### **Module 1 - Create a Kubernetes cluster**

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.

Start the cluster, by running the *minikube start* command:

minikube start

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.

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 version of the client and as well as the server. The client version is the kubectl version; the server version is the Kubernetes version installed on the master. You can also see details about the build.

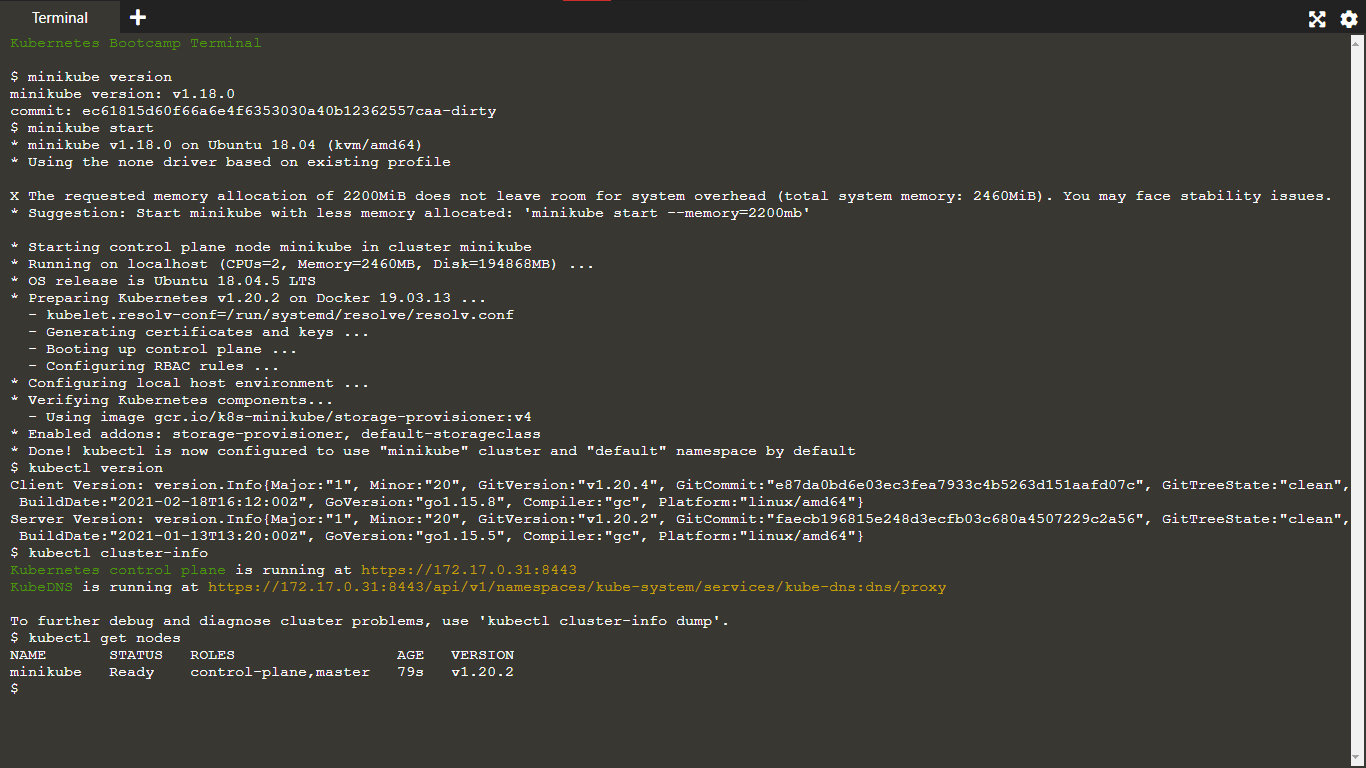
Let’s view the cluster details. We’ll do that by running kubectl cluster-info:

kubectl cluster-info

During this tutorial, we’ll be focusing on the command line for deploying and exploring our application. 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 its status is ready (it is ready to accept applications for deployment).



# **Using kubectl to Create a Deployment**

### **Objectives**

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

### **Kubernetes Deployments**

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 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

A Deployment is responsible for creating and updating instances of your application

## **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.

Applications need to be packaged into one of the supported container formats in order to be deployed on Kubernetes

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).

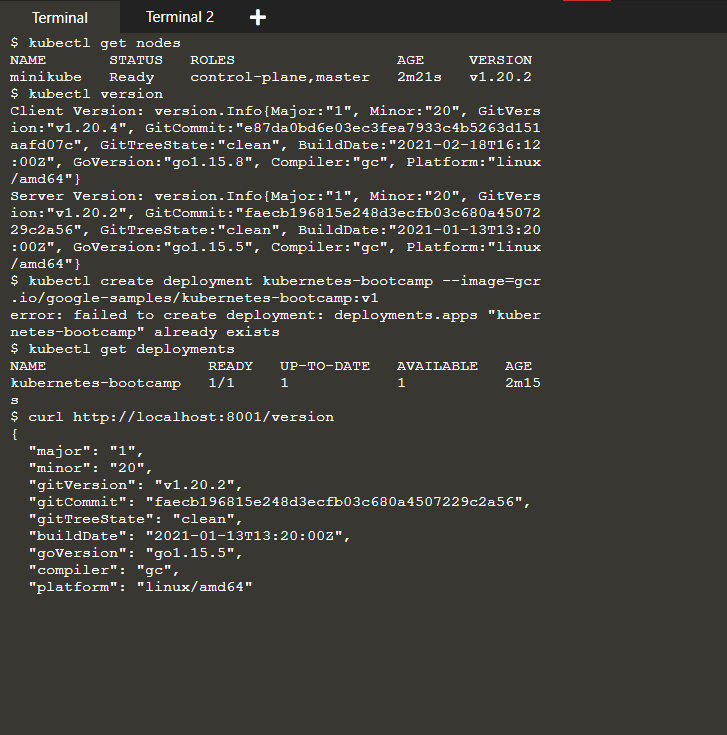
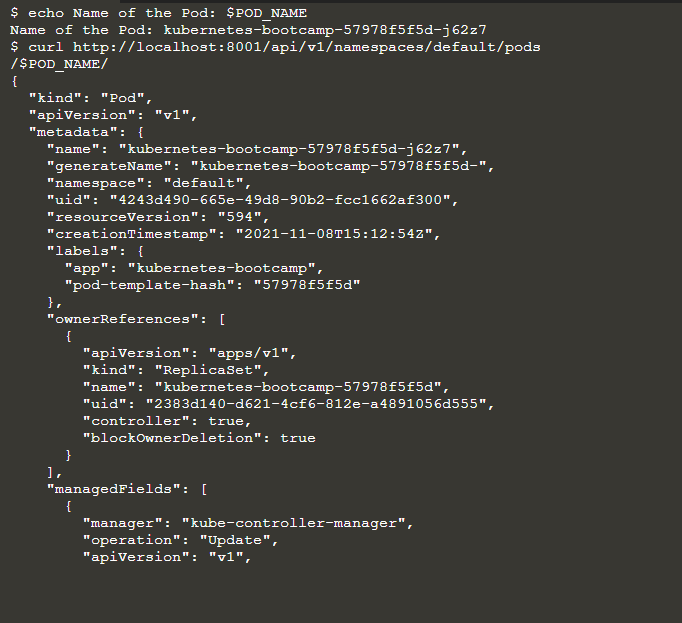
kubectl create deployment kubernetes-bootcamp --image=gcr.io/google-samples/kubernetes-bootcamp:v1

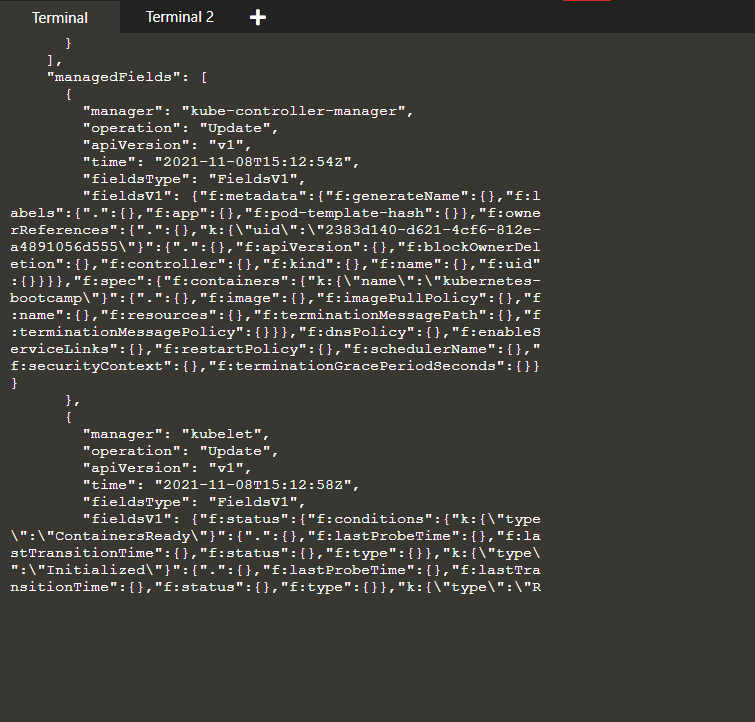
Great! You just deployed your first application by creating a deployment. This performed a few things for you:

* 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 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.

echo -e "\n\n\n\e[92mStarting Proxy. After starting it will not output a response. Please click the first Terminal Tab\n";

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. For example, we can query the version directly through the API using the curl command:

curl http://localhost:8001/version

*Note: Check the top of the terminal. The proxy was run in a new tab (Terminal 2), and the recent commands were executed the original tab (Terminal 1). The proxy still runs in the second tab, and this allowed our curl command to work using localhost:8001.*

**If Port 8001 is not accessible, ensure that the kubectl proxy started above is running.**

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

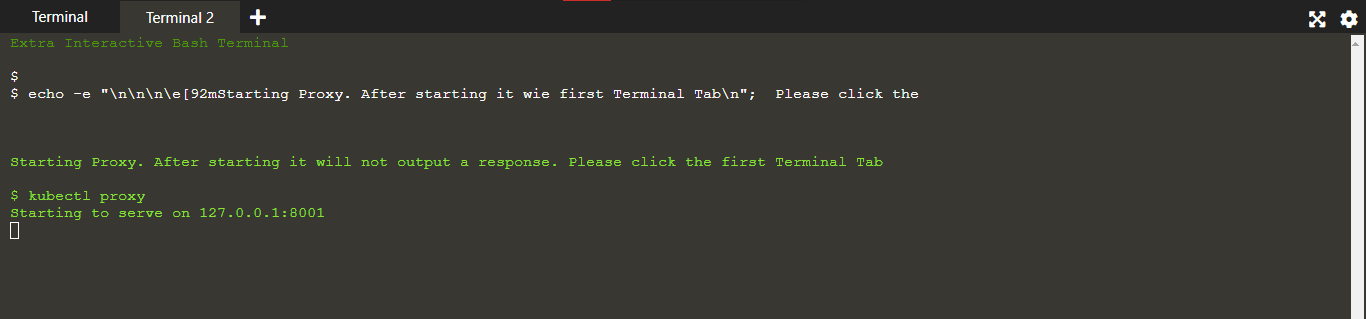
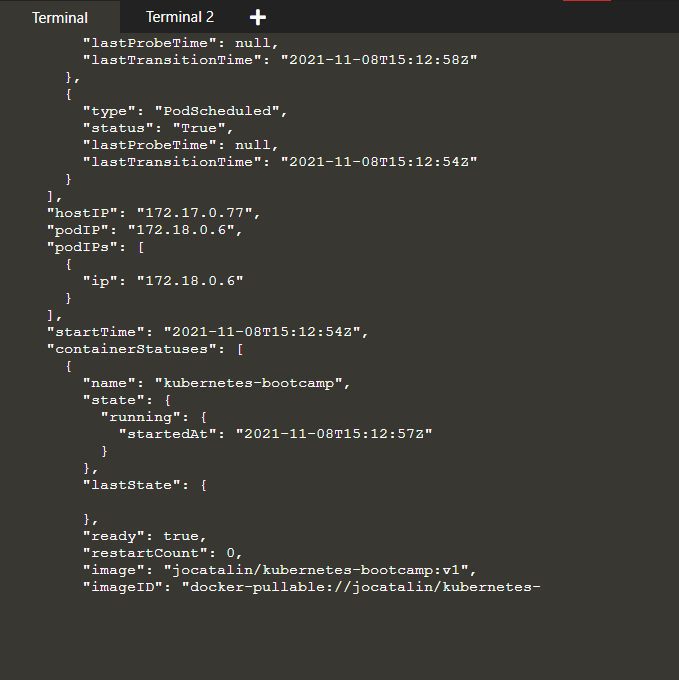
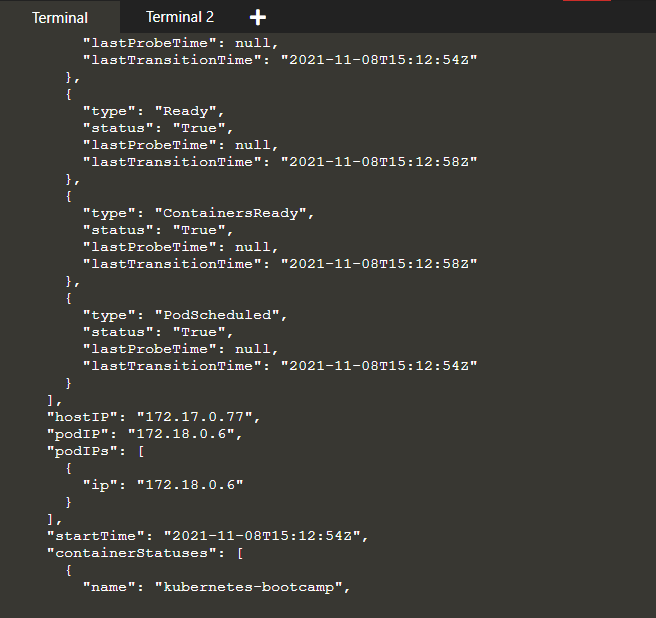
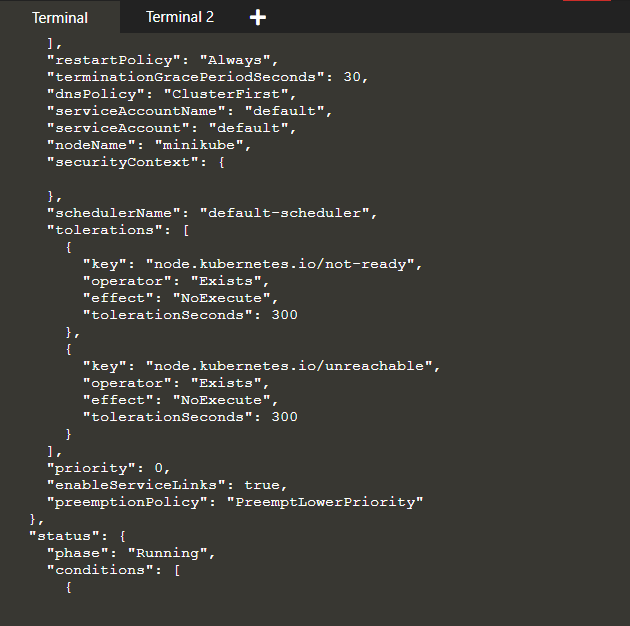
First we need to get the Pod name, and we'll store in the environment variable POD\_NAME:

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

You can access the Pod through the API by running:

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

In order for the new deployment to be accessible without using the Proxy, a Service is required which will be explained in the next modules.



#### 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 then it means the interactive environment is still reloading it's previous state. Please wait a couple of seconds and list the Pods again. You can continue once you see the one Pod running.

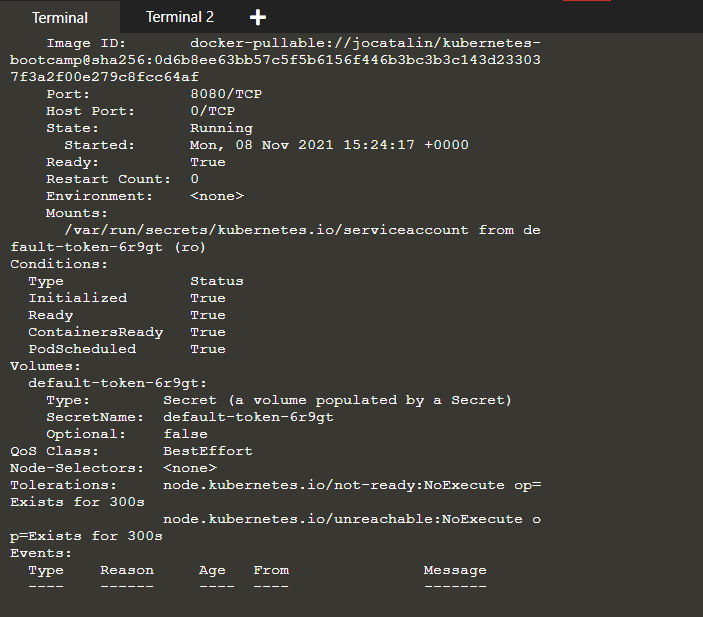
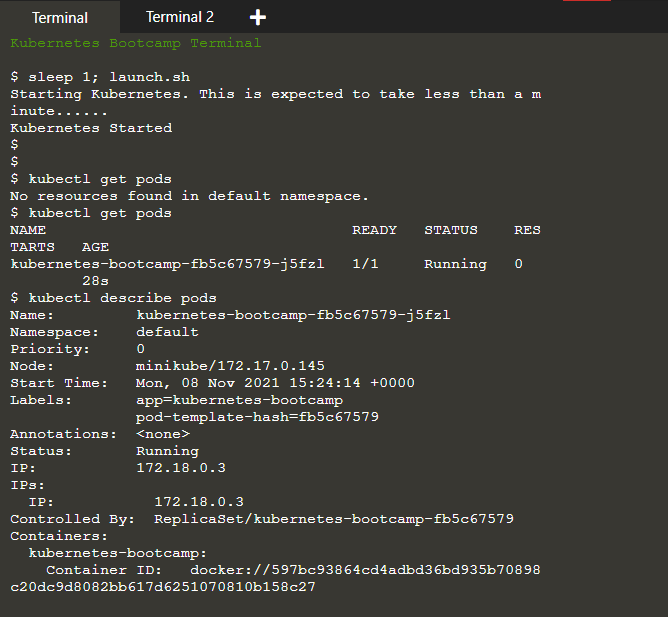
Next, to view what containers are inside that Pod and what images are used to build those containers we run the describe pods command:

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:

echo -e "\n\n\n\e[92mStarting Proxy. After starting it will not output a response. Please click the first Terminal Tab\n"; 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.

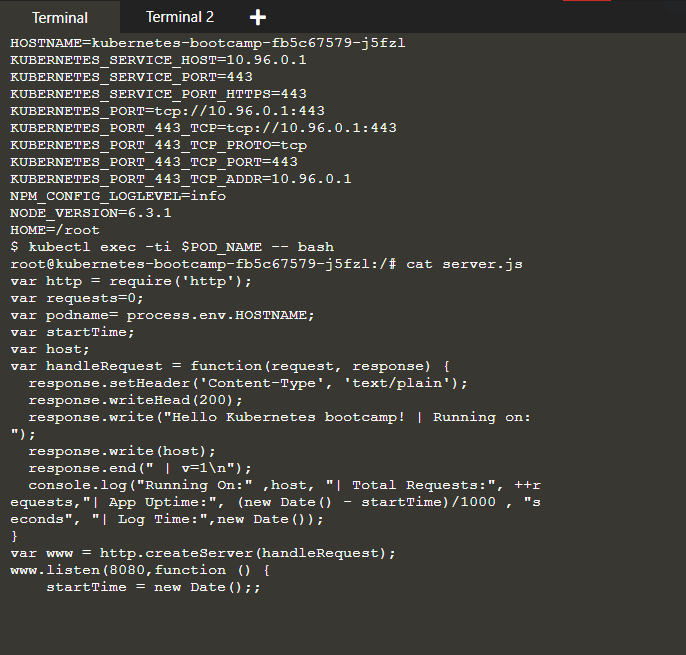
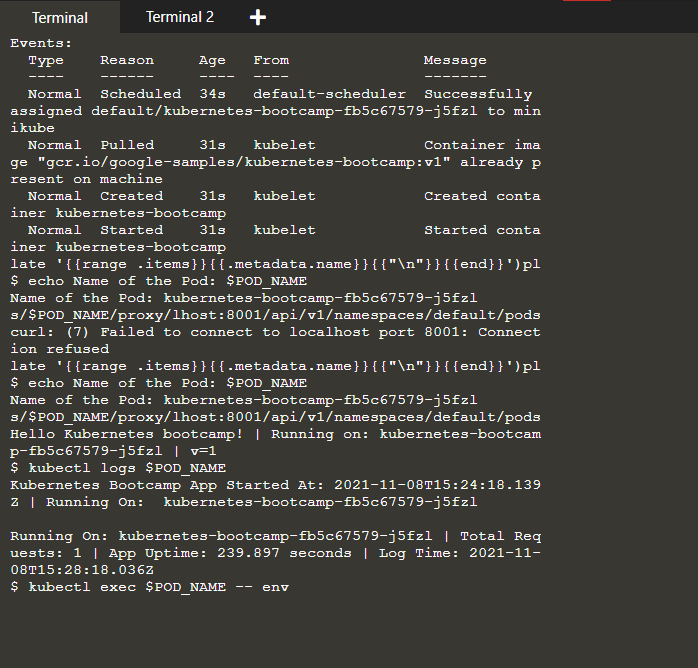
curl http://localhost:8001/api/v1/namespaces/default/pods/$POD\_NAME/proxy/

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

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 Pod. If you cannot connect to localhost:8080, check to make sure you have run the kubectl exec command and are launching the command from within the Pod

To close your container connection type exit.

