Volumes, Ingress, Helm as well as other workloads

Manifests are under `classfiles/k8s-2`.

1. Create a Persistent Volume

Based on https://kubernetes.io/docs/tasks/configure-pod-container/configure-persistent-volume-storage/

As admin, create a Persistent Volume

You will create a simple hostPath volume on the Minikube server. This server has an empty /data directory where the PV resides.

This is the manifest `pv.yaml` for your first persistent volume:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: myvol
  labels:
    type: local
spec:
  storageClassName: testing
  capacity:
    storage: 2Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/data/website"
```

Analyze a few details of this manifest.

- The storage class is set to *testing*. When a user makes a persistent volume claim of the same storage class, this volume may be used to satisfy ("bind") the claim.
- `ReadWriteOnce` means that the volume can be mounted read/write, but only by one container at a time.
- The volume is of type `hostPath`, which means that it corresponds to a directory on the pod's host.
  Be aware that this is **not suitable for production**.

Create the volume.

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

Check if it was created on the Minikube host. Log on with `minikube ssh` and list `/data`. Or in a single command:
`ssh $(minikube ip) -i $(minikube ssh-key) -l docker ls /data`. You will find that there is nothing.

Have a closer look at the volume

```
$ kubectl get pv
...
  persistentVolumeReclaimPolicy: Retain
  storageClassName: testing
  volumeMode: Filesystem
status:
  phase: Available
$ kubectl describe pv myvol
```

Compare the output of the two commands. The format is different, but the content is almost identical.

A few details are interesting. The **`persistentVolumeReclaimPolicy`** specifies what happens with a PV when it is released from its PVC. `myvol` has a policy of `Retain`, which means the PV and the data on it will be kept until the administrator releases it manually.
Other possible values are `Delete` (delete the PV) and `Recycle` (remove the data but keep the PV). `Recycle` is deprecated; instead, dynamic provisioning is recommended.

The **`volumeMode`** indicates that the volume is supposed to be used with a filesystem, not as a raw block device.

The **`phase`** is `available`, which indicates that the volume is currently not claimed. During the volume's life, it may be `bound` (i.e., claimed), then `released`. A PV can also be in the `failed` phase, which means that something went wrong when claiming it.

2. Create and use a Persistent Volume Claim

Use the `pvc.yaml` manifest for claiming storage:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mypvc
spec:
  storageClassName: testing
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

With this manifest, you make a claim for 1GB, read-write, attachable to a single container.
Note the details that you do not provide: Nothing about location, identity, name etc. of the volume. All you request is the size and a mode of read-write-once.

```
$ kubectl apply -f pvc.yaml
$ kubectl get pvc
NAME    STATUS   VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS   AGE
mypvc   Bound    myvol    2Gi        RWO            testing        19s
$ kubectl get pvc mypvc -o yaml
$ kubectl describe pvc mypvc
Name:          mypvc
Namespace:     default
StorageClass:  testing
Status:        Bound
Volume:        myvol
Labels:        <none>
Annotations:   pv.kubernetes.io/bind-completed: yes
               pv.kubernetes.io/bound-by-controller: yes
Finalizers:    [kubernetes.io/pvc-protection]
Capacity:      1Gi
Access Modes:  RWO
VolumeMode:    Filesystem
Used By:       <none>
Events:        <none>
```

This PVC is bound to volume `myvol`. The capacity is 1Gi, although `myvol` is larger than that. The volume is *bound*, but the claim is currently *not used* by a pod.

Check the volume's status. It should be bound as well.

```
$ kubectl get pv myvol
```

Again enter the minikube server with `minikube ssh` and check if there is anything in `/data`. It should be empty. Alternatively, use: `ssh $(minikube ip) -i $(minikube ssh-key) -l docker ls /data`.

Mount the volume in a pod

Use the pod.yaml manifest shown below.

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  imagePullSecrets:
  - name: regcred
  volumes:
    - name: podvol
      persistentVolumeClaim:
        claimName: mypvc
  containers:
    - name: mycontainer
      image: nginx
      ports:
        - containerPort: 80
          name: "http-server"
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: podvol
```

So far, your pods used an emptyDir volume, which is ephemeral and is deleted when the pod is deleted. The volume that this pod uses comes from the PVC that you just created. The volume's name, `podvol`, is used in the container's mount.

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

Again list `/data` on the Minikube server. Now, a subdirectory `website` exists.

Test the volume

Log on to the Minikube server and create a file named `/data/website/index.html` with the content `<h1>Success!!!</h1>`. You can do this in a single command:

```
$ echo '<h1>Success!!!</h1>' |
ssh $(minikube ip) -i $(minikube ssh-key) -l docker sudo tee /data/website/index.html
```

To confirm that this web page is really served by the NGINX pod, you need to create a service that enables external access. Copy the `nodePort.yaml` manifest from the k8s-1 directory.

To make the service work, you need to make a small change to the pod. Try to figure this out by yourself. The solution is in `pod-solution.yaml`.

Apply the nodePort manifest and test the web site with

```
$ curl $(minikube ip):30000
<h1>Success!!!</h1>
```

This proves that the PV is indeed used as web site by the NGINX pod.

3. Other volume exercises

Try making another claim

Create pvc2.yaml with `name: mypvc2` and the same storage class as `mypvc`. Apply and check it.

```
$ kubectl apply -f pvc2.yaml
$ kubectl get pvc
NAME                            STATUS    VOLUME    CAPACITY   ACCESS MODES   STORAGECLASS   AGE
persistentvolumeclaim/mypvc     Bound     myvol     2Gi        RWX            testing        4m44s
persistentvolumeclaim/mypvc2    Pending                                       testing        11s
```

Pending means that there is currently no storage available to bind this PVC. A new PV has to be created first; until then, it can't be used by a pod.

Delete the pod and the PVC

```
$ kubectl delete -f pod.yaml
$ kubectl get all
$ ssh $(minikube ip) -i $(minikube ssh-key) -l docker cat  /data/index.html
<h1>Success!!!</h1>
```

This proves that the volume still exists after deleting the pod, and that its content is untouched.

```
$ kubectl delete pvc mypvc
$ kubectl get pv,pvc
```

The PV has status released, which prevents it to be used by another claim. This is so to protect any data that might be stored on it.

```
$ kubectl -f pvc.yaml
$ kubectl get pv,pvc
```

The new PVC is Pending since the PV can't be bound and there is no other PV that could be used.

Connect the volume to a Deployment

Delete all PVCs, PVs and pods.

A volume can be shared amoung pods and allows you to create a web site that is handled by several web servers. To enable this, **change the access mode** in both PV and PVC to `ReadOnlyMany` and apply them.

Copy `development.yaml` from the previous exercise and modify the volume part at the very end. Replace `emptyDir` with `persistentVolumeClaim` as follows:

```
      volumes:
      - name: data
        persistentVolumeClaim:
          claimName: mypvc
```

Launch the deployment. If the nodePort service doesn't exist anymore, recreate it, then test the web site with `curl $(minikube ip):30000.`

4. Dynamic provisioning

So far, somebody has to create PVs in order to satisfy volume claims. This is a very manual and cumbersome method. Volumes can also be claimed dynamically, i.e. a PV is allocated from a storage pool at the moment the PVC is created. The Minikube server has a dynamic storage provider available as storage class *standard*.

Create a new PVC manifest dynpvc.yaml

It uses the Minikube standard storage class.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mydynpvc
spec:
  storageClassName: standard
  accessModes:
    - ReadOnlyMany
  resources:
    requests:
      storage: 1Gi
```

Apply it.

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

Confirm that this created a new PV

```
$ kubectl describe pvc mydynpvc
...
Events:
  Type    Reason                Age                From
Message
  ----    ------                ----               ----
-------
  Normal  ExternalProvisioning  18s (x2 over 18s)  persistentvolume-controller
waiting for a volume to be created, either by external provisioner "k8s.io/minikube-hostpath" or
manually created by system administrator
  Normal  Provisioning          18s                k8s.io/minikube-hostpath_minikube_09925dca-37b2-
4d3d-94f9-690658da929a  External provisioner is provisioning volume for claim "default/mydynpvc"
  Normal  ProvisioningSucceeded  18s                k8s.io/minikube-hostpath_minikube_09925dca-37b2-
4d3d-94f9-690658da929a  Successfully provisioned volume pvc-268cbfc8-87f3-4264-93ce-56db927a92be
$ kubectl get pv,pvc
NAME                                                        CAPACITY   ACCESS MODES   RECLAIM POLICY
STATUS   CLAIM             STORAGECLASS   REASON   AGE
persistentvolume/myvol                                      2Gi        ROX            Retain
Bound    default/mypvc     testing                 12m
persistentvolume/pvc-268cbfc8-87f3-4264-93ce-56db927a92be   1Gi        ROX            Delete
Bound    default/mydynpvc  standard                40s

NAME                              STATUS   VOLUME                                     CAPACITY   ACCESS
MODES   STORAGECLASS   AGE
persistentvolumeclaim/mydynpvc    Bound    pvc-268cbfc8-87f3-4264-93ce-56db927a92be   1Gi        ROX
standard       40s
persistentvolumeclaim/mypvc       Bound    myvol                                      2Gi        ROX
testing        13m
```

The creation of the PVC immediately triggers the automatic creation of a PV by Minikube's provisioning component *minikube-hostpath*.

```
$ kubectl describe pv pvc-268cbfc8-87f3-4264-93ce-56db927a92be
Source:
    Type:          HostPath (bare host directory volume)
    Path:          /tmp/hostpath-provisioner/default/mydynpvc
```

This reveals the precise location of the dynamically allocated PV.

5. Deploy Wordpress with persistent volumes and a secret

Analyze the WordPress application

The necessary manifests are in in `classfiles/k8s-2/wordpress`.

There are six files, three for the database, three for the website. `db-deploy.yaml`, `db-pvc.yaml` and `db-service.yaml` define a mysql deployment, claim a volume that is used for keeping the database, and a service to reach the database pod.

View `db-deploy.yaml` and answer these questions: Which environment variable is set? Where does its value come from? Which PVC is used, and where is it mounted?

View `db-pvc.yaml`. Compare the volume claim's name with the name in `db-deploy.yaml`. Notice that the storage class is not specified, so that the default storage class in the Kubernetes cluster will be used. In Minikube, the default is `standard`; it's a storage class with dynamic provisioning.

View `db-service.yaml`. Notice its name. What service type is it?
The last line, `ClusterIP: None`, produces a service without IP address, a so-called **headless** service.

Now check the files for the website.

View `wp-deploy.yaml`. Notice that the same environment variable is set as in the DB deployment. There is a second variable. What is it for, and what is its value? Compare the value with the content of `db-service.yaml`.

View `wp-pvc.yaml`. Apart from the name, it should be the same as for the database.

View `wp-service.yaml`. Which service type is this? How can the service be reached over the network?

Launch the application and fail

The easiest way to deploy all six manifests is

```
$ kubectl apply -f .
```

Don't forget the dot at the end. It refers to the current directory. This command applies all YAML files it finds here.

Check success:

```
$ kubectl get all
NAME                                   READY   STATUS                       RESTARTS   AGE
pod/wordpress-6b7855f7dc-blk69         0/1     CreateContainerConfigError   0          2m58s
pod/wordpress-mysql-6c479567b-xn8dm    0/1     CreateContainerConfigError   0          2m58s
...
```

The pods are in error. What is the problem? Get a detailed description.

```
$ kubectl describe pod/wordpress-6b7855f7dc-blk69
Events:
  Type     Reason            Age                  From               Message
  ----     ------            ----                 ----               -------
  Warning  FailedScheduling  3m18s                default-scheduler  0/1 nodes are available: 1
persistentvolumeclaim "wp-pv-claim" not found.
  Normal   Scheduled         3m17s                default-scheduler  Successfully assigned default/wordpress-
6b7855f7dc-blk69 to minikube
  Normal   Pulling           3m17s                kubelet            Pulling image "wordpress:4.8-apache"
  Normal   Pulled            2m53s                kubelet            Successfully pulled image "wordpress:4.8-
apache" in 23.730995879s
  Warning  Failed            49s (x12 over 2m53s)  kubelet            Error: secret "mysql-pass" not found
```

The first warning, *FailedScheduling*, is not a problem. The pod wants to start running before its persistent volume is ready. It becomes ready in the next message, *Scheduled*.

However, the last line is the error that causes the pod to fail. What is this secret?

Create a secret containing the DB password

Both deployments rely on a Kubernetes secret to configure the database password:

```
    spec:
      containers:
        ...
        env:
        - name: MYSQL_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysql-pass
              key: password
```

The secret must be named *mysql-pass* and the password must be the value of a key named *password*.
Instead of creating a manifest, use the command line:

```
$ kubectl create secret generic mysql-pass --from-literal=password=pw
$ kubectl get secret
```

Launch the application and test it

Try again.

```
$ kubectl apply -f .
```

A quick check if everything is up and running:

```
$ kubectl get all
$ kubectl get pv,pvc
```

Notice this service:

```
service/wp-service     NodePort    10.101.129.15   <none>        80:31234/TCP   116s
```

The WordPress website can be accessed by the Minikube node's IP address and port 31234. Use **`minikube ip`** to retrieve the IP address and try it in the browser.

Minikube offers a shortcut. It opens the browser window with this command: **`minikube service wp-service`**. Try this as well.

6. Add an ingress to WordPress

The current URL for the WordPress site is ugly. It consists of the Minikube host's IP address and a random port number. An ingress allows mapping a nicer URL to the WordPress service.

Prepare for the ingress

Delete everything except the secret. Don't forget that these commands literally delete everything!

```
$ kubectl delete all --all
```

The volumes remain. You need to delete them separately.

```
$ kubectl delete pv,pvc --all
```

Minikube doesn't include an ingress by default. You need to add it. Optionally, view complete instructions.

```
$ minikube addons enable ingress
$ kubectl get pods -n ingress-nginx
NAME                                        READY   STATUS      RESTARTS   AGE
ingress-nginx-admission-create--1-7pg6d     0/1     Completed   0          2m42s
ingress-nginx-admission-patch--1-9zdnf      0/1     Completed   1          2m42s
ingress-nginx-controller-69bdbc4d57-l884x   1/1     Running     0          2m42s
```

The second command may only be successful after a few seconds. The Ingress controller runs as a pod in its own namespace *ingress-nginx*. Find out more about it.

```
$ kubectl describe pod ingress-nginx-controller-69bdbc4d57-l884x -n ingress-nginx | grep Controlled
Controlled By:  ReplicaSet/ingress-nginx-controller-69bdbc4d57
$ kubectl describe ReplicaSet/ingress-nginx-controller-69bdbc4d57 -n ingress-nginx | grep Controlled
Controlled By:  Deployment/ingress-nginx-controller
```

The controller is actually embedded in a deployment.

Create an ingress for the WordPress site

Copy the ingress to the current directory and inspect it.

```
$ cp ingress/wp-ingress.yaml .
$ cat wp-ingress.yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: wp-ingress
spec:
  rules:
  - host: wpsite.local
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: wp-service
            port:
              number: 80
```

This ingress looks for HTTP requests from a certain host, *wpsite.local*. Such requests are redirected to the wp-service.

Add host name resolution

To access the WordPress web site, hostname *wpsite.local* must resolve to the Minikube IP address. You will add a line to /etc/hosts to achieve this. Prior to this, make a backup copy of /etc/hosts.

```
$ minikube ip    # use this address in the echo command below
$ sudo -i
# cp /etc/hosts /etc/hosts.backup
# echo MINIKUBE_IP_ADDRESS wpsite.local >> /etc/hosts
# exit
```

Apply and test the ingress manifest

```
$ kubectl apply -f wp-ingress.yaml
```

Enter http://wpsite.local in your browser or use `curl -L wpsite.local` to access the WordPress site.

Add a second web site to the cluster

This time, use the command line for creating simple Kubernetes objects. Launch a deployment based on nginx, then expose a service to it.

```
$ kubectl create deployment web --image=nginx
$ kubectl expose deployment web --type=NodePort --port=80
$ kubectl get pod,svc
```

To test access to the second web site, use the service's node port, for example 31015:

```
web             NodePort    10.109.13.119   <none>        80:31015/TCP   55s
```

```
$ curl $(minikube ip):31015
```

You should get NGINX's default main page.

Alter the ingress to give access to the second web site via host name *othersite.local*. The file `ingress/othersite.yaml` contains the correct code.

```
$ cat ingress/othersite.yaml
spec:
  rules:
  - host: wpsite.local
...
  - host: othersite.local
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: web
            port:
              number: 80
$ kubectl apply -f ingress/othersite.yaml
```

You have to add othersite.local to /etc/hosts. It is sufficient to add it at the end of the last line. This command does the trick:

$ sudo sed -i "s/wpsite.local/wpsite.local othersite.local/" /etc/hosts

Check the result:

```
$ tail -n1 /etc/hosts
192.168.49.2 wpsite.local othersite.local
```

(your IP address will be different)

Now you have two sites hosted on your Kubernetes cluster. Put *othersite.local* into the browser's address bar to see the effect.

7. Deploy Wordpress with Helm

Remove all workloads and volumes. Never use these commands in a production environment:

```
$ kubectl delete all --all
$ kubectl delete pvc,pv --all
```

Launch WordPress from Helm charts

Go to the artifact hub at http://artifacthub.io and find the Bitnami WordPress description (direct link).

Add Bitnami's Helm repository.

```
$ helm repo add bitnami https://charts.bitnami.com/bitnami
$ helm repo list
```

Deploy a default WordPress application. Below, bitnami is the repo, and wordpress the chart. Read the output.

```
$ helm install myblog bitnami/wordpress
...
Your WordPress site can be accessed through the following DNS name from within your cluster:

    myblog-wordpress.default.svc.cluster.local (port 80)

To access your WordPress site from outside the cluster follow the steps below:

1. Get the WordPress URL by running these commands:

  NOTE: It may take a few minutes for the LoadBalancer IP to be available.
        Watch the status with: 'kubectl get svc --namespace default -w myblog-wordpress'
...
```

Explore the resources that were created and test access to the blog

Check which workload resources and services were created.

```
$ kubectl get all
```

What kind of workload is used for the WordPress site, and what kind for the database? What service types were created?

You should see that a WordPress and a MariaDB application are running. There is a LoadBalancer service which remains pending, since no loadbalancer is configured in the cluster:

```
myblog-wordpress    LoadBalancer    10.110.47.229    <pending>     80:31757/TCP,443:30342/TCP    8m49s
```

Access the blog using the Minikube IP address and the port number from the LoadBalancer service.

Download and inspect the Helm charts

```
$ helm pull bitnami/wordpress
```

After this command, you will find a file named `wordpress-12.1.20.tgz` (or similar) in the current directory.

Unpack it with `tar xf wordpress-12.1.20.tgz`. This creates a subdirectory `wordpress`. Under `wordpress`, the manifest templates are in the `templates` subdirectory, the default values for template variables in a file named `values.yaml`. Explore those files to get a feeling for Helm charts.

Delete the application

To stop an application that was created from Helm charts, just run this one command:

```
$ helm delete myblog
```

8. Try other workload types

Download example manifests from the kubernetes.io web site. You will have to edit the StatefulSet and remove the storage class specification, since there is no such storage class in the cluster. The Job and CronJob templates can (probably) be tried unchanged.

StatefulSet

Get the manifest from https://kubernetes.io/docs/concepts/workloads/controllers/statefulset and remove the storage class. Apply it and list the resources it creates. Enter one of the pods and install ping:

```
root@web-1:/# apt update && apt install iputils-ping -y
root@web-1:/# ping web-0.nginx
```

Job

```
$ kubectl apply -f https://kubernetes.io/examples/controllers/job.yaml
```

Watch the pod. It should be in a status of *Completed* eventually. To see the output, use `kubectl logs`.

CronJob

Get the manifest from https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/ and try it out. Again, use `kubectl logs` to see the output.