Persistent Installation of MySQL and WordPress on Kubernetes

This example describes how to run a persistent installation of WordPress and MySQL on Kubernetes. We'll use the mysql and wordpress official Docker images for this installation. (The WordPress image includes an Apache server).

Demonstrated Kubernetes Concepts:

- Persistent Volumes to define persistent disks (disk lifecycle not tied to the Pods).
- Services to enable Pods to locate one another.
- External Load Balancers to expose Services externally.
- Deployments to ensure Pods stay up and running.
- Secrets to store sensitive passwords.

Quickstart

Put your desired MySQL password in a file called `password.txt` with no trailing newline. The first `tr` command will remove the newline if your editor added one.

Note: if your cluster enforces ***selinux*** and you will be using Host Path for storage, then please follow this extra step.

```
tr --delete '\n' <password.txt >.strippedpassword.txt &&
mv .strippedpassword.txt password.txt
kubectl create -f https://raw.githubusercontent.com/kuber
netes/kubernetes/master/examples/mysql-wordpress-pd/local
-volumes.yaml
kubectl create secret generic mysql-pass --from-file=pass
word.txt
kubectl create -f https://raw.githubusercontent.com/kuber
netes/kubernetes/master/examples/mysql-wordpress-pd/mysql
-deployment.yaml
kubectl create -f https://raw.githubusercontent.com/kuber
netes/kubernetes/master/examples/mysql-wordpress-pd/wordp
ress-deployment.yaml
```

Table of Contents

<!-- BEGIN MUNGE: GENERATED_TOC -->

- Persistent Installation of MySQL and WordPress on Kubernetes
    - Quickstart

- Table of Contents
- Cluster Requirements
- Decide where you will store your data
    - Host Path
        - SELinux
    - GCE Persistent Disk
- Create the MySQL Password Secret
- Deploy MySQL
- Deploy WordPress
- Visit your new WordPress blog
- Take down and restart your blog
- Next Steps

<!-- END MUNGE: GENERATED_TOC -->

Cluster Requirements

Kubernetes runs in a variety of environments and is inherently modular. Not all clusters are the same. These are the requirements for this example.

- Kubernetes version 1.2 is required due to using newer features, such
  at PV Claims and Deployments. Run `kubectl version` to see your
  cluster version.
- Cluster DNS will be used for service discovery.
- An external load balancer

will be used to access WordPress.

- Persistent Volume Claims
  are used. You must create Persistent Volumes in your cluster to be
  claimed. This example demonstrates how to create two types of volumes, but any volume is sufficient.

Consult a
Getting Started Guide
to set up a cluster and the
kubectl command-line client.

Decide where you will store your data

MySQL and WordPress will each use a
Persistent Volume
to store their data. We will use a Persistent Volume Claim to claim an available persistent volume. This example covers HostPath and GCEPersistentDisk volumes. Choose one of the two, or see
Types of Persistent Volumes
for more options.

Host Path

Host paths are volumes mapped to directories on the host. **These should be used for testing or single-node clusters only**. The data

will not be moved between nodes if the pod is recreated on a new node. If the pod is deleted and recreated on a new node, data will be lost.

SELinux

On systems supporting selinux it is preferred to leave it enabled/enforcing.
However, docker containers mount the host path with the "*svirt_sandbox_file_t*" label type, which is incompatible with the default label type for /tmp ("*tmp_t*"), resulting in a permissions error when the mysql container attempts to `chown` */var/lib/mysql*.
Therefore, on selinx systems using host path, you should pre-create the host path directory (/tmp/data/) and change it's selinux label type to "*svirt_sandbox_file_t*", as follows:

```
## on every node:
mkdir -p /tmp/data
chmod a+rwt /tmp/data  # match /tmp permissions
chcon -Rt svirt_sandbox_file_t /tmp/data
```

Continuing with host path, create the persistent volume objects in Kubernetes using

local-volumes.yaml:

```
export KUBE_REPO=https://raw.githubusercontent.com/kubern
etes/kubernetes/master
kubectl create -f $KUBE_REPO/examples/mysql-wordpress-pd/
local-volumes.yaml
```

GCE Persistent Disk

This storage option is applicable if you are running on
Google Compute Engine.

Create two persistent disks. You will need to create the disks in the same GCE zone as the
Kubernetes cluster. The default setup script will create the cluster in the `us-central1-b` zone, as seen in the
config-default.sh file. Replace
`<zone>` below with the appropriate zone. The names `wordpress-1` and
`wordpress-2` must match the `pdName` fields we have specified in
gce-volumes.yaml.

```
gcloud compute disks create --size=20GB --zone=<zone> wor
dpress-1
gcloud compute disks create --size=20GB --zone=<zone> wor
dpress-2
```

Create the persistent volume objects in Kubernetes for those disks:

```
export KUBE_REPO=https://raw.githubusercontent.com/kubern
etes/kubernetes/master
kubectl create -f $KUBE_REPO/examples/mysql-wordpress-pd/
gce-volumes.yaml
```

Create the MySQL Password Secret

Use a Secret object
to store the MySQL password. First create a file (in the same directory as the wordpress sample files) called
`password.txt` and save your password in it. Make sure to not have a trailing newline at the end of the password. The first `tr` command will remove the newline if your editor added one. Then, create the Secret object.

```
tr --delete '\n' <password.txt >.strippedpassword.txt &&
mv .strippedpassword.txt password.txt
kubectl create secret generic mysql-pass --from-file=pass
word.txt
```

This secret is referenced by the MySQL and WordPress pod configuration
so that those pods will have access to it. The MySQL pod will set the

database password, and the WordPress pod will use the password to access the database.

Deploy MySQL

Now that the persistent disks and secrets are defined, the Kubernetes pods can be launched. Start MySQL using mysql-deployment.yaml.

```
kubectl create -f $KUBE_REPO/examples/mysql-wordpress-pd/mysql-deployment.yaml
```

Take a look at mysql-deployment.yaml, and note that we've defined a volume mount for `/var/lib/mysql`, and then created a Persistent Volume Claim that looks for a 20G volume. This claim is satisfied by any volume that meets the requirements, in our case one of the volumes we created above.

Also look at the `env` section and see that we specified the password by referencing the secret `mysql-pass` that we created above. Secrets can have multiple key:value pairs. Ours has only one key `password.txt` which was the name of the file we used to create the secret. The MySQL image sets the database password using the `MYSQL_ROOT_PASSWORD` environment variable.

It may take a short period before the new pod reaches the `Running` state. List all pods to see the status of this new pod.

```
kubectl get pods
```

```
NAME                              READY     STATUS    RESTART
S   AGE
wordpress-mysql-cqcf4-9q8lo   1/1       Running   0
    1m
```

Kubernetes logs the stderr and stdout for each pod. Take a look at the logs for a pod by using `kubectl log`. Copy the pod name from the `get pods` command, and then:

```
kubectl logs <pod-name>
```

```
...
2016-02-19 16:58:05 1 [Note] InnoDB: 128 rollback segment
(s) are active.
2016-02-19 16:58:05 1 [Note] InnoDB: Waiting for purge to
 start
2016-02-19 16:58:05 1 [Note] InnoDB: 5.6.29 started; log
sequence number 1626007
2016-02-19 16:58:05 1 [Note] Server hostname (bind-addres
s): '*'; port: 3306
2016-02-19 16:58:05 1 [Note] IPv6 is available.
```

```
2016-02-19 16:58:05 1 [Note]   - '::' resolves to '::';
2016-02-19 16:58:05 1 [Note] Server socket created on IP:
 '::'.
2016-02-19 16:58:05 1 [Warning] 'proxies_priv' entry '@ r
oot@wordpress-mysql-cqcf4-9q8lo' ignored in --skip-name-r
esolve mode.
2016-02-19 16:58:05 1 [Note] Event Scheduler: Loaded 0 ev
ents
2016-02-19 16:58:05 1 [Note] mysqld: ready for connection
s.
Version: '5.6.29'  socket: '/var/run/mysqld/mysqld.sock'
 port: 3306  MySQL Community Server (GPL)
```

Also in mysql-deployment.yaml we created a
service to allow other pods to reach this mysql instance. The name is `wordpress-mysql` which resolves to the pod IP.

Up to this point one Deployment, one Pod, one PVC, one Service, one Endpoint,
two PVs, and one Secret have been created, shown below:

```
kubectl get deployment,pod,svc,endpoints,pvc -l app=wordp
ress -o wide && \
  kubectl get secret mysql-pass && \
  kubectl get pv
```

```
NAME                    DESIRED   CURRENT   UP-TO-DATE
```

```
 AVAILABLE   AGE
deploy/wordpress-mysql   1         1         1
 1           3m
NAME                                READY     STATUS
 RESTARTS   AGE       IP            NODE
po/wordpress-mysql-3040864217-40soc   1/1       Running
 0          3m        172.17.0.2    127.0.0.1
NAME                CLUSTER-IP   EXTERNAL-IP   PORT(S)
   AGE       SELECTOR
svc/wordpress-mysql   None         <none>        3306/TCP
   3m        app=wordpress,tier=mysql
NAME                 ENDPOINTS          AGE
ep/wordpress-mysql   172.17.0.2:3306   3m
NAME                STATUS    VOLUME       CAPACITY   AC
CESSMODES   AGE
pvc/mysql-pv-claim   Bound     local-pv-2   20Gi       RW
O           3m
NAME         TYPE      DATA      AGE
mysql-pass   Opaque    1         3m
NAME         CAPACITY   ACCESSMODES   STATUS      CLAIM
                 REASON    AGE
local-pv-1   20Gi       RWO           Available
                           3m
local-pv-2   20Gi       RWO           Bound       default
/mysql-pv-claim            3m
```

Deploy WordPress

Next deploy WordPress using
wordpress-deployment.yaml:

```
kubectl create -f $KUBE_REPO/examples/mysql-wordpress-pd/
wordpress-deployment.yaml
```

Here we are using many of the same features, such as a volume claim for persistent storage and a secret for the password.

The WordPress image accepts the
database hostname through the environment variable `WORDPRESS_DB_HOST`. We set the env value to the name of the MySQL service we created: `wordpress-mysql`.

The WordPress service has the setting `type: LoadBalancer`. This will set up the wordpress service behind an external IP.

Find the external IP for your WordPress service. **It may take a minute
to have an external IP assigned to the service, depending on your
cluster environment.**

```
kubectl get services wordpress
```

```
NAME        CLUSTER-IP     EXTERNAL-IP     PORT(S)   AGE
```

```
wordpress   10.0.0.5       1.2.3.4         80/TCP    19h
```

Visit your new WordPress blog

Now, we can visit the running WordPress app. Use the external IP of the service that you obtained above.

```
http://<external-ip>
```

You should see the familiar WordPress init page.

> *Warning: Do not leave your WordPress installation on this page. If it is found by another user, they can set up a website on your instance and use it to serve potentially malicious content. You should either continue with the installation past the point at which you create your username and password, delete your instance, or set*
> *up a firewall to restrict access.*

Take down and restart your blog

Set up your WordPress blog and play around with it a bit. Then, take down its pods and bring them back up again. Because you used persistent disks, your blog state will be preserved.

All of the resources are labeled with `app=wordpress`, so you can easily bring them down using a label selector:

```
kubectl delete deployment,service -l app=wordpress
kubectl delete secret mysql-pass
```

Later, re-creating the resources with the original commands will pick up the original disks with all your data intact. Because we did not delete the PV Claims, no other pods in the cluster could claim them after we deleted our pods. Keeping the PV Claims also ensured recreating the Pods did not cause the PD to switch Pods.

If you are ready to release your persistent volumes and the data on them, run:

```
kubectl delete pvc -l app=wordpress
```

And then delete the volume objects themselves:

```
kubectl delete pv local-pv-1 local-pv-2
```

or

```
kubectl delete pv wordpress-pv-1 wordpress-pv-2
```

<!-- BEGIN MUNGE: GENERATED_ANALYTICS -->

<!-- END MUNGE: GENERATED_ANALYTICS -->