**kubectl basics**

Like minikube, kubectl comes installed in the online terminal. Type kubectl in the terminal to see its usage. The common format of a kubectl command is: kubectl action resource This performs the specified action (like create, describe) on the specified resource (like node, container). You can use --help after the command to get additional info about possible parameters (kubectl get nodes --help).

Check that kubectl is configured to talk to your cluster, by running the kubectl versioncommand:

kubectl version

OK, 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:

kubectl get nodes

Here we see the available nodes (1 in our case). Kubernetes will choose where to deploy our application based on Node available resources.

**Deploy our app**

Let’s run our first app on Kubernetes with the kubectl run command. 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). We want to run the app on a specific port so we add the --portparameter:

kubectl run kubernetes-bootcamp --image=gcr.io/google-samples/kubernetes-bootcamp:v1 --port=8080

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 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 Docker container on your node.

**View our app**

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 in Module 4.

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.

We will open a second terminal window to run the proxy.

kubectl proxy

We now have a connection between our host (the online terminal) and the Kubernetes cluster. The proxy enables direct access to the API from these terminals.

You can see all those APIs hosted through the proxy endpoint, now available at through [http://localhost:8001](http://localhost:8001/). 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 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

Now we can make an HTTP request to the application running in that pod:

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

**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 podscommand:

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.

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

**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.

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*

Let’s verify that our application is running. We’ll use the kubectl get command 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 servicecommand:

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.

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

kubectl delete service -l run=kubernetes-bootcamp

**Step 1: Scaling a deployment**

To list your deployments use the get deploymentscommand: 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 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.

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

**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 podscommand:

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 image command, 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

**Getting started with Kubeadm**

**Step 1 - Initialise Master**

Kubeadm has been installed on the nodes. Packages are available for Ubuntu 16.04+, CentOS 7 or HypriotOS v1.0.1+.

The first stage of initialising the cluster is to launch the master node. The master is responsible for running the control plane components, etcd and the API server. Clients will communicate to the API to schedule workloads and manage the state of the cluster.

**Task**

The command below will initialise the cluster with a known token to simplify the following steps.

kubeadm init --token=102952.1a7dd4cc8d1f4cc5 --kubernetes-version $(kubeadm version -o short)

In production, it's recommend to exclude the token causing kubeadm to generate one on your behalf.

To manage the Kubernetes cluster, the client configuration and certificates are required. This configuration is created when *kubeadm* initialises the cluster. The command copies the configuration to the users home directory and sets the environment variable for use with the CLI.

sudo cp /etc/kubernetes/admin.conf $HOME/

sudo chown $(id -u):$(id -g) $HOME/admin.conf

export KUBECONFIG=$HOME/admin.conf

**Running Commands**

**1)kubeadm init --kubernetes-version $(kubeadm version -o short) --apiserver-advertise-address='192.168.50.5' --pod-network-cidr=192.168.0.0/16**

**2) For Non-Root Users:  
mkdir -p $HOME/.kube**

**sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config**

**sudo chown $(id -u):$(id -g) $HOME/.kube/config**

**3)for root user**

**export KUBECONFIG=/etc/kubernetes/admin.conf**

**4)Set Up CNI**

**For Canal to work correctly, --pod-network-cidr=10.244.0.0/16 has to be passed to kubeadm init. Note that Canal works on amd64 only.**

**kubectl apply -f https://docs.projectcalico.org/v3.1/getting-started/kubernetes/installation/hosted/canal/rbac.yaml**

**kubectl apply -f** [**https://docs.projectcalico.org/v3.1/getting-started/kubernetes/installation/hosted/canal/canal.yaml**](https://docs.projectcalico.org/v3.1/getting-started/kubernetes/installation/hosted/canal/canal.yaml)

**5)To run pods on the master**

**kubectl taint nodes --all node-role.kubernetes.io/master-**

**4) For Joining**

**kubeadm join 192.168.50.5:6443 --token as7dw7.secz0e69efcbvxmj --discovery-token-ca-cert-hash sha256:25487ea8c31a16df2846283ae975b53bcc6ea08c945b3a0dccb454d092037491**

**5) run app**

kubectl run docker-demo --image=dragod812/docker-demo --port=8080

kubectl get deployments

kubectl proxy