From Pods to Deployments

Manifests are under the `classfiles/k8s-1` directory.

To avoid hitting the Docker pull limit, obtain a Docker account and create a secret that encapsulates your Docker login:

```
$ kubectl create secret docker-registry regcred --docker-server=https://index.docker.io/v1/ --docker-username=YOURACCOUNT --docker-password=YOURPASSWORD --docker-email=YOUREMAIL
```

This secret needs to appear in all pods that pull Docker Hub images.

1. Run a pod with ephemeral volume

Craft the manifest and run the pod

Create a manifest named `pod.yaml` with this content and launch it:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  imagePullSecrets:
  - name: regcred
  containers:
  - name: nginx
    image: nginx
    volumeMounts:
    - name: website
      mountPath: /usr/share/nginx/html
  volumes:
  - name: website
    emptyDir:

$ kubectl apply -f pod.yaml
```

This pod consists of one container and one volume. The volume, named *website*, is of type *emptyDir*, which means that it resides on the container host and is ephemeral: It is discarded when the pod is deleted.
The container mounts the volume under `/usr/share/nginx/html`, the default location for NGINX content.

Ask yourself: What is the use of an ephemeral container in a pod?

Check whether it started without error:

```
$ kubectl get pod
NAME    READY   STATUS    RESTARTS   AGE
nginx   1/1     Running   0          80s
$ kubectl describe pod nginx
...
Events:
  Type    Reason     Age    From               Message
  ----    ------     ----   ----               -------
  Normal  Scheduled  3m56s  default-scheduler  Successfully assigned default/nginx to minikube
  Normal  Pulling    3m55s  kubelet            Pulling image "nginx"
  Normal  Pulled     3m52s  kubelet            Successfully pulled image "nginx" in 3.440449574s
  Normal  Created    3m52s  kubelet            Created container nginx
  Normal  Started    3m52s  kubelet            Started container nginx
```

This looks good.

Demonstrate that a volume survives termination of the pod's container

Enter the container and create a web page on it.

```
$ kubectl exec -it nginx -- bash
root@nginx:/# cd /usr/share/nginx/html/
root@nginx:/usr/share/nginx/html# echo 'Success!!!' > index.html
root@nginx:/usr/share/nginx/html# exit
```

To show that the volume survives the crash of the pod's container, enter Minikube and kill the main container process.

```
$ minikube ssh
$ ps -ef|grep nginx
```

There should only be one nginx master process.

```
$ kill -9 NGINX_PID
$ ps -ef | grep nginx
```

A new master is running after a few seconds. This is so because containers in Kubernetes pods have the `restart=always` option set by default. The crashed container is immediately replaced.

Leave Minikube and check that the volume is unchanged.

```
$ exit
$ kubectl exec nginx -- cat /usr/share/nginx/html/index.html
Success!!!
$ kubectl get pod
```

The command should report one restart.

2. Enable external access to the pod

Create a NodePort service

Create a manifest named `nodeport.yaml` with this content:

```
kind: Service
apiVersion: v1
metadata:
  name: mynginx
spec:
  type: NodePort
  ports:
    - port: 80
      nodePort: 30000
  selector:
    name: nginx
```

This is a NodePort service. Access to the **IP address of any worker node** and port 30000 is forwarded. The **selector** determines to which pods it will be forwarded.

```
$ kubectl apply -f nodeport.yaml
$ kubectl get svc mynginx
NAME      TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
mynginx   NodePort   10.108.33.205   <none>        80:30000/TCP   61m
```

The Cluster-IP address is only accessible from inside the cluster, i.e. from other pods.

Test the service

It should be possible to access the pod via the Minikube server's IP address.

```
$ IP=$(minikube ip)
$ curl $IP:30000
```

Connection is refused. What is the problem?

The service selects pods with name=nginx, as evidenced by `kubectl describe svc mynginx | grep Selector`. This selector looks at pods labels - but this pod has no label. Consequently, the service doesn't know to which pods to send the traffic it receives.

Correct the error

Edit `pod.yaml` and add a label to the pod's metadata.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    name: nginx
spec:
...
```

Update the pod.

```
$ kubectl apply -f pod.yaml
$ curl $IP:30000
Success!!!
```

Add a second pod to the service

To launch a second identical pod, you have to change the name in the manifest:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-2
  labels:
    name: nginx
...
```

Launch it and add a web page.

```
$ kubectl apply -f pod.yaml
$ kubectl exec -it nginx-2 -- bash
root@nginx-2:/# echo This is pod 2 > /usr/share/nginx/html/index.html
root@nginx-2:/# exit
```

The second pod has the same label as the first. What does this mean for the service?

```
$ curl $IP:30000
This is pod 2
$ curl $IP:30000
This is pod 2
$ curl $IP:30000
Success!!!
$ curl $IP:30000
Success!!!
```

A service provides access to pods in a round-robin fashion. It lives independently from pods - when no pod corresponds to its selector, the service continues to exist, but can't deliver any traffic to the application.

3. Put the pods in a set

It is possible to launch each pod separately, but Kubernetes can manage a group of pods as a ReplicaSet and ensure that the configured number of identical pods is always running.

Create a ReplicaSet of nginx pods

The replicaset.yaml file describes a set of three pods, but misses the pod specification under the template key:

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: mypods
  labels:
    app: mynginxapp
spec:
  replicas: 3
  selector:
    matchLabels:
      name: nginx    # MUST match pod label below
  template:
    # copy pod metadata and spec here
```

Add the nginx pod under the `template` key. It needs to be indented like the comment in the last line.
Ensure that a label `name: nginx` exists. The solution is in `replicaset-solution.yaml`.

```
$ kubectl apply -f replicaset.yaml
$ kubectl get pod,rs
NAME                READY   STATUS              RESTARTS   AGE
pod/mypods-dcdcj    0/1     ContainerCreating   0          7s
pod/mypods-hg2m4    0/1     ContainerCreating   0          7s
pod/mypods-xlhz6    0/1     ContainerCreating   0          7s

NAME                      DESIRED   CURRENT   READY   AGE
replicaset.apps/mypods    3         3         0       7s
```

The ReplicaSet starts with all pods in state ContainerCreating, and no pod ready. Repeat the command after a few seconds. Eventually, all pods should be Running and three pods should be ready.

Check what happens when a pod disappears

Delete one of the pods and display your resources.

```
$ kubectl delete pod/mypods-tq5fq ; kubectl get pod,rs
pod "mypods-tq5fq" deleted
NAME                READY   STATUS              RESTARTS   AGE
pod/mypods-244hb    0/1     ContainerCreating   0          1s
pod/mypods-dcdcj    1/1     Running             0          22m
pod/mypods-xlhz6    1/1     Running             0          22m

NAME                      DESIRED   CURRENT   READY   AGE
replicaset.apps/mypods    3         3         2       22m
```

Add a web page to each pod and test it.

```
$ for p in $(kubectl get pod -o name)
do kubectl exec $p -- "echo This is pod $p > /usr/share/nginx/html/index.html"
done
$ for p in $(kubectl get pod -o name)
do kubectl exec $p -- cat /usr/share/nginx/html/index.html
done
This is pod pod/mypods-dcdcj
This is pod pod/mypods-244hb
This is pod pod/mypods-xlhz6
```

Check the service

The service is still up and distributes traffic to all pods with label `name=nginx`.

Check this by running `curl $IP:30000` several times.

Scale the replicaset up

Edit replicaset.yaml and replace the three replicas by five. Then:

```
$ kubectl apply -f replicaset.yaml
$ kubectl get pod,rs
```

There should be five pods now.

4. The horizontal autoscaler

The Horizontal Autoscaler (HPA) watches CPU usage and other metrics and scales a set of pods up or down according to scaling rules. It works with ReplicaSets but also Deployments and StatefulSets.

Enable autoscaling in Minikube

The HPA needs metrics, which are collected by the metric server. By default, Minikube disables it.

```
$ kubectl get pod -n kube-system | grep metric
```

If there is no metrics-server pod, enable it and check again.

```
$ minikube addons enable metrics-server
$ kubectl get pod -n kube-system | grep metric
```

Create pods with resource limits

Another prerequisite for the HPA is resource limits for pods. Currently, your pods are not limited.
Add a CPU resource limit to the replicaset manifest:

```
    spec:
      imagePullSecrets:
      - name: regcred
      containers:
      - name: nginx
        image: nginx
        volumeMounts:
        - name: website
          mountPath: /usr/share/nginx/html
        resources:
          limits:
            cpu: 250m
```

This means that each container gets up to 25% of a host CPU core.

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

Create a horizontal autoscaler

You can either define a horizontal autoscaler with a manifest or use the `kubectl autoscale` command. The manifest permits detailed control of parameters like the time the HPA leaves the pods alone after scaling. For simplicity, you will use the command.

```
$ kubectl autoscale rs mypods --cpu-percent=50 --min=1 --max=10
horizontalpodautoscaler.autoscaling/mypods autoscaled
$ kubectl get hpa
NAME     REFERENCE           TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
mypods   ReplicaSet/mypods   0%/50%    1         10        1          13s
```

After a minute or so, check the ReplicaSet.

```
$ kubectl get rs
```

```
NAME      DESIRED    CURRENT    READY    AGE
mypods    1          1          1        1m34s
```

The desired number of pods was scaled down to 1, since they don't use the CPU.

Automatically scale up

Create CPU load on the remaining pod.

```
$ kubectl exec NAME_of_pod -- bash -c "while true; do true; done" &
```

Use the --`watch` (or its abbreviated version `-w`) option to repeatedly check the HPA.

```
$ kubectl get hpa --watch
```

The HPA won't immediately see a difference, but after a few seconds, you will see the result of scaling up:

```
NAME      REFERENCE            TARGETS   MINPODS    MAXPODS    REPLICAS    AGE
mypods    ReplicaSet/mypods   100%/50%   1          10         2           15m
```

then a bit later, when the second pod is up:

```
mypods    ReplicaSet/mypods    50%/50%   1          10         2           15m
```

At this point, the HPA determines that CPU usage is right on target and won't scale up or down.

Automatically scale down

Get an estimation of pods' CPU (and memory) usage.

```
$ kubectl top pod
```

What will happen when you delete the pod that loads the CPU? Think that two controllers are involved - the ReplicaSet controller, and the HPA controller. How do they act?

Run `kubectl get --watch rs,hpa` in a separate window. Delete the high-CPU pod and observe.

5. Deployments

A deployment is like a ReplicaSet but adds update and rollback features.

First, clean up with `kubectl delete all --all`. `all` means all resource *types*, `--all` means all resources. This command is equivalent to the UNIX `rm -rf` and should normally be used with extreme caution (or not at all).

Create a deployment

A simple Deployment manifest looks almost identical to a ReplicaSet manifest.

```
$ cp replicaset.yaml deployment.yaml
$ vi deployment.yaml
```

Make two changes:

1. Replace ReplicaSet with Deployment (and optionally change the Deployment's name)
2. Use a not quite up-to-date image. To find a suitable version, you could use the Docker Hub web site, or `podman search --filter=is-official nginx` followed by the `skopeo list-tags` command.

After the changes, `deployment.yaml` looks like this:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rollingpods
  labels:
...
    spec:
      imagePullSecrets:
      - name: regcred
      containers:
      - name: nginx
        image: nginx:1.20.1
        volumeMounts:
```

Apply the manifest and observe the initial rollout of the application.

```
$ kubectl apply -f deployment.yaml; kubectl rollout status deployment rollingpods
deployment.apps/rollingpods created
Waiting for deployment "rollingpods" rollout to finish: 0 out of 3 new replicas have been updated...
Waiting for deployment "rollingpods" rollout to finish: 0 of 3 updated replicas are available...
Waiting for deployment "rollingpods" rollout to finish: 1 of 3 updated replicas are available...
Waiting for deployment "rollingpods" rollout to finish: 2 of 3 updated replicas are available...
deployment "rollingpods" successfully rolled out
```

Update the Deployment's image

Edit `deployment.yaml` and set the image to `nginx:latest`.

Before applying the manifest, save the list of your pods.

```
$ kubectl get pod > deploypods
```

In a separate window, run `kubectl get --watch rs` to see the list of your ReplicaSets.

Apply the manifest and observe the rollout in both terminal windows.

```
$ kubectl apply -f deployment.yaml; kubectl rollout status deploy rollingpods
deployment.apps/rollingpods configured
Waiting for deployment "rollingpods" rollout to finish: 0 out of 3 new replicas have been updated...
Waiting for deployment "rollingpods" rollout to finish: 1 out of 3 new replicas have been updated...
Waiting for deployment "rollingpods" rollout to finish: 1 out of 3 new replicas have been updated...
Waiting for deployment "rollingpods" rollout to finish: 1 out of 3 new replicas have been updated...
Waiting for deployment "rollingpods" rollout to finish: 2 out of 3 new replicas have been updated...
Waiting for deployment "rollingpods" rollout to finish: 2 out of 3 new replicas have been updated...
Waiting for deployment "rollingpods" rollout to finish: 2 old replicas are pending termination...
Waiting for deployment "rollingpods" rollout to finish: 1 old replicas are pending termination...
Waiting for deployment "rollingpods" rollout to finish: 1 old replicas are pending termination...
deployment "rollingpods" successfully rolled out
```

What is that *termination* about? Compare the list of pods you recorded before the update with the current list. You will see that new pods have been created. The update does not leave the old pods in place. Think of the consequences. What if the app's data were stored on pods' ephemeral volumes?

Meanwhile, in the other window, you see that a new ReplicaSet appears and slowly scales up to the desired number of pods. The old ReplicaSet is scaled down to 0 but is not deleted. This allows the Deployment controller to roll back changes.

Note: You can also update the image imperatively with
`kubectl set image deployment/rollingpods nginx=nginx:latest`.

Update other Deployment fields

A rollout is triggered whenever the pod template in the Deployment is modified. To see this, add a second label `app: rollapp` to `deployment.yaml` and apply it.

```
  template:
    # copy pod metadata and spec here
    metadata:
      name: nginxpod
      labels:
        name: nginx
        app: rollapp
```

Rollout history

Kubernetes keeps a history of rollouts. This will become important when you roll back.

```
$ kubectl rollout history deploy rollingpods
deployment.apps/rollingpods
REVISION  CHANGE-CAUSE
1         <none>
2         <none>
3         <none>
4         <none>
```

It would be nice if a little more information were in the CHANGE-CAUSE column. You can see details of each rollout this way:

```
$ kubectl rollout history deploy rollingpods --revision=2
deployment.apps/rollingpods with revision #2
Pod Template:
  Labels:     name=nginx
       pod-template-hash=686c75db94
  Containers:
   nginx:
    Image:    nginx:latest
    Port:     <none>
    Host Port:      <none>
    Environment:    <none>
    Mounts:
      /usr/share/nginx/html from website (rw)
  Volumes:
   website:
    Type:     EmptyDir (a temporary directory that shares a pod's lifetime)
    Medium:
    SizeLimit:      <unset>
```

But it only shows the updated pod. It doesn't tell you what was changed. Moreover, stepping through all revisions would be tedious.

History annotations

The best thing you can do is **annotate** the last rollout. There are two methods: On the command line or in the manifest.

Use the command line to annotate the last rollout.

```
$ kubectl annotate deploy rollingpods kubernetes.io/change-cause="added label app=rollapp"
deployment.apps/rollingpods annotated
$ kubectl rollout history deploy rollingpods
deployment.apps/rollingpods
REVISION  CHANGE-CAUSE
1         <none>
2         <none>
3         <none>
4         added label app=rollapp
```

Try adding an annotation to the manifest. Change the volume name to "data" and add this annotation to `deployment.yaml`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    kubernetes.io/change-cause: changed volume name from website to data
  name: rollingpods
```

Apply and check the history.

The ReplicaSets also have annotations related to rollouts.

```
$ kubectl describe rs ONE_OF_THE _REPLICASETS | more
```

You should see `deployment.kubernetes.io/revision` and `deployment.kubernetes.io/revision-history`.

Rolling back

You want the old volume name back. Undo the last change:

```
$ kubectl rollout undo deploy rollingpods
```

View the new history. The last revision has not disappeared but become second-to-last, so that you can still go back.

Check whether the volume name is back to "website".

```
$ kubectl describe deploy rollingpods
$ kubectl get deploy rollingpods -o yaml
```

Both commands show annotations. Notice that the rollout revision is also recorded as an annotation named `deployment.kubernetes.io/revision`. Furthermore, the `describe` command shows all events that have taken place. Each rollout becomes a scaling event.

Now go back to the original version of the Deployment. Check the result by listing the history and describing the Deployment.

```
$ kubectl rollout undo deploy --to-revision=1
```

Deployment strategy

A rolling update keeps old pods running until the updated pods are up. Sometimes, you want all old pods to be terminated before the new ones are rolled out. Example: Your application doesn't tolerate two versions deployed at the same time. In this case, set the deployment strategy to Recreate.

Add the deployment strategy to deployment.yaml:

```
spec:
  strategy:
    type: Recreate
  replicas: 3
  selector:
...
```

Make a change to the pod template and apply the manifest. Check the rollout status. The sequence of events is different from a default RollingUpdate strategy; in particular, there is no "pending termination", since the old pods are already terminated.

```
$ kubectl apply -f deployment.yaml ; kubectl rollout status deploy rollingpods
deployment.apps/rollingpods configured
Waiting for deployment "rollingpods" rollout to finish: 0 out of 3 new replicas have been updated...
Waiting for deployment "rollingpods" rollout to finish: 0 out of 3 new replicas have been updated...
Waiting for deployment "rollingpods" rollout to finish: 0 out of 3 new replicas have been updated...
Waiting for deployment "rollingpods" rollout to finish: 0 out of 3 new replicas have been updated...
Waiting for deployment "rollingpods" rollout to finish: 0 of 3 updated replicas are available...
Waiting for deployment "rollingpods" rollout to finish: 1 of 3 updated replicas are available...
Waiting for deployment "rollingpods" rollout to finish: 2 of 3 updated replicas are available...
```