Crunchy Containers

Examples for the Openshift Environment

The examples/openshift directory containers examples for running the Crunchy containers in an Openshift environment.

The examples are explained below.

Openshift Example 1 - master.json

This openshift template will create a single master PostgreSQL instance.

Running the example:

```
oc create -f master.json | oc create -f -
```

You can see what passwords were generated by running this command:

```
oc describe pod pg-master | grep PG
```

You can run the following command to test the database, entering the value of PG_PASSWORD from above for the password when prompted:

```
psql -h pg-master.pgproject.svc.cluster.local -U testuser userdb
```

Openshift Example 2 - slave.json

This openshift template will create a single slave PostgreSQL instance that will connect to the master created in Example 1.

You will need to edit the slave.json file and enter the PG_MASTER_PASSWORD value generated in Example 1 as found in the master environment variables.

Running the example:

```
oc create -f slave.json | oc create -f -
```

You can run the following command to test the database, entering the value of PG_PASSWORD from above for the password when prompted:

```
psql -c 'select * from pg_stat_replication' -h pg-master.pgproject.svc.cluster.local
-U master userdb
psql -c 'create table foo (id int)' -h pg-master.pgproject.svc.cluster.local -U master
userdb
psql -c 'insert into foo values (123)' -h pg-master.pgproject.svc.cluster.local -U
master userdb
psql -c 'table foo' -h pg-slave.pgproject.svc.cluster.local -U master userdb
```

If replication is working, you should see a row returned in the pg_stat_replication table query and also a row returned from the pg-slave query.

Openshift Example 3 - pgpool for master slave examples

You can create a pgpool service that will work with the master and slave created in the previous Example 1 and Example 2.

You will need to edit the pgpool-for-master-slave-rc.json and supply the testuser password that was generated when you created the master slave pods, then run the following command to deploy the pgpool service:

```
oc process -f pgpool-rc.json | oc create -f -
```

Next, you can access the master slave cluster via the pgpool service by entering the following command:

```
psql -h pgpool-rc -U testuser userdb
```

when prompted, enter the password for the PG_USERNAME testuser that was generated for the pg-master pod.

At this point, you can enter SELECT and INSERT statements and pgpool will proxy the SQL commands to the master or slave(s) depending on the type of SQL command. Writes will always be sent to the master, and reads will be sent (round-robin) to the slave(s).

Openshift Example 4 - master-slave-rc.json

This example is similar to the previous examples but builds a master pod, and a single slave that can be scaled up using a replication controller. The master is implemented as a single pod since it can not be scaled like read-only slaves.

Running the example:

```
oc create -f master-slave-rc.json | oc create -f -
```

Connect to the PostgreSQL instances with the following:

```
psql -h pg-master-rc.pgproject.svc.cluster.local -U testuser userdb
psql -h pg-slave-rc.pgproject.svc.cluster.local -U testuser userdb
```

Here is an example of increasing or scaling up the Postgres 'slave' pods to 2:

```
oc scale rc pg-slave-rc-1 --replicas=2
```

Enter the following commands to verify the PostgreSQL replication is working.

```
psql -c 'table pg_stat_replication' -h pg-master-rc.pgproject.svc.cluster.local -U
master postgres
psql -h pg-slave-rc.pgproject.svc.cluster.local -U master postgres
```

You can see that the slave service is load balancing between multiple slaves by running a command as follows, run the command multiple times and the ip address should alternate between the slaves:

```
psql -c 'select inet_server_addr()' -h pg-slave-rc -U master postgres
```

Openshift Example 5 - Performing a Backup

This example assumes you have a master database pod running called pg-master as created by Openshift Example 1 and that you have configured NFS as decribed in the Prerequisites section of this document..

You can perform a database backup by executing the following step:

- as openshift admin, run:

```
oc create -f pv-backups.json
```

- as normal user, run:

```
oc create -f pvc-backups.json
oc create -f backup-job-nfs.json
```

A successful backup will perform pg_basebackup on the pg-master and store the backup in the NFS mounted volume under a directory named pg-master, each backup will be stored in a subdirectory with a timestamp as the name. This allows any number of backups to be kept.

The pv-backups.json specifies a ReclaimPolicy of Retain to tell Openshift that we want to keep the

volume contents after the removal of the PV.

Lastly, in order to rerun this same backup, you will need to remove the Job, PVC, and PV used in this example, this is done as follows:

- as admin user

```
oc delete pv backup-pg-master
```

- as normal user

```
oc delete pvc backup-claim-pg-master
oc delete job backupjob-pg-master
```

Openshift Example 5 - NFS Example

I have provided an example of using NFS for the postgres data volume. On my test nfs server, I had to set the exports file entry as follows:

```
/jeffnfs * (rw,insecure,sync)
```

First, you can only create persistent volumes as a cluster admin, you can login in as the admin user as follows:

```
oc login -u system:admin
```

To run it, you would execute the following as the openshift administrator:

```
oc create -f master-nfs-pv.json
```

Then as the normal openshift user account, create the Persistence Volume Claim and database pod as follows:

```
oc create -f master-nfs-pvc.json
oc process -f master-nfs.json | oc create -f -
```

This will create a single master postgres pod that is using an NFS volume to store the postgres data files.

Openshift Example 6 - Restore Example

I have provided an example of restoring a database pod using an existing backup archive located

on an NFS volume.

First, locate the database backup you want to restore, for example:

```
/jeffnfs/pg-master/2016-01-29:22:34:20
```

Next, * edit the master-restore-pv.json file to use that path in building the PV, * edit the master-restore-pv.json file to use a unique label * and then execute as the openshift superuser:

```
oc login -u system:admin
oc create -f master-restore-pv.json
```

Next, * edit the master-restore-pvc.json file, specify the same unique label used in the master-restore-pv.json file. * Then execute as the normal test user:

```
oc create -f master-restore-pvc.json
```

Next, create a database pod as the normal user:

```
oc process -f master-restore.json | oc create -f -
```

When the database pod starts, it will copy the backup files to the database directory inside the pod and start up postgres as usual.

The restore only takes place if:

- the /pgdata directory is empty
- the /backups directory contains a valid postgresql.conf file

Openshift Example 7 - Failover Example

An example of performing a database failover is described in the following steps:

- create a master and slave replication using master-slave-rc.json

```
oc process -f master-slave-rc.json | oc create -f -
```

- scale up the number of slaves to 2

```
oc scale rc pg-slave-rc-1 --replicas=2
```

- delete the master pod

```
oc delete pod pg-master-rc
```

- exec into a slave and create a trigger file to being the recovery process, effectively turning the slave into a master

```
oc exec -it pg-slave-rc-1-lt5a5
touch /tmp/pg-failover-trigger
```

- change the label on the slave to pg-master-rc instead of pg-slave-rc

```
oc edit pod/pg-slave-rc-1-lt5a5
original line: labels/name: pg-slave-rc
updated line: labels/name: pg-master-rc
```

```
or alternatively:
```

```
oc label --overwrite=true pod pg-slave-rc-1-lt5a5 name=pg-master-rc
```

You can test the failover by creating some data on the master and then test to see if the slaves have the replicated data.

```
psql -c 'create table foo (id int)' -U master -h pg-master-rc postgres
psql -c 'table foo' -U master -h pg-slave-rc postgres
```

After a failover, you would most likely want to create a database backup and be prepared to recreate your cluster from that backup.

Openshift Example 8 - Master Slave Deployment using NFS

This example uses NFS volumes for the master and the slaves. In some scenarios, customers might want to have all the Postgres instances using NFS volumes for persistence.

Relevant files for this example:

- master-slave-rc-nfs.json This file creates the master and slave deployment, creating pods and services where the slave is controlled by a Replication Controller, allowing you to scale up the slaves.

To run the example, follow these steps:

- as the openshift admin, create the required PV(s) using this command:

```
oc create -f master-slave-rc-nfs-pv.json
oc create -f master-slave-rc-nfs-pv2.json
```

This will create a PV for the master and another PV for the slaves. * as the project user, create the required PVC(s) using this command:

```
oc create -f master-slave-rc-nfs-pvc2.json
oc create -f master-slave-rc-nfs-pvc.json
```

This will create a PVC for the master and another PVC for the slaves. * as the project user, create the master slave deployment:

```
oc process -f master-slave-rc-nfs.json | oc create -f -
```

If you examing your NFS directory, you will see postgres data directories created and used by your master and slave pods.

Next, add some test data to the master:

```
psql -c 'create table testtable (id int)' -U master -h pg-master-rc-nfs postgres
psql -c 'insert into testtable values (123)' -U master -h pg-master-rc-nfs postgres
```

Next, add a new slave:

```
oc scale rc pg-slave-rc-nfs-1 --replicas=2
```

At this point, you should see the new NFS directory created by the new slave pod, and you should also be able to test that replication is working on the new slave:

```
psql -c 'table testtable' -U master -h pg-slave-rc-nfs postgres
```

Openshift Example 9 - Master with pgbadger add-in

This example uses a version of master.json but also adds the pgbadger container to the pg-master pod. pgbadger is then served up on port 10000. Each time you do a GET on http://pg-master:10000/api/badgergenerate it will run pgbadger against the database log files running in the pg-master container.

To run the example, you first need to build the crunchy-ose-pgbadger container imagee. Next, run the following:

```
oc create -f master-badger.json  | oc create -f -
```

try the following command to see the generated HTML output:

```
curl http://pg-master:10000/api/badgergenerate
```

You can view this output in a browser if you allow port forwarding from your container to your server host using a command like this:

```
socat tcp-listen:10001,reuseaddr,fork tcp:pg-master:10000
```

This command maps port 10000 of the service/container to port 10001 of the local server. You can now use your browser to see the badger report.

This is a short-cut way to expose a service to the external world, Openshift would normally configure a Router whereby you could 'expose' the service in an Openshift way. Here is the docs on installing the Openshift Router:

```
https://docs.openshift.com/enterprise/3.0/install_config/install/deploy_router.html
```

Openshift Example 10 - Master with readiness probe

This example uses a version of master.json but also adds a Kubernetes readiness probe specific for postgresql. This readiness probe uses the postgres pg_isready utility to attempt a connection to postgres from within the container using the postgres user and postgres database as the parameters.

Run the following:

```
oc create -f master-ready.json  | oc create -f -
```

Openshift Example 11 - Master with secrets

This example uses a version of master.json but also adds a Kubernetes Secret that is used to hold the username and password for the PG_USER and PG_PASSWORD values.

The secret uses a base64 encoded string to represent the values to be read by the container during initialization:

```
oc create -f pguser-secret.json
oc process -f master-secret.json | oc create -f -
```

The secrets are mounted in the /pguser volume within the container and read during initialization. The container scripts create a Postgres user with those values. The PG_USER and PG_PASSWORD environment variables are NOT set when using this mechanism. Other environment variables can be overridden using secrets as well, see the check-for-secrets.sh script that gets executed during database initialization.

Openshift Tips

Tip 1: Finding the Postgresql Passwords

The passwords used for the PostgreSQL user accounts are generated by the Openshift 'process' command. To inspect what value was supplied, you can inspect the master pod as follows:

```
oc get pod pg-master-rc-1-n5z8r -o json
```

Look for the values of the environment variables: - PG_USER - PG_PASSWORD - PG_DATABASE

Tip 2: Examining a backup job log

Database backups are implemented as a Kubernetes Job. A Job is meant to run one time only and not be restarted by Kubernetes. To view jobs in Openshift you enter:

```
oc get jobs
oc describe job backupjob
```

You can get detailed logs by referring to the pod identifier in the job 'describe' output as follows:

```
oc logs backupjob-pxh2o
```

Tip 3: Backup Lifecycle

Backups require the use of network storage like NFS in Openshift. There is a required order of using NFS volumes in the manner we do database backups.

So, first off, there is a one-to-one relationship between a PV (persistent volume) and a PVC (persistence volume claim). You can NOT have a one-to-many relationship between PV and PVC(s).

So, to do a database backup repeatably, you will need to following this general pattern: * as openshift admin user, create a unique PV (e.g. backup-pv-mydatabase) * as a project user, create a unique PVC (e.g. backup-pvc-mydatabase) * reference the unique PVC within the backup-job template * execute the backup job template * as a project user, delete the job * as a project user, delete the pvc * as openshift admin user, delete the unique PV

This procedure will need to be scripted and executed by the devops team when performing a

database backup.

Tip 4: Persistent Volume Matching

Restoring a database from an NFS backup requires the building of a PV which maps to the NFS backup archive path. For example, if you have a backup at /backups/pg-foo/2016-01-29:22:34:20 then we create a PV that maps to that NFS path. We also use a "label" on the PV so that the specific backup PV can be identified.

We use the pod name in the label value to make the PV unique. This way, the related PVC can find the right PV to map to and not some other PV. In the PVC, we specify the same "label" which lets Kubernetes match to the correct PV.

Tip 5: Restore Lifecycle

To perform a database restore, we do the following: * locate the NFS path to the database backup we want to restore with * edit a PV to use that NFS path * edit a PV to specify a unique label * create the PV * edit a PVC to use the previously created PV, specifying the same label used in the PV * edit a database template, specifying the PVC to be used for mounting to the /backup directory in the database pod * create the database pod

If the /pgdata directory is blank AND the /backup directory contains a valid postgres backup, it is assumed the user wants to perform a database restore.

The restore logic will copy /backup files to /pgdata before starting the database. It will take time for the copying of the files to occur since this might be a large amount of data and the volumes might be on slow networks. You can view the logs of the database pod to measure the copy progress.

Tip 6: Password Mgmt

Remember that if you do a database restore, you will get whatever user IDs and passwords that were saved in the backup. So, if you do a restore to a new database and use generated passwords, the new passwords will not be the same as the passwords stored in the backup!

You have various options to deal with managing your passwords.

- externalize your passwords using secrets instead of using generated values
- manually update your passwords to your known values after a restore

Note that you can edit the environment variables when there is a 'dc' using, currently only the slaves have a 'dc' to avoid the possiblity of creating multiple masters, this might need to change in the future, to better support password management:

```
oc env dc/pg-master-rc PG_MASTER_PASSWORD=foo PG_MASTER=user1
```

Tip 7: Log Aggregation

Openshift can be configured to include the EFK stack for log aggregation. Openshift Administrators can configure the EFK stack as documented here:

https://docs.openshift.com/enterprise/3.1/install_config/aggregate_logging.html

Tip 8: nss_wrapper

If an Openshift deployment requires that random generated UIDs be supported by containers, the Crunchy containers can be modifed similar to those located here to support the use of nss_wrapper to equate the random generated UIDs/GIDs by openshift with the postgres user:

https://github.com/openshift/postgresql/blob/master/9.4/root/usr/share/container-scripts/postgresql/common.sh

Tip 9: build box setup

golang is required to build the pgbadger container, on RH 7.2, golang is found in the 'server optional' repository and needs to be enabled to install.

golang is required to build the pgbadger container, on RH 7.2, golang is found in the 'server optional' repository and needs to be enabled to install.

Tip 10: encoding secrets

You can use kubernetes secrets to set and maintain your database credentials. Secrets requires you base64 encode your user and password values as follows:

```
echo -n 'myuserid' | base64
```

You will paste these values into your JSON secrets files for values.