Examples & Use Cases - Crunchy Containers for PostgreSQL

Crunchy Data Solutions, Inc.

Version 1.4.1, 2017-06-22

Table of Contents

Container Examples

basic - Running a single database container

Docker

Create the container with this command:

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

This script will do the following:

- create a data volume to persist PostgreSQL data files to
- start up a container named **basic**
- initialize the database using the passed in environment variables, building a user, database, schema, and table based on the environment variable values.
- maps the PostgreSQL port of 5432 in the container to your local host port of 12000.

To start the instance, run the following commands:

```
docker start basic
```

The container creates a default database called **userdb**, a default user called **testuser** with a default password of **password**, you can use this to connect from your local host as follows:

```
psql -h localhost -p 12000 -U testuser -W userdb
```

To shut down the instance, run the following commands:

```
docker stop basic
```

Kubernetes

This example starts a single postgres container and service, the most simple of examples.

Running the example:

```
examples/kube/basic/run.sh
kubectl get pod basic
kubectl get service basic
kubectl logs basic
```

After the database starts up you can connect to it as follows:

```
psql -h basic -U postgres postgres
```

OpenShift

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
```

master-replica - Creating a master and replica database cluster

Docker

Run the container with this command:

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

This script will do the following:

- create a docker volume using the local driver for the master
- create a docker volume using the local driver for the replica
- start up a container named master binding to port 12000
- start up a container named replica binding to port 12002
- initialize the database using the passed in environment variables, building a user, database, schema, and table based on the environment variable values.
- maps the PostgreSQL port of 5432 in the container to your local host port of 12000.

To start the instance, run the following commands:

```
docker start master replica
```

The container creates a default database called **userdb**, a default user called **testuser** with a default password of **password**, you can use this to connect from your local host as follows:

```
psql -h localhost -p 12007 -U testuser -W userdb
psql -h localhost -p 12008 -U testuser -W userdb
```

To shut down the instance, run the following commands:

```
docker stop master replica
```

Kubernetes

This example starts a master pod, master service, replica pod, and replica service. The replica is a replica of the master. This example uses emptyDir volumes for persistence. This example does not allow you to scale up the replicas.

Running the example:

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

It takes about a minute for the replica to begin replicating with the master. To test out replication, see if replication is underway with this command:

```
psql -h master -U postgres postgres -c 'table pg_stat_replication'
```

If you see a line returned from that query it means the master is replicating to the replica. Try creating some data on the master:

```
psql -h master -U postgres postgres -c 'create table foo (id int)'
psql -h master -U postgres postgres -c 'insert into foo values (1)'
```

Then verify that the data is replicated to the replica:

```
psql -h replica -U postgres postgres -c 'table foo'
```

OpenShift

Run the following command to deploy a master and replica database cluster:

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

Similarly to the previous example on **basic**, you can view the generated passwords by running this command:

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

You can then connect to the database instance as follows using the password shown with the previous command:

```
psql -h ms-master -U testuser -W userdb
```

master-replica-dc - Master and scaling replica example

Kubernetes

This example starts a master pod, master service, replica pod, and replica service. The replica is a replica of the master. This example uses emptyDir volumes for persistence. This example runs the replicas in a Deployment. A deployment controller lets you scale up the replicas and create an initial replica set.

Running the example:

```
examples/kube/master-replica-dc/run.sh
```

You can insert data in the master and make sure it replicates to the replicas using the commands from Example 2 above. Replace **master** with the **master-dc** name and **replica** with **replica-dc**.

This example creates 2 replicas when it initially starts. To scale up the number of replicas, run this command:

```
kubectl get deployment
kubectl scale --current-replicas=2 --replicas=3 deployment/replica-dc
kubectl get deployment
kubectl get pod
```

You can verify that you now have 3 replicas by running this query on the master:

```
psql -h master-dc -U postgres postgres -c 'table pg_stat_replication'
```

OpenShift

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:

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

Connect to the PostgreSQL instances with the following:

```
psql -h master-dc.pgproject.svc.cluster.local -U testuser userdb
psql -h replica-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 replica-dc-1 --replicas=2
```

To check the **master** default password, enter the following command and look for the **PG_MASTER_USER** and **PG_MASTER_PASSWORD** variables:

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

Enter the following commands to verify the PostgreSQL replication is working, using the password for master found with the previous command.

```
psql -c 'table pg_stat_replication' -h master-dc.pgproject.svc.cluster.local -U master
postgres
psql -h replica-dc.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 replica-dc -U master postgres
```

master-replica-rc - Master and scaling replica example

Kubernetes

This example starts a master pod, master service, replica pod, and replica service. The replica is a replica of the master. This example uses emptyDir volumes for persistence. This example runs the replicas in a Replication Controller. A replication controller lets you scale up the replicas and create an initial replica set. Deployments will likely be the preferred way to create a replica set going forward but I wanted to provide an example for thoroughness.

Running the example:

```
examples/kube/master-replica-rc/run.sh
```

You can also scale up the number of replicas using this replication controller mechanism. The command to scale up is as follows:

```
kubectl get rc
kubectl scale rc replica-rc --replicas=3
kubectl get pod
```

master-replica-rc-pvc - Master/Slave deployment using pvc

OpenShift

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.

To run the example, follow these steps:

As the project user, create the master replica deployment:

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

Note: The **master-replica.json** 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.

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 m-s-rc-pvc-master postgres
psql -c 'insert into testtable values (123)' -U master -h m-s-rc-pvc-master postgres
```

Next, add a new replica:

```
oc scale rc m-s-rc-pvc-replica-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 m-s-rc-pvc-replica postgres
```

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

Kubernetes

Starting in release 1.2.8, the postgres container can accept an environment variable named PGDATA_PATH_OVERRIDE. If set, the /pgdata/subdir path will use a path subdir name of your choosing instead of the default which is the hostname of the container.

This example shows how a Deployment of a master postgres is supported. A pod is a deployment uses a hostname generated by Kubernetes, so if you want to restart the master pod, you will get a different hostname as defined by the Deployment. For finding the /pgdata that pertains to the pod, you will need to specify a /pgdata/subdir name that never changes, and that is the purpose of the PGDATA_PATH_OVERRIDE env var.

Start the example as follows:

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

This will create the following in your Kube environment:

- create a master-dc service, uses a PVC to persist postgres data
- create a replica-dc service, uses emptyDir persistence
- create a master-dc Deployment of replica count 1 for the master postgres database pod
- create a replica-dc Deployment of replica count 2 for the replica(s)

The persisted master postgres data is found under /pgdata/master-dc. If you delete the master pod, the Deployment will create another pod for the master, and will be able to start up immediately since we are using the same /pgdata/master-dc data directory.

OpenShift

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.

master using gluster fs

Kubernetes

This example deploys a master database container that uses a gluster file system as the persistent volume.

Set up gluster according to https://wiki.centos.org/SpecialInterestGroup/Storage/gluster-Quickstart

Start the example as follows:

```
cd $CCPROOT/examples/kube/gluster
./run.sh
```

This will start a container and service for the master database.

You can access the master database as follows:

```
psql -h master-gluster -U postgres postgres
```

This example has a mount point of /mnt/gluster which is mapped to the gluster fs at yourhost:/gv0

Tip

Create a static route from your host to 10.0.0.0/16 if you want to test the user interfaces of the metrics tools.

On my host, 114, and my bridge, br1, this worked for me:

```
ip route add 10.0.0.0/16 via 192.168.0.114 dev br1
```

master-pvc - Master using PVC

OpenShift

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
```

backup - Performing a backup

Docker

In order to run this backup script, you first need to edit run.sh to specify your host IP address you are running on. The script assumes you are going to backup the **basic** container created in the first example, so you need to ensure that container is running.

Run the backup with this command:

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

This script will do the following:

- start up a backup container named basicbackup
- run pg_basebackup on the container named master
- store the backup in /tmp/backups/master directory
- exit after the backup

Kubernetes

This example depends on the basic example being run prior to this example!

This example performs a database backup on the basic database. The backup is stored in the /nfsfileshare backup path which is also a dependency. See the installation docs on how to set up the NFS server on this host.

Running the example:

```
examples/kube/backup-job/run.sh
```

Things to point out with this example include its use of persistent volumes and volume claims to store the backup data files to an NFS server.

You can view the persistent volume information as follows:

```
kubectl get pvc
kubectl get pv
```

The Kube Job type executes a pod and then the pod exits. You can view the Job status using this command:

```
kubectl get job
```

While the backup pod is running, you can view the pod as follows:

```
kubectl get pod
```

You should find the backup archive in this location:

```
ls /nfsfileshare/basic
```

Tip

You can view the backup pod log using the **docker logs** command on the exited container. Use **docker ps -a | grep backup** to locate the container.

OpenShift

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 Step 5 of the install.adoc.

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-restore - Restoration of backup example

Docker

In order to run this backup script, you first need to edit run.sh to specify your host IP address you are running on. The script assumes you are going to backup the container created in Example 2.

Run the backup with this command:

```
cd $CCPROOT/examples/docker/restore
./run.sh
```

This script will do the following:

- start up a container named master-restore
- copy the backup files from Example 3 into /pgdata
- start up the container using the backup files
- maps the PostgreSQL port of 5432 in the container to your local host port of 12001 as to not conflict with the master running in the previous example.

Kubernetes

This example assumes you have run the backup-job example prior to this example!

You will need to find a backup you want to use for running this example, you will need the timestamped directory path under /nfsfileshare/basic/. Edit the master-restore.json file and update the BACKUP_PATH setting to specify the NFS backup path you want to restore with, example:

```
"name": "BACKUP_PATH",
"value": "basic/2016-05-27-14-35-33"
```

This example runs a postgres container passing in the backup location. The startup of the container will use rsync to copy the backup data to this new container, and then launch postgres which will use the backup data to startup with.

Running the example:

```
examples/kube/master-restore/run.sh
```

Test the restored database as follows:

```
psql -h restored-master -U postgres postgres
```

OpenShift

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:

```
cd $CCPROOT/examples/openshift/master-restore
./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 - Manual database failover

OpenShift

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

- create a master and replica replication

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

- scale up the number of replicas to 2

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

- delete the master pod

```
oc delete pod master-dc
```

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

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

- change the label on the replica to master-dc instead of replica-dc

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

- or alternatively:

```
oc label --overwrite=true pod replica-dc-1-lt5a5 name=master-dc
```

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 master-dc postgres
psql -c 'table foo' -U master -h replica-dc postgres
```

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

pgbackrest - Backup Utility

Kubernetes

Starting in release 1.2.5, the pgbackrest utility has been added to the crunchy-postgres container. See the pgbackrest Documentation for details on how this feature works within the container suite.

Start the example as follows:

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

This will create the following in your Kube environment:

- A configMap named backrestconf which contains the pgbackrest.conf file
- master-backrest pod with pgbackrest archive enabled. An initial stanza db will be created on initialization
- master-backrest service

The crunchy-pvc will be used for /pgdata, and crunchy-pvc2 for the /backrestrepo. Examine the /backrestrepo location to view the archive directory and ensure WAL archiving is working. See backrest.adoc for steps to backup and restore using pgbackrest.

OpenShift

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
```

pgbackrest restore - Restoring backups made with pgbackrest

Kubernetes & OpenShift

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.

pgadmin4 - PostgreSQL web user interface

Docker

This example, $CCPROOT/examples/docker/pgadmin4, provides a container that runs the pgadmin4 web application.

To run this example, run the following:

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

You should now be able to browse to http://YOURLOCALIP: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.

Kubernetes

This example deploys the pgadmin4 (beta4) web user interface for Postgresql.

Start the container as follows:

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

This will start a container and service for pgadmin4. You can browse the user interface at http://pgadmin4.default.svc.cluster.local:5050

See the pgadmin4 documentation for more details at http://pgadmin.org

The example uses pgadmin4 configuration files which are mounted at an NFS mount point, this NFS data directory is mounted into the container and used by the pgadmin4 application to persist metadata.

OpenShift

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.

crunchy-proxy - A smart proxy for PostgreSQL

Docker

A **crunchy-proxy** example is provided that will run a container that is configured to be used with the master and replica example provided in the **master-replica** example.

You can create the proxy by running:

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

This proxy will listen on localhost:12432. You can access the **master-replica** cluster by:

```
psql -h localhost -p 12432 -U postgres postgres
```

See this link for details on the **crunchy-proxy**: https://github.com/CrunchyData/crunchy-proxy

You might consider **crunchy-proxy** over pgpool and pgbouncer if you need load-balancing and smart SQL routing.

Kubernetes

This example assumes you have run the master-replica example prior to this example!

This example runs a crunchy-proxy pod that creates a special purpose proxy to a postgres cluster (master and replica).

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.

If you run the example in minikube, you will need to manually copy the crunchy-proxy-config.json file to a file on the minikube named **/data/config.json**.

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

Start by running the proxy container:

```
cd $CCPROOT/examples/kube/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.

OpenShift

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.

custom-config - Custom Configuration Files

OpenShift

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 container the custom setup.sql file found in the example directory.

custom-config sync - Custom Configuration Files with Sync Replica

OpenShift

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.

configmap - database credentials from a configmap

OpenShift

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.

pgpool - pgpool pod example

Docker

A pgpool example is provided that will run a pgpool container that is configured to be used with the master and replica example provided in the **master-replica** example. After running those commands to create a master and replica, you can create a pgpool container by running the following example command:

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

Enter the following command to connect to the pgpool that is mapped to your local port 12003:

```
psql -h localhost -U testuser -p 12003 userdb
```

You will enter the password of **password** when prompted. At this point you can execute both INSERT and SELECT statements on the pgpool connection. Pgpool will direct INSERT statements to the master and SELECT statements will be sent round-robin to both master and replica.

Kubernetes

This example assumes you have run the master-replica example prior to this example!

This example runs a pgpool pod that creates a special purpose proxy to a postgres cluster (master and replica).

Running the example:

```
examples/kube/pgpool/run.sh
```

The example is configured to allow the **testuser** to connect to the **userdb** database as follows:

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

OpenShift

You can create a pgpool service that will work with the master and replica created in the previous example.

You will need to edit the pgpool-rc.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 -U testuser userdb
psql -h pgpool -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 -U testuser userdb -c 'show pool_nodes'
```

pgbadger - pgbadger pod

Docker

A pgbadger example is provided that will run a HTTP server that when invoked, will generate a pgbadger report on a given database.

pgbadger reads the log files from a database to product an HTML report that shows various Postgres statistics and graphs.

To run the example, modify the run-badger.sh script to refer to the Docker container that you want to run pgbadger against, also referring to the container's data directory, then run the example as follows:

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

After execution, the container will run and provide a simple HTTP command you can browse to view the report. As you run queries against the database, you can invoke this URL to generate updated reports:

```
curl http://127.0.0.1:14000/api/badgergenerate
```

Kubernetes

This example runs a pod that includes a database container and a pgbadger container. A service is also created for the pod.

Running the example:

```
examples/kube/badger/run.sh
```

You can access pgbadger at:

```
curl http://badger:10000/api/badgergenerate
```

Tip

You can view the database container logs using this command:

```
kubectl logs -c server badger
```

OpenShift

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 are the docs on installing OpenShift on a router:

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

metrics & collect - Metrics collection

Docker

You can collect various Postgres metrics from your database container by running a crunchy-collect container that points to your database container.

Metrics collection requires you run the crunchy 'scope' set of containers that includes:

- prometheus
- prometheus push gateway
- grafana

To start this set of containers, run the following:

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

These metrics are described in this document.

An example has been provided that runs a database container and also the associated metrics collection container, run the example as follows:

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

Every 3 minutes the collection container will collect postgres metrics and push them to the crunchy prometheus database. You can graph them using the crunchy grafana container.

Kubernetes

This example starts up prometheus, grafana, and prometheus gateway.

It is required to view or capture metrics collected by crunchy-collect.

Running the example:

```
examples/kube/metrics/run.sh
```

This will start up 3 containers and services:

- prometheus (http://crunchy-prometheus:9090)
- prometheus gateway (http://crunchy-promgateway:9091)
- grafana (http://crunchy-grafana:3000)

If you want your metrics and dashboards to persist to NFS, run this script:

```
examples/kube/metrics/run-pvc.sh
```

In the docs folder of the github repo, check out the metrics.adoc for details on the exact metrics being collected.

This example runs a pod that includes a database container and a metrics collection container. A service is also created for the pod.

Running the example:

```
examples/kube/collect/run.sh
```

You can view the collect container logs using this command:

```
kubectl logs -c collect master-collect
```

You can access the database or drive load against it using this command:

```
psql -h master-collect -U postgres postgres
```

OpenShift

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:

```
cd $CCPROOT/examples/openshift/metrics
./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-job - PostgreSQL vacuum command

Docker

You can perform a postgres vacuum command by running the crunchy-vacuum container. You specify a database to vacuum using environment variables.

An example is shown in the $CCPROOT/examples/docker/vacuum/run.sh script and can be run as follows:

```
cd $CCPROOT/examples/docker/vacuum
./run.sh
```

This example performs a vacuum on a single table in the master postgres database. Vacuum is controlled via the following environment variables:

- 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

Kubernetes

This example assumes you have run the basic example prior to this example!

This example runs a Job which performs a SQL VACUUM on a particular table (testtable) in the basic database instance.

Running the example:

```
examples/kube/vacuum-job/run.sh
```

Verify the job completed:

```
kubectl get job
```

Look at the docker log of the vacuum job's pod:

```
docker logs $(docker ps -a | grep crunchy-vacuum | cut -f 1 -d' ')
```

OpenShift

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

crunchy-dba - cron scheduler for simple DBA tasks

Kubernetes

The crunchy-dba container implements a cron scheduler. The purpose of the crunchy-dba container is to offer a way to perform simple DBA tasks that occur on some form of schedule such as backup jobs or running a vacuum on a single Postgres database container. Both of these examples are provided as scripts.

You can either run the crunchy-dba container as a single pod or include the container within a database pod.

The crunchy-dba container makes use of a Service Account to perform the startup of scheduled jobs. The Kube Job type is used to execute the scheduled jobs with a Restart policy of Never.

The script to schedule vacuum on a regular schedule is executed through the following commands:

```
cd $CCPROOT/examples/kube/dba
./run-vac.sh
```

To run the script for scheduled backups, run the following in the same directory:

```
./run-backup.sh
```

Individual parameters for both can be modified within their respective JSON files; please see link:containers.adoc for a full list of what can be modified.

custom-setup - Custom setup.sql

Docker

You can use your own version of the setup.sql SQL file to customize the initialization of database data and objects when the container and database are created.

An example is shown in the $CCPROOT/examples/docker/custom-setup/run.sh script and can be run as follows:

```
cd $CCPROOT/examples/docker/custom-setup
./run.sh
```

This works by placing a file named, setup.sql, within the /pgconf mounted volume directory. Portions of the setup.sql file are required for the crunchy container to work, see comments within the sample setup.sql file.

pgbouncer

Docker

The pgbouncer utility can be used to provide a connection pool to postgres databases. The crunchy-pgbouncer container also contains logic that lets it perform a failover from a master to a slave database.

To test this failover, you first create a running master/slave cluster as follows:

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

An example is shown in the $CCPROOT/examples/docker/pgbouncer/run.sh script and can be run as follows:

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

This example configures pgbouncer to provide connection pooling for the master and pg-replica databases. It also sets the FAILOVER environment variable which will cause a failover to be triggered if the master database can not be reached.

To trigger the failover, stop the master database:

```
docker stop master
```

At this point, the pgbouncer will notice that the master is not reachable and touch the trigger file on the configured slave database to start the failover. The pgbouncer container will then reconfigure pgbouncer to relabel the slave database into the master database so clients to pgbouncer will be able to connect to the master as before the failover.

To just log into the database from the pgbouncer connection pool you would enter the following using the password "password":

```
psql -h localhost -p 12005 -U testuser master
```

Kubernetes

This example assumes you have run the master-replica example prior to this example!

This example runs a crunchy-pgbouncer container to look for the master within a postgres cluster, if it can not find the master it will proceed to cause a failover to a replica. It will also configure a pgbouncer container that sets up a connection pool to the configured master and replica.

Running the example:

```
examples/kube/pgbouncer/run.sh
```

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

```
psql -h pgbouncer -U postgres master
psql -h pgbouncer -U postgres replica
```

The names **master** and **replica** are pgbouncer configured names and don't necessarily have to match the database name in the actual Postgres instance.

View the pgbouncer log as follows:

```
kubectl log pgbouncer
```

Next, test the failover capability within the crunchy-watch container using the following:

```
kubectl delete pod master
```

Take another look at the pgbouncer log and you will see it trigger the failover to the replica pod. After this failover you should be able to execute the command:

```
psql -h pgbouncer -U postgres master
```

OpenShift

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
./run.sh
```

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 master-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 master-dc.openshift.svc.cluster.local -U master postgres
```

and access the replica as follows:

```
psql -h replica-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 replication

Docker

This example, $CCPROOT/examples/docker/sync, provides a streaming replication configuration that includes both synchronous and asynchronous slaves.

To run this example, run the following:

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

You can test the replication status on the master by using the following command and the password "password":

```
psql -h 127.0.0.1 -p 12000 -U postgres postgres -c 'table pg_stat_replication'
```

You should see 2 rows, 1 for the async slave and 1 for the sync slave. The sync_state column shows values of async or sync.

You can test replication to the slaves by entering some data on the master like this, and then querying the slaves for that data:

```
psql -h 127.0.0.1 -p 12000 -U postgres postgres -c 'create table foo (id int)'
psql -h 127.0.0.1 -p 12000 -U postgres postgres -c 'insert into foo values (1)'
psql -h 127.0.0.1 -p 12002 -U postgres postgres -c 'table foo'
psql -h 127.0.0.1 -p 12003 -U postgres postgres -c 'table foo'
```

Kubernetes

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:

```
examples/kube/sync/run.sh
```

Connect to the **mastersync** and **replicasync** databases as follows:

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

OpenShift

This example deploys a PostgreSQL cluster with a master, a synchrounous 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.

statefulsets

Kubernetes

This example deploys a statefulset named **pgset**. The statefulset is a new feature in Kubernetes as of version 1.5. Statefulsets have replaced PetSets going forward.

This example creates 2 Postgres containers to form the set. At startup, each container will examine its hostname to determine if it is the first container within the set of containers.

The first container is determined by the hostname suffix assigned by Kube to the pod. This is an ordinal value starting with **0**.

If a container sees that it has an ordinal value of **0**, it will update the container labels to add a new label of:

```
name=$PG_MASTER_HOST
```

In this example, PG_MASTER_HOST is specified as **pgset-master**.

By default, the containers specify a value of **name=pgset-replica**

There are 2 services that end user applications will use to access the PostgreSQL cluster, one service (pgset-master) routes to the master container and the other (pgset-replica) to the replica containers.

```
$ kubectl get service
NAME            CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
kubernetes      10.96.0.1       <none>        443/TCP    22h
pgset           None            <none>        5432/TCP   1h
pgset-master    10.97.168.138   <none>        5432/TCP   1h
pgset-replica   10.97.218.221   <none>        5432/TCP   1h
```

Start the example as follows:

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

You can access the master database as follows:

```
psql -h pgset-master -U postgres postgres
```

You can access the replica databases as follows:

```
psql -h pgset-replica -U postgres postgres
```

You can scale the number of containers using this command, this will essentially create an additional replica databse:

```
kubectl scale pgset --replica=3
```

OpenShift

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.

statefulset using dynamic provisioning

Kubernetes

The example in **examples/statefulset-dyn** is almost an exact copy of the previous statefulset example, however, this example uses Dynamic Storage Provisioning to automatically create Persistent Volume Claims based on StorageClasses. This Kube feature is available on Google Container Engine which this example was tested upon.

You can run the example as follows:

```
cd $CCPROOT/examples/kube/statefulset-dyn
./run.sh
```

This will create a StorageClass named **slow** which you can view using:

```
kubectl get storageclass
NAME      TYPE
slow      kubernetes.io/gce-pd
```

The example causes Kube to create the required PVCs automatically:

```
kubectl get pvc
NAME             STATUS    VOLUME                                     CAPACITY
ACCESSMODES   STORAGECLASS   AGE
pgdata-pgset-0   Bound     pvc-06334f6f-371b-11e7-9bda-42010a8000e9   1Gi        RWX
slow          5m
pgdata-pgset-1   Bound     pvc-063795b3-371b-11e7-9bda-42010a8000e9   1Gi        RWX
slow          5m
```

More information on dynamic storage provisioning can be found here: https://kubernetes.io/docs/concepts/storage/persistent-volumes/

secret

OpenShift

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
```

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
```

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.

pitr - PITR (point in time recovery)

Docker

This example, $CCPROOT/examples/docker/pitr, provides an example of performing a point in time recovery.

To run this example, run the following to create a database container:

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

It takes about 1 minute for the database to become ready for use after initially starting.

This database is created with the ARCHIVE_MODE and ARCHIVE_TIMEOUT environment variables set. See the pitr.adoc for more details on these settings. Warning: this example writes the WAL segment files to the /tmp directory…running it for a long time could fill up your /tmp!

Next, we will create a base backup of that database using this:

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

At this point, WAL segment files are created every 60 seconds that contain any database changes. These segments are stored in the /tmp/master-data/master-wal directory.

Next, create some data in your database using this command:

```
psql -h 127.0.0.1 -p 12000 -U postgres postgres -c "select
pg_create_restore_point('beforechanges')"
psql -h 127.0.0.1 -p 12000 -U postgres postgres -c 'create table pitrtest (id int)'
psql -h 127.0.0.1 -p 12000 -U postgres postgres -c "select
pg_create_restore_point('afterchanges')"
psql -h 127.0.0.1 -p 12000 -U postgres postgres -c "select
pg_create_restore_point('nomorechanges')"
psql -h 127.0.0.1 -p 12000 -U postgres postgres -c "checkpoint"
```

Next, stop the database to avoid conflicts with the WAL files while attempting to do a restore from them:

```
docker stop master-pitr
```

The commands above set restore point labels which we can use to mark the points in the recovery process we want to reference when creating our restored database. Points before and after the test table were made.

Next, lets edit the restore script to use the base backup files created in the step above. You can view the backup path name under /tmp/backups/master-pitr directory. You will see a value like **2016-09-21-21-03-29**. Copy and paste that value into the run-restore-pitr.sh script in the **BACKUP** environment variable.

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

Next, lets see if we can restore the database before we created the test table in the last command, we will restore using the backup and will use the **beforechanges** label as the restore target name in the PITR:

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

The WAL segments are read and applied when restoring from the database backup. At this point, you should be able to verify that the database was restored to the point before creating the test table:

```
psql -h 127.0.0.1 -p 12001 -U postgres postgres -c 'table pitrtest'
```

This sql command should show that the pitrtest table does not exist at this recovery time.

PostgreSQL allows you to pause the recovery process if the target name or time is specified. This pause would allow a DBA a chance to review the recovery time/name and see if this is what they want or expect. If so, the DBA can run the following command to resume and complete the recovery:

```
psql -h 127.0.0.1 -p 12001 -U postgres postgres -c 'select pg_xlog_replay_resume()'
```

Until you run the statement above, the database will be left in read-only mode.

Next, run the script to restore the database to the **afterchanges** restore point, do this by updating the RECOVERY_TARGET_NAME to **afterchanges**:

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

After this restore, you should be able to see the test table:

```
psql -h 127.0.0.1 -p 12001 -U postgres postgres -c 'table pitrtest'
psql -h 127.0.0.1 -p 12001 -U postgres postgres -c 'select pg_xlog_replay_resume()'
```

Lastly, lets recovery using all of the WAL files, this will get the restored database as current as possible, edit the script to remove the RECOVERY_TARGET_NAME environment setting completely:

```
./run-restore-pitr.sh
sleep 30
psql -h 127.0.0.1 -p 12001 -U postgres postgres -c 'table pitrtest'
psql -h 127.0.0.1 -p 12001 -U postgres postgres -c 'create table foo (id int)'
```

At this point, you should be able to create new data in the restored database and the test table should be present. When you recover the entire WAL history, resuming the recovery is not necessary to enable writes.

Other options exist for performing a PITR, see the pitr.adoc for full details.

Kubernetes

This example is identical to the OpenShift PITR example; please see below for details on how the PITR example works.

The only differences are the following:

- paths are **examples/kube/pitr**
- JSON and scripts are modifed to work with Kube
- **kubectl** commands are used instead of **oc** commands
- database services resolve to **default.svc.cluster.local** instead of **openshift.svc.cluster.local**

See PITR Documentation for details on PITR concepts and how PITR is implemented within the Suite.

OpenShift

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/pitr, 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
```

pgaudit - PGAudit

Docker

This example, $CCPROOT/examples/docker/pgaudit, provides an example of enabling pgaudit output. As of release 1.3, pgaudit is included in the crunchy-postgres container and is added to the postgres shared library list in the postgresql.conf.

Given the numerous ways pgaudit can be configured, the exact pgaudit configuration is left to the user to define. pgaudit allows you to configure auditing rules either in postgresql.conf or within your SQL script.

For this test, we place pgaudit statements within a SQL script and verify that auditing is enabled and working. If you choose to configure pgaudit via a postgresql.conf file, then you will need to define your own custom postgresql.conf file and mount it to override the default postgresql.conf file.

To run this example, run the following to create a database container:

```
cd $CCPROOT/examples/docker/pgaudit
./run.sh
```

This starts a database on port 12005 on localhost. You can then run the test script as follows:

```
./test-pgaudit.sh
```

This test executes a SQL file which contains pgaudit configuration statements as well as executes some basic SQL commands. These SQL commands will cause pgaudit to create audit log messages in the pg_log log file created by the database container.

swarm - Docker swarm

Docker

This example shows how to run a master and replica database container on a Docker Swarm (v.1.12) cluster.

First, set up a cluster, the Kubernetes libvirt coreos cluster example works well, see coreos-libvirt-cluster.

Next, on each node, create the Swarm using these Swarm Install instructions.

Includes the command on the manager node:

```
docker swarm init --advertise-addr 192.168.10.1
```

Then the command on all the worker nodes:

```
 docker swarm join \
     --token SWMTKN-1-65cn5wa1qv76l8l45uvlsbprogyhlprjpn27p1qxjwqmncn37o-
015egopg4jhtbmlu04faon82u \
         192.168.10.1.37
```

Before creating Swarm services, for service discovery you need to define an overlay network to be used by the services you will create. Create the network like this:

```
docker network create --driver overlay crunchynet
```

We want to have the master database always placed on a specific node. This is accomplished using node constraints as follows:

```
docker node inspect kubernetes-node-1 | grep ID
docker node update --label-add type=master 18yrb7m650umx738rtevojpqy
```

In the above example, the kubernetes-node-1 node with ID 18yrb7m650umx738rtevojpqy has a user defined label of **master** added to it. The master service specifies **master** as a constraint when created; this tells Swarm to place the service on that specific node. The replica specifies a constraint of **node.labels.type != master** to have the replica always placed on a node that is not hosting the master service.

After you set up the Swarm cluster, you can then run the **$CCPROOT/examples/docker/swarm-service** example as follows on the **Swarm Manager Node**:

```
cd $CCPROOT/examples/docker/swarm-service
./run.sh
```

You can then find the nodes that are running the master and replica containers by:

```
docker service ps master
docker service ps replica
```

Given the PostgreSQL replica service is named **replica**, you can scale up the number of replica containers by running this command:

```
docker service scale replica=2
docker service ls
```

You can verify you have two replicas within PostgreSQL by viewing the **pg_stat_replication** table, the password is **password**, when logged into the kubernetes-node-1 host:

```
docker exec -it $(docker ps -q) psql -U postgres -c 'table pg_stat_replication'
postgres
```

You should see a row for each replica along with its replication status.

watch - Automated failover

Docker

This example shows how to run the crunchy-watch container to perform an automated failover. For the example to work, the host on which you are running needs to allow read-write access to /run/docker.sock. The crunchy-watch container runs as the postgres user, so adjust the file permissions of /run/docker.sock accordingly.

Run the example as follows (depends on master-replica example being run prior):

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

This will start the watch container which tests every few seconds whether the master database is running, if not, it will trigger a failover (using docker exec) on the replica host.

Test it out by stopping the master:

```
docker stop master
docker logs watch
```

Look at the watch container logs to see it perform the failover.

Kubernetes

This example assumes you have run the master-replica example prior to this example!

This example runs a crunchy-watch container to look for the master within a postgres cluster, if it can not find the master it will proceed to cause a failover to a replica.

Running the example:

```
examples/kube/watch/run.sh
```

Check out the log of the watch container as follows:

```
kubectl log watch
```

Then trigger a failover using this command:

```
kubectl delete pod master
```

Resume watching the watch container’s log and verify that it detects the master is not reachable and performs a failover on the replica.

A final test is to see if the old replica is now a fully functioning master by inserting some test data into it as follows:

```
psql -h master -U postgres postgres -c 'create table failtest (id int)'
```

The above command still works because the watch container has changed the labels of the replica to make it a master, so the master service will still work and route now to the new master even though the pod is named replica.

Tip

You can view the labels on a pod with this command:

```
kubectl describe pod replica | grep Label
```

OpenShift

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 :

```
cd $CCPROOT/examples/openshift/watch
oc create -f watch-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:

```
./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 master-dc
oc logs watch
oc get pod
```

workshop - OpenShift Workshop

OpenShift

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.

upgrade - pg_upgrade

Kubernetes

Starting in release 1.3.1, the upgrade container will let you perform a pg_upgrade on a 9.5 database converting its data to a 9.6 version.

This example assumes you have run **master-pvc** using a 9.5 image such as **centos7-9.5-1.4.1** prior to running this upgrade.

Prior to starting this example, shut down the **master-pvc** database using the **examples/kube/master-pvc/cleanup.sh** script.

Prior to running this example, make sure your CCP_IMAGE_TAG environment variable is using a 9.6 image such as **centos7-9.6-1.4.1**.

Start the upgrade as follows:

```
cd $CCPROOT/examples/kube/upgrade
./run.sh
```

This will create the following in your Kube environment:

- a Kube Job running the **crunchy-upgrade** container
- a new data directory name **master-upgrade** found in the **pgnewdata** PVC

If successful, the Job will end with a Successful status, verify the results of the Job by examining the Job's pod log:

```
kubectl get pod -a -l job-name=upgrade-job
kubectl logs -l job-name=upgrade-job
```

You can verify the upgraded database by running the **examples/kube/master-upgrade** example, this example will mount the newly created and upgraded database files. Database tables and data that were in the **master-pvc** test database should be found in the **master-upgrade** database.

google cloud environment - Kubernetes cluster

Kubernetes

The Postgres Container Suite was tested on Google Container Engine.

Here is a link to set up a Kube cluster on GCE: https://kubernetes.io/docs/getting-started-guides/gce

Setup the persistent disks using GCE disks by first editing **examples/envvars.sh** and set the GCE settings to match your GCE environment.

Then create the PVs used by the examples, passing in the **gce** value as a parameter, this will cause the GCE disks to be created:

```
examples/pv/create-pv.sh gce
examples/pv/create-pvc.sh
```

Here is a link that describes more information on GCE persistent disk: https://cloud.google.com/container-engine/docs/tutorials/persistent-disk/

To have the persistent disk examples work, you will need to specify a **fsGroup** setting in the **SecurityContext** of each pod script as follows:

```
        "securityContext": {
         "fsGroup": 26
         },
```

For our Postgres container, we have specified a UID of 26 as the user which corresponds to the **fsGroup** value.

Docker - Tips

Send a signal to PostgreSQL

First, find the PID of the postmaster:

```
docker exec -it master cat /pgdata/master/postmaster.pid
```

Then, send it the signal to kill it or other signal depending on what you want to do:

```
docker exec -it master kill -SIGTERM 22
```

OpenShift - Tips

Find 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 ms-master -o json | grep PG
```

Look for the values of the environment variables:

- PG_USER
- PG_PASSWORD
- PG_DATABASE

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 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
```

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

Examine 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
```

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.

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.

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

nss_wrapper

If an OpenShift deployment requires that random generated UIDs 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 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

DNS configuration

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.

DNS host entry

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.

System policies for PV creation/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
```

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.

anyuid permissions

For my testing, I created a user named **test** on OSE, then I ran 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 most likely a smarter and more precise way to grant this permission but this is one suggested process.