Kubernetes Tutorial

Kubernetes (or k8s for short) is a container orchestration system that allows us to deploy Docker containers across a cluster of machines. It automates scheduling and assignment of containers to physical machines, as well as fault-tolerance, scaling, and application lifecycle management. This tutorial gives a quick-start introduction to using Kubernetes with our testing cluster, but you will have to refer to additional material to get the full picture. We try to recommend useful references throughout.

Prerequisites

You should have the following setup already completed before you begin:

1. A username and password for our Gitlab instance at git.webis.de
2. If you are located **outside Weimar university**, you should have obtained a VPN certificate from cert-server.webis.de, and have your OpenVPN client configured correctly to use it. If you have never worked with us before, the person who introduced you will have set this up for you.

Connecting to the Test Cluster

Install the *kubectl* commandline client

Refer to the official tutorial on kubernetes.io, and complete the appropriate "Install kubectl on $OPERATING_SYSTEM" step. For subsequent configuration, return here and proceed below.

Authenticate to the cluster and Configure *kubectl*

Visit openid.webis.de/k8s-login to authenticate to the Kubernetes cluster with your Gitlab credentials. After successful authentication, you will see a sequence of commands that you should paste into your terminal. Do so.

Test the Connection

Run the following command in your terminal:

```
kubectl get nodes
```

You should see output similar to the following:

```
NAME      STATUS   ROLES    AGE    VERSION
webis6    Ready    <none>   132d   v1.14.1
webis7    Ready    <none>   132d   v1.14.1
webis8    Ready    <none>   132d   v1.14.1
webis9    Ready    <none>   132d   v1.14.1
webis10   Ready    <none>   132d   v1.14.1
```

Create a Personal Namespace

Namespaces isolate different applications on a Kubernetes cluster. For now, we'll create one for you to do the initial tutorial steps in. In the following, we assume that `[username]` is a name that uniquely identifies you — substitute something appropriate, preferably your university login or similar.

```
kubectl create namespace [username]
kubectl config set-context webis6 --namespace=[username]
```

Kubernetes Architecture

A Kubernetes cluster consists of several daemon processes. We will review only the two most important ones below. You can find much more in-depth information in the relevant documentation.

`kube-apiserver`

Runs on a single physical node that serves as the cluster manager (although failover is possible). Handles all communication of the k8s cluster with the outside world (especially with clients that want to deploy applications), as well as within the cluster (for example, nodes report their state and resource availability)

The apiserver for our testing cluster runs on the machine `webis6.medien.uni-weimar.de`.

`kubelet`

Runs on every cluster node where containers will be deployed. Works with the local `docker` daemon to achieve this. Receives instructions from the apiserver, and communicates its status there.

A Brief Tour of Some Important Concepts in Kubernetes

Below, we will very briefly review some of the basic concepts you may encounter when deploying your code on Kubernetes.

General Procedure

The `kubectl` client on your local machine talks to the cluster's `kube-apiserver` — they communicate using the Kubernetes REST API (if you really wanted to, you could use something like `curl` as a client instead). Whenever you want anything to happen on the cluster, you send a declarative specification of the desired state, expressed in terms of the basic resource objects specified by the API. In what follows, we'll use YAML to express our resource specifications.

For each of the examples below, we'll follow this basic procedure:

1. Create a text file containing the resource description, say `my-resource.yaml`
2. Send it to the cluster using the `kubectl` client. For example:

```
bash kubectl apply -f my-resource.yaml
```

(instead of `apply`, other operations, such as `patch`, `replace`, or `delete` are available)

We'll walk through doing this for the most important types of resources.

Pods

Pods are the smallest deployable units in Kubernetes. A pod consists of one or more Docker containers that together perform a single task.

A First Pod

Create a file my-first-pod.yaml with the following contents:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-first-pod
spec:
  containers:
    - name: my-container
      image: alpine
      command: ['sh', '-c', 'echo Hello, Kubernetes!']
```

This file describes your pod, in particular, it specifies which containers it consists of, and which commands they should run. In this case, we're spawning a single container using the standard `alpine` image (a minimal Linux distribution). The container's root process will spawn a shell that prints a simple message to `stdout` and then terminates immediately.

Deploy this pod like this:

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

Then look at its state and output:

```
kubectl get pod
kubectl describe pod my-first-pod
kubectl logs my-first-pod
```

By the time you get to the `kubectl get`, it will likely have already terminated. The cluster will, by default, restart it a few times until eventually giving up (Pods are expected to run indefinitely).

We can get rid of it now:

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

A Second Pod

Let's define a new pod that doesn't immediately terminate:

my-second-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: my-second-pod
spec:
  containers:
    - name: my-container
      image: alpine
      command: ['sh', '-c', 'echo Hello, Kubernetes! &&
          ↪ touch /keep-running && while [ -e /keep-
          ↪ running ]; do sleep 10; done && echo Bye!']
```

The command has become a bit more complex: the container now creates an empty file on startup, and as long as that file still exists it will keep running.

We'll send our pod specification to the cluster:

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

In the list of running pods (`kubectl get pod`), the Kubernetes cluster will eventually show our new pod as running, and it will stay that way. We can run additional commands in an existing pod, which is very useful for debugging, because you can spawn an interactive shell:

```
kubectl exec -it my-second-pod sh
```

(the option `-i` means you want to pass stdin to the pod, and `-t` that stdin is a terminal)

Inside that new shell, we can write a new file, and then make it terminate:

```
echo "hello" > /my-new-file
cat /my-new-file
rm /keep-running
```

The pod will terminate within ten seconds and your shell session along with it, but the cluster will restart the pod immediately. However, the file you created is gone:

```
kubectl exec my-second-pod -- cat /my-new-file
```

```
cat: can't open '/my-new-file': No such file or directory
command terminated with exit code 1
```

Volumes

All storage in the pod we've just created is ephemeral, that means any on-disk changes are lost once it terminates or restarts. Volumes are Kubernetes' storage abstraction, and they supply pods with persistent state.

Kubernetes supports many different storage backends, but we'll only look at a small number of examples below.

Short-term Persistence with *emptyDir*

We'll create a new pod with a modified spec.

pod-with-emptydir.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-emptydir
spec:
  volumes:
    - name: my-volume
      emptyDir: {}
```

```
9   containers:
10    - name: my-container
11      image: alpine
12      command: ['sh', '-c', 'echo Hello, Kubernetes! &&
            ↪ touch /keep-running && while [ -e /keep-
            ↪ running ]; do sleep 10; done && echo Bye!']
13      volumeMounts:
14        - mountPath: /myvol
15          name: my-volume
```

Run: `kubectl apply -f` pod-with-emptydir.yaml

We've added two new things here:

1. A new key `volumes` in our pod spec, which contains a list of all the volumes we use, in particular their name, which storage backend they use, and any options (in this case, none). Our volume is named `my-volume` and uses Kubernetes' `emptyDir` storage backend — this creates an empty directory on the Kubernetes host, and shares it with our pod.
2. A new key `volumeMounts` inside our pod's container specification, which specifies which volumes the container should mount were. We could also specify mount options here, e.g. for read-only access.

Exercise: Using what you've learned before, spawn a shell inside your pod, create a new file under the `/myvol` directory, and then cause the pod to terminate (by deleting `/keep-running`). After Kubernetes restarts your pod, see if the file is still there.

Spoiler: it is.

The contents of `emptyDir` volumes persist across pod restarts. This is possible because whenever a pod terminates due to some error, Kubernetes will reschedule it on the same host. However, once you manually delete your pod, your `emptyDir` volumes are irrevocably gone. This is useful e.g. for caching, where your pods may create temporary data that are useful to keep around, but not critically important.

PersistentVolumes and *rbd*

Persistent volumes exist independently of the pod lifecycle. They supply your Kubernetes pods with storage that persists across multiple deployments, or that can be attached to different pods over time. They do introduce a lot of additional complexity, but we will skip over most of that here (refer to the official documentation if open questions remain).

There are two relevant Kubernetes API objects, PersistentVolume and PersistentVolumeClaim. For the purposes of this tutorial, you will only interact with the `PersistentVolumeClaim` object, which describes the storage you need.

The cluster will automatically manage the underlying volume that satisfies your request.

my-first-pvc.yaml

```yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: my-first-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Mi
  storageClassName: ceph-rbd
```

Run: `kubectl apply -f` my-first-pvc.yaml

The keys under `spec` describe the volume that we want. Here, we're requesting one megabyte worth of storage, using a StorageClass we've named `ceph-rbd`. Storage classes are defined by the administrators of a particular Kubernetes cluster to make different storage backends available to Kubernetes — in our case, we're using a Ceph cluster that provides virtual block devices to our containers.

You will be able to see your claim in the output of `kubectl get pvc`. The STATUS column will show *Pending* as the cluster provisions your volume, and switch to *Bound* once it's ready for use. Next, we'll define a Pod that mounts this storage into one of its containers.

Run: `kubectl apply -f` pod-with-pv.yaml

This pod works very similarly to the one with the `emptyDir` before. However, we now use our persistent virtual block device that we've just created instead of a temporary directory on the host file system, and mount that in the `/myvol` directory.

Exercise: Deploy the pod above to the cluster, spawn a shell inside its container, and write something to a file in the `/myvol` directory. Inspect the output of `kubectl describe pod pod-with-pv` — The *From* column in table at the end of the output tells you on which physical host the pod is running (They are named "webis6" through "webis10"). Delete your pod, and re-deploy it. Repeat until it gets deployed on a different host than it was before. You will find that the contents of `/myvol` are preserved across deployments.

Cleaning up

Let's get rid of the pod we just created, and allow the cluster to free up the storage we used:

```
kubectl delete pod/pod-with-pv pvc/my-first-pvc
```

ConfigMaps

ConfigMaps can share textual that is specified directly in the YAML file with your containers. As the name implies, they are intended for configuration, but are also useful to store simple scripts inline, in the same place as your deployment files.

Let's create a ConfigMap, and a Pod that uses it in one go.

pod-with-configmap.yaml

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-first-configmap
data:
  my-script: |-
    #!/bin/sh
    echo "I am a container startup script defined in a ConfigMap!"
    echo "I will do nothing for the next hour."
    echo "By the way: the value of the FOO environment variable is '${FOO}'."
    sleep 3600
    echo "Byeee"
  my-other-key: "my other value"
---
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-configmap
spec:
  volumes:
    - name: my-configmap-volume
      configMap:
        name: my-first-configmap
        defaultMode: 0700
  containers:
    - name: my-container
      image: alpine
      command: ['/bin/startup.sh']
      volumeMounts:
        - name: my-configmap-volume
          subPath: my-script
```

```
          mountPath: /bin/startup.sh
          readOnly: true
      env:
        - name: FOO
          valueFrom:
            configMapKeyRef:
              name: my-first-configmap
              key: my-other-key
```

Run: `kubectl apply -f` pod-with-configmap.yaml

The example above introduces several new concepts: You can specify multiple API objects in a single YAML file; they are separated by lines containing three dashes. Here, we specify a ConfigMap object that contains two keys (the value of one of them happens to be the code of a shell script, written with the YAML syntax for multi-line strings). Below the `---`, we define a pod that makes use of this ConfigMap, and we see two ways that ConfigMaps can be used:

1. **Mounted into the container's file system.** To this end, we give the pod a volume of the `configMap` type that refers to our ConfigMap above (we also specify that its contents should be mounted with execute permissions). In the container's `volumeMounts` key, we then mount the value of the ConfigMap's `my-script` key as the file `/bin/startup.sh` (we introduce some new options to volume mounts that allow us to mount an individual key as a file in an existing directory. We could also mount the entire ConfigMap in an empty directory as we did before, and would then get one file per key). We use `/bin/startup.sh` as the command the container should execute.
2. **Via environment variables.** In the `env` key of the container specification, we set the value of the environment variable `FOO` to the value of our ConfigMap's second key. Note that the startup script prints out the value of this environment variable.

Use `kubectl logs pod-with-configmap` to see the output of the startup script, and verify that this works.

Pod Networking

According to the Kubernetes networking model, each pod in the cluster gets its own IP address, in a virtual, cluster-internal IP range. Every pod can communicate with every other pod given its IP, no matter where it runs in the cluster (containers inside the same pod share an IP address and can communicate via the `localhost` interface**. We will quickly demonstrate this.

Exercise: Start any two of the pods we've used in this tutorial before (except for the first one that terminates immediately). Using `kubectl` you can see their

internal IP, and which physical host they have been deployed on (we'll use a custom output format to show the columns we're interested in):

```
kubectl get pod -o custom-columns=Name:metadata.name,Host
    ↪ :spec.nodeName,PodIP:status.podIP
```

The output will look something like this:

```
Name                  Host      PodIP
my-second-pod         webis10   10.23.151.179
pod-with-configmap    webis6    10.23.204.126
```

Our two pods run on different physical hosts in this case (you may delete and re-create one of them if that's not the case for you). Open two terminals side-by-side, and spawn a shell in each of the two pods.

In the first pod, run

```
nc -v -l -p 1234
```

(This starts netcat in listening mode on Port 1234)

Switch to the terminal with the shell for the other pod, and run

```
nc -v <ip-of-the-first-pod> 1234
```

(Refer to the table you produced with `kubectl get` above for the correct IP)

Type any line of text followed by enter, it should show up in the other pod's terminal (this also works in the other direction). Type Ctrl+C to get out of netcat.

Deployments

When you run complex applications on a Kubernetes cluster, you will generally not create individual pods directly. Instead, you will create a Deployment which declaratively specifies a set of pods that you would like to exist. This has several advantages over creating individual pods: for example, you can easily specify a pod template that should be replicated in a given number of copies for load-balancing and failover purposes. When you change the desired number of replicas, the cluster will create/destroy pods automatically to match.

To illustrate, we will use the flask server example from our docker tutorial. Let's build this with a tag and push it to Dockerhub:

```
cd examples/flask-server
docker build . -t <your-dockerhub-username>/flask-server
docker push <your-dockerhub-username>/flask-server
```

(You may have to create an account and run `docker login` first)

Once it's on Dockerhub, we can deploy it to the kubernetes cluster as well. We will use the aforementioned Deployment controller this time:

flask-server-deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: flask-server-deployment
  labels:
    app: my-flask-server
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-flask-server
  template:
    metadata:
      labels:
        app: my-flask-server
    spec:
      containers:
        - name: flask-server
          image: webis/flask-server
```

Run `kubectl apply -f` flask-server-deployment.yaml

When looking at the file, we note a few novelties: First of all, it specifies a Deployment API object. The key `spec.template` contains the description of the pods that this deployment will create — this subtree has the same keys as the pods that we've created directly up to now. Second, we are using Kubernetes' Labels and Selectors mechanism for the first time. In brief, labels allow us to annotate any object we create with arbitrary key-value pairs. We use this to give both our deployment and our pods a common `app` label. This is necessary, so that the Deployment can identify the pods that it manages — we tell it how to identify its pods using the `selector` key.

When you run `kubectl get all`, you will notice that a total of three new objects have been created. Example output might look like this:

```
NAME                                          READY   STATUS    RESTARTS   AGE
pod/flask-server-deployment-6d9c7f59fc-gvc4x   1/1     Running   0          16s

NAME                                     READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/flask-server-deployment   1/1     1            1           16s

NAME                                                DESIRED   CURRENT   READY   AGE
replicaset.apps/flask-server-deployment-6d9c7f59fc   1         1         1       16s
```

Our deployment controller manages a ReplicaSet, which it turn manages the pods, and ensures that the desired number are running.

Exercise: Change the pod spec to use your own image. Increase the number of `replicas` in the deployment spec. Run `kubectl apply` again and observe what happens via `kubectl get all`. *(you should find that the desired number of replicas start with the new image. Only after they are up will your old pod be terminated.)*

Publishing Ports

Of course, we'll actually want to use the flask server at some point. In order to make that happen, we have to make the port that flask uses known to the cluster. For that, we'll deploy a modified version:

flask-server-deployment-ports.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: flask-server-deployment
  labels:
    app: my-flask-server
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-flask-server
  template:
    metadata:
      labels:
        app: my-flask-server
    spec:
      containers:
        - name: flask-server
          image: webis/flask-server
          ports:
            - name: flask-server
              containerPort: 5000
              protocol: TCP
```

Run: `kubectl apply -f` flask-server-deployment-ports.yaml

The only addition is the `ports` key in the container section of the pod specification. It assigns the name `flask-server` to the internal container port 5000.

You can access this port in a local browser by running:

```
kubectl port-forward deployment.apps/flask-server-
    ↪ deployment 12345:flask-server
```

Then, visit localhost:12345 in your browser. You should see the "Hello World" message from the flask app.

Services

If we want to make our little flask app visible to the world, the approach with `kubectl port-forward` is less than ideal. This is where Services come in. While there is a multitude of different implementations (many of them specific to the infrastructure provided by public clouds such as AWS or GCE), we will only cover the basic NodePort services here. These work as follows: you assign a port number to your service, in a range allowed by the cluster; once the service is deployed, *every physical cluster node* listens on this port number, and forwards any connections to it to your service. The following example will illustrate this:

flask-service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: flask-service
  labels:
    app: my-flask-server
spec:
  type: NodePort
  ports:
    - port: 80
      targetPort: flask-server
      nodePort: 31999
  selector:
    app: my-flask-server
```

Run: `kubectl apply -f` flask-service.yaml

Since NodePorts must be unique cluster-wide, you will have to change this port number until you find one that nobody else is using; as currently configured, our cluster allows ports from 30000 up to 32767 for NodePort services. Find one that's available.

Just like Deployments, services identify their associated pods with a selector. Every service has a name and (at least) one service port that is mapped to the named port from our deployment. The service's name functions as an internal DNS name: in our case, you can access it as `http://flask-service:80` from anywhere inside the kubernetes cluster. To illustrate, spawn a shell in one of our previous pods, for example:

```
kubectl exec -it my-second-pod sh
```

Once in, install curl:

```
apk add --no-cache curl
```

And use it to access the flask app:

```
curl flask-service:80
```

You should see the Hello world page's HTML source.

In addition we have defined the public-facing NodePort. Assuming the number you used above is 31999, you should be able to access the service in your browser using any of the following URLs:

```
http://webis6:31999/
http://webis7:31999/
http://webis8:31999/
http://webis9:31999/
http://webis10:31999/
```

(of course, the service and the deployment can be specified in the same file using the three-dashes syntax)

Things We Didn't Cover

Kubernetes defines (many) more kinds of API objects. A few that may be relevant are briefly mentioned below:

- InitContainers can be added to a pod to run initialization code before the pod's regular containers are created.
- Secrets work similarly to ConfigMaps, but are intended for sensitive data like API keys or passwords. They are stored in encrypted form.
- StatefulSets manage groups of pods that are created from the same template, but have some run-time state that makes them unique; as an example, consider a distributed database, where each pod runs the same code, but has its own unique database shard.
- DaemonSets manage groups of pods where every node (or some subset) of the cluster should run exactly one copy.
- Jobs and CronJobs define pods that do not run indefinitely as daemon process, but rather are expected to terminate at some point. Jobs run once, CronJobs run periodically.