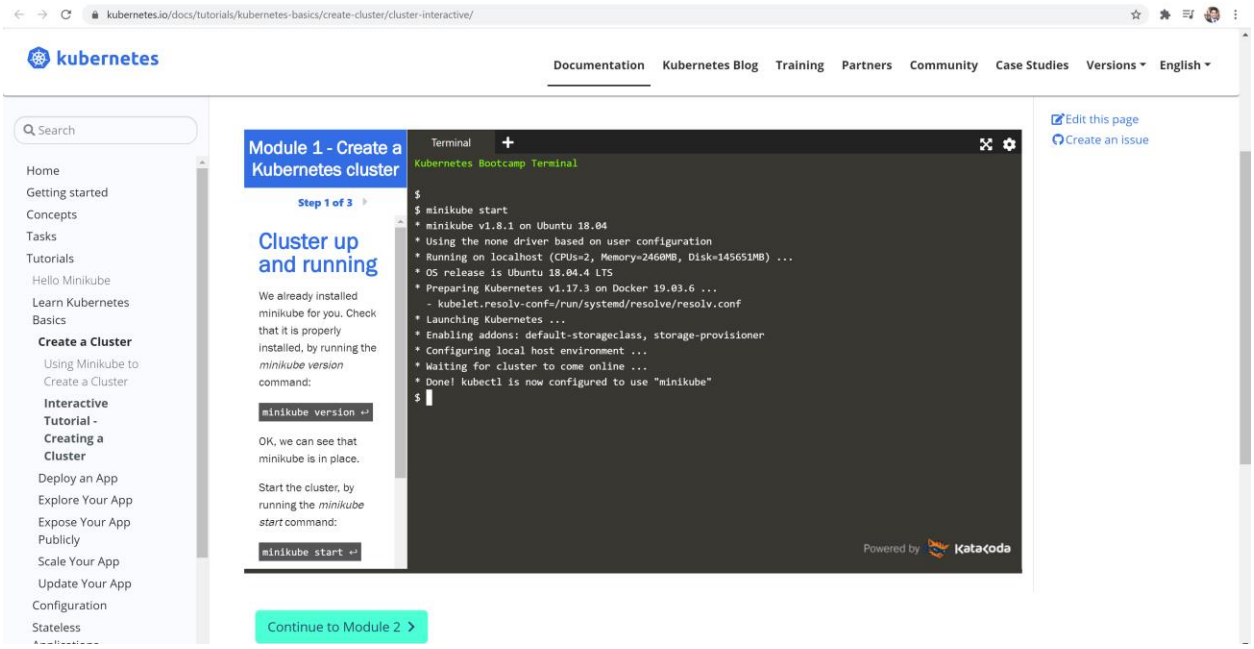
**Kubernetes** is an open-source platform from Goggle designed for the containerized applications to run when and where we want with the ease to update the applications for new features and bugs fixes while allowing these applications to run 24/7. Thus, Kubernetes allows the containerized applications to run as loosely coupled applications on a cluster of machines which are highly available and automates the distribution and scheduling of the application containers running on the cluster efficiently.

Kubernetes Cluster follows a Master Slave Architecture where master controls all the nodes, nodes are the worker nodes assigned tasks to be completed by the master.

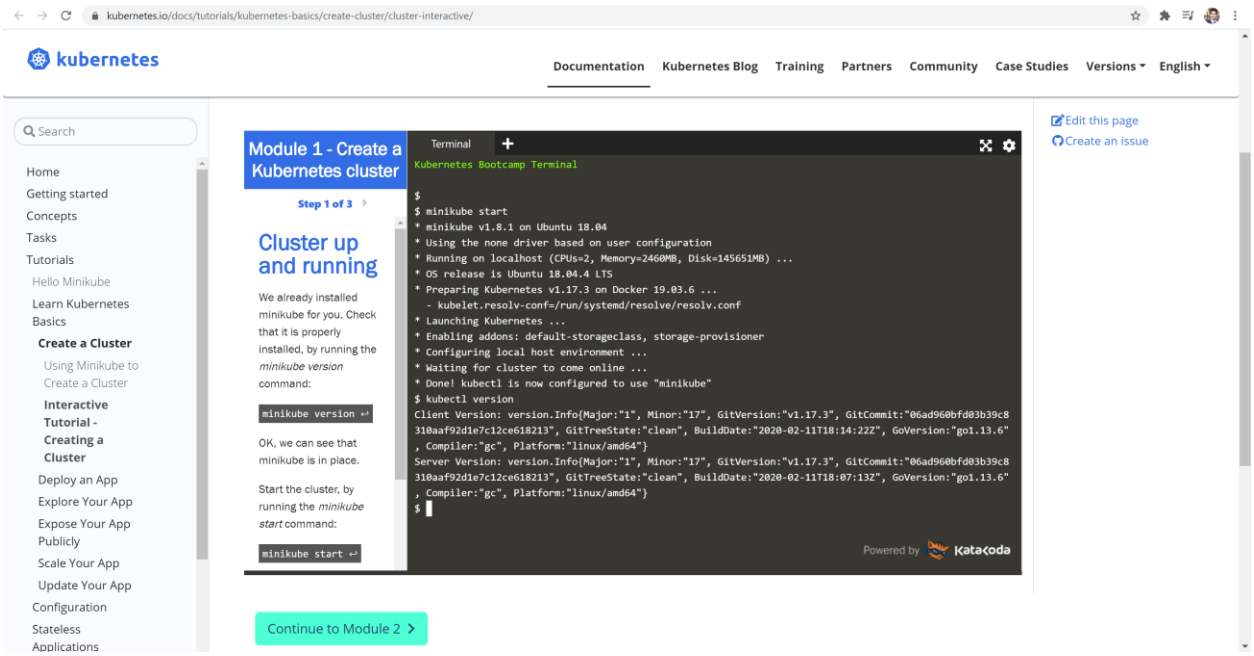
## Module 1 - Create a Kubernetes Cluster

We check if Minikube is running properly or not. Minikube is a lightweight Kubernetes implementation that creates a VM and deploys a simple cluster containing only one node. Kubernetes abstracts the deployment of applications in a cluster without tying them specifically to a machine in a cluster creating a loosely coupled connection. Cluster is started using the "minikube start" command. To be able to communicate with Kubernetes Cluster, we use kubectl which is kind of same as normal command line interface like putty or Windows Powershell we would use to interact with the applications for running the commands. Kubectl version command is used to check if this CLI is up and running or not.

Cluster up and running

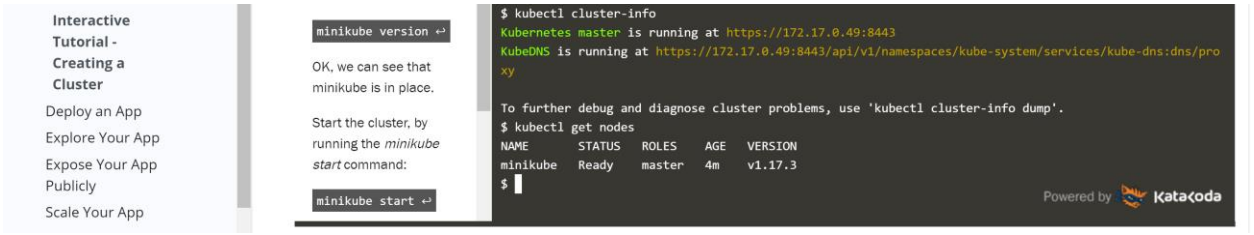



Cluster Version




Kubectl is configured on both the master and the nodes. kubectl version is the client version whereas Kubernetes version is the master installed on the master node. “kubectl cluster-info” command shows the cluster details indicating the master is running and the location at which the master is running. “kubectl get nodes” command is used to show all the nodes in the cluster to be used for deployment. We only have 1 node which is having Status as “Ready” meaning it is ready for deployment and it is a Master Node as indicated by the Roles.

Cluster details

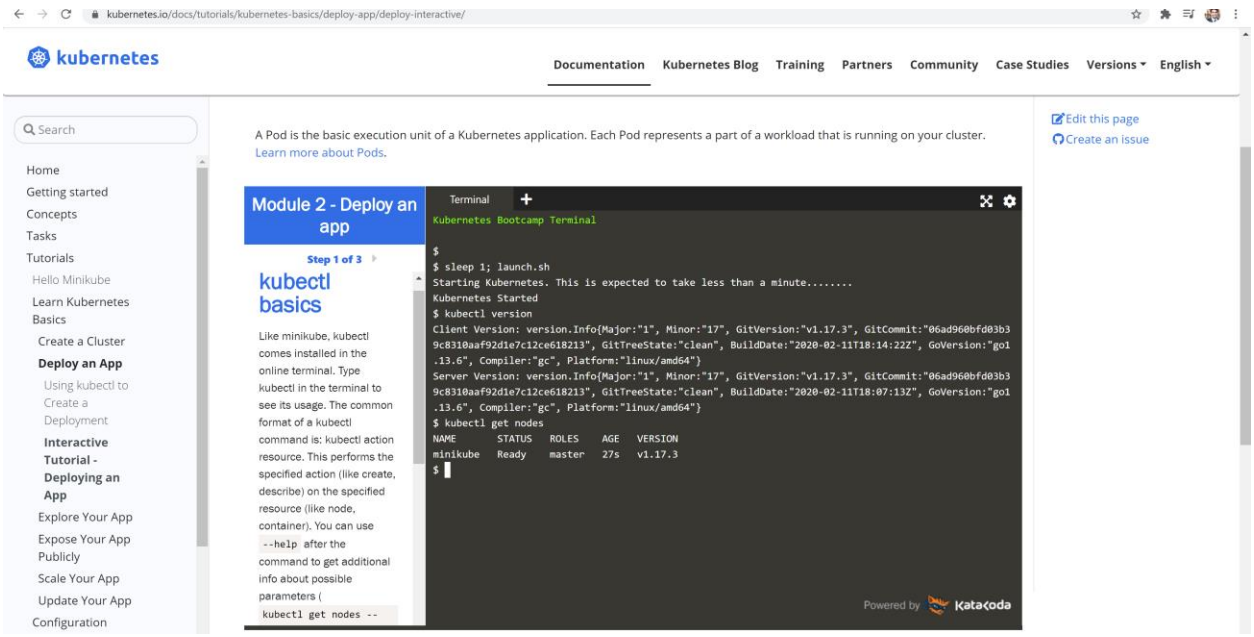



## Module 2 - Deploy an App

Once the Kubernetes Cluster is up and running, e would deploy our containerized application on top of this cluster. This deployment instructs Kubernetes on how to create and update the instances of the application.

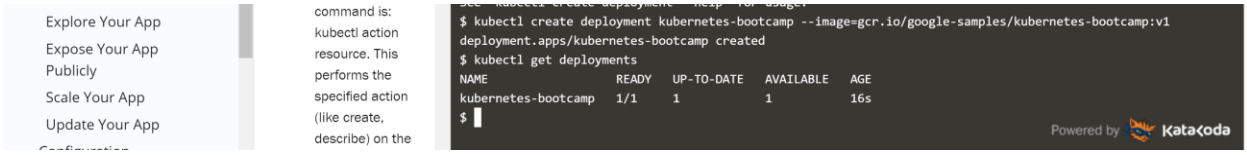
kubectl basics

“kubectl version” command is used to check if the kubectl CLI is up and running for us to interact with the cluster to deploy our application along with checking the nodes in the cluster using the “kubectl get nodes”.



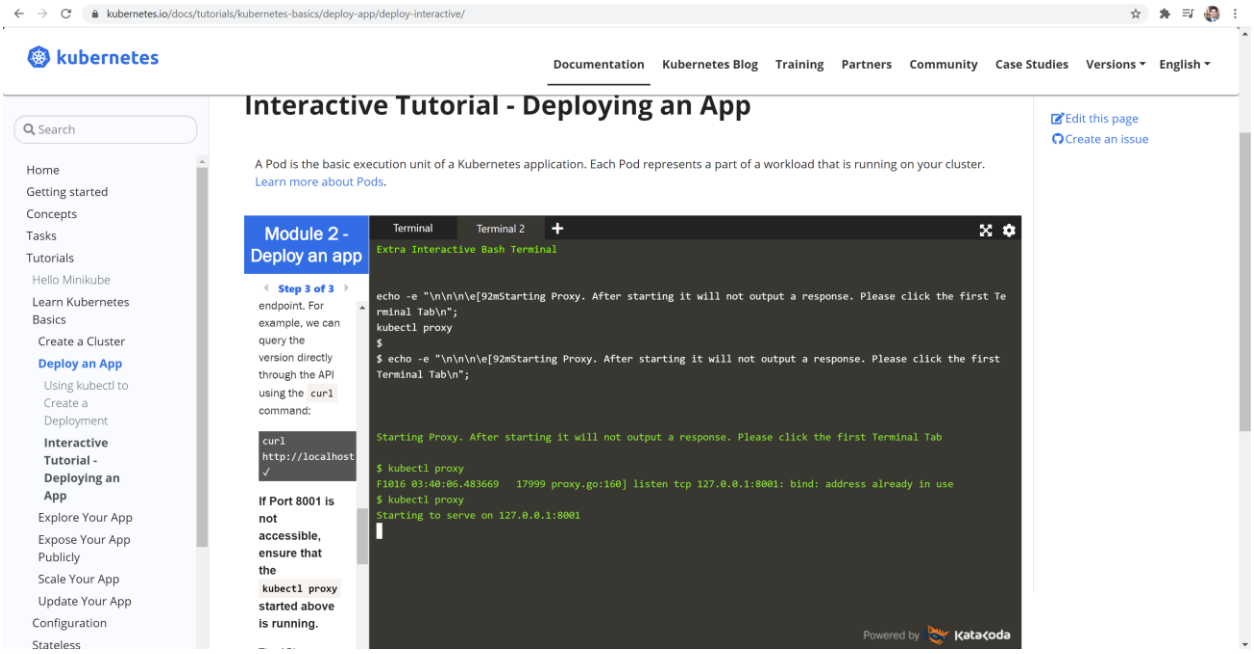

Deploying the application using the “kubectl create deployment” command with Kubernetes-bootcamp being the name of the deployment and image location of the base image using the image parameter. The Application deployment is successful using the image file specified which can be checked by using “kubectl get deployments” showing the deployment running on the single instance.

Deploy our app

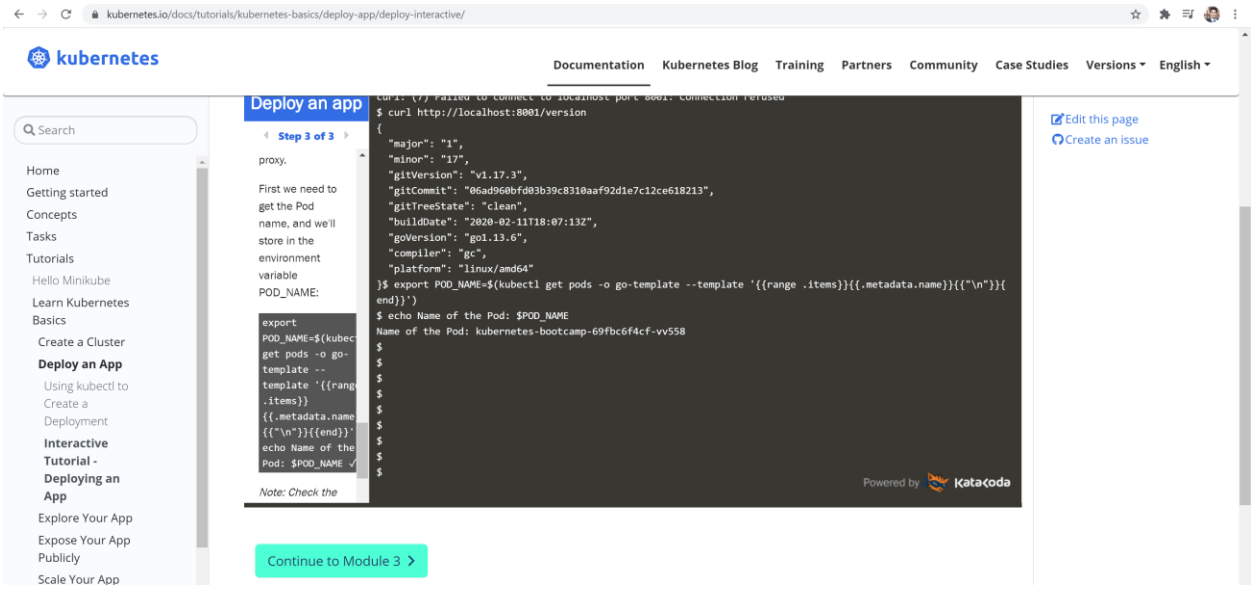



View our app

We create proxy to communicate with the internal private and isolated network which cannot be accessed from the outside. When using kubectl we are interacting through API endpoint to interact with the application deployed on this private Kubernetes cluster. So, this proxy created would forward the communications to this internal private network for us to interact with the application from outside enabling direct access to all the API from the terminal. Terminal 2 opens up when starting the proxy.




Now, we are getting the POD Name and storing it in the environment variable POD_NAME for later use to connect to the containerized application running on that pod.



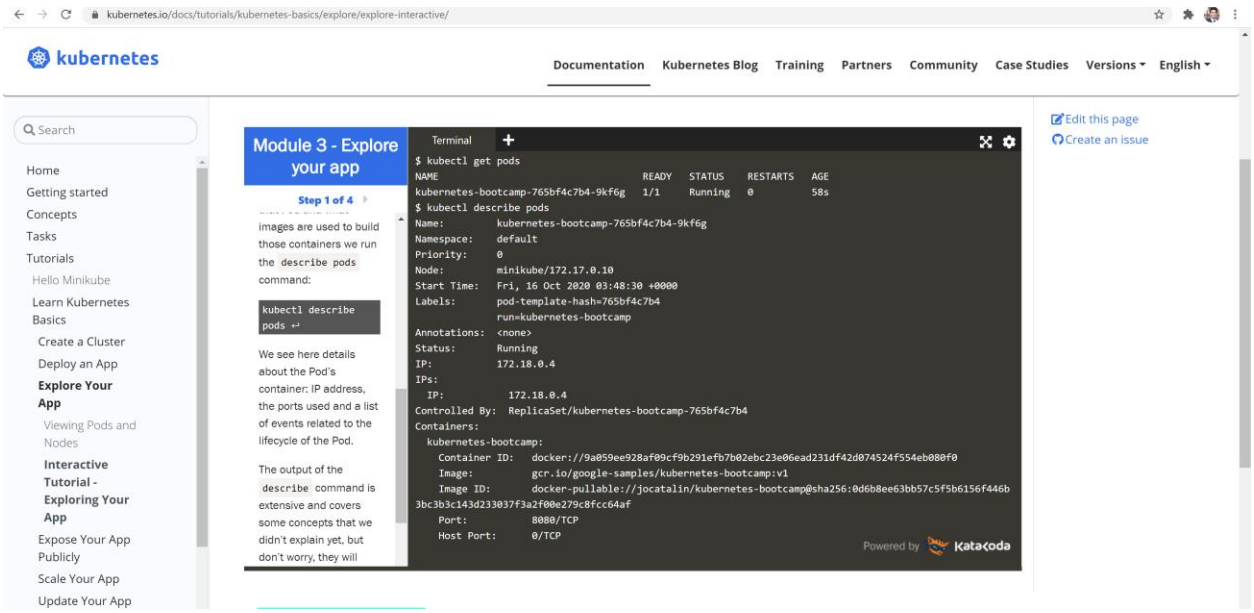

## Module 3 - Explore Your App

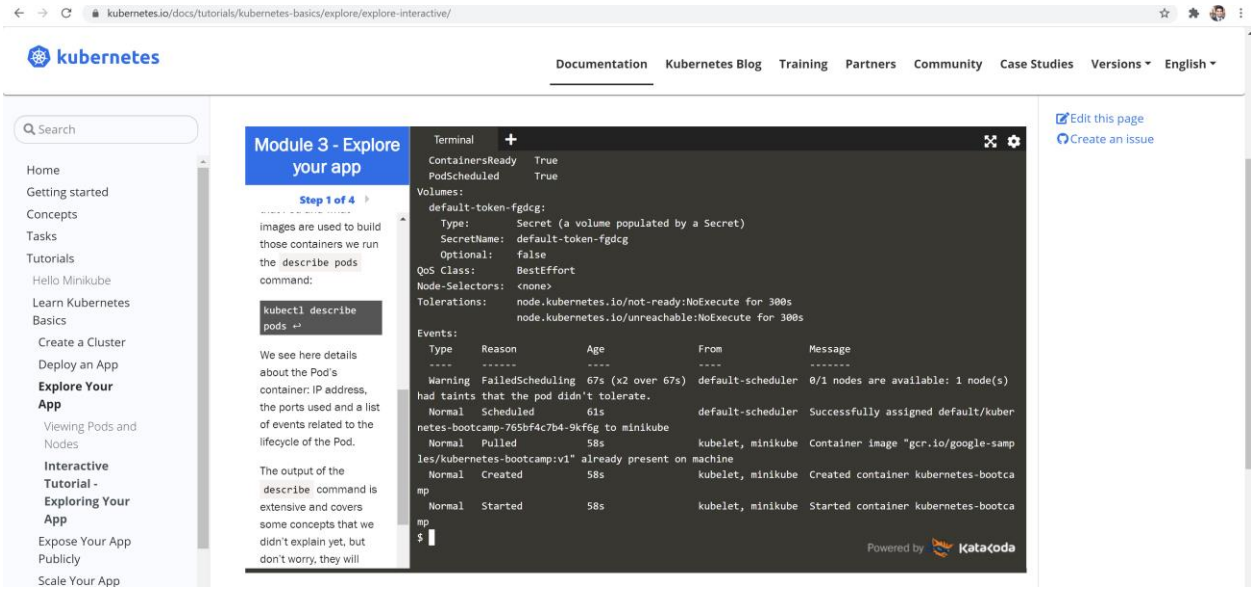
Check application configuration

When we did the Deployment, Kubernetes by default created a Pod to host our application instance. A Pod is a Kubernetes abstraction that represents a group of one or more application containers and some shared resources for those containers which include shared storage,

networking etc. So, when create deployment on Kubernetes, Kubernetes by default creates a Pod which has containers in it. Each Pod is associated with a particular Node.

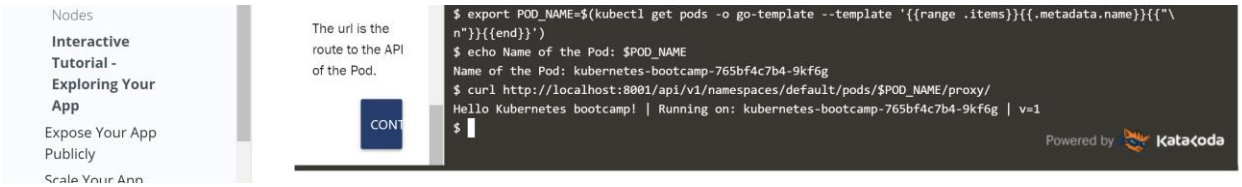
“kubectl get pods” commands shows the pods running whereas “kubectl describe pods” would give detailed description of the pods including the containers information which are located inside the pod, IP address, ports being used etc.



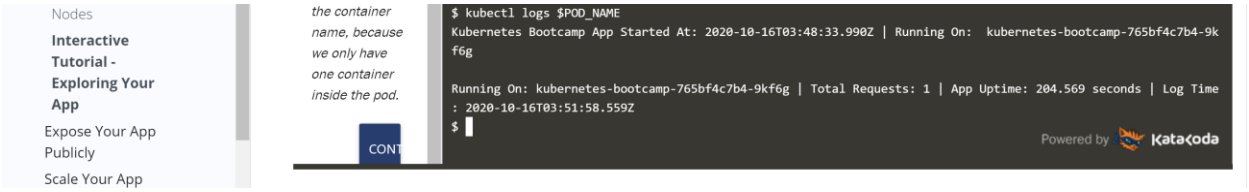

Show the app in the terminal

We are getting the POD Name and querying the POD directly through the use of proxy. In order to be able to see the output of the application, we use curl request running on port 8001 with the url being the way to the API of the POD.



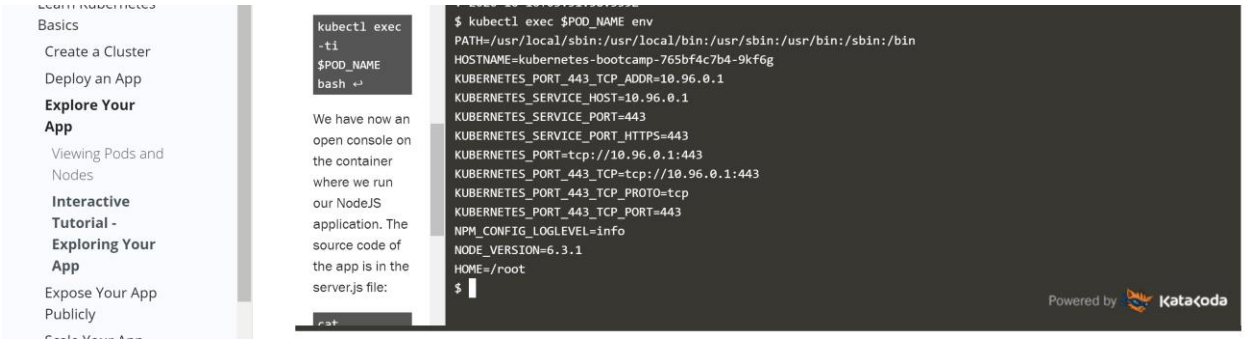

View the container logs

"kubectl logs $POD_NAME" command is used to get/retrieve the logs of the container inside the POD. Since we have only 1 POD we are not explicitly specifying the POD Name.

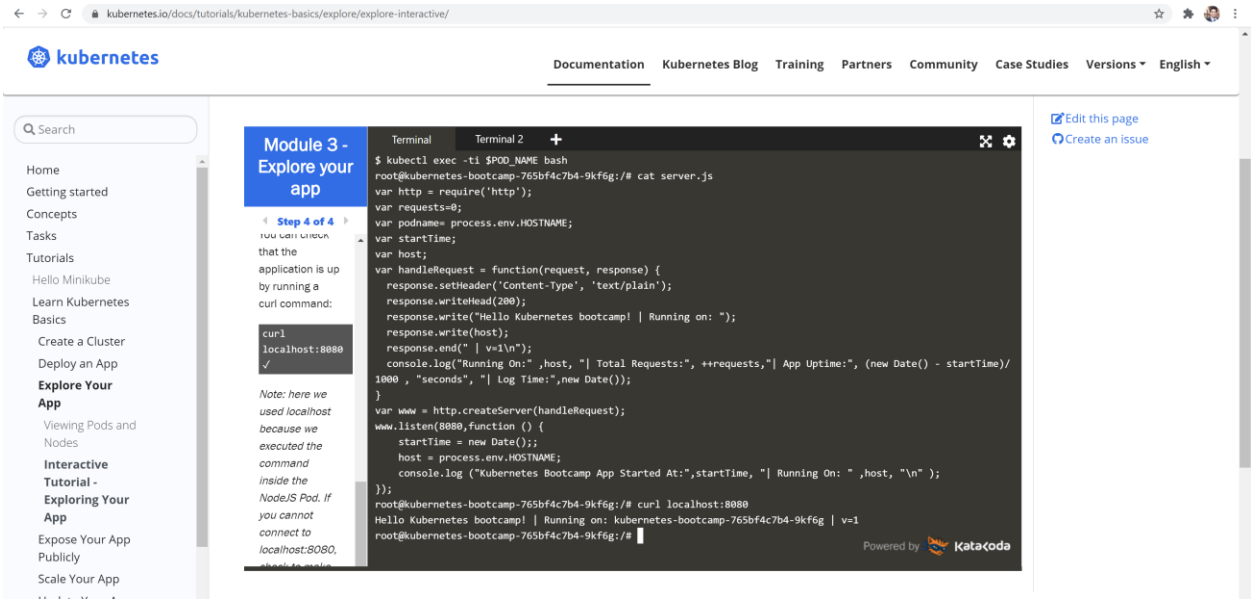



Executing command on the container

Executing the commands directly on the container contained in the pod once the pod is up and running. To execute the commands, we use "exec" command.




Running the Nodejs application using the open console on the container after starting a bash session. The source code the application is in server.js file and this application can be checked to be up and running using the curl command.



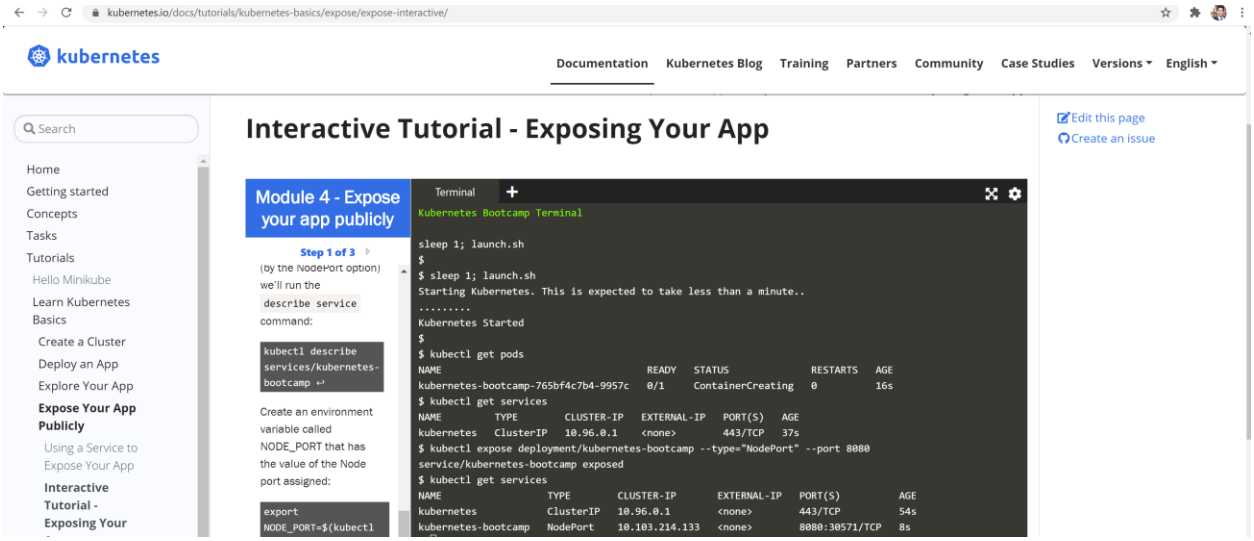

## Module 4 - Expose Your App Publicly

Service in Kubernetes is an abstraction to define logical set of pods and a policy through which we can access this logical set of pods enabling the loose coupling between the dependent pods. Services route the traffic across a set of dependent pods.

Create a new service

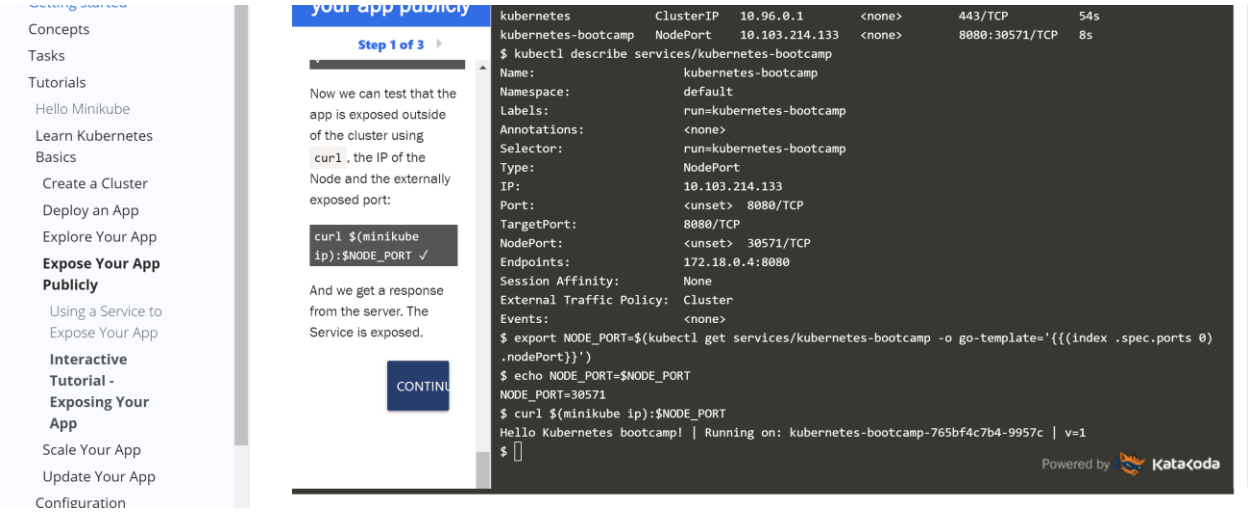
We check for existing pods using the “kubectl get pods” command. “kubectl get services” gives the list of current services from the cluster. We have a Kubernetes service running which is created by default when minikube starts the cluster.

To create a new service and expose it to the external traffic we use “kubectl expose” command with NodePort as a parameter. We now have successfully created this new service.



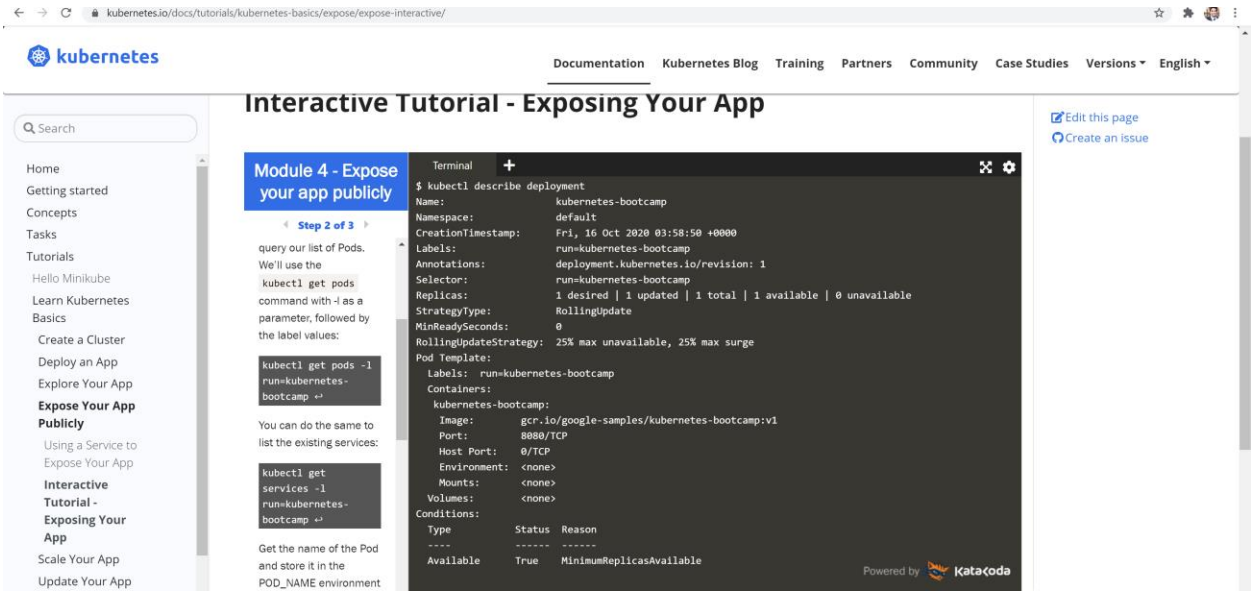
We describe the service using “kubectl describe service” command to get more details about the service such as the port on which this service is running, IP address etc.

To check if the application is exposed to external traffic, we use the UP address and the port mentioned in the describe command output to access the application using the curl command.




Using labels

When the deployment was created, the pod was automatically given a label which can be viewed using the “kubectl describe deployment” command.

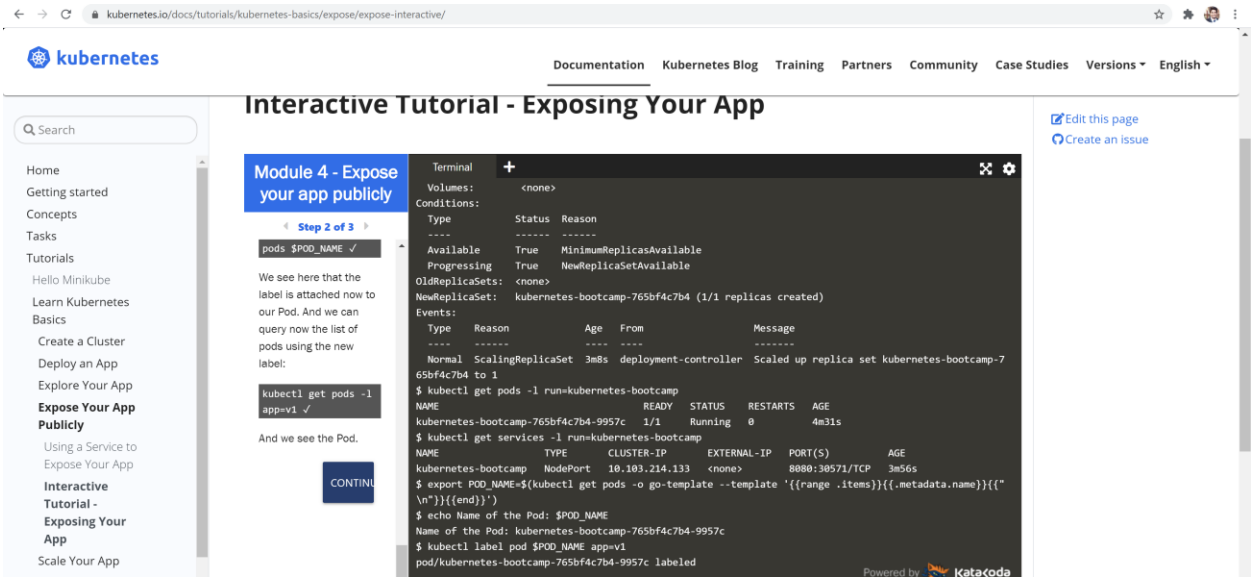



Using the pod label, we got from the above to query the list of pods using the parameter -l to specify the pod label to get its details. To get the service details, same “kubectl get” command is used. Also, storing the pod name in the environment variable POD_NAME.

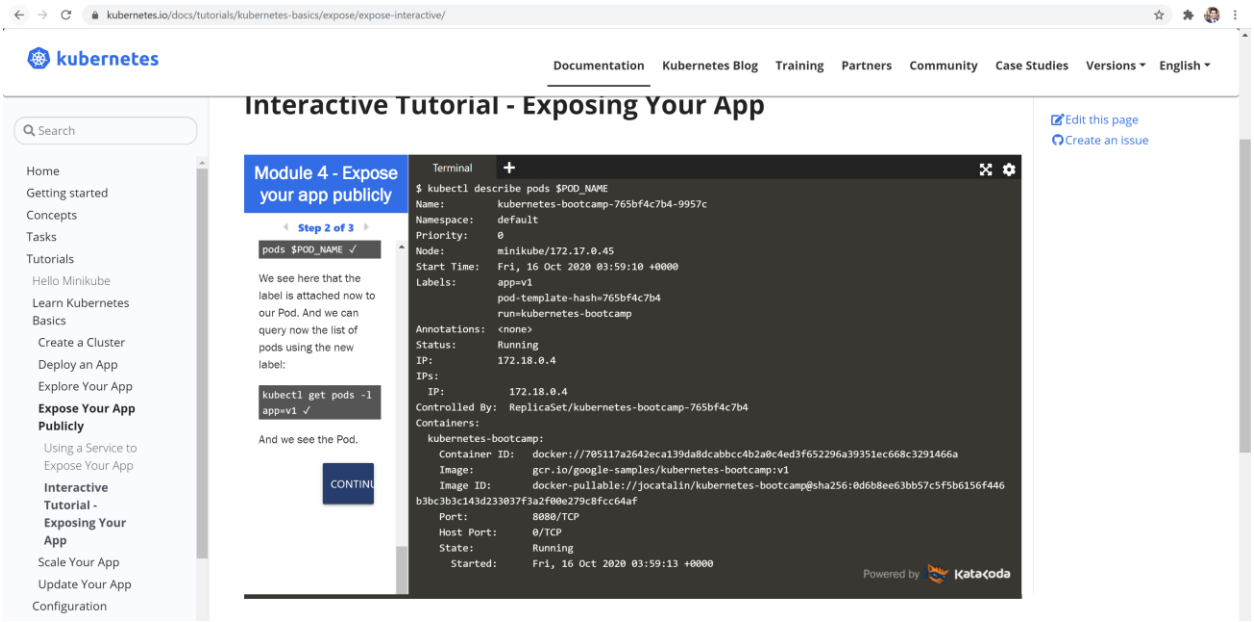
We can also change the name of the label using the following command:

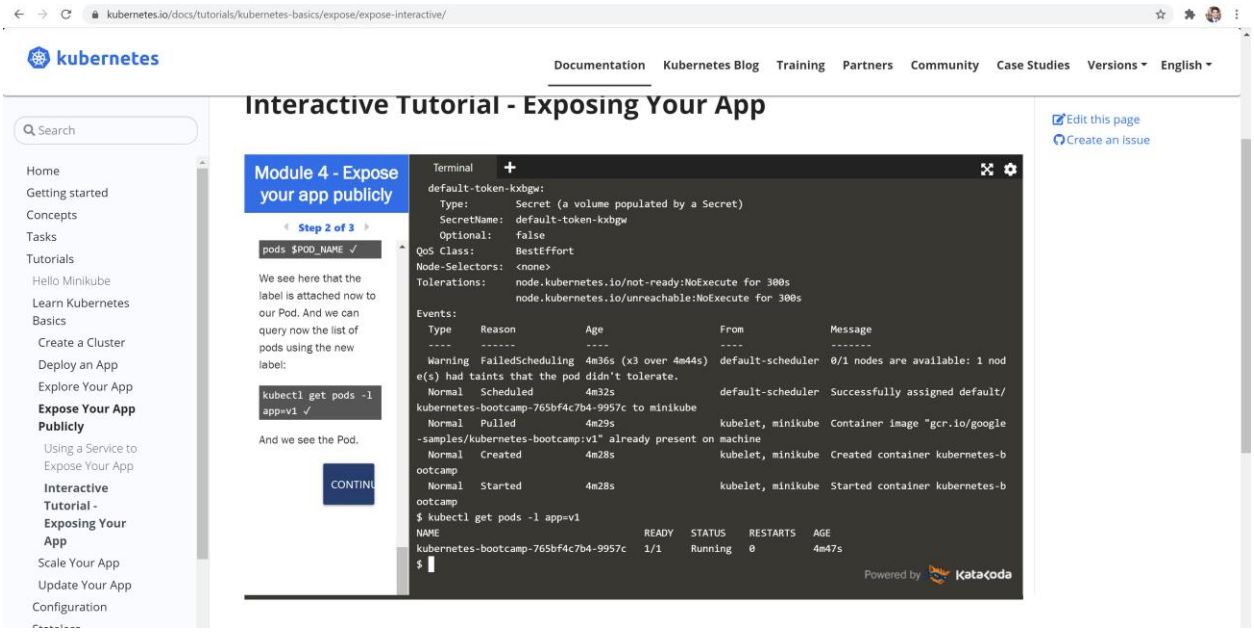
```
kubectl label pod $POD_NAME app=v1
```

New label is assigned to the current POD Name stored in the POD_NAME variable.




Now describing the pod again, we can see that new label is added to the pod while the old label has not been deleted yet.

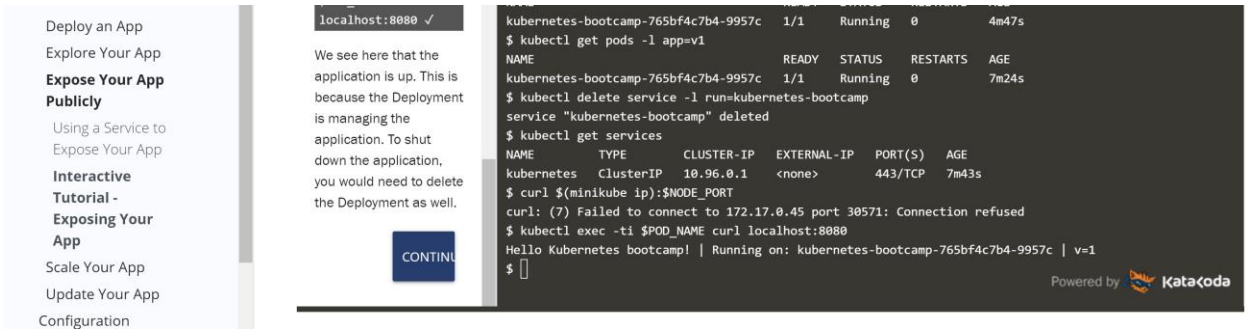





Deleting a service

We now delete the service using the "kubectl delete service" command and if the service is deleted or not using the "kubectl get services" command.

Now to check if the app is exposed to the external traffic which was achieved by using the service which was deleted, we used "curl" command which shows it is not accessible from the outside. But it can be accessed internally using the kubectl with the app being up and running.



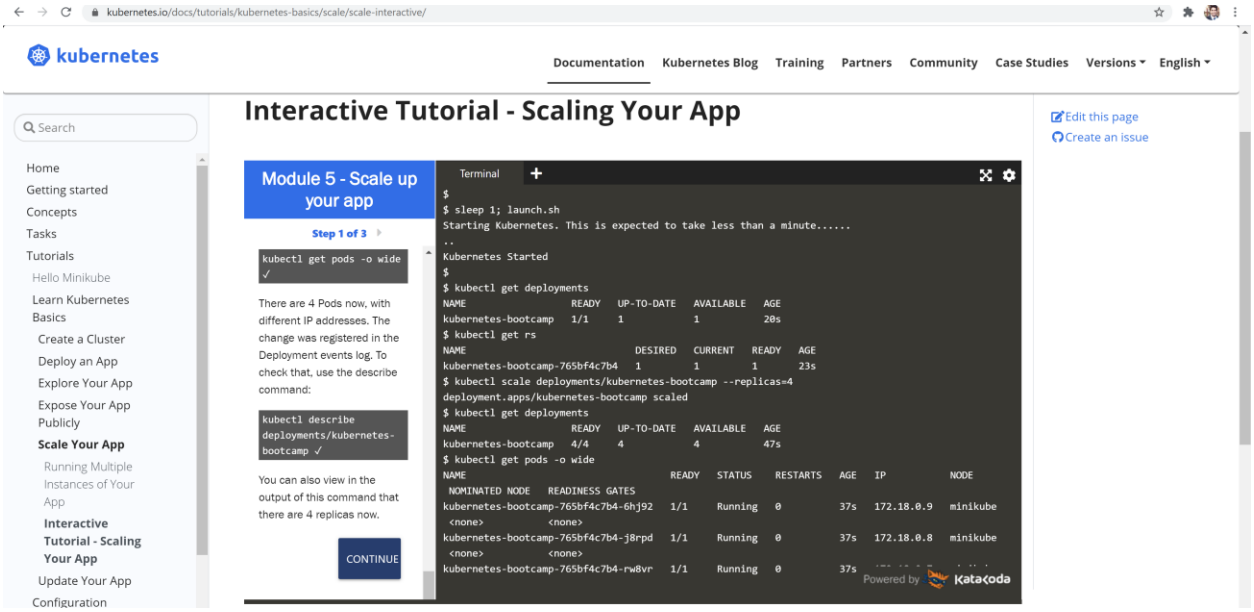

**Module 5 - Scale Your App**

Scaling a deployment

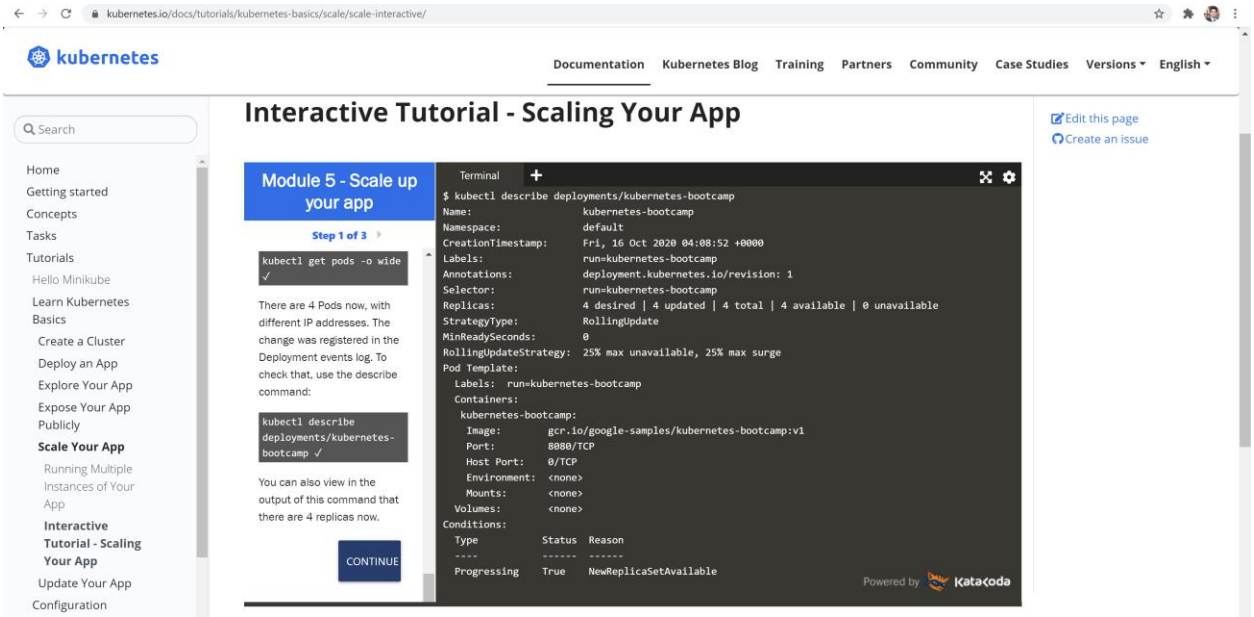
After deploying the app and creating service to expose the app to the outside world, we only have 1 pod. With the growing needs of the business and when the traffic increases, we would need to scale our application to accept more number of requests than before causing us to scale up the application.

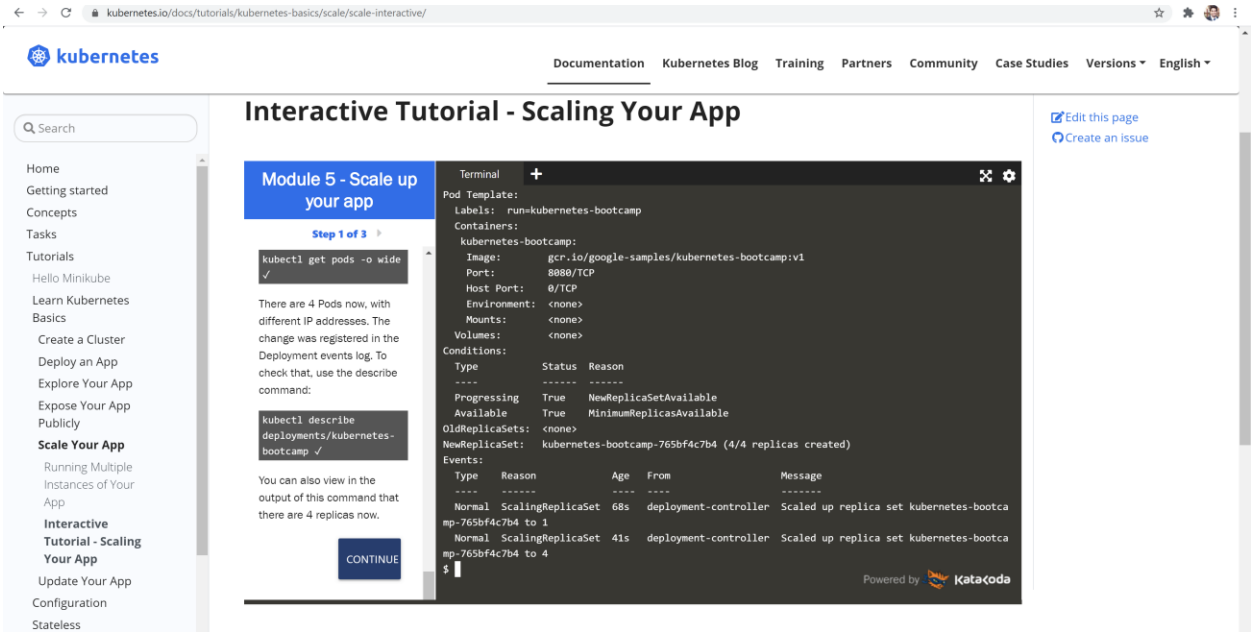
Scaling up means creating new pods as and when required. Multiple instances of an application can be run by distributing the load among these pods and we can perform Rolling updates without downtime as the other pods would be working to cater to the business without downtime.

We first check the deployments which shows only 1 pod running. We scale up the deployment by increasing the replicas in the deployment to now 4 up and running instances. These 4 instances along with their different IP addresses is visible by using “kubectl get pods -o wide” command. “kubectl get rs” command gives the ReplicaSet information now there is only 1 replica available.



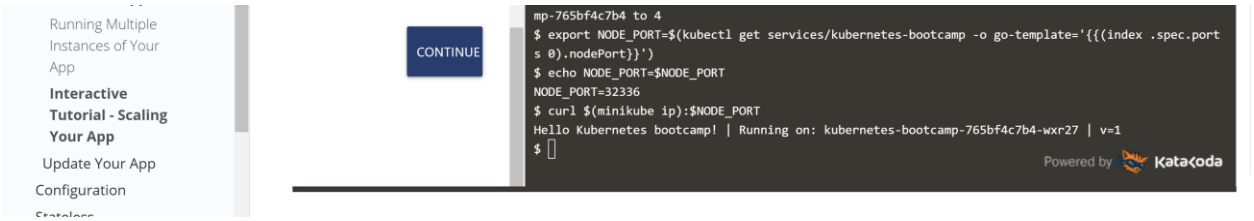

Describe command shows changes in the deployments events log



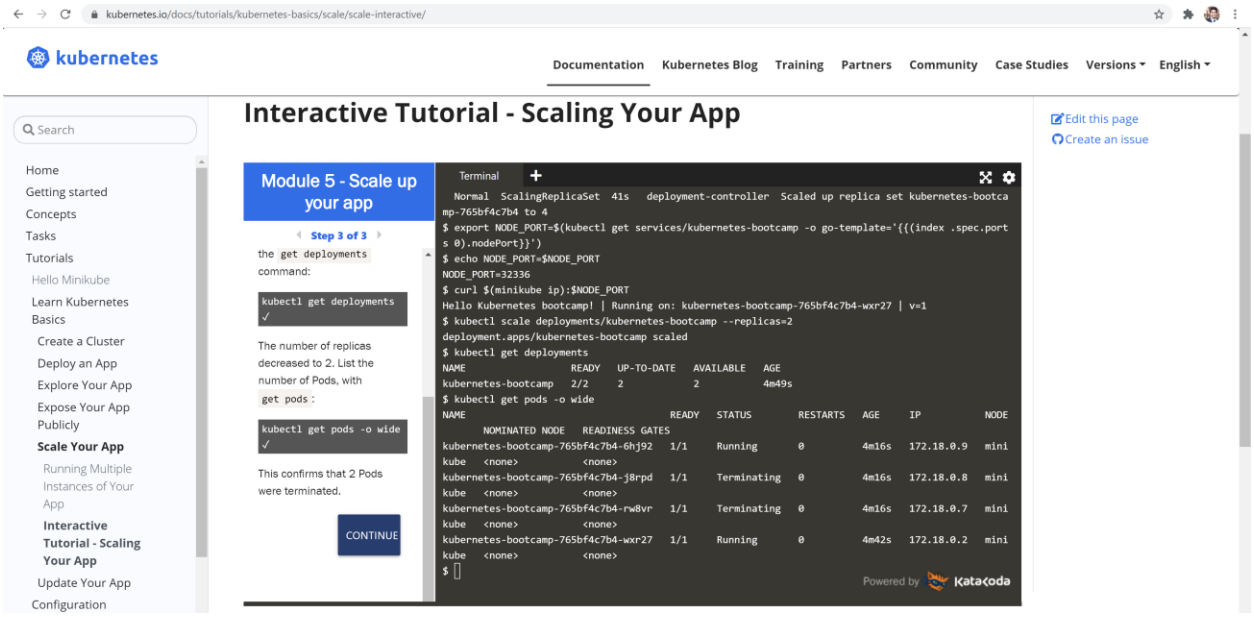



Load Balancing

Created an environment variable for storing port of the node to access the node using the curl command. Curl command reaches/hits the different pods with different IP addresses when trying to access the application showing the Load Balancing is working perfectly.




As we scaled up the application when the number of requests increased, we can scale down the application by reducing the deployment replicas when the hits can be handled by smaller number of pods. We have scaled down the application to 2 replicas from 4 and both of them are up and running as shown below.

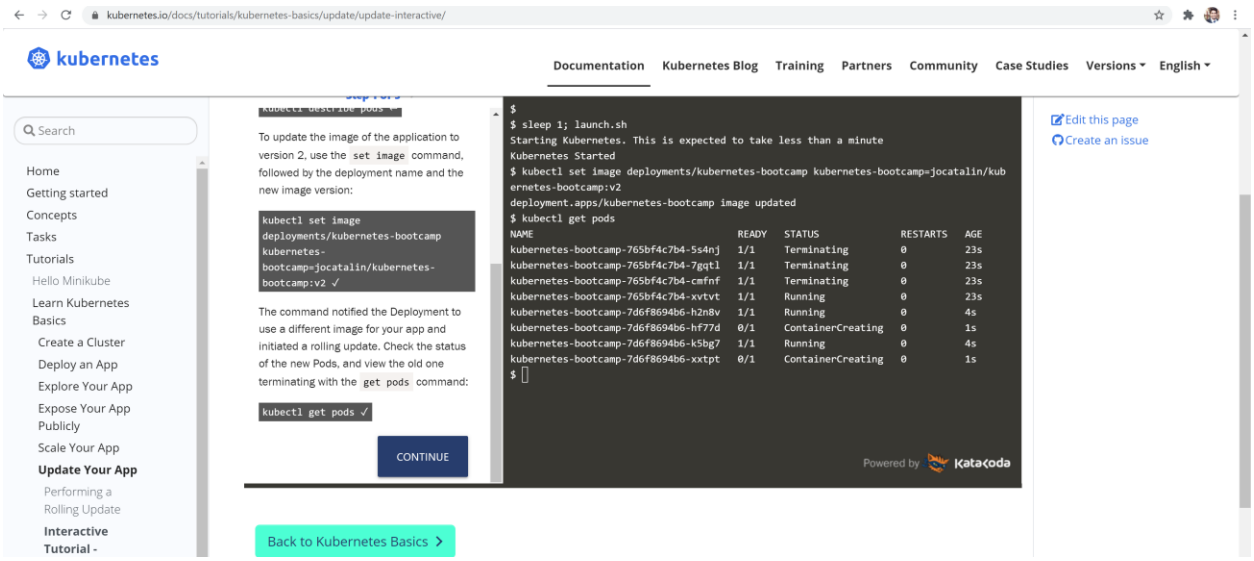



## Module 6 - Update Your App

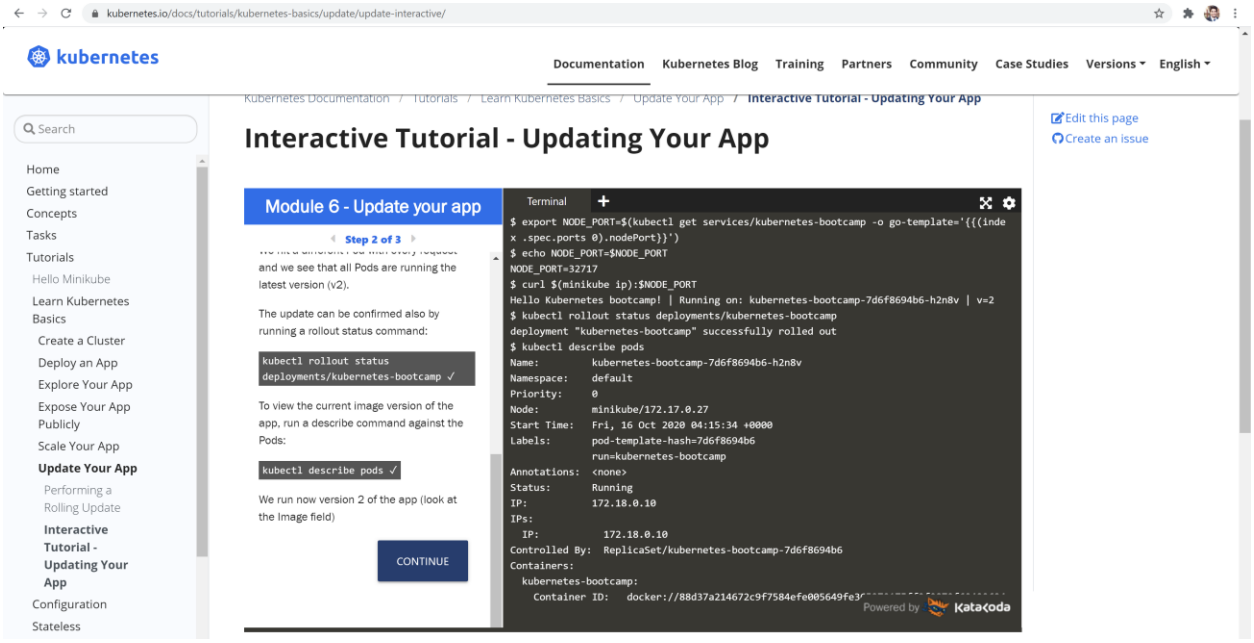
Business users want the application to be running 24/7 along with developers enhance the user experience by updating the applications to new version to fix bugs or have new functionalities without downtime. Rolling updates allow to update the pods one after other to maintain zero downtime.

Update the version of the app

We are updating the image of the application to the newer version using the “kubectl set image” command. This command is followed by name of the deployment and the new image version in this case to v2 from v1. To check the status of the new pods running and the old ones being terminated we use “get pods” command.

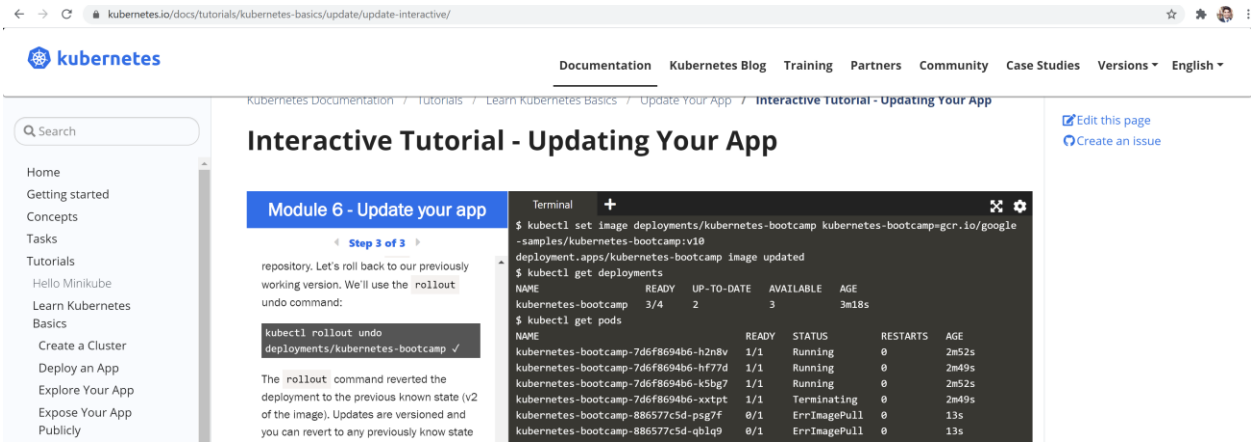


We store the node port in a variable and get the IP address, using the curl command to access the app to verify its running status. We can see that the pod is running with the newer version 2. To check the status of the successful rollout of the update, we use the command "kubectl rollout status Name of the deployment". Rolling update is done successfully as can be seen below.

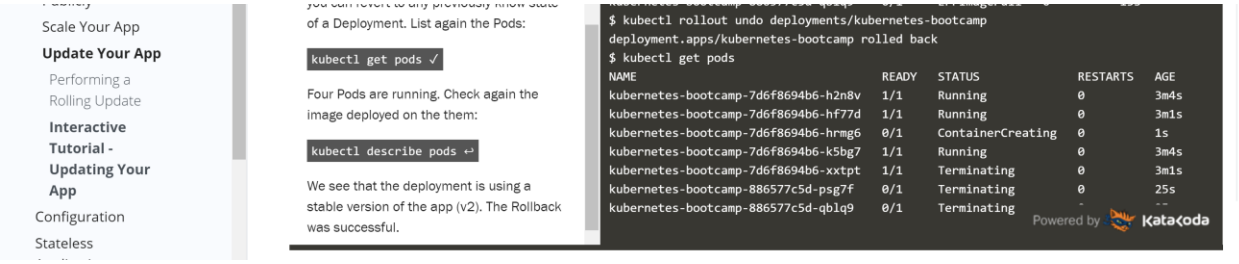



Rollback an update

We now perform another rolling update to the newer version v10. But by using the get deployments command we are not able to see the desired number of pods. Also, by checking the pods, we can see there is no update rolled out as the image for the newer update to version v10 is not found in the repository. So, rollout is not done properly as indicated.



We use the “rollout undo” command to go back to the previous version and not do this rollout update to the version v10. After the rollout undo is done to go back to the previous state, all the pods are running successfully as we can see.




**References:**

**https://kubernetes.io/docs/tutorials/kubernetes-basics/ - Kubernetes Basics Tutorial**