Kubernetes Continued

Scaling without disruption

You may be wondering at this point how Kubernetes can add or remove pods and containers whenever it wants without affecting the consumers of your application.

What happens if a customer is making a request that gets routed to a container which Kubernetes decides to terminate?

The answer to this lies in the clever use of services and load balancing.

Services

Services are the main component Kubernetes uses for routing. They act as an endpoint for DNS and as a load balancer within a cluster.

Let's say you have an API - `api.mycompany.com`and the DNS for this domain is registered with a service called API.

We create a deployment which tells Kubernetes to run 3 pods for that service.

Under the hood Kubernetes will start 3 pods, each having a single container running in them. Each of those containers will get an internal IP address for traffic to be routed to. For example let's say the containers have the IP addresses `10.0.0.1`, `10.0.0.2` and `10.0.0.3` and each container is running the application on port `5000`.

Services

The service in Kubernetes will have these 3 IP addresses registered as routing addresses for requests coming in.

The routing table will look something like this:

IP	Port
10.0.0.1	5000
10.0.0.2	5000
10.0.0.3	5000

When a request for `api.mycompany.com` makes its way to the service, the service will route it to one of the three IP address and port combinations it has registered and it will be served by a container running at that address.

Something goes wrong

Unfortunately there's a memory leak with our application, and 10.0.0.3 starts to show signs of becoming slower, unresponsive and requests start failing. Kubernetes notices this from the liveness health check. Under the hood Kubernetes will start draining any HTTP connections to `10.0.0.3`, making sure any existing ones complete without allowing new ones to start.

Once all of the HTTP connections have been finished, it will then stop routing traffic to `10.0.0.3`, instead choosing to route only to `10.0.0.1` and `10.0.0.2`.

For a short while we'll have reduced capacity as only 2 containers will be serving our traffic. However we will reduce the number of failing requests because our existing healthy containers can still serve them.

Something goes wrong

During this time the route table will look like this:

IP	Port
10.0.0.1	5000
10.0.0.2	5000
~~10.0.0.3~~ (Deleted)	~~5000~~ (Deleted)

While this is happening Kubernetes is starting up a new container with the IP address `10.0.0.4` and is waiting for it to start. Once it's readiness check has completed the IP address will be added to the routing table for the service and requests will start being routed to it.

Something goes wrong

The new routing table will then look like this:

IP	Port
10.0.0.1	5000
10.0.0.2	5000
10.0.0.4	5000

The unresponsive container `10.0.0.3` will be asked politely by Kubernetes to shut down by sending a SIGTERM command and it'll be given a grace period to stop any work it's doing and exit gracefully. If the container has still not exited after the grace period then Kubernetes will assume it has become completely unresponsive and will send it a SIGKILL instead. This will terminate the process immediately and free up any resources it was using on the cluster.

And that is how Kubernetes minimises disruption to the consumers of our applications.

Kubernetes pros and cons

Kubernetes isn't for everyone. Some companies will set up a cluster and realise that it does require a lot of effort to maintain and manage. Sometimes companies like this might be better off trying alternative solutions.

Kubernetes

Pros

- Extremely powerful
- Extensible and pluggable architecture
- Allows containerised applications to scale seamlessly

Cons

- Very complicated to set up and maintain
- Complicated to secure
- Managing the cluster can be a full time job in larger organisations
- Requires specialists to use and maintain
- Updating a kubernetes cluster can be very involved and scary

Managed vs. Unmanaged Kubernetes

One of the things that most people discover when starting to work with Kubernetes is the huge complexity involved. Kubernetes is huge, it does many, many complex things and can be very scary when first starting out.

Most companies that use Kubernetes have specialists whose sole purpose is to maintain the Kubernetes cluster. It requires lots of training to become comfortable with managing a cluster on the scale that most companies require.

Managed vs. Unmanaged Kubernetes

If you are going to try Kubernetes it can be worth trying one of the managed Kubernetes services by some of the major cloud providers, such as:

- Google Kubernetes Engine (GKE)
- Amazon Elastic Kubernetes Service (EKS)
- DigitalOcean Kubernetes

These managed clusters are pre-installed and ready to go. They can help take out some of the initial teething issues you might face when learning Kubernetes.

Minikube

Minikube is a version of Kubernetes that runs on your local machine for development purposes. It's a single node cluster that runs on Windows, Linux and MacOS.

It helps developers build applications for Kubernetes without requiring a full cluster. They can build and test locally using Minikube.

We'll be using Minikube in our practical lesson to help simulate a real Kubernetes cluster.

kubectl

Learning Objectives

After this lesson, you will be able to understand:

- What kubectl is
- How it is related to Kubernetes
- How to perform some basic commands

Introduction

kubectl is a command line tool that allows you to control Kubernetes clusters. It is the primary way that you interact and manage a cluster. You can view information and make changes, such as deployments, services, scaling and configuration through this command line tool.

The kubectl application itself is a wrapper around Kubernetes API calls, so any commands sent through it are converted to HTTP requests against the Kubernetes API running on your cluster.

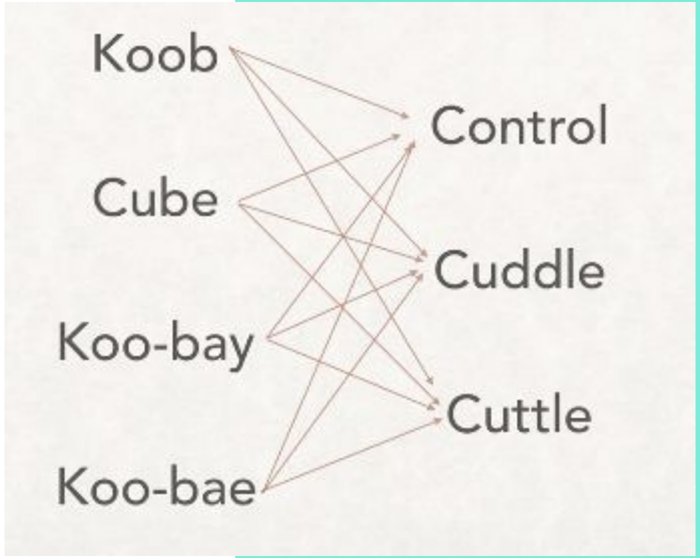
Pronunciation

As with most things in Kubernetes land, it's controversial 😄

The most common ways I've heard it pronounced are:

- *cube-control* ✅
- *cube-see-tee-ell* ✅
- *cube-cuttle* ✅

If you use these you should be fine.



Installation

You can install kubectl by following these instructions.

Configuration

Before using kubectl with a Kubernetes cluster you will need to let kubectl know where to send it's requests. These are set up with different contexts.

You won't need to do this for our lesson, but here are the instructions for future reference

Syntax

It's actually quite easy to become familiar with kubectl as you usually combine a `command` or verb, such as `get` or `describe` with the resource you wish to interact with, such as a `pod` or `service`.

Most commands follow the pattern:

```
kubectl [command] [TYPE] [NAME] [flags]
```

Common Commands

Here are a listing of common commands. We'll be doing some of these in our practical lesson so don't worry about them yet, these are just for examples:

`kubectl -h`
Get a list of available commands.

`kubectl cluster-info`
Get information about the cluster.

`kubectl version`
Get the version of kubectl you are currently running.

`kubectl apply -f /path/to/file.yml`
Applies the configuration in a yaml file against the current cluster.

`kubectl get pods`
Get all pods currently running on the cluster.

`kubectl describe pod <pod-name>`
Show detailed information about a given pod, such as it's IP address, health checks and many other things.

`kubectl get services`
Get all services currently configured on the cluster.

`kubectl describe service <service-name>`
Show detailed information about a given service.

In Conclusion

We'll be using the `kubectl` command line tool in our practical lesson with instructions, so you'll get hands-on working with it then.

Resources

- https://kubernetes.io/docs/reference/kubectl/overview/
- https://kubernetes.io/docs/reference/kubectl/cheatsheet/

Kubernetes Practical with MiniKube

Learning Objectives

After this lesson, you will be able to:

- View the state of your cluster
- Load the Kubernetes dashboard
- Deploy an application to your cluster
- Scale your application up and down
- Observe the cluster heal itself

Minikube kubectl vs Standalone kubectl

Minikube comes with it's own version of kubectl which is compatible with the currently installed version of Minikube. It will always be safe to perform kubectl commands through Minikube.

You can do this by prepending `minikube` in front of all `kubectl` commands, for example:

```
minikube kubectl <command>
```

Although you will usually be pretty safe just running kubectl against your Minikube cluster, they should be compatible. If this is preferable you can just run kubectl directly, like so:

```
kubectl <command>
```

Either way you shouldn't have problems.

Viewing kubectl configuration

If you want to use kubectl directly then by default it talks to localhost, so it will work with Minikube. To configure kubectl to talk to a remote cluster, such as your development or production cluster, then you need to configure it.

Your kubectl instance should be configured to correctly talk to the local Minikube instance so you shouldn't need to change anything. To view kubectl configuration you can run:

```
kubectl config view
```

This allows you to see the current configuration for kubectl, including clusters and contexts configured.

Getting Familiar with the Cluster

To start the local Kubernetes cluster through minikube run the following:

```
minikube start
```

This will start the Minikube container on your local machine that will manage the cluster for you.

Run `docker ps -a` and you'll see the minikube container started.

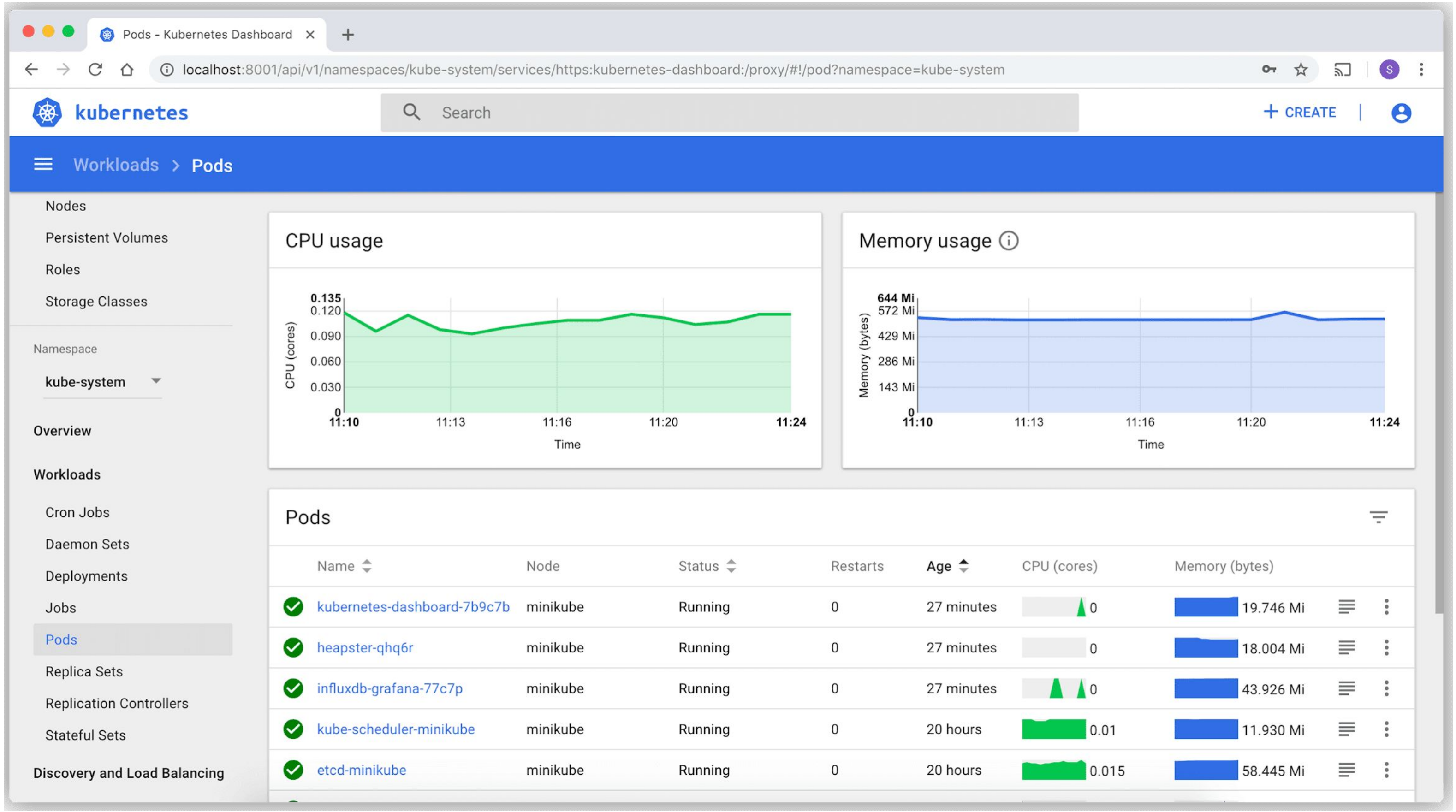
Dashboard

Once Minikube has started we can view the dashboard. Run the following command to open the dashboard:

```
minikube dashboard
```

This will open a new browser and show you what your cluster is currently doing, how many pods and services there are. Everything you need to see how your cluster is performing.

Dashboard



Pods

Name	Node	Status	Restarts	Age	CPU (cores)	Memory (bytes)
kubernetes-dashboard-7b9c7b	minikube	Running	0	27 minutes	0	19.746 Mi
heapster-qhq6r	minikube	Running	0	27 minutes	0	18.004 Mi
influxdb-grafana-77c7p	minikube	Running	0	27 minutes	0	43.926 Mi
kube-scheduler-minikube	minikube	Running	0	20 hours	0.01	11.930 Mi
etcd-minikube	minikube	Running	0	20 hours	0.015	58.445 Mi

Cluster Information

Running the following will show you some basic information about the cluster:

```
kubectl cluster-info
```

Showing Pods

Let's see what pods we have running on our cluster, it should be none at this point:

```
kubectl get pods
```

This shows us all user created pods. It doesn't show us the underlying pods and containers that Kubernetes itself uses to maintain the cluster. Kubernetes runs a number of it's own pods behind the scenes to manage the cluster too.

To view all of these pods you can use the get pods command with the `-A` flag, this will include the system ones too:

```
kubectl get pods -A
```

You can get all sorts of resources from Kubernetes just by using the `get` command. For a full listing of them use:

```
kubectl get --help
```

Before we get started

Before we get started creating a deployment it might be an idea to open a new terminal window so we can watch some of the behaviour of Kubernetes as we enter commands.

Open a new terminal window, place it somewhere visible and type the following:

```
watch kubectl get pods
```

This will repeat the command for us every 2 seconds, so we can see our cluster changing in real time as we make changes.

We'll call this the watch terminal for the remainder of the lesson and occasionally look at it to see what's happening.

Creating a deployment

Let's start a pod running on our new minikube cluster. We do this by using the `create deployment` command:

```
kubectl create deployment hello-node --image=trimhall/my-example:latest
```

While this is running, take a look at your watch terminal, what can you see? There will be a status field which can give you an indication of what each individual pod is doing.

Pod Status

The most common statuses you'll see might be the following:

Status	Description
`Pending`	Kubernetes is getting ready to initialise the pod
`ContainerCreating`	Pod is creating the underlying containers
`ErrImagePull`	Kubernetes was unable to pull the Docker image you wanted to run.
`Running`	The pod is happily running.
`Completed`	The pod ran and exited cleanly.
`CrashLoopBackOff`	The pod is failing to start correctly and crashing, Kubernetes will keep trying to start it but will use an exponential backoff.
`Terminating`	The pod is being shut down

Getting the Pods

Once you've created the deployment you can use the get pods command to view it.

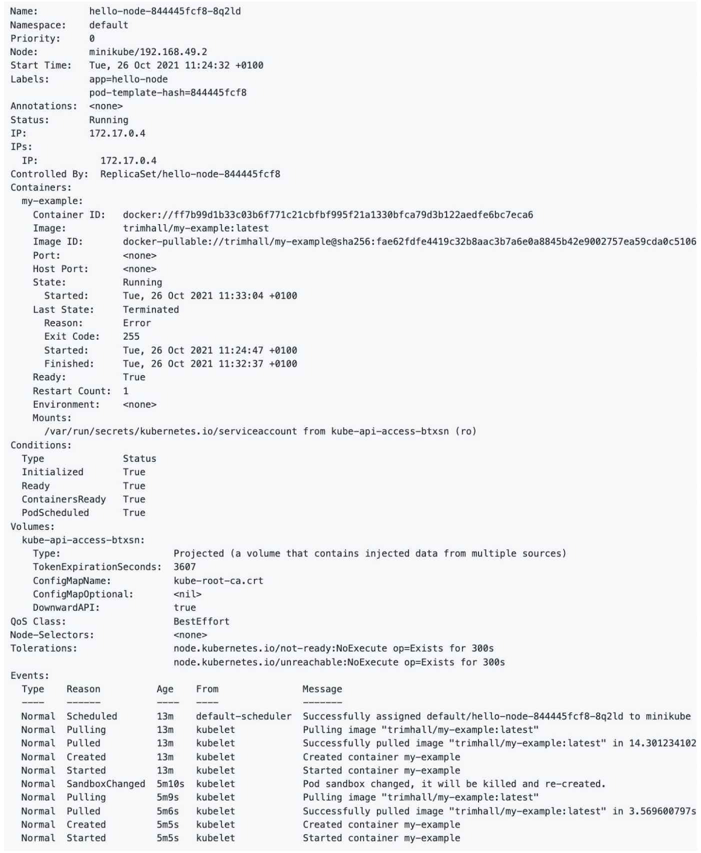
```
kubectl get pods
```

You should see something like the following:

```
NAME                           READY    STATUS     RESTARTS    AGE

hello-node-844445fcf8-8q2ld    1/1      Running    0           4m19s
```

Congratulations! You’ve now got a running pod with a container



```
Name:         hello-node-844445fcf8-8q2ld
Namespace:    default
Priority:     0
Node:         minikube/192.168.49.2
Start Time:   Tue, 26 Oct 2021 11:24:32 +0100
Labels:       app=hello-node
              pod-template-hash=844445fcf8
Annotations:  <none>
Status:       Running
IP:           172.17.0.4
IPs:
  IP:           172.17.0.4
Controlled By:  ReplicaSet/hello-node-844445fcf8
Containers:
  my-example:
    Container ID:   docker://ff7b99d1b33c03b6f771c21cbfbf995f21a1330bfca79d3b122aedfe6bc7eca6
    Image:          trimhall/my-example:latest
    Image ID:       docker-pullable://trimhall/my-example@sha256:fae62fdfe4419c32b8aac3b7a6e0a8845b42e9002757ea59cda0c5106
    Port:           <none>
    Host Port:      <none>
    State:          Running
      Started:      Tue, 26 Oct 2021 11:33:04 +0100
    Last State:     Terminated
      Reason:       Error
      Exit Code:    255
      Started:      Tue, 26 Oct 2021 11:24:47 +0100
      Finished:     Tue, 26 Oct 2021 11:32:37 +0100
    Ready:          True
    Restart Count:  1
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-btxsn (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  kube-api-access-btxsn:
    Type:                    Projected (a volume that contains injected data from multiple sources)
    TokenExpirationSeconds:  3607
    ConfigMapName:           kube-root-ca.crt
    ConfigMapOptional:       <nil>
    DownwardAPI:             true
QoS Class:                   BestEffort
Node-Selectors:              <none>
Tolerations:                 node.kubernetes.io/not-ready:NoExecute op=Exists for 300s
                             node.kubernetes.io/unreachable:NoExecute op=Exists for 300s
Events:
  Type    Reason          Age    From               Message
  ----    ------          ----   ----               -------
  Normal  Scheduled       13m    default-scheduler  Successfully assigned default/hello-node-844445fcf8-8q2ld to minikube
  Normal  Pulling         13m    kubelet            Pulling image "trimhall/my-example:latest"
  Normal  Pulled          13m    kubelet            Successfully pulled image "trimhall/my-example:latest" in 14.301234102
  Normal  Created         13m    kubelet            Created container my-example
  Normal  Started         13m    kubelet            Started container my-example
  Normal  SandboxChanged  5m10s  kubelet            Pod sandbox changed, it will be killed and re-created.
  Normal  Pulling         5m9s   kubelet            Pulling image "trimhall/my-example:latest"
  Normal  Pulled          5m6s   kubelet            Successfully pulled image "trimhall/my-example:latest" in 3.569600797s
  Normal  Created         5m5s   kubelet            Created container my-example
  Normal  Started         5m5s   kubelet            Started container my-example
```

Let's have a closer look at the pod to see more information about it:

```
kubectl describe pod <pod-name>
```

In the above example we'd type:

```
kubectl describe pod
hello-node-844445fcf8-8q2ld
```

The results of this display lots of details about the pod, including what containers are running, it's IP address and any events that have happened

Exposing Ports

This pod is quietly running in our cluster without being exposed to the world. Let's expose the ports through Kubernetes so we can view it in our browser:

```
kubectl expose deployment hello-node --type=NodePort --port=3000
```

This exposes the port within the Kubernetes network and creates a service.

Exposing Ports

You can confirm this by using the following command to see that a service has been added:

```
kubectl get services
```

It'll list something like this:

```
NAME         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
hello-node   NodePort    10.107.211.49   <none>        3000:32201/TCP   10m
kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP          23m
```

These results contain both the kubernetes internal service and our newly created service.

Taking a Closer Look

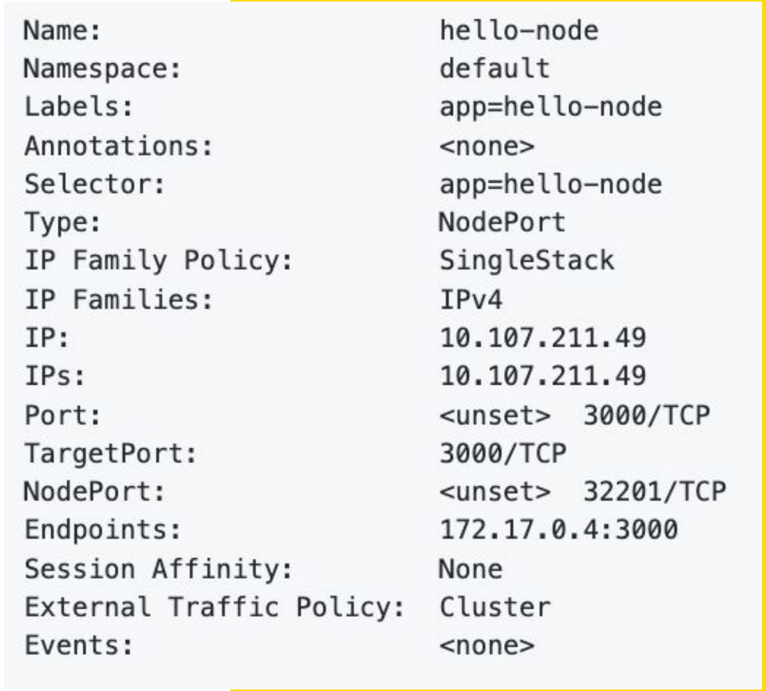
Let's have a closer look at the service:

```
kubectl describe service hello-node
```

You'll notice we're using `NodePort` type here. Within the internal Kubernetes network this service has an IP address. Any requests get routed to that internal IP.

Even though we're exposing the port on that service (`3000`) it's still only exposed on that internal IP `10.107.170.101`.

Results:



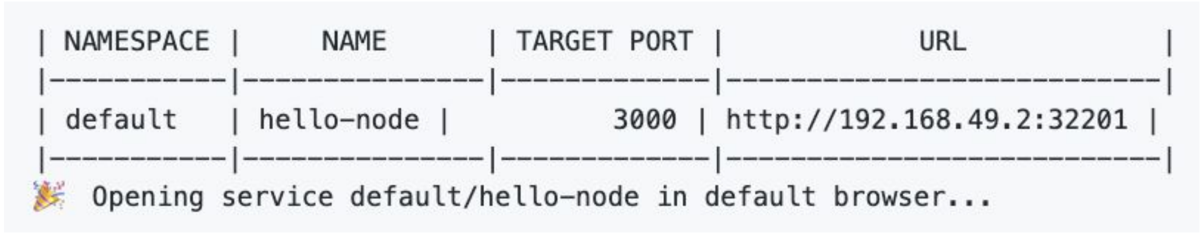
```
Name:                     hello-node
Namespace:                default
Labels:                   app=hello-node
Annotations:              <none>
Selector:                 app=hello-node
Type:                     NodePort
IP Family Policy:         SingleStack
IP Families:              IPv4
IP:                       10.107.211.49
IPs:                      10.107.211.49
Port:                     <unset>  3000/TCP
TargetPort:               3000/TCP
NodePort:                 <unset>  32201/TCP
Endpoints:                172.17.0.4:3000
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
```

Taking a Closer Look

To view the application running within that pod we need to proxy requests through minikube. To do this run the following command:

```
minikube service hello-node
```

You'll see something like this:



```
| NAMESPACE |     NAME      | TARGET PORT |            URL            |
|-----------|---------------|-------------|---------------------------|
| default   | hello-node |          3000 | http://192.168.49.2:32201 |
|-----------|---------------|-------------|---------------------------|
🎉  Opening service default/hello-node in default browser...
```

This proxies requests through minikube to the service inside your cluster. You can open that URL in your browser to view the running website!

Scaling Up

Now we have our application running in Kubernetes we might want to scale it up so we're running more than one instance of the application.
Let's increase the number of pods (and containers) to 3:

```
kubectl scale deployments/hello-node --replicas=3
```

Then to see it in action, run:

```
kubectl get pods
```

In here you should now see 3 pods up and running:

```
NAME                            READY    STATUS     RESTARTS     AGE
nginx-example-5bf5d67944-8kz54  1/1      Running    0            72s
nginx-example-5bf5d67944-gtwg4  1/1      Running    0            29m
nginx-example-5bf5d67944-pc67g  1/1      Running    0            72s
```

Scaling Up

```
NAME                              READY   STATUS    RESTARTS   AGE
nginx-example-5bf5d67944-8kz54    1/1     Running   0          72s
nginx-example-5bf5d67944-gtwg4    1/1     Running   0          29m
nginx-example-5bf5d67944-pc67g    1/1     Running   0          72s
```

You can see the newer pods started by the AGE column.

Go crazy, you can add as many replicas as you want, provided your machine has the available resources.

Here you can refresh your browser and your request will be routed to different containers underneath the service. Kubernetes is load balancing between the 3 different pods and containers you have running.

Each time you refresh your browser you should see the hostname change to the name of the Docker container running.

Scaling Down

Now let's say we want to scale back down, the number of users have tailed off and we don't need to have so many containers running. We can use the same command to scale back down, just set the replicas to `1` instead.

```
kubectl scale deployments/nginx-example --replicas=1
```

If you look at our watch terminal after executing this command you'll see the statuses of some of our pods turn to `Terminating`.

Scaling Down

Refresh your browser a few times. You shouldn't notice any difference. All the requests are still being processed successfully and you're not being given any errors, even though the containers are being terminated under the hood.

Kubernetes is gracefully terminating all the containers.

First it ensures that traffic doesn't get routed to those that are being torn down, so your requests only hit pods that are running. Then it'll tell those containers to exit nicely.

Keep Refreshing

Keep refreshing the web page while it scales down, you'll notice that none of the requests fail. This is because Kubernetes is routing all the requests to the working containers under the hood.

If they shut down nicely within 30 seconds they'll disappear from our listing. If they don't they will be forcibly killed by Kubernetes to free up resources.

Now check the watch window and you should only have 1 pod running now:

```
NAME                              READY    STATUS     RESTARTS    AGE
nginx-example-5bf5d67944-gtwg4    1/1      Running    0           53m
```

You've successfully scaled back down!

Auto Healing

Now that we've deployed an application to Kubernetes, successfully scaled up and down, let's see how it handles a container exiting through a simulated error.

It's a bit tricky to directly access running containers on our cluster through the Docker Engine when we're using Minikube and Kubernetes. Minikube kind of runs the containers within its own container instance.

Before starting, take a note of the name of the pod running your application now. We'll need it to compare against later.

In order to get access to our running container, we need to first jump into the Minikube container so we can then execute Docker commands and see what the cluster is running.

Let’s Jump into the Minikube container first

First find what container ID Minikube has. To do this run:

```
docker ps
```

Find the container ID for the minikube cluster. Then type:

```
docker exec -it <container-id> /bin/sh
```

This will open a shell prompt inside our Minikube Docker container. We've got access to the operating system running inside of the Minikube container.

See What the Cluster is Running

Now that we're *in* the cluster, we can see what containers we have running. Again run the following command:

```
docker ps -a
```

Find your container ID in the listing. Now we want to forcibly kill this container, simulating an application exiting ungracefully so that we can see what Kubernetes does.

Let's kill it:

```
docker rm -f <container-id>
```

Now We Watch

Watch what happens in your watch terminal. The old pod that you recorded before should have disappeared. The underlying container has exited and Kubernetes has removed the pod.

But you should have a shiny new one in it's place!

That is Kubernetes automatically starting a new pod to maintain the desired state of 1 pod. Fantastic!

Bonus Steps

Want to have a look *inside* that single container you've got running in your Kubernetes cluster? Let's do it. We can jump into a running container as if we were SSH'ing to another machine. This time we don't need to go via the Minikube container. We can just do:

```
kubectl exec -it <pod-name> /bin/sh
```

(Note: If this doesn't work you can use `/bin/bash` on the end instead, it depends what shell script is running on the underlying container operating system.)

Wrap-ups

Discussion:

Post-Lab

When you've completed the lab, we'll discuss:

- What surprised you?
- What challenged you?
- What do you want to learn more about?

Next Steps

Why not try signing up for one of the following managed Kubernetes services? These can be easier and quicker to set up compared to building your own cluster.

Check them out:

- Google Kubernetes Engine (GKE)
- Amazon Elastic Kubernetes Service (EKS)
- DigitalOcean Kubernetes

Resources

- https://minikube.sigs.k8s.io/docs/start/