Kubernetes in Depth
Lab 1
Core Kubernetes objects

1 References and cheat sheets

The official reference for all `kubectl` commands:
https://kubernetes.io/docs/reference/kubectl/quick-reference/

https://spacelift.io/blog/kubernetes-cheat-sheet
https://www.bluematador.com/learn/kubectl-cheatsheet
https://www.geeksforgeeks.org/kubectl-cheatsheet/

2 Lab setup

You should have basic Kubernetes installed through 2 possible approaches:

Approach a: Minikube (together with Docker Desktop)

Approach b: Rancher Desktop

The root folders for all the various projects here can be found in the `Lab 1` subfolder in the main `labcode` folder of your downloaded zip for this workshop.

Create a dedicated directory (e.g. `C:\kuberneteslabs`) on your local machine to hold all the source code files and YAML manifests that we will be creating from now onwards.

3 Starting a Kubernetes cluster

There are a variety of ways to install a Kubernetes cluster. For this workshop, we will use either Minikube or Rancher Desktop

3.1 Starting up with Minikube

Minikube is a lightweight tool that allows developers to run a **single-node Kubernetes cluster** on the local machine. It is specifically designed for testing, development, and experimentation with Kubernetes without requiring a full-scale multi-node cluster.

For Minikube installation, when starting it, you will need to specify a driver to allow it to be deployed accordingly as either a VM, a container or bare metal.

The preferred approach on Windows would be to start Minikube using the Docker driver, which we will do for the remaining lab sessions

Make sure that you have Docker Desktop running before attempting to start it.

In a PowerShell terminal, start Minikube's single node Kubernetes cluster with:

```
minikube start --driver=docker
```

When you start Minikube with the Docker driver, Minikube creates a Docker container that acts as a single-node Kubernetes cluster. Within this container, Kubernetes manages pods as Docker containers.

Check the status of the cluster with:

```
minikube status
```

Check that the cluster nodes are running with:

```
kubectl get nodes
```

```
NAME       STATUS   ROLES           AGE   VERSION
minikube   Ready    control-plane   63m   v1.30.0
```

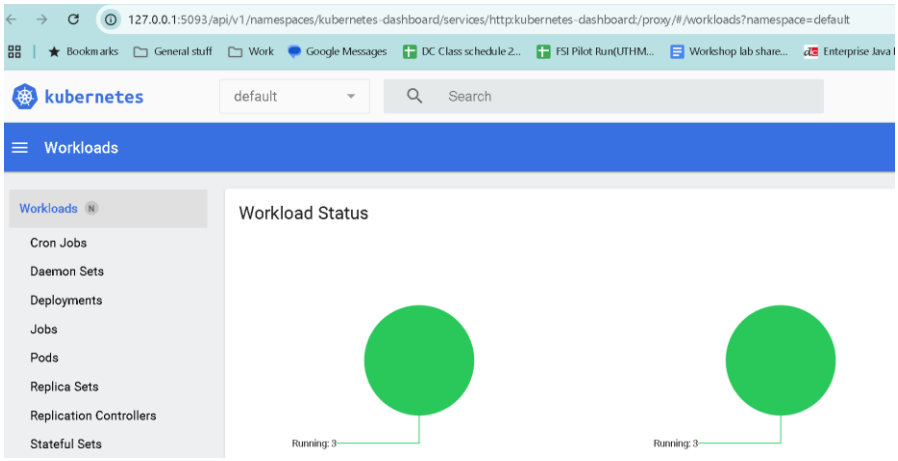
Notice that the Minikube single node cluster is running within a Docker container with a specific configuration and port mapping:

```
docker ps
```

Finally, you can start the built-in dashboard UI that allows you to monitor your various Kubernetes objects (pods, ReplicaSets, Deployments, Services) as an alternative to checking on these objects using the standard `kubectl` commands that we will be working with later.

```
minikube dashboard
```

This should start the Kubernetes dashboard as a service within your Minikube cluster, and will also open a new browser tab to access the dashboard UI, via a random free port available on localhost. The dashboard runs as a web application backed by the Kubernetes API server. When you interact with the dashboard, it communicates directly with the Kubernetes API server to fetch data or apply changes.



You will need to open a new PowerShell terminal to continue typing `kubectl` commands.

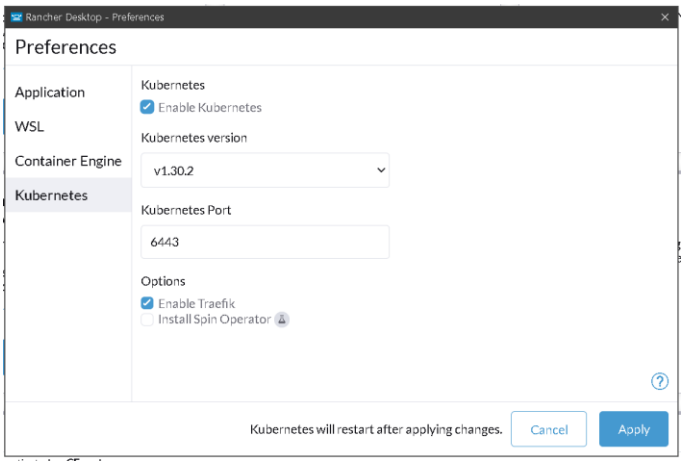
When you are complete with your lab session for the day, you can shut down the Minikube single-node cluster with:

```
minikube stop
```

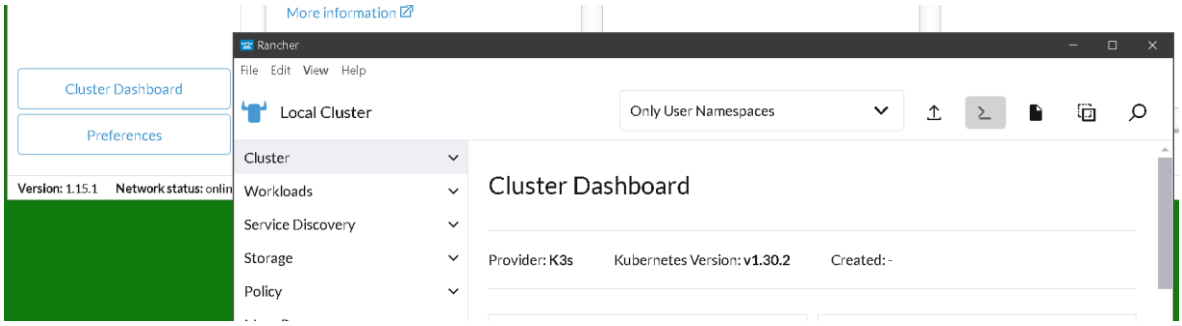
The next time you start it again, all the existing workload objects (such as pods, replicasets, deployments and services) that you had which were active in it prior to shut down will be restarted and reactivated again for use.

3.2 Starting with Rancher Desktop

Rancher Desktop already comes preinstalled with Kubernetes based on K3s, which by default should be already enabled. However, if you had disabled Kubernetes while working on your Docker labs, then remember to enable it again via preferences:



Once it is up and running you should be able to access the cluster dashboard in a separate UI window from the option at the bottom of the main pane:



This will allow you to monitor your various Kubernetes objects (pods, ReplicaSets, Deployments, Services) as an alternative to checking on these objects using the standard `kubectl` commands that we will be working with later.

Check that the cluster nodes are running by typing into a PowerShell terminal:

```
kubectl get nodes
```

```
NAME        STATUS   ROLES                  AGE   VERSION
machinename Ready    control-plane,master   63m   v1.30.0
```

4 Objects and workload resources

Objects are persistent entities used to describe the desired state of your cluster (record of intent), which includes info such as which containerized applications are running and on which nodes and the resources available to those applications. Once an object is created, Kubernetes will ensure it is persisted and try to configure the cluster state to reflect the object description

Objects are managed via the Kubernetes API, primarily through the `kubectl` CLI tool which makes the API calls. The most commonly used objects are:

- Pods (the basic unit of computing in Kubernetes)
- ReplicaSets
- Deployments

- Services
- Volumes
- Namespaces

Among all these objects, workload resource objects are specifically used for managing pods instead of managing them directly. Examples include: Deployment and ReplicaSet, StatefulSet, etc

5 Intro to pods

Pods are the smallest deployable unit of computing in Kubernetes. A pod contains one more containers with shared storage (volumes) and network resources. They run on worker nodes and have their own IP addresses

The most common model is one-container-per-Pod: where the Pod acts as a wrapper around a single container. It is also possible to have multiple containers within a Pod: if these containers are closely coupled and need to share network / storage resources (advanced use case)

Kubernetes will by default pull images from a DockerHub repo (either official repo or custom user repo) to create a container within a single pod. Here, we will create a single pod running a single container based on the image from the official DockerHub nginx repo:

You can execute all these `kubectl` CLI commands in a PowerShell terminal in a folder that you have created specifically for these labs:

```
kubectl run my-pod --image=nginx
```

Once the pod and the container within it is created, Kubernetes will determine the most suitable worker node in the cluster to place the pod on, depending on distribution of resources in the cluster. At the moment, we are only working with a single node cluster, so there is only one node available to run that pod on.

You can check on the status of all running pods with:

```
kubectl get pods
```

When you start a pod, you will initially see a showing the status of the pod as ContainerCreating before it eventually transitions to Running, if you run the command above fast enough.

You can get slightly more detailed info on all running pods with:

```
kubectl get pods -o wide
```

The columns and their meanings:

a) NAME: The name of the pod, which is my-pod. This is a unique identifier for the pod within the namespace.
b) READY: Indicates the number of containers in the pod that are ready compared to the total number of containers in the pod. 1/1 means there is one container in the pod, and it is ready.
c) STATUS: The overall status of the pod. Running means the pod is up and functioning as expected.

d) RESTARTS: The number of times the containers in the pod have been restarted due to failures or other issues.
e) AGE: The amount of time that has passed since the pod was created.
f) IP: The internal IP address assigned to the pod within the Kubernetes cluster network.
g) NODE: The name of the Kubernetes node where the pod is running on
h) NOMINATED NODE: Used for scheduling purposes in scenarios where a pod is waiting to be scheduled to a specific node.
i) READINESS GATES: Represents additional checks or conditions that must be satisfied for the pod to be considered "ready" beyond the standard readiness probes.

To obtain more descriptive info on any active running pod, reference the pod name:

```
kubectl describe pod my-pod
```

Some of the relevant info from this description include name, IP address, node that the pod is running on, image, state, and the list of events within the pod up to the current point of time. These typically indicates relevant activities such as starting the pod, pulling the relevant image from a DockerHub repo and starting a container from this image within the pod.

```
Name:             my-pod
Namespace:        default
Node:             minikube/192.168.49.2
IP:               10.244.0.9
Image:          nginx
State:          Running
Ready:          True
......
Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  47s   default-scheduler  Successfully assigned default/my-pod to ganesha
  Normal  Pulling    46s   kubelet            Pulling image "nginx"
  Normal   Pulled      43s   kubelet              Successfully pulled image "nginx" in 2.623s
(2.623s including waiting). Image size: 187694648 bytes.
  Normal  Created    43s   kubelet            Created container my-pod
  Normal  Started    43s   kubelet            Started container my-pod
```

You can remove a running pod by referencing the pod name again:

```
kubectl delete pod my-pod
```

Check that it has been successfully deleted with:

```
kubectl get pods
```

5.1 Using manifest to create a pod

There are 2 main ways to create objects (pods, deployments, services, volumes, etc):

a) The imperative approach, that involves typing CLI commands such as `kubectl run` to specify the objects that are to be created and their related options (demonstrated earlier)

b) The declarative approach using a manifest configuration YAML file. This is conceptually equivalent to Docker Compose files that allow us to specify a number of services / containers that are started simultaneously and configured in a single file.

An example of a manifest to declaratively create a pod:

Create a dedicated directory on your local machine to hold all the configuration manifests that we will be creating from now onwards.

Create the configuration manifest file below in this directory. We will use this to generate the same pod that we just created

`pod.yaml`

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  labels:
    env: production
    tier: frontend
spec:
  containers:
  - name: my-pod-container
    image: nginx
    ports:
      - containerPort: 80
```

Some of the key sections in this YAML:

- `apiVersion`: Specifies the Kubernetes API version. This is usually v1 for basic objects such as pods
- `kind`: Specifies the type of Kubernetes object being created: pod, deployment, ReplicaSet, service, etc
- `metadata` section - contains information about the object, such as its name and labels.
- `spec` section - this outlines the specification of the object in detail, which will differ between objects. For the case of a pod, this will be the containers that the pod runs, which includes the images the containers are generated from, ports they use, volume mounts, environment variables, etc.

A core part in the `metadata` section for all objects (particularly pods) are the labels. Labels are user-defined key-value pairs attached to objects. They act like tags that provide a way for users to organize and group resources in a Kubernetes cluster.

Key points on labels:

a) Labels can be attached to objects at creation time, and subsequently modified.
b) You can attach one or more labels to a single object (in the example above, we have 2).
c) Labels are not unique (unlike `name`), and it is normal to have multiple objects with identical labels in a large cluster. For e.g. the 2 labels that we have above might be attached to all objects that implement the front-end portion of a web app functionality on a production server.

d) Labels are meant to be used with selectors from objects such as Deployments and ReplicaSets

Generate a new pod based on the configuration manifest with:

```
kubectl apply -f pod.yaml
```

Check that it is started with:

```
kubectl get pods
```

As before, you can get more info on it with:

```
kubectl describe pod my-pod
```

You should be able to see the info provided here is nearly identical to the case when we generated the pods using the imperative approach, with some minor differences: for e.g. the container here has its own name explicitly specified in the YAML : `my-pod-container` which is distinct from the name of the pod: `my-pod`

You can also obtain the logs from the application (`nginx`) running in the container:

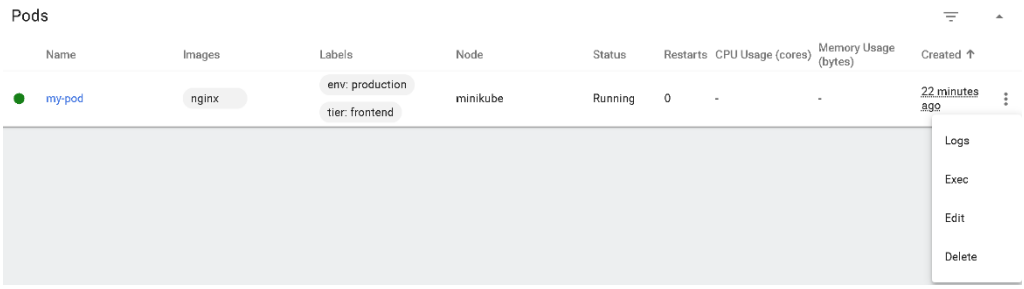
```
kubectl logs my-pod -c my-pod-container
```

You can also open an interactive shell terminal into this running container, in very similar way to what we have done with Docker commands:

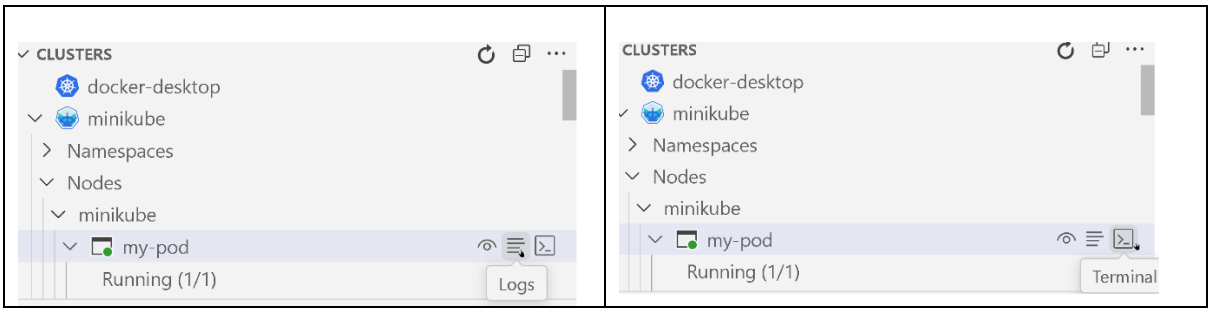
```
kubectl exec my-pod -c my-pod-container -it -- /bin/bash
```

You can then type standard Linux commands in this shell

Both the logs and terminal shell in the container can also be obtained in the Minikube dashboard in the Pods section



It is also accessible via the Kubernetes extension in VS Code.



Finally, you can delete the pod directly by referencing its name as we did before:

```
kubectl delete pod my-pod
```

or you can reference the YAML specification that was originally used to create it:

```
kubectl delete -f pod.yaml
```

You can also manage various activities related to pods (creation, configuration, deletion, etc) from the specific dashboard areas (Minikube / Rancher Desktop) or VS Code Kubernetes extension.

6 Intro to ReplicaSets

A ReplicaSet (RS) is an object that ensures there is always a stable set of running pods for a specific workload. The ReplicaSet configuration defines a specific number of identical pods required, and works to ensure that this number is maintained. This means:

- If a pod is evicted or fails, it creates more pods to compensate for the loss.
- If there are too many pods created, it will remove the excess pods

Typically, we will not create ReplicaSets directly, they are usually autogenerated as part of the creation of a Deployment object, which is the primary way in which we manage pods in Kubernetes. However, we will explicitly create one below separate from a Deployment object to demonstrate their functionality clearly.

An example of a manifest to declaratively create a ReplicaSet is shown at:
https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/#example

Create a configuration manifest file in an empty directory. We will use this to generate a ReplicaSet.

`replicaset.yaml`

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myapp-replicaset
  labels:
    usage: controlProduction
spec:
  selector:
    matchLabels:
      env: production
  replicas: 3
  template:
    metadata:
```

```
    name: my-pod
    labels:
      env: production
      tier: frontend
  spec:
    containers:
    - name: my-pod-container
      image: nginx
      ports:
        - containerPort: 80
```

Some of the key sections in this YAML are identical to that of the pod. There are some differences as well as additional elements:

- `apiVersion`: Notice that this is `apps/v1` for a ReplicaSet (which is a workload resource used to manage a pod), vs just `v1` for the Pod earlier.
- `selector`: Specifies a label query used to identify the set of Pods that this ReplicaSet manages.
- `matchLabels`: A map of key-value pairs for matching labels. Any pods that have these key value pairs as their labels will come under the control of this particular ReplicaSet.
- `replicas`: The required number of pod replicas to be maintained at all times. If omitted, defaults to 1.
- `template`: Contains the pod template for creating new pods. The elements that follow after this are exactly the same as the ones found in the our previous YAML manifest for specifying a pod `pod.yaml` (for e.g. `metadata, name, spec, containers, image, ports`, etc)

An important point to note here is that typically the `template` will contain a label (in this example, `env: production`) that is matched by an identical `selector matchLabels` key value pair. This means that this ReplicaSet will try to ensure that exactly 3 pod replicas with the specs identified in that template are always running at any one time.

Labels and selectors are key to using pods and workload resources that manage them (such as ReplicaSets and Deployments).

Create the ReplicaSet with:

```
kubectl apply -f replicaset.yaml
```

NOTE: Sometimes, you will also see the `kubectl create` command being used with a configuration manifest instead of `kubectl apply`. This is fine if the configuration manifest is being used for the first to create the resource. Subsequently, after that you will need to use `kubectl apply` to update configuration changes to that existing resource.

Get info on all available ReplicaSets and their associated pod replicas:

```
kubectl get replicaset
```

The relevant info here would be the desired number of pod replicas vs the number that are actually ready. It will take some time here for all 3 pod replicas to become ready from the time when the configuration is applied to generate the pods.

The relevant info here for the columns are:

- NAME: The name of the ReplicaSet.
- DESIRED: The number of pods specified in the ReplicaSet's `spec.replicas` field, which is the desired number of pods the ReplicaSet controller is instructed to always try to maintain.
- CURRENT: The actual number of pods currently managed by the ReplicaSet (not necessarily running). This number can differ from DESIRED during scaling or while the cluster resolves issues with pods.
- READY: The number of pods that are in the Running state and have passed their readiness probes (if defined). Readiness probes are used to determine if a pod is ready to accept traffic or work.
- AGE: The amount of time since the ReplicaSet was created.

To get more detailed info on a specific ReplicaSet

```
kubectl describe replicaset myapp-replicaset
```

Here you will see relevant info that was also specified in the YAML manifest, such as the selector and labels for both the ReplicaSet and the Pod template, the containers for the Pod replicas and the events pertaining to the creation of these pod replicas by the ReplicaSet controller.

When we check on the pods:

```
kubectl get pods
```

You will see each of the pod replicas have a random number sequence appended to the ReplicaSet name in order to uniquely identify each of them and also indicate that they belong to the same ReplicaSet.

Try deleting any one of these pod replicas

```
kubectl delete pod myapp-replicaset-xxxx
```

And check again on the running pods after a short while:

```
kubectl get pods
```

A new pod replica will be generated to replace the deleted one (it has a new unique name different from the one that was just deleted). The ReplicaSet object will always automatically try to ensure that the number of pod replicas is equal to that in its `spec.replicas` field and the pods to be replicated have `labels` that match those specified by its `selector.matchLabels` key value pair. This is key functionality of the ReplicaSet.

Lets now create new pod with a random name based on the same image `nginx`:

```
kubectl run another-pod --image=nginx
```

If we check on the running pods after a short while:

```
kubectl get pods
```

We should see 4 pods now running. Now if we check the detailed info for this newly created pod:

```
kubectl describe pod another-pod
```

You will notice its label is `run=another-pod` (this is a default label automatically generated when none is provided). Since this label does not match the `selector.matchLabels` key value pair of the ReplicaSet (`env : production`), the ReplicaSet ignores this pod in enforcing its functionality.

Let's try to generate a new pod using the previous manifest that we had for our pod: `pod.yaml`

```
kubectl apply -f pod.yaml
```

And if we check again immediately with

```
kubectl get pods
```

we will see that the newly created pod was terminated as soon as it was started

The key point to note here is that the key value pair for the `labels` in the YAML manifest `pod.yaml` matches the key value pair for the `selector matchLabels` section of the ReplicaSet, , which means that the pod generated from the application of this YAML will be managed or targeted by this ReplicaSet. Since there are already 3 pod replicas with this label already running, any extra pod replicas will immediately be terminated by the ReplicaSet to ensure that the number constantly stays at 3.

If we check the event history log of the ReplicaSet

```
kubectl describe replicaset myapp-replicaset
```

You will see the last event was the deletion of that newly created pod

To change the number of replicas to a higher number in order to scale the cluster:

```
kubectl edit replicaset myapp-replicaset
```

This opens up a copy of the original YAML file in the default editor of the system (for Windows, this will be Notepad). This copy will not affect the original YAML file contents and any changes made here will immediately be applied to the ReplicaSet. Change the key value pair `replicas : 3` to `replicas : 4` and save and close the editor

Now check the number of running pods again with:

```
kubectl get pods
```

You will see 4 pod replicas under the management of that ReplicaSet.

Another way to change the number of replicas is via with:

```
kubectl scale replicaset myapp-replicaset --replicas=2
```

Now check the number of running pods again with:

```
kubectl get pods
```

To delete an existing ReplicaSet, you can either explicitly specify the ReplicaSet based on its name:

```
kubectl delete replicaset myapp-replicaset
```

or

or use the manifest file that was originally used to create it with:

```
kubectl delete -f replicaset.yaml
```

You can also delete the one extra last running pod:

```
kubectl delete pod another-pod
```

7 Intro to Deployments

Although it is possible to work with pods directly and also via ReplicaSets, most commonly we will use Kubernetes workload resources to manage these pods. The most commonly used workload resource for this purpose is a Deployment, and there are common scenarios for which a Deployment is ideal for.

The root folder for this project is: `kub-action`

The main app for this project (`app.js`) simply returns some basic HTML content and will intentionally crash the app (`process.exit(1)`) when a HTTP request is send to the `/error` endpoint (which we will make use of later in a later lab to demonstrate deployment rollback).

In a PowerShell terminal in this root project folder, build a custom image using the Dockerfile:

```
docker build -t kub-first-app .
```

Up to this point of time, the pods we have created encapsulate containers generated from base images from official DockerHub repos. In real life projects, we will often use pods with containers from our custom images generated through a Dockerfile like the example above.

Check that the image has been created correctly in the local Docker registry via the Docker Desktop / Rancher Desktop UI or by typing:

```
docker images
```

Let's attempt to create a new deployment from the CLI using this custom image directly with a command:

```
kubectl create deployment first-app --image=kub-first-app
```

and check for it:

```
kubectl get deployments
```

To check the reason why it is failing, type:

```
kubectl get pods

kubectl describe pod first-app
```

Notice that the pod has the same name as the Deployment name, but with an additional random sequence (a hash) appended at the end (e.g. `first-app-xxxxx`). This hash will be unique for each individual pod that is associated with the Deployment (at the moment there is only one)

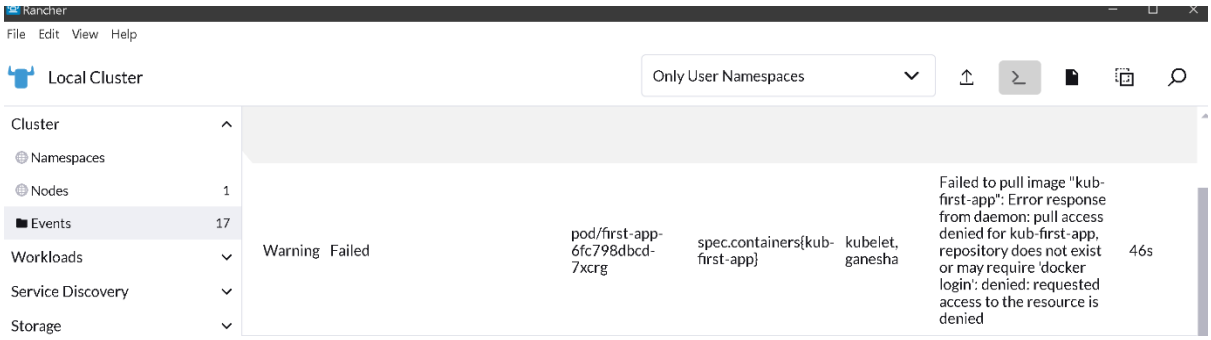
Note: You can also get all this status info from the dashboard (Minikube / Rancher Desktop) that you started earlier.

On Minikube

Events

	Name	Reason	Message	Source
	first-app-6fc798dbcd-sld42.17eb91f2c09737(	BackOff	Back-off pulling image "kub-first-app"	kubelet minikube
●	first-app-6fc798dbcd-sld42.17eb91f2c097f94	Failed	Error: ImagePullBackOff	kubelet minikube
●	first-app-6fc798dbcd-sld42.17eb91f28c2623(	Failed	Failed to pull image "kub-first-app": Error response from daemon: pull access denied for kub-first-app, repository does not exist or may require 'docker login': denied: requested access to the resource	kubelet minikube

On Rancher Cluster Dashboard, in the Events area, sort on Last Seen



NOTE: There is a bug in the sorting of events in the Rancher Desktop dashboard, which is based on alphabetical sequence of letters of the time, rather than actual chronological order which is what you would naturally expect (so for e.g. 6m20s will be sorted ahead of 70s).

Here the image pull error is because the image only exists on the local Docker registry on our local machine, and not within the container that is running the Kubernetes cluster (minikube or your local machine for the case of Rancher Desktop). Kubernetes default pull policy is to always in the first instance obtain the image specified imperatively or declaratively from the public Docker Hub registry.

Delete the deployment:

```
kubectl delete deployment first-app
```

Now, lets push this new image we built to our DockerHub account. If you are not yet logged in locally at the CLI to your DockerHub account, login first by typing:

```
docker login
```

Retag the current image appropriately to enable it to be pushed correctly to this new repo in your DockerHub account:

```
docker tag kub-first-app dockerhubaccount/kub-first-app
```

Verify that the image has been retagged correctly in the local Docker registry via the Docker Desktop / Rancher Desktop UI or by typing:

```
docker images
```

Then push it:

```
docker push dockerhubaccount/kub-first-app
```

Verify that this new image is in fact in the repo as shown on the DockerHub page for this repo:

Next, we can repeat the previous deployment using this DockerHub image:

```
kubectl create deployment first-app --image=dockerhubaccount/kub-first-app
```

And verify that the deployment and the pods associated with it is up and running.

```
kubectl get deployments

kubectl get pods
```

Once the image has being successfully pulled from DockerHub to the Kubernetes cluster, it will be stored in the local image registry there.

When we create a new deployment, a ReplicaSet will also be automatically generated and associated with this deployment in order to control the replication of the pods associated with that deployment. For a simple imperative creation of a deployment using the CLI command above, the ReplicaSet will specify exactly one replica for all pods specified in the pod template section.

Double check on this with:

```
kubectl get replicasets
```

Notice that both the pod and ReplicaSet have names that are the same as the Deployment name, but with an additional random sequence appended at the end (e.g. `first-app-xxxxx`)

We can get more detailed info on the ReplicaSet and the single running pod it controls with with:

```
kubectl describe replicaset first-app

kubectl describe pod first-app
```

Some of the key info to note in the output:

- The pod specifies that it is controlled by the given ReplicaSet, and the ReplicaSet in turn specifies that it is controlled by the Deployment.

- The name of the container within the pod is `kub-first-app`, same as the name of the image
- The autogenerated selector and label for the ReplicaSet matches the autogenerated label for the single running pod as well.

We can get more detailed info on the deployment in a similar way:

```
kubectl describe deployment first-app
```

You will also see a Pod Template spec here, as well as a reference to the ReplicaSet it controls in the NewReplicaSet section.

You can also manage various activities related to deployments (creation, configuration, deletion, etc) from the specific dashboard areas (Minikube / Rancher Desktop)

7.1 Working around the default image pull policy

If you want to work around the default image pull policy of Kubernetes (which is pull from Docker Hub), there are some options available to you:

Option A: If you are using Minikube, a possible approach is to build your image within the local registry of the Docker container running Minikube as explained in detailed in the steps below:

https://stackoverflow.com/questions/42564058/how-can-i-use-local-docker-images-with-minikube

https://minikube.sigs.k8s.io/docs/handbook/pushing/#1-pushing-directly-to-the-in-cluster-docker-daemon-docker-env

Option B: If you are using Rancher Desktop, a possible approach is to use `nerdctl` command line tool (which needs to be used with `containerd` enabled as the Container Engine in the Preferences menu in Rancher Desktop), and to use the correct namespace in building the image:

https://github.com/rancher-sandbox/rancher-desktop/issues/952

Option C: Configure a local private registry (external to the local registry of the Docker container that Minikube is in) that runs as a Docker container, and then configure secrets to allow Kubernetes to authenticate and pull from that local private registry:

https://stackoverflow.com/questions/58654118/pulling-local-repository-docker-image-from-kubernetes

https://spacelift.io/blog/kubernetes-imagepullpolicy#running-a-container-from-an-image-in-a-container-registry

https://stackoverflow.com/questions/36874880/kubernetes-cannot-pull-local-image

8 Intro to Services

A Service object is an abstraction which defines a logical set of Pods and a policy by which to access them. The primary use is to direct either internal or external network traffic to the correct Pod replicas in the cluster. Services need to be created in order to expose pods for access: whether this access is internally from other pods within the cluster or externally from outside the cluster.

Large scale apps can be deployed across multiple pod replicas that run across many worker nodes. When a user (an external client) interacts with the app, their request needs to be routed to any one of the available Pod replicas. Services act as a proxy to redirect incoming network traffic coming into the service to one of the available Pod replicas associated with a deployment.

There are many different service types, each with their specific characteristics.

So far, we have not accessed any pods externally.
To expose one or more pods associated with a deployment within the cluster for external access from localhost, you will typically use one of these 2 possible types of services: LoadBalancer or NodePort.

The simplest way of exposing the current deployment for quick access from localhost for development work is through a NodePort service. A NodePort service is exposed externally through a specified static port (called the Nodeport) that is allocated on every worker node by K8s when the NodePort service is created.

External traffic from localhost sent to any worker node at the NodePort will be routed / forwarded to the NodePort service, which then in turn forwards it to the final destination pod or pod replicas

`External host → Node:NodePort → Service → Pod`

If you are using Minikube (which is a single node cluster), the NodePort service will be bound to this single node.

The typical use case for NodePort service is:

a) Local development clusters (Minikube on Docker Desktop)
b) Bare-metal clusters where you do not want or cannot provision cloud load balancers.
c) Debugging / simple demonstrations

```
kubectl expose deployment first-app --type=NodePort   --name=first-app-service --port=8080 --target-port=8080
```

The `--port` flag specifies the port on the service to expose while the `--target-port` flag specifies the port on the pod that the service will forward traffic to. Typically, these 2 values are set to be the same to simplify configuration.

Any other pod / container within the cluster that wishes to communicate with the `first-app` pod will forward the traffic to the NodePort service at the `--port` value, and the service in turn will in turn forward the traffic to the `first-app` pod at the `--target-port` flag value.

Check that it started with:

```
kubectl get services
```

The relevant info here for the columns are:

- NAME: Name of the service
- TYPE: Type of the service
- CLUSTER-IP: Internal IP address assigned to the service within the cluster. This is the address used by other pods within the cluster to communicate with this service. This IP is stable and does not change as long as the service exists.
- EXTERNAL-IP: This indicates whether the service is exposed to the outside world with an external IP. For a NodePort service, this is often <none> because the service relies on the IPs of the nodes and their NodePort for external access, rather than a single external IP
- PORT(S): This indicates the port configuration for the service. It consists of two parts:
    - TargetPort (8080) - the port on which the exposed pod `first-app` is listening and to which the service will forward traffic to.
    - NodePort (a random high value, typically in the range of 30000 and above)e.g. 32767): The port on all cluster nodes which external services will send traffic to which in turn will be routed to the NodePort service
- AGE: This indicates how long the service has been running since its creation.

We can get more detailed info here with:

```
kubectl describe service first-app-service
```

Notice that the selector here targets the label (`app=first-app`) for the single pod in the deployment, which allows us to identify clearly which exposing it for external access.

We will see later on how to declaratively specify a service in a manifest configuration, the same way that we have done for pods, ReplicaSets and deployments.

8.1 Accessing service using Minikube

If you are using Minikube to implement your K8s cluster, you can access the service with:

```
minikube service first-app-service
```

This produces an output similar to below:

```
|-----------|-------------------|-------------|---------------------------|
| NAMESPACE |       NAME        | TARGET PORT |            URL            |
|-----------|-------------------|-------------|---------------------------|
| default   | first-app-service |        8080 | http://192.168.49.2:32636 |
|-----------|-------------------|-------------|---------------------------|
* Starting tunnel for service first-app-service.
|-----------|-------------------|-------------|-----------------------|
| NAMESPACE |       NAME        | TARGET PORT |          URL          |
|-----------|-------------------|-------------|-----------------------|
| default   | first-app-service |             | http://127.0.0.1:7357 |
|-----------|-------------------|-------------|-----------------------|
```

The first URL specifies the IP address of the Minikube single node (`192.168.49.2`) and the Nodeport (`32636`), which is the port on the Node which external applications on the localhost (such as a browser or REST client) will send traffic to which in turn will be routed by the Node to the service, which will in turn route traffic to the target port 8080 on the pod being exposed (`first-app`)

`External host` → `Node:NodePort` → `Service` → `Pod`

However, since Minikube is running on Docker in this instance, the IP address of this node and is NodePort is NOT directly accessible from the host networking system as Docker's networking system is isolated from the host.

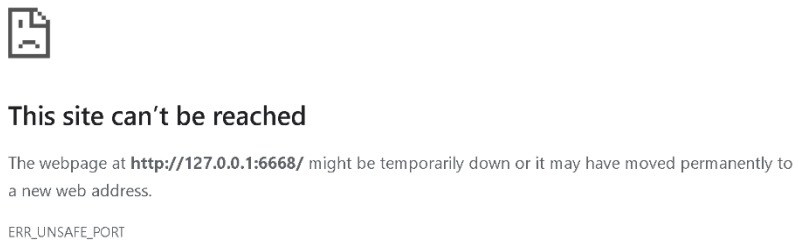
Instead, MiniKube must provide an alternative way to access the Nodeport on the Node: it creates a tunnel from a random port on local networking system (`localhost`) to the Nodeport. This port (`7357`) is randomly assigned from the range of free ports on the local machine. You can then use the loopback address (`127.0.0.1`) or `localhost` at this tunnelling port (`7357`) to forward HTTP traffic to the Node at port (`32636`) , which is then subsequently forwarded to service, which in turn forwards it to the target port 8080 on the destination pod (`first-app`)

`External host` → `localhost:7357` → Minikube tunnel → `Node:Nodeport` (`http://192.168.49.2:32636`) → `Service` -> `first-app` pod (`targetPort 8080`)

The tunnelling URL and port will be automatically opened in a browser tab when you use the `minikube service` command. You should be able to see the HTML response returned from `app.js` in the new browser tab view.

The PowerShell terminal will freeze at this point and needs to be open in order to keep the tunnelling active. If you press Ctrl+C, the IP address and port specified are no longer valid.

NOTE: If you are using Chrome and in the new browser tab view, you obtain an error message similar to the one shown below, then this is an issue with Chrome blocking access to specific ports for security reasons:



You should still be able to access the port by sending the HTTP request from Postman, or just press Ctrl+C to end the tunneling service and reissuing the command:

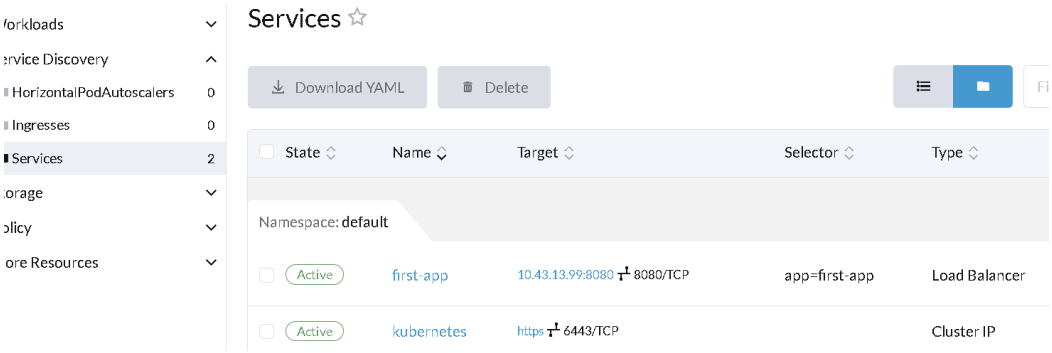
```
minikube service first-app-service
```

which will now provide tunnelling from a new random localhost port.

At this point, you will need to open a new PowerShell terminal to type further commands.

8.2 Accessing service on Rancher Desktop

If you are using Rancher Desktop, you should be able to view the Service in the Rancher Cluster Dashboard. You can drill into this Service to determine the random Nodeport number that was generated earlier.



For Rancher, an additional tunnelling connection from localhost to the Nodeport service as is the case in Minikube is not necessary as Rancher is a complete, standalone K8s installation that is not relying on some other Docker platform. The standard setup is normally configured to forward traffic directed to `localhost` to any existing Nodeport services at the random port value generated earlier (in the range of 30000 and above). You can try directing a HTTP request to `localhost` at this Nodeport port value, for e.g.

http://localhost:32636

If `localhost` doesn't work, you can use the virtualized network interface IP address provided by Rancher Desktop for the nodes. Run the following command to find the existing cluster nodes and their IP addresses:

```
kubectl get nodes -o wide
```

The EXTERNAL-IP or INTERNAL-IP column for your node that the pod is currently running on may give the correct IP address. Use that in the browser:

```
http://<Node-IP>:32636
```

8.3 Scaling Pod replicas and load balancing

Previously, we have seen how ReplicaSets will ensure that Pod replicas are kept to the specified number in the spec: either by generating new pods to replace pods that have been deleted or deleting any additional extra pods that have been generated.

As we have seen earlier, when we create a new deployment, a ReplicaSet will also be automatically autogenerated and this ReplicaSet will specify exactly one replica for all pods specified in the pod template section. Double check again with:

```
kubectl get replicasets
```

There is already a specific command to explicitly crash the app related to an API endpoint path implemented in `app.js`

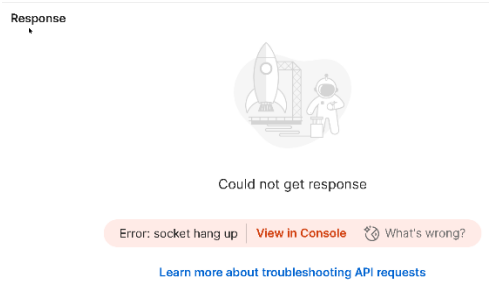
```
app.get('/error', (req, res) => {
```

```
  process.exit(1);
});
```

Use a REST client (Postman) to send a HTTP GET request to this endpoint, for e.g.

`localhost:xxxx/error`

the app will crash and there will be no response returned indicated in Postman



If you immediately check via a command (or in any of the dashboard UIs available) immediately after issuing this GET request:

`kubectl get pods`

you will see that the container has an ERROR status

```
NAME                         READY   STATUS    RESTARTS        AGE
first-app-7cd8895d96-lf7jp   0/1     Error     1 (2m23s ago)   8h
```

After approximately 15 - 30 seconds, the pod should revert to CrashLoopBackoff status

```
NAME                         READY   STATUS             RESTARTS      AGE
first-app-7cd8895d96-lf7jp   0/1     CrashLoopBackOff   1 (16s ago)   8h
```

Attempting to access this app at its root domain: / while the app is in either ERROR or CrashLoopBackOff status will result in an error.

Eventually, the app will return to Running status and it should again be accessible at its root domain: /

As we have seen before, the ReplicaSet for this deployment will attempt ensure that there is the required number of pod replicas running (in this example, it is one).

You can try crashing it again repeated by sending GET requests from Postman to this specific endpoint (`/error`). You will notice it will eventually restart again with a longer time period between each restart. The number of previous restarts is shown in the RESTARTS column. You can check on this repeatedly with:

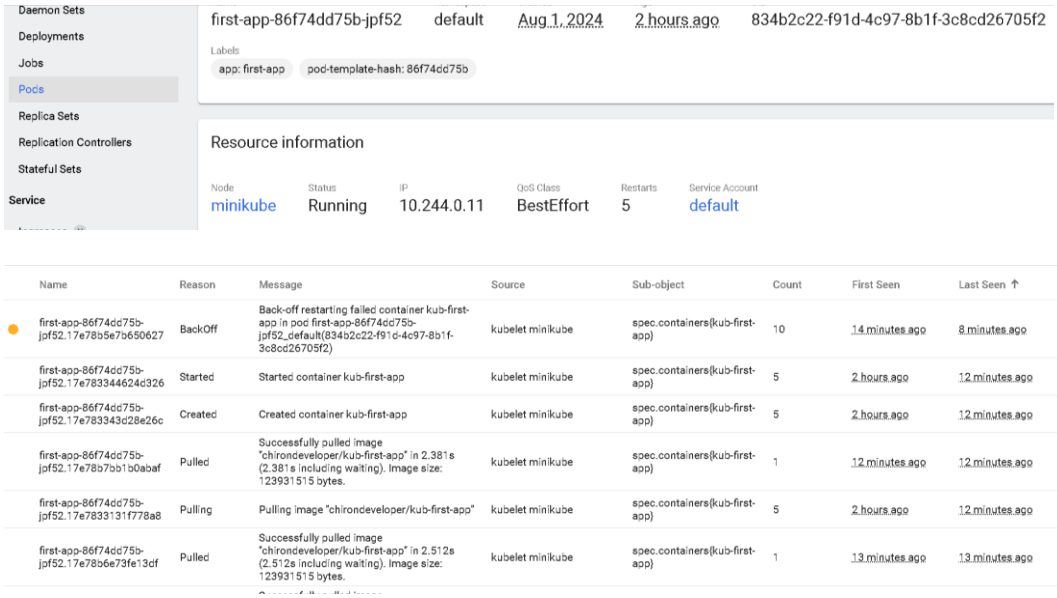
`kubectl get pods`

To get more detailed info on the pod itself, we can issue the standard command:

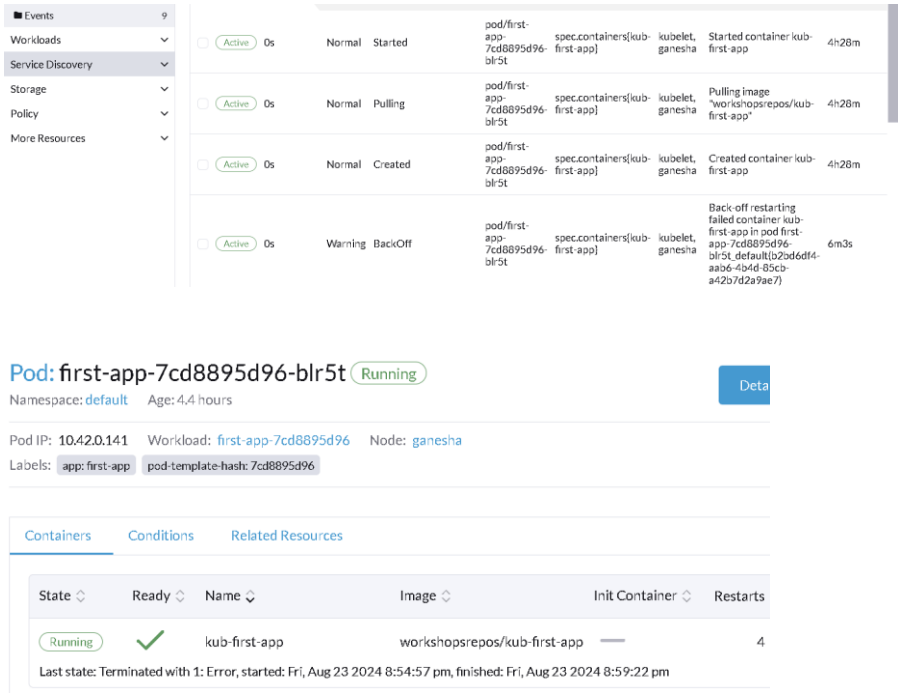
```
kubectl describe pod first-app
```

This will show the number of restarts, and the series of latest events in the event list: backoff, creating new container, pulling image, and starting the container.

You should also be able to see the same detailed info on the Minikube dashboard:



The same info is also available on the Rancher Desktop Cluster dashboard.



Notice that the restart process for the first time when container crashes included the following events: `pulling`, `pulled`, `created`, `started` and also `backoff` (to prevent too quickly a restart). Subsequently, after that the restart is primarily just the backoff event.

We can now change the required number of replicas imperatively with:

```
kubectl scale deployment/first-app --replicas=3 --record
```

We add the `--record` option to record this command as part of the revision history for this particular deployment as an annotation. Note that the `--record` option is deprecated and likely to be dropped in future versions: we will show an alternative approach to record the details of a particular change made to a deployment as part of the deployment's history.

To check the revision history (log of all changes made to a particular deployment), use:

```
kubectl rollout history deployment/first-app
```

Here, the `CHANGE-CAUSE` column details the most recent command that caused a change.

Now if we check the ReplicaSet again, we should see this change:

```
kubectl get replicasets
```

And also the number of pods available:

```
kubectl get pods
```

We should be able to see the 3 pods now also in the dashboard

To verify that load balancing is in effect, we will trigger a crash in a browser tab by sending a HTTP Request again through Postman to the `/error` endpoint. Check with:

```
kubectl get pods
```

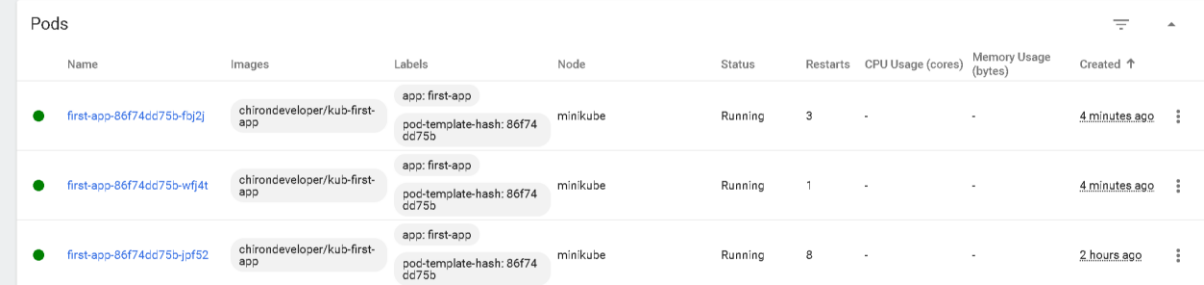
For every request sent, one pod will go down, so if you sent multiple requests successively quickly, you can take down more than one pod - but make sure that there is at least one pod still left running.

With at least one pod still alive (in Running status), send a HTTP request back to the root domain /. You will see that the app is still accessible - which indicates the NodePort service automatically redirects incoming HTTP traffic to any of the remaining running pods.

You can view the process of the pods going down and coming back live again in the dashboard. An example is show for Minikube (it should be similar as well for the case of Rancher Desktop)

Pods

Name	Images	Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Usage (bytes)	Created
first-app-86f74dd75b-fbj2j	chirondeveloper/kub-first-app	app: first-app pod-template-hash: 86f74dd75b	minikube	Error	2	-	-	4 minutes ago
first-app-86f74dd75b-wfj4t	chirondeveloper/kub-first-app	app: first-app pod-template-hash: 86f74dd75b	minikube	Running	1	-	-	4 minutes ago
first-app-86f74dd75b-jpf52	chirondeveloper/kub-first-app	app: first-app pod-template-hash: 86f74dd75b	minikube	Error	7	-	-	2 hours ago



You can scale it back down to 1 replica if you wish with:

```
kubectl scale deployment/first-app --replicas=1 --record
```

You will now see the 2 other pods terminating in the background with:

```
kubectl get pods
```

This is also visible from the dashboard. Eventually, you will be left with exactly one running pod.

If you check the revision history again with:

```
kubectl rollout history deployment/first-app
```

This time you will see that the latest change overwrites the previous revision and appears as revision #1 as it the same type as the previous revision (i.e. scaling replicas).

9 Updating and rolling back deployments

As applications evolve over multiple versions through the lifetime of a project due to bug fixes and new customer requirements, we will need to update the pods in the cluster with different image versions.

Earlier we saw how to update deployments with multiple different image versions (i.e. with different tags). We now repeat the same process here but using the custom image we have being working with.

Update `app.js` to simulate new changes in the project code base:

```
app.get('/', (req, res) => {
  res.send(`
    <h1>Hello from this NodeJS app!</h1>
    <p>Try sending a request to /error and see what happens</p>
    <h2>Some new addition for v2 of this app</h2>
  `);
});
```

Build a new image with a different tag from the one that we had previously used for pushing to our DockerHub account:

```
docker build -t dockerHubAccount/kub-first-app:v2 .
```

Check that the image has been created correctly in the local Docker registry via the Docker Desktop / Rancher Desktop UI or by typing:

```
docker images
```

Then push this updated image to DockerHub in the usual way:

```
docker push dockerHubAccount/kub-first-app:v2
```

To update the deployment to a new image, we also need to specify which current container and image to update with which future image. A sample format would be:

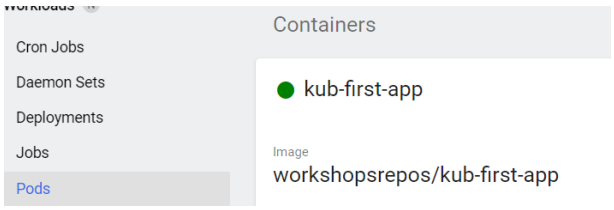
```
kubectl set image deployment/first-app  container-name=new-image-to-use
```

To obtain the container name running in a pod, you can use the describe command to get detailed info:

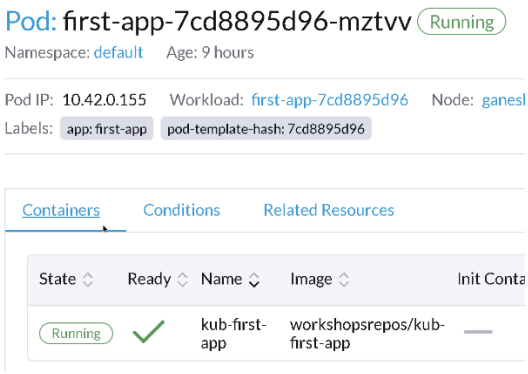
```
kubectl describe pod first-app
```

```
Name:             first-app-7cd8895d96-vjhc2
Namespace:        default
….
….
Controlled By:  ReplicaSet/first-app-7cd8895d96
Containers:
  kub-first-app:
```

It is also visible from the Minikube cluster dashboard



The equivalent view from the Rancher cluster dashboard



Typically, the name of the container will be the same as the tag of the image it was generated from, in this case: `kub-first-app`

The command then becomes for our case:

```
kubectl set image deployment/first-app kub-first-app=dockerHubAccount/kub-first-app:v2 --record
```

You can view the status of the ongoing update immediately after applying it with:

```
kubectl rollout status deployment/first-app
```

You should now be able to view the latest changes you have incorporated in your source code by refreshing the webpage at the specific URL and port number that you accessed earlier.

If you check the revision history:

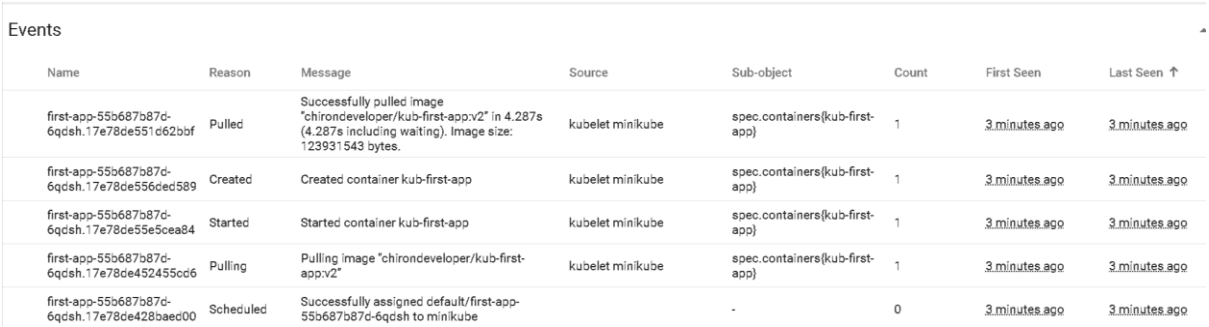
```
kubectl rollout history deployment/first-app
```

You should now see this 2nd command recorded in the history

You can verify that this latest updated image version was pulled and used to create the latest version of the container in the pod with:

```
kubectl describe pod first-app
```

These actions should show up in the Event list.
This is also visible in the Events portion of the Minikube Dashboard.



We will simulate an error occurring in the latest image deployment by attempting to update the deployment with a non-existing image from DockerHub, for e.g.

```
kubectl set image deployment/first-app kub-first-app=dockerhubaccount/kub-first-app:vxxx --record
```

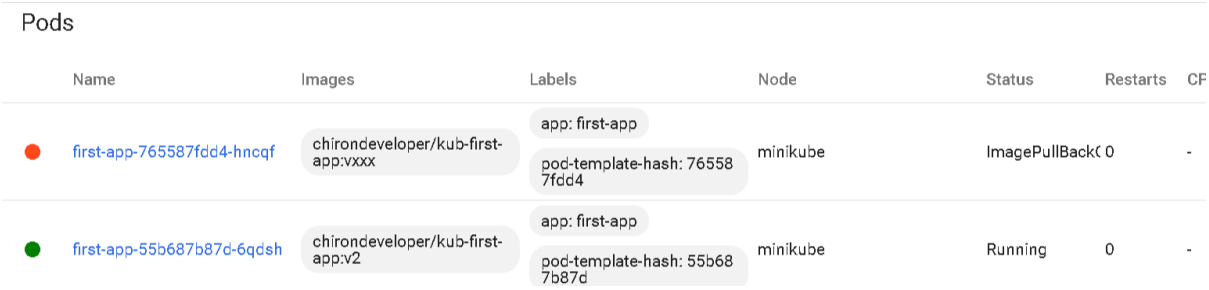
Then immediately check on the status of the deployment with:

```
kubectl rollout status deployment/first-app
```

After a while some error messages coming out:

```
Waiting for deployment "first-app" rollout to finish: 1 old replicas are pending termination...
```

In the events section for this pod in the dashboard, we can see events regarding an error when trying to pull the image, backing off for a short period of time, and reattempts the pull again.



Kubernetes provides several deployment strategies that dictate how to upgrade or downgrade different versions of applications. Selecting an appropriate strategy can help to eliminate or minimize downtime.

The default strategy for Kubernetes is a rolling update. This incrementally replaces the current running pods with new ones. The new Pods are scheduled on Nodes with available resources, and Kubernetes waits for those new Pods to start before removing the old Pods.

In this situation, the existing pod is not terminated, since the new pod replica is not starting up successfully due to issues with pulling a non-existent image. Because of the rolling update strategy that Kubernetes uses, it doesn't shut down the old pod before the new pod is up and running.

In a new PowerShell terminal, we verify again which pods are stuck:

```
kubectl get pods
```

Before we roll back this problematic deployment, lets again check the revision history to verify that the CLI command that caused this problem is recorded:

```
kubectl rollout history deployment/first-app
```

And let's now roll back this problematic deployment, which clearly won't complete here.

```
kubectl rollout undo deployment/first-app
```

And check pods again:

```
kubectl get pods
```

The original one pod is running, and the problematic pod is gone. You can also verify this in the Minikube dashboard.

If we check the revision history again, we can see that the latest revision #4 is actually obtained by applying revision #2 (which is the reason why it no longer appears in the list):

```
kubectl rollout history deployment/first-app
```

We can always drill down into the details of a particular revision to double confirm which particular image was used in any particular one of the deployment revisions in the past:

```
kubectl rollout history deployment/first-app --revision=3
```

Revision #3 is the one that caused the error related to attempting to pull a non-existent image. Here, we get info about the specific image that caused the error.

```
kubectl rollout history deployment/first-app --revision=4
```

Revsion #4 above is the latest update that rolled back to the previous deployment using a working image. It is actually revision #2 now applied to become #4 in the sequence.

We have seen that you can undo the current deployment to revert back to the previous one in the revision history with `kubectl rollout undo`. However, we can also optionally roll back to a deployment configuration further back in the history, for e.g. the first deployment with an extra CLI option as shown below:

First we double check the revision that we wish to roll back to, for e.g.:

```
kubectl rollout history deployment/first-app --revision=1
```

Now we rollback our deployment to this particular revision with:

```
kubectl rollout undo deployment/first-app --to-revision=1
```

Now if you access the web page again at the root domain / you should see the webpage corresponding to the first initial deployment based on our first initial build (`docker build -t kub-first-app .`) without the extra web page content that we introduced later.

You can also verify this by checking with:

```
kubectl describe pod first-app
```

or checking the listing of the pod on the Dashboard

And if you check the history again:

```
kubectl rollout history deployment/first-app
```

You will see that revision #1 has now become the latest revision in the list (#5).

So far, we have been updating the configuration through explicit CLI commands (such as `kubectl set image`, `kubectl scale`, etc) which changes the image for the pod container in the deployment or changes the number of pod replicas.

However, we also have the option of changing the various configuration options of the deployment directly via the in-memory settings for the deployment with:

```
kubectl edit deployment first-app --record
```

This opens up all the current in-memory configuration settings for this deployment in a configuration file via the default editor for the system (Notepad for Windows, for e.g.) and allows you to change their values. Here, increase the number of replicas to 4

```
replicas : 4
```

Save and exit Notepad.
Now if you check on the deployment again, you should see the number of pod replicas have increased to 4.

```
kubectl get deployments
```

And this change is also recorded in the revision history accordingly:

```
kubectl rollout history deployment/first-app
```

It overwrites the previous latest revision, which was also related to a scaling operation.

Let's delete the existing deployment and its associated service with:

```
kubectl delete service first-app-service

kubectl delete deployment first-app
```

The deployments, ReplicaSets, services and pods should now disappear from the dashboard pane.

10 Configuration manifest for deployment and service

Similar to the case of the pods and ReplicaSets earlier, we can create a configuration file to create and change the configuration of a deployment and its associated service.

An example of a manifest to declaratively create a Deployment is shown at:
https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#creating-a-deployment

We will create a new configuration manifest to create a deployment that uses the custom images that we had generated earlier. Place this into the dedicated folder you created earlier: this does not need to be in the same folder as the root folder for the current project: `kub-action`

```
deployment-custom.yaml
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: second-app-deployment
  labels:
    usage: controlProduction
spec:
  selector:
    matchLabels:
      env: production
  replicas: 1
  template:
    metadata:
      name: my-pod
      labels:
        env: production
        tier: backend
    spec:
      containers:
        - name: my-pod-container
          image: dockerhubaccount/kub-first-app:v2
          ports:
            - containerPort: 8080
```

Make sure to replace `dockerhubaccount` with your actual DockerHub account name before proceeding.

All the key sections in this YAML are nearly identical to the manifest for a ReplicaSet that we saw in an earlier lab (with the exception of the `kind` element). This makes sense since the creation of a Deployment will automatically generate a ReplicaSet to control the replication of the pods associated with that deployment.

We repeat again the meanings of the elements for a ReplicaSet manifest here:

- `apiVersion`: This is `app/v1` for a Deployment, which like the ReplicaSet is a workload resource used to manage a pod, vs just `v1` for the Pod earlier.
- `selector`: Specifies a label query used to identify the set of Pods that this Deployment manages.
- `matchLabels`: A map of key-value pairs for matching labels. Any pods that have these key value pairs as their labels will come under the control of this particular Deployment.
- `replicas`: The required number of pod replicas to be maintained at all times. If omitted, defaults to 1.
- `template`: Contains the pod template for creating new pods. The elements that follow after this are exactly the same as the ones found in the our previous YAML manifest for specifying a pod `pod.yaml` (for e.g. `metadata`, `name`, `spec`, `containers`, `image`, `ports`, etc)

To create the deployment specified in this configuration file:

```
kubectl apply -f deployment-custom.yaml
```

As usual check that it is up and running with:

```
kubectl get deployments

kubectl get pods
```

In Kubernetes, a pod specified through a deployment manifest (such as the case here) will always have same name as the Deployment name, but with an additional random sequence appended at the end (e.g. `first-app-xxxxx`). Therefore, the `spec.template.metadata.name` property value of `my-pod` will not apply, unlike in the case of standalone pod specification such as `pod.yaml` earlier.

Next, create a configuration manifest to declaratively create a NodePort service that will expose this deployment for access:

`service-custom.yaml`

```yaml
apiVersion: v1
kind: Service
metadata:
  name: service-backend
spec:
  selector:
    env: production
  ports:
    - protocol: 'TCP'
      port: 8080
      targetPort: 8080
  type: NodePort
```

The `selector` element here targets the label (`env: production`) for the single pod in the deployment in order to expose it for external access.
The other key elements are:

- `port: 8080` - Specifies the port on which the service is exposed to external clients. Clients that wish to communicate with the service will send requests to this port.
- `targetPort: 8080` - Specifies the internal port on the pod that the service will forward traffic to. It can be a port number or a name defined in the pod's container configuration. In this case, the `my-pod-container` is listening on port 8080 (as specified in `deployment-custom.yaml` and will therefore receive all traffic directed to the service.

It is not a strict convention for the `port` and `targetPort` values to be identical, but it is common practice in many cases because it simplifies configuration and reduces confusion. This approach is often followed when there is no specific need to expose the service on a different port than the one used by the containers in the pods targeted by the service.

Create this service in the usual way:

```
kubectl apply -f service-custom.yaml
```

Check that it is available with:

```
kubectl get services
```

Here, you will see a mapping between the `port` value specified in `service-custom.yaml (8080)` and the random NodePort generated by Kubernetes. This means that all traffic directed into the Node at the Nodeport will be redirected to port 8080 on the service, which in turn will be redirected to the `targetPort` of 8080 (where the app in the container is actually listening on)

As you can see, the 2 core port values here are the Nodeport (which the external client must use in order to direct traffic to the service inside the Node) and the `targetPort`, the internal port within the pod that the app in the container is actually listening on. The `port` on the service itself is not too relevant since the mapping between this port and the Nodeport is generated by Kubernetes when the service is created. Thus, it makes sense for the service `port` and `targetPort` to have identical values most of the time to simplify configuration.

You can get more info on the service with:

```
kubectl describe service service-backend
```

If you are using Minikube to implement your K8s cluster, you can access the service with:

```
minikube service first-app-service
```

Similar to the case previously, Minikube creates a tunnel from a port on localhost to the Nodeport. This port is randomly assigned from the range of free ports on the local machine. You can then use the loopback address (`127.0.0.1`) or `localhost` at this tunnelling port to forward HTTP traffic to the service `first-app`, which automatically is accessed through a browser tab when you use the `minikube service` command.

For Rancher, an additional tunnelling connection from localhost to the Nodeport service as is the case in Minikube is usually not necessary. The standard setup is normally configured to forward traffic directed to `localhost` to any existing Nodeport services at the random port value generated earlier (in the range of 30000 and above). You can try directing a HTTP request to `localhost` at this Nodeport port value, for e.g.

http://localhost:34567/

This is the original specified application container port (8080) and Rancher automatically maps it to an equivalent port on localhost for case of a Service of type `NodePort`.

10.1 Performing updates and rollbacks via manifest

We have done this before previously in an imperative manner (for e.g. through `kubectl set image` ….), and here we demonstrate updates with this new deployment but this time through

modification of the configuration manifest. We will also add an option that makes it easier for us to track the updates that were applied to the configuration manifest:

Change this in `deployment-custom.yaml`

```
  replicas: 3
```

Then apply it again to update the deployment state, but this time specifying the

```
kubectl apply -f deployment-custom.yaml --record
```

Check that there are now 3 pod replicas running:

```
kubectl get deployments

kubectl get pods
```

We can check the deployment revision history in the same way that we did in the past:

```
kubectl rollout history deployment/second-app-deployment
```

Notice that this time (as opposed to imperative updates in the past), recording the `kubectl` commands executed doesn't actually tell us exactly what was the specific nature of the update performed, since the nature of the update is specified as modifications with the configuration manifest itself.

Make another change in `deployment-custom.yaml` to go use an earlier version of the image

```
    spec:
     containers:
       - name: my-pod-container
         image: dockerhubaccount/kub-first-app
```

Then apply it again to update the deployment state:

```
kubectl apply -f deployment-custom.yaml --record
```

Reload and check that the webpage renders the HTML corresponding to the initial version of `app.js`

You can also check that the 3 new pod replicas have been created with the reverted image, and the other existing 3 pod replicas are terminated:

```
kubectl get pods

kubectl describe pod second-app-deployment
```

Note that in the command above, we omitted the final segment of the unique sequence of the actual name of one of the 3 pod replicas (i.e. `second-app-deployment-xxxx`): in that case, description of all 3 pod replicas (whose names start with the deployment name `second-app-deployment`) will be listed.

Now if we again check the revision history:

```
kubectl rollout history deployment/second-app-deployment
```

we obtain exactly the same CHANGE CAUSE as the 1st revision, which as mentioned earlier, does not help us to really understand the nature of the change without further drilling down into the revision itself using its associated number, as follows:

```
kubectl rollout history deployment/second-app-deployment --revision=1

kubectl rollout history deployment/second-app-deployment --revision=2
```

As an alternative to recording the `kubectl` command executed (which does not help us in this situation), we can also directly provide our own custom annotation for each revision that we make. So for the latest revision #2, we can do this with:

```
kubectl annotate deployments second-app-deployment kubernetes.io/change-cause="Reverted deployment back to the original image kub-first-app"
```

We should now see this custom message applied to revision #2 when we check the revision history:

```
kubectl rollout history deployment/second-app-deployment
```

Just as we have done before previously using an imperative approach, we can also revert back to an previous revision with the same command that we used before in the past:

```
kubectl rollout undo deployment/second-app-deployment
```

Reload and check that the webpage renders the HTML corresponding to the latest version of app.js (incorporated in the image `kub-first-app:v2`).

If we check the deployment revision history again:

```
kubectl rollout history deployment/second-app-deployment
```

We will see that revision #1 (which was based on this latest image) has now being applied to become revision #3 in the revision sequence.
We can also verify that `kub-first-app:v2` is indeed the latest image being used by checking any one of the 3 pod replicas:

```
kubectl describe pod second-app-deployment
```

We can now proceed to annotate this latest revision with a more explicit custom message of our own as we have done previously:

```
kubectl annotate deployments second-app-deployment kubernetes.io/change-cause="Using deployment based on image kub-first-app:v2"
```

Now if we check again:

```
kubectl rollout history deployment/second-app-deployment
```

We will now have much more user-friendly annotations to record and describe all our revisions, which is ultimately the way to go if we are updating our deployments purely by making changes to the configuration manifest (vs an imperative approach, where `--record` option to simply record the actual `kubectl` command being issued is still a viable option).
NOTE: The `--record` option is currently deprecated; once it is removed, you should then use `kubectl annotate` to describe all revisions, regardless of whether it is imperative or declarative.

To delete the running deployment, we could either use an existing command that we used for imperatively declared deployments, i.e.

```
kubectl delete deployment second-app-deployment
```

Or we could use the deployment file to specify deletion of all objects / resources specified in there with:

```
kubectl delete -f deployment-custom.yaml
```

10.2 Merging manifest YAML files

If we wish, we can create a single manifest configuration file that includes all the configuration for all the objects that are relevant for our cluster (deployments, services and any other related workload resources).

Create a `master-deployment.yaml` file that merges the previous 2 separate config files. The key part here is the use of the --- as per YAML syntax to denote the start of a new 2nd config document within the single file.

`master-deployment.yaml`

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: second-app-deployment
  labels:
    usage: controlProduction
spec:
  selector:
    matchLabels:
      env: production
  replicas: 1
  template:
```

```
    metadata:
      name: my-pod
      labels:
        env: production
        tier: backend
    spec:
      containers:
        - name: my-pod-container
          image: dockerhubaccount/kub-first-app:v2
          ports:
            - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: service-backend
spec:
  selector:
    env: production
  ports:
    - protocol: 'TCP'
      port: 8080
      targetPort: 8080
  type: NodePort
```

Assuming you had brought down the deployment earlier, we can also bring down the service associated with it with:

```
kubectl delete -f service-custom.yaml
```

Check that the deployment and service and all their associated objects (such as pods and replicasets) are both removed with:

```
kubectl get all
```

Then restart both of them again with this single manifest:

```
kubectl apply -f master-deployment.yaml
```

And check that all objects associated with them are created with:

```
kubectl get all
```

If you are using Minikube to implement your K8s cluster, access the service as usual with:

```
minikube service service-backend
```

If you are using Rancher Desktop, you should be able to directly access this service through the port 8080 on localhost
http://localhost:8080/

10.3 Configuring the image pull policy

Kubernetes has an image pull policy that can cause subtle errors if not understood properly.

To see this in action, make a change to `app.js` to add in more additional output statements with a variety of HTML headers with random content such as:

`app.js`

```
app.get('/', (req, res) => {
  res.send(`
    <h1>Hello from this NodeJS app!</h1>
    <p>Try sending a request to /error and see what happens</p>
    <p>More new stuff </p>
    <p>And lots more stuff </p>
  `);
});
```

Now build it again with exactly the same image name that we pushed up to our DockerHub account most recently:

```
docker build -t dockerHubAccount/kub-first-app:v2 .
```

and then push it to the DockerHub account repo:

```
docker push dockerHubAccount/kub-first-app:v2
```

Now if you attempt to get the latest image that you just pushed by reapplying the configuration manifest:

```
kubectl apply -f master-deployment.yaml
```

The output message will indicate that nothing has changed, and that Kubernetes did not pull down this latest image update from your DockerHub account to create a new pod replica (verify for yourself by refreshing the webpage and noticing that it still renders the same).

This reason for this because the default image pull policy (if none is specified) is according to the rules:

a) If the image tag is `:latest`, the `imagePullPolicy` will be automatically set to Always.
b) If the image tag isn't `:latest`, the `imagePullPolicy` will be automatically set to `IfNotPresent`.
c) And if you don't set any image tag, the `imagePullPolicy` will be automatically set to latest image and Always value.

I

In our situation, the image tag isn't `:latest` (it is `v2`), therefore the `imagePullPolicy` will be automatically set to `IfNotPresent`. This means Kubernetes will only pull the image when it doesn't already exist on the node. In this case, the image with that specific tag already exists from a previous successful pull from DockerHub and Kubernetes does not keep track of timestamps of revisions on the image, and so does not know that it has changed from the last time it was applied in the existing deployment.

This is problematic because applying the configuration manifest again will not update our pod with the latest container image unless we explicitly build the image again with a completely new tag (for e.g. *dockerHubAccount*`/kub-first-app:v3` and then use this new tag in the spec for the container in the configuration manifest.

To get around this subtle error, we can configure the image pull policy, so that we always pull the image from the repository. This ensures that if we ever do a rebuild of that image without changing its tag and push it to DockerHub account, Kubernetes will always pull the image from DockerHub whenever the configuration manifest is applied.

Make this change to `master-deployment.yaml` to configure the policy in the way just described.

`master-deployment.yaml`

```
….
    spec:
      containers:
        - name: my-pod-container
          image: dockerhubaccount/kub-first-app:v2
          imagePullPolicy: Always

……
```

Now if you reapply the `master-deployment.yaml` file with the new `imagePullPolicy`

```
kubectl apply -f master-deployment.yaml
```

You see a message indicating a configuration change for the deployment, resulting in the previous pod being terminated and a new pod being created with a container created by pulling down the latest updated image (even though the image tag did not change).

Check on this with:

```
kubectl get pods
```

If this does not happen automatically, you may need to bring down the existing deployment first before restarting it:

```
kubectl delete -f master-deployment.yaml

kubectl apply -f master-deployment.yaml
```

Check the webpage at the service URL / port and refresh it to see that the latest changes you made to `app.js` are now finally present

Finally, when you are done, clean up all deployments and services:

```
kubectl delete -f master-deployment.yaml
```

Check that they are all eventually removed with:

```
kubectl get all
```