
# Docker Examples - Crunchy Containers for PostgreSQL

Crunchy Data Solutions, Inc.

Version 1.2.5, 2016-10-26

# Standalone Examples

We provide some examples of running the Crunchy containers in the examples/standalone directory. Those examples are explained below.

## Example 1 - Running a single standalone database

Create the container with this command:

```
./run-pg.sh
```

This script will do the following:

- start up a container named crunchy-pg
- create a data directory for the database in /tmp/crunchy-pg
- 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.

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

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

To shut down the instance, run the following commands:

```
docker stop crunchy-pg
```

To start the instance, run the following commands:

```
docker start crunchy-pg
```

## Example 2 - Creating a master database

Run the container with this command:

```
./run-pg-master.sh
```

This script will do the following:

- start up a container named master

- create a data directory for the database in /tmp/master-data
- 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.

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

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

To shut down the instance, run the following commands:

```
docker stop master
```

To start the instance, run the following commands:

```
docker start master
```

# Example 3 - Performing a backup

In order to run this backup script, you first need to edit run-backup.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:

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

This script will do the following:

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

# Example 4 - Performing a restore

In order to run this backup script, you first need to edit run-restore.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:

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

This script will do the following:

- start up a container named masterrestore
- 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.

# Example 5 - Running PostgreSQL in Master-Slave Configuration

The container can be passed environment variables that will cause it to assume a PostgreSQL replication configuration with a master instance streaming to a slave replica instance.

The following env variables are specified for this configuration option:

- PG_MASTER_USER - The username used for master-slave replication value=master
- PG_MASTER_PASSWORD - The password for the PG master user
- PG_USER - The username that clients will use to connect to PG server value=user
- PG_PASSWORD - The password for the PG master user
- PG_DATABASE - The name of the database that will be created value=userdb
- PG_ROOT_PASSWORD - The password for the PG admin

This examples assumes you have run Example 1, and that the master container is running.

For running the master-slave configuration , you can run the following scripts:

```
run-pg-replica.sh
```

You can verify that replication is working by connecting to the replica as follows:

```
psql -h 127.0.0.1 -p 12002 -U postgres postgres
```

If you have created tables or data in the master database, they should show up in this replicated copy of that database.

# Example 6 - pgpool

A pgpool example is provided that will run a pgpool container that is configured to be used with the master and slave example provided in the run-pg-master.sh and run-pg-replica.sh scripts. After

running those commands to create a master and replica, you can create a pgpool container by running the following example command:

```
sudo ./run-pgpool.sh
```

Enter the following command to connect to the pgpool that is mapped to your local port 12002, in this case the host is named jeffded.crunchy.lab, fill in your hostname instead:

```
psql -h jeffded.crunchy.lab -U testuser -p 12002 testdb
```

You will enter the password of testpsw 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.

# Example 7 - pgbadger

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:

```
sudo ./run-badger.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
```

# Example 8 - dns

Some users will need or want a DNS name to resolve to their container names. The crunchy-dns container provides the following: * listens to a Docker URL or socket for events that would indicate a container is created or destroyed * implements the consul.io DNS server * registers new container information into the DNS server * deregisters container information from the DNS serve when a container is destroyed

Start the crunchy-dns server by running its container as follows:

```
sudo ./examples/standalone/run-dnsbridge.sh
```

This is a privledged container and will bind to your local IP address at port 53. At this point, you can now start a new Postgres container and you should be able to do a DNS lookup as follows:

```
dig @192.168.122.138 containername.service.dc1.crunchy.lab
```

In this example, the local IP address of the DNS container is 192.168.122.138 and it assumes you have started a container named containername

You can alter the DNS domain name within the startup script if desired.

You can also browse the consul web UI at:

```
http://<your ip address>:8500/ui
```

# Example 9 - metrics collection

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:

```
sudo ./examples/standalone/metrics/run-metrics.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:

```
sudo ./examples/standalone/run-collect.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.

# Example 10 - vacuum

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 examples/standalone/run-vacuum.sh script and can be run as follows:

```
sudo ./examples/standalone/run-pg-master.sh
sleep 30
sudo ./examples/standalone/run-vacuum.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

# Example 11 - custom setup.sql

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 examples/standalone/custom-setup/run.sh script and can be run as follows:

```
sudo ./examples/standalone/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.

# Example 12 - pgbouncer

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:

```
sudo ./examples/standalone/run-master.sh
sleep 20
sudo ./examples/standalone/run-pg-replica.sh
```

An example is shown in the examples/standalone/pgbouncer/run-pgbouncer.sh script and can be run as follows:

```
sudo ./examples/standalone/pgbouncer/run-pgbouncer.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.

# Example 13 - synchronous replication

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

To run this example, run the following:

```
./examples/standalone/sync/run-sync-master.sh
sleep 20
./examples/standalone/sync/run-sync-slave.sh
./examples/standalone/sync/run-async-slave.sh
```

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

```
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'
```

# Example 14 - pgadmin4

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

To run this example, run the following:

```
./examples/standalone/pgadmin4/run-pgadmin4.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.

# Example 15 - PITR (point in time recovery)

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

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

```
cd ./examples/standalone/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.asciidoc 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.asciidoc for full details.

# Tips - send a signal to postgres

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
```

# Legal Notices