluster.** The master 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 master. The node should also have tools for handling container operations, such as [Docker](https://www.docker.com/) or [rkt](https://coreos.com/rkt/). A Kubernetes cluster that handles production traffic should have a minimum of three nodes.
* When you deploy applications on Kubernetes, you tell the master to start the application containers. The master schedules the containers to run on the cluster's nodes.
* **The nodes communicate with the master using the Kubernetes API**, which the master 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.
* 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.

MiniKube Commands

(Minikube is a lightweight Kubernetes implementation that creates a VM on your local machine and deploys a simple cluster containing only one node)

* minikube version
* minikybe start

Kubectl Commands

( To interact with Kubernetes during this bootcamp we’ll use the command line interface, kubectl)

* kubectl version
* kubectl cluster-info
* kubectl get nodes
* kubectl –help
* kubectl action resource
* kubectl run *deployment\_name app\_image\_location port*
* kubectl get deployments
* kubectl proxy
* export POD\_NAME=$(kubectl get pods -o go-template --template '{{range .items}}{{.metadata.name}}{{"\n"}}{{end}}')
* echo Name of the Pod: $POD\_NAME
* **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
* kubectl get pods
* kubectl describe pods

Kubectl Proxy Commands

* export POD\_NAME=$(kubectl get pods -o go-template --template '{{range .items}}{{.metadata.name}}{{"\n"}}{{end}}') echo Name of the Pod: $POD\_NAME
* curl http://localhost:8001/api/v1/namespaces/default/pods/$POD\_NAME/proxy/

# **Using kubectl to Create a Deployment**

* Once you have a running Kubernetes cluster, you can deploy your containerized applications on top of it.
* To do so, you create a Kubernetes **Deployment** configuration. The Deployment instructs Kubernetes how to create and update instances of your application.
* Once you've created a Deployment, the Kubernetes master schedules mentioned application instances onto 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 it.
* **This provides a self-healing mechanism to address machine failure or maintenance.**
* 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.
* 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.
* The run command creates a new deployment.
* We need to provide the deployment name and app image location (include the full repository url for images hosted outside Docker hub).
* The kubectl 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 its running.
* The proxy enables direct access to the API from these terminals

## **Kubernetes Pods**

* *A Pod is a group of one or more application containers (such as Docker or rkt) and includes shared storage (volumes), IP address and information about how to run them.*
* When you created a Deployment, 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 or rkt), 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.
* *Containers should only be scheduled together in a single Pod if they are tightly coupled and need to share resources such as disk.*

## **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 Master. A Node can have multiple pods, and the Kubernetes master automatically handles scheduling the pods across the Nodes in the cluster. The Master'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 Master and the Node; it manages the Pods and the containers running on a machine.
* A container runtime (like Docker, rkt) responsible for pulling the container image from a registry, unpacking the container, and running the application.

*Containers should only be scheduled together in a single Pod if they are tightly coupled and need to share resources such as disk.*

#### Step 1 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, list the Pods again.

Next, to view what containers are inside that Pod and what images are used to build those containers we run the 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 command 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 command can be used to get detailed information about most of the kubernetes primitives: node, pods, deployments. The describe output is designed to be human readable, not to be scripted against.

#### Step 2 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 window. Click on the command below to automatically open a new terminal and run the proxy:

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/proxy/

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

#### Step 3 View the container logs

Anything that the application would normally send to STDOUT 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.

#### Step 4 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 command and use the name of the Pod as a parameter. Let’s list the environment variables:

kubectl exec $POD\_NAME env

Again, 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 localhost:8080

Note: here we used localhost because we executed the command inside the NodeJS container

To close your container connection type exit.

### Overview of Kubernetes Services

* Kubernetes [Pods](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/) are mortal. Pods in fact have a [lifecycle](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/). When a worker node dies, the Pods running on the Node are also lost.
* A [ReplicationController](https://kubernetes.io/docs/user-guide/replication-controller/" \l "what-is-a-replicationcontroller) might then dynamically drive the cluster back to desired state via creation of new Pods to keep your application running. As another example, consider an image-processing backend with 3 replicas. Those replicas are fungible; the front-end system should not care about backend replicas or even if a Pod is lost and recreated. That said, each Pod in a Kubernetes cluster has a unique IP address, even Pods on the same Node, so there needs to be a way of automatically reconciling changes among Pods so that your applications continue to function.
* A Service in Kubernetes is an abstraction which defines a logical set of Pods and a policy by which to access them.
* Services enable a loose coupling between dependent Pods. A Service is defined using YAML [(preferred)](https://kubernetes.io/docs/concepts/configuration/overview/#general-config-tips)or JSON, like all Kubernetes objects. The set of Pods targeted by a Service is usually determined by a *LabelSelector* (see below for why you might want a Service without including **selector** in the spec).
* 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 ServiceSpec:
* *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* - Exposes the Service using an arbitrary name (specified by **externalName** in the spec) by returning a CNAME record with the name. No proxy is used. This type requires v1.7 or higher of **kube-dns**.
* More information about the different types of Services can be found in the [Using Source IP](https://kubernetes.io/docs/tutorials/services/source-ip/) tutorial. Also see [Connecting Applications with Services](https://kubernetes.io/docs/concepts/services-networking/connect-applications-service).
* Additionally, note that there are some use cases with Services that involve not defining **selector** in the spec. A Service created without **selector** will also not create the corresponding Endpoints object. This allows users to manually map a Service to specific endpoints. Another possibility why there may be no selector is you are strictly using type: ExternalName

#### Step 1 Create a new service

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

kubectl get pods

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 (minikube does not support the LoadBalancer option yet).

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

Let’s run again the get services command:

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 (by the NodePort option) we’ll run the describe service command:

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 of the Node and the externally exposed port:

curl $(minikube ip):$NODE\_PORT

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 describe deployment command you can see the name of the 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 run=kubernetes-bootcamp

You can do the same to list the existing services:

kubectl get services -l run=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 command followed by the object type, object name and the new label:

kubectl label pod $POD\_NAME app=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 app=v1

And we see the Pod.

#### Step 3 Deleting a service

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

kubectl delete service -l run=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 $(minikube ip):$NODE\_PORT

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

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

We see here that the application is up.

### Scaling an application

In the previous modules we created a [Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/), and then exposed it publicly via a [Service](https://kubernetes.io/docs/concepts/services-networking/service/). The Deployment created only one Pod for running our application. When traffic increases, we will need to scale the application to keep up with user demand.

**Scaling** is accomplished by changing the number of replicas in a Deployment

### Summary:

* Scaling a Deployment

*You can create from the start a Deployment with multiple instances using the --replicas parameter for the kubectl run command*

## **Scaling overview**

[PreviousNext](https://kubernetes.io/docs/tutorials/kubernetes-basics/scale/scale-intro/#myCarousel)

Scaling out a Deployment will ensure new Pods are created and scheduled to Nodes with available resources. Scaling will increase the number of Pods to the new desired state. Kubernetes also supports [autoscaling](http://kubernetes.io/docs/user-guide/horizontal-pod-autoscaling/) of Pods, but it is outside of the scope of this tutorial. Scaling to zero is also possible, and it will terminate all Pods of the specified Deployment.

Running multiple instances of an application will require a way to distribute the traffic to all of them. Services have an integrated load-balancer that will distribute network traffic to all Pods of an exposed Deployment. Services will monitor continuously the running Pods using endpoints, to ensure the traffic is sent only to available Pods.

*Scaling is accomplished by changing the number of replicas in a Deployment.*

Once you have multiple instances of an Application running, you would be able to do Rolling updates without downtime. We'll cover that in the next module. Now, let's go to the online terminal and scale our application.

#### Step 1: Scaling a deployment

To list your deployments use the get deployments command: kubectl get deployments

We should have 1 Pod. If not, run the command again. This shows:

The DESIRED state is showing the configured number of replicas

The CURRENT state show how many replicas are running now

The UP-TO-DATE is the number of replicas that were updated to match the desired (configured) state

The AVAILABLE state shows how many replicas are actually AVAILABLE to the users

Next, let’s scale the Deployment to 4 replicas. We’ll use the kubectl scale command, followed by the deployment type, name and desired number of instances:

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

To list your Deployments once again, use get deployments:

kubectl get deployments

The change was applied, and we have 4 instances of the application available. Next, let’s check if the number of Pods changed:

kubectl get pods -o wide

There are 4 Pods now, with different IP addresses. The change was registered in the Deployment events log. To check that, use the describe command:

kubectl describe deployments/kubernetes-bootcamp

You can also view in the output of this command that there are 4 replicas now.

#### Step 2: Load Balancing

Let’s check that the Service is load-balancing the traffic. To find out the exposed IP and Port we can use the describe service as we learned in the previously Module:

kubectl describe services/kubernetes-bootcamp

Create an environment variable called NODE\_PORT that has a value as the Node port:

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

Next, we’ll do a curl to the exposed IP and port. Execute the command multiple times:

curl $(minikube ip):$NODE\_PORT

We hit a different Pod with every request. This demonstrates that the load-balancing is working.

#### Step 3: Scale Down

To scale down the Service to 2 replicas, run again the scale command:

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

List the Deployments to check if the change was applied with the get deployments command:

kubectl get deployments

The number of replicas decreased to 2. List the number of Pods, with get pods:

kubectl get pods -o wide

This confirms that 2 Pods were terminated.

# **Performing a Rolling Update**

### 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. **Rolling updates** allow Deployments' update to take place with zero downtime by incrementally updating Pods instances with new ones. The new Pods will be scheduled on Nodes with available resources.

In the previous module we scaled our application to run multiple instances. This is a requirement for performing updates without affecting application availability. By default, the maximum number of Pods that can be unavailable during the update and the maximum number of new Pods that can be created, is one. Both options can be configured to either numbers or percentages (of Pods). In Kubernetes, updates are versioned and any Deployment update can be reverted to previous (stable) version.

### Summary:

* Updating an app

*Rolling updates allow Deployments' update to take place with zero downtime by incrementally updating Pods instances with new ones.*

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

*If a Deployment is exposed publicly, the Service will load-balance the traffic only to available Pods during the update.*

#### Step 1: Update the version of the app

To list your deployments use the get deployments command: kubectl get deployments

To list the running Pods use the get pods command:

kubectl get pods

To view the current image version of the app, run a describe command against the Pods (look at the Image field):

kubectl describe pods

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

kubectl set image deployments/kubernetes-bootcamp kubernetes-bootcamp=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 command:

kubectl get pods

#### Step 2: Verify an update

First, let’s check that the App is running. To find out the exposed IP and Port we can use describe service:

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

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

curl $(minikube ip):$NODE\_PORT

We hit a different Pod with every request and we see that all Pods are running the latest version (v2).

The update can be confirmed also by running a rollout status command:

kubectl rollout status deployments/kubernetes-bootcamp

To view the current image version of the app, run a describe command against the Pods:

kubectl describe pods

We run now version 2 of the app (look at the Image field)

#### Step 3: Rollback an update

Let’s perform another update, and deploy image tagged as 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

And something is wrong… We do not have the desired number of Pods available. List the Pods again:

kubectl get pods

A describe command on the Pods should give more insights:

kubectl describe pods

There is no image called v10 in the repository. Let’s roll back to our previously working version. We’ll use the rollout undo command:

kubectl rollout undo deployments/kubernetes-bootcamp

The rollout command reverted the deployment to the previous known state (v2 of the image). Updates are versioned and you can revert to any previously know state of a Deployment. List again the Pods:

kubectl get pods

Four Pods are running. Check again the image deployed on the them:

kubectl describe pods

We see that the deployment is using a stable version of the app (v2). The Rollback was successful.