JTC02 Kubernetes Labs

Lab0 - Lab information

Kubernetes is a container orchestrator to provision, manage, and scale applications. In other words, Kubernetes allows you to manage the lifecycle of containerized applications within a cluster of nodes (which are a collection of worker machines, for example, VMs, physical machines etc.).

The key paradigm of kubernetes is it’s Declarative model. The user provides the "desired state" and Kubernetes will do it's best make it happen. If you need 5 instances, you do not start 5 separate instances on your own but rather tell Kubernetes that you need 5 instances and Kubernetes will reconcile the state automatically. Simply at this point you need to know that you declare the state you want and Kubernetes makes that happen. If something goes wrong with one of your instances and it crashes, Kubernetes still knows the desired state and creates a new instances on an available node.

In this Lab you will learn the basic concepts for working with Kubernetes.

If you want to learn more about Kubernetes, please take course JTC80.

Lab sources

All the source code for the lab is available here:

https://github.com/niklaushirt/training

Lab overview

- Lab 1: Provides an refresher/overview over Kubernetes.
- Lab 2: This lab walks you through running your first Pod on Kubernetes.
- Lab 3: In this Lab you will deploy a Web Application on Kubernetes, first the Frontend and then the Backend components.
- Lab 4: Builds on lab 3 and shows you how to scale an application on Kubernetes.
- Lab 5: Teaches you the basics of persisting data in Kubernetes with Volumes

Lab0 - Lab semantics

Nomenclatures

Shell Commands

The commands that you are going to execute to progress the Labs will look like this (you DO NOT have to execute this!):

THIS IS AN EXAMPLE - DO NOT EXECUTE THIS!

```Bash
kubectl create -f redis-slave-service.yaml

 > Output Line 1
 > Output Line 2
 > Output Line 3
 ...
```

> **IMPORTANT NOTE:** The example output of a command is prefixed by ">" in order to make it more distinguishable.
>
> So in the above example you would only enter/copy-paste `kubectl create -f redis-slave-service.yaml` and the output from the command is "Output Line 1" to "Output Line 3"

Code Examples

Code examples are presented like this:

```YAML
apiVersion: lab.ibm.com/v1beta1
kind: MyResource
metadata:
  name: example
spec:
  size: 3
  image: busybox
```

This is only for illustration and is not being actively used in the Labs.

Lab 0 - Prepare the Lab environment

Before starting the Labs, let's make sure that we have the latest source code from the GitHub repository:

https://github.com/niklaushirt/training

1. Open a Terminal window by clicking on the Termnial icon in the left sidebar - we will use this extensively later as well

2. Execute the following commands to initialize your Training Environment

   ```bash
   ./welcome.sh
   ```

   This will

   - pull the latest example code from my GitHub repository
   - start minikube if not already running
   - installs the registry
   - installs the Network Plugin (Cilium)
   - starts the Personal Training Environment

   > During this you will have to provide a name (your name) that will be used to show your progress in the Instructor Dashboard in order to better assist you.

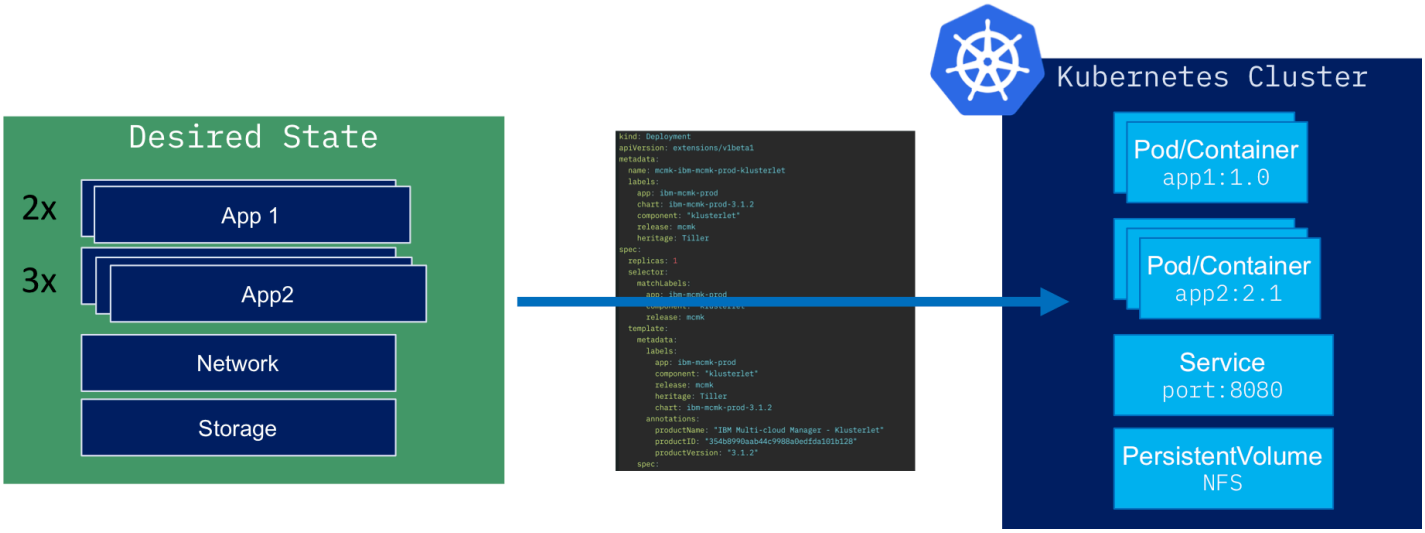
Lab 1. Get to know Kubernetes

Kubernetes was developed by Google as part of the Borg project and handed off to the open source community in 2015. Kubernetes combines more than 15 years of Google research in running a containerized infrastructure with production work loads, open source contributions, and Docker container management tools to provide an isolated and secure app platform that is portable, extensible, and self-healing in case of failovers.

Kubernetes is a solution that automates the orchestration of Container workloads. It is declarative, which means that you define the desired state and Kubernetes ensures that the state is matched at any given moment. For example you want to deploy three instances of a Container and one dies it will automatically restart a new one to match the desired state.

If you want to learn more about Kubernetes, please follow Course JTC80 Kubernetes Introduction.

In order to define the desired state, we use Resources (aka Objects). There are a lot of different types of Resources but the most important ones that we will be using are: * Pods - Smallest deployment unit, usually runs one Container inside * ReplicaSets - Controls the number of Pods running * Deployment - defines the deployment of a certain Container - creates a ReplicaSet * Service - defines how to expose the Container on the network

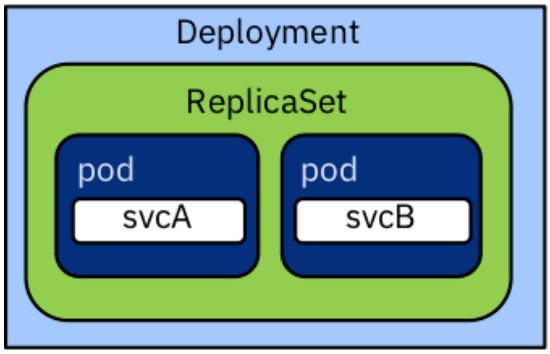


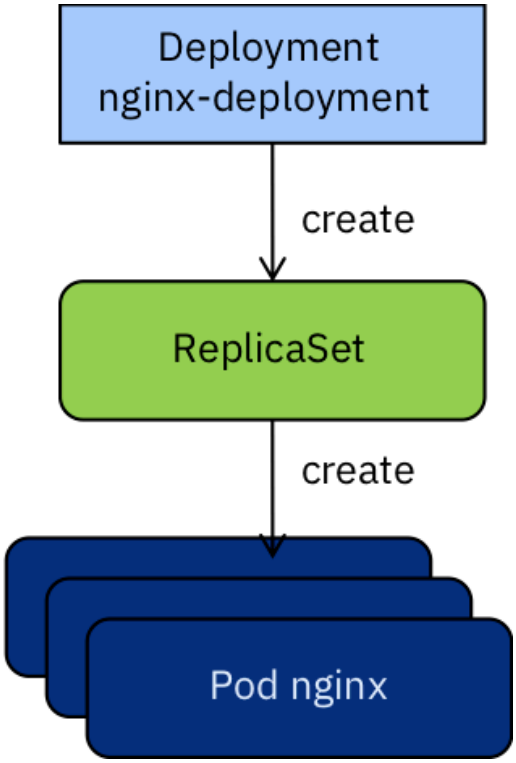
Deployment

A Deployment provides declarative updates for Pods (and ReplicaSets).

You describe a desired state in a Deployment, and the Deployment Controller changes the actual state to the desired state at a controlled rate. You can define Deployments to create new ReplicaSets, or to remove existing Deployments and adopt all their resources with new Deployments.

```yaml
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 2
  template:
     metadata:
       labels:
         app: nginx
     spec:
       containers:
       - name: nginx
          image: nginx:1.7.9
          ports:
          - containerPort: 80
```





This yaml file defines (amongst others):

Field	Description
metadata.name	Name of the Deployment
spec.replicas	The number of Pods to run simultaneuosly
spec.template.spec.containers.image	The Container image to run
spec.template.spec.containers.ports	The networking ports that should be exposed

More information on Deployments can be found here.

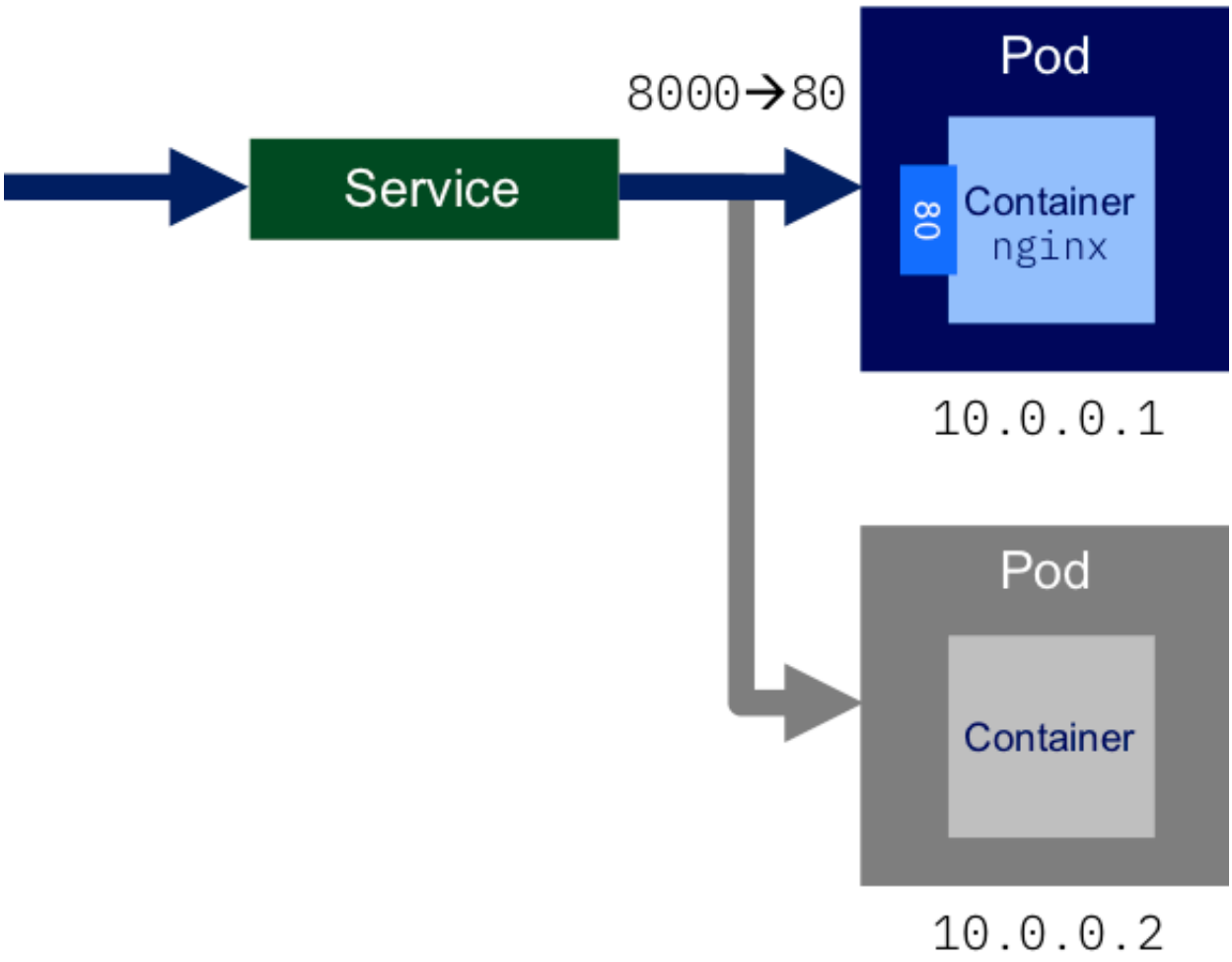
Service

An abstract way to expose an application running on a set of Pods as a network service.

With Kubernetes you don’t need to modify your application to use an unfamiliar service discovery mechanism. Kubernetes gives Pods their own IP addresses and a single DNS name for a set of Pods, and can load-balance across them.

YAML

```yaml
kind: Service
metadata:
  name: nginx-service
spec:
  ports:
  - port: 8000
    targetPort: 80
    protocol: TCP
  selector:
    app: nginx
```



This yaml file defines (amongst others):

Field	Description
metadata.name	Name of the Service
spec.ports	Port mapping to be exposed (port 80 of the container is exposed as port 8080)
spec. selector.app	Defines the Deployment to map the Servcie to

More information on Services can be found here.

Lab 2. Deploy your first Pod

Introduction

You will learn what a pod is, deploy your first container, configure Kubernetes, and interact with Kubernetes in the command line.

The base elements of Kubernetes are pods. Kubernetes will choose how and where to run them. You can also see a `Pod` as an object that requests some CPU and RAM. Kubernetes will take those requirements and decide where to run them.

A `Pod` can be killed and restarted whenever the system has/wants to. So a `Pod` is **ephemeral** and it **will** be destroyed at some point.

1. Create the Pod

Let's start to deploy the `nginx` docker image. It's a simple webserver that is used widely in the Kubernetes world.

Bash

```
kubectl run nginx --image=nginx

> kubectl run --generator=deployment/apps.v1 is DEPRECATED and will be removed in
 a future version. Use kubectl run --generator=run-pod/v1 or kubectl create inste
ad.
> deployment.apps/nginx created
```

2. List the Pods

Now list all the `Pods` running in Kubernetes. `kubectl get` is the `ls` of Kubernetes.

Bash

```
kubectl get pods

> NAME                     READY   STATUS              RESTARTS   AGE
> nginx-755464dd6c-2rbgj   0/1     ContainerCreating   0          32s
```

Note the name of the Pod which is randomly generated!

3. Get the yaml manifest

Now we can have a look at the `yaml` description of our `Pod`

```bash
kubectl get pods <name of pod from step 2> -o yaml

> apiVersion: v1
> kind: Pod
> metadata:
>   creationTimestamp: "2019-11-14T08:51:33Z"
>   generateName: nginx-755464dd6c-
>   labels:
>     pod-template-hash: 755464dd6c
>     run: nginx
>   name: nginx-755464dd6c-2rbgj
>   namespace: default
>   ownerReferences:
...
```

Use the name of the Pod that you wrote down in step #2.

Here is a simplified version that contains the main elements:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-755464dd6c-2rbgj
spec:
  containers:
  - name: nginx
    image: nginx  #hosted on hub.docker.com
```

The Kubernetes manifest represents a desired state. We do not write the steps to achieve this state. Kubernetes will handle it for us.

Let's have a look at the most important fields:

- `apiVersion` : the version of the Kubernetes API we will be using, `v1` here
- `kind` : what resource this object represents, `Pod` for this example
- `metadata` : some metadata about this pod, like its name.
- `spec` : specification of the desired behavior of this `Pod`
    - `containers` : the list of containers to start in this `Pod`
    - `name` : the name of the container
    - `image` : which image to start

4. Describe the Pod

`Describe` gives us a lot of information about the status and events of a Kubernetes object.

Bash

```
kubectl describe pods <name of pod from step 2>

...

Events:
  Type    Reason     Age   From               Message

----    ------     ----  ----               -------

  Normal  Scheduled  90s   default-scheduler  Successfully assigned default/nginx-75
minikube
  Normal  Pulling    82s   kubelet, minikube  Pulling image "nginx"
  Normal  Pulled     36s   kubelet, minikube  Successfully pulled image "nginx"
  Normal  Created    36s   kubelet, minikube  Created container nginx
  Normal  Started    34s   kubelet, minikube  Started container nginx
```

5. Clean-up

Now to wrap this up, let's delete the `Pod`

Bash

```
kubectl delete deployment nginx
```

Congratulations!!! This concludes Lab 2 on deploying your first Pod

Lab 3. Set up and deploy your first application

Learn how to deploy an application to a Kubernetes cluster.

Once your client is configured, you are ready to deploy your first application, `k8sdemo`.

Lab 3 - Deploy the frontend application

In this part of the lab we will deploy an application called `k8sdemo` that has already been built and uploaded to DockerHub under the name `niklaushirt/k8sdemo`.

We will use the following yaml:

YAML
```yaml
kind: Deployment
metadata:
  name: k8sdemo
  namespace: default
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: k8sdemo
    spec:
      containers:
      - name: k8sdemo
        image: niklaushirt/k8sdemo:1.0.0
    ...
          env:
          - name: PORT
            value : "3000"
          - name: APPLICATION_NAME
            value: k8sdemo
          - name: BACKEND_URL
            value: http://k8sdemo-backend-service.default.svc:3000/api
```

It defines the Container image to deploy (with 1 instances/replicas) and an environment variable BACKEND_URL that points to the backend servcie we will be deploying in the next section.

If you wish you can have a look at the complete yaml file:

Bash
```bash
gedit ~/training/deployment/demoapp.yaml
```

Now let's deploy it

1. Start by running `k8sdemo`

   ```Bash
   kubectl create -f ~/training/deployment/demoapp.yaml
   ```

   This action will take a bit of time. To check the status of the running application, you can use `kubectl get pods`.

   You should see output similar to the following:

   ```Bash
   kubectl get pods -n default

   > NAME                       READY     STATUS    RESTARTS   AGE
   > k8sdemo-7d46f69d68-bd2cw   0/1       Running   0          17s
   ```

2. Eventually, the status should show up as `1/1 Running`.

   ```Bash
   kubectl get pods -n default

   > NAME                       READY     STATUS              RESTARTS   AGE
   > k8sdemo-7d46f69d68-bd2cw   1/1       Running             0          5m
   ```

   The end result of the run command is to create a Deployment resource that manages the lifecycle of those pods.

3. Once the status reads `Running`, we need to expose that deployment as a service so we can access it through the IP of the worker nodes. The `k8sdemo` application listens on port 3000.

   Run:

   ```Bash
   kubectl expose deployment k8sdemo --name k8sdemo-service -n default --type="No
   dePort" --port=3000

   > service "k8sdemo-service" exposed
   ```

4. To find the port used on that worker node, examine your new service:

```bash
kubectl get service -n default k8sdemo-service

> NAME              TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)
AGE
> k8sdemo-service   NodePort   10.109.30.227   <none>        3000:30931/TCP
11m
```

We can see that our in this example `<nodeport>` is `31208`. We can see in the output the port mapping from 3000 inside the pod exposed to the cluster on port 30931. This port in the 31000 range is automatically chosen, and **will probably be different for you**.

5. `k8sdemo` is now running on your cluster, and exposed to the internet. You can open the webpage directly by typing:

```bash
minikube service k8sdemo-service
```

where k8sdemo-service is the name of the exposed kubernetes service.

6. Your `k8sdemo` should now be open in the your default browser. However it will show an error, because we have not yet deployed the backend.

Testing DEMO_API STATUS: ERROR
Trying to reach backend ....

Lab 3 - Deploy the application backend

In this part of the lab we will deploy the application backend called `k8sdemo-backend` that has already been built and uploaded to DockerHub under the name `niklaushirt/k8sdemo-backend`.

1. Start by running `k8sdemo-backend`

   ```Bash
   kubectl create -f ~/training/deployment/demoapp-backend.yaml
   ```

   This action will take a bit of time. To check the status of the running application, you can use `kubectl get pods`.

   You should see output similar to the following:

   ```Bash
   kubectl get pods -n default

   > NAME                                READY     STATUS    RESTARTS   AGE
   > k8sdemo-7d46f69d68-xcgcw            0/1       Running   0          13m
   > k8sdemo-backend-9c777544b-cp59q     0/1       Running   0          1m
   > k8sdemo-backend-9c777544b-jqjz9     0/1       Running   0          1m
   ```

   Eventually, the status should show up as `1/1 Running`.

   ```Bash
   kubectl get pods -n default

   > NAME                                READY     STATUS    RESTARTS   AGE
   > k8sdemo-7d46f69d68-xcgcw            1/1       Running   0          13m
   > k8sdemo-backend-9c777544b-cp59q     1/1       Running   0          6m35s
   > k8sdemo-backend-9c777544b-jqjz9     1/1       Running   0          6m35s
   ```

   The end result of the run command is to create a Deployment resource that manages the lifecycle of those pods.

2. Once the status reads `Running`, we need to expose that deployment as a service so we can access it through the IP of the worker nodes. The `k8sdemo` application listens on port 3000. Run:

   ```Bash
   kubectl expose deployment k8sdemo-backend --name k8sdemo-backend-service -n de
   fault --type="NodePort" --port=3000

   > service "k8sdemo-service" exposed
   ```

3. Now reload the webpage and verify, that it shows

   Testing DEMO_API STATUS: OK

   ....

4. If you reload the webpage several times, you should see, tht the IP Address of the backend API Pod is changing between the two Pods that have been spun up.

Congratulations!!! This concludes Lab 3 on deploying a web application to Kubernetes

We will be using this deployment in the following Labs.

Lab 4: Scale and Update Deployments

In this lab, you'll learn how to update the number of instances a deployment has and how to modify the API backend.

> For this lab, you need a running deployment of the `k8sdemo` application from the previous lab. If you deleted it, recreate it.

Scale apps with replicas

A *replica* is a copy of a pod that contains a running service. By having multiple replicas of a pod, you can ensure your deployment has the available resources to handle increasing load on your application.

1. `kubectl` provides a `scale` subcommand to change the size of an existing deployment. Let's increase our capacity from a single running instance of `k8sdemo` up to 10 instances:

   ```
   Bash
   kubectl scale --replicas=4 deployment k8sdemo-backend -n default

   > deployment "k8sdemo" scaled
   ```

   Kubernetes will now try to make reality match the desired state of 4 replicas by starting 2 new pods with the same configuration as the first.

2. To verify that your changes have been rolled out, you can run:

   ```
   Bash
   kubectl get pods -n default

   > NAME                              READY     STATUS    RESTARTS   AGE
   > k8sdemo-7d46f69d68-xcgcw          1/1       Running   0          19m
   > k8sdemo-backend-9c777544b-cp59q   1/1       Running   0          12m
   > k8sdemo-backend-9c777544b-jqjz9   1/1       Running   0          12m
   > k8sdemo-backend-9c777544b-lwssx   1/1       Running   0          12m
   > k8sdemo-backend-9c777544b-t5mlq   1/1       Running   0          12m
   ```

   You should see output listing 4 replicas of your deployment.

Congratulations!!! This concludes Lab 4 on scaling and updating Deployments

Lab 5: Stateful Deployments

As you know a `Pod` is mortal, meaning it can be destroyed by Kubernetes anytime, and with it it's local data, memory, etc. So it's perfect for stateless applications. Of course, in the real world we need a way to store our data, and we need this data to be persistent in time.

So let's have a look on how how can we deploy a stateful application with a persistent storage in Kubernetes?

For this Lab we will deploy a mysql application.

Persistent Volumes

As stated above, the state of a `Pod` is destroyed with it, so it's lost. For a database we need to be able to keep the data between restarts of the `Pods`.

That's where the `PersistentVolume` comes in.

As we have seen, a `Pod` as something that requests CPU & RAM.

A `PersistentVolume` as something that provides a storage on disk. Kubernetes handles a lot of different kinds of volumes. From local disk storage to s3, over 25 as of this writing.

First we create the `PersistentVolume` where our mysql data will be stored. It is a piece of storage in the cluster that has been provisioned by a cluster administrator. It is a resource in the cluster just like a node is a resource of the cluster.

YAML

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mysql-pv-volume
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
```

Let's review some parameters:

- `capacity` : the capacity of the volume

    - `storage` : the volume size - here `1Gb`
- `accessModes` : how this volume will be accessed, here ReadWriteOnce
    - `ReadWriteOnce` : the volume can be mounted as read-write by a single node
    - `ReadOnlyMany` : the volume can be mounted read-only by many nodes
    - `ReadWriteMany` : the volume can be mounted as read-write by many nodes
- `hostPath` : where the storage will be stored on the host, here `/mnt/data` (this is specific to the storage type)

We will use `hostPath` for this example, it is a simple type of `Volume` but you should **never** use it in production environements!

1. Create the PersistentVolume

```bash
kubectl apply -f ~/training/volumes/1-simple-mysql-pv.yaml

> persistentvolume/mysql-pv-volume created
```

2. Check that the Volume has been created

```bash
kubectl get pv

> NAME              CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLA
IM   STORAGECLASS   REASON   AGE
> mysql-pv-volume   1Gi          RWO            Retain           Available
                            4s
```

PersistentVolumeClaims

Now that we have a storage, we need to claim it, make it available for our `Pods` . So we need a `PersistentVolumeClaim` . It is a request for storage by a user. It is similar to a pod. Pods consume node resources and `PersistentVolumeClaim` consume `PersistentVolume` resources.

A `PersistentVolumeClaim` as something that requests a storage on disk and makes it available for the `Pod` think of it as an abstraction over the hard drives of the Kubernetes nodes - a fancy name for local hard drive.

The manifest for the `PersistentVolumeClaim` is quite similar to the `PersistentVolume` :

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pv-claim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

1. Create the PersistentVolume

```bash
kubectl apply -f ~/training/volumes/2-simple-mysql-pvc.yaml

> persistentvolumeclaim/mysql-pv-claim created
```

2. Check that the Volume has been created. Make sure that the STATUS reads Bound!

```bash
kubectl get pvc

> NAME             STATUS   VOLUME                                     CAPACIT
Y   ACCESS MODES   STORAGECLASS   AGE
> mysql-pv-claim   Bound    pvc-918d5a94-06c7-11ea-80af-080027c4f0ca   1Gi
    RWO            standard       5s
```

Create the stateful deployment

Now let's create the stateful `deployment` for mysql.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: mysql
spec:
 selector:
  matchLabels:
   app: mysql
 template:
  metadata:
   labels:
    app: mysql
  spec:
   containers:
  - image: mysql:5.6
     name: mysql
     env:
      - name: MYSQL_ROOT_PASSWORD
       value: password
     ports:
      - containerPort: 3306
       name: mysql
     volumeMounts:
       - name: mysql-persistent-storage
       mountPath: /var/lib/mysql
   volumes:
     - name: mysql-persistent-storage
     persistentVolumeClaim:
      claimName: mysql-pv-claim
```

We define several things here:

- `env` : the list of environment variables to pass to the container
  - `name` : the name of the env variable
  - `value` : the value of the env variable Here we pass the mysql root password to the container
- `volumeMounts` : the volumes, that will be mapped into the container
  - `name` : the name of the volume to mount
  - `mountPath` : where in the container to mount the volume Here we map out the path where mysql stores its database files
- `volumes` : the volumes to request access to
  - `name` : the name of the volume, same as `volumeMounts.name`
  - `persistentVolumeClaim` : the PersistentVolumeClaim we want

- `claimName` : the name of the claim

1. Create the `Deployment` (this creates the deployment as well as the service to access the mysql container)

```bash
kubectl apply -f ~/training/volumes/3-simple-mysql-deployment.yaml

> deployment.apps/mysql created
> service/mysql created
```

2. Verify that the `Deployment` has been created and is running (READY must be 1/1)

```bash
kubectl get deployment

> NAME    READY   UP-TO-DATE   AVAILABLE   AGE
> mysql   0/1     1            0           8s
```

Access the mysql database

1. Now let's access the mysql container by running a mysql client in a temporary container:

```bash
kubectl run -it --rm --image=mysql:5.6 --restart=Never mysql-client -- mysql -
h mysql -ppassword

If you don't see a command prompt, try pressing enter.

mysql>
```

2. Create a new database in mysql:

```bash
mysql> CREATE DATABASE testing;

> Query OK, 1 row affected (0.01 sec)
```

3. Check that it has been created

```
mysql> show databases;

> +--------------------+
> | Database           |
> +--------------------+
> | information_schema |
> | mysql              |
> | performance_schema |
> | testing            |
> +--------------------+
> 4 rows in set (0.00 sec)
```

4. Exit the mysql client

```
mysql> exit

> Bye
> pod "mysql-client" deleted
```

5. Get the list of Pods

```
kubectl get pods

> NAME                     READY   STATUS    RESTARTS   AGE
> mysql-7b9b7999d8-v4r5l   1/1     Running   0          4m26s
```

6. Delete the Pod (you'll have to use the name of your Pod, which is dynamic)

```
kubectl delete pod mysql-7b9b7999d8-v4r5l

> pod "mysql-7b9b7999d8-v4r5l" deleted
```

7. A new Pod has been spun up

```
kubectl get pods

> NAME                     READY   STATUS    RESTARTS   AGE
> mysql-7b9b7999d8-vqln7   1/1     Running   0          18s
```

8. Recreate a mysql client

```
kubectl run -it --rm --image=mysql:5.6 --restart=Never mysql-client -- mysql -
h mysql -ppassword

If you don't see a command prompt, try pressing enter.

mysql>
```

9. Verify that the database testing is still there because it has been persisted in the PV through the PVC.

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| testing            |
+--------------------+
4 rows in set (0.00 sec)
```

10. Exit the mysql client

```
mysql> exit

> Bye
> pod "mysql-client" deleted
```

Conclusion

- We have created a PersistentVolume that represents physical storage in the Kubernetes cluster.
- We have created a PersistentVolumeClaim that maps this volume like a virtual harddisk and makes it available to Pods
- We have created a mysql deployment that persists its data to the PersistentVolume
- We have connected to the mysql instance and created a new database within
- We have killed the mysql Pod, which would normally make it loose the data (the new database)
- We have reconnected to the mysql instance to verify that the newly created database is still present

Congratulations!!! This concludes Lab 5 on stateful Deployments

Hint Lab5_Volumes

No hint available

Complete Lab5_Volumes

Confirm Lab5_Volumes complete

Task Lab6_Cleanup

Clean-up

Delete the elements that we have deployed in order to go back to normal.

1. Delete the demo app and the mysql deployment

```bash
kubectl delete -f ~/training/deployment/demoapp.yaml
kubectl delete -f ~/training/deployment/demoapp-service.yaml
kubectl delete -f ~/training/deployment/demoapp-backend.yaml
kubectl delete -f ~/training/deployment/demoapp-backend-service.yaml
kubectl delete -f ~/training/volumes/3-simple-mysql-deployment.yaml
```