



Openshift Installation, Examples, and Tips - Crunchy Containers

Crunchy Data Solutions, Inc.

Version 1.4.1, 2017-05-10

Openshift Installation

Install Openshift Enterprise

See the OSE installation guide for details on how to install OSE on your host. The main instructions are here:

https://docs.openshift.com/enterprise/3.2/install_config/install/index.html

Install Openshift Origin

The easiest way to get Openshift Origin up and running is found here: https://github.com/openshift/origin/blob/master/docs/cluster_up_down.md

Examples for the Openshift Environment

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

The examples are explained below.

Some of the examples make use of NFS for a persistent volume file system type, see the NFS Note at the bottom of this document for details on how NFS is configured in these examples.

Notes on NFS installation and file system permissions are in the install.asciidoc documentation.

For running the examples that require persistent volumes, you will need to run the following script:

```
cd $CCPROOT/examples/pv
./create-pv.sh
./create-pvc.sh
```

If you are wanting to run the examples on a minishift instance you will need to create the PVs using hostPath as follows:

```
oc login -u system:admin
./create-pv.sh hostpath
oc login -u developer
./create-pvc.sh
```

Additional steps are required to allow persistence to work on minishift including:

```
oc login -u system:admin
oc edit scc restricted
```

Above, you will change `runAsUser.Type` strategy to `RunAsAny`.

On the `boot2docker` instance running `minishift`, you will need to set the host path permissions as follows:

```
chmod 777 /mnt/sda1/data
```

The `NAMESPACE` environment variable is set to indicate which Openshift project you want various example objects to use. This variable is set to **default** within the `examples/envvars.sh` script. Set this to match your project configuration.

basic - Running a single database container

This openshift template will create a single master PostgreSQL instance.

Running the example:

```
cd $CCPROOT/examples/openshift/basic  
./run.sh
```

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

```
oc describe pod basic | 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 basic.openshift.svc.cluster.local -U testuser userdb
```

pgpool - pgpool for master replica examples

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

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

```
cd $CCPROOT/examples/openshift/pgpool  
./run.sh
```

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

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

when prompted, enter the password for the PG_USERNAME testuser that was set for the pg-master pod, typically it is **password**.

At this point, you can enter SELECT and INSERT statements and pgpool will proxy the SQL commands to the master or replica(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 replica(s).

You can view the nodes that pgpool is configured for by running:

```
psql -h pgpool-rc -U testuser userdb -c 'show pool_nodes'
```

master-replica-rc-dc - master replica with scalable replicas

This example is similar to the previous examples but builds a master pod, and a single replica 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 replicas.

Running the example:

```
oc create -f master-replica-rc-dc-replicas-only.json | oc create -f -
```

Connect to the PostgreSQL instances with the following:

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

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

```
oc scale rc pg-replica-rc-dc-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-replica-rc.pgproject.svc.cluster.local -U master postgres
```

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

the replicas:

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

backup - Performing a Full Backup

This example assumes you have a database pod running called **basic** as created by the **basic** example and that you have configured NFS as described in the Prerequisites section of this document..

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

```
cd $CCPROOT/examples/openshift/backup-job  
./run.sh
```

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 `examples/openshift/crunchy-pv-backup.json` specifies a `persistentVolumeReclaimPolicy` of **Retain** to tell Openshift that we want to keep the volume contents after the removal of the PV.

master-pvc - master using a PVC Example

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

```
cd $CCPROOT/examples/openshift/master-pvc  
./run.sh
```

master-restore - example of restoring a database from a backup

This is 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:

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

Then create the pod:

```
./run.sh
```

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

Failover Example

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

- create a master and replica replication using master-replica-rc-dc-replicas-only.json

```
oc process -f master-replica-rc-dc-replicas-only.json | oc create -f -
```

- scale up the number of replicas to 2

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

- delete the master pod

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

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

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

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

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

- or alternatively:

```
oc label --overwrite=true pod pg-replica-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 replicas 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-replica-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.

master-replica-rc-pvc - Master Slave Deployment using pvc

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

Relevant files for this example:

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

To run the example, follow these steps:

As the project user, create the master replica deployment:

```
./run.sh
```

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

Next, add some test data to the master:

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

Next, add a new replica:

```
oc scale rc pg-replica-rc-pvc-1 --replicas=2
```

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

```
psql -c 'table testtable' -U master -h pg-replica-rc-pvc postgres
```

badger - pgbadger example

This example creates a pod that contains a database container and a pgbadger container.

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.

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.

To run the example:

```
cd $CCPROOT/examples/openshift/badger
./run.sh
```

try the following command to see the generated HTML output:

```
curl http://badger-example: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
```

secret - database with 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.

This example allows you to set the Postgresql passwords using Kube Secrets.

The secret uses a base64 encoded string to represent the values to be read by the container during initialization. The encoded password value is **password**. Run the example as follows:

```
cd $CCPROOT/examples/openshift/secret/run.sh
./run.sh
```

The secrets are mounted in the **/pguser**, **/pgmaster**, **/pgroot** volumes within the container and read during initialization. The container scripts create a Postgres user with those values, and sets the passwords for the master user and postgres superuser using the mounted secret volumes.

When using secrets, you do NOT have to specify the following env vars if you specify all three secrets volumes:

- PG_USER
- PG_PASSWORD
- PG_ROOT_PASSWORD
- PG_MASTER_USER
- PG_MASTER_PASSWORD

You can test the container as follows, in all cases, the password is **password**:

```
psql -h secret-pg -U pguser1 postgres
psql -h secret-pg -U postgres postgres
psql -h secret-pg -U master postgres
```

watch - Automated Failover

This example shows how a form of automated failover can be configured for a master and replica deployment.

First, create a master and a replica, in this case the replica lives in a Deployment which can scale up:

```
cd $CCPROOT/examples/openshift/master-replica-dc
./run.sh
```

Next, create an Openshift service account which is used by the crunchy-watch container to perform the failover, also set policies that allow the service account the ability to edit resources within the openshift and default projects :

```
oc create -f sa.json
oc policy add-role-to-group edit system:serviceaccounts -n openshift
oc policy add-role-to-group edit system:serviceaccounts -n default
```

Next, create the container that will 'watch' the Postgresql cluster:

```
cd $CCPROOT/examples/openshift/watch  
./run.sh
```

At this point, the watcher will sleep every 20 seconds (configurable) to see if the master is responding. If the master doesn't respond, the watcher will perform the following logic:

- log into openshift using the service account
- set its current project
- find the first replica pod
- delete the master service saving off the master service definition
- create the trigger file on the first replica pod
- wait 20 seconds for the failover to complete on the replica pod
- edit the replica pod's label to match that of the master
- recreate the master service using the stored service definition
- loop through the other remaining replica and delete its pod

At this point, clients when access the master's service will actually be accessing the new master. Also, Openshift will recreate the number of replicas to its original configuration which each replica pointed to the new master. Replication from the master to the new replicas will be started as each new replica is started by Openshift.

To test it out, delete the master pod and view the watch pod log:

```
oc delete pod pg-master-rc-dc  
oc logs watch  
oc get pod
```

metrics - Metrics Collection

This example shows how postgres metrics can be collected and stored in prometheus and graphed with grafana.

First, create the crunchy-metrics pod which contains the prometheus data store and the grafana graphing web application:

```
./run.sh
```

At this point, you can view the prometheus web console at crunchy-metrics:9090, the prometheus push gateway at crunchy-metrics:9091, and the grafana web app at crunchy-metrics:3000.

Next, start a postgres pod that has the crunchy-collect container as follows:

```
cd $CCPROOT/examples/openshift/collect
./run.sh
```

At this point, metrics will be collected every 3 minutes and pushed to prometheus. You can build graphs off the metrics using grafana.

vacuum - Vacuum job

This example shows how you can run a vacuum job against a postgres database container.

The crunchy-vacuum container image exists to allow a DBA a way to run a job either one-off or scheduled to perform a variety of vacuum operations.

To run the vacuum a single time, an example is included as follows from the examples/openshift directory:

```
cd $CCPROOT/examples/openshift/master-replica
./run.sh
cd ../vacuum-job
./run.sh
```

This will start a vacuum container that runs as a Kube Job type. It will run once. The crunchy-vacuum image is executed, passed in the Postgres connection parameters to the single-master postgres container. The type of vacuum performed is dictated by the environment variables passed into the job. The complete set of environment variables read by the vacuum job include:

- `VAC_FULL` - when set to true adds the `FULL` parameter to the `VACUUM` command
- `VAC_TABLE` - when set, allows you to specify a single table to vacuum, when not specified, the entire database tables are vacuumed
- `JOB_HOST` - required variable is the postgres host we connect to
- `PG_USER` - required variable is the postgres user we connect with
- `PG_DATABASE` - required variable is the postgres database we connect to
- `PG_PASSWORD` - required variable is the postgres user password we connect with
- `PG_PORT` - allows you to override the default value of 5432
- `VAC_ANALYZE` - when set to true adds the `ANALYZE` parameter to the `VACUUM` command
- `VAC_VERBOSE` - when set to true adds the `VERBOSE` parameter to the `VACUUM` command
- `VAC_FREEZE` - when set to true adds the `FREEZE` parameter to the `VACUUM` command

custom-config - Custom Configuration Files

This example shows how you can use your own customized version of `setup.sql` when creating a postgres database container.

If you mount a /pgconf volume, crunchy-postgres will look at that directory for postgresql.conf, pg_hba.conf, and setup.sql. If it finds one of them it will use that file instead of the default files. Currently, if you specify a postgresql.conf file, you also need to specify a pg_hba.conf file.

The example shows how a custom setup.sql file can be used. Run it as follows:

```
cd $CCPROOT/examples/openshift/custom-config  
./run.sh
```

This will start a database container that will use an NFS mounted /pgconf directory that will contain the custom setup.sql file found in the example directory.

custom-config-sync - Custom Configuration Files with Sync Replica

This example shows how you can use your own customized version of postgresql.conf and pg_hba.conf to override the default configuration. It also specifies a sync replica in the postgresql.conf and starts up a sync replica.

If you mount a /pgconf volume, crunchy-postgres will look at that directory for postgresql.conf, pg_hba.conf, and setup.sql. If it finds one of them it will use that file instead of the default files. Currently, if you specify a postgresql.conf file, you also need to specify a pg_hba.conf file.

Run it as follows:

```
cd $CCPROOT/examples/openshift/custom-config-sync  
./run.sh
```

This will start a **csmaster** container that will use the custom config files when the database is running. It will also create a sync replica named **cssyncreplica**, this replica is connected to the master via streaming replication.

pgbouncer - pgbouncer

This example shows how you can use the crunchy-pgbouncer container when running under Openshift.

The example assumes you have run the master/replica example found here:

```
$CCPROOT/examples/openshift/master-replica-dc
```

Then you would start up the pgbouncer container using the following example:

```
cd $CCPROOT/examples/openshift/pgbouncer
./run.sh
```

The example assumes you have an NFS share path of /nfsfileshare/! NFS is required to mount the pgbouncer configuration files which are then mounted to /pgconf in the crunchy-pgbouncer container.

If you mount a /pgconf volume, crunchy-postgres will look at that directory for postgresql.conf, pg_hba.conf, and setup.sql. If it finds one of them it will use that file instead of the default files.

Test the example by killing off the master database container as follows:

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

Then watch the pgbouncer log as follows to confirm it detects the loss of the master:

```
oc logs pgbouncer
```

After the failover is completed, you should be able to access the new master using the master service as follows:

```
psql -h pg-master-rc-dc.openshift.svc.cluster.local -U master postgres
```

and access the replica as follows:

```
psql -h pg-replica-rc-dc.openshift.svc.cluster.local -U master postgres
```

or via the pgbouncer proxy as follows:

```
psql -h pgbouncer.openshift.svc.cluster.local -U master master
```

sync - synchronous replica

This example deploys a PostgreSQL cluster with a master, a synchronous replica, and an asynchronous replica. The two replicas share the same Service.

Running the example:

```
cd $CCPROOT/examples/openshift/sync
./run.sh
```

Connect to the **master** and **replica** databases as follows:

```
psql -h master -U postgres postgres -c 'create table mister (id int)'
psql -h master -U postgres postgres -c 'insert into mister values (1)'
psql -h master -U postgres postgres -c 'table pg_stat_replication'
psql -h replica -U postgres postgres -c 'select inet_server_addr(), * from mister'
psql -h replica -U postgres postgres -c 'select inet_server_addr(), * from mister'
psql -h replica -U postgres postgres -c 'select inet_server_addr(), * from mister'
```

This set of queries will show you the IP address of the Postgres replica container, notice it changes because of the round-robin Service proxy we are using for both replicas. The example queries also show that both replicas are replicating from the master.

pgadmin4 - pgadmin4

This example, `examples/openshift/pgadmin4`, provides a container that runs the pgadmin4 web application.

To run this example, run the following:

```
cd $CCPROOT/examples/openshift/pgadmin4
./run.sh
```

This script creates the **pgadmin4** pod and service, it will expose port 5050.

You should now be able to browse to <http://pgadmin4.openshift.svc.cluster.local:5050> and log into the web application using a user ID of **admin@admin.org** and password of **password**. Replace YOURLOCALIP with whatever your local IP address happens to be.

workshop - workshop

This example, `$CCPROOT/examples/openshift/workshop`, provides an example of using Openshift Templates to build pods, routes, services, etc.

You use the **oc new-app** command to create objects from the JSON templates. This is an alternative way to create Openshift objects instead of using **oc create**.

This example is used within a joint Redhat-Crunchy workshop that is given at various conferences to demonstrate Openshift and Crunchy Containers working together. Thanks to Steven Pousty from Redhat for this example!

See the README file within the workshop directory for instructions on running the example.

pitrr - PITR (point in time recovery)

This is a complex example. For details on how PITR is implemented within the Suite, see the [PITR Documentation](#) for details and background.

This example, `$CCPROOT/examples/openshift/pitrr`, provides an example of performing a PITR using

Openshift.

Lets start by running the example database container:

```
cd $CCPROOT/examples/openshift/pitr
./run-master-pitr.sh
```

This step will create a database container, **master-pitr**. This container is configured to continuously write WAL segment files to a mounted volume (/pgwal).

After you start the database, you will create a base backup using this command:

```
./run-master-pitr-backup.sh
```

This will create a backup and write the backup files to a persistent volume (/pgbackup).

Next, lets create some recovery targets within the database, run the SQL commands against the **master-pitr** database as follows:

```
./run-sql.sh
```

This will create recovery targets named **beforechanges**, **afterchanges**, and **nomorechanges**. It will create a table, **pitrtest**, between the **beforechanges** and **afterchanges** targets. It will also run a SQL CHECKPOINT to flush out the changes to WAL segments.

Next, now that we have a base backup and a set of WAL files containing our database changes, we can shut down the **master-pitr** database to simulate a database failure. Do this by running the following:

```
oc delete pod master-pitr
```

Next, we will create 3 different restored database containers based upon the base backup and the saved WAL files.

First, we restore prior to the **beforechanges** recovery target. This recovery point is **before** the **pitrtest** table is created.

Edit the master-pitr-restore.json file, and edit the environment variable to indicate we want to use the **beforechanges** recovery point:

```
{, {
  "name": "RECOVERY_TARGET_NAME",
  "value": "beforechanges"
}, {
```

Then run the following to create the restored database container:

```
./run-restore-pitr.sh
```

After the database has restored, you should be able to perform a test to see if the recovery worked as expected:

```
psql -h master-pitr-restore.openshift.svc.cluster.local -U postgres postgres -c 'table
pitrtest'
psql -h master-pitr-restore.openshift.svc.cluster.local -U postgres postgres -c
'create table foo (id int)'
psql -h master-pitr-restore.openshift.svc.cluster.local -U postgres postgres -c
'select pg_xlog_replay_resume()'
psql -h master-pitr-restore.openshift.svc.cluster.local -U postgres postgres -c
'create table foo (id int)'
```

The output of these command should show that the **pitrtest** table is not present. It should also show that you can not create a new table because the database is paused in recovery mode. Lastly, if you execute a **resume** command, it will show that you can now create a table as the database has fully recovered.

You can also test that if **afterchanges** is specified, that the **pitrtest** table is present but that the database is still in recovery mode.

Lastly, you can test a full recovery using **all** of the WAL files, if you remove the **RECOVERY_TARGET_NAME** environment variable completely.

The NFS portions of this example depend upon an NFS file system with the following path configurations be present:

```
/nfsfileshare
/nfsfileshare/backups
/nfsfileshare/WAL
```

backrest - pgbackrest example

This example shows how to enable pgbackrest as the archiver within the crunchy-postgres container. See the [pgbackrest Documentation](#) for details and background.

Start by running the example database container:

```
cd $CCPROOT/examples/openshift/backrest
./run.sh
```

This will create the following:

- PV/PVC for /pgconf and /backrestrepo volumes
- master database pod
- master service

The run.sh script copies the pgbackrest.conf configuration file to /nfsfileshare/pgconf which is our NFS file path.

The archive files are written to the NFS path of /nfsfileshare/backrestrepo.

The presence of /pgconf/pgbackrest.conf is what is used to determine whether pgbackrest will be used as the archive command or not. You will need to specify the ARCHIVE_TIMEOUT environment variable as well to use this.

After you run the example, you should see archive files being written to the /backrestrepo volume (/nfsfileshare/backrestrepo).

You can create a backup using backrest using this command within the container:

```
pgbackrest --stanza=db backup --db-path=/pgdata/master-backrest/ --log-path=/tmp
--repo-path=/backrestrepo -conf=/pgconf/pgbackrest.conf
```

backrest restore

This assumes you have run the pgbackrest example above. There are two options to choose from when performing a restore, DELTA and FULL. A FULL is the default; a DELTA will only occur if the environment variable DELTA is specified in the restore-job spec. Consult the pgbackrest user guide to determine which is best suited to run.

Steps for FULL restore

- Delete the master-backrest pod, if still running
- Empty the PGDATA directory (remove all files)
- Navigate to the backrest_restore examples directory. Execute the full-restore.sh script.
- Check the restore logs (db-restore.log) in the /backrestrepo mountpoint for success. You can also view the logs of the completed job pod with `kubectl get pod -a`
- Re-create the master-backrest pod in the backrest examples directory. The database will recover.

Steps for DELTA restore

- Delete the master-backrest pod, if still running
- `rm postmaster.pid` from PGDATA.
- Navigate to the backrest_restore examples directory. Execute the delta-restore.sh script.
- Check the restore logs (db-restore.log) in the /backrestrepo mountpoint for success. You can also view the logs of the completed job pod with `kubectl get pod -a`

- Re-create the master-backrest pod in the backrest examples directory. The database will recover only files that have changed from the last backup.

configmap- database credentials from a configmap

This example shows how to use a configmap to store the postgresql.conf and pg_hba.conf files to be used when overriding the default configuration within the container.

Start by running the database container:

```
cd $CCPROOT/examples/openshift/configmap
./run.sh
```

The files, pg_hba.conf and postgresql.conf, in the example directory are used to create a configmap object within OpenShift. Within the run.sh script, the configmap is created, and notice within the configmap.json file how the /pgconf mount is related to the configmap.

statefulset- a StatefulSet example

This example shows how to use a StatefulSet (available in OpenShift Origin 1.5) to create a PostgreSQL cluster.

Build the example by:

```
cd $CCPROOT/examples/openshift/statefulset
./run.sh
```

This will create a statefulset named pgset, which will create 2 pods, pgset-0 and pgset-1:

```
oc get statefulset
oc get pod
```

A service is created for the master and another service for the replica:

```
oc get service
```

The statefulset ordinal value of 0 is used to determine which pod will act as the PostgreSQL master, all other ordinal values will assume the replica role.

crunchy-proxy- a Postgres-aware proxy

This example shows how to use the **crunchy-proxy** to act as a smart proxy to a PostgreSQL cluster. The example depends upon the **master-replica** example being run prior.

crunchy-proxy offers a high performance alternative to pgbouncer and pgpool.

The proxy example copies a configuration file to the PV_PATH and starts up the **crunchy-proxy** within a Deployment.

The proxy reads the configuration file from a **/config** volume mount and begins execution.

Start by running the proxy container:

```
cd $CCPROOT/examples/openshift/crunchy-proxy
./run.sh
```

The proxy will listen on port 5432 as specified in the configuration file. The example creates a Service named **crunchy-proxy** that you can use to access the configured PostgreSQL backend containers from the **master-replica** example.

See the following link for more information on the **crunchy-proxy**:

<https://github.com/CrunchyData/crunchy-proxy>

Test the proxy by running psql commands via the proxy connection:

```
psql -h crunchy-proxy -U postgres postgres
```

SQL "reads" will be sent to the PostgreSQL replica database if your SQL includes the **crunchy-proxy** read annotation. SQL statements that do not include the read annotation will be sent to the master database container within the PostgreSQL cluster.

master-deployment- deploy a master and replica both in a Deployment

This example shows how to deploy a master pod in a Deployment and use the PGDATA_PATH_OVERRIDE env var to determine the /pgdata path. With the override, you can restart the master pod and it will be able to find the original postgres data path.

The example also starts a replica pod within a Deployment of its own so that you can scale up the replica pods.

Start by running the example:

```
cd $CCPROOT/examples/openshift/master-deployment
./run.sh
```

this will start a master-dc service, a replica-dc service, a master-dc deployment with replicas count of 1 and a replica-dc deployment with replicas count of 2.

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: Password Management

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 replicas have a 'dc' to avoid the possibility 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 4: 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 5: nss_wrapper

If an Openshift deployment requires that random generated UUIDs be supported by containers, the Crunchy containers can be modified similar to those located here to support the use of nss_wrapper to equate the random generated UUIDs/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 6: 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.

docker to be installed.

You can keep yum from upgrading docker by including this line in your /etc/yum.conf file:

```
exclude=docker-1.9* docker-selinux-1.9*
```

Tip 7: DNS configuration for Openshift development

As of OSE 3.3, the following DNS modifications are not typically necessary any longer....but I'm leaving them here as a reference....

Luke Meyer from Redhat wrote an excellent blog on how to configure dnsmasq and Openshift, it is located here:

<http://developers.redhat.com/blog/2015/11/19/dns-your-openshift-v3-cluster/>

Key things included in this blog are:

- configuring dhcp to include the local IP address in /etc/resolv.conf upon boot
- configuring dnsmasq
- configuring openshift dns to listen on another port

In my dev setup, I have openshifts DNS listening on 127.0.0.1:8053. I have my dnsmasq listening on the local IP address 192.168.0.109:53

Therefore in my /etc/dhcp/dhclient.conf I have this config:

```
prepend domain-name-servers 192.168.0.109;
```

If you don't have your DNS configured correctly, replication controllers and deployment configs basically will not work.

Tip 8: system policies for pv creation and listing

For my testing, I wanted to allow the **system** user to be able to create and list persistent volumes, as of OSE 3.3, I had to enter these commands as the **root** user after installation to modify the policies:

```
oadm policy add-role-to-user cluster-reader system
oc describe clusterPolicyBindings :default
oadm policy add-cluster-role-to-user cluster-reader system
oc describe clusterPolicyBindings :default
oc describe clusterPolicyBindings :default
oadm policy add-cluster-role-to-user cluster-admin system
```

Tip 9: anyuid permissions

For my testing, I create a user named **test** on OSE, then I run the following command to grant it permission to use the **anyuid** SCC:

```
oc adm policy add-scc-to-group anyuid system:authenticated
```

This says that any authenticate user can run with the anyuid SCC which lets them create PVCs and use the **fsGroup** setting for the Postgres containers to work using NFS. There is probably a smarter and more precise way to grant this permission?

Tip 10: NFS Setup

To control the permissions of the NFS file system certain examples make use of the **supplementalGroups** security context setting for pods. In these examples, we specify the GID of the **nfsnobody** group (65534). If you want to use a different GID for the supplementalGroup then you will need to alter the NFS examples accordingly.

When the pod runs, the pod user is UID **26** which is the postgres user ID. By specifying the **supplementalGroup** the pod will also be added to the **nfsnobody** group. So, when you set up your NFS mount, you can specify the permissions to be as follows:

```
drwxrwx---. 3 nfsnobody nfsnobody 23 Dec 16 11:28 nfsfileshare
```

This restricts **other** users from writing to the NFS share, but will allow the **nfsnobody** group to have write access. This way, the NFS mount permissions can be managed to only allow certain pods write access.

Also, remember that on systems with SELinux set to enforcing mode that you will need to allow NFS write permissions by running this command:

```
sudo setsebool -P virt_use_nfs 1
```

Note that supplementalGroup settings are required for NFS but you would use the fsGroup setting for the AWS file system. Check out this link for details: https://docs.openshift.org/latest/install_config/persistent_storage/pod_security_context.html

Tip 11: 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 12: 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 13: 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 14: DNS host entry and DeploymentConfig

If your openshift environment can not resolve your hostname via a DNS server (external to openshift!), you will get errors when trying to create a DeploymentConfig. So, you can either install dnsmasq and reconfigure openshift for that, or, you can run a DNS server on another host and add the openshift host entry to that DNS server. I use the skybridge2 Docker container for this purpose. You have to remember to adjust your /etc/resolv.conf to specify this new DNS server.

Tip 15: Setting up Docker storage

I typically set up Docker storage this way:

- add an extra IDE drive to my VM
- fdisk /dev/sd? to format the drive
- vgcreate /dev/sd?1 to create a volume group on the new drive partition
- add VG=docker-vg to /etc/sysconfig/docker-storage-setup
- run docker-storage-setup to use that new volume group

Follow the instructions available [on the main Openshift documentation page](#) to better understand how to customize Docker storage according to your needs.

Openshift Dedicated Environment

Here are instructions for running examples on an Openshift Dedicated environment. For Dedicated, we have built a set of templates that can be installed into a Dedicated instance to help automate the creation of Crunchy Container Suite containers.

Installation

If you've already set up your environment as described in [install documentation](#), use the following commands to install the templates into the Openshift environment:

```
oc login -u system:admin
cd $CCPROOT/examples/templates
./create-all.sh
```

Example Details

Each example will build a template to be later used by users when they want to deploy a Crunchy container.

The templates are installed by running the following script within each example directory:

```
./run.sh
```

When you run the examples, there are variable substitutions taking place to set the image path and image tags within the Openshift templates. This substitution allows for better support of different deployments and deployment environments.

database backup

Template Name is **crunchy-postgres-database-backup**

Example is found here:

```
cd $CCPROOT/examples/templates/backup
./run.sh
```

This template will create the following:

- Job which generates a backup container

This example deploys a Job which results in a Pod created which will run the crunchy-backup container. It will create a backup of a database and store the backup files in a PVC.

PGBadger

Template Name is **crunchy-postgres-database-with-pgbadger**

Example is found here:

```
cd $CCPROOT/examples/templates/badger
./run.sh
```

This template will create the following:

- database container (emptyDir)
- pgbadger container
- database service

Metrics Collection

Template Name is **crunchy-postgres-database-with-collect**

Example is found here:

```
cd $CCPROOT/examples/templates/collect
./run.sh
```

This template will create the following:

- database container (emptyDir)
- collect container
- database service

This example will pull postgres metrics from the database and push them to the metrics backend.

Metrics Collection and Badger Analysis

Template Name is **crunchy-postgres-database-with-collect-badger**

Example is found here:

```
cd $CCPROOT/examples/templates/master-collect-badger
./run.sh
```

This template will create the following:

- master database container (emptyDir)
- collect container
- badger container
- master database service

This example will pull postgres metrics from the database and push them to the metrics backend as well as provide a pgbadger container for log analysis.

Master - Replica Database Cluster

Template Name is **crunchy-postgres-master-replica-emptydir**

Example is found here:

```
cd $CCPROOT/examples/templates/master-replica
./run.sh
```

This template will create the following:

- database container as the master (emptyDir)
- database container as the replica (emptyDir)
- database service for the master
- database service for the replica

This example deploys a master-replica configuration which uses emptyDir for persistence.

Master - Replica Database Cluster using DeploymentConfig for Replica

Template Name is **crunchy-postgres-master-replica-deployment-emptydir**

Example is found here:

```
cd $CCPROOT/examples/templates/master-slave-dc
./run.sh
```

This template will create the following:

- database container as the master (emptyDir)
- DeploymentConfig for the replica (emptyDir) allowing scaling
- database service for the master

- database service for the replica

This example deploys a master-slave configuration which uses emptyDir for persistence. The DeploymentConfig allows you to scale up the number of replica containers.

Restore Database

Template Name is **crunchy-postgres-database-restore-emptydir**

Example is found here:

```
cd $CCPROOT/examples/templates/master-restore
./run.sh
```

This template will create the following:

- database container (emptyDir)
- database service

This example performs a database restore using a backup archive found in a PVC.

Metrics

Template Name is **crunchy-metrics**

Example is found here:

```
cd $CCPROOT/examples/templates/metrics
./run.sh
```

This template will create the following:

- prometheus container (PVC for persistence)
- grafana container (PVC for persistence)
- promgateway container
- services for prometheus, grafana, and promgateway

This example deploys all the required metrics containers which are used to store and visualize collected postgres metrics. The example uses a single PVC to store all the metrics and dashboard data.

pgadmin4

Template Name is **crunchy-pgadmin4-using-pvc**

Example is found here:

```
cd $CCPROOT/examples/templates/pgadmin4
./run.sh
```

This template will create the following:

- pgadmin4 container (PVC for persistence)
- service for pgadmin4

This example deploys the pgadmin4 container and uses a PVC to persist the data.

pgbouncer

Template Name is **crunchy-pgbouncer**

Example is found here:

```
cd $CCPROOT/examples/templates/pgbouncer
./run.sh
```

This template will create the following:

- pgbouncer container
- PVC to hold configuration
- service for pgbouncer

pgpool

Template Name is **crunchy-postgres-pgpool-template**

Example is found here:

```
cd $CCPROOT/examples/templates/pgpool
./run.sh
```

This template will create the following:

- pgpool container within a DeploymentConfig
- service for pgpool

This example deploys the pgpool container and service. The DeploymentConfig allows you to scale up the pgpool container.

replica with DeploymentConfig

Template names is **crunchy-postgres-replica-deployment-emptydir**

Example is found here:

```
cd $CCPROOT/examples/templates/replica-dc
./run.sh
```

These templates create the following:

- replica database container using emptyDir volume and a DeploymentConfig allowing scaling
- service for replica

This example deploys a single replica container which uses emptyDir that can be scaled up within a DeploymentConfig.

secrets

Template names are **master-with-secrets-example**.

Example is found here:

```
cd $CCPROOT/examples/templates/secret
./run.sh
```

This run.sh script creates the following:

- master database container using emptyDir volume
- service for database
- secrets to hold database credentials

single master database

Template names are **crunchy-postgres-database-emptydir** and **crunchy-postgres-database-pvc**

Example is found here:

```
cd $CCPROOT/examples/templates/single-master
./run.sh
```

These templates create the following:

- master database container using emptyDir volume or a PVC
- service for database

This example deploys a single database container which uses emptyDir or a PVC for volume persistence.

replica database

Template names are **crunchy-postgres-replica-emptydir** and **crunchy-postgres-replica-pvc**.

Example is found here:

```
cd $CCPROOT/examples/templates/single-replica
./run.sh
```

This template will create the following:

- replica database container using emptyDir or PVC as the data volume
- service to the replica

This example deploys a replica database pod using either emptyDir or a PVC to store data. A service is also created to the replica.

synchronous replica database

Template Name is **crunchy-postgres-master-sync-replica-pvc**

Example is found here:

```
cd $CCPROOT/examples/templates/sync
./run.sh
```

This template will create the following:

- PVC for the master database
- PVC for the replica database
- master database container using PVC as the data volume
- replica database container using PVC as the data volume
- service to the master database
- service to the replica database

This example deploys a database pod using a PVC to store data. A service is also created to the database. This example creates a synchronous replica.

watch

Template Name is **crunchy-cluster-watch-template**

Example is found here:

```
cd $CCPROOT/examples/templates/watch  
./run.sh
```

This template will create the following:

- watch pod for watching a database cluster

This example deploys a watch pod that watches a database master, if the master dies, it will trigger a failover to a replica. This pod uses a ServiceAccount which is required to be defined.

Legal Notices

Copyright © 2017 Crunchy Data Solutions, Inc.

CRUNCHY DATA SOLUTIONS, INC. PROVIDES THIS GUIDE "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Crunchy, Crunchy Data Solutions, Inc. and the Crunchy Hippo Logo are trademarks of Crunchy Data Solutions, Inc.