Introduction

Contents

Crunchy Data Container Suite

Crunchy Container Suite is a collection of container images for PostgreSQL databases and various tools to manage them. The container images provided aim to be highly configurable, easy to deploy and reliable.

The suite is cloud agnostic and certified for production deployment with leading container platform technologies. Crunchy Container Suite supports all major public clouds and enables enables hybrid cloud deployments.

Crunchy Container Suite includes:

- Compatibility with Docker, Kubernetes, Red Hat OpenShift, VMWare Enterprise PKS
- Cloud-agnostic: build your own database-as-a-service in any public, private, or hybrid cloud

- No forks: 100% open source, native PostgreSQL
- Backup, restore, & disaster recovery for terabytes of data
- Graphical database administration tools for simple point-and-click database management
- Open source database monitoring and graphing tools to help analyze and create administration strategies
- PostGIS, robust open source GIS functionality, included by default
- Access to certified builds for Crunchy Certified PostgreSQL
- No proprietary software licensing fees

Why Crunchy Containers?

Enterprise production PostgreSQL deployments require more than just a database. Advanced capabilities like high availability, high-performant disaster recovery for terabytes of data, and monitoring solutions are requirements for your enterprise database cluster.

By providing the necessary microservices, including containers for scaling, high-availability, disaster recovery, monitoring, and more, Crunchy Container Suite will meet your production compliance, security, and performance requirements and give you a trusted open source database experience.

Elastic PostgreSQL

From creating uniformly managed, cloud-native production deployments to allowing your engineering team to provision databases that meet your compliance requirements, Crunchy Container Suite gives your organization the flexibility to deploy your own personalized database-as-a-service tailored to your needs.

Open Source For Enterprise

Crunchy Container Suite comes with essential open source tools for PostgreSQL management at scale, and lets you use powerful extensions like geospatial management with PostGIS.

Compliance At Scale

Deploy Crunchy Certified PostgreSQL with Crunchy Container Suite to harness the security of a Common Criteria EAL 2+ certified database on trusted platforms such as Red Hat OpenShift or Pivotal Container Service.

Overview

The following provides a high level overview of each of the container images.

CentOS vs Red Hat UBI Images

The Crunchy Container suite provides three different OS images: `centos7`, `ubi7`, and `ubi8-minimal`. Both images utilize Crunchy Ceritifed RPM's for the installation of PostgreSQL, and outside of the base images utilized to build the containers and any packages included within them (either CentOS or UBI), both are effectively the same. The `ubi7` and `ubi8-minimal` images are available to active Crunchy Data Customers only, and are built using the Red Hat Universal Base Image (UBI).

Please note that as of version 4.2.2 of the Crunchy Containers Suite, the `ubi7` images have replaced the `rhel7` images included in previous versions of the container suite. For more information on Red Hat UBI, please see the following link:

https://www.redhat.com/en/blog/introducing-red-hat-universal-base-image

Database Images

Crunchy Container Suite provides two types of PostgreSQL database images:

- Crunchy PostgreSQL
- Crunchy PostGIS

Supported major versions of these images are:

- 13

- 12
- 11
- 10

Crunchy PostgreSQL

Crunchy PostgreSQL is an unmodified deployment of the PostgreSQL relational database. It supports the following features:

- Asynchronous and synchronous replication
- Mounting custom configuration files such as `pg_hba.conf`, `postgresql.conf` and `setup.sql`
- Can be configured to use SSL authentication
- Logging to container logs
- Dedicated users for: administration, monitoring, connection pooler authentication, replication and user applications.
- pgBackRest backups built into the container
- Archiving WAL to dedicated volume mounts
- Extensions available in the PostgreSQL contrib module.
- Enhanced audit logging from the pgAudit extension
- Enhanced database statistics from the pg_stat_tatements extensions
- Python Procedural Language from the PL/Python extensions
- Backups
- Logical: `pg_dump` and `pg_restore` are included as logical backup and restore tools. These can be used to export and import SQL that recreates the database
- Physical: `pg_basebackup` is included as a physical backup tool. This can be used to backup the files that comprise the database
- Benchmarking with `pgbench`

Running Modes The Crunchy PostgreSQL container can be run in modes, specified by setting the `MODE` environment variable, enabling different features. Detailed information for configuring the `crunchy-postgres` container running modes can be found in the container specification [documentation]({{< relref "/container-specifications/crunchy-postgres" >}}).

Backup

The `backup` mode allows users to create pg_basebackup physical backups. The backups created by Crunchy Backup can be mounted to the Crunchy PostgreSQL container to restore databases.

pgBasebackup-restore

The `pgbasebasebackup-restore` mode allows users to restore from a pg_basebackup physical backup using `rsync`.

pgDump

The `pgdump` mode creates a logical backup of the database using the pg_dump tool. It supports the following features:

- `pg_dump` individual databases
- `pg_dump` all databases
- various formats of backups: plain (SQL), custom (compressed archive), directory (directory with one file for each table and blob being dumped with a table of contents) and tar (uncompressed tar archive)
- Logical backups of database sections such as: DDL, data only, indexes, schema

pgRestore

The `pgrestore` mode allows users to restore a PostgreSQL database from `pg_dump` logical backups using the pg_restore tool.

Sqlrunner

The `sqlrunner` mode uses psql to run all of the SQL files that are provided in the `/pgconf` volume.

pgBench

The `pgbench` mode allows users to run benchmarking tests on PostgreSQL using pgbench

Crunchy PostgreSQL PostGIS

The Crunchy PostgreSQL PostGIS mirrors all the features of the Crunchy PostgreSQL image but additionally provides the following geospatial extensions:

- PostGIS
- PostGIS Topology
- PostGIS Tiger Geocoder
- FuzzyStrMatch

Backup and Restoration Options

Crunchy Container Suite provides support for two types of backups:

- Physical - backups of the files that comprise the database
- Logical - an export of the SQL that recreates the database

Physical backup and restoration tools included in the Crunchy Container suite are:

- pgBackRest PostgreSQL images
- pg_basebackup - provided by the Crunchy PostgreSQL image

Logical backup and restoration tools are:

- pg_dump - provided by the Crunchy PostgreSQL image
- pg_restore - provided by the Crunchy PostgreSQL image

Crunchy BackRest Restore

The Crunchy BackRest Restore image restores a PostgreSQL database from pgBackRest physical backups. This image supports the following types of restores:

- Full - all database cluster files are restored and PostgreSQL replays Write Ahead Logs (WAL) to the latest point in time. Requires an empty data directory.
- Delta - missing files for the database cluster are restored and PostgreSQL replays Write Ahead Logs (WAL) to the latest point in time.
- PITR - missing files for the database cluster are restored and PostgreSQL replays Write Ahead Logs (WAL) to a specific point in time.

Visit the official pgBackRest website for more information: https://pgbackrest.org/

Administration

The following images can be used to administer and maintain Crunchy PostgreSQL database containers.

Crunchy pgAdmin4

The Crunchy pgAdmin4 images allows users to administer their Crunchy PostgreSQL containers via a graphical user interface web application.

Visit the official pgAdmin4 website for more information: https://www.pgadmin.org/

Crunchy Upgrade

The Crunchy Upgrade image allows users to perform major upgrades of their Crunchy PostgreSQL containers. The following upgrade versions of PostgreSQL are available:

- 13
- 12
- 11
- 10
- 9.6
- 9.5

Performance and Monitoring

The following images can be used to understand how Crunchy PostgreSQL containers are performing over time using tools such as pgBadger.

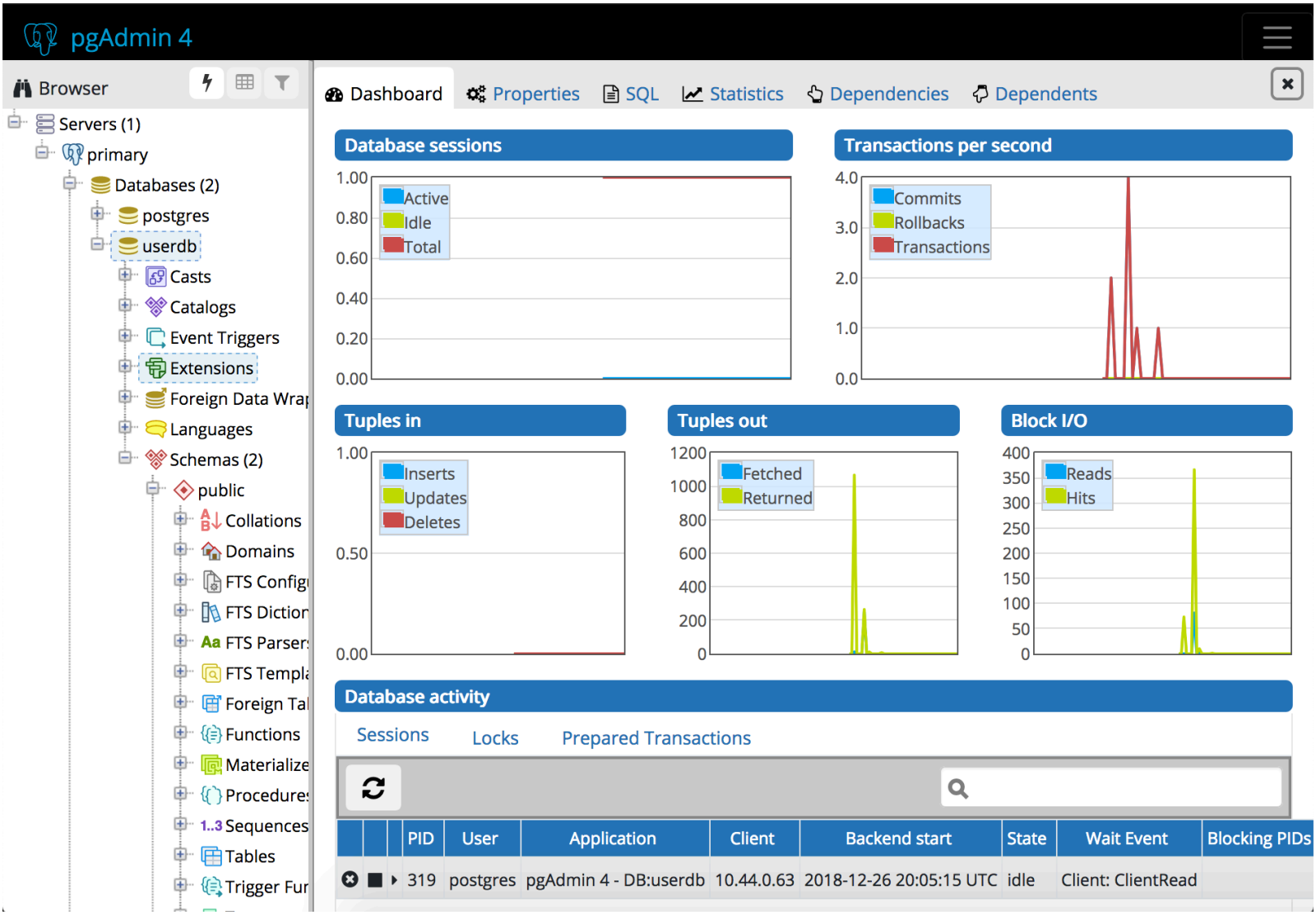

Figure 1: pgadmin4

Crunchy pgBadger

The Crunchy pgBadger image provides a tool that parses PostgreSQL logs and generates an in-depth statistical report. Crunchy pgBadger reports include:

- Connections
- Sessions
- Checkpoints
- Vacuum
- Locks
- Queries

Additionally Crunchy pgBadger can be configured to store reports for analysis over time.

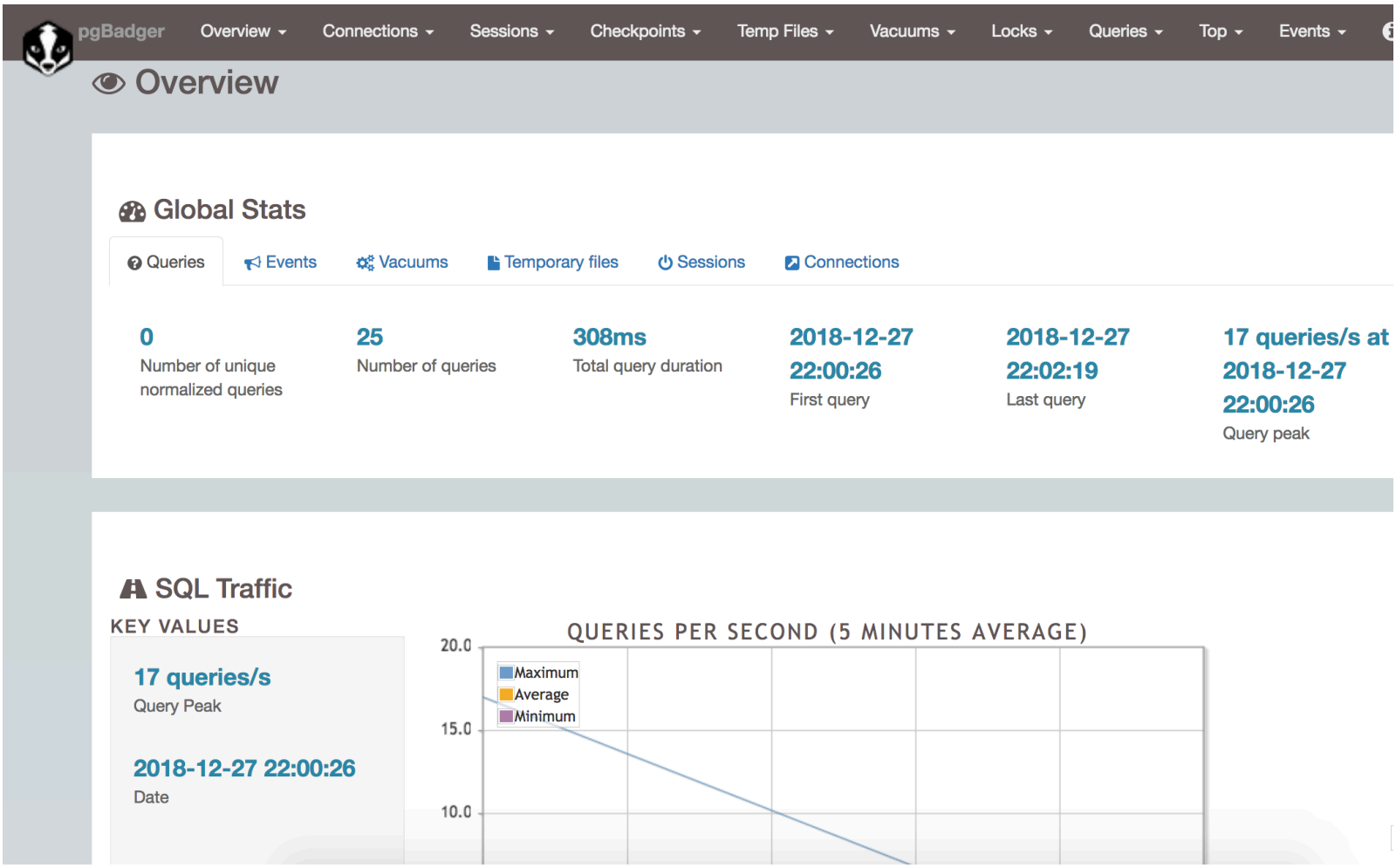

Figure 2: pgbadger

Visit the official pgBadger website for more information: https://pgbadger.darold.net/

Connection Pooling and Logical Routers

Crunchy pgBouncer

The Crunchy pgBouncer image provides a lightweight PostgreSQL connection pooler. Using pgBouncer, users can lower overhead of opening new connections and control traffic to their PostgreSQL databases. Crunchy pgBouncer supports the following features:

- Connection pooling
- Drain, Pause, Stop connections to Crunchy PostgreSQL containers
- Dedicated pgBouncer user for authentication queries
- Dynamic user authentication

Visit the official pgBouncer website for more information: https://pgbouncer.github.io

Crunchy pgPool II

The Crunchy pgPool image provides a logical router and connection pooler for Crunchy PostgreSQL containers. pgPool examines SQL queries and redirects write queries to the primary and read queries to replicas. This allows users to setup a single entrypoint for their applications without requiring knowledge of read replicas. Additionally pgPool provides connection pooling to lower overhead of opening new connections to Crunchy PostgreSQL containers.

Visit the official pgPool II website for more information: http://www.pgpool.net

Supported Platforms

Crunchy Container Suite supports the following platforms:

- *Docker 1.13+*
- *Kubernetes 1.12+*
- *OpenShift Container Platform 3.11*
- ***VMWare Enterprise PKS 1.3+***

  title: "Client User Guide" date: draft: false weight: 3 —

User Guide

Overview

This guide is intended to get you up and running with the Crunchy Container Suite, and therefore provides guidance for deploying the Crunchy Container Suite within your own environment. This includes guidance for standing-up and configuring your environment in order to run Crunchy Containers examples that can be found in the next section.

Please see the following sections in order to properly setup and configure your environment for the Crunchy Container Suite (***please feel free to skip any sections that have already been completed within your environment***):

1. Platform Installation
2. Crunchy Container Suite Installation
3. Storage Configuration
4. Example Guidance

Once your environment has been configured according to instructions provided above, you will be able to run the Crunchy Container Suite examples. These examples will demonstrate the various capabilities provided by the Crunchy Container Suite, including how to properly configure and deploy the various containers within the suite, and then utilize the features and services provided by those containers. The examples therefore demonstrate how the Crunchy Container Suite can be utilized to effectively deploy a PostgreSQL database cluster within your own environment, that meets your specific needs and contains the PostgreSQL features and services that you require.

Platform Installation

In order to run the examples and deploy various containers within the Crunchy Container Suite, you will first need access to an environment containing one of the following supported platforms:

- Docker 1.13+ (https://www.docker.com/)
- Kubernetes 1.8+ (https://kubernetes.io/)
- OpenShift Container Platform 3.11 (https://www.openshift.com/products/container-platform/)

Links to the official website for each of these platform are provided above. Please consult the official documentation for instructions on how to install and configure these platforms in your environment.

Crunchy Container Suite Installation

Once you have access to an environment containing one of the supported platforms, it is then necessary to properly configure that environment in order to run the examples, and therefore deploy the various containers included in the Crunchy Container Suite. This can be done by following the Crunchy Container Suite Installation Guide.

Please note that as indicated within the Installation Guide, certain steps may require administrative access and/or privileges. Therefore, please work with your local System Administrator(s) as needed to setup and configure your environment according to the steps defined within this guide. Additionally, certain steps are only applicable to certain platforms and/or environments, so please be sure to follow all instructions that are applicable to your target environment.

Storage Configuration

Once you have completed all applicable steps in the Installation Guide, you can then proceed with configuring storage in your environment. The specific forms of storage supported by the Crunchy Containers Suite, as well as instructions for configuring and enabling those forms of storage, can be found in the Storage Configuration guide. Therefore, please review and follow steps in the Storage Configuration guide in order to properly configure storage in your environment according to your specific storage needs.

Example Guidance

With the Installation Guide and Storage Configuration complete, you are almost ready to run the examples. However, prior to doing so it is recommended that you first review the documentation for Running the Examples, which describes various conventions utilized in the examples, while also providing any other information, resources and guidance relevant to successfully running the Crunchy Container Suite examples in your environment. The documentation for running the examples can be found here.

Running the Examples

The Kubernetes and OpenShift examples in this guide have been designed using single-node Kubernetes/OCP clusters whose host machines provide any required supporting infrastructure or services (e.g. local HostPath storage or access to an NFS share). Therefore, for the best results when running these examples, it is recommended that you utilize a single-node architecture as well.

Additionally, the examples located in the **kube** directory work on both Kubernetes and OpenShift. Please ensure the `CCP_CLI` environment variable is set to the correct binary for your environment, as shown below:

```
# Kubernetes
export CCP_CLI=kubectl

# OpenShift
export CCP_CLI=oc
```

NOTE: *Set the `CCP_CLI` environment variable in `.bashrc` to ensure the examples will work properly in your environment*

Example Conventions

The examples provided in Crunchy Container Suite are simple examples that are meant to demonstrate key Crunchy Container Suite features. These examples can be used to build more production level deployments as dictated by user requirements specific to their operating environments.

The examples generally follow these conventions: - There is a **run.sh** script that you will execute to start the example - There is a **cleanup.sh** script that you will execute to shutdown and cleanup the example - Each example will create resources such as Secrets, ConfigMaps, Services, and PersistentVolumeClaims, all which follow a naming convention of `<example name>-<optional description suffix>`. For example, an example called **primary** might have a PersistentVolumeClaim called **primary-pgconf** to describe the purpose of that particular PVC. - The folder names for each example give a clue as to which Container Suite feature it demonstrates. For instance, the `examples/kube/pgaudit` example demonstrates how to enable the **pg_audit** capability in the **crunchy-postgres** container.

Helpful Resources

Here are some useful resources for finding the right commands to troubleshoot and modify containers in the various environments shown in this guide:

- Docker Cheat Sheet
- Kubectl Cheat Sheet

- OpenShift Cheat Sheet
- Helm Cheat Sheet

Crunchy Container Suite Examples

Now that your environment has been properly configured for the Crunchy Container Suite and you have reviewed the guidance for running the examples, you are ready to run the Crunchy Container Suite examples. Therefore, please proceed to the next section in order to find the examples that can now be run in your environment. — title: "Examples" date: draft: false weight: 4 — — title: "PostgreSQL Primary" date: draft: false weight: 1 — # PostgreSQL Container Example

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

The container creates a default database called *userdb*, a default user called *testuser* and a default password of *password*.

For all environments, the script additionally creates:

- A persistent volume claim
- A crunchy-postgres container named *primary*
- The database using predefined environment variables

And specifically for the Kubernetes and OpenShift environments:

- A pod named *primary*
- A service named *primary*
- A PVC named *primary-pgdata*
- The database using predefined environment variables

To shutdown the instance and remove the container for each example, run the following:

```
./cleanup.sh
```

Docker

To create the example and run the container:

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

Connect from your local host as follows:

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

Kubernetes and OpenShift

To create the example:

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

Connect from your local host as follows:

```
psql -h primary -U postgres postgres
```

Helm

This example resides under the `$CCPROOT/examples/helm` directory. View the README to run this example using Helm here.

Custom Configuration

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

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 container to work; please see comments within the sample `setup.sql` file.

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

Docker

This example can be run as follows for the Docker environment:

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

Kubernetes and OpenShift

Running the example:

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

SSL Authentication

This example shows how you can configure PostgreSQL to use SSL for client authentication.

The example requires SSL certificates and keys to be created. Included in the examples directory is a script to create self-signed certificates (server and client) for the example: `$CCPROOT/examples/ssl-creator.sh`.

The example creates a client certificate for the user `testuser`. Furthermore, the server certificate is created for the server name `custom-config-ssl`.

This example can be run as follows for the Docker environment:

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

And the example can be run in the following directory for the Kubernetes and OpenShift environments:

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

A required step to make this example work is to define in your `/etc/hosts` file an entry that maps `custom-config-ssl` to the service IP address for the container.

For instance, if your service has an address as follows:

```
${CCP_CLI} get service
NAME                CLUSTER-IP       EXTERNAL-IP   PORT(S)                      AGE
custom-config-ssl   172.30.211.108   <none>        5432/TCP
```

Then your `/etc/hosts` file needs an entry like this:

```
172.30.211.108 custom-config-ssl
```

For production Kubernetes and OpenShift installations, it will likely be preferred for DNS names to resolve to the PostgreSQL service name and generate server certificates using the DNS names instead of the example name `custom-config-ssl`.

If as a client it's required to confirm the identity of the server, `verify-full` can be specified for `ssl-mode` in the connection string. This will check if the server and the server certificate have the same name. Additionally, the proper connection parameters must be specified in the connection string for the certificate information required to trust and verify the identity of the server (`sslrootcert` and `sslcrl`), and to authenticate the client using a certificate (`sslcert` and `sslkey`):

```
psql "postgresql://testuser@custom-config-ssl:5432/userdb?\
sslmode=verify-full&\
sslrootcert=$CCPROOT/examples/kube/custom-config-ssl/certs/ca.crt&\
sslcrl=$CCPROOT/examples/kube/custom-config-ssl/certs/ca.crl&\
sslcert=$CCPROOT/examples/kube/custom-config-ssl/certs/client.crt&\
sslkey=$CCPROOT/examples/kube/custom-config-ssl/certs/client.key"
```

To connect via IP, `sslmode` can be changed to `require`. This will verify the server by checking the certificate chain up to the trusted certificate authority, but will not verify that the hostname matches the certificate, as occurs with `verify-full`. The same connection parameters as above can be then provided for the client and server certificate information.

```
psql "postgresql://testuser@IP_OF_PGSQL:5432/userdb?\
sslmode=require&\
sslrootcert=$CCPROOT/examples/kube/custom-config-ssl/certs/ca.crt&\
sslcrl=$CCPROOT/examples/kube/custom-config-ssl/certs/ca.crl&\
sslcert=$CCPROOT/examples/kube/custom-config-ssl/certs/client.crt&\
sslkey=$CCPROOT/examples/kube/custom-config-ssl/certs/client.key"
```

You should see a connection that looks like the following:

```
psql (11.10)
SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384, bits: 256, compression:
   off)
Type "help" for help.

userdb=>
```

Replication

This example starts a primary and a replica pod containing a PostgreSQL database.

The container creates a default database called *userdb*, a default user called *testuser* and a default password of *password*.

For the Docker environment, the script additionally creates:

- A docker volume using the local driver for the primary
- A docker volume using the local driver for the replica
- A container named *primary* binding to port 12007
- A container named *replica* binding to port 12008
- A mapping of the PostgreSQL port 5432 within the container to the localhost port 12000
- The database using predefined environment variables

And specifically for the Kubernetes and OpenShift environments:

- emptyDir volumes for persistence
- A pod named *pr-primary*
- A pod named *pr-replica*
- A pod named *pr-replica-2*
- A service named *pr-primary*
- A service named *pr-replica*
- The database using predefined environment variables

To shutdown the instance and remove the container for each example, run the following:

```
./cleanup.sh
```

Docker

To create the example and run the container:

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

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
```

Kubernetes and OpenShift

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

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

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

```
${CCP_CLI?} exec -ti pr-primary -- psql -d postgres -c 'table pg_stat_replication'
```

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

```
${CCP_CLI?} exec -ti pr-primary -- psql -d postgres -c 'create table foo (id int)'
${CCP_CLI?} exec -ti pr-primary -- psql -d postgres -c 'insert into foo values (1)'
```

Then verify that the data is replicated to the replica:

```
${CCP_CLI?} exec -ti pr-replica -- psql -d postgres -c 'table foo'
```

primary-replica-dc

If you wanted to experiment with scaling up the number of replicas, you can run the following example:

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

You can verify that replication is working using the same commands as above.

```
${CCP_CLI?} exec -ti primary-dc -- psql -d postgres -c 'table pg_stat_replication'
```

Helm

This example resides under the `$CCPROOT/examples/helm` directory. View the README to run this example using Helm here.

Synchronous Replication

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

To shutdown the instance and remove the container for each example, run the following:

```
./cleanup.sh
```

Docker

To run this example, run the following:

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

You can test the replication status on the primary by using the following command and the password *password*:

```
psql -h 127.0.0.1 -p 12010 -U postgres postgres -c 'table pg_stat_replication'
```

You should see 2 rows; 1 for the asynchronous replica and 1 for the synchronous replica. The `sync_state` column shows values of async or sync.

You can test replication to the replicas by first entering some data on the primary, and secondly querying the replicas for that data:

```
psql -h 127.0.0.1 -p 12010 -U postgres postgres -c 'create table foo (id int)'
psql -h 127.0.0.1 -p 12010 -U postgres postgres -c 'insert into foo values (1)'
psql -h 127.0.0.1 -p 12011 -U postgres postgres -c 'table foo'
psql -h 127.0.0.1 -p 12012 -U postgres postgres -c 'table foo'
```

Kubernetes and OpenShift

Running the example:

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

Connect to the *primarysync* and *replicasync* databases as follows for both the Kubernetes and OpenShift environments:

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

This set of queries will show you the IP address of the PostgreSQL replica container. Note the changing IP address due to the round-robin service proxy being used for both replicas. The example queries also show that both replicas are replicating successfully from the primary.

Geospatial (PostGIS)

An example is provided that will run a PostgreSQL with PostGIS pod and service in Kubernetes and OpenShift and a container in Docker.

The container creates a default database called *userdb*, a default user called *testuser* and a default password of *password*.

You can view the extensions that postgres-gis has enabled by running the following command and viewing the listed PostGIS packages:

```
psql -h postgres-gis -U testuser userdb -c '\dx'
```

To validate that PostGIS is installed and which version is running, run the command:

```
psql -h postgres-gis -U testuser userdb -c "SELECT postgis_full_version();"
```

You should expect to see output similar to:

```
postgis_full_version
-----------------------------------------------------------------------------------------------------------
 POSTGIS="2.4.8 r16113" PGSQL="100" GEOS="3.5.0-CAPI-1.9.0 r4084" PROJ="Rel. 4.8.0, 6 March 2012"
    GDAL="GDAL 1.11.4, released 2016/01/25" LIBXML="2.9.1" LIBJSON="0.11" TOPOLOGY RASTER
(1 row)
```

As an exercise for invoking some of the basic PostGIS functionality for validation, try defining a 2D geometry point while giving inputs of longitude and latitude through this command.

```
psql -h postgres-gis -U testuser userdb -c "select ST_MakePoint(28.385200,-81.563900);"
```

You should expect to see output similar to:

```
                st_makepoint
--------------------------------------------
 0101000000516B9A779C623C40B98D06F0166454C0
(1 row)
```

To shutdown the instance and remove the container for each example, run the following:

```
./cleanup.sh
```

Docker

Create the container as follows:

```
cd $CCPROOT/examples/docker/postgres-gis
./run.sh
```

Enter the following command to connect to the postgres-gis container that is mapped to your local port 12000:

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

Kubernetes and OpenShift

Running the example:

```
cd $CCPROOT/examples/kube/postgres-gis
./run.sh
```

pgPool Logical Router Example

An example is provided that will run a *pgPool II* container in conjunction with the *primary-replica* example provided above.

You can execute both `INSERT` and `SELECT` statements after connecting to pgpool. The container will direct `INSERT` statements to the primary and `SELECT` statements will be sent round-robin to both the primary and replica.

The container creates a default database called *userdb*, a default user called *testuser* and a default password of *password*.

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

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

To shutdown the instance and remove the container for each example, run the following:

```
./cleanup.sh
```

Docker

Create the container as follows:

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

The example is configured to allow the *testuser* to connect to the *userdb* database.

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

Kubernetes and OpenShift

Run the following command to deploy the pgpool service:

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

The example is configured to allow the *testuser* to connect to the *userdb* database.

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

pgBackRest Examples

Written and maintained by David Steele, pgBackRest is a utility that provides backup and restore functionality for PostgreSQL databases. pgBackRest is available for use within the Crunchy Container Suite, and can therefore be utilized to provide an effective backup and restore solution for any database clusters deployed using the crunchy-postgres or crunchy-postgres-gis containers. The following section will provide an overview of how pgBackRest can be utilized within the Crunchy Container Suite, including examples for enabling and configuring pgBackRest, and then utilizing pgBackRest to backup and restore various PostgreSQL database clusters. For more detailed information about pgBackRest, please visit the official pgBackRest website.

Configuration Overview

In order to enable pgBackRest within a crunchy-postgres or crunchy-postgres-gis container, environment variable `PGBACKREST` must be set to `true` during deployment of the container (`PGBACKREST=true`). This will setup the proper pgBackRest configuration, ensure any required pgBackRest repositories and directories are created, and will create the proper pgBackRest stanza.

Please note that setting `PGBACKREST=true` is all that is needed to configure and enable pgBackRest within a crunchy-postgres or crunchy-postgres-gis container. When enabled, default environment variables will be set for pgBackRest as follows, unless they are otherwise explicitly defined and provided during deployment of the container:

```
export PGBACKREST_STANZA="db"
export PGBACKREST_PG1_PATH="/pgdata/${PGDATA_DIR}"
export PGBACKREST_REPO1_PATH="/backrestrepo/${PGDATA_DIR}-backups"
export PGBACKREST_LOG_PATH="/tmp"
```

As shown above, a stanza named `db` is created by default, using the default values provided for both `PGBACKREST_PG1_PATH` and `PGBACKREST_REPO1_PATH`. Variable `PGDATA_DIR` represents the name of the database cluster's data directory, which will either be the hostname of the container or the value specified for variable `PGDATA_PATH_OVERRIDE` during deployment of the container. Please see the crunchy-postgres and/or crunchy-postgres-gis container specifications for additional details.

While setting `PGBACKREST` to `true` provides a simple method for enabling pgBackRest within a crunchy-postgres or crunchy-postgres-gis container, pgBackRest is also fully configurable and customizable via the various environment variables supported by pgBackRest. This applies to the crunchy-backrest-restore container as well, which is also configured using pgBackRest environment variables when performing database restores. Therefore, during the deployment of any container container containing pgBackRest (crunchy-postgres, crunchy-postgres-gis or crunchy-backrest-restore), environment variables should be utilized to configure and customize the pgBackRest utility as needed and ensure the desired backup and restore functionality is achieved. For instance, the following environment variables could be specified upon deployment of the crunchy-backrest-restore container in order to perform delta restore to a specific point-in-time:

```
PGBACKREST_TYPE=time
PITR_TARGET="2019-10-27 16:53:05.590156+00"
PGBACKREST_DELTA=y
```

Database restores can be performed via the crunchy-backrest-restore container, which offers full pgBackRest restore capabilities, such as full, point-in-time and delta restores. Further information and guidance for performing both backups and restores using the Crunchy Container Suite and pgBackRest will be provided in the examples below.

In addition to providing the backup and restoration capabilities discussed above, pgBackRest supports the capability to asynchronously push and get write ahead logs (WAL) to and from a WAL archive. To enable asychronous WAL archiving within a crunchy-postgres or crunchy-postgres-gis container, pgBackRest environment variable `PGBACKREST_ARCHIVE_ASYNC` must be set to `"y"` during deployment (`PGBACKREST_ARCHIVE_ASYNC=y`). This will automatically enable WAL archiving within the container if not otherwise explicitly enabled, set the proper `pgbackrest archive` command within the `postgresql.conf` configuration file, and ensure the proper spool path has been created.

If a spool path is not explicitly provided using environment variable `PGBACKREST_SPOOL_PATH`, this variable will default as follows:

```
# Environment variable XLOGDIR="true"
export PGBACKREST_SPOOL_PATH="/pgdata/${PGDATA_DIR}"

# Environment variable XLOGDIR!=true
export PGBACKREST_SPOOL_PATH="/pgwal/${PGDATA_DIR}/spool"
```

As shown above, the default location of the spool path depends on whether or not `XLOGDIR=true`, with `XLOGDIR` enabling the storage of WAL to the `/pgwal` volume within the container. Being that pgBackRest recommends selecting a spool path that is as close to the WAL as possible, this provides a sensible default for the spool directory. However, `PGBACKREST_SPOOL_PATH` can also be explicitly configured during deployment to any path desired. And once again, `PGDATA_DIR` represents either the hostname of the container or the value specified for variable `PGDATA_PATH_OVERRIDE`.

The examples below will demonstrate the pgBackRest backup, restore and asynchronous archiving capabilities described above, while also providing insight into the proper configuration of pgBackBackrest within the Crunchy Container Suite. For more information on these pgBackRest capabilities and associated configuration, please consult the official pgBackRest documentation.

Kubernetes and OpenShift

The pgBackRest examples for Kubernetes and OpenShift can be configured to use the PostGIS images by setting the following environment variable when running the examples:

```
export CCP_PG_IMAGE='-gis'
```

Backup

In order to demonstrate the backup and restore capabilities provided by pgBackRest, it is first necessary to deploy a PostgreSQL database, and then create a full backup of that database. This example will therefore deploy a crunchy-postgres or crunchy-postgres-gis container containing a PostgreSQL database, which will then be backed up manually by executing a `pgbackrest backup` command. ***Please note that this example serves as a prequisite for the restore examples that follow, and therefore must be run prior to running those examples.***

Start the example as follows:

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

This will create the following in your Kubernetes environment:

- A deployment named **backrest** containing a PostgreSQL database with pgBackRest configured
- A service named **backrest** for the PostgreSQL database
- A PV and PVC for the PGDATA directory
- A PV and PVC for the pgBackRest backups and archives directories

Once the **backrest** deployment is running, use the `pgbackrest info` command to verify that pgbackrest has been properly configured and WAL archiving is working properly:

```
$ ${CCP_CLI} exec <backrest pod name> -- pgbackrest info \
  --stanza=db \
  --repo1-path=/backrestrepo/backrest-backups

pg_pid=126
stanza: db
    status: error (no valid backups)
    cipher: none

    db (current)
        wal archive min/max (11-1): 000000010000000000000001 / 000000010000000000000003
```

An output similar to the above indicates that pgBackRest was properly configured upon deployment of the pod, the **db** stanza has been created, and WAL archiving is working properly. The error next to **status** is expected being that a backup has not yet been generated.

Now that we have verified that pgBackRest is properly configured and enabled, a backup of the database can be generated. Being that this is the first backup of the database, we will take create a **full** backup:

```
$ ${CCP_CLI} exec <backrest pod name> -- pgbackrest backup \
  --stanza=db \
  --pg1-path=/pgdata/backrest \
  --repo1-path=/backrestrepo/backrest-backups \
  --log-path=/tmp \
  --type=full

pg_pid=138
WARN: option repo1-retention-full is not set, the repository may run out of space
      HINT: to retain full backups indefinitely (without warning), set option
         'repo1-retention-full' to the maximum.
```

The warning displayed is expected, since backup retention has not been configured for this example. Assuming no errors are displayed, a full backup has now been successfully created.

Restore

pgBackRest provides numerous methods and strategies for restoring a PostgreSQL database. The following section will demonstrate three forms of database restores that can be accomplished when using pgBackRest with the Crunchy Container Suite:

- **Full:** restore all database files into an empty PGDATA directory
- **point-in-time Recovery (PITR):** restore a database to a specific point-in-time using an empty PGDATA directory
- **Delta:** restore a database to a specific point-in-time using an existing PGDATA directory

Full This example will demonstrate a full database restore to an empty PGDATA directory. ***Please ensure the Backup example is currently running and a full backup has been generated prior to running this example.***

Prior to running the full restore, we will first make a change to the currently running database, which will we will then verify still exists following the restore. Create a simple table in the database as follows:

```
$ ${CCP_CLI} exec <backrest pod name> -- psql -c "create table backrest_test_table (id int)"
CREATE TABLE
```

Now verify that the new table exists:

```
$ ${CCP_CLI} exec <backrest pod name> -- psql -c "table backrest_test_table"
 id
----
(0 rows)
```

With the table in place, we can now start the full restore as follows:

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

This will create the following in your Kubernetes environment:

- A Kubernetes job named **backrest-full-restore-job** which will perform the restore using the crunchy-backrest-restore container
- A PV and PVC for the new PGDATA directory that will contain the restored database. The directory will initially be empty, as required pgBackRest when performing a full restore, and will then contain the restored database upon completion of the restore.

Please note that a brand new PV and PVC are created when running the restore to clearly indicate that the database will be restored into an entirely new (i.e. empty) volume as required by pgBackRest. The names of the new PV and PVC are as follows:

- **PV:** ${CCP_NAMESPACE}-br-new-pgdata
- **PVC:** br-new-pgdata

You can verify that the restore has completed successfully by verifying that the Kubernetes job has completed successfully:

```
$ ${CCP_CLI} get jobs
NAME                        COMPLETIONS   DURATION   AGE
backrest-full-restore-job   1/1           15s        58s
```

Once the job is complete, the post restore script can then be run, which will create a new deployment named **backrest-full-restored** containing the restored database:

```
cd $CCPROOT/examples/kube/backrest/full
./post-restore.sh
```

Finally, once the **backrest-full-restored** deployment is running we can verify that the restore was successful by verifying that the table created prior to the restore still exists:

```
$ ${CCP_CLI} exec <backrest restored pod name> -- psql -c "table backrest_test_table"
 id
----
(0 rows)
```

Please note that the default behavior of pgBackRest is to recover to the end of the WAL archive stream, which is why the full restore contained all changes made since the initial full backup was taken, including the creation of table **backrest_test_table**. pgBackRest therefore played the entire WAL archive stream for all changes that occurred up until the restore.

*As a reminder, please remember to run the cleanup script for the **Backup** example after running the cleanup script for this example.*

PITR As demonstrated with the full restore above, the default behavior of pgBackRest is to recover to the end of the WAL archive stream. However, pgBackRest also provides the ability to recover to a specific point-in-time utilizing the WAL archives created since the last backup. This example will demonstrate how pgBackRest can be utilized to perform a point-in-time recovery (PITR) and therefore recover the database to specific point-in-time specified by the user. ***Please ensure that the Backup example is currently running and a full backup has been generated prior to running this example.***

Prior to running the PITR restore, we will first verify the current state of the database, after which we will then make a change to the database. This will allow us to verify that the PITR is successful by providing a method of verifying that the database has been restored to its current state following the restore.

To verify the current state of the database, we will first verify that a table called **backrest_test_table** does not exist in the database.

```
$ ${CCP_CLI} exec <backrest pod name> -- psql -c " table backrest_test_table"
ERROR:  relation "backrest_test_table" does not exist
LINE 1:  table backrest_test_table
               ^
command terminated with exit code 1
```

Next, capture the current timestamp, which will be used later in the example when performing the restore:

```
$ ${CCP_CLI} exec <backrest pod name> -- psql -c "select current_timestamp"
       current_timestamp
-------------------------------
 2019-10-27 16:53:05.590156+00
(1 row)
```

Now create table **backrest_test_table**:

```
$ ${CCP_CLI} exec <backrest pod name> -- psql -c "create table backrest_test_table (id int)"
CREATE TABLE
```

Then verify that the new table exists:

```
$ ${CCP_CLI} exec <backrest pod name> -- psql -c "table backrest_test_table"
 id
----
(0 rows)
```

With the table in place, we can now start the PITR. However, the timestamp captured above must also be provided in order to instruct pgBackRest to recover to that specific point-in-time. This is done using the `CCP_BACKREST_TIMESTAMP` variable, which allows us to then start the PITR as follows (replace the timestamp in the command below with the timestamp you captured above):

```
cd $CCPROOT/examples/kube/backrest/pitr
CCP_BACKREST_TIMESTAMP="2019-10-27 16:53:05.590156+00" ./run.sh
```

This will create the following in your Kubernetes environment: - A Kubernetes job named **backrest-pitr-restore-job** which will perform the restore using the crunchy-backrest-restore container

Additionally, when this example is run, the following pgBackRest environment variables are provided to the crunchy-backrest-restore container in order to initiate PITR restore to the point-in-time specified by the timestamp (in additional to any other pgBackRest variables required by the Crunchy Container Suite and pgBackRest):

```
PGBACKREST_TYPE=time
PITR_TARGET="${CCP_BACKREST_TIMESTAMP}"
```

As can be seen above, the timestamp provided for `CCP_BACKREST_TIMESTAMP` is used to populate variable `PITR_TARGET`, and therefore specify the point-in-time to restore the database to, while `PGBACKREST_TYPE` is set to `time` to indicate that a PITR should be performed.

Please note that the following pgBackRest environment variable is also set when performing the PITR, which results in a restore to a new/empty directory within an existing PV:

```
PGBACKREST_PG1_PATH=/pgdata/backrest-pitr-restored
```

You can verify that the restore has completed successfully by verifying that the Kubernetes job has completed successfully:

```
$ ${CCP_CLI} get jobs
NAME                        COMPLETIONS   DURATION   AGE
backrest-pitr-restore-job   1/1           15s        58s
```

Once the job is complete, the post restore script can then be run, which will create a new deployment named **backrest-pitr-restored** containing the restored database:

```
cd $CCPROOT/examples/kube/backrest/pitr
./post-restore.sh
```

Finally, once the **backrest-pitr-restored** deployment is running we can verify that the restore was successful by verifying that the table created prior to the restore no longer exists:

```
$ ${CCP_CLI} exec <backrest restored pod name> -- psql -c " table backrest_test_table"
ERROR:  relation "backrest_test_table" does not exist
LINE 1:  table backrest_test_table
               ^
command terminated with exit code 1
```

*As a reminder, please remember to run the cleanup script for the **Backup** example after running the cleanup script for this example.*

Delta By default, pgBackRest requires a clean/empty directory in order to perform a restore. However, pgBackRest also provides an another option when performing the restore in the form of the **delta** option, which allows the restore to be run against an existing PGDATA directory. With the delta option enabled, pgBackRest will use checksums to determine which files in the directory can be preserved, and which need to be restored (please note that pgBackRest will also remove any files that are not present in the backup). This example will again demonstrate a point-in-time recovery (PITR), only this time the restore will occur within the existing PGDATA directory by specifying the **delta** option during the restore. ***Please ensure that the Backup example is currently running and a full backup has been generated prior to running this example.***

Prior to running the delta restore, we will first verify the current state of the database, and we will then make a change to the database. This will allow us to verify that the delta restore is successful by providing a method of verifying that the database has been restored to its current state following the restore.

To verify the current state of the database, we will first verify that a table called **backrest_test_table** does not exist in the database.

```
$ ${CCP_CLI} exec <backrest pod name> -- psql -c " table backrest_test_table"
ERROR:  relation "backrest_test_table" does not exist
LINE 1:  table backrest_test_table
               ^
command terminated with exit code 1
```

Next, capture the current timestamp, which will be used later in the example when performing the restore:

```
$ ${CCP_CLI} exec <backrest pod name> -- psql -c "select current_timestamp"
       current_timestamp
-------------------------------
 2019-10-27 16:53:05.590156+00
(1 row)
```

Now create table **backrest_test_table**:

```
$ ${CCP_CLI} exec <backrest pod name> -- psql -c "create table backrest_test_table (id int)"
CREATE TABLE
```

Then verify that the new table exists:

```
$ ${CCP_CLI} exec <backrest pod name> -- psql -c "table backrest_test_table"
 id
----
(0 rows)
```

With the table in place, we can now start the delta restore. When running the restore example the timestamp captured above must also be provided in order to instruct pgBackRest to recover to that specific point-in-time. This is done using the `CCP_BACKREST_TIMESTAMP` variable, which allows us to then start the delta restore as follows (replace the timestamp in the command below with the timestamp you captured above):

```
cd $CCPROOT/examples/kube/backrest/delta
CCP_BACKREST_TIMESTAMP="2019-10-27 16:53:05.590156+00" ./run.sh
```

This will create the following in your Kubernetes environment: - A Kubernetes job named **backrest-delta-restore-job** which will perform the restore using the crunchy-backrest-restore container

Additionally, when this example is run, the following pgBackRest environment variables are provided to the crunchy-backrest-restore container in order to initiate a delta restore to the point-in-time specified by the timestamp (in additional to any other pgBackRest variables required by the Crunchy Container Suite and pgBackRest):

```
PGBACKREST_TYPE=time
PITR_TARGET="${CCP_BACKREST_TIMESTAMP}"
PGBACKREST_DELTA=y
```

As can be seen above, the timestamp provided for `CCP_BACKREST_TIMESTAMP` is used to populate variable `PITR_TARGET`, and therefore specify the point-in-time to restore to, while `PGBACKREST_TYPE` is set to `time` to indicate that a PITR should be performed. `PGBACKREST_DELTA` is set to `y` to indicate that the delta option should be utilized when performing the restore.

It's also worth noting that the following pgBackRest environment variable is also set when performing the delta restore, which results in a restore within the existing PGDATA directory utilized by the database deployed when running the **Backup** example:

```
PGBACKREST_PG1_PATH=/pgdata/backrest
```

You can then verify that the restore has completed successfully by verifying that the Kubernetes job has completed successfully:

```
$ ${CCP_CLI} get jobs
NAME                          COMPLETIONS   DURATION   AGE
backrest-delta-restore-job    1/1           15s        58s
```

Once the job is complete, the post restore script can then be run, which will create a new deployment named **backrest-delta-restored** containing the restored database:

```
cd $CCPROOT/examples/kube/backrest/delta
./post-restore.sh
```

Finally, once the **backrest-delta-restored** deployment is running we can verify that the restore was successful by verifying that the table created prior to the restore no longer exists:

```
$ ${CCP_CLI} exec <backrest restored pod name> -- psql -c " table backrest_test_table"
ERROR:  relation "backrest_test_table" does not exist
LINE 1:  table backrest_test_table
               ^
command terminated with exit code 1
```

*As a reminder, please remember to run the cleanup script for the **Backup** example after running the cleanup script for this example.*

Async Archiving

pgBackRest supports the capability to asynchronously push and get write ahead logs (WAL) to and from a WAL archive. Asynchronous archiving can improve performance by parallelizing operations, while also reducing the number of connections to remote storage. For more information on async archiving and its benefits, please see the official pgBackRest documentation. This example will demonstrate how asynchronous archiving can be enabled within a crunchy-postgres or crunchy-postgres-gis container, while then also demonstrating the creation of a differential backup.

Start the example as follows:

```
cd $CCPROOT/examples/kube/backrest/async-archiving
./run.sh
```

This will create the following in your Kubernetes environment: - A deployment named **backrest-async-archive** containing a PostgreSQL database with pgBackRest configured - A service named **backrest-async-archive** for the PostgreSQL database - A PV and PVC for the PGDATA directory - A PV and PVC for the pgBackRest backups and archives directories

Additionally, the following variable will be set during deployment of the pod in order to enable asynchronous archiving:

```
PGBACKREST_ARCHIVE_ASYNC=y
```

This will also result in the creation of the required spool path, which we can see by listing the contents of the `/pgdata` directory in the backrest-async-archive deployment:

```
$ ${CCP_CLI} exec <backrest async archive pod name> -- ls /pgdata
backrest-async-archive
backrest-async-archive-backups
backrest-async-archive-spool
```

Once the database is up an running, a full backup can be taken:

```
${CCP_CLI} exec <backrest async archive pod name> -- pgbackrest backup \
  --stanza=db \
  --pg1-path=/pgdata/backrest-async-archive \
  --repo1-path=/backrestrepo/backrest-async-archive-backups \
  --log-path=/tmp \
  --type=full
```

And once a full backup has been taken, other types of backups can also be taken using pgBackRest, such as a differential backup:

```
${CCP_CLI} exec <backrest async archive pod name> -- pgbackrest backup \
  --stanza=db \
  --pg1-path=/pgdata/backrest-async-archive \
  --repo1-path=/backrestrepo/backrest-async-archive-backups \
  --log-path=/tmp \
  --type=diff
```

The following command can then be run to verify that both backups were created successfully:

```
${CCP_CLI} exec <backrest async archive pod name> -- pgbackrest info \
  --stanza=db \
  --repo1-path=/backrestrepo/backrest-async-archive-backups
```

Docker

Backup

In order to demonstrate the backup and restore capabilities provided by pgBackRest, it is first necessary to deploy a PostgreSQL database, and then create a full backup of that database. This example will therefore deploy a crunchy-postgres or crunchy-postgres-gis container containing a PostgreSQL database, which will then be backed up manually by executing a `pgbackrest backup` command. ***Please note that this example serves as a prequisite for the restore examples that follow, and therefore must be run prior to running those examples.***

Start the example as follows:

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

This will create the following in your Docker environment: - A container named **backrest** containing a PostgreSQL database with pgBackRest configured - A volume for the PGDATA directory - A volume for the pgBackRest backups and archives directories

Once the **backrest** container is running, use the `pgbackrest info` command to verify that pgbackrest has been properly configured and WAL archiving is working properly:

```
$ docker exec backrest pgbackrest info \
  --stanza=db \
  --repo1-path=/backrestrepo/backrest-backups

pg_pid=126
stanza: db
    status: error (no valid backups)
    cipher: none

    db (current)
        wal archive min/max (11-1): 000000010000000000000001 / 000000010000000000000003
```

An output similar to the above indicates that pgBackRest was properly configured upon deployment of the container, the **db** stanza has been created, and WAL archiving is working properly. The error next to **status** is expected being that a backup has not yet been generated.

Now that we have verified that pgBackRest is properly configured and enabled, a backup of the database can be generated. Being that this is the first backup of the database, we will take create a **full** backup:

```
$ docker exec backrest pgbackrest backup \
  --stanza=db \
  --pg1-path=/pgdata/backrest \
  --repo1-path=/backrestrepo/backrest-backups \
  --log-path=/tmp \
  --type=full

pg_pid=138
WARN: option repo1-retention-full is not set, the repository may run out of space
      HINT: to retain full backups indefinitely (without warning), set option
         'repo1-retention-full' to the maximum.
```

The warning displayed is expected, since backup retention has not been configured for this example. Assuming no errors are displayed, a full backup has now been successfully created.

Restore

pgBackRest provides numerous methods and strategies for restoring a PostgreSQL database. The following section will demonstrate three forms of database restores that can be accomplished when using pgBackRest with the Crunchy Container Suite: - **Full:** restore all database files into an empty PGDATA directory - **point-in-time Recovery (PITR):** restore a database to a specific point-in-time using an empty PGDATA directory - **Delta:** restore a database to a specific point-in-time using an existing PGDATA directory

Full This example will demonstrate a full database restore to an empty PGDATA directory. ***Please ensure the Backup example is currently running and a full backup has been generated prior to running this example.***

Prior to running the full restore, we will first make a change to the currently running database, which will we will then verify still exists following the restore. Create a simple table in the database as follows:

```
$ docker exec backrest psql -c "create table backrest_test_table (id int)"
CREATE TABLE
```

Now verify that the new table exists:

```
$ docker exec backrest psql -c "table backrest_test_table"
 id
----
(0 rows)
```

With the table in place, we can now start the full restore as follows:

```
cd $CCPROOT/examples/docker/backrest/full
./run.sh
```

This will create the following in your Docker environment: - A container named **backrest-full-restore** which will perform the restore using the crunchy-backrest-restore container - A volume for the new PGDATA directory that will contain the restored database. The directory will initially be empty, as required pgBackRest when performing a full restore, and will then contain the restored database upon completion of the restore.

Please note that a brand new PV and PVC are created when running the restore to clearly indicate that the database will be restored into an entirely new (i.e. empty) volume as required by pgBackRest. The names of the new PV and PVC are as follows: - **PV:** ${CCP_NAMESPACE}-br-new-pgdata - **PVC:** br-new-pgdata

You can verify that the restore has completed successfully by verifying that the container has finished running and has exited without errors:

```
docker ps -a
```

Once the container has finished running, the post restore script can then be run, which will create a new container named **backrest-full-restored** containing the restored database:

```
cd $CCPROOT/examples/docker/backrest/full
./post-restore.sh
```

Finally, once the **backrest-full-restored** container is running we can verify that the restore was successful by verifying that the table created prior to the restore still exists:

```
$ docker exec backrest-full-restored psql -c "table backrest_test_table"
 id
----
(0 rows)
```

Please note that the default behavior of pgBackRest is to recover to the end of the WAL archive stream, which is why the full restore contained all changes made since the initial full backup was taken, including the creation of table **backrest_test_table**. pgBackRest therefore played the entire WAL archive stream for all changes that occurred up until the restore.

*As a reminder, please remember to run the cleanup script for the **Backup** example after running the cleanup script for this example.*

PITR As demonstrated with the full restore above, the default behavior of pgBackRest is to recover to the end of the WAL archive stream. However, pgBackRest also provides the ability to recover to a specific point-in-time utilizing the WAL archives created since the last backup. This example will demonstrate how pgBackRest can be utilized to perform a point-in-time recovery (PITR) and therefore recover the database to specific point-in-time specified by the user. ***Please ensure that the Backup example is currently running and a full backup has been generated prior to running this example.***

Prior to running the PITR restore, we will first verify the current state of the database, after which we will then make a change to the database. This will allow us to verify that the PITR is successful by providing a method of verifying that the database has been restored to its current state following the restore.

To verify the current state of the database, we will first verify that a table called **backrest_test_table** does not exist in the database.

```
$ docker exec backrest psql -c "table backrest_test_table"
ERROR:  relation "backrest_test_table" does not exist
LINE 1:  table backrest_test_table
               ^
command terminated with exit code 1
```

Next, capture the current timestamp, which will be used later in the example when performing the restore:

```
$ docker exec backrest psql -c "select current_timestamp"
       current_timestamp
-------------------------------
 2019-10-27 16:53:05.590156+00
(1 row)
```

Now create table **backrest_test_table**:

```
$ docker exec backrest psql -c "create table backrest_test_table (id int)"
CREATE TABLE
```

Then verify that the new table exists:

```
$ docker exec backrest psql -c "table backrest_test_table"
 id
----
(0 rows)
```

With the table in place, we can now start the PITR. However, the timestamp captured above must also be provided in order to instruct pgBackRest to recover to that specific point-in-time. This is done using the `CCP_BACKREST_TIMESTAMP` variable, which allows us to then start the PITR as follows (replace the timestamp in the command below with the timestamp you captured above):

```
cd $CCPROOT/examples/docker/backrest/pitr
CCP_BACKREST_TIMESTAMP="2019-10-27 16:53:05.590156+00" ./run.sh
```

This will create the following in your Docker environment: - A container named **backrest-pitr-restore** which will perform the restore using the crunchy-backrest-restore container

Additionally, when this example is run, the following pgBackRest environment variables are provided to the crunchy-backrest-restore container in order to initiate PITR to the point-in-time specified by the timestamp (in additional to any other pgBackRest variables required by the Crunchy Container Suite and pgBackRest):

```
PGBACKREST_TYPE=time
PITR_TARGET="${CCP_BACKREST_TIMESTAMP}"
```

As can be seen above, the timestamp provided for `CCP_BACKREST_TIMESTAMP` is used to populate variable `PITR_TARGET`, and therefore specify the point-in-time to restore the database to, while `PGBACKREST_TYPE` is set to `time` to indicate that a PITR should be performed.

Please note that the following pgBackRest environment variable is also set when performing the PITR, which results in a restore to a new/empty directory within an existing PV:

```
PGBACKREST_PG1_PATH=/pgdata/backrest-pitr-restored
```

You can verify that the restore has completed successfully by verifying that the container has finished running and has exited without errors:

```
docker ps -a
```

Once the container has finished running, the post restore script can then be run, which will create a new container named **backrest-pitr-restored** containing the restored database:

```
cd $CCPROOT/examples/docker/backrest/pitr
./post-restore.sh
```

Finally, once the **backrest-pitr-restored** container is running we can verify that the restore was successful by verifying that the table created prior to the restore no longer exists:

```
$ docker exec backrest-pitr-restored psql -c "table backrest_test_table"
ERROR:  relation "backrest_test_table" does not exist
LINE 1:  table backrest_test_table
               ^
command terminated with exit code 1
```

*As a reminder, please remember to run the cleanup script for the **Backup** example after running the cleanup script for this example.*

Delta By default, pgBackRest requires a clean/empty directory in order to perform a restore. However, pgBackRest also provides an another option when performing the restore in the form of the **delta** option, which allows the restore to be run against an existing PGDATA directory. With the delta option enabled, pgBackRest will use checksums to determine which files in the directory can be preserved, and which need to be restored (please note that pgBackRest will also remove any files that are not present in the backup). This example will again demonstrate a point-in-time recovery (PITR), only this time the restore will occur within the existing PGDATA directory by specifying the **delta** option during the restore. ***Please ensure that the Backup example is currently running and a full backup has been generated prior to running this example.***

Prior to running the delta restore, we will first verify the current state of the database, and we will then make a change to the database. This will allow us to verify that the delta restore is successful by providing a method of verifying that the database has been restored to its current state following the restore.

To verify the current state of the database, we will first verify that a table called **backrest_test_table** does not exist in the database.

```
$ docker exec backrest psql -c "table backrest_test_table"
ERROR:  relation "backrest_test_table" does not exist
LINE 1:  table backrest_test_table
               ^
command terminated with exit code 1
```

Next, capture the current timestamp, which will be used later in the example when performing the restore:

```
$ docker exec backrest psql -c "select current_timestamp"
       current_timestamp
-------------------------------
 2019-10-27 16:53:05.590156+00
(1 row)
```

Now create table **backrest_test_table**:

```
$ docker exec backrest psql -c "create table backrest_test_table (id int)"
CREATE TABLE
```

Then verify that the new table exists:

```
$ docker exec backrest psql -c "table backrest_test_table"
 id
----
(0 rows)
```

With the table in place, we can now start the delta restore. When running the restore example the timestamp captured above must also be provided in order to instruct pgBackRest to recover to that specific point-in-time. This is done using the `CCP_BACKREST_TIMESTAMP` variable, which allows us to then start the delta restore as follows (replace the timestamp in the command below with the timestamp you captured above):

```
cd $CCPROOT/examples/docker/backrest/delta
CCP_BACKREST_TIMESTAMP="2019-10-27 16:53:05.590156+00" ./run.sh
```

This will create the following in your Docker environment: - A container named **backrest-delta-restore** which will perform the restore using the crunchy-backrest-restore container

Additionally, when this example is run, the following pgBackRest environment variables are provided to the crunchy-backrest-restore container in order to initiate a delta restore to the point-in-time specified by the timestamp (in additional to any other pgBackRest variables required by the Crunchy Container Suite and pgBackRest):

```
PGBACKREST_TYPE=time
PITR_TARGET="${CCP_BACKREST_TIMESTAMP}"
PGBACKREST_DELTA=y
```

As can be seen above, the timestamp provided for `CCP_BACKREST_TIMESTAMP` is used to populate variable `PITR_TARGET`, and therefore specify the point-in-time to restore to, while `PGBACKREST_TYPE` is set to `time` to indicate that a PITR should be performed. `PGBACKREST_DELTA` is set to `y` to indicate that the delta option should be utilized when performing the restore.

It's also worth noting that the following pgBackRest environment variable is also set when performing the delta restore, which results in a restore within the existing PGDATA directory utilized by the database deployed when running the **Backup** example:

```
PGBACKREST_PG1_PATH=/pgdata/backrest
```

You can verify that the restore has completed successfully by verifying that the container has finished running and has exited without errors:

```
docker ps -a
```

Once the container has finished running, the post restore script can then be run, which will create a new container named **backrest-delta-restored** containing the restored database:

```
cd $CCPROOT/examples/docker/backrest/delta
./post-restore.sh
```

Finally, once the **backrest-delta-restored** container is running we can verify that the restore was successful by verifying that the table created prior to the restore no longer exists:

```
$ docker exec backrest-delta-restored psql -c "table backrest_test_table"
ERROR:  relation "backrest_test_table" does not exist
LINE 1:  table backrest_test_table
               ^
command terminated with exit code 1
```

*As a reminder, please remember to run the cleanup script for the **Backup** example after running the cleanup script for this example.*

Async Archiving

pgBackRest supports the capability to asynchronously push and get write ahead logs (WAL) to and from a WAL archive. Asynchronous archiving can improve performance by parallelizing operations, while also reducing the number of connections to remote storage. For more information on async archiving and its benefits, please see the official pgBackRest documentation. This example will demonstrate how asynchronous archiving can be enabled within a crunchy-postgres or crunchy-postgres-gis container, while then also demonstrating the creation of a differential backup.

Start the example as follows:

```
cd $CCPROOT/examples/docker/backrest/async-archive
./run.sh
```

This will create the following in your Docker environment:

- A container named **backrest-async-archive** containing a PostgreSQL database with pgBackRest configured
- A volume for the PGDATA directory
- A volume for the pgBackRest backups and archives directories

Additionally, the following variable will be set during deployment of the container in order to enable asynchronous archiving:

```
PGBACKREST_ARCHIVE_ASYNC=y
```

This will also result in the creation of the required spool path, which we can see by listing the contents of the `/pgdata` directory in the backrest-async-archive container:

```
$ docker exec backrest-async-archive ls /pgdata
backrest-async-archive
backrest-async-archive-backups
backrest-async-archive-spool
```

Once the database is up an running, a full backup can be taken:

```
docker exec backrest-async-archive pgbackrest backup \
  --stanza=db \
  --pg1-path=/pgdata/backrest-async-archive \
  --repo1-path=/backrestrepo/backrest-async-archive-backups \
  --log-path=/tmp \
  --type=full
```

And once a full backup has been taken, other types of backups can also be taken using pgBackRest, such as a differential backup:

```
docker exec backrest-async-archive pgbackrest backup \
  --stanza=db \
  --pg1-path=/pgdata/backrest-async-archive \
  --repo1-path=/backrestrepo/backrest-async-archive-backups \
  --log-path=/tmp \
  --type=diff
```

The following command can then be run to verify that both backups were created successfully:

```
docker exec backrest-async-archive pgbackrest info \
  --stanza=db \
  --repo1-path=/backrestrepo/backrest-async-archive-backups
```

pgBaseBackup Examples

The script assumes you are going to backup the *primary* container created in the first example, so you need to ensure that container is running. This example assumes you have configured storage as described in the Storage Configuration documentation. Things to point out with this example include its use of persistent volumes and volume claims to store the backup data files.

A successful backup will perform `pg_basebackup` on the *primary* container and store the backup in the `$CCP_STORAGE_PATH` volume under a directory named `$CCP_NAMESPACE-primary-backups`. Each backup will be stored in a subdirectory with a timestamp as the name, allowing any number of backups to be kept.

The backup script will do the following:

- Start up a backup container or Kubernetes job named *backup*
- Run `pg_basebackup` on the container named *primary*
- Store the backup in the `/backup` volume
- Exit after the backup

When you are ready to restore from the backup, the restore example runs a `pgbasebackup-restore` container or Kubernetes job in order to restore the database into a new pgdata directory, specifically using rsync to copy the backup data into the new directory. A `crunchy-postgres` container then deployed using the restored database, deploying a new PostgreSQL DB that utilizes the original backed-up data.

The restore scripts will do the following:

- Start up a container or Kubernetes job named *restore*
- Copy the backup files from the previous backup example into `/pgdata`
- Deploy a `crunchy-postgres` container using the restored database

To shutdown and remove any artifacts from the example, run the following under each directory utilized when running the example:

```
./cleanup.sh
```

Docker

Perform a backup of *primary* using `pg_basebackup` by running the following script:

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

When you're ready to restore, a *restore* example is provided. In order to run the restore, a path for the backup must be provided to the `pgabasebackup-restore` container using the `BACKUP_PATH` environment variable. To get the correct path, check the logs for the `backup` container:

```
docker logs backup | grep BACKUP_PATH
Fri May 10 01:40:06 UTC 2019 INFO: BACKUP_PATH is set to
   /pgdata/primary-backups/2019-05-10-01-40-06.
```

`BACKUP_PATH` can also be discovered by looking at the backup mount directly (if access to the storage is available to the user).

When you are ready to restore from the backup, first update the `$CCPROOT/examples/docker/pgbasebackup/full/run.sh` script used to run restore example with the proper `BACKUP_PATH`:

```
--env BACKUP_PATH=primary-backups/2019-05-09-11-53-32 \
```

Then run the restore:

```
cd $CCPROOT/examples/docker/pgbasebackup/full
./run.sh
```

Once the restore is complete, the restored database can then be deployed by running the post restore script:

```
./post-restore.sh
```

You can then test the restored database as follows:

```
docker exec -it pgbasebackup-full-restored psql
```

Kubernetes and OpenShift

Perform a backup of *primary* using `pg_basebackup` by running the following script:

```
cd $CCPROOT/examples/kube/pgbasebackup/backup
./run.sh
```

This runs a Kubernetes job, which deploys a pod that performs the backup using `pg_basebackup`, and then exits. You can view the status of the job by running the following command:

```
${CCP_CLI} get job
```

When you're ready to restore, a *restore* example is provided. In order to run the restore, a path for the backup must be provided to the `pgabasebackup-restore` container using the `BACKUP_PATH` environment variable. To get the correct path, check the logs for the `backup` job:

```
kubectl logs backup-txcvm | grep BACKUP_PATH
Fri May 10 01:40:06 UTC 2019 INFO: BACKUP_PATH is set to
   /pgdata/primary-backups/2019-05-10-01-40-06.
```

`BACKUP_PATH` can also be discovered by looking at the backup mount directly (if access to the storage is available to the user).

When you are ready to restore from the backup, first update `$CCPROOT/examples/kube/pgbasebackup/full/restore.json` with the proper `BACKUP_PATH`:

```
{
    "name": "BACKUP_PATH",
    "value": "primary-backups/2019-05-08-18-28-45"
}
```

Then run the restore:

```
cd $CCPROOT/examples/kube/pgbasebackup/full
./run.sh
```

Once the restore is complete, the restored database can then be deployed by running the post restore script:

```
./post-restore.sh
```

You can then test the restored database as follows:

```
kubectl exec -it pgbasebackup-full-restored-7d866cd5f7-qr6w8 -- psql
```

pgDump & pgRestore Examples

The following examples will demonstrate how the crunchy-pgdump container can be utilized to create a database backup using the **pg_dump** utility, while also demonstrating how the backup created can then be utilized to restore the database using the **pg_restore** utility.

Backup (pg_dump)

The script assumes you are going to backup the *primary* example and that container is running.

This example assumes you have configured a storage filesystem as described in the Storage Configuration document.

A successful backup will perform pg_dump/pg_dumpall on the primary and store the resulting files in the mounted volume under a directory named `<HOSTNAME>-backups` as a sub-directory, then followed by a unique backup directory based upon a date and timestamp - allowing any number of backups to be kept.

For more information on how to configure this container, please see the Container Specifications document.

To shutdown the instance and remove the container for each example, run the following:

```
./cleanup.sh
```

Docker

Run the backup with this command:

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

Kubernetes and OpenShift

Running the example:

```
cd $CCPROOT/examples/kube/pgdump/backup
./run.sh
```

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

```
${CCP_CLI} get job
```

The `pgdump.json` file within that directory specifies options that control the behavior of the pgdump job. Examples of this include whether to run pg_dump vs pg_dumpall and advanced options for specific backup use cases.

Restore (pg_restore)

The script assumes that the pg_dump backup example above has been run. Therefore, the primary example should still be running and pg_dump backup should have been successfully created.

This example assumes you have configured a storage filesystem as described in the Storage Configuration document.

Successful use of the `crunchy-pgrestore` container will run a job to restore files generated by pg_dump/pg_dumpall to a container via psql/pg_restore; then container will terminate successfully and signal job completion.

For more information on how to configure this container, please see the Container Specifications document.

To shutdown the instance and remove the container for each example, run the following:

```
./cleanup.sh
```

Docker

Run the restore with this command:

```
cd $CCPROOT/examples/docker/pgdump/pgrestore
./run.sh
```

Kubernetes and OpenShift

By default, the crunchy-pgrestore container will automatically restore from the most recent backup. If you want to restore to a specific backup, edit the `pgrestore.json` file and update the `PGRESTORE_BACKUP_TIMESTAMP` setting to specify the backup path you want to restore with. For example:

```
"name":"PGRESTORE_BACKUP_TIMESTAMP",
"value":"2018-03-27-14-35-33"
```

Running the example:

```
cd $CCPROOT/examples/kube/pgdump/pgrestore
./run.sh
```

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

```
${CCP_CLI} get job
```

The `pgrestore.json` file within that directory specifies options that control the behavior of the pgrestore job.

title: “pgBouncer” date: draft: false weight: 51 —

pgBouncer Connection Pooling Example

Crunchy pgBouncer is a lightweight connection pooler for PostgreSQL databases.

The following examples create the following containers:

- pgBouncer Primary
- pgBouncer Replica
- PostgreSQL Primary
- PostgreSQL Replica

In Kubernetes and OpenShift, this example will also create:

- pgBouncer Primary Service
- pgBouncer Replica Service
- Primary Service
- Replica Service
- PostgreSQL Secrets
- pgBouncer Secrets

To cleanup the objects created by this example, run the following in the `pgbouncer` example directory:

```
./cleanup.sh
```

For more information on `pgBouncer`, see the official website.

This example uses a custom configuration to create the pgbouncer user and an auth function in the primary for the pgbouncer containers to authenticate against. It takes advantage of the post-startup-hook and a custom sql file mounted in the /pgconf directory.

Docker

Run the `pgbouncer` example:

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

Once all containers have deployed and are ready for use, `psql` to the target databases through `pgBouncer`:

```
psql -d userdb -h 0.0.0.0 -p 6432 -U testuser
psql -d userdb -h 0.0.0.0 -p 6433 -U testuser
```

To connect to the administration database within `pgbouncer`, connect using `psql`:

```
psql -d pgbouncer -h 0.0.0.0 -p 6432 -U pgbouncer
psql -d pgbouncer -h 0.0.0.0 -p 6433 -U pgbouncer
```

Kubernetes and OpenShift

OpenShift: If custom configurations aren't being mounted, an *emptydir* volume is required to be mounted at `/pgconf`.

Run the `pgbouncer` example:

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

Once all containers have deployed and are ready for use, `psql` to the target databases through `pgBouncer`:

```
psql -d userdb -h pgbouncer-primary -p 6432 -U testuser
psql -d userdb -h pgbouncer-replica -p 6432 -U testuser
```

To connect to the administration database within `pgbouncer`, connect using `psql`:

```
psql -d pgbouncer -h pgbouncer-primary -p 6432 -U pgbouncer -c "SHOW SERVERS"
psql -d pgbouncer -h pgbouncer-replica -p 6432 -U pgbouncer -c "SHOW SERVERS"
```

pgBadger Example

pgbadger is a PostgreSQL tool that reads the log files from a specified database in order to produce a HTML report that shows various PostgreSQL statistics and graphs. This example runs the pgbadger HTTP server against a crunchy-postgres container and illustrates how to view the generated reports.

The port utilized for this tool is port 14000 for Docker environments and port 10000 for Kubernetes and OpenShift environments.

The container creates a default database called *userdb*, a default user called *testuser* and a default password of *password*.

To shutdown the instance and remove the container for each example, run the following:

```
./cleanup.sh
```

Docker

Run the example as follows:

```
cd $CCPROOT/examples/docker/pgbadger
./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 -L http://127.0.0.1:10000/api/badgergenerate
```

Kubernetes and OpenShift

Running the example:

```
cd $CCPROOT/examples/kube/pgbadger
./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 -L http://pgbadger:10000/api/badgergenerate
```

You can view the database container logs using these commands:

```
${CCP_CLI} logs pgbadger -c pgbadger
${CCP_CLI} logs pgbadger -c postgres
```

pgBench Example

pgbench is a simple program for running benchmark tests on PostgreSQL. It runs the same sequence of SQL commands over and over, possibly in multiple concurrent database sessions, and then calculates the average transaction rate (transactions per second). By default, pgbench tests a scenario that is loosely based on TPC-B, involving five SELECT, UPDATE, and INSERT commands per transaction. However, it is easy to test other cases by writing your own transaction script files.

For more information on how to configure this container, please see the Container Specifications document.

Docker

This example requires the primary example to be running.

Run the example as follows:

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

After execution check the pgBench container logs for the benchmark results:

```
$ docker logs pgbench
dropping old tables...
creating tables...
generating data...
100000 of 100000 tuples (100%) done (elapsed 0.06 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done.
scale option ignored, using count from pgbench_branches table (1)
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 1
number of transactions actually processed: 1/1
latency average = 8.969 ms
tps = 111.498084 (including connections establishing)
tps = 211.041835 (excluding connections establishing)
```

Kubernetes and OpenShift

This example requires the primary example to be running.

Run the example as follows:

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

After execution check the pgBench container logs for the benchmark results:

```
$ ${CCP_CLI?} logs <name of pgbench pod>
dropping old tables...
creating tables...
generating data...
100000 of 100000 tuples (100%) done (elapsed 0.06 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done.
scale option ignored, using count from pgbench_branches table (1)
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 1
number of transactions actually processed: 1/1
latency average = 8.969 ms
tps = 111.498084 (including connections establishing)
tps = 211.041835 (excluding connections establishing)
```

pgBench Custom Transaction Example

The Crunch pgBench image supports mounting a custom transaction script (`transaction.sql`), which can be mounted to the `/pgconf` for auto-detection and configuration by the container.

This allows users to setup custom benchmarking scenarios for advanced use cases.

This example requires the primary example to be running.

Run the example as follows:

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

After execution check the pgBench container logs for the benchmark results:

```
$ docker logs pgbench-custom
dropping old tables...
creating tables...
generating data...
100000 of 100000 tuples (100%) done (elapsed 0.06 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done.
scale option ignored, using count from pgbench_branches table (1)
starting vacuum...end.
transaction type: /pgconf/transactions.sql
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 1
number of transactions actually processed: 1/1
latency average = 8.969 ms
tps = 111.498084 (including connections establishing)
tps = 211.041835 (excluding connections establishing)
```

Kubernetes and OpenShift

This example requires the primary example to be running.

Run the example as follows:

```
cd $CCPROOT/examples/kube/pgbench-custom
./run.sh
```

After execution check the pgBench container logs for the benchmark results:

```
dropping old tables...
creating tables...
generating data...
100000 of 100000 tuples (100%) done (elapsed 0.06 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done.
scale option ignored, using count from pgbench_branches table (1)
starting vacuum...end.
transaction type: /pgconf/transactions.sql
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 1
number of transactions actually processed: 1/1
latency average = 8.969 ms
tps = 111.498084 (including connections establishing)
tps = 211.041835 (excluding connections establishing)
```

Centralized Logging Example

The logs generated by containers are critical for deployments because they provide insights into the health of the system. PostgreSQL logs are very detailed and there is some information that can only be obtained from logs (but not limited to):

- Connections and Disconnections of users
- Checkpoint Statistics
- PostgreSQL Server Errors

Aggregrating container logs across multiple hosts allows administrators to audit, debug problems and prevent repudiation of misconduct.

In the following example we will demonstrate how to setup Kubernetes and OpenShift to use centralized logging by using an EFK (Elasticsearch, Fluentd and Kibana) stack. Fluentd will run as a daemonset on each host within the Kubernetes cluster and extract container logs, Elasticsearch will consume and index the logs gathered by Fluentd and Kibana will allow users to explore and visualize the logs via a web dashboard.

To learn more about the EFK stack, see the following:

- https://www.elastic.co/products/elasticsearch
- https://www.fluentd.org/architecture
- https://www.elastic.co/products/kibana

Configure PostgreSQL for Centralized Logging

By default, Crunchy PostgreSQL logs to files in the `/pgdata` directory. In order to get the logs out of the container we need to configure PostgreSQL to log to `stdout`.

The following settings should be configured in `postgresql.conf` to make PostgreSQL log to `stdout`:

```
log_destination = 'stderr'
logging_collector = off
```

Changes to logging settings require a restart of the PostgreSQL container to take effect.

Deploying the EFK Stack On OpenShift Container Platform

OpenShift Container Platform can be installed with an EFK stack. For more information about configuring OpenShift to create an EFK stack, see the official documentation:

- https://docs.openshift.com/container-platform/3.11/install_config/aggregate_logging.html

Deploying the EFK Stack On Kubernetes

First, deploy the EFK stack by running the example using the following commands:

```
cd $CCPROOT/examples/kube/centralized-logging/efk
./run.sh
```

Elasticsearch is configured to use an `emptyDir` volume in this example. Configure this example to provide a persistent volume when deploying into production.

Next, verify the pods are running in the `kube-system` namespace:

```
${CCP_CLI?} get pods -n kube-system --selector=k8s-app=elasticsearch-logging
${CCP_CLI?} get pods -n kube-system --selector=k8s-app=fluentd-es
${CCP_CLI?} get pods -n kube-system --selector=k8s-app=kibana-logging
```

If all pods deployed successfully, Elasticsearch should already be receiving container logs from Fluentd.

Next we will deploy a PostgreSQL Cluster (primary and replica deployments) to demonstrate PostgreSQL logs are being captured by Fluentd.

Deploy the PostgreSQL cluster by running the following:

```
cd $CCPROOT/examples/kube/centralized-logging/postgres-cluster
./run.sh
```

Next, verify the pods are running:

```
${CCP_CLI?} get pods --selector=k8s-app=postgres-cluster
```

With the PostgreSQL successfully deployed, we can now query the logs in Kibana.

We will need to setup a port-forward to the Kibana pod to access it. To do that we first get the name of the pod by running the following command:

```
${CCP_CLI?} get pod --selector=k8s-app=kibana-logging -n kube-system
```

Next, start the port-forward:

```
${CCP_CLI?} port-forward <KIBANA POD NAME> 5601:5601 -n kube-system
```

To access the web dashboard navigate in a browser to `127.0.0.1:5601`.

First, click the `Discover` tab and setup an index pattern to use for queries.

The index pattern name we will use is `logstash-*` because Fluentd is configured to generate logstash style logs.

Next we will configure the `Time Filter field name` to be `@timestamp`.

Now that our index pattern is created, we can query for the container logs.

Click the `Discover` tab and use the following queries:

```
# KUBERNETES
CONTAINER_NAME: *primary* AND MESSAGE: ".*LOG*"
# OpenShift
kubernetes.pod_name: "primary" AND log
```

For more information about querying Kibana, see the official documentation: https://www.elastic.co/guide/en/beats/packetbeat/current/kib queries-filters.html

To delete the centralized logging example run the following:

```
${CCP_ROOT?}/examples/kube/centralized-logging/efk/cleanup.sh
```

To delete the cluster roles required by the EFK stack, as an administrator, run the following:

```
${CCP_ROOT?}/examples/kube/centralized-logging/efk/cleanup-rbac.sh
```

pgAudit Enhanced Logging

This example provides an example of enabling pg_audit output. As of release 1.3, pg_audit is included in the crunchy-postgres container and is added to the PostgreSQL shared library list in `postgresql.conf`.

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

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

Docker

Run the following to create a database container:

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

This starts an instance of the pg_audit container (running crunchy-postgres) on port 12005 on localhost. The test script is then automatically executed.

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

Kubernetes and OpenShift

Run the following:

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

This script will create a PostgreSQL pod with the pgAudit extension configured and ready to use

Once the pod is deployed successfully run the following command to test the extension:

```
cd $CCPROOT/examples/kube/pgaudit
./test-pgaudit.sh
```

This example has been configured to log directly to stdout of the pod. To view the PostgreSQL logs, run the following:

```
$CCP_CLI logs pgaudit
```

pgAdmin4 example

This example deploys the pgadmin4 v2 web user interface for PostgreSQL without TLS.

After running the example, you should be able to browse to http://127.0.0.1:5050 and log into the web application with the following configured credentials:

- Username : *admin@admin.com*
- Password: *password*

If you are running this example using Kubernetes or OpenShift, it is required to use a port-forward proxy to access the dashboard.

To start the port-forward proxy run the following:

```
${CCP_CLI} port-forward pgadmin4-http 5050:5050
```

To access the pgAdmin4 dashboard through the proxy, navigate to *http://127.0.0.1:5050* in a browser.

See the pgAdmin4 documentation for more details.

To shutdown the instance and remove the container for each example, run the following:

```
./cleanup.sh
```

Docker

To run this example, run the following:

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

Kubernetes and OpenShift

Start the container as follows:

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

An emptyDir with write access must be mounted to the `/run/httpd` directory in OpenShift.

pgAdmin4 with TLS

This example deploys the pgadmin4 v2 web user interface for PostgreSQL with TLS.

After running the example, you should be able to browse to https://127.0.0.1:5050 and log into the web application with the following configured credentials:

- Username : *admin@admin.com*
- Password: *password*

If you are running this example using Kubernetes or OpenShift, it is required to use a port-forward proxy to access the dashboard.

To start the port-forward proxy run the following:

```
${CCP_CLI} port-forward pgadmin4-https 5050:5050
```

To access the pgAdmin4 dashboard through the proxy, navigate to *https://127.0.0.1:5050* in a browser.

See the pgadmin4 documentation for more details.

To shutdown the instance and remove the container for each example, run the following:

```
./cleanup.sh
```

Docker

To run this example, run the following:

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

Kubernetes and OpenShift

Start the container as follows:

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

An emptyDir with write access must be mounted to the `/run/httpd` directory in OpenShift.

Major Upgrade

This example assumes you have run *primary* using a PostgreSQL 12 or 13 image such as `ubi8-{{< param postgresVersion13 >}}-{{< param containersVersion >}}` prior to running this upgrade.

The upgrade container will let you perform a `pg_upgrade` from a PostgreSQL version 9.5, 9.6, 10, 11, 12, or 13 database to the available any of the higher versions of PostgreSQL versions that are currently support which are 9.6, 10, 11, 12, and 13. It does not do multi-version upgrades so you will need to for example do a 10 to 11 and then a 11 to 12 to get to version 12.

Prior to running this example, make sure your `CCP_IMAGE_TAG` environment variable is using the next major version of PostgreSQL that you want to upgrade to. For example, if you're upgrading from 12 to 13, make sure the variable references a PostgreSQL 13 image such as `ubi8-{{< param postgresVersion13 >}}-{{< param containersVersion >}}`.

This will create the following in your Kubernetes environment:

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

Data checksums on the Crunchy PostgreSQL container were enabled by default in version 2.1.0. When trying to upgrade, it's required that both the old database and the new database have the same data checksums setting. Prior to upgrade, check if `data_checksums` were enabled on the database by running the following SQL: `SHOW data_checksums`

Kubernetes and OpenShift

Before running the example, ensure you edit `upgrade.json` and update the `OLD_VERSION` and `NEW_VERSION` parameters to the major release version relevant to your situation.

First, delete the existing primary deployment:

```
${CCP_CLI} delete deployment primary
```

Then start the upgrade as follows:

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

If successful, the Job will end with a **successful** status. Verify the results of the Job by examining the Job's pod log:

```
${CCP_CLI} get pod -l job-name=upgrade
${CCP_CLI} logs -l job-name=upgrade
```

You can verify the upgraded database by running the `post-upgrade.sh` script in the `examples/kube/upgrade` directory. This will create a PostgreSQL pod that mounts the upgraded volume.

Overview

This document serves four purposes:

1. Ensure you have the prerequisites for building the images in Crunchy Container Suite
2. Make sure your local machine has all the pieces needed to run the examples in the GitHub repository
3. Run the images as standalone containers in Docker
4. Instruct you how to install the Crunchy Container Suite into Kubernetes or OpenShift

Where applicable, we will try to denote which installations and steps are required for the items above.

When we set up the directories below, you will notice they seem to be quite deeply nested. We are setting up a Go programming language workspace. Go has a specific folder structure for it's workspaces with multiple projects in a workspace. If you are **not** going build the container images you can ignore the deep directories below, but it will not hurt you if you follow the directions exactly.

Requirements

These instructions are developed and on for the following operating systems:

- **CentOS 7**
- **RHEL 7**

We also assume you are using the Docker provided with the distributions above. If you have installed Docker CE or EE on your machine, please create a VM for this work or uninstall Docker CE or EE.

The images in Crunchy Container Suite can run on different environments including:

- **Docker 1.13+**
- **OpenShift Container Platform 3.11**
- **Kubernetes 1.8+**

Initial Installs

Please note that *golang* is only required if you are building the containers from source. If you do not plan on building the containers then installing *git* is sufficient.

CentOS 7 only

```
$ sudo yum -y install epel-release
$ sudo yum -y install golang git
```

RHEL 7 only

```
$ sudo subscription-manager repos --enable="rhel-7-server-extras-rpms"
   --enable="rhel-7-server-optional-rpms"
$ sudo yum -y install epel-release
$ sudo yum -y install golang git
```

Clone GitHub repository

Make directories to hold the GitHub clone that also work with the Go workspace structure

```
$ mkdir -p $HOME/cdev/src/github.com/crunchydata $HOME/cdev/pkg $HOME/cdev/bin
$ cd $HOME/cdev/src/github.com/crunchydata
$ git clone https://github.com/crunchydata/crunchy-containers
$ cd crunchy-containers
$ git checkout v{{< param containersVersion >}}
```

Your Shell Environment

We have found, that because of the way Go handles different projects, you may want to create a separate account if are plan to build the containers and work on other Go projects. You could also look into some of the GOPATH wrappers.

If your goal is to simply run the containers, any properly configured user account should work.

Now we need to set the project paths and software version numbers. Edit your $HOME/.bashrc file with your favorite editor and add the following information. You can leave out the comments at the end of each line starting with #:

```
export GOPATH=$HOME/cdev        # set path to your new Go workspace
export GOBIN=$GOPATH/bin        # set bin path
export PATH=$PATH:$GOBIN        # add Go bin path to your overall path
export CCP_BASEOS=ubi8       # centos7 for CentOS, ubi8 for Red Hat Universal Base Image
export CCP_PGVERSION={{< param postgresVersionShort >}}        # The PostgreSQL major version
export CCP_PG_FULLVERSION={{< param postgresVersion >}}
export CCP_POSTGIS_VERSION=3.1  # The PostGIS version
export CCP_VERSION={{< param containersVersion >}}
export CCP_IMAGE_PREFIX=crunchydata # Prefix to put before all the container image names
export CCP_IMAGE_TAG=$CCP_BASEOS-$CCP_PG_FULLVERSION-$CCP_VERSION   # Used to tag the images
export CCP_POSTGIS_IMAGE_TAG=$CCP_BASEOS-$CCP_PG_FULLVERSION-$CCP_POSTGIS_VERSION-$CCP_VERSION #
   Used to tag images that include PostGIS
export CCPROOT=$GOPATH/src/github.com/crunchydata/crunchy-containers    # The base of the clone
   github repo
export CCP_SECURITY_CONTEXT=""
export CCP_CLI=kubectl          # kubectl for K8s, oc for OpenShift
export CCP_NAMESPACE=demo       # Change this to whatever namespace/openshift project name you
   want to use
```

It will be necessary to refresh your `.bashrc` file in order for the changes to take effect.

```
. ~/.bashrc
```

At this point we have almost all the prequesites required to build the Crunchy Container Suite.

Building UBI Containers With Supported Crunchy Enterprise Software

Before you can build supported containers on UBI and Crunchy Supported Software, you need to add the Crunchy repositories to your approved Yum repositories. Crunchy Enterprise Customer running on UBI can login and download the Crunchy repository key and yum repository from https://access.crunchydata.com/ on the downloads page. Once the files are downloaded please place them into the `$CCPROOT/conf` directory (defined above in the environment variable section).

Install Docker

The OpenShift and Kubernetes (KubeAdm) instructions both have a section for installing docker. Installing docker now won't cause any issues but you may wish to configure Docker storage before bringing everything up. Configuring Docker Storage is different from *Storage Configuration* referenced later in the instructions and is not covered here.

For a basic docker installation, you can follow the instructions below. Please refer to the respective installation guide for the version of Kubernetes you are installing for more specific details.

Install Docker

```
sudo yum -y install docker
```

It is necessary to add the `docker` group and give your user access to that group:

```
sudo groupadd docker
sudo usermod -a -G docker <username>
```

Logout and login again as the same user to allow group settings to take effect.

Enable Docker service and start Docker (once all configuration is complete):

```
sudo systemctl enable docker.service
sudo systemctl start docker.service
```

At this point you should be able to build the containers. Please to go to Building the Containers page and continue from there.

Install PostgreSQL

You only need to install PostgreSQL locally if you want to use the examples - it is not required for either building the containers or installing the containers into Kubernetes.

There are a variety of ways you can download PostgreSQL.

For specific installation instructions for installing PostgreSQL 12 on CentOS, please visit the Crunchy Data Developer Portal:

https://www.crunchydata.com/developers/download-postgres/binaries/postgresql12

Configuring Storage for Kubernetes Based Systems

In addition to the environment variables we set earlier, you will need to add environment variables for Kubernetes storage configuration. Please see the Storage Configuration document for configuring storage using environment variables set in `.bashrc`.

Don't forget to:

```
source ~/.bashrc
```

OpenShift Installation

Use the OpenShift installation guide to install OpenShift Enterprise on your host. Make sure to choose the proper version of OpenShift you want to install. The main instructions for 3.11 are here and you'll be able to select a different version there, if needed:

https://docs.openshift.com/container-platform/3.11/install/index.html

Kubernetes Installation

Make sure your hostname resolves to a single IP address in your /etc/hosts file. The NFS examples will not work otherwise and other problems with installation can occur unless you have a resolving hostname.

You should see a single IP address returned from this command:

```
$ hostname --ip-address
```

Installing Kubernetes

We suggest using Kubeadm as a simple way to install Kubernetes.

See Kubeadm for installing the latest version of Kubeadm.

See Create a Cluster for creating the Kubernetes cluster using **Kubeadm**. Note: We find that Weave networking works particularly well with the container suite.

Please see here to view the official documentation regarding configuring DNS for your Kubernetes cluster.

Post Kubernetes Configuration

In order to run the various examples, Role Based Account Control will need to be set up. Specifically, the **cluster-admin** role will need to be assigned to the Kubernetes user that will be utilized to run the examples. This is done by creating the proper **ClusterRoleBinding**:

```
$ kubectl create clusterrolebinding cluster-admin-binding \
--clusterrole cluster-admin --user someuser
```

If you are running on GKE, the following command can be utilized to auto-populate the **user** option with the account that is currently logged into Google Cloud:

```
$ kubectl create clusterrolebinding cluster-admin-binding \
--clusterrole cluster-admin --user $(gcloud config get-value account)
```

If more than one user will be running the examples on the same Kubernetes cluster, a unique name will need to be provided for each new **ClusterRoleBinding** created in order to assign the **cluster-admin** role to every user. The example below will create a **ClusterRoleBinding** with a unique value:

```
$ kubectl create clusterrolebinding <unique>-cluster-admin-binding \
    --clusterrole cluster-admin \
    --user someuser
```

If you are running on GKE, the following can be utilized to create a unique **ClusterRoleBinding** for each user, with the user's Google Cloud account prepended to the name of each new **ClusterRoleBinding**:

```
$ kubectl create clusterrolebinding "$(gcloud config get-value account)-cluster-admin-binding" \
    --clusterrole cluster-admin \
    --user $(gcloud config get-value account)
```

Helm

Some Kubernetes Helm examples are provided in the following directory as one option for deploying the Container Suite.

```
$CCPROOT/examples/helm/
```

Once you have your Kubernetes environment configured, it is simple to get Helm up and running. Please refer to this document to get Helm installed and configured properly.

Configuring Namespace and Permissions

In Kubernetes, a concept called a **namespace** provides the means to separate created resources or components into individual logically grouped partitions. In OpenShift, *namespace* is referred to as a *project*.

It is considered a best practice to have dedicated namespaces for projects in both testing and production environments.

All examples in the Crunchy Container Suite operate within the namespace defined by the environment variable `$CCP_NAMESPACE`. The default we use for namespace is 'demo' but it can be set to any valid namespace name. The instructions below illustrate how to set up and work within new namespaces or projects in both Kubernetes and OpenShift.

Kubernetes

This section will illustrate how to set up a new Kubernetes namespace called **demo**, and will then show how to provide permissions to that namespace to allow the Kubernetes examples to run within that namespace.

First, view currently existing namespaces:

```
$ kubectl get namespace
NAME          STATUS    AGE
default       Active    21d
kube-public   Active    21d
kube-system   Active    21d
```

Then, create a new namespace called **demo**:

```
$ kubectl create -f $CCPROOT/conf/demo-namespace.json
namespace "demo" created
$ kubectl get namespace demo
NAME      STATUS    AGE
demo      Active    7s
```

Then set the namespace as the default for the current context:

When a namespace is not explicitly stated for a command, Kubernetes uses the namespace specified by the currently set context.

```
$ kubectl config set-context $(kubectl config current-context) --namespace=demo
```

We can verify that the namespace was set correctly through the following command:

```
$ kubectl config view | grep namespace:
    namespace: demo
```

OpenShift

This section assumes an administrator has already logged in first as the **system:admin** user as directed by the OpenShift Installation Guide.

For our development purposes only, we typically specify the OCP Authorization policy of `AllowAll` as documented here:

https://docs.openshift.com/container-platform/3.11/install_config/configuring_authentication.html#AllowAllPasswordIdentityProvider

We do not recommend this authentication policy for a production deployment of OCP.

For the best results, it is recommended that you run the examples with a user that has **NOT** been assigned the **cluster-admin** cluster role.

Log into the system as a user:

```
$ oc login -u <user>
```

The next step is to create a **demo** namespace to run the examples within. The name of this OCP project will be what you supply in the CCP_NAMESPACE environment variable:

```
$ oc new-project demo --description="Crunchy Containers project"
   --display-name="Crunchy-Containers"
Now using project "demo" on server "https://127.0.0.1:8443".

$ export CCP_NAMESPACE=demo
```

If we view the list of projects, we can see the new project has been added and is "active".

```
$ oc get projects
NAME        DISPLAY NAME         STATUS
demo        Crunchy-Containers   Active
myproject   My Project           Active
```

If you were on a different project and wanted to switch to the demo project, you would do so by running the following:

```
$ oc project demo
Now using project "demo" on server "https://127.0.0.1:8443".
```

When self-provisioning a new project using the `oc new-project` command, the current user (i.e., the user you used when logging into OCP with the `oc login` command) will automatically be assigned to the **admin** role for that project. This will allow the user to create the majority of the objects needed to successfully run the examples. However, in order to create the **Persistent Volume** objects needed to properly configure storage for the examples, an additional role is needed. Specifically, a new role is needed that can both create and delete **Persistent Volumes**.

Using the following two commands, create a new Cluster Role that has the ability to create and delete persistent volumes, and then assign that role to your current user:

Please be aware that the following two commands require privileges that your current user may not have. In the event that you are unable to run these commands, and do not have access to a user that is able to run them (e.g., the **system:admin** user that is created by default when installing OCP), please contact your local OCP administrator to run the commands on your behalf, or grant you the access required to run them yourself.

```
$ oc create clusterrole crunchytester --verb="list,create,delete" --resource=persistentvolumes
clusterrole "crunchytester" created

$ oc adm policy add-cluster-role-to-user crunchytester someuser
cluster role "crunchytester" added: "someuser"
```

Your user should now have the roles and privileges required to run the examples.

Storage Configuration

Available Storage Types

The Crunchy Container Suite is officially tested using two different storage backends:

- HostPath (single node testing)
- NFS (single and multi-node testing)

Other storage backends work as well, including GCE, EBS, ScaleIO, and others, but may require you to modify various examples or configuration.

The Crunchy Container Suite is tested, developed, and examples are provided that use the various storage types listed above. This ensures that customers have a high degree of choices when it comes to choosing a volume type. HostPath and NFS allow precise host path choices for where database volumes are persisted. HostPath and NFS also allow governance models where volume creation is performed by an administrator instead of the application/developer team.

Where customers desire a dynamic form of volume creation (e.g. self service), storage classes are also supported within the example set.

Environment variables are set to determine how and what storage is to be used.

NOTE: *When running the examples using HostPath or NFS storage, the run scripts provided in the examples will create directories using the following pattern:*

```
$CCP_STORAGE_PATH/$CCP_NAMESPACE-<EXAMPLE_NAME>
```

HostPath

HostPath is the simplest storage backend to setup. It is only feasible on a single node but is sufficient for testing the examples. In your `.bashrc` file, add the following variables to specify the proper settings for your the HostPath storage volume:

```
export CCP_SECURITY_CONTEXT=""
export CCP_STORAGE_PATH=/data
export CCP_STORAGE_MODE=ReadWriteMany
export CCP_STORAGE_CAPACITY=400M
```

NOTE: *It may be necessary to grant your user in OpenShift or Kubernetes the rights to modify the* ***hostaccess*** *SCC. This can be done with the following command:*

```
oadm policy add-scc-to-user hostaccess $(oc whoami)
```

NFS

NFS can also be utilized as a storage mechanism. Instructions for setting up a NFS can be found in the **Configuration Notes for NFS** section below.

For testing with NFS, include the following variables in your **.bashrc** file, providing the proper configuration details for your NFS:

```
export CCP_SECURITY_CONTEXT='"supplementalGroups": [65534]'
export CCP_STORAGE_PATH=/nfsfileshare
export CCP_NFS_IP=<IP OF NFS SERVER>
export CCP_STORAGE_MODE=ReadWriteMany
export CCP_STORAGE_CAPACITY=400M
```

In the example above the group ownership of the NFS mount is assumed to be **nfsnobody** or **65534**. Additionally, it is recommended that root not be squashed on the NFS share (using `no_root_squash`) in order to ensure the proper directories can be created, modified and removed as needed for the various container examples.

Additionally, the examples in the Crunchy Container suite need access to the NFS in order to create the directories utilized by the examples. The NFS should therefore be mounted locally so that the `run.sh` scripts contained within the examples can complete the proper setup.

Configuration Notes for NFS

- Most of the Crunchy containers run as the postgres UID (26), but you will notice that when `supplementalGroups` is specified, the pod will include the `nfsnobody` group in the list of groups for the pod user
- If you are running your NFS system with SELinux in enforcing mode, you will need to run the following command to allow NFS write permissions:

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

- Detailed instructions for setting up a NFS server on Centos 7 can be found using the following link:

  http://www.itzgeek.com/how-tos/linux/centos-how-tos/how-to-setup-nfs-server-on-centos-7-rhel-7-fedora-22.html
- If you are running your client on a VM, you will need to add **insecure** to the exportfs file on the NFS server due to the way port translation is done between the VM host and the VM instance. For more details on this bug, please see the following link:

  http://serverfault.com/questions/107546/mount-nfs-access-denied-by-server-while-mounting

- A suggested best practice for tuning NFS for PostgreSQL is to configure the PostgreSQL fstab mount options like so:

  ```
  proto=tcp,suid,rw,vers=3,proto=tcp,timeo=600,retrans=2,hard,fg,rsize=8192,wsize=8192
  ```

  And to then change your network options as follows:

  ```
  MTU=9000
  ```

- If interested in mounting the same NFS share multiple times on the same mount point, look into the noac mount option

When using NFS storage and two or more clusters share the same NFS storage directory, the clusters need to be uniquely named.

Dynamic Storage

Dynamic storage classes can be used for the examples. There are various providers and solutions for dynamic storage, so please consult the Kubernetes documentation for additional details regarding supported storage choices. The environment variable `CCP_STORAGE_CLASS` is used in the examples to determine whether or not to create a PersistentVolume manually, or if it will be created dynamically using a StorageClass. In the case of GKE, the default StorageClass is named **default**. Storage class names are determined by the Kubernetes administrator and can vary.

Using block storage requires a security context to be set as follows:

```
export CCP_SECURITY_CONTEXT='"fsGroup":26'
export CCP_STORAGE_CLASS=standard
export CCP_STORAGE_MODE=ReadWriteOnce
export CCP_STORAGE_CAPACITY=400M
```

- Crunchy pgadmin4
- Crunchy pgbackrest
- Crunchy pgbackrest-repo
- Crunchy pgbadger
- Crunchy pgbouncer
- Crunchy pgpool
- Crunchy postgres-gis
- Crunchy postgres
- Crunchy upgrade

The crunchy-pgadmin4 container executes the pgAdmin4 web application.

pgAdmin4 provides a web user interface to PostgreSQL databases. A sample screenshot is below:

Features

The following features are supported by the crunchy-pgadmin4 container:

- Expose port (5050 by default) which is the web server port.
- Mount a certificate and key to the `/certs` directory and set `ENABLE_TLS` to true to activate HTTPS mode.
- Set username and password for login via environment variables.

Restrictions

- An emptyDir, with write access, must be mounted to the `/run/httpd` directory in OpenShift.

Packages

The crunchy-pgadmin4 Docker image contains the following packages (versions vary depending on PostgreSQL version):

- PostgreSQL ({{}} and {{}})
- pgAdmin4
- CentOS 7, UBI 8 - publicly available
- UBI 7, UBI 8 - customers only

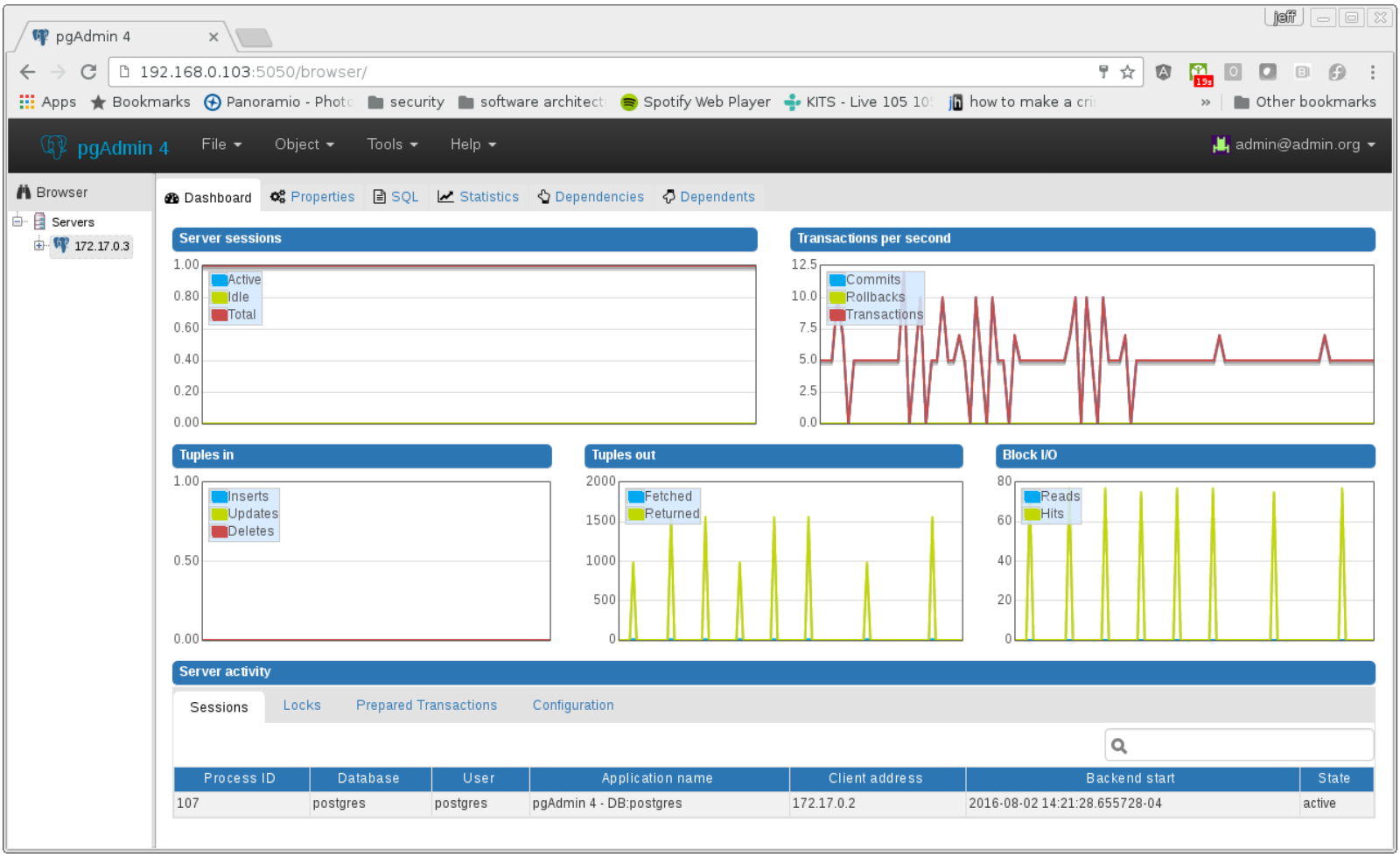

Figure 3: pgAdmin4

Environment Variables

Required

Name	Default	Description
PGADMIN_SETUP_EMAIL	None	Set this value to the email address used for pgAdmin4 login.
PGADMIN_SETUP_PASSWORD	None	Set this value to a password used for pgAdmin4 login. This should be a strong pass
SERVER_PORT	5050	Set this value to a change the port pgAdmin4 listens on.
SERVER_PATH	/	Set this value to customize the path of the URL that will be utilized to access the
ENABLE_TLS	FALSE	Set this value to true to enable HTTPS on the pgAdmin4 container. This requires

Optional

Name	Default	Description
CRUNCHY_DEBUG	FALSE	Set this to true to enable debugging in logs. Note: this mode can reveal secrets in logs.

The crunchy-pgbackrest-repo container acts as a pgBackRest remote repository for the Postgres cluster to use for storing archive files and backups.

See the pgBackRest guide for more details.

Volumes

The following volumes are mounted by the `crunchy-pgbackrest-repo` container:

The `crunchy-pgbackrest` container is used for pgBackRest functions including backup, restore, info, stanza creation and as the pgBackRest remote repository.

See the pgBackRest guide for more details.

Running Modes

The `crunchy-pgbackrest` image can be run in modes to enable different functionality. The `MODE` environment variable must be set to run the image in the required mode. Each mode uses environment variables to configure how the container will be run.

Running Mode	MOD
pgBackRest mode is used for taking pgBackRest backups, retrieving info and stanza creation.	pgb
pgBackRest Repo mode acts as a pgBackRest remote repository for the Postgres cluster to use for storing archive files and backups.	pgb
pgBackRest Restore mode executes a pgBackRest restore independent of the Crunchy PostgreSQL Operator.	pgb

Volumes

The following volumes are mounted by the `crunchy-pgbackrest` container:

- Mounted `pgbackrest.conf` configuration file via the `/pgconf` volume (? not sure about this)
- `/backrestrepo` volume used by the pgbackrest backup tool to store pgBackRest archives
- `/pgdata` volume used to store the data directory contents for the PostgreSQL database
- `/sshd` volume that contains the SSHD configuration from the `backrest-repo-config` secret

Major Packages

The crunchy-backrest-restore Docker image contains the following packages (versions vary depending on PostgreSQL version):

- PostgreSQL ({{}} and {{}})
- pgBackRest (2.33)
- CentOS 7, UBI 8 - publicly available
- UBI 7, UBI 8 - customers only

Environment Variables

pgbackrest Mode

Name	Default	Description
COMMAND	None	Stores the pgBackRest command to execute.
COMMAND_OPTS	None	Options to append the the chosen pgbackrest command.
CRUNCHY_DEBUG	FALSE	Set this to true to enable debugging in logs. Note: this mode c
MODE	None	Sets the container mode. Accepted values are `pgbackrest`, `pgb`
NAMESPACE	None	Namespace where the pod lives.
PGBACKREST_DB_PATH	None	PostgreSQL data directory. (deprecated)
PGBACKREST_REPO_PATH	None	Path where backups and archive are stored.
PGBACKREST_REPO_TYPE	None	Type of storage used for the repository.
PGBACKREST_STANZA	None	Defines the backup configuration for a specific PostgreSQL dat
PGHA_PGBACKREST_LOCAL_GCS_STORAGE	None	Indicates whether or not local and gcs storage should be enabl
PGHA_PGBACKREST_LOCAL_S3_STORAGE	None	Indicates whether or not local and s3 storage should be enable
PGHA_PGBACKREST_S3_VERIFY_TLS	None	Indicates whether or not TLS should be verified when making
PITR_TARGET	None	Store the PITR target for a pgBackRest restore.
PODNAME	None	Stores the name of the pod to exec into for command executio

pgbackrest-repo Mode

Name	Default	Description
CRUNCHY_DEBUG	FALSE	Set this to true to enable debugging in logs. Note: this mode can reveal secr
MODE	None	Sets the container mode. Accepted values are `pgbackrest`, `pgbackrest-repo`
PGBACKREST_DB_PATH	None	PostgreSQL data directory. (deprecated)
PGBACKREST_DB_HOST	None	PostgreSQL host for operating remotely via SSH. (deprecated)
PGBACKREST_LOG_PATH	None	Path where log files are stored.
PGBACKREST_PG1_PORT	None	Port that PostgreSQL is running on.
PGBACKREST_PG1_SOCKET_PATH	None	PostgreSQL unix socket path.
PGBACKREST_REPO_PATH	None	Path where backups and archive are stored.
PGBACKREST_STANZA	None	Defines the backup configuration for a specific PostgreSQL database cluster.

pgbackrest-restore Mode

Name	Default	Description
CRUNCHY_DEBUG	FALSE	Set this to true to enable debugging in logs. Note: this mode can reveal secrets in logs.
MODE	None	Sets the container mode. Accepted values are `pgbackrest`, `pgbackrest-repo` and `pgba`
BACKREST_CUSTOM_OPTS	None	Custom pgBackRest options can be added here to customize pgBackRest restores.
PGBACKREST_DELTA	None	Enables pgBackRest delta restore mode. Used when a user needs to restore to a volum
PGBACKREST_PG1_PATH	None	Path where PostgreSQL data directory can be found. This variable can also be used to
PGBACKREST_STANZA	None	Must be set to the desired stanza for restore.
PGBACKREST_TARGET	None	PostgreSQL timestamp used when restoring up to a point in time. Required for Point I

The crunchy-pgbadger container executes the pgBadger utility, which generates a PostgreSQL log analysis report using a small HTTP server running on the container. This log report can be accessed through the URL **http://<>:10000/api/badgergenerate**.

Features

The following features are supported by the crunchy-pgbadger container:

- Generate a full report by default
- Optional custom options for more advanced use cases (such as `incremental` reports)
- Report persistence on a volume

Packages

The crunchy-badger Docker image contains the following packages:

- pgBadger
- CentOS 7, UBI 8 - publicly available
- UBI 7, UBI 8 - customers only

Environment Variables

Optional

Name	Default	Description
BADGER_TARGET	None	Only used in standalone mode to specify the name of the container. Also used to find
BADGER_CUSTOM_OPTS	None	For a list of optional flags, see the official pgBadger documentation.
CRUNCHY_DEBUG	FALSE	Set this to true to enable debugging in logs. Note: this mode can reveal secrets in logs
PGBADGER_SERVICE_PORT	10000	Set the service port for the pgBadger process.

pgBouncer is a lightweight connection pooler for PostgreSQL databases.

Features

The following features are supported by the crunchy-pgbouncer container:

- crunchy-pgbouncer uses `auth_query` to authenticate users. This requires the `pgbouncer` username and password in `users.txt`. Automatically generated from environment variables, see Restrictions.
- Mount a custom `users.txt` and `pgbouncer.ini` configurations for advanced usage.
- Tune pooling parameters via environment variables.
- Connect to the administration database in pgBouncer to view statistics of the target databases.

Packages

The crunchy-pgbouncer Docker image contains the following packages (versions vary depending on PostgreSQL version):

- PostgreSQL ({{}} and {{}})
- pgBouncer
- CentOS 7, UBI 8 - publicly available
- UBI 7, UBI 8 - customers only

Restrictions

- OpenShift: If custom configurations aren't being mounted, an **emptydir** volume is required to be mounted at `/pgconf`.
- Superusers cannot connect through the connection pooler.
- User is required to configure the database for auth_query, see pgbouncer.ini file for configuration details.

Environment Variables

Required

Name	Default	Description
PGBOUNCER_PASSWORD	None	The password of the pgBouncer role in PostgreSQL. Must be also set on the primary data
PG_SERVICE	None	The hostname of the database service.

Optional

Name	Default	Description
DEFAULT_POOL_SIZE	20	How many server connections to allow per user/database pair.
MAX_CLIENT_CONN	100	Maximum number of client connections allowed.
MAX_DB_CONNECTIONS	Unlimited	Do not allow more than this many connections per-database.
MIN_POOL_SIZE	0	Adds more server connections to pool if below this number.
POOL_MODE	Session	When a server connection can be reused by other clients. Possible va
RESERVE_POOL_SIZE	0	How many additional connections to allow per pool. 0 disables.

Name	Default	Description
RESERVE_POOL_TIMEOUT	5	If a client has not been serviced in this many seconds, pgbouncer ena
QUERY_TIMEOUT	0	Queries running longer than that are canceled.
IGNORE_STARTUP_PARAMETERS	extra_float_digits	Set to ignore particular parameters in startup packets.
PG_PORT	5432	The port to use when connecting to the database.
CRUNCHY_DEBUG	FALSE	Set this to true to enable debugging in logs. Note: this mode can rev

The crunchy-pgpool container executes the pgPool II utility. pgPool can be used to provide a smart PostgreSQL-aware proxy to a PostgreSQL cluster, both primary and replica, so that applications only have to work with a single database connection.

PostgreSQL replicas are read-only whereas a primary is capable of receiving both read and write actions.

The default pgPool examples use a Secret to hold the set of pgPool configuration files used by the examples. The Secret is mounted into the `pgconf` volume mount where the container will look to find configuration files. If you do not specify your own configuration files via a Secret then you can specify environment variables to the container that it will attempt to use to configure pgPool, although this is not recommended for production environments. The pgpool container allows for local login with a properly configured password file.

Features

The following features are supported by the `crunchy-postgres` container:

- Basic invocation of pgPool II

Packages

The crunchy-pgpool Docker image contains the following packages (versions vary depending on PostgreSQL version):

- PostgreSQL ({{}} and {{}})
- pgPool II
- CentOS 7, UBI 8 - publicly available
- UBI 7, UBI 8 - customers only

Environment Variables

Required

Name	Default	Description
PG_USERNAME	None	Username for the PostgreSQL role being used.
PG_PASSWORD	None	Password for the PostgreSQL role being used.
PG_PRIMARY_SERVICE_NAME	None	Database host to connect to for the primary node.
PG_REPLICA_SERVICE_NAME	None	Database host to connect to for the replica node.

Optional

Name	Default	Description
CRUNCHY_DEBUG	FALSE	Set this to true to enable debugging in logs. Note: this mode can reveal secrets in logs.

PostgreSQL (pronounced “post-gress-Q-L”) is an open source, ACID compliant, relational database management system (RDBMS) developed by a worldwide team of volunteers. The crunchy-postgres-gis container image is unmodified, open source PostgreSQL packaged and maintained by professionals. This image is identical to the crunchy-postgres image except it includes the open source geospatial extension PostGIS for PostgreSQL.

For more information about configuration options for the `crunchy-postgres-gis` please reference the [`crunchy-postgres`]({{< relref "/container-specifications/crunchy-postgres" >}}) documentation. The `crunchy-postgres-gis` image is built using the `crunchy-postgres` image and supports the same features, packages, running modes, and volumes.

Features

In addition to features provided by the `crunchy-postgres` container, the following features are supported by the `crunchy-postgres-gis` container:

- PostGIS

Running Modes

The `crunchy-postgres-gis` Docker image can be run in modes to enable functionality. The `MODE` environment variable must be set to run the image in the correct mode. Each mode uses environment variables to configure how the container will be run. More information about these environment variables and custom configurations for each mode can be found in the following pages:

- Crunchy PostgreSQL({{< relref "/container-specifications/crunchy-postgres" >}}): `postgres`
- [Crunchy backup]({{< relref "/container-specifications/crunchy-postgres" >}}): `backup`
- [Crunchy pgbasebackup restore]({{< relref "/container-specifications/crunchy-postgres" >}}-restore): `pgbasebackup-restore`
- [Crunchy pgbench]({{< relref "/container-specifications/crunchy-postgres" >}}): `pgbench`
- [Crunchy pgdump]({{< relref "/container-specifications/crunchy-postgres" >}}): `pgdump`
- [Crunchy pgrestore]({{< relref "/container-specifications/crunchy-postgres" >}}): `pgrestore`
- [Crunchy sqlrunner]({{< relref "/container-specifications/crunchy-postgres/sqlrunner" >}}): `sqlrunner`

The crunchy-upgrade container contains multiple versions of PostgreSQL in order to perform a `pg_upgrade` between major versions of PostgreSQL. This includes the following combinations:

- PostgreSQL 9.5 / PostgreSQL 9.6
- PostgreSQL 9.6 / PostgreSQL 10
- PostgreSQL 10 / PostgreSQL 11
- PostgreSQL 11 / PostgreSQL 12
- PostgreSQL 12 / PostgreSQL 13

Features

The following features are supported by the crunchy-upgrade container:

- Supports a pg_upgrade of the PostgreSQL database.
- Doesn't alter the old database files.
- Creates the new database directory.

Restrictions

- Does **not** currently support a PostGIS upgrade.
- Supports upgrades from:
- 9.5 to 9.6
- 9.6 to 10
- 10 to 11
- 11 to 12
- 12 to 13

Packages

The crunchy-upgrade Docker image contains the following packages (versions vary depending on PostgreSQL version):

- PostgreSQL ({{}} and {{}})
- CentOS 7, UBI 8 - publicly available
- UBI 7, UBI 8 - customers only

Environment Variables

Required

Name	Default	Description
OLD_DATABASE_NAME	None	Refers to the database (pod) name that we want to convert.
NEW_DATABASE_NAME	None	Refers to the database (pod) name that is given to the upgraded database.
OLD_VERSION	None	The PostgreSQL version of the old database.
NEW_VERSION	None	The PostgreSQL version of the new database.

Optional

Name	Default	Description
PG_LOCALE	Default locale	If set, the locale you want to create the database with.
CHECKSUMS	true	Enables `data-checksums` during initialization of the database. Can only be set during initial
XLOGDIR	None	If set, initdb will use the specified directory for WAL.
CRUNCHY_DEBUG	FALSE	Set this to true to enable debugging in logs. Note: this mode can reveal secrets in logs.

Data checksums on the Crunchy PostgreSQL container were enabled by default in version 2.1.0. When trying to upgrade, it's required that both the old database and the new database have the same data checksums setting. Prior to upgrade, check if `data_checksums` were enabled on the database by running the following SQL: `SHOW data_checksums`

Build From Source

There are many cases where you may want to build the containers from source, such as working on a patch to contribute a feature. This guide provides the instructions to get you set up to build from source.

Requirements

- CentOS 7 or Red Hat 7 environment. The instructions below are set up for CentOS 7, but you can read the installation guide for additional instructions
- `go` version 1.13+
- `buildah` 1.14.9+
- `git`

Setup

1. On your system, be sure you have installed the requirements. You can do so with the following commands:

```
sudo yum -y install epel-release
sudo yum -y install golang git buildah
```

2. Create a fork of the Crunchy Container Suite repository and clone it to the environment using the following commands below:

```
mkdir -p $HOME/cdev/src/github.com/crunchydata $HOME/cdev/pkg $HOME/cdev/bin
cd $HOME/cdev/src/github.com/crunchydata
git clone https://github.com/crunchydata/crunchy-containers
cd crunchy-containers
```

3. Set up your local environmental variables to reference the containers you want to build. For instance, to build the latest version, you would use the following variables:

```
export GOPATH=$HOME/cdev        # set path to your new Go workspace
export GOBIN=$GOPATH/bin        # set bin path
export PATH=$PATH:$GOBIN        # add Go bin path to your overall path
export CCP_BASEOS=ubi8       # centos7 for CentOS, ubi8 for Red Hat Universal Base Image
export CCP_PGVERSION={{< param postgresVersionShort >}}       # The PostgreSQL major version
export CCP_PG_FULLVERSION={{< param postgresVersion >}}
export CCP_POSTGIS_VERSION=3.1
export CCP_VERSION={{< param containersVersion >}}
export CCP_IMAGE_PREFIX=crunchydata # Prefix to put before all the container image names
export CCP_IMAGE_TAG=$CCP_BASEOS-$CCP_PG_FULLVERSION-$CCP_VERSION   # Used to tag the images
export CCP_POSTGIS_IMAGE_TAG=$CCP_BASEOS-$CCP_PG_FULLVERSION-$CCP_POSTGIS_VERSION-$CCP_VERSION #
   Used to tag images that include PostGIS
export CCPROOT=$GOPATH/src/github.com/crunchydata/crunchy-containers    # The base of the clone
   github repo
```

You can save these variables to be set each time you open up your shell by adding them to your `.bashrc` file:

```
cat >> ~/.bashrc <<-EOF
export GOPATH=$HOME/cdev
export GOBIN=$GOPATH/bin
export PATH=$PATH:$GOBIN
export CCP_BASEOS=ubi8
export CCP_PGVERSION={{< param postgresVersionShort >}}
export CCP_PG_FULLVERSION={{< param postgresVersion >}}
export CCP_POSTGIS_VERSION=3.1
export CCP_VERSION={{< param containersVersion >}}
export CCP_IMAGE_PREFIX=crunchydata
export CCP_IMAGE_TAG=$CCP_BASEOS-$CCP_PG_FULLVERSION-$CCP_VERSION
export CCP_POSTGIS_IMAGE_TAG=$CCP_BASEOS-$CCP_PG_FULLVERSION-$CCP_POSTGIS_VERSION-$CCP_VERSION
export CCPROOT=$GOPATH/src/github.com/crunchydata/crunchy-containers
EOF
```

4. Download the GPG Keys and repository files that are required to pull in the packages used to build the containers:

```
cd $CCPROOT
curl
   https://api.developers.crunchydata.com/downloads/repo/rpm-centos/postgresql12/crunchypg12.repo
   > conf/crunchypg12.repo
curl
   https://api.developers.crunchydata.com/downloads/repo/rpm-centos/postgresql11/crunchypg11.repo
   > conf/crunchypg11.repo
curl https://api.developers.crunchydata.com/downloads/gpg/RPM-GPG-KEY-crunchydata-dev >
   conf/RPM-GPG-KEY-crunchydata-dev
```

5. Run the setup script to download the remaining dependencies

```
cd $CCPROOT
make setup
```

Build

You can now build the containers:

```
cd $CCPROOT
make all
```

if you want to build an individual container such as crunchy-postgres, you need to reference the individual name:

```
cd $CCPROOT
make postgres
```

To learn how to build each container, please review the Makefile targets within the Makefile in the repository.

If you would like to submit an feature / issue for us to consider please submit an issue to the official GitHub Repository.

If you would like to work any current or open issues, please update the issue with your efforts so that we can avoid redundant or unnecessary work.

If you have any questions, you can submit a Support - Question and Answer issue and we will work with you on how you can get more involved.

So you decided to submit an issue and work it. Great! Let's get it merged in to the codebase. The following will go a long way to helping get the fix merged in quicker:

1. Fork the Github repository and make a branch off of the `master` branch
2. Create a Pull Request from your Fork back to the `master` branch.
3. Update the checklists in the Pull Request description.
4. Reference which issues this Pull Request is resolving.

Kubernetes

Troubleshooting kubeadm

509 Certificate Errors

If you see `Unable to connect to the server: x509: certificate has expired or is not yet valid`, try resetting ntp. This generally indicates that the date/time is not set on local system correctly.

If you see `Unable to connect to the server: x509: certificate signed by unknown authority (possibly because of "crypto/rsa: verification error" while trying to verify candidate authority certificate "kubernetes")`, try running these commands as a regular user:

```
mv  $HOME/.kube $HOME/.kube.bak
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

gcloud Errors

If you see the error `ERROR: (gcloud.container.clusters.get-credentials)Unable to create private file [/etc/kubernetes/adm [Errno 1] Operation not permitted: '/etc/kubernetes/admin.conf'`, create a backup of admin.conf and delete the admin.conf before attempting to reconnect to the cluster.

gcloud Authentication Example

The commands used to authenticate to gcloud are the following:

```
gcloud auth login
gcloud config set project <your gcloud project>
gcloud auth configure-docker
```

If you see gcloud authentication errors, execute `gcloud config list` then re-authenticate using the commands from above. Finally, rerun `gcloud config list` - the results should show different values if authentication was indeed the issue.

OpenShift Container Platform

Troubleshooting OpenShift Container Platform: Basics