Kubernetes Basics

Objective: Putting together the basics of what we've learned, thus far. The participate will demonstrate working knowledge of basic commands and functionality.

Preparation: Simply open one or two terminal windowns, a web browser, and look to see if anything is running that might interfer with this lab and resolve it.

Outcome: Once successfully completed the participate will have a working knowledge and experience with Kubernetes basics.

Data Files: No files needed.

Step 1. Starting Minikube

> *NOTE: Skip #1 & #2 if you are not running Minikube, or if you are doing the lab in Katacoda's Playground environment.*

1. We already installed Minikube for you. Check that it is properly installed, by running the Minikube version command:

```
$ minikube version
```

OK, we can see that Minikube is in place.

2. Start the cluster, by running the Minikube start command:

```
$ minikube start --logtostderr
```

Great! You now have a running Kubernetes cluster in your online terminal. Minikube started a virtual machine for you, and a Kubernetes cluster is now running in that VM.

3. To interact with Kubernetes during this bootcamp we'll use the command line interface, kubectl. We'll explain kubectl in detail in the next modules, but for now, we're just going to look at some cluster information. To check if kubectl is installed you can run the kubectl version command:

```
$ kubectl version
```

OK, kubectl is configured and we can see both the client and the server Kubernetes version: 1.5. The client version is the kubectl version; the server version is the Kubernetes version installed on the master. Here, you can also see details about the build.

4. Let's view the cluster details. We'll do that by running kubectl cluster-info:

```
$ kubectl cluster-info
```

Step 2. Creating a Cluster

We have a running master and a dashboard. The Kubernetes dashboard allows you to view your applications in a UI. During this tutorial, we'll be focusing on the command line for deploying and exploring our application.

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

```
$ kubectl get nodes
```

This command shows all nodes that can be used to host our applications. Now we have only one node, and we can see that it's status is ready (it is ready to accept applications for deployment).

2. To list your deployments use the get deployments command:

```
$ kubectl get deployments
```

> *NOTE: Notice we don't currently have any from this lesson.*

3. We want to run the app on a specific port so we add the --port parameter:

```
$ kubectl run kubernetes-bootcamp --image=docker.io/jocatalin
/kubernetes-bootcamp:v1 --port=8080
```

4. To list your deployments use the get deployments command:

```
$ kubectl get deployments
```

Step 3. Kubernetes Basic Network

By default deployed applications are visible only inside the Kubernetes cluster.

1. To view the application output without exposing it externally, we'll create a route between our terminal and the Kubernetes cluster using a proxy:

```
$ kubectl proxy
```

We now have a connection between our host (the online terminal) and the Kubernetes cluster. The started proxy enables direct access to the API. The app runs inside a Pod (we'll cover the Pod concept in next module).

2. Get the name of the Pod and store it in the POD_NAME environment variable:

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

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

```
$ curl http://localhost:8001/api/v1/proxy/namespaces/default/
pods/$POD_NAME/
```

Output:

```
Hello Kubernetes bootcamp! | Running on: kubernetes-bootcamp-
390780338-l3pk6 | v=1
```

> *NOTE: The url is the route to the API of the Pod.*

Step 4. Exploring Your App

1. To get logs from the container, we'll use 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.*

2. We can execute commands directly on the container. 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
```

Output:

```
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/b
in:/sbin:/bin
```

```
HOSTNAME=kubernetes-bootcamp-390780338-0621d
KUBERNETES_SERVICE_HOST=10.0.0.1
KUBERNETES_SERVICE_PORT=443
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_PORT=tcp://10.0.0.1:443
KUBERNETES_PORT_443_TCP=tcp://10.0.0.1:443
KUBERNETES_PORT_443_TCP_PROTO=tcp
KUBERNETES_PORT_443_TCP_PORT=443
KUBERNETES_PORT_443_TCP_ADDR=10.0.0.1
NPM_CONFIG_LOGLEVEL=info
NODE_VERSION=6.3.1
HOME=/root
```

> *NOTE: Again, worth mentioning that the name of the container itself can be omitted in this case because we have only one running in the Pod.*

3. Next let's start a bash session in the Pod's container:

```
$ kubectl exec -ti $POD_NAME bash
```

4. 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
```

Output:

```
var http = require('http');
var requests=0;
var podname= process.env.HOSTNAME;
var startTime;
var host;
var handleRequest = function(request, response) {
  response.setHeader('Content-Type', 'text/plain');
  response.writeHead(200);
  response.write("Hello Kubernetes bootcamp! | Running on: ")
;
  response.write(host);
  response.end(" | v=1\n");
  console.log("Running On:" ,host, "| Total Requests:", ++req
uests,"| App Uptime:", (new Date() - startTime)/1000 , "secon
ds", "| Log Time:",new Date());
}
var www = http.createServer(handleRequest);
www.listen(8080,function () {
    startTime = new Date();;
    host = process.env.HOSTNAME;
    console.log ("Kubernetes Bootcamp App Started At:",startT
ime, "| Running On: " ,host, "\n" );
});
```

5. You can check that the application is up by running a curl

command:

```
# curl localhost:8080
```

Output:

```
Hello Kubernetes bootcamp! | Running on: kubernetes-
bootcamp-390780338-0621d | v=1
```

> *NOTE: here we used localhost because we executed the command inside the NodeJS container*

Close your container connection:

```
exit
```

Step 5. Expose Your App Publicly

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

```
$ kubectl get pods
```

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

```
$ kubectl get services
```

3. 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="NodeP
ort" --port 8080
```

4. Let's run again the get services command:

```
$ kubectl get services
```

We have now a running Service called kubernetes-bootcamp.

5. To find out what port was opened externally (by the NodePort option) we'll run the describe service command:

```
$ kubectl describe services/kubernetes-bootcamp
```

6. Create an environment variable called NODE_PORT that has as value the Node port:

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

7. 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
```

Output:

```
Hello Kubernetes bootcamp! | Running on: kubernetes-
bootcamp-390780338-8d0rf | v=1
```

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

8. The Deployment created automatically a label for our Pod. With describe deployment command you can see the name of the label:

```
$ kubectl describe deployment
```

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

```
$ kubectl get pods -l run=kubernetes-bootcamp
```

10. You can do the same to list the existing services:

```
$ kubectl get services -l run=kubernetes-bootcamp
```

11. Get the name of the Pod and store it in the POD_NAME environment variable:

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

12. 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
```

13. 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
```

14. 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.

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

```
$ kubectl delete service -l run=kubernetes-bootcamp
```

16. Confirm that the service is gone:

```
$ kubectl get services
```

17. 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
```

> *NOTE: This command should fail, because the Service has been removed.*

18. 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.

Step 6. Scale Your App

1. List your deployments use the get deployments command:

```
$ kubectl get deployments
```

We should have 1 Pod.

- 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

2. 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
```

3. To list your Deployments once again, use get deployments:

```
$ kubectl get deployments
```

4. 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
```

5. 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.

6. 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:

```
$ kubectl expose deployment/kubernetes-bootcamp --type="NodeP
ort" --port 8080
$ kubectl describe services/kubernetes-bootcamp
```

7. Create an environment variable called NODE_PORT that has as value the Node port:

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

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

```
$ curl $(minikube ip):$NODE_PORT
```

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

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

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

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

```
$ kubectl get deployments
```

11. 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.

Step 7. Update Your App

1. To list your deployments use the get deployments command:

```
$ kubectl get deployments
```

2. To list the running Pods use the get pods command:

```
$ kubectl get pods
```

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

```
$ kubectl describe pods
```

4. 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 kubernete
s-bootcamp=jocatalin/kubernetes-bootcamp:v2
```

5. 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
```

6. 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
```

7. Create an environment variable called NODE_PORT that has as value the Node port:

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

8. 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).

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

```
$ kubectl rollout status deployments/kubernetes-bootcamp
```

10. 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)!

11. Let's perform another update, and deploy image tagged as v10 :

```
$ kubectl set image deployments/kubernetes-bootcamp kubernete
s-bootcamp=jocatalin/kubernetes-bootcamp:v10
```

12. Use get deployments to see the status of the deployment:

```
$ kubectl get deployments
```

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

```
$ kubectl get pods
```

14. A describe command on the Pods should give more insights:

```
$ kubectl describe pods
```

15. 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
```

16. 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
```

17. 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.

Conclusion

And with that we'll conclude this lab. Now, you've demonstrated a working knowledge of Kubernetes, using kubectl to interact with the Kubernetes API.