Crunchy Data Postgres Operator

Contents

Latest Release: 4.2.0

The *postgres-operator* is a controller that runs within a Kubernetes cluster that provides a means to deploy and manage PostgreSQL clusters.

Use the postgres-operator to:

- deploy PostgreSQL containers including streaming replication clusters
- scale up PostgreSQL clusters with extra replicas
- add pgpool, pgbouncer, and metrics sidecars to PostgreSQL clusters
- apply SQL policies to PostgreSQL clusters
- assign metadata tags to PostgreSQL clusters
- maintain PostgreSQL users and passwords
- perform minor upgrades to PostgreSQL clusters
- load simple CSV and JSON files into PostgreSQL clusters
- perform database backups

Deployment Requirements

The Operator is validated for deployment on Kubernetes, OpenShift, and VMware Enterprise PKS clusters. Some form of storage is required, NFS, HostPath, and Storage Classes are currently supported.

The Operator includes various components that get deployed to your Kubernetes cluster as shown in the following diagram and detailed in the Design.

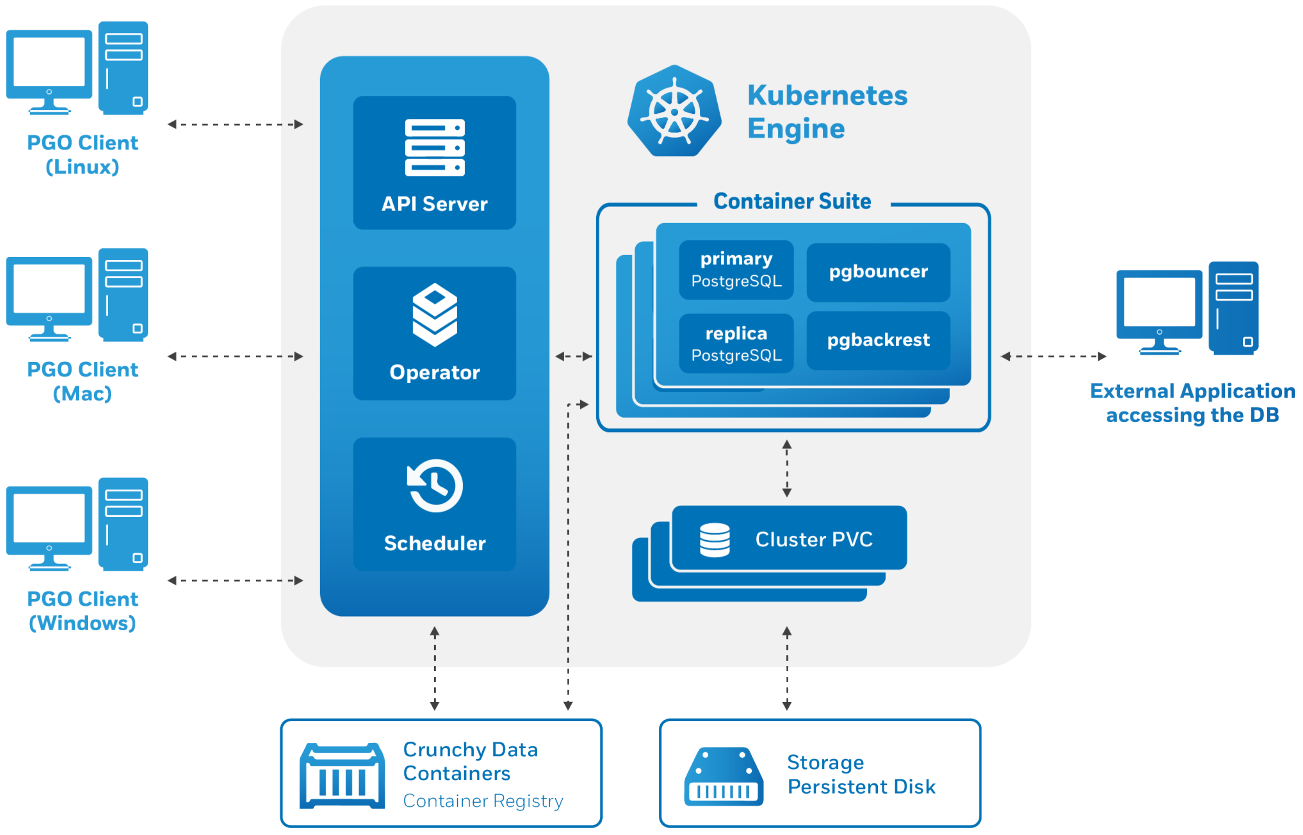

Figure 1: Architecture

The Operator is developed and tested on CentOS and RHEL Linux platforms but is known to run on other Linux variants.

Documentation

The following documentation is provided:

- pgo CLI Syntax and Examples
- Installation
- Configuration
- pgo.yaml Configuration
- Security
- Design Overview
- Developing
- Upgrading the Operator
- Contributing

If you are new to the Crunchy PostgreSQL Operator and interested in installing the Crunchy PostgreSQL Operator in your environment, please start here: - Installation via Bash - Installation via Ansible

If you have the Crunchy PostgreSQL Operator installed in your environment, and are interested in installation of the client interface, please start here: - PGO Client Install

If you have the Crunchy PostgreSQL and Client Interface installed in your environment and are interested in guidance on the use of the Crunchy PostgreSQL Operator, please start here: - PGO CLI Overview — title: “Design” date: draft: false weight: 4 —

Provisioning

So, what does the Postgres Operator actually deploy when you create a cluster?

On this diagram, objects with dashed lines are components that are optionally deployed as part of a PostgreSQL Cluster by the operator. Objects with solid lines are the fundamental and required components.

For example, within the Primary Deployment, the *metrics* container is completely optional. That component can be deployed using either the operator configuration or command line arguments if you want to cause metrics to be collected from the Postgres container.

Replica deployments are similar to the primary deployment but are optional. A replica is not required to be created unless the capability for one is necessary. As you scale up the Postgres cluster, the standard set of components gets deployed and replication to the primary is started.

Notice that each cluster deployment gets its own unique Persistent Volumes. Each volume can use different storage configurations which provides fined grained placement of the database data files.

There is a Service created for the primary Postgres deployment and a Service created for any replica Postgres deployments within a given Postgres cluster. Primary services match Postgres deployments using a label *service-name* of the following format:

```
service-name=mycluster
service-name=mycluster-replica
```

The *postgres-operator* design incorporates the following concepts:

Custom Resource Definitions

Kubernetes Custom Resource Definitions are used in the design of the PostgreSQL Operator to define the following:

- Cluster - *pgclusters*
- Backup - *pgbackups*
- Policy - *pgpolicies*
- Tasks - *pgtasks*

Metadata about the PostgreSQL cluster deployments are stored within these CRD resources which act as the source of truth for the PostgreSQL Operator. The Operator makes use of custom resource definitions to maintain state and resource definitions as offered by the Operator.

In this above diagram, the CRDs heavily used by the Operator include:

- pgcluster - defines the Postgres cluster definition, new cluster requests are captured in a unique pgcluster resource for that Postgres cluster
- pgtask - workflow and other related administration tasks are captured within a set of pgtasks for a given pgcluster
- pgbackup - when you run a pgbasebackup, a pgbackup is created to hold the workflow and status of the last backup job, this CRD will eventually be deprecated in favor of a more general pgtask resource
- pgreplica - when you create a Postgres replica, a pgreplica CRD is created to define that replica

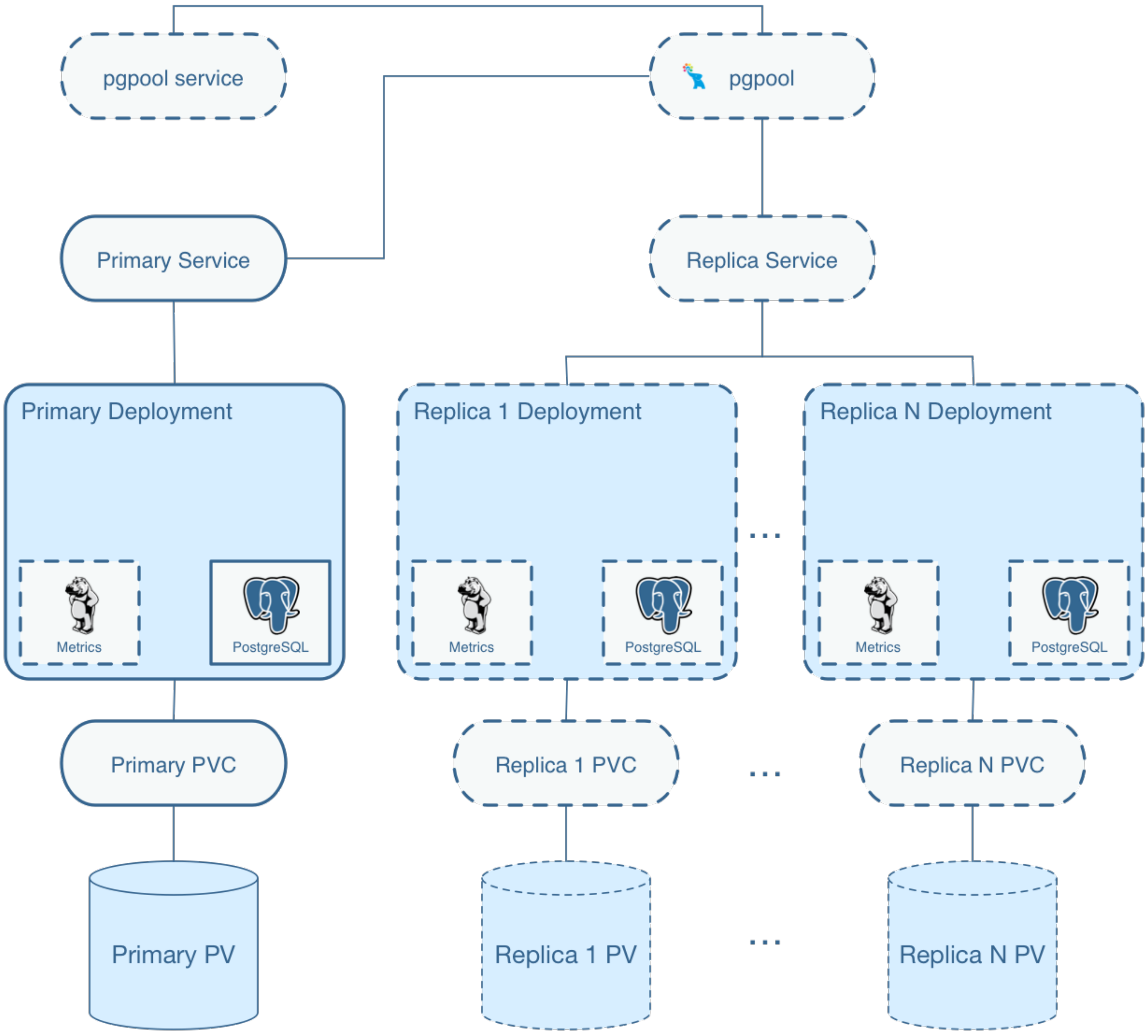

Figure 2: Reference

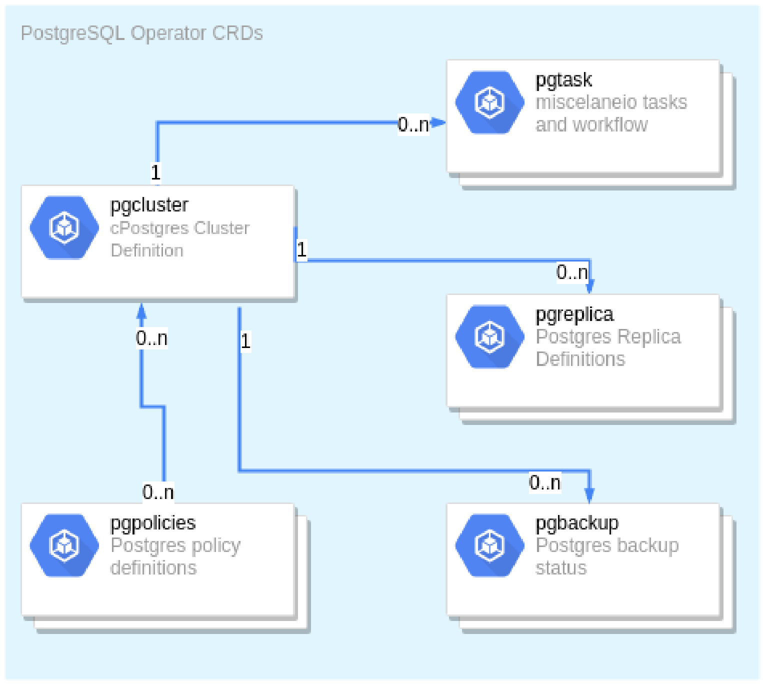
PostgreSQL Operator CRDs
pgtask
miscelaneio tasks and workflow
pgcluster
cPostgres Cluster Definition
pgreplica
Postgres Replica Definitions
pgpolicies
Postgres policy definitions
pgbackup
Postgres backup status
1
0..n
1
0..n
0..n
0..n
1
0..n

Figure 3: Reference

Security Context Constraints

In an Openshift installation of the PostgreSQL Operator, a custom Security Context Constraint (SCC) is used to customize the control permissions for the *pgcluster* pods. The PGO SCC adds file system group (fsgroup) 26 to the pod's existing security context and will be used in place of Openshift's default *restricted* SCC.

Event Listeners

Kubernetes events are created for the Operator CRD resources when new resources are made, deleted, or updated. These events are processed by the Operator to perform asynchronous actions.

As events are captured, controller logic is executed within the Operator to perform the bulk of operator logic.

REST API

A feature of the Operator is to provide a REST API upon which users or custom applications can inspect and cause actions within the Operator such as provisioning resources or viewing status of resources.

This API is secured by a RBAC (role based access control) security model whereby each API call has a permission assigned to it. API roles are defined to provide granular authorization to Operator services.

Command Line Interface

One of the unique features of the Operator is the pgo command line interface (CLI). This tool is used by a normal end-user to create databases or clusters, or make changes to existing databases.

The CLI interacts with the REST API deployed within the *postgres-operator* deployment.

Node Affinity

You can have the Operator add a node affinity section to a new Cluster Deployment if you want to cause Kubernetes to attempt to schedule a primary cluster to a specific Kubernetes node.

You can see the nodes on your Kube cluster by running the following:

```
kubectl get nodes
```

You can then specify one of those names (e.g. kubeadm-node2) when creating a cluster;

```
pgo create cluster thatcluster --node-label=kubeadm-node2
```

The affinity rule inserted in the Deployment uses a *preferred* strategy so that if the node were down or not available, Kubernetes will go ahead and schedule the Pod on another node.

When you scale up a Cluster and add a replica, the scaling will take into account the use of `--node-label`. If it sees that a cluster was created with a specific node name, then the replica Deployment will add an affinity rule to attempt to schedule

Fail-over

Manual and automated fail-over are supported in the Operator within a single Kubernetes cluster.

Manual failover is performed by API actions involving a *query* and then a *target* being specified to pick the fail-over replica target.

Automatic fail-over is performed by the Operator by evaluating the readiness of a primary. Automated fail-over can be globally specified for all clusters or specific clusters.

Users can configure the Operator to replace a failed primary with a new replica if they want that behavior.

The fail-over logic includes:

- deletion of the failed primary Deployment
- pick the best replica to become the new primary
- label change of the targeted Replica to match the primary Service
- execute the PostgreSQL promote command on the targeted replica

pgbackrest Integration

The Operator integrates various features of the pgbackrest backup and restore project. A key component added to the Operator is the *pgo-backrest-repo* container, this container acts as a pgBackRest remote repository for the Postgres cluster to use for storing archive files and backups.

The following diagrams depicts some of the integration features:

In this diagram, starting from left to right we see the following:

- a user when they enter *pgo backup mycluster –backup-type=pgbackrest* will cause a pgo-backrest container to be run as a Job, that container will execute a *pgbackrest backup* command in the pgBackRest repository container to perform the backup function.

- a user when they enter *pgo show backup mycluster –backup-type=pgbackrest* will cause a *pgbackrest info* command to be executed on the pgBackRest repository container, the *info* output is sent directly back to the user to view

- the Postgres container itself will use an archive command, *pgbackrest archive-push* to send archives to the pgBackRest repository container

- the user entering *pgo create cluster mycluster –pgbackrest* will cause a pgBackRest repository container deployment to be created, that repository is exclusively used for this Postgres cluster

- lastly, a user entering *pgo restore mycluster* will cause a *pgo-backrest-restore* container to be created as a Job, that container executes the *pgbackrest restore* command

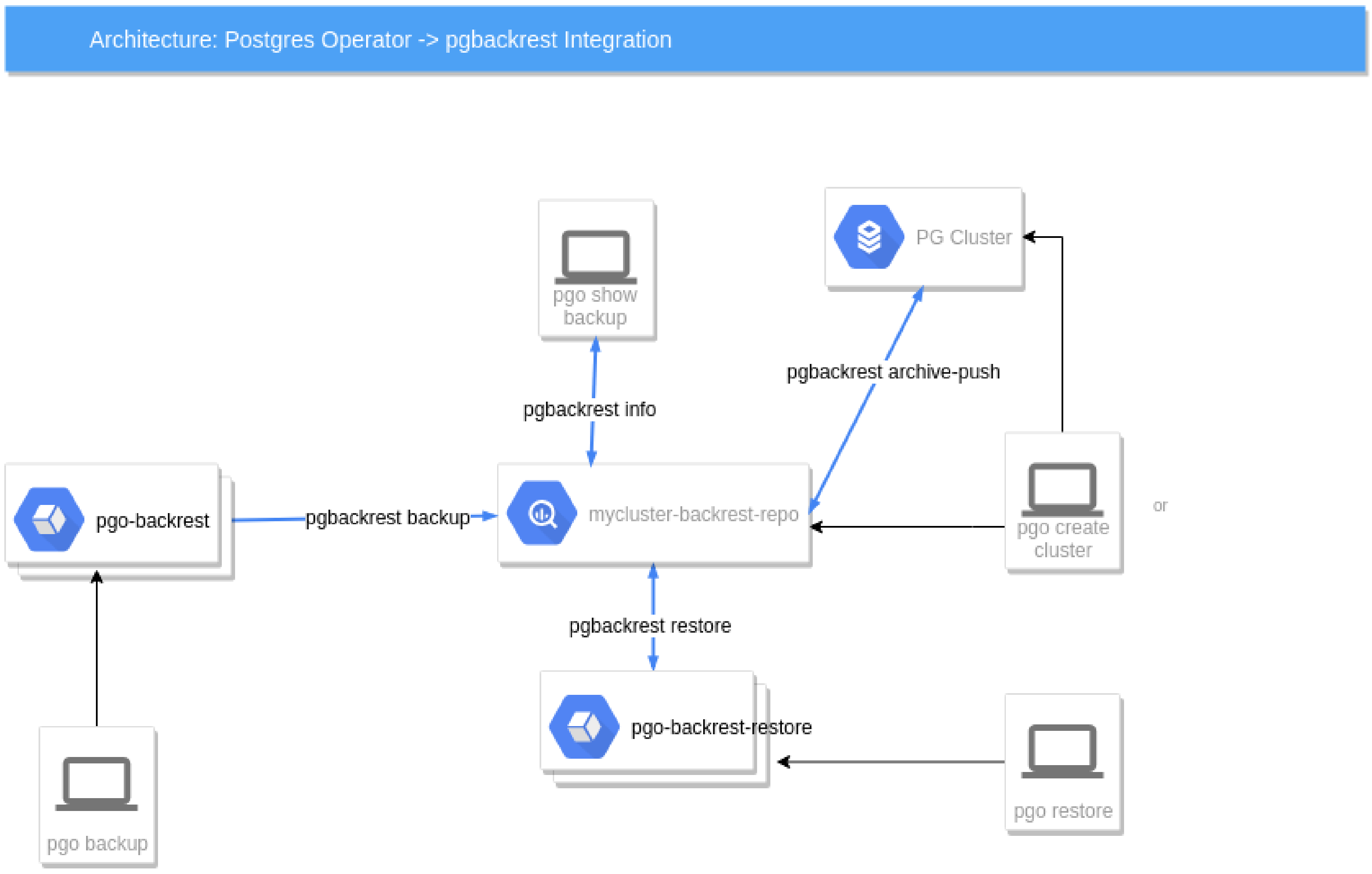

Figure 4: alt text

pgbackrest Restore

The pgbackrest restore command is implemented as the *pgo restore* command. This command is destructive in the sense that it is meant to *restore* a PG cluster meaning it will revert the PG cluster to a restore point that is kept in the pgbackrest repository. The prior primary data is not deleted but left in a PVC to be manually cleaned up by a DBA. The restored PG cluster will work against a new PVC created from the restore workflow.

When doing a *pgo restore*, here is the workflow the Operator executes:

- turn off autofail if it is enabled for this PG cluster
- allocate a new PVC to hold the restored PG data
- delete the the current primary database deployment
- update the pgbackrest repo for this PG cluster with a new data path of the new PVC
- create a pgo-backrest-restore job, this job executes the *pgbackrest restore* command from the pgo-backrest-restore container, this Job mounts the newly created PVC
- once the restore job completes, a new primary Deployment is created which mounts the restored PVC volume

At this point the PG database is back in a working state. DBAs are still responsible to re-enable autofail using *pgo update cluster* and also perform a pgBackRest backup after the new primary is ready. This version of the Operator also does not handle any errors in the PG replicas after a restore, that is left for the DBA to handle.

Other things to take into account before you do a restore:

- if a schedule has been created for this PG cluster, delete that schedule prior to performing a restore
- If your database has been paused after the target restore was completed, then you would need to run the psql command select pg_wal_replay_resume() to complete the recovery, on PG 9.6/9.5 systems, the command you will use is select pg_xlog_replay_resume(). You can confirm the status of your database by using the built in postgres admin functions found [here:] (https://www.postgresql.org/docs/current/functions-admin.html#FUNCTIONS-RECOVERY-CONTROL-TABLE)
- a pgBackRest restore is destructive in the sense that it deletes the existing primary deployment for the cluster prior to creating a new deployment containing the restored primary database. However, in the event that the pgBackRest restore job fails, the `pgo restore` command be can be run again, and instead of first deleting the primary deployment (since one no longer exists), a new primary will simply be created according to any options specified. Additionally, even though the original primary deployment will be deleted, the original primary PVC will remain.
- there is currently no Operator validation of user entered pgBackRest command options, you will need to make sure to enter these correctly, if not the pgBackRest restore command can fail.
- the restore workflow does not perform a backup after the restore nor does it verify that any replicas are in a working status after the restore, it is possible you might have to take actions on the replica to get them back to replicating with the new restored primary.
- pgbackrest.org suggests running a pgbackrest backup after a restore, this needs to be done by the DBA as part of a restore
- when performing a pgBackRest restore, the **node-label** flag can be utilized to target a specific node for both the pgBackRest restore job and the new (i.e. restored) primary deployment that is then created for the cluster. If a node label is not specified, the restore job will not target any specific node, and the restored primary deployment will inherit any node labels defined for the original primary deployment.

PGO Scheduler

The Operator includes a cronlike scheduler application called `pgo-scheduler`. Its purpose is to run automated tasks such as PostgreSQL backups or SQL policies against PostgreSQL clusters created by the Operator.

PGO Scheduler watches Kubernetes for configmaps with the label `crunchy-scheduler=true` in the same namespace the Operator is deployed. The configmaps are json objects that describe the schedule such as:

- Cron like schedule such as: * * * * *
- Type of task: `pgbackrest`, `pgbasebackup` or `policy`

Schedules are removed automatically when the configmaps are deleted.

PGO Scheduler uses the `UTC` timezone for all schedules.

Schedule Expression Format

Schedules are expressed using the following rules:

```
Field name   | Mandatory? | Allowed values  | Allowed special characters
----------   | ---------- | --------------  | --------------------------
Seconds      | Yes        | 0-59            | * / , -
Minutes      | Yes        | 0-59            | * / , -
Hours        | Yes        | 0-23            | * / , -
Day of month | Yes        | 1-31            | * / , - ?
Month        | Yes        | 1-12 or JAN-DEC | * / , -
Day of week  | Yes        | 0-6 or SUN-SAT  | * / , - ?
```

pgBackRest Schedules

pgBackRest schedules require pgBackRest enabled on the cluster to backup. The scheduler will not do this on its own.

pgBaseBackup Schedules

pgBaseBackup schedules require a backup PVC to already be created. The operator will make this PVC using the backup commands:

```
pgo backup mycluster
```

Policy Schedules

Policy schedules require a SQL policy already created using the Operator. Additionally users can supply both the database in which the policy should run and a secret that contains the username and password of the PostgreSQL role that will run the SQL. If no user is specified the scheduler will default to the replication user provided during cluster creation.

Custom Postgres Configurations

Users and administrators can specify a custom set of Postgres configuration files be used when creating a new Postgres cluster. The configuration files you can change include -

- postgresql.conf
- pg_hba.conf
- setup.sql

Different configurations for PostgreSQL might be defined for the following -

- OLTP types of databases
- OLAP types of databases
- High Memory
- Minimal Configuration for Development
- Project Specific configurations
- Special Security Requirements

Global ConfigMap If you create a *configMap* called *pgo-custom-pg-config* with any of the above files within it, new clusters will use those configuration files when setting up a new database instance. You do *NOT* have to specify all of the configuration files. It is entirely up to your use case to determine which to use.

An example set of configuration files and a script to create the global configMap is found at

```
$PGOROOT/examples/custom-config
```

If you run the *create.sh* script there, it will create the configMap that will include the PostgreSQL configuration files within that directory.

Config Files Purpose The *postgresql.conf* file is the main Postgresql configuration file that allows the definition of a wide variety of tuning parameters and features.

The *pg_hba.conf* file is the way Postgresql secures client access.

The *setup.sql* file is a Crunchy Container Suite configuration file used to initially populate the database after the initial *initdb* is run when the database is first created. Changes would be made to this if you wanted to define which database objects are created by default.

Granular Config Maps Granular config maps can be defined if it is necessary to use a different set of configuration files for different clusters rather than having a single configuration (e.g. Global Config Map). A specific set of ConfigMaps with their own set of PostgreSQL configuration files can be created. When creating new clusters, a `--custom-config` flag can be passed along with the name of the ConfigMap which will be used for that specific cluster or set of clusters.

Defaults If there is no reason to change the default PostgreSQL configuration files that ship with the Crunchy Postgres container, there is no requirement to. In this event, continue using the Operator as usual and avoid defining a global configMap.

pgbackrest Configuration

The PostgreSQL Operator integrates various features of the pgbackrest backup and restore project.

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

The following diagrams depicts some of the integration features:

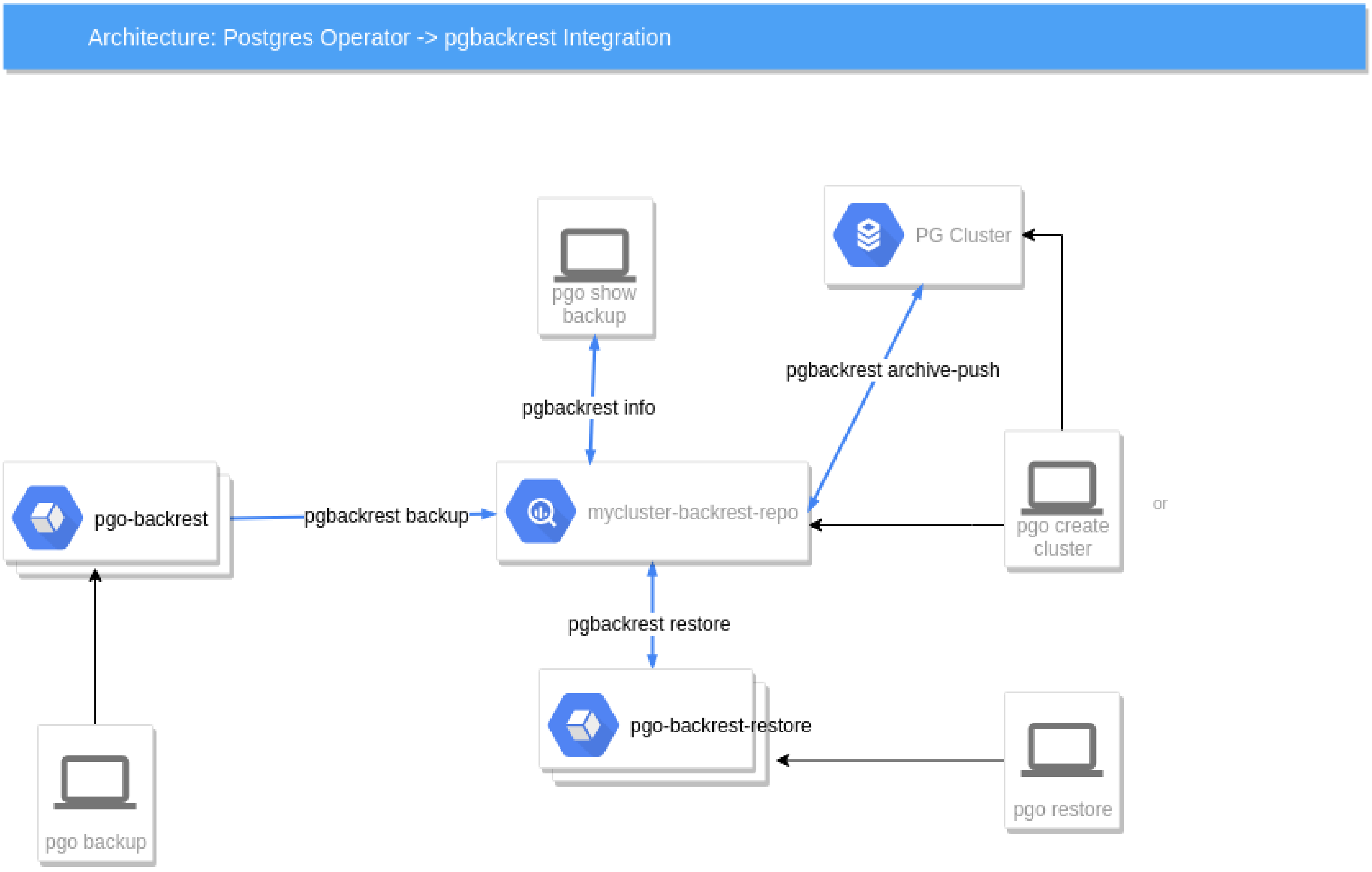

Figure 5: alt text

In this diagram, starting from left to right we see the following:

- a user when they enter *pgo backup mycluster –backup-type=pgbackrest* will cause a pgo-backrest container to be run as a Job, that container will execute a *pgbackrest backup* command in the pgBackRest repository container to perform the backup function.
- a user when they enter *pgo show backup mycluster –backup-type=pgbackrest* will cause a *pgbackrest info* command to be executed on the pgBackRest repository container, the *info* output is sent directly back to the user to view
- the PostgreSQL container itself will use an archive command, *pgbackrest archive-push* to send archives to the pgBackRest repository container
- the user entering *pgo create cluster mycluster –pgbackrest* will cause a pgBackRest repository container deployment to be created, that repository is exclusively used for this Postgres cluster
- lastly, a user entering *pgo restore mycluster* will cause a *pgo-backrest-restore* container to be created as a Job, that container executes the *pgbackrest restore* command

pgbackrest Restore

The pgbackrest restore command is implemented as the *pgo restore* command. This command is destructive in the sense that it is meant to *restore* a PG cluster meaning it will revert the PG cluster to a restore point that is kept in the pgbackrest repository. The prior primary data is not deleted but left in a PVC to be manually cleaned up by a DBA. The restored PG cluster will work against a new PVC created from the restore workflow.

When doing a *pgo restore*, here is the workflow the PostgreSQL Operator executes:

- turn off autofail if it is enabled for this PG cluster
- allocate a new PVC to hold the restored PG data
- delete the the current primary database deployment
- update the pgbackrest repo for this PG cluster with a new data path of the new PVC
- create a pgo-backrest-restore job, this job executes the *pgbackrest restore* command from the pgo-backrest-restore container, this Job mounts the newly created PVC
- once the restore job completes, a new primary Deployment is created which mounts the restored PVC volume

At this point the PostgreSQL database is back in a working state. DBAs are still responsible to re-enable autofail using *pgo update cluster* and also perform a pgBackRest backup after the new primary is ready. This version of the PostgreSQL Operator also does not handle any errors in the PG replicas after a restore, that is left for the DBA to handle.

Other things to take into account before you do a restore:

- if a schedule has been created for this PostgreSQL cluster, delete that schedule prior to performing a restore
- If your database has been paused after the target restore was completed, then you would need to run the psql command select pg_wal_replay_resume() to complete the recovery, on PostgreSQL 9.6/9.5 systems, the command you will use is select pg_xlog_replay_resume(). You can confirm the status of your database by using the built in postgres admin functions found [here:] (https://www.postgresql.org/docs/current/functions-admin.html#FUNCTIONS-RECOVERY-CONTROL-TABLE)
- a pgBackRest restore is destructive in the sense that it deletes the existing primary deployment for the cluster prior to creating a new deployment containing the restored primary database. However, in the event that the pgBackRest restore job fails, the `pgo restore` command be can be run again, and instead of first deleting the primary deployment (since one no longer exists), a new primary will simply be created according to any options specified. Additionally, even though the original primary deployment will be deleted, the original primary PVC will remain.
- there is currently no Operator validation of user entered pgBackRest command options, you will need to make sure to enter these correctly, if not the pgBackRest restore command can fail.
- the restore workflow does not perform a backup after the restore nor does it verify that any replicas are in a working status after the restore, it is possible you might have to take actions on the replica to get them back to replicating with the new restored primary.
- pgbackrest.org suggests running a pgbackrest backup after a restore, this needs to be done by the DBA as part of a restore
- when performing a pgBackRest restore, the **node-label** flag can be utilized to target a specific node for both the pgBackRest restore job and the new (i.e. restored) primary deployment that is then created for the cluster. If a node label is not specified, the restore job will not target any specific node, and the restored primary deployment will inherit any node labels defined for the original primary deployment.

pgbackrest AWS S3 Support

The PostgreSQL Operator supports the use AWS S3 storage buckets for the pgbackrest repository in any pgbackrest-enabled cluster. When S3 support is enabled for a cluster, all archives will automatically be pushed to a pre-configured S3 storage bucket, and that same bucket can then be utilized for the creation of any backups as well as when performing restores. Please note that once a storage type has been selected for a cluster during cluster creation (specifically `local`, `s3`, or *both*, as described in detail below), it cannot be changed.

The PostgreSQL Operator allows for the configuration of a single storage bucket, which can then be utilized across multiple clusters. Once S3 support has been enabled for a cluster, pgbackrest will create a `backrestrepo` directory in the root of the configured S3 storage bucket (if it does not already exist), and subdirectories will then be created under the `backrestrepo` directory for each cluster created with S3 storage enabled.

S3 Configuration In order to enable S3 storage, you must provide the required AWS S3 configuration information prior to deploying the Operator. First, you will need to add the proper S3 bucket name, AWS S3 endpoint and AWS S3 region to the `Cluster` section of the `pgo.yaml` configuration file (additional information regarding the configuration of the `pgo.yaml` file can be found here) :

```
Cluster:
  BackrestS3Bucket: containers-dev-pgbackrest
  BackrestS3Endpoint: s3.amazonaws.com
  BackrestS3Region: us-east-1
```

You will then need to specify the proper credentials for authenticating into the S3 bucket specified by adding a **key** and **key secret** to the `$PGOROOT/conf/pgo-backrest repo/aws-s3-credentials.yaml` configuration file:

```
---
aws-s3-key: ABCDEFGHIJKLMNOPQRST
aws-s3-key-secret: ABCDEFG/HIJKLMNOPQSTU/VWXYZABCDEFGHIJKLM
```

Once the above configuration details have been provided, you can deploy the Operator per the PGO installation instructions.

Enabling S3 Storage in a Cluster With S3 storage properly configured within your PGO installation, you can now select either local storage, S3 storage, or *both* when creating a new cluster. The type of storage selected upon creation of the cluster will determine the type of storage that can subsequently be used when performing pgbackrest backups and restores. A storage type is specified using the `--pgbackrest-storage-type` flag, and can be one of the following values:

- `local` - pgbackrest will use volumes local to the container (e.g. Persistent Volumes) for storing archives, creating backups and locating backups for restores. This is the default value for the `--pgbackrest-storage-type` flag.
- `s3` - pgbackrest will use the pre-configured AWS S3 storage bucket for storing archives, creating backups and locating backups for restores
- `local,s3` (both) - pgbackrest will use both volumes local to the container (e.g. persistent volumes), as well as the pre-configured AWS S3 storage bucket, for storing archives. Also allows the use of local and/or S3 storage when performing backups and restores.

For instance, the following command enables both `local` and `s3` storage in a new cluster:

```
pgo create cluster mycluster --pgbackrest-storage-type=local,s3 -n pgouser1
```

As described above, this will result in pgbackrest pushing archives to both local and S3 storage, while also allowing both local and S3 storage to be utilized for backups and restores. However, you could also enable S3 storage only when creating the cluster:

```
pgo create cluster mycluster --pgbackrest-storage-type=s3 -n pgouser1
```

Now all archives for the cluster will be pushed to S3 storage only, and local storage will not be utilized for storing archives (nor can local storage be utilized for backups and restores).

Using S3 to Backup & Restore As described above, once S3 storage has been enabled for a cluster, it can also be used when backing up or restoring a cluster. Here a both local and S3 storage is selected when performing a backup:

```
pgo backup mycluster --backup-type=pgbackrest --pgbackrest-storage-type=local,s3 -n pgouser1
```

This results in pgbackrest creating a backup in a local volume (e.g. a persistent volume), while also creating an additional backup in the configured S3 storage bucket. However, a backup can be created using S3 storage only:

```
pgo backup mycluster --backup-type=pgbackrest --pgbackrest-storage-type=s3 -n pgouser1
```

Now pgbackrest will only create a backup in the S3 storage bucket only.

When performing a restore, either `local` or `s3` must be selected (selecting both for a restore will result in an error). For instance, the following command specifies S3 storage for the restore:

```
pgo restore mycluster --pgbackrest-storage-type=s3 -n pgouser1
```

This will result in a full restore utilizing the backups and archives stored in the configured S3 storage bucket.

Please note that because `local` is the default storage type for the `backup` and `restore` commands, `s3` must be explicitly set using the `--pgbackrest-storage-type` flag when performing backups and restores on clusters where only S3 storage is enabled.

AWS Certificate Authority The PostgreSQL Operator installation includes a default certificate bundle that is utilized by default to establish trust between pgbackrest and the AWS S3 endpoint used for S3 storage. Please modify or replace this certificate bundle as needed prior to deploying the Operator if another certificate authority is needed to properly establish trust between pgbackrest and your S3 endpoint.

The certificate bundle can be found here: `$PGOROOT/pgo-backrest-repo/aws-s3-ca.crt`.

When modifying or replacing the certificate bundle, please be sure to maintain the same path and filename.

Custom PostgreSQL SSL Configurations

The PostgreSQL Operator can create clusters that use SSL authentication by utilizing custom configmaps.

Configuration Files for SSL Authentication Users and administrators can specify a custom set of PostgreSQL configuration files to be used when creating a new PostgreSQL cluster. This example uses the files below-

- postgresql.conf
- pg_hba.conf
- pg_ident.conf

along with generated security certificates, to setup a custom SSL configuration.

Config Files Purpose The *postgresql.conf* file is the main PostgreSQL configuration file that allows the definition of a wide variety of tuning parameters and features.

The *pg_hba.conf* file is the way Postgresql secures client access.

The *pg_ident.conf* is the ident map file and defines user name maps.

ConfigMap Creation 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 the script called by create.sh to create self-signed certificates (server and client) for the example:

```
$PGOROOT/examples/ssl-creator.sh.
```

Additionally, this script requires the certstrap utility to be installed. An install script is provided to install the correct version for the example if not already installed.

The relevant configuration files are located in the configs directory and will configure the clsuter to use SSL client authentication. These, along with the client certificate for the user 'testuser' and a server certificate for 'pgo-custom-ssl-container', will make up the necessary configuration items to be stored in the 'pgo-custom-ssl-config' configmap.

Example Steps Run the script as follow:

```
cd $PGOROOT/examples/custom-config-ssl
./create.sh
```

This will generate a configmap named 'pgo-custom-ssl-config'.

Once this configmap is created, run

```
pgo create cluster customsslcluster --custom-config pgo-custom-ssl-config -n ${PGO_NAMESPACE}
```

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

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

```
${PGO_CMD} get service -n ${PGO_NAMESPACE}
NAME                    CLUSTER-IP       EXTERNAL-IP   PORT(S)                   AGE
customsslcluster        172.30.211.108   <none>        5432/TCP
```

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

```
172.30.211.108 pgo-custom-ssl-container
```

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 pgo-custom-ssl-container.

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@pgo-custom-ssl-container:5432/userdb?sslmode=verify-full&sslrootcert=/home/
```

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. i

```
psql
   "postgresql://testuser@IP_OF_PGSQL:5432/userdb?sslmode=require&sslrootcert=/home/pgo/odev/src/gith
```

You should see a connection that looks like the following:

```
psql (12.1)
SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384, bits: 256, compression:
   off)
Type "help" for help.

userdb=>
```

Important Notes Because SSL will be required for connections, certain features of the Operator will not function as expected. These include the following:

```
pgo test
pgo load
pgo apply
```

Direct API Calls

The API can also be accessed by interacting directly with the API server. This can be done by making curl calls to POST or GET information from the server. In order to make these calls you will need to provide certificates along with your request using the `--cacert`, `--key`, and `--cert` flags. Next you will need to provide the username and password for the RBAC along with a header that includes the content type and the `--insecure` flag. These flags will be the same for all of your interactions with the API server and can be seen in the following examples.

The most basic example of this interaction is getting the version of the API server. You can send a GET request to `$PGO_APISERVER_URL/vers` and this will send back a json response including the API server version. This is important because the server version and the client version must match. If you are using `pgo` this means you must have the correct version of the client but with a direct call you can specify the client version as part of the request.

The API server is setup to work with the pgo command line interface so the parameters that are passed to the server can be found by looking at the related flags. For example, the series parameter used in the `create` example below is the same as the `-e, --series` flag that is described in the pgo cli docs.

Get API Server Version

```
curl --cacert $PGO_CA_CERT --key $PGO_CLIENT_KEY --cert $PGO_CA_CERT \
-u pgoadmin:examplepassword -H "Content-Type:application/json" --insecure \
-X GET $PGO_APISERVER_URL/version
```

You can create a cluster by sending a POST request to `$PGO_APISERVER_URL/clusters`. In this example `--data` is being sent to the API URL that includes the client version that was returned from the version call, the namespace where the cluster should be created, the name of the new cluster and the series number. Series sets the number of clusters that will be created in the namespace.

Create Cluster

```
curl --cacert $PGO_CA_CERT --key $PGO_CLIENT_KEY --cert $PGO_CA_CERT \
-u pgoadmin:examplepassword -H "Content-Type:application/json" --insecure \
-X POST --data \
  '{"ClientVersion":"4.2.0",
  "Namespace":"pgouser1",
  "Name":"mycluster",
  "Series":1}' \
$PGO_APISERVER_URL/clusters
```

The last two examples show you how to `show` and `delete` a cluster. Notice how instead of passing `"Name":"mycluster"` you pass `"Clustername":"mycluster"`to reference a cluster that has already been created. For the show cluster example you can replace `"Clustername":"mycluster"` with `"AllFlag":true` to show all of the clusters that are in the given namespace.

Show Cluster

```
curl --cacert $PGO_CA_CERT --key $PGO_CLIENT_KEY --cert $PGO_CA_CERT \
-u pgoadmin:examplepassword -H "Content-Type:application/json" --insecure \
-X POST --data \
  '{"ClientVersion":"4.2.0",
  "Namespace":"pgouser1",
  "Clustername":"mycluster"}' \
$PGO_APISERVER_URL/showclusters
```

Delete Cluster

```
curl --cacert $PGO_CA_CERT --key $PGO_CLIENT_KEY --cert $PGO_CA_CERT \
-u pgoadmin:examplepassword -H "Content-Type:application/json" --insecure \
-X POST --data \
  '{"ClientVersion":"4.2.0",
  "Namespace":"pgouser1",
  "Clustername":"mycluster"}' \
$PGO_APISERVER_URL/clustersdelete
```

Considerations for PostgreSQL Operator Deployments in Multi-Zone Cloud Environments

Overview When using the PostgreSQL Operator in a Kubernetes cluster consisting of nodes that span multiple zones, special consideration must be taken to ensure all pods and the associated volumes re scheduled and provisioned within the same zone.

Given that a pod is unable mount a volume that is located in another zone, any volumes that are dynamically provisioned must be provisioned in a topology-aware manner according to the specific scheduling requirements for the pod.

This means that when a new PostgreSQL cluster is created, it is necessary to ensure that the volume containing the database files for the primary PostgreSQL database within the PostgreSQL clluster is provisioned in the same zone as the node containing the PostgreSQL primary pod that will be accesing the applicable volume.

Dynamic Provisioning of Volumes: Default Behavoir By default, the Kubernetes scheduler will ensure any pods created that claim a specific volume via a PVC are scheduled on a node in the same zone as that volume. This is part of the default Kubernetes multi-zone support.

However, when using Kubernetes dynamic provisioning, volumes are not provisioned in a topology-aware manner.

More specifically, when using dynamnic provisioning, volumes wills not be provisioned according to the same scheduling requirements that will be placed on the pod that will be using it (e.g. it will not consider node selectors, resource requirements, pod affinity/anti-affinity, and various other scheduling requirements). Rather, PVCs are immediately bound as soon as they are requested, which means volumes are provisioned without knowledge of these scheduling requirements.

This behavior defined using the `volumeBindingMode` configuration applicable to the Storage Class being utilized to dynamically provision the volume. By default,`volumeBindingMode` is set to `Immediate`.

This default behavoir for dynamic provisioning can be seen in the Storage Class definition for a Google Cloud Engine Persistent Disk (GCE PD):

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: example-sc
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
volumeBindingMode: Immediate
```

As indicated, `volumeBindingMode` indicates the default value of `Immediate`.

Issues with Dynamic Provisioning of Volumes in PostgreSQL Operator Unfortunately, the default setting for dynamic provisioning of volumes in mulit-zone Kubernetes cluster environments results in undesired behavoir when using the PostgreSQL Operator.

Within the PostgreSQL Operator, a **node label** is implemented as a `preferredDuringSchedulingIgnoredDuringExecution` node affinity rule, which is an affinity rule that Kubernetes will attempt to adhere to when scheduling any pods for the cluster, but *will not guarantee*. More information on node affinity rules can be found here).

By using `Immediate` for the `volumeBindingMode` in a multi-zone cluster environment, the scheduler will ignore any requested *(but not mandatory)* scheduling requirements if necessary to ensure the pod can be scheduled. The scheduler will ultimately schedule the pod on a node in the same zone as the volume, even if another node was requested for scheduling that pod.

As it relates to the PostgreSQL Operator specifically, a node label specified using the `--node-label` option when creating a cluster using the `pgo create cluster` command in order target a specific node (or nodes) for the deployment of that cluster.

Therefore, if the volume ends up in a zone other than the zone containing the node (or nodes) defined by the node label, the node label will be ignored, and the pod will be scheduled according to the zone containing the volume.

Configuring Volumes to be Topology Aware In order to overcome this default behavior, it is necessary to make the dynamically provisioned volumes topology aware.

This is accomplished by setting the `volumeBindingMode` for the storage class to `WaitForFirstConsumer`, which delays the dynamic provisioning of a volume until a pod using it is created.

In other words, the PVC is no longer bound as soon as it is requested, but rather waits for a pod utilizing it to be creating prior to binding. This change ensures that volume can take into account the scheduling requirements for the pod, which in the case of a multi-zone cluster means ensuring the volume is provisioned in the same zone containing the node where the pod has be scheduled. This also means the scheduler should no longer ignore a node label in order to follow a volume to another zone when scheduling a pod, since the volume will now follow the pod according to the pods specificscheduling requirements.

The following is an example of the the same Storage Class defined above, only with `volumeBindingMode` now set to `WaitForFirstConsumer`:

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: example-sc
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
volumeBindingMode: WaitForFirstConsumer
```

Additional Solutions If you are using a version of Kubernetes that does not support `WaitForFirstConsumer`, an alternate *(and now deprecated)* solution exists in the form of parameters that can be defined on the Storage Class definition to ensure volumes are provisioned in a specific zone (or zones).

For instance, when defining a Storage Class for a GCE PD for use in Google Kubernetes Engine (GKE) cluster, the **zone** parameter can be used to ensure any volumes dynamically provisioned using that Storage Class are located in that specific zone. The following is an example of a Storage Class for a GKE cluster that will provision volumes in the **us-east1** zone:

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: example-sc
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
  replication-type: none
  zone: us-east1
```

Once storage classes have been defined for one or more zones, they can then be defined as one or more storage configurations within the pgo.yaml configuration file (as described in the PGO YAML configuration guide).

From there those storage configurations can then be selected when creating a new cluster, as shown in the following example:

```
pgo create cluster mycluster --storage-config=example-sc
```

With this approach, the pod will once again be scheduled according to the zone in which the volume was provisioned.

However, the zone parameters defined on the Storage Class bring consistency to scheduling by guaranteeing that the volume, and therefore also the pod using that volume, are scheduled in a specific zone as defined by the user, bringing consistency and predictability to volume provisioning and pod scheduling in multi-zone clusters.

For more information regarding the specific parameters available for the Storage Classes being utilizing in your cloud environment, please see the Kubernetes documentation for Storage Classes.

Lastly, while the above applies to the dynamic provisioning of volumes, it should be noted that volumes can also be manually provisioned in desired zones in order to achieve the desired topology requirements for any pods and their volumes.

pgo show upgrade

Show upgrade information

Synopsis

Show upgrade information. For example:

```
pgo show upgrade mycluster
pgo show upgrade [flags]
```

Options

```
  -h, --help   help for upgrade
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo show - Show the description of a cluster

Auto generated by spf13/cobra on 27-Mar-2019

pgo show user

Show user information

Synopsis

Show users on a cluster. For example:

```
pgo show user --all
pgo show user mycluster
pgo show user --selector=name=nycluster
```

```
pgo show user [flags]
```

Options

```
      --all                show all clusters.
      --expired string     Shows passwords that will expire in X days.
  -h, --help               help for user
  -s, --selector string    The selector to use for cluster filtering.
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo show - Show the description of a cluster

Auto generated by spf13/cobra on 4-Oct-2019

pgo show workflow

Show workflow information

Synopsis

Show workflow information for a given workflow. For example:

```
pgo show workflow 25927091-b343-4017-be4b-71575f0b3eb5
```

```
pgo show workflow [flags]
```

Options

```
  -h, --help   help for workflow
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo show - Show the description of a cluster

Auto generated by spf13/cobra on 4-Oct-2019

pgo status

Display PostgreSQL cluster status

Synopsis

Display namespace wide information for PostgreSQL clusters. For example:

```
pgo status
```

```
pgo status [flags]
```

Options

```
  -h, --help            help for status
  -o, --output string   The output format. Currently, json is the only supported value.
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo - The pgo command line interface.

Auto generated by spf13/cobra on 4-Oct-2019

pgo test

Test cluster connectivity

Synopsis

TEST allows you to test the connectivity for a cluster. For example:

```
pgo test mycluster
pgo test --selector=env=research
pgo test --all
```

```
pgo test [flags]
```

Options

```
      --all                 test all resources.
  -h, --help                help for test
  -o, --output string       The output format. Currently, json is the only supported value.
  -s, --selector string     The selector to use for cluster filtering.
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo - The pgo command line interface.

Auto generated by spf13/cobra on 4-Oct-2019

pgo update

Update a pgouser, pgorole, or cluster

Synopsis

The update command allows you to update a pgouser, pgorole, or cluster. For example:

```
pgo update pgouser someuser --pgouser-password=somenewpassword
pgo update pgouser someuser --pgouser-roles="role1,role2"
pgo update pgouser someuser --pgouser-namespaces="pgouser2"
pgo update user mycluster --username=testuser --selector=name=mycluster --password=somepassword
pgo update pgorole somerole --pgorole-permission="Cat"
pgo update namespace mynamespace
pgo update cluster --selector=name=mycluster --autofail=false
pgo update cluster --all --autofail=true
```

```
pgo update [flags]
```

Options

```
  -h, --help   help for update
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo - The pgo command line interface.
- pgo update cluster - Update a PostgreSQL cluster
- pgo update namespace - Update a namespace, applying Operator RBAC
- pgo update pgorole - Update a pgorole
- pgo update pgouser - Update a pgouser
- pgo update user - Update a postgres user

Auto generated by spf13/cobra on 4-Oct-2019

pgo update cluster

Update a PostgreSQL cluster

Synopsis

Update a PostgreSQL cluster. For example:

```
pgo update cluster mycluster --autofail=false
pgo update cluster mycluster myothercluster --autofail=false
pgo update cluster --selector=name=mycluster --autofail=false
pgo update cluster --all --autofail=true
```

```
pgo update cluster [flags]
```

Options

```
      --all                all resources.
      --autofail           autofail default is false.
  -h, --help               help for cluster
      --no-prompt          No command line confirmation.
  -s, --selector string    The selector to use for cluster filtering.
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo update - Update a pgouser, pgorole, or cluster

Auto generated by spf13/cobra on 4-Oct-2019

pgo update namespace

Update a namespace, applying Operator RBAC

Synopsis

UPDATE allows you to update a Namespace. For example: pgo update namespace mynamespace

```
pgo update namespace [flags]
```

Options

```
  -h, --help   help for namespace
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo update - Update a pgouser, pgorole, or cluster

Auto generated by spf13/cobra on 4-Oct-2019

pgo update pgorole

Update a pgorole

Synopsis

UPDATE allows you to update a pgo role. For example: pgo update pgorole somerole –permissions="Cat,Ls

```
pgo update pgorole [flags]
```

Options

```
  -h, --help                 help for pgorole
      --no-prompt            No command line confirmation.
      --permissions string   The permissions to use for updating the pgorole permissions.
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo update - Update a pgouser, pgorole, or cluster

Auto generated by spf13/cobra on 4-Oct-2019

pgo update pgouser

Update a pgouser

Synopsis

UPDATE allows you to update a pgo user. For example: pgo update pgouser myuser –pgouser-roles=somerole pgo update pgouser myuser –pgouser-password=somepassword –pgouser-roles=somerole pgo update pgouser myuser –pgouser-password=somepassword –no-prompt

```
pgo update pgouser [flags]
```

Options

```
      --all-namespaces               all namespaces.
  -h, --help                         help for pgouser
      --no-prompt                    No command line confirmation.
      --pgouser-namespaces string    The namespaces to use for updating the pgouser roles.
      --pgouser-password string      The password to use for updating the pgouser password.
      --pgouser-roles string         The roles to use for updating the pgouser roles.
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo update - Update a pgouser, pgorole, or cluster

Auto generated by spf13/cobra on 4-Oct-2019

Operator Eventing

The Operator creates events from the various life-cycle events going on within the Operator logic and driven by pgo users as they interact with the Operator and as Postgres clusters come and go or get updated.

Event Watching

There is a pgo CLI command:

```
pgo watch alltopic
```

This command connects to the event stream and listens on a topic for event real-time. The command will not complete until the pgo user enters ctrl-C.

This command will connect to localhost:14150 (default) to reach the event stream. If you have the correct priviledges to connect to the Operator pod, you can port forward as follows to form a connection to the event stream:

```
kubectl port-forward postgres-operator-XXXXXX 14150:4150 -n pgo
```

Event Topics

The following topics exist that hold the various Operator generated events:

```
alltopic
clustertopic
backuptopic
loadtopic
postgresusertopic
policytopic
pgpooltopic
pgbouncertopic
pgotopic
pgousertopic
```

Event Types

The various event types are found in the source code at https://github.com/CrunchyData/postgres-operator/blob/master/events/eventtype.g

Event Testing

To test the event logic, you can run the test case for events as follows:

```
# create a connection locally to the event stream
kubectl port-forward postgres-operator-XXXXXX 14150:4150 -n pgo

# specify the event address
export EVENT_ADDR=localhost:14150

# run the test using foomatic as the name of the test cluster
# and pgouser1 as the name of the namespace to test against
cd testing/events
go test -run TestEventCreate -v --kubeconfig=/home/<yourhomedir>/.kube/config
   -clustername=foomatic -namespace=pgouser1
```

Event Deployment

The Operator events are published and subscribed via the NSQ project software (https://nsq.io/). NSQ is found in the pgo-event container which is part of the postgres-operator deployment.

You can see the pgo-event logs by issuing the elog bash function found in the examples/envs.sh script.

NSQ looks for events currently at port 4150. The Operator sends events to the NSQ address as defined in the EVENT__ADDR environment variable.

If you want to disable eventing when installing with Bash, set the following environment variable in the Operator Deployment: "name": "DISABLE__EVENTING" "value": "true"

To disable eventing when installing with Ansible, add the following to your inventory file: pgo__disable__eventing='true' — title: "pgo__update__user" — ## pgo update user

Update a postgres user

Synopsis

UPDATE allows you to update a pgo user. For example: //change a password, set valid days for 40 days from now pgo update user mycluster –username=someuser –password=foo –valid-days=40 //expire password for a user pgo update user mycluster –username=someuser –expire-user //Update all passwords older than the number of days specified pgo update user mycluster –expired=45 –password-length=8

```
pgo update user [flags]
```

Options

```
      --all                     all clusters.
      --expire-user             Performs expiring a user if set to true.
      --expired string          Updates passwords that will expire in X days using an autogenerated
         password.
  -h, --help                    help for user
      --password string         Specifies the user password when updating a user password or
         creating a new user.
      --password-length int     If no password is supplied with the expired flag, this is the length
         of the auto generated password (default 22)
  -s, --selector string         The selector to use for cluster filtering.
      --username string         Updates the postgres user on selective clusters.
      --valid-days int          Sets passwords for new users to X days. (default 30)
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo update - Update a pgouser, pgorole, or cluster

Auto generated by spf13/cobra on 4-Oct-2019

pgo upgrade

Perform an upgrade

Synopsis

UPGRADE performs an upgrade on a PostgreSQL cluster. For example:

pgo upgrade mycluster

This upgrade will update the CCPImageTag of the deployment for the following: primary, replicas, and backrest-repo. The running containers are upgraded one at a time, sequentially, in the following order: replicas, backrest-repo, then primary.

```
pgo upgrade [flags]
```

Options

```
      --ccp-image-tag string   The CCPImageTag to use for cluster creation. If specified,
         overrides the pgo.yaml setting.
  -h, --help                   help for upgrade
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo - The pgo command line interface.

Auto generated by spf13/cobra on 4-Oct-2019

pgo user

Manage PostgreSQL users

Synopsis

USER allows you to manage users and passwords across a set of clusters. For example:

```
pgo user --selector=name=mycluster --update-passwords
pgo user --change-password=bob --expired=300 --selector=name=mycluster --password=newpass
```

```
pgo user [flags]
```

Options

```
      --change-password string   Updates the password for a user on selective clusters.
      --db string                Grants the user access to a database.
      --expired string           required flag when updating passwords that will expire in X days
         using --update-passwords flag.
  -h, --help                     help for user
      --password string          Specifies the user password when updating a user password or
         creating a new user.
      --password-length int      If no password is supplied, this is the length of the auto
         generated password (default 12)
  -s, --selector string          The selector to use for cluster filtering.
      --update-passwords         Performs password updating on expired passwords.
      --valid-days int           Sets passwords for new users to X days. (default 30)
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo - The pgo command line interface.

Auto generated by spf13/cobra on 23-Jul-2019

pgo version

Print version information for the PostgreSQL Operator

Synopsis

VERSION allows you to print version information for the postgres-operator. For example:

```
pgo version
```

```
pgo version [flags]
```

Options

```
  -h, --help   help for version
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo - The pgo command line interface.

Auto generated by spf13/cobra on 4-Oct-2019

pgo watch

Print watch information for the PostgreSQL Operator

Synopsis

WATCH allows you to watch event information for the postgres-operator. For example: pgo watch –pgo-event-address=localhost:14150 alltopic pgo watch alltopic

```
pgo watch [flags]
```

Options

```
  -h, --help                       help for watch
  -a, --pgo-event-address string   The address (host:port) where the event stream is. (default
      "localhost:14150")
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo - The pgo command line interface.

Auto generated by spf13/cobra on 4-Oct-2019

Configuring Encryption of PostgreSQL Operator API Connection

The PostgreSQL Operator REST API connection is encrypted with keys stored in the *pgo.tls* Secret.

The pgo.tls Secret can be generated prior to starting the PostgreSQL Operator or you can let the PostgreSQL Operator generate the Secret for you if the Secret does not exist.

Adjust the default keys to meet your security requirements using your own keys. The *pgo.tls* Secret is created when you run:

```
make deployoperator
```

The keys are generated when the RBAC script is executed by the cluster admin:

```
make installrbac
```

In some scenarios like an OLM deployment, it is preferable for the Operator to generate the Secret keys at runtime, if the pgo.tls Secret does not exit when the Operator starts, a new TLS Secret will be generated.

In this scenario, you can extract the generated Secret TLS keys using:

```
kubectl cp <pgo-namespace>/<pgo-pod>:/tmp/server.key /tmp/server.key -c apiserver
kubectl cp <pgo-namespace>/<pgo-pod>:/tmp/server.crt /tmp/server.crt -c apiserver
```

example of the command below:

```
kubectl cp pgo/postgres-operator-585584f57d-ntwr5:tmp/server.key /tmp/server.key -c apiserver
kubectl cp pgo/postgres-operator-585584f57d-ntwr5:tmp/server.crt /tmp/server.crt -c apiserver
```

This server.key and server.crt can then be used to access the *pgo-apiserver* from the pgo CLI by setting the following variables in your client environment:

```
export PGO_CA_CERT=/tmp/server.crt
export PGO_CLIENT_CERT=/tmp/server.crt
export PGO_CLIENT_KEY=/tmp/server.key
```

You can view the TLS secret using:

```
kubectl get secret pgo.tls -n pgo
```

or

```
oc get secret pgo.tls -n pgo
```

If you create the Secret outside of the Operator, for example using the default installation script, the key and cert that are generated by the default installation are found here:

```
$PGOROOT/conf/postgres-operator/server.crt
$PGOROOT/conf/postgres-operator/server.key
```

The key and cert are generated using the *deploy/gen-api-keys.sh* script.

That script gets executed when running:

```
make installrbac
```

You can extract the server.key and server.crt from the Secret using the following:

```
oc get secret pgo.tls -n $PGO_OPERATOR_NAMESPACE -o jsonpath='{.data.tls\.key}' | base64 --decode
   > /tmp/server.key
oc get secret pgo.tls -n $PGO_OPERATOR_NAMESPACE -o jsonpath='{.data.tls\.crt}' | base64 --decode
   > /tmp/server.crt
```

This server.key and server.crt can then be used to access the *pgo-apiserver* REST API from the pgo CLI on your client host.

PostreSQL Operator RBAC

The *conf/postgres-operator/pgorole* file is read at start up time when the operator is deployed to the Kubernetes cluster. This file defines the PostgreSQL Operator roles whereby PostgreSQL Operator API users can be authorized.

The *conf/postgres-operator/pgouser* file is read at start up time also and contains username, password, role, and namespace information as follows:

```
username:password:pgoadmin:
pgouser1:password:pgoadmin:pgouser1
pgouser2:password:pgoadmin:pgouser2
pgouser3:password:pgoadmin:pgouser1,pgouser2
readonlyuser:password:pgoreader:
```

The format of the pgouser server file is:

```
<username>:<password>:<role>:<namespace,namespace>
```

The namespace is a comma separated list of namespaces that user has access to. If you do not specify a namespace, then all namespaces is assumed, meaning this user can access any namespace that the Operator is watching.

A user creates a *.pgouser* file in their $HOME directory to identify themselves to the Operator. An entry in .pgouser will need to match entries in the *conf/postgres-operator/pgouser* file. A sample *.pgouser* file contains the following:

```
username:password
```

The format of the .pgouser client file is:

```
<username>:<password>
```

The users pgouser file can also be located at:

/etc/pgo/pgouser

or it can be found at a path specified by the PGOUSER environment variable.

If the user tries to access a namespace that they are not configured for within the server side *pgouser* file then they will get an error message as follows:

```
Error: user [pgouser1] is not allowed access to namespace [pgouser2]
```

The following list shows the current complete list of possible pgo permissions that you can specify within the *pgorole* file when creating roles:

Permission	Description
ApplyPolicy	allow *pgo apply*
Cat	allow *pgo cat*
CreateBackup	allow *pgo backup*
CreateBenchmark	allow *pgo create benchmark*
CreateCluster	allow *pgo create cluster*
CreateDump	allow *pgo create pgdump*
CreateFailover	allow *pgo failover*
CreatePgbouncer	allow *pgo create pgbouncer*
CreatePgpool	allow *pgo create pgpool*
CreatePolicy	allow *pgo create policy*
CreateSchedule	allow *pgo create schedule*
CreateUpgrade	allow *pgo upgrade*
CreateUser	allow *pgo create user*
DeleteBackup	allow *pgo delete backup*
DeleteBenchmark	allow *pgo delete benchmark*
DeleteCluster	allow *pgo delete cluster*
DeletePgbouncer	allow *pgo delete pgbouncer*

Permission	Description
DeletePgpool	allow *pgo delete pgpool*
DeletePolicy	allow *pgo delete policy*
DeleteSchedule	allow *pgo delete schedule*
DeleteUpgrade	allow *pgo delete upgrade*
DeleteUser	allow *pgo delete user*
DfCluster	allow *pgo df*
Label	allow *pgo label*
Load	allow *pgo load*
Ls	allow *pgo ls*
Reload	allow *pgo reload*
Restore	allow *pgo restore*
RestoreDump	allow *pgo restore* for pgdumps
ShowBackup	allow *pgo show backup*
ShowBenchmark	allow *pgo show benchmark*
ShowCluster	allow *pgo show cluster*
ShowConfig	allow *pgo show config*
ShowPolicy	allow *pgo show policy*
ShowPVC	allow *pgo show pvc*
ShowSchedule	allow *pgo show schedule*
ShowNamespace	allow *pgo show namespace*
ShowUpgrade	allow *pgo show upgrade*
ShowWorkflow	allow *pgo show workflow*
Status	allow *pgo status*
TestCluster	allow *pgo test*
UpdateCluster	allow *pgo update cluster*
User	allow *pgo user*
Version	allow *pgo version*

If the user is unauthorized for a pgo command, the user will get back this response:

```
Error:  Authentication Failed: 401
```

Making Security Changes

Importantly, it is necesssary to redeploy the PostgreSQL Operator prior to giving effect to the user security changes in the pgouser and pgorole files:

```
make deployoperator
```

Performing this command will recreate the *pgo-config* ConfigMap that stores these files and is mounted by the Operator during its initialization.

Installation of PostgreSQL Operator RBAC

Please note, installation of the PostgreSQL Operator RBAC requires Kubernetes Cluster-Admin.

The first step is to install the PostgreSQL Operator RBAC configuration. This can be accomplished by running:

```
make installrbac
```

This script will install the PostreSQL Operator Custom Resource Definitions, CRD’s and creates the following RBAC resources on your Kubernetes cluster:

Setting	Definition
Custom Resource Definitions (crd.yaml)	pgbackups
	pgclusters
	pgpolicies
	pgreplicas
	pgtasks
	pgupgrades
Cluster Roles (cluster-roles.yaml)	pgopclusterrole
	pgopclusterrolecrd
Cluster Role Bindings (cluster-roles-bindings.yaml)	pgopclusterbinding
	pgopclusterbindingcrd
Service Account (service-accounts.yaml)	postgres-operator
	pgo-backrest
Roles (rbac.yaml)	pgo-role
	pgo-backrest-role
Role Bindings (rbac.yaml)	pgo-backrest-role-binding
	pgo-role-binding

Note that the cluster role bindings have a naming convention of pgopclusterbinding-$PGO_OPERATOR_NAMESPACEandpgopclusterbindin$

The PGO_OPERATOR_NAMESPACE environment variable is added to make each cluster role binding name unique and to support more than a single PostgreSQL Operator being deployed on the same Kubernertes cluster.

Upgrading the Operator

Various Operator releases will require action by the Operator administrator of your organization in order to upgrade to the next release of the Operator. Some upgrade steps are automated within the Operator but not all are possible at this time.

This section of the documentation shows specific steps required to upgrade different versions of the Postgres Operator depending on your current environment.

[Upgrade Postgres Operator to 3.5] ({{< relref "upgrade/upgradeto35.md" >}})

[Postgres Operator 3.5 Minor Version Upgrade] ({{< relref "upgrade/upgrade35.md" >}})

[Upgrade Postgres Operator from 3.5 to 4.1] ({{< relref "upgrade/upgrade35to4.md" >}})

[Upgrade Postgres Operator from 4.0.1 to 4.1.0 (Bash)] ({{< relref "upgrade/upgrade40to41_bash.md" >}})

[Upgrade Postgres Operator from 4.0.1 to 4.1.0 (Ansible)] ({{< relref "upgrade/upgrade40to41_ansible.md" >}})

[Upgrade Postgres Operator from 4.1.0 to a patch release] ({{< relref "upgrade/upgrade41.md" >}})

Upgrading A Postgres Cluster

Using the operator, it is possible to upgrade a postgres cluster in place. When a pgo upgrade command is issued, and a –CCPImageTag is specified, the operator will upgrade each replica and the primary to the new CCPImageTag version. It is important to note that the postgres version of the new container should be compatible with the current running version. There is currently no version check done to ensure compatibility.

The upgrade is accomplished by updating the CCPImageTag version in the deployment, which causes the old pod to be terminated and a new pod created with the updated deployment specification.

When the upgrade starts, each replica is upgraded seqentially, waiting for the previous replica to go ready before updating the next. After the replicas complete, the primary is then upgraded to the new image. The upgrade process respects the *autofail* setting as provided in the pgo.yaml or as a command line flag, if applicable.

When the cluster is not in *autofail* mode, the deployments simply create a new pod when updated, terminating the old one. When autofail is enabled and the primary deployment is updated, the cluster behaves as though the primary had failed and begins the failover process. See *Automatic Failover* in the *Overview* section for more details on this and replica replacement.

At this time, the backrest-repo container is not upgraded during this upgrade as it is part of the postgres operator release and is updated with the operator.

Minor Upgrade Example

In this example, we are upgrading a cluster from PostgreSQL 11.5 to 11.6 using the `crunchy-postgres:centos7-11.6-4.2.0` container:

`pgo upgrade mycluster --ccp-image-tag=centos7-11.6-4.2.0`

For more information, please see the `pgo upgrade` documentation [here.] ({{< relref "operatorcli/cli/pgo_upgrade.md" >}})

Operator Namespaces

The Operator itself knows which namespace it is running within by referencing the PGO_OPERATOR_NAMESPACE environment variable at startup time from within its Deployment definition.

The PGO_OPERATOR_NAMESPACE environment variable a user sets in their .bashrc file is used to determine what namespace the Operator is deployed into. The PGO_OPERATOR_NAMESPACE variable is referenced by the Operator during deployment.

The .bashrc NAMESPACE environment variable a user sets determines which namespaces the Operator will watch.

Namespace Watching

The Operator at startup time determines which namespaces it will service based on what it finds in the NAMESPACE environment variable that is passed into the Operator containers within the deployment.json file.

The NAMESPACE variable can hold different values which determine the namespaces which will be *watched* by the Operator.

The format of the NAMESPACE value is modeled after the following document:

https://github.com/operator-framework/operator-lifecycle-manager/blob/0.12.0/doc/design/operatorgroups.md

OwnNamespace Example

Prior to version 4.0, the Operator was deployed into a single namespace and Postgres Clusters created by it were created in that same namespace.

To achieve that same deployment model you would use variable settings as follows:

```
export PGO_OPERATOR_NAMESPACE=pgo
export NAMESPACE=pgo
```

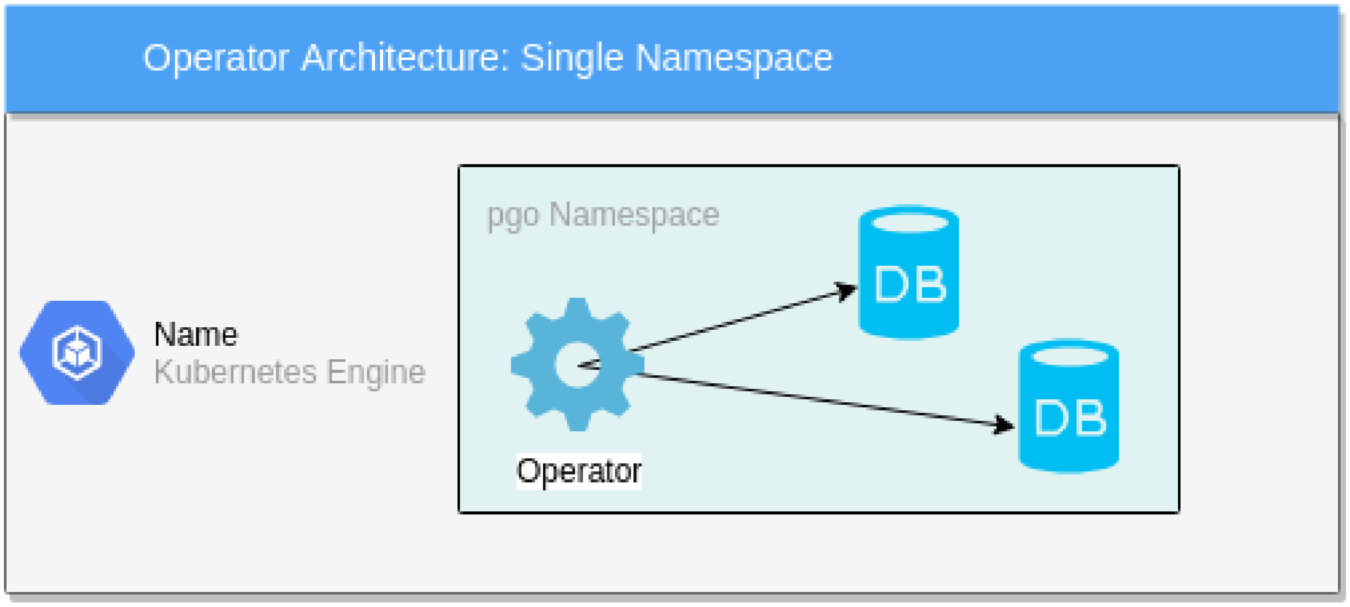

Figure 6: Reference

SingleNamespace Example

To have the Operator deployed into its own namespace but create Postgres Clusters into a different namespace the variables would be as follows:

```
export PGO_OPERATOR_NAMESPACE=pgo
export NAMESPACE=pgouser1
```

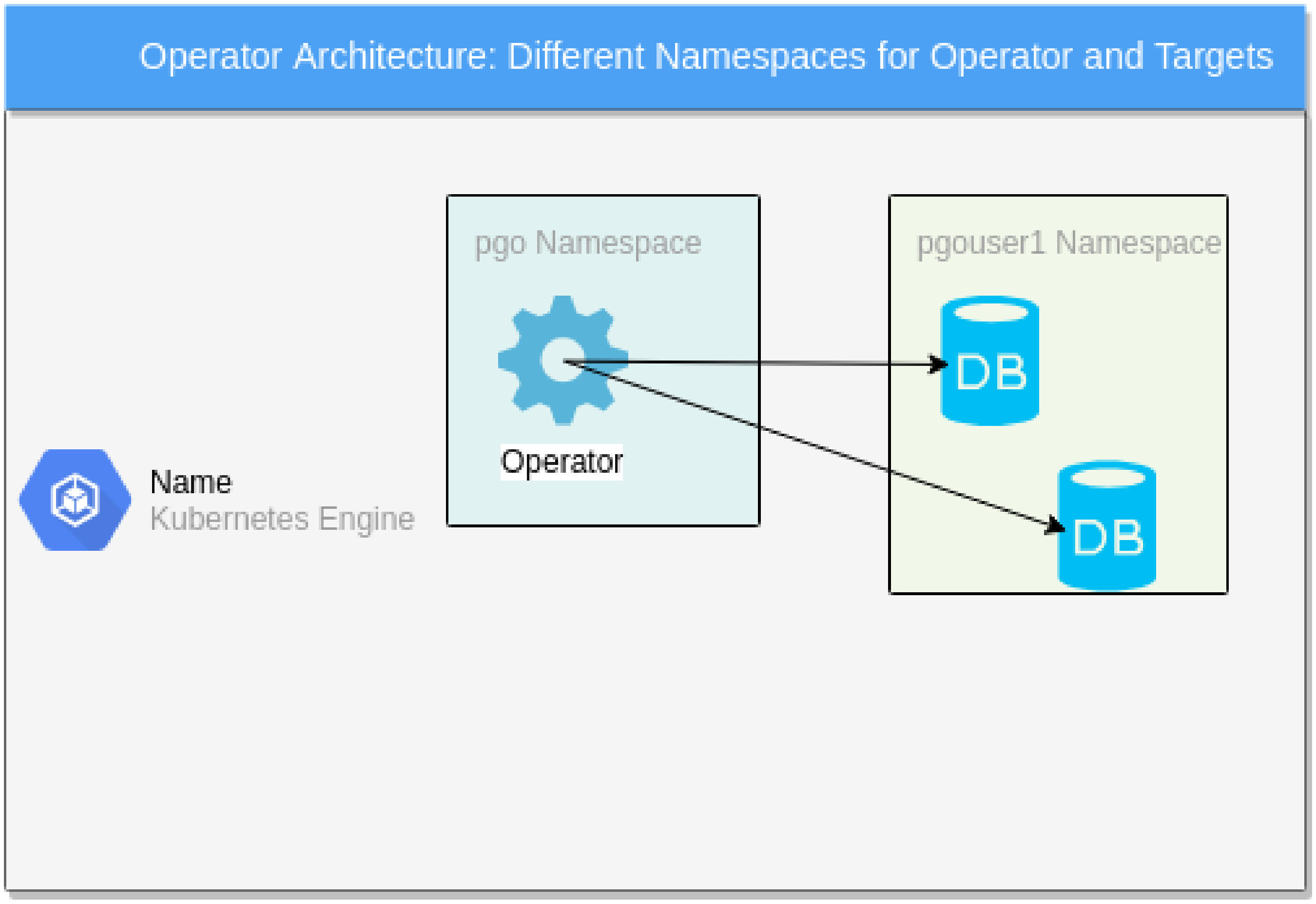
Operator Architecture: Different Namespaces for Operator and Targets
pgo Namespace
Operator
pgouser1 Namespace
DB
DB
Name
Kubernetes Engine

Figure 7: Reference

MultiNamespace Example

To have the Operator deployed into its own namespace but create Postgres Clusters into more than one namespace the variables would be as follows:

```
export PGO_OPERATOR_NAMESPACE=pgo
export NAMESPACE=pgouser1,pgouser2
```

AllNamespaces Example

To have the Operator deployed into its own namespace but create Postgres Clusters into any target namespace the variables would be as follows:

```
export PGO_OPERATOR_NAMESPACE=pgo
export NAMESPACE=
```

Here the empty string value represents *all* namespaces.

RBAC

To support multiple namespace watching, the Operator deployment process changes somewhat from 3.X releases.

Each namespace to be watched requires its own copy of the the following resources for working with the Operator:

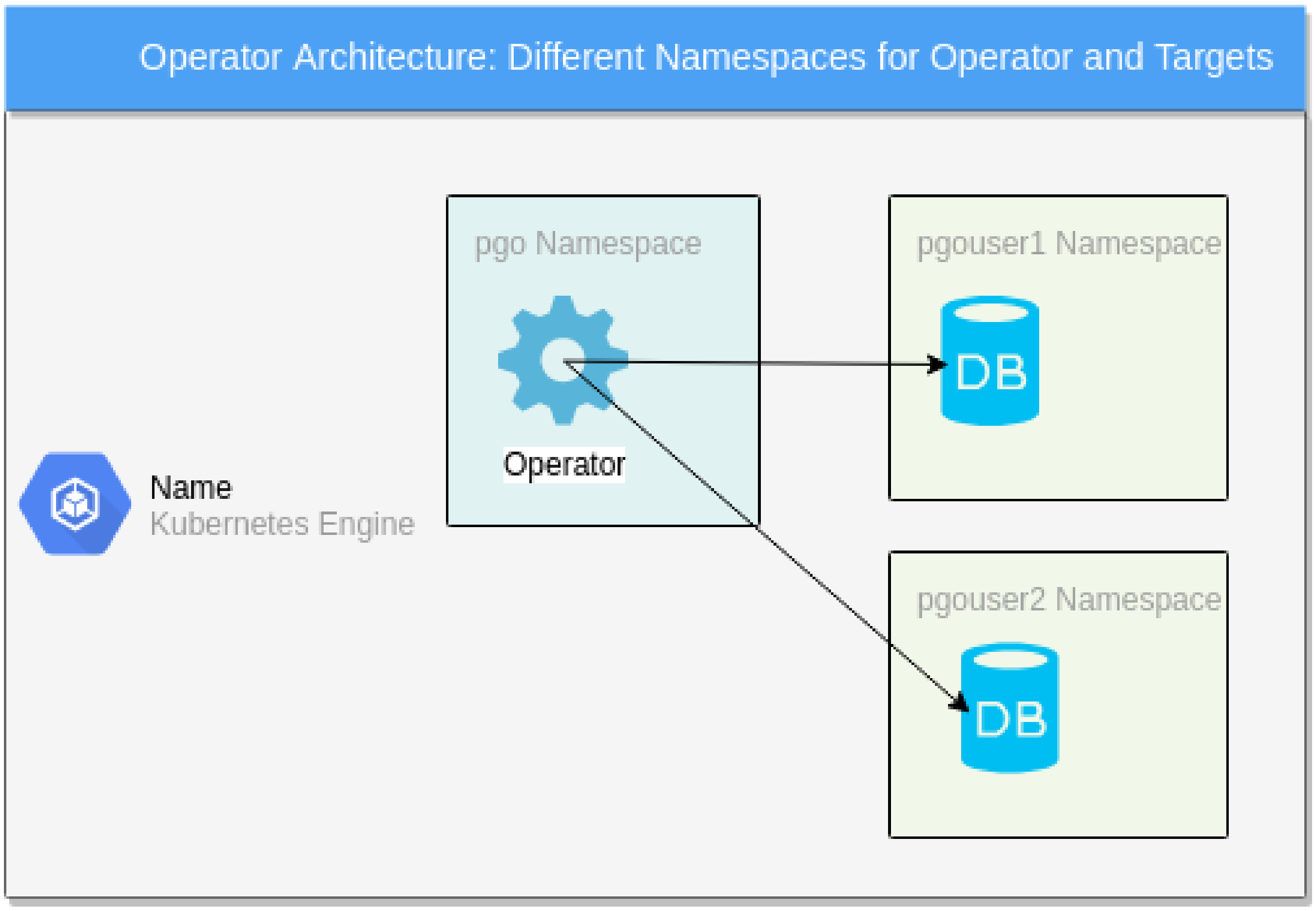

Figure 8: Reference

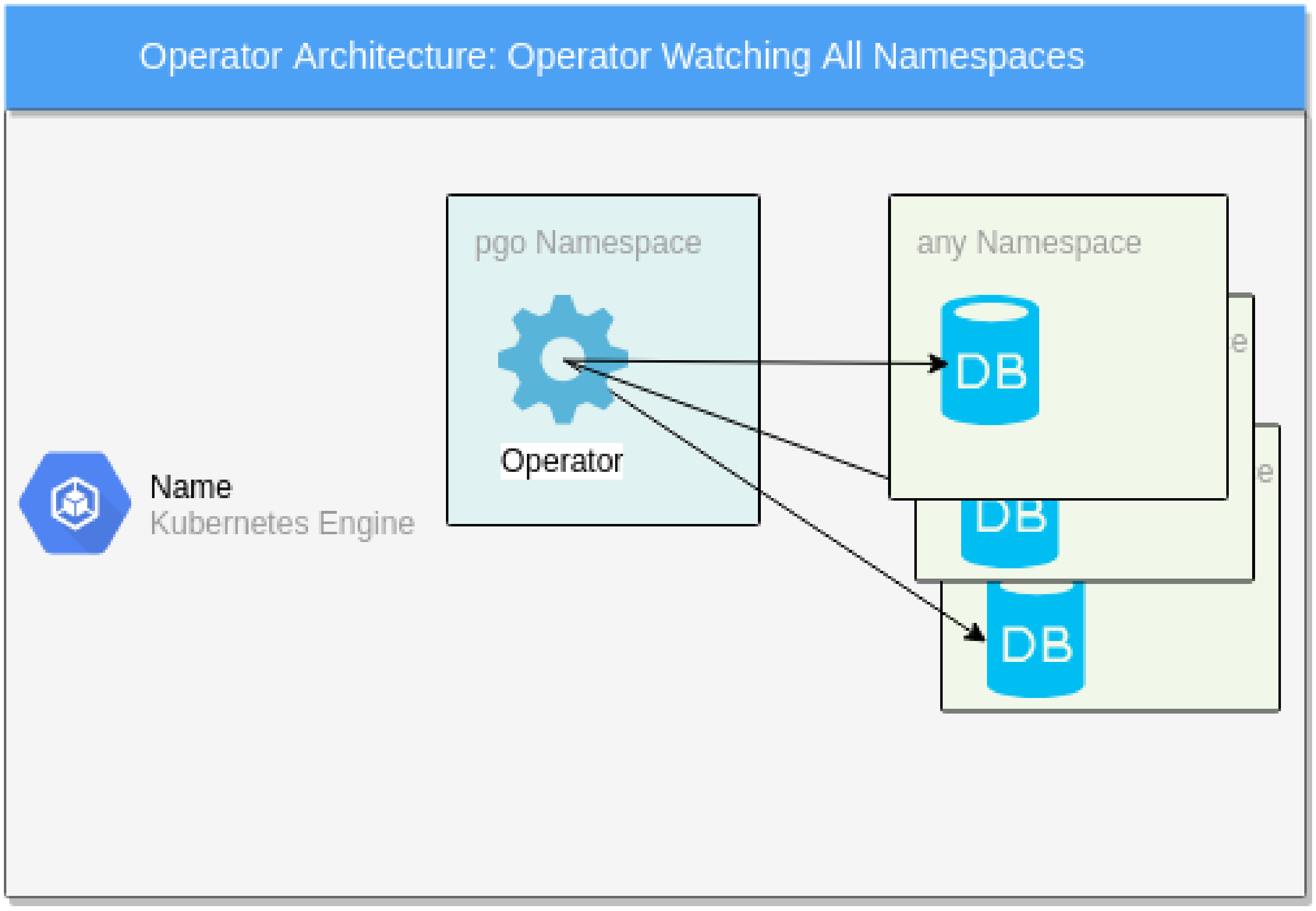

Figure 9: Reference

```
serviceaccount/pgo-backrest
secret/pgo-backrest-repo-config
role/pgo-role
rolebinding/pgo-role-binding
role/pgo-backrest-role
rolebinding/pgo-backrest-role-binding
```

When you run the install-rbac.sh script, it iterates through the list of namespaces to be watched and creates these resources into each of those namespaces.

If you need to add a new namespace that the Operator will watch after an initial execution of install-rbac.sh, you will need to run the following for each new namespace:

```
create-target-rbac.sh YOURNEWNAMESPACE $PGO_OPERATOR_NAMESPACE
```

The example deployment creates the following RBAC structure on your Kube system after running the install scripts:

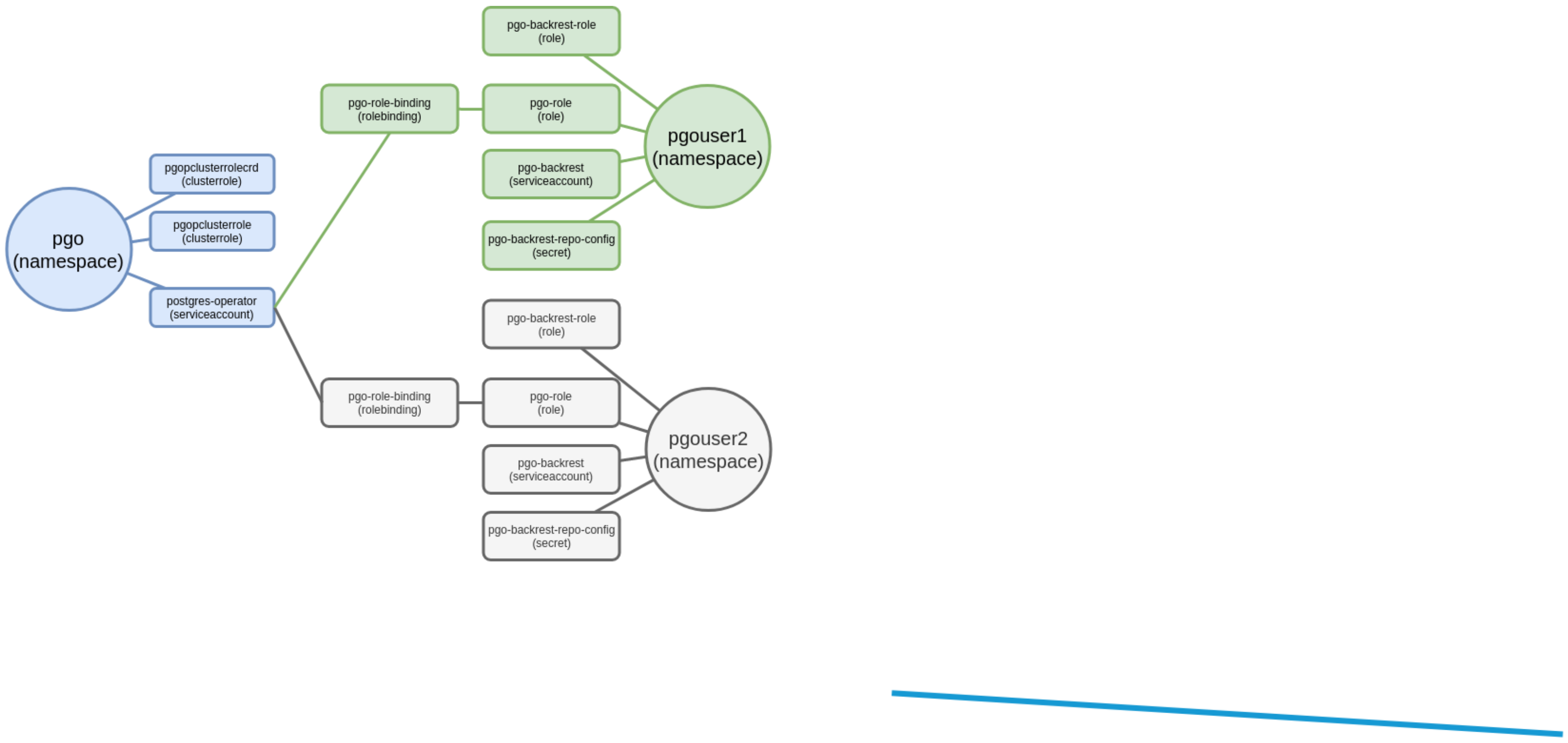

Figure 10: Reference

pgo Clients and Namespaces

The *pgo* CLI now is required to identify which namespace it wants to use when issuing commands to the Operator.

Users of *pgo* can either create a PGO_NAMESPACE environment variable to set the namespace in a persistent manner or they can specify it on the *pgo* command line using the *–namespace* flag.

If a pgo request doe not contain a valid namespace the request will be rejected.

Upgrading Postgres Operator 3.5 Minor Versions

This procedure will give instructions on how to upgrade Postgres Operator 3.5 minor releases.

{{% notice info %}}

As with any upgrade, please ensure you have taken recent backups of all relevant data!

{{% / notice %}}

Prerequisites. You will need the following items to complete the upgrade:

- The latest 3.5.X code for the Postgres Operator available

- The latest 3.5.X PGO client binary
- Finally, these instructions assume you are executing from $COROOT in a terminal window and that you are using the same user from your previous installation. This user must also still have admin privileges in your Kubernetes or Openshift environment.

Step 0 Run `pgo show config` and save this output to compare at the end to ensure you don't miss any of your current configuration changes.

Step 1 Update environment variables in the bashrc

```
export CO_VERSION=3.5.X
```

If you are pulling your images from the same registry as before this should be the only update to the 3.5.X environment variables.

source the updated bash file:

```
source ~/.bashrc
```

Check to make sure that the correct CO_IMAGE_TAG image tag is being used. With a centos7 base image and version 3.5.X of the operator your image tag will be in the format of `centos7-3.5.4`. Verify this by running echo $CO_IMAGE_TAG.

Step 2 Update the pgo.yaml file in `$COROOT/conf/postgres-operator/pgo.yaml`. Use the config that you saved in Step 0. to make sure that you have updated the settings to match the old config. Confirm that the yaml file includes the correct images for the version that you are upgrading to:

For Example:

```
CCPImageTag: centos7-10.9-2.3.3
COImageTag: centos7-3.5.4
```

Step 3 Install the 3.5.X Operator:

```
make deployoperator
```

Verify the Operator is running:

```
kubectl get pod -n <operator namespace>
```

Step 4 Update the PGO client binary to 3.5.X by replacing the binary file with the new one. Run which pgo to ensure you are replacing the current binary.

Step 5 Make sure that any and all configuration changes have been updated.
Run:

```
pgo show config
```

This will print out the current configuration that the operator is using. Ensure you made any configuration changes required, you can compare this output with Step 0 to ensure no settings are missed. If you happened to miss a setting, update the pgo.yaml file and rerun make deployoperator

Step 6 The Operator is now upgraded to 3.5.X. Verify this by running:

```
pgo version
```

Postgres Operator Container Upgrade Procedure

At this point, the Operator should be running the latest minor version of 3.5, and new clusters will be built using the appropriate specifications defined in your pgo.yaml file. For the existing clusters, upgrades can be performed with the following steps.

{{% notice info %}}

Before beginning your upgrade procedure, be sure to consult the [Compatibility Requirements Page] ({{< relref "configuration/compatibility.md" >}}) for container dependency information.

{{% / notice %}}

First, update the deployment of each replica, one at a time, with the new image version:

```
kubectl edit deployment.apps/yourcluster
```

then edit the line containing the image value, which will be similar to the following

```
image: crunchydata/crunchy-postgres:centos7-11.3-2.4.0
```

When this new deployment is written, it will kill the pod and recreate it with the new image. Do this for each replica, waiting for the previous pod to upgrade completely before moving to next.

Once the replicas have been updated, update the deployment of primary by updating the `image:` line in the same fashion, waiting for it to come back up.

Now, similar to the steps above, you will need to update the pgcluster `ccpimagetag:` to the new value:

```
kubectl edit pgcluster yourcluster
```

To check everything is now working as expected, execute

```
pgo test yourcluster
```

To validate the database connections and execute

```
pgo show cluster yourcluster
```

To check the various cluster elements are listed as expected.

There is a bug in the operator where the image version for the backrest repo deployment is not updated with a pgo upgrade. As a workaround for this you need to redeploy the backrest shared repo deployment with the correct image version.

First you will need to get a copy of the yaml file that defines the cluster:

```
kubectl get deployment <cluster-name>-backrest-shared-repo -o yaml >
   <cluster-name>-backrest-repo.yaml
```

You can then edit the yaml file so that the deployment will use the correct image version: edit `<cluster-name>-backrest-repo.yaml`

set to the image, for example:

```
crunchydata/pgo-backrest-repo:centos7-3.5.4
```

Next you will need to delete the current backrest repo deployment and recreate it with the updated yaml:

```
kubectl delete deployment <cluster-name>-backrest-shared-repo
kubectl create -f <cluster-name>-backrest-repo.yaml
```

Verify that the correct images are being used for the cluster. Run `pgo show cluster <cluster-name>` on your cluster and check the version. Describe each of the pods in your cluster and verify that the image that is being used is correct.

```
pgo show cluster <cluster-name>
kubectl get pods
kubectl describe pod <cluster-name>-<id>
kubectl describe pod <cluster-name>-backrest-shared-repo-<id>
```

Finally, make sure that the correct version of pgbackrest is being used and verify backups are working. The versions of pgbackrest that are returned in the primary and backrest pods should match:

```
kubectl get pods
kubectl exec -it <cluster-name>-<id> -- pgbackrest version
kubectl exec -it <cluster-name>-backrest-shared-repo-<id> -- pgbackrest version
pgo backup <cluster-name> --backup-type=pgbackrest
```

You've now completed the upgrade and are running Crunchy PostgreSQL Operator v3.5.X, you can confirm this by running pgo version from the command line and running

```
pgo show cluster <cluster-name>
```

on each cluster. For this minor upgrade, most existing settings and related services (such as pgbouncer, backup schedules and existing policies) are expected to work, but should be tested for functionality and adjusted or recreated as necessary.

Upgrading a Cluster from Version 3.5.x to PGO 4.1

This section will outline the procedure to upgrade a given cluster created using Postgres Operator 3.5.x to PGO version 4.1

{{% notice info %}}

As with any upgrade, please ensure you have taken recent backups of all relevant data!

{{% / notice %}}

Prerequisites. You will need the following items to complete the upgrade:

- The latest PGO 4.1 code for the Postgres Operator available
- The latest PGO 4.1 client binary

Step 0 Create a new Centos/Redhat user with the same permissions are the existing user used to install the Crunchy Postgres Operator. This is necessary to avoid any issues with environment variable differences between 3.5 and 4.1.

Step 1 For the cluster(s) you wish to upgrade, scale down any replicas, if necessary, then delete the cluster

```
pgo delete cluster <clustername>
```

{{% notice warning %}}

Please note the name of each cluster, the namespace used, and be sure not to delete the associated PVCs or CRDs!

{{% /notice %}}

Step 2 Delete the 3.5.x version of the operator by executing:

```
$COROOT/deploy/cleanup.sh
$COROOT/deploy/remove-crd.sh
```

Step 3 Log in as your new Linux user and install the 4.1 Postgres Operator.

Be sure to add the existing namespace to the Operator's list of watched namespaces (see the [Namespace] ({{< relref "gettingstarted/Design/namespace.md" >}}) section of this document for more information) and make sure to avoid overwriting any existing data storage.

Step 4 Once the Operator is installed and functional, create a new 4.1 cluster with the same name as was used previously. This will allow the new cluster to utilize the existing PVCs.

```
pgo create cluster <clustername> -n <namespace>
```

Step 5 Manually update the old leftover Secrets to use the new label as defined in 4.1:

```
kubectl label secret/<clustername>-postgres-secret pg-cluster=<clustername> -n <namespace>
kubectl label secret/<clustername>-primaryuser-secret pg-cluster=<clustername> -n <namespace>
kubectl label secret/<clustername>-testuser-secret pg-cluster=<clustername> -n <namespace>
```

Step 6 To verify cluster status, run pgo test -n Output should be similar to:

```
psql -p 5432 -h 10.104.74.189 -U postgres postgres is Working
psql -p 5432 -h 10.104.74.189 -U postgres userdb is Working
psql -p 5432 -h 10.104.74.189 -U primaryuser postgres is Working
psql -p 5432 -h 10.104.74.189 -U primaryuser userdb is Working
psql -p 5432 -h 10.104.74.189 -U testuser postgres is Working
psql -p 5432 -h 10.104.74.189 -U testuser userdb is Working
```

Step 7 Scale up to the required number of replicas, as needed.

Postgres Operator Ansible Upgrade Procedure from 4.0.1 to 4.1.0

This procedure will give instructions on how to upgrade to Postgres Operator 4.1.0 when using the Ansible installation method.

{{% notice info %}}

As with any upgrade, please ensure you have taken recent backups of all relevant data!

{{% / notice %}}

Prerequisites. You will need the following items to complete the upgrade:

- The latest 4.1.0 code for the Postgres Operator available

These instructions assume you are executing in a terminal window and that your user has admin privileges in your Kubernetes or Openshift environment.

Step 0 For the cluster(s) you wish to upgrade, scale down any replicas, if necessary (see `pgo scaledown --help` for more information on command usage) page for more information), then delete the cluster

```
pgo delete cluster <clustername>
```

{{% notice warning %}}

Please note the name of each cluster, the namespace used, and be sure not to delete the associated PVCs or CRDs!

{{% /notice %}}

Step 1 Save a copy of your current inventory file with a new name (such as `inventory.backup)` and checkout the latest 4.1 tag of the Postgres Operator.

Step 2 Update the new inventory file with the appropriate values for your new Operator installation, as described in the [Ansible Install Prerequisites] ({{< relref "installation/install-with-ansible/prerequisites.md" >}}) and the [Compatibility Requirements Guide] ({{< relref "configuration/compatibility.md" >}}).

Step 3 Now you can upgrade your Operator installation and configure your connection settings as described in the [Ansible Update Page] ({{< relref "installation/install-with-ansible/updating-operator.md" >}}).

Step 4 Verify the Operator is running:

```
kubectl get pod -n <operator namespace>
```

And that it is upgraded to the appropriate version

```
pgo version
```

Step 5 Once the Operator is installed and functional, create new 4.1 clusters with the same name as was used previously. This will allow the new clusters to utilize the existing PVCs.

```
pgo create cluster <clustername> -n <namespace>
```

Step 6 To verify cluster status, run pgo test -n Output should be similar to:

```
psql -p 5432 -h 10.104.74.189 -U postgres postgres is Working
psql -p 5432 -h 10.104.74.189 -U postgres userdb is Working
psql -p 5432 -h 10.104.74.189 -U primaryuser postgres is Working
psql -p 5432 -h 10.104.74.189 -U primaryuser userdb is Working
psql -p 5432 -h 10.104.74.189 -U testuser postgres is Working
psql -p 5432 -h 10.104.74.189 -U testuser userdb is Working
```

Step 7 Scale up to the required number of replicas, as needed.

Postgres Operator Bash Upgrade Procedure from 4.0.1 to 4.1.0

This procedure will give instructions on how to upgrade to Postgres Operator 4.1.0 when using the Bash installation method.

{{% notice info %}}

As with any upgrade, please ensure you have taken recent backups of all relevant data!

{{% / notice %}}

Prerequisites. You will need the following items to complete the upgrade:

- The latest 4.1.0 code for the Postgres Operator available
- The latest 4.1.0 PGO client binary

Finally, these instructions assume you are executing from $PGOROOT in a terminal window and that your user has admin privileges in your Kubernetes or Openshift environment.

Step 0 You will most likely want to run:

```
pgo show config -n <any watched namespace>
```

Save this output to compare once the procedure has been completed to ensure none of the current configuration changes are missing.

Step 1 For the cluster(s) you wish to upgrade, scale down any replicas, if necessary (see `pgo scaledown --help` for more information on command usage) page for more information), then delete the cluster

```
pgo delete cluster <clustername>
```

{{% notice warning %}}

Please note the name of each cluster, the namespace used, and be sure not to delete the associated PVCs or CRDs!

{{% /notice %}}

Step 2 Delete the 4.0.1 version of the Operator by executing:

```
$PGOROOT/deploy/cleanup.sh
$PGOROOT/deploy/remove-crd.sh
$PGOROOT/deploy/cleanup-rbac.sh
```

Step 3 Update environment variables in the bashrc:

```
export PGO_VERSION=4.1.0
```

If you are pulling your images from the same registry as before this should be the only update to the existing 4.0.1 environment variables.

You will need the following new environment variables:

```
# PGO_INSTALLATION_NAME is the unique name given to this Operator install
# this supports multi-deployments of the Operator on the same Kube cluster
export PGO_INSTALLATION_NAME=devtest

# for setting the pgo apiserver port, disabling TLS or not verifying TLS
# if TLS is disabled, ensure setip() function port is updated and http is used in place of https
export PGO_APISERVER_PORT=8443          # Defaults: 8443 for TLS enabled, 8080 for TLS disabled
export DISABLE_TLS=false
export TLS_NO_VERIFY=false

# for disabling the Operator eventing
export DISABLE_EVENTING=false
```

There is a new eventing feature in 4.1.0, so if you want an alias to look at the eventing logs you can add the following:

```
elog () {
$PGO_CMD  -n "$PGO_OPERATOR_NAMESPACE" logs `$PGO_CMD  -n "$PGO_OPERATOR_NAMESPACE" get pod
   --selector=name=postgres-operator -o jsonpath="{.items[0].metadata.name}"` -c event
}
```

Finally source the updated bash file:

```
source ~/.bashrc
```

Step 4 Ensure you have checked out the latest 4.1.0 version of the source code and update the pgo.yaml file in `$PGOROOT/conf/postgres-op`

You will want to use the 4.1.0 pgo.yaml file and update custom settings such as image locations, storage, and resource configs.

Step 5 Create an initial Operator Admin user account. You will need to edit the `$PGOROOT/deploy/install-bootstrap-creds.sh` file to configure the username and password that you want for the Admin account. The default values are:

```
export PGOADMIN_USERNAME=pgoadmin
export PGOADMIN_PASSWORD=examplepassword
```

You will need to update the `$HOME/.pgouser`file to match the values you set in order to use the Operator. Additional accounts can be created later following the steps described in the 'Operator Security' section of the main [Bash Installation Guide] ({{< relref "installation/operator-install.md" >}}). Once these accounts are created, you can change this file to login in via the PGO CLI as that user.

Step 6 Install the 4.1.0 Operator:

Setup the configured namespaces:

```
make setupnamespaces
```

Install the RBAC configurations:

```
make installrbac
```

Deploy the Postgres Operator:

```
make deployoperator
```

Verify the Operator is running:

```
kubectl get pod -n <operator namespace>
```

Step 7 Next, update the PGO client binary to 4.1.0 by replacing the existing 4.0 binary with the latest 4.1.0 binary available.

You can run:

```
which pgo
```

to ensure you are replacing the current binary.

Step 8 You will want to make sure that any and all configuration changes have been updated. You can run:

```
pgo show config -n <any watched namespace>
```

This will print out the current configuration that the Operator will be using.

To ensure that you made any required configuration changes, you can compare with Step 0 to make sure you did not miss anything. If you happened to miss a setting, update the pgo.yaml file and rerun:

```
make deployoperator
```

Step 9 The Operator is now upgraded to 4.1.0 and all users and roles have been recreated. Verify this by running:

```
pgo version
```

Step 10 Once the Operator is installed and functional, create new 4.1 clusters with the same name as was used previously. This will allow the new clusters to utilize the existing PVCs.

```
pgo create cluster <clustername> -n <namespace>
```

Step 11 To verify cluster status, run pgo test -n Output should be similar to:

```
psql -p 5432 -h 10.104.74.189 -U postgres postgres is Working
psql -p 5432 -h 10.104.74.189 -U postgres userdb is Working
psql -p 5432 -h 10.104.74.189 -U primaryuser postgres is Working
psql -p 5432 -h 10.104.74.189 -U primaryuser userdb is Working
psql -p 5432 -h 10.104.74.189 -U testuser postgres is Working
psql -p 5432 -h 10.104.74.189 -U testuser userdb is Working
```

Step 12 Scale up to the required number of replicas, as needed.

Upgrading Postgres Operator from 4.1.0 to a patch release

This procedure will give instructions on how to upgrade Postgres Operator 4.1 patch releases.

{{% notice info %}}

As with any upgrade, please ensure you have taken recent backups of all relevant data!

{{% / notice %}}

Prerequisites You will need the following items to complete the upgrade:

- The latest 4.1.X code for the Postgres Operator available
- The latest 4.1.X PGO client binary
- Finally, these instructions assume you are executing from $COROOT in a terminal window and that you are using the same user from your previous installation. This user must also still have admin privileges in your Kubernetes or Openshift environment.

Step 1 Run `pgo show config` and save this output to compare at the end to ensure you do not miss any of your current configuration changes.

Step 2 Update environment variables in the `.bashrc` file:

```
export CO_VERSION=4.1.X
```

If you are pulling your images from the same registry as before this should be the only update to the 4.1.X environment variables.

source the updated bash file:

```
source ~/.bashrc
```

Check to make sure that the correct CO_IMAGE_TAG image tag is being used. With a centos7 base image and version 4.1.X of the operator your image tag will be in the format of `centos7-4.1.1`. Verify this by running echo `$CO_IMAGE_TAG`.

Step 3 Update the pgo.yaml file in `$COROOT/conf/postgres-operator/pgo.yaml`. Use the config that you saved in Step 1. to make sure that you have updated the settings to match the old config. Confirm that the yaml file includes the correct images for the updated version.

For example, to update to versions 4.1.1:

```
CCPImageTag: centos7-11.6-4.1.1
COImageTag: centos7-4.1.1
```

Step 4 Install the 4.1.X Operator:

```
make deployoperator
```

Verify the Operator is running:

```
kubectl get pod -n <operator namespace>
```

Step 5 Update the `pgo` client binary to 4.1.x by replacing the binary file with the new one.

Run `which pgo` to ensure you are replacing the current binary.

Step 6 Make sure that any and all configuration changes have been updated. Run:

```
pgo show config
```

This will print out the current configuration that the operator is using. Ensure you made any configuration changes required, you can compare this output with Step 1 to ensure no settings are missed. If you happened to miss a setting, update the `pgo.yaml` file and rerun `make deployoperator`.

Step 7 The Postgres Operator is now upgraded to 4.1.X.

Verify this by running:

```
pgo version
```

Postgres Operator Container Upgrade Procedure

At this point, the Operator should be running the latest minor version of 4.1, and new clusters will be built using the appropriate specifications defined in your pgo.yaml file. For the existing clusters, upgrades can be performed with the following steps.

{{% notice info %}}

Before beginning your upgrade procedure, be sure to consult the [Compatibility Requirements Page]({{< relref "configuration/compatibility.md" >}}) for container dependency information.

{{% / notice %}}

You can upgrade each cluster using the following command:

```
pgo upgrade -n <clusternamespace> --ccp-image-tag=centos7-11.6-4.1.1 <clustername>
```

This process takes a few momnets to complete.

To check everything is now working as expected, execute:

```
pgo test yourcluster
```

To check the various cluster elements are listed as expected:

```
pgo show cluster -n <clusternamespace> <clustername>
```

You've now completed the upgrade and are running Crunchy PostgreSQL Operator v4.1.X! For this minor upgrade, most existing settings and related services (such as pgBouncer, backup schedules and existing policies) are expected to work, but should be tested for functionality and adjusted or recreated as necessary.

Upgrading to Version 3.5.0 From Previous Versions

This procedure will give instructions on how to upgrade to Postgres Operator 3.5

{{% notice info %}}

As with any upgrade, please ensure you have taken recent backups of all relevant data!

{{% / notice %}}

For clusters created in prior versions that used pgbackrest, you will be required to first create a pgbasebackup for those clusters.

After upgrading to Operator 3.5, you will need to restore those clusters from the pgbasebackup into a new cluster with `--pgbackrest` enabled. This is due to the new pgbackrest shared repository being implemented in 3.5. This is a breaking change for anyone that used pgbackrest in Operator versions prior to 3.5.

The pgingest CRD is removed in Operator 3.5. You will need to manually remove it from any deployments of the operator after upgrading to this version. This includes removing ingest related permissions from the pgorole file. Additionally, the API server now removes the ingest related API endpoints.

Primary and replica labels are only applicable at cluster creation and are not updated after a cluster has executed a failover. A new service-name label is applied to PG cluster components to indicate whether a deployment/pod is a primary or replica. service-name is also the label now used by the cluster services to route with. This scheme allows for an almost immediate failover promotion and avoids the pod having to be bounced as part of a failover. Any existing PostgreSQL clusters will need to be updated to specify them as a primary or replica using the new service-name labeling scheme.

The autofail label was moved from deployments and pods to just the pgcluster CRD to support autofail toggling.

The storage configurations in pgo.yaml support the MatchLabels attribute for NFS storage. This will allow users to have more than a single NFS backend,. When set, this label (key=value) will be used to match the labels on PVs when a PVC is created.

The UpdateCluster permission was added to the sample pgorole file to support the new pgo update CLI command. It was also added to the pgoperm file.

The pgo.yaml adds the PreferredFailoverNode setting. This is a Kubernetes selector string (e.g. key=value). This value if set, will cause fail-over targets to be preferred based on the node they run on if that node is in the set of *preferred*.

The ability to select nodes based on a selector string was added. For this to feature to be used, multiple replicas have to be in a ready state, and also at the same replication status. If those conditions are not met, the default fail-over target selection is used.

The pgo.yaml file now includes a new storage configuration, XlogStorage, which when set will cause the xlog volume to be allocated using this storage configuration. If not set, the PrimaryStorage configuration will be used.

The pgo.yaml file now includes a new storage configuration, BackrestStorage, will cause the pgbackrest shared repository volume to be allocated using this storage configuration.

The pgo.yaml file now includes a setting, AutofailReplaceReplica, which will enable or disable whether a new replica is created as part of a fail-over. This is turned off by default.

See the GitHub Release notes for the features and other notes about a specific release.

Documentation

The documentation website is generated using Hugo.

Hosting Hugo Locally (Optional)

If you would like to build the documentation locally, view the official Installing Hugo guide to set up Hugo locally.

You can then start the server by running the following commands -

```
cd $PGOROOT/hugo/
hugo server
```

The local version of the Hugo server is accessible by default from *localhost:1313*. Once you've run *hugo server*, that will let you interactively make changes to the documentation as desired and view the updates in real-time.

Contributing to the Documentation

All documentation is in Markdown format and uses Hugo weights for positioning of the pages.

The current production release documentation is updated for every tagged major release.

When you're ready to commit a change, please verify that the documentation generates locally.

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

If you would like to work the issue, please add that information in the issue so that we can confirm we are not already working no need to duplicate efforts.

If you have any question 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. Create a pull request from your fork to the `master` branch.
2. Update the checklists in the Pull Request Description.
3. Reference which issues this Pull Request is resolving.

TLS Configuration

Should you desire to alter the default TLS settings for the Postgres Operator, you can set the following variables as described below.

Server Settings

To disable TLS and make an unsecured connection on port 8080 instead of connecting securely over the default port, 8443, set:

Bash environment variables

```
DISABLE_TLS=true
PGO_APISERVER_PORT=8080
```

Or inventory variables if using Ansible

```
pgo_disable_tls='true'
pgo_apiserver_port=8080
```

To disable TLS verifcation, set the follwing as a Bash environment variable

```
export TLS_NO_VERIFY=false
```

Or the following in the inventory file is using Ansible

```
pgo_tls_no_verify='false'
```

Connection Settings

If TLS authentication has been disabled, or if the Operator's apiserver port is changed, be sure to update the PGO_APISERVER_URL accordingly.

For example with an Ansible installation,

```
export PGO_APISERVER_URL='https://<apiserver IP>:8443'
```

would become

```
export PGO_APISERVER_URL='http://<apiserver IP>:8080'
```

With a Bash installation,

```
setip()
{
   export PGO_APISERVER_URL=https://`$PGO_CMD -n "$PGO_OPERATOR_NAMESPACE" get service
      postgres-operator -o=jsonpath="{.spec.clusterIP}"`:8443
}
```

would become

```
setip()
{
   export PGO_APISERVER_URL=http://`$PGO_CMD -n "$PGO_OPERATOR_NAMESPACE" get service
      postgres-operator -o=jsonpath="{.spec.clusterIP}"`:8080
}
```

Client Settings

Finally, if TLS has been disabled for the Operator's apiserver, the PGO client connection must be set to match the given settings.

Two options are available, either the Bash environment variable

```
DISABLE_TLS=true
```

must be configured, or the –disable-tls flag must be included when using the client, i.e.

```
pgo version --disable-tls
```

Operator Lifecycle Management Support

The PostgreSQL Operator supports Red Hat OLM (Operator Lifecycle Manager) to a degree starting with the PostgreSQL Operator 4.0 release.

The PostgreSQL Operator supports the different deployment models as documented here

The PostgreSQL Operator is available for download in OperatorHub.io

Prerequisites

The following is required prior to installing PostgreSQL Operator:

- Kubernetes v1.13+
- Red Hat OpenShift v3.11+
- VMWare Enterprise PKS 1.3+
- `kubectl` or `oc` configured to communicate with Kubernetes

Container Ports

The API server port is required to connect to the API server with the `pgo` cli. The `nsqd` and `nsqadmin` ports are required to connect to the event stream and listen for real-time events.

Container	Port
API Server	8443
nsqadmin	4151
nsqd	4150

Service Ports

This is a list of service ports that are used in the PostgreSQL Operator. Verify that the ports are open and can be used.

Service	Port
PostgreSQL	5432
pgpool	5432
pgbouncer	5432
pgbackrest	2022
postgres-exporter	9187

Application Ports

This is a list of ports used by application containers that connect to the PostgreSQL Operator. If you are using one of these apps, verify that the service port for that app is open and can be used.

App	Port
pgbadger	10000
Grafana	3000
Prometheus	9090

For various scripts used by the Operator, the `expenv` utility is required as are certain environment variables.

Download the `expenv` utility from its Github Releases page, and place it into your PATH (e.g. $HOME/odev/bin).

The following environment variables are heavily used in the Bash installation procedures and may be used in Operator helper scripts.

Variable	Ansible Inventory	Example
`DISABLE_EVENTING`	pgo_disable_eventing	false
`DISABLE_TLS`	pgo_disable_tls	false
`GOPATH`		$HOME/odev \vert Golangprojectdirectory`GOBIN` \vert$ |GOPATH/bin
`NAMESPACE`	namespace	pgouser1

Variable	Ansible Inventory	Example
`PGOROOT`		$GOPATH/src/github.com/crunchydata/postgres-operator|Operatorreposito$
`PGO_CMD`		kubectl
`PGO_CLIENT_CERT`		$PGOROOT/conf/postgres-operator/server.crt|TLSClientcertificate`PGO_C$
`PGO_IMAGE_PREFIX`	pgo__image__prefix	crunchydata
`PGO_IMAGE_TAG`	pgo__image__tag	$PGO_BASEOS-$PGO_VERSION
`PGO_INSTALLATION_NAME`	pgo__installation__name	devtest
`PGO_OPERATOR_NAMESPACE`	pgo__operator__namespace	pgo
`PGO_VERSION`		4.2.0
`TLS_NO_VERIFY`	pgo__tls__no__verify	false

{{% notice tip %}} `examples/envs.sh` contains the above variable definitions as well {{% /notice %}}

A full installation of the Operator includes the following steps:

- create a project structure
- configure your environment variables
- configure Operator templates
- create security resources
- deploy the operator
- install pgo CLI (end user command tool)

Operator end-users are only required to install the pgo CLI client on their host and can skip the server-side installation steps. pgo CLI clients are provided for Linux, Mac, and Windows clients.

The Operator can be deployed by multiple methods including:

- default installation
- Ansible playbook installation
- Openshift Console installation using OLM

Default Installation - Create Project Structure

The Operator follows a golang project structure, you can create a structure as follows on your local Linux host:

```
mkdir -p $HOME/odev/src/github.com/crunchydata $HOME/odev/bin $HOME/odev/pkg
cd $HOME/odev/src/github.com/crunchydata
git clone https://github.com/CrunchyData/postgres-operator.git
cd postgres-operator
git checkout v4.2.0
```

This creates a directory structure under your HOME directory name *odev* and clones the current Operator version to that structure.

Default Installation - Configure Environment

Environment variables control aspects of the Operator installation. You can copy a sample set of Operator environment variables and aliases to your *.bashrc* file to work with.

```
cat $HOME/odev/src/github.com/crunchydata/postgres-operator/examples/envs.sh >> $HOME/.bashrc
source $HOME/.bashrc
```

To manually configure the environment variables, refer to the [environment documentation]({{< relref "common-env.md" >}}).

For various scripts used by the Operator, the *expenv* utility is required, download this utility from the Github Releases page, and place it into your PATH (e.g. $HOME/odev/bin). {{% notice tip %}}There is also a Makefile target that includes is *expenv* and several other dependencies that are only needed if you plan on building from source:

```
make setup
```

{{% /notice %}}

Default Installation - Namespace Creation

The default installation will create 3 namespaces to use for deploying the Operator into and for holding Postgres clusters created by the Operator.

Creating Kube namespaces is typically something that only a priviledged Kube user can perform so log into your Kube cluster as a user that has the necessary priviledges.

On Openshift if you do not want to install the Operator as the system administrator, you can grant cluster-admin priviledges to a user as follows:

```
oc adm policy add-cluster-role-to-user cluster-admin pgoinstaller
```

In the above command, you are granting cluster-admin priviledges to a user named pgoinstaller.

The *NAMESPACE* environment variable is a comma separated list of namespaces that specify where the Operator will be provisioing PG clusters into, specifically, the namespaces the Operator is watching for Kube events. This value is set as follows:

```
export NAMESPACE=pgouser1,pgouser2
```

This means namespaces called *pgouser1* and *pgouser2* will be created as part of the default installation.

{{% notice warning %}}In Kubernetes versions prior to 1.12 (including Openshift up through 3.11), there is a limitation that requires an extra step during installation for the operator to function properly with watched namespaces. This limitation does not exist when using Kubernetes 1.12+. When a list of namespaces are provided through the NAMESPACE environment variable, the setupnamespaces.sh script handles the limitation properly in both the bash and ansible installation.

However, if the user wishes to add a new watched namespace after installation, where the user would normally use pgo create namespace to add the new namespace, they should instead run the add-targeted-namespace.sh script or they may give themselves cluster-admin privileges instead of having to run setupnamespaces.sh script. Again, this is only required when running on a Kubernetes distribution whose version is below 1.12. In Kubernetes version 1.12+ the pgo create namespace command works as expected.

{{% /notice %}}

The *PGO_OPERATOR_NAMESPACE* environment variable is a comma separated list of namespace values that the Operator itself will be deployed into. For the installation example, this value is set as follows:

```
export PGO_OPERATOR_NAMESPACE=pgo
```

This means a *pgo* namespace will be created and the Operator will be deployed into that namespace.

Create the Operator namespaces using the Makefile target:

```
make setupnamespaces
```

Note: The setupnamespaces target only creates the namespace(s) specified in PGO_OPERATOR_NAMESPACE environment variable

The Design section of this documentation talks further about the use of namespaces within the Operator.

Default Installation - Configure Operator Templates

Within the Operator *conf* directory are several configuration files and templates used by the Operator to determine the various resources that it deploys on your Kubernetes cluster, specifically the PostgreSQL clusters it deploys.

When you install the Operator you must make choices as to what kind of storage the Operator has to work with for example. Storage varies with each installation. As an installer, you would modify these configuration templates used by the Operator to customize its behavior.

Note: when you want to make changes to these Operator templates and configuration files after your initial installation, you will need to re-deploy the Operator in order for it to pick up any future configuration changes.

Here are some common examples of configuration changes most installers would make:

Storage

Inside `conf/postgres-operator/pgo.yaml` there are various storage configurations defined.

```
PrimaryStorage: gce
XlogStorage: gce
ArchiveStorage: gce
BackupStorage: gce
ReplicaStorage: gce
```

```
  gce:
    AccessMode:  ReadWriteOnce
    Size:  1G
    StorageType:  dynamic
    StorageClass:  standard
    Fsgroup:  26
```

Listed above are the *pgo.yaml* sections related to storage choices. *PrimaryStorage* specifies the name of the storage configuration used for PostgreSQL primary database volumes to be provisioned. In the example above, a NFS storage configuration is picked. That same storage configuration is selected for the other volumes that the Operator will create.

This sort of configuration allows for a PostgreSQL primary and replica to use different storage if you want. Other storage settings like *AccessMode*, *Size*, *StorageType*, *StorageClass*, and *Fsgroup* further define the storage configuration. Currently, NFS, HostPath, and Storage Classes are supported in the configuration.

As part of the Operator installation, you will need to adjust these storage settings to suit your deployment requirements. For users wanting to try out the Operator on Google Kubernetes Engine you would make the following change to the storage configuration in pgo.yaml:

For NFS Storage, it is assumed that there are sufficient Persistent Volumes (PV) created for the Operator to use when it creates Persistent Volume Claims (PVC). The creation of Persistent Volumes is something a Kubernetes cluster-admin user would typically provide before installing the Operator. There is an example script which can be used to create NFS Persistent Volumes located here:

```
./pv/create-nfs-pv.sh
```

That script looks for the IP address of an NFS server using the environment variable PGO_NFS_IP you would set in your .bashrc environment.

A similar script is provided for HostPath persistent volume creation if you wanted to use HostPath for testing:

```
./pv/create-pv.sh
```

Adjust the above PV creation scripts to suit your local requirements, the purpose of these scripts are solely to produce a test set of Volume to test the Operator.

Other settings in *pgo.yaml* are described in the pgo.yaml Configuration section of the documentation.

Operator Security

The Operator implements its own RBAC (Role Based Access Controls) for authenticating Operator users access to the Operator REST API.

A default admin user is created when the operator is deployed. Create a .pgouser in your home directory and insert the text from below:

```
pgoadmin:examplepassword
```

The format of the .pgouser client file is:

```
<username>:<password>
```

To create a unique administrator user on deployment of the operator edit this file and update the .pgouser file accordingly:

```
$PGOROOT/deploy/install-bootstrap-creds.sh
```

After installation users can create optional Operator users as follows:

```
pgo create pgouser someuser --pgouser-namespaces="pgouser1,pgouser2"
   --pgouser-password=somepassword --pgouser-roles="somerole,someotherrole"
```

Note, you can also store the pgouser file in alternate locations, see the Security documentation for details.

Operator security is discussed in the Security section Security of the documentation.

Adjust these settings to meet your local requirements.

Default Installation - Create Kube RBAC Controls

The Operator installation requires Kubernetes administrators to create Resources required by the Operator. These resources are only allowed to be created by a cluster-admin user. To install on Google Cloud, you will need a user account with cluster-admin priviledges. If you own the GKE cluster you are installing on, you can add cluster-admin role to your account as follows:

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

Specifically, Custom Resource Definitions for the Operator, and Service Accounts used by the Operator are created which require cluster permissions.

Tor create the Kube RBAC used by the Operator, run the following as a cluster-admin Kube user:

```
make installrbac
```

This set of Resources is created a single time unless a new Operator release requires these Resources to be recreated. Note that when you run *make installrbac* the set of keys used by the Operator REST API and also the pgbackrest ssh keys are generated.

Verify the Operator Custom Resource Definitions are created as follows:

```
kubectl get crd
```

You should see the *pgclusters* CRD among the listed CRD resource types.

See the Security documentation for a description of the various RBAC resources created and used by the Operator.

Default Installation - Deploy the Operator

At this point, you as a normal Kubernetes user should be able to deploy the Operator. To do this, run the following Makefile target:

```
make deployoperator
```

This will cause any existing Operator to be removed first, then the configuration to be bundled into a ConfigMap, then the Operator Deployment to be created.

This will create a postgres-operator Deployment and a postgres-operator Service.Operator administrators needing to make changes to the Operator configuration would run this make target to pick up any changes to pgo.yaml, pgo users/roles, or the Operator templates.

Default Installation - Completely Cleaning Up

You can completely remove all the namespaces you have previously created using the default installation by running the following:

```
make cleannamespaces
```

This will permanently delete each namespace the Operator installation created previously.

pgo CLI Installation

Most users will work with the Operator using the *pgo* CLI tool. That tool is downloaded from the GitHub Releases page for the Operator (https://github.com/crunchydata/postgres-operator/releases). Crunchy Enterprise Customer can download the pgo binaries from https://access.crunchydata.com/ on the downloads page.

The *pgo* client is provided in Mac, Windows, and Linux binary formats, download the appropriate client to your local laptop or workstation to work with a remote Operator.

{{% notice info %}}

If TLS authentication was disabled during installation, please see the [TLS Configuration Page] ({{< relref "gettingstarted/Design/tls.md" >}}) for additional configuration information.

{{% / notice %}}

Prior to using *pgo*, users testing the Operator on a single host can specify the *postgres-operator* URL as follows:

```
    $ kubectl get service postgres-operator -n pgo
    NAME                CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
    postgres-operator   10.104.47.110   <none>        8443/TCP   7m
    $ export PGO_APISERVER_URL=https://10.104.47.110:8443
    pgo version
```

That URL address needs to be reachable from your local *pgo* client host. Your Kubernetes administrator will likely need to create a network route, ingress, or LoadBalancer service to expose the Operator REST API to applications outside of the Kubernetes cluster. Your Kubernetes administrator might also allow you to run the Kubernetes port-forward command, contact your adminstrator for details.

Next, the *pgo* client needs to reference the keys used to secure the Operator REST API:

```
    export PGO_CA_CERT=$PGOROOT/conf/postgres-operator/server.crt
    export PGO_CLIENT_CERT=$PGOROOT/conf/postgres-operator/server.crt
    export PGO_CLIENT_KEY=$PGOROOT/conf/postgres-operator/server.key
```

You can also specify these keys on the command line as follows:

```
pgo version --pgo-ca-cert=$PGOROOT/conf/postgres-operator/server.crt
   --pgo-client-cert=$PGOROOT/conf/postgres-operator/server.crt
   --pgo-client-key=$PGOROOT/conf/postgres-operator/server.key
```

{{% notice tip %}} if you are running the Operator on Google Cloud, you would open up another terminal and run *kubectl port-forward …* to forward the Operator pod port 8443 to your localhost where you can access the Operator API from your local workstation. {{% /notice %}}

At this point, you can test connectivity between your laptop or workstation and the Postgres Operator deployed on a Kubernetes cluster as follows:

```
pgo version
```

You should get back a valid response showing the client and server version numbers.

Verify the Installation

Now that you have deployed the Operator, you can verify that it is running correctly.

You should see a pod running that contains the Operator:

```
kubectl get pod --selector=name=postgres-operator -n pgo
NAME                                 READY     STATUS    RESTARTS   AGE
postgres-operator-79bf94c658-zczf6   3/3       Running   0          47s
```

That pod should show 3 of 3 containers in *running* state and that the operator is installed into the *pgo* namespace.

The sample environment script, examples/env.sh, if used creates some bash functions that you can use to view the Operator logs. This is useful in case you find one of the Operator containers not in a running status.

Using the pgo CLI, you can verify the versions of the client and server match as follows:

```
pgo version
```

This also tests connectivity between your pgo client host and the Operator server.

Developing

The Postgres-Operator is an open source project hosted on GitHub.

Developers that wish to build the Operator from source or contribute to the project via pull requests would set up a development environment through the following steps.

Create Kubernetes Cluster

We use either OpenShift Container Platform or kubeadm to install development clusters.

Create a Local Development Host

We currently build on CentOS 7 and RHEL 7 hosts. Others operating systems are possible, however we do not support building or running the Operator on other operating systems at this time.

Perform Manual Install

You can follow the manual installation method described in this documentation to make sure you can deploy from your local development host to your Kubernetes cluster.

Build Locally

You can now build the Operator from source on local on your development host. Here are some steps to follow:

Get Build Dependencies

Run the following target to install a golang compiler, and any other build dependencies:

```
make setup
```

Compile

You will build all the Operator binaries and Docker images by running:

```
make all
```

This assumes you have Docker installed and running on your development host.

By default, the Makefile will use buildah to build the container images, to override this default to use docker to build the images, set the IMGBUILDER variable to `docker`

The project uses the golang dep package manager to vendor all the golang source dependencies into the *vendor* directory. You typically do not need to run any *dep* commands unless you are adding new golang package dependencies into the project outside of what is within the project for a given release.

After a full compile, you will have a *pgo* binary in `$HOME/odev/bin` and the Operator images in your local Docker registry.

Release

You can perform a release build by running:

```
make release
```

This will compile the Mac and Windows versions of *pgo*.

Deploy

Now that you have built the Operator images, you can push them to your Kubernetes cluster if that cluster is remote to your development host.

You would then run:

```
make deployoperator
```

To deploy the Operator on your Kubernetes cluster. If your Kubernetes cluster is not local to your development host, you will need to specify a config file that will connect you to your Kubernetes cluster. See the Kubernetes documentation for details.

Debug

Debug level logging in turned on by default when deploying the Operator.

Sample bash functions are supplied in *examples/envs.sh* to view the Operator logs.

You can view the Operator REST API logs with the *alog* bash function.

You can view the Operator core logic logs with the *olog* bash function.

You can view the Scheduler logs with the *slog* bash function.

These logs contain the following details:

```
Timestamp
Logging Level
Message Content
Function Information
File Information
PGO version
```

Additionally, you can view the Operator deployment Event logs with the *elog* bash function.

You can enable the *pgo* CLI debugging with the following flag:

```
pgo version --debug
```

You can set the REST API URL as follows after a deployment if you are developing on your local host by executing the *setip* bash function.

Crunchy Data PostgreSQL Operator Playbooks

The Crunchy Data PostgreSQL Operator Playbooks contain Ansible roles for installing and managing the Crunchy Data PostgreSQL Operator.

Features

The playbooks provided allow users to:

- install PostgreSQL Operator on Kubernetes and OpenShift
- install PostgreSQL Operator from a Linux, Mac or Windows(Ubuntu subsystem)host
- generate TLS certificates required by the PostgreSQL Operator
- configure PostgreSQL Operator settings from a single inventory file
- support a variety of deployment models

Resources

- Ansible
- Crunchy Data
- Crunchy Data PostgreSQL Operator Documentation
- Crunchy Data PostgreSQL Operator Project

Prerequisites

The following is required prior to installing Crunchy PostgreSQL Operator using Ansible:

- postgres-operator playbooks source code for the target version
- Ansible 2.5+

Kubernetes Installs

- Kubernetes v1.11+
- Cluster admin privileges in Kubernetes
- kubectl configured to communicate with Kubernetes

OpenShift Installs

- OpenShift v3.09+
- Cluster admin privileges in OpenShift
- oc configured to communicate with OpenShift

Installing from a Windows Host

If the Crunchy PostgreSQL Operator is being installed from a Windows host the following are required:

- Windows Subsystem for Linux (WSL)
- Ubuntu for Windows

Environment

Ensure the appropriate [environment variables]({{< relref "common-env.md" >}}) are set.

Permissions

The installation of the Crunchy PostgreSQL Operator requires elevated privileges.
It is required that the playbooks are run as a `cluster-admin` to ensure the playbooks can install:

- Custom Resource Definitions
- Cluster RBAC
- Create required namespaces

{{% notice warning %}}In Kubernetes versions prior to 1.12 (including Openshift up through 3.11), there is a limitation that requires an extra step during installation for the operator to function properly with watched namespaces. This limitation does not exist when using Kubernetes 1.12+. When a list of namespaces are provided through the NAMESPACE environment variable, the setupnamespaces.sh script handles the limitation properly in both the bash and ansible installation.

However, if the user wishes to add a new watched namespace after installation, where the user would normally use pgo create namespace to add the new namespace, they should instead run the add-targeted-namespace.sh script or they may give themselves cluster-admin privileges instead of having to run setupnamespaces.sh script. Again, this is only required when running on a Kubernetes distribution whose version is below 1.12. In Kubernetes version 1.12+ the pgo create namespace command works as expected.

{{% /notice %}}

Obtaining Operator Ansible Role

There are two ways to obtain the Crunchy PostgreSQL Operator Roles:

- Clone the postgres-operator project

- `postgres-operator-playbooks` RPM provided for Crunchy customers via the Crunchy Access Portal.

GitHub Installation

All necessary files (inventory, main playbook and roles) can be found in the `ansible` directory in the postgres-operator project.

RPM Installation using Yum

Available to Crunchy customers is an RPM containing all the necessary Ansible roles and files required for installation using Ansible. The RPM can be found in Crunchy's yum repository. For information on setting up `yum` to use the Crunchy repoistory, see the Crunchy Access Portal.

To install the Crunchy PostgreSQL Operator Ansible roles using `yum`, run the following command on a RHEL or CentOS host:

```
sudo yum install postgres-operator-playbooks
```

- Ansible roles can be found in: `/usr/share/ansible/roles/crunchydata`
- Ansible playbooks/inventory files can be found in: `/usr/share/ansible/postgres-operator/playbooks`

Once installed users should take a copy of the `inventory` file included in the installation using the following command:

```
cp /usr/share/ansible/postgres-operator/playbooks/inventory ${HOME?}
```

Configuring the Inventory File

The `inventory` file included with the PostgreSQL Operator Playbooks allows installers to configure how the operator will function when deployed into Kubernetes. This file should contain all configurable variables the playbooks offer.

Requirements

The following configuration parameters must be set in order to deploy the Crunchy PostgreSQL Operator.

Additionally, `storage` variables will need to be defined to provide the Crunchy PostgreSQL Operator with any required storage configuration. Guidance for defining `storage` variables can be found further in this documentation.

{{% notice tip %}} You should remove or comment out variables either either the `kubernetes` or `openshift` variables if you are not being using them for your environment. Both sets of variables cannot be used at the same time. {{% /notice %}}

- `archive_mode`
- `archive_timeout`
- `auto_failover`
- `auto_failover_sleep_secs`
- `auto_failover_replace_replica`
- `backup_storage`
- `backrest`
- `backrest_storage`
- `badger`
- `ccp_image_prefix`
- `ccp_image_tag`
- `create_rbac`
- `db_name`
- `db_password_age_days`
- `db_password_length`
- `db_port`
- `db_replicas`
- `db_user`
- `exporterport`
- `kubernetes_context` (Comment out if deploying to am OpenShift environment)
- `metrics`
- `openshift_host` (Comment out if deploying to a Kubernetes environment)
- `openshift_password` (Comment out if deploying to a Kubernetes environment)
- `openshift_skip_tls_verify` (Comment out if deploying to a Kubernetes environment)
- `openshift_token` (Comment out if deploying to a Kubernetes environment)
- `openshift_user` (Comment out if deploying to a Kubernetes environment)
- `pgbadgerport`
- `pgo_admin_password`
- `pgo_admin_perms`
- `pgo_admin_role_name`
- `pgo_admin_username`
- `pgo_client_version`
- `pgo_image_prefix`
- `pgo_image_tag`
- `pgo_installation_name`
- `pgo_operator_namespace`
- `primary_storage`
- `replica_storage`
- `scheduler_timeout`

Configuration Parameters

Name	Default
`archive_mode`	true
`archive_timeout`	60
`auto_failover`	false
`auto_failover_replace_replica`	false
`auto_failover_sleep_secs`	9

Name	Default
`backrest`	false
`backrest_aws_s3_bucket`	
`backrest_aws_s3_endpoint`	
`backrest_aws_s3_key`	
`backrest_aws_s3_region`	
`backrest_aws_s3_secret`	
`backrest_storage`	storageos
`backup_storage`	storageos
`badger`	false
`ccp_image_prefix`	crunchydata
`ccp_image_tag`	
`cleanup`	false
`create_rbac`	true
`crunchy_debug`	false
`delete_metrics_namespace`	false
`delete_operator_namespace`	false
`delete_watched_namespaces`	false
`db_name`	userdb
`db_password_age_days`	60
`db_password_length`	20
`db_port`	5432
`db_replicas`	1
`db_user`	testuser
`exporterport`	9187
`grafana_admin_password`	
`grafana_admin_username`	admin
`grafana_install`	true
`grafana_storage_access_mode`	
`grafana_storage_class_name`	
`grafana_volume_size`	
`kubernetes_context`	
`log_statement`	none
`metrics`	false
`metrics_namespace`	metrics
`namespace`	
`openshift_host`	
`openshift_password`	
`openshift_skip_tls_verify`	
`openshift_token`	
`openshift_user`	
`pgbadgerport`	10000
`pgo_admin_username`	admin
`pgo_admin_password`	
`pgo_admin_perms`	DeleteNamespace,CreateNamespace,UpdatePgorole,ShowPgorole,DeletePgorole,CreatePgorole,Upda

Name	Default
`pgo_admin_role_name`	pgoadmin
`pgo_apiserver_port`	8443
`pgo_client_install`	true
`pgo_client_version`	
`pgo_disable_eventing`	false
`pgo_disable_tls`	false
`pgo_image_prefix`	crunchydata
`pgo_image_tag`	
`pgo_installation_name`	
`pgo_operator_namespace`	
`pgo_tls_no_verify`	false
`primary_storage`	storageos
`prometheus_install`	true
`prometheus_storage_access_mode`	
`prometheus_storage_class_name`	
`replica_storage`	storageos
`scheduler_timeout`	3600
`service_type`	ClusterIP
`pgo_cluster_admin`	false

{{% notice tip %}} To retrieve the `kubernetes_context` value for Kubernetes installs, run the following command:

```
kubectl config current-context
```

{{% /notice %}}

Storage

Kubernetes and OpenShift offer support for a wide variety of different storage types, and by default, the `inventory` is pre-populated with storage configurations for some of these storage types. However, the storage types defined in the `inventory` can be modified or removed as needed, while additional storage configurations can also be added to meet the specific storage requirements for your PG clusters.

The following `storage` variables are utilized to add or modify operator storage configurations in the `inventory`:

Name	Required	Description
`storage<ID>_name`	Yes	Set to specify a name for the storage configuratio
`storage<ID>_access_mode`	Yes	Set to configure the access mode of the volumes c
`storage<ID>_size`	Yes	Set to configure the size of the volumes created w
`storage<ID>_class`	Required when using the `dynamic` storage type	Set to configure the storage class name used whe
`storage<ID>_fs_group`	Required when using a storage class	Set to configure any filesystem groups that shoul
`storage<ID>_supplemental_groups`	Required when using NFS storage	Set to configure any supplemental groups that sh
`storage<ID>_type`	Yes	Set to either `create` or `dynamic` to configure the

The `ID` portion of `storage` prefix for each variable name above should be an integer that is used to group the various `storage` variables into a single storage configuration. For instance, the following shows a single storage configuration for NFS storage:

```
storage3_name='nfsstorage'
storage3_access_mode='ReadWriteMany'
storage3_size='1G'
storage3_type='create'
```

```
storage3_supplemental_groups=65534
```

As this example storage configuration shows, integer `3` is used as the ID for each of the `storage` variables, which together form a single storage configuration called `nfsstorage`. This approach allows different storage configurations to be created by defining the proper `storage` variables with a unique ID for each required storage configuration.

Additionally, once all storage configurations have been defined in the `inventory`, they can then be used to specify the default storage configuration that should be utilized for the various PG pods created by the operator. This is done using the following variables, which are also defined in the `inventory`:

```
backrest_storage='nfsstorage'
backup_storage='nfsstorage'
primary_storage='nfsstorage'
replica_storage='nfsstorage'
```

With the configuration shown above, the `nfsstorage` storage configuration would be used by default for the various containers created for a PG cluster (i.e. containers for the primary DB, replica DB's, backups and/or `pgBackRest`).

Examples

The following are additional examples of storage configurations for various storage types.

Generic Storage Class The following example defines a storageTo setup storage1 to use the storage class `fast`

```
storage5_name='storageos'
storage5_access_mode='ReadWriteOnce'
storage5_size='300M'
storage5_type='dynamic'
storage5_class='fast'
storage5_fs_group=26
```

To assign this storage definition to all `primary` pods created by the Operator, we can configure the `primary_storage=storageos` variable in the inventory file.

GKE The storage class provided by Google Kubernetes Environment (GKE) can be configured to be used by the Operator by setting the following variables in the `inventory` file:

```
storage8_name='gce'
storage8_access_mode='ReadWriteOnce'
storage8_size='300M'
storage8_type='dynamic'
storage8_class='standard'
storage8_fs_group=26
```

To assign this storage definition to all `primary` pods created by the Operator, we can configure the `primary_storage=gce` variable in the inventory file.

Considerations for Multi-Zone Cloud Environments

When using the Operator in a Kubernetes cluster consisting of nodes that span multiple zones, special consideration must betaken to ensure all pods and the volumes they require are scheduled and provisioned within the same zone. Specifically, being that a pod is unable mount a volume that is located in another zone, any volumes that are dynamically provisioned must be provisioned in a topology-aware manner according to the specific scheduling requirements for the pod. For instance, this means ensuring that the volume containing the database files for the primary database in a new PostgreSQL cluster is provisioned in the same zone as the node containing the PostgreSQL primary pod that will be using it.

For instructions on setting up storage classes for multi-zone environments, see the PostgreSQL Operator Documentation.

Resource Configuration

Kubernetes and OpenShift allow specific resource requirements to be specified for the various containers deployed inside of a pod. This includes defining the required resources for each container, i.e. how much memory and CPU each container will need, while also allowing resource limits to be defined, i.e. the maximum amount of memory and CPU a container will be allowed to consume. In support of this capability, the Crunchy PGO allows any required resource configurations to be defined in the `inventory`, which can the be utilized by the operator to set any desired resource requirements/limits for the various containers that will be deployed by the Crunchy PGO when creating and managing PG clusters.

The following `resource` variables are utilized to add or modify operator resource configurations in the `inventory`:

Name	Required	Description
`resource<ID>_requests_memory`	Yes	The amount of memory required by the container.
`resource<ID>_requests_cpu`	Yes	The amount of CPU required by the container.
`resource<ID>_limits_memory`	Yes	The maximum amount of memory that can be consumed by the container.
`resource<ID>_limits_cpu`	Yes	The maximum amount of CPU that can be consumed by the container.

The `ID` portion of `resource` prefix for each variable name above should be an integer that is used to group the various `resource` variables into a single resource configuration. For instance, the following shows a single resource configuration called `small`:

```
resource1_name='small'
resource1_requests_memory='512Mi'
resource1_requests_cpu=0.1
resource1_limits_memory='512Mi'
resource1_limits_cpu=0.1
```

As this example resource configuration shows, integer `1` is used as the ID for each of the `resource` variables, which together form a single resource configuration called `small`. This approach allows different resource configurations to be created by defining the proper `resource` variables with a unique ID for each required resource configuration.

Additionally, once all resource configurations have been defined in the `inventory`, they can then be used to specify the default resource configurations that should be utilized for the various PG containers created by the operator. This is done using the following variables, which are also defined in the `inventory`:

```
default_container_resources='large'
default_load_resources='small'
default_lspvc_resources='small'
default_rmdata_resources='small'
default_backup_resources='small'
default_pgbouncer_resources='small'
default_pgpool_resources='small'
```

With the configuration shown above, the `large` resource configuration would be used by default for all database containers, while the `small` resource configuration would then be utilized by default for the various other containers created for a PG cluster.

Understanding `pgo_operator_namespace` & `namespace`

The Crunchy PostgreSQL Operator can be configured to be deployed and manage a single namespace or manage several namespaces. The following are examples of different types of deployment models configurable in the `inventory` file.

Single Namespace

To deploy the Crunchy PostgreSQL Operator to work with a single namespace (in this example our namespace is named `pgo`), configure the following `inventory` settings:

```
pgo_operator_namespace='pgo'
namespace='pgo'
```

Multiple Namespaces

To deploy the Crunchy PostgreSQL Operator to work with multiple namespaces (in this example our namespaces are named `pgo`, `pgouser1` and `pgouser2`), configure the following `inventory` settings:

```
pgo_operator_namespace='pgo'
namespace='pgouser1,pgouser2'
```

Deploying Multiple Operators

The 4.0 release of the Crunchy PostgreSQL Operator allows for multiple operator deployments in the same cluster.
To install the Crunchy PostgreSQL Operator to multiple namespaces, it's recommended to have an `inventory` file for each deployment of the operator.

For each operator deployment the following inventory variables should be configured uniquely for each install.

For example, operator could be deployed twice by changing the `pgo_operator_namespace` and `namespace` for those deployments:

Inventory A would deploy operator to the `pgo` namespace and it would manage the `pgo` target namespace.

```
# Inventory A
pgo_operator_namespace='pgo'
namespace='pgo'
...
```

Inventory B would deploy operator to the `pgo2` namespace and it would manage the `pgo2` and `pgo3` target namespaces.

```
# Inventory B
pgo_operator_namespace='pgo2'
namespace='pgo2,pgo3'
...
```

Each install of the operator will create a corresponding directory in `$HOME/.pgo/<PGO NAMESPACE>` which will contain the TLS and `pgouser` client credentials.

Deploying Grafana and Prometheus

PostgreSQL clusters created by the operator can be configured to create additional containers for collecting metrics.
These metrics are very useful for understanding the overall health and performance of PostgreSQL database deployments over time. The collectors included by the operator are:

- PostgreSQL Exporter - PostgreSQL metrics

The operator, however, does not install the necessary timeseries database (Prometheus) for storing the collected metrics or the front end visualization (Grafana) of those metrics.

Included in these playbooks are roles for deploying Granfana and/or Prometheus. See the `inventory` file for options to install the metrics stack.

{{% notice tip %}} At this time the Crunchy PostgreSQL Operator Playbooks only support storage classes. {{% /notice %}}

Installing Ansible on Linux, MacOS or Windows Ubuntu Subsystem

To install Ansible on Linux or MacOS, see the official documentation provided by Ansible.

Install Google Cloud SDK (Optional)

If Crunchy PostgreSQL Operator is going to be installed in a Google Kubernetes Environment the Google Cloud SDK is required.

To install the Google Cloud SDK on Linux or MacOS, see the official Google Cloud documentation.

When installing the Google Cloud SDK on the Windows Ubuntu Subsystem, run the following commands to install:

```
wget https://sdk.cloud.google.com --output-document=/tmp/install-gsdk.sh
# Review the /tmp/install-gsdk.sh prior to running
chmod +x /tmp/install-gsdk.sh
/tmp/install-gsdk.sh
```

Installing

The following assumes the proper prerequisites are satisfied we can now install the PostgreSQL Operator.

The commands should be run in the directory where the Crunchy PostgreSQL Operator playbooks is stored. See the `ansible` directory in the Crunchy PostgreSQL Operator project for the inventory file, main playbook and ansible roles.

Installing on Linux

On a Linux host with Ansible installed we can run the following command to install the PostgreSQL Operator:

```
ansible-playbook -i /path/to/inventory --tags=install --ask-become-pass main.yml
```

If the Crunchy PostgreSQL Operator playbooks were installed using `yum`, use the following commands:

```
export ANSIBLE_ROLES_PATH=/usr/share/ansible/roles/crunchydata

ansible-playbook -i /path/to/inventory --tags=install --ask-become-pass \
    /usr/share/ansible/postgres-operator/playbooks/main.yml
```

Installing on MacOS

On a MacOS host with Ansible installed we can run the following command to install the PostgreSQL Operator.

```
ansible-playbook -i /path/to/inventory --tags=install --ask-become-pass main.yml
```

Installing on Windows Ubuntu Subsystem

On a Windows host with an Ubuntu subsystem we can run the following commands to install the PostgreSQL Operator.

```
ansible-playbook -i /path/to/inventory --tags=install --ask-become-pass main.yml
```

Verifying the Installation

This may take a few minutes to deploy. To check the status of the deployment run the following:

```
# Kubernetes
kubectl get deployments -n <NAMESPACE_NAME>
kubectl get pods -n <NAMESPACE_NAME>

# OpenShift
oc get deployments -n <NAMESPACE_NAME>
oc get pods -n <NAMESPACE_NAME>
```

Configure Environment Variables

After the Crunchy PostgreSQL Operator has successfully been installed we will need to configure local environment variables before using the `pgo` client.

{{% notice info %}}

If TLS authentication was disabled during installation, please see the [TLS Configuration Page] ({{< relref "gettingstarted/Design/tls.md" >}}) for additional configuration information.

{{% / notice %}}

To configure the environment variables used by `pgo` run the following command:

Note: `<PGO_NAMESPACE>` should be replaced with the namespace the Crunchy PostgreSQL Operator was deployed to.

```
cat <<EOF >> ~/.bashrc
export PGOUSER="${HOME?}/.pgo/<PGO_NAMESPACE>/pgouser"
export PGO_CA_CERT="${HOME?}/.pgo/<PGO_NAMESPACE>/client.crt"
export PGO_CLIENT_CERT="${HOME?}/.pgo/<PGO_NAMESPACE>/client.crt"
export PGO_CLIENT_KEY="${HOME?}/.pgo/<PGO_NAMESPACE>/client.pem"
export PGO_APISERVER_URL='https://127.0.0.1:8443'
EOF
```

Apply those changes to the current session by running:

```
source ~/.bashrc
```

Verify pgo Connection

In a separate terminal we need to setup a port forward to the Crunchy PostgreSQL Operator to ensure connection can be made outside of the cluster:

```
# If deployed to Kubernetes
kubectl port-forward <OPERATOR_POD_NAME> -n <OPERATOR_NAMESPACE> 8443:8443

# If deployed to OpenShift
oc port-forward <OPERATOR_POD_NAME> -n <OPERATOR_NAMESPACE> 8443:8443
```

On a separate terminal verify the `pgo` can communicate with the Crunchy PostgreSQL Operator:

```
pgo version
```

If the above command outputs versions of both the client and API server, the Crunchy PostgreSQL Operator has been installed successfully.

Installing

PostgreSQL clusters created by the Crunchy PostgreSQL Operator can optionally be configured to serve performance metrics via Prometheus Exporters. The metric exporters included in the database pod serve realtime metrics for the database container. In order to store and view this data, Grafana and Prometheus are required. The Crunchy PostgreSQL Operator does not create this infrastructure, however, they can be installed using the provided Ansible roles.

Prerequisites

The following assumes the proper prerequisites are satisfied we can now install the PostgreSQL Operator.

At a minimum, the following inventory variables should be configured to install the metrics infrastructure:

Name	Default	Description
`ccp_image_prefix`	crunchydata	Configures the image prefix used when creating containers from Crunchy Container S
`ccp_image_tag`		Configures the image tag (version) used when creating containers from Crunchy Con
`grafana_admin_username`	admin	Set to configure the login username for the Grafana administrator.
`grafana_admin_password`		Set to configure the login password for the Grafana administrator.
`grafana_install`	true	Set to true to install Crunchy Grafana to visualize metrics.
`grafana_storage_access_mode`		Set to the access mode used by the configured storage class for Grafana persistent vo
`grafana_storage_class_name`		Set to the name of the storage class used when creating Grafana persistent volumes.
`grafana_volume_size`		Set to the size of persistent volume to create for Grafana.
`kubernetes_context`		When deploying to Kubernetes, set to configure the context name of the kubeconfig
`metrics`	false	Set to true enable performance metrics on all newly created clusters. This can be dis
`metrics_namespace`	metrics	Configures the target namespace when deploying Grafana and/or Prometheus
`openshift_host`		When deploying to OpenShift, set to configure the hostname of the OpenShift cluste
`openshift_password`		When deploying to OpenShift, set to configure the password used for login.
`openshift_skip_tls_verify`		When deploying to Openshift, set to ignore the integrity of TLS certificates for the O
`openshift_token`		When deploying to OpenShift, set to configure the token used for login (when not us
`openshift_user`		When deploying to OpenShift, set to configure the username used for login.
`prometheus_install`	true	Set to true to install Crunchy Prometheus timeseries database.
`prometheus_storage_access_mode`		Set to the access mode used by the configured storage class for Prometheus persisten
`prometheus_storage_class_name`		Set to the name of the storage class used when creating Prometheus persistent volum

{{% notice tip %}} Administrators can choose to install Grafana, Prometheus or both by configuring the `grafana_install` and `prometheus_install` variables in the inventory files. {{% /notice %}}

The following commands should be run in the directory where the Crunchy PostgreSQL Operator playbooks are located. See the `ansible` directory in the Crunchy PostgreSQL Operator project for the inventory file, main playbook and ansible roles.

{{% notice tip %}} At this time the Crunchy PostgreSQL Operator Playbooks only support storage classes. For more information on storage classes see the official Kubernetes documentation. {{% /notice %}}

Installing on Linux

On a Linux host with Ansible installed we can run the following command to install the Metrics stack:

```
ansible-playbook -i /path/to/inventory --tags=install-metrics main.yml
```

If the Crunchy PostgreSQL Operator playbooks were installed using `yum`, use the following commands:

```
export ANSIBLE_ROLES_PATH=/usr/share/ansible/roles/crunchydata

ansible-playbook -i /path/to/inventory --tags=install-metrics --ask-become-pass \
    /usr/share/ansible/postgres-operator/playbooks/main.yml
```

Installing on MacOS

On a MacOS host with Ansible installed we can run the following command to install the Metrics stack:

```
ansible-playbook -i /path/to/inventory --tags=install-metrics main.yml
```

Installing on Windows

On a Windows host with the Ubuntu subsystem we can run the following commands to install the Metrics stack:

```
ansible-playbook -i /path/to/inventory --tags=install-metrics main.yml
```

Verifying the Installation

This may take a few minutes to deploy. To check the status of the deployment run the following:

```
# Kubernetes
kubectl get deployments -n <NAMESPACE_NAME>
kubectl get pods -n <NAMESPACE_NAME>

# OpenShift
oc get deployments -n <NAMESPACE_NAME>
oc get pods -n <NAMESPACE_NAME>
```

Verify Grafana

In a separate terminal we need to setup a port forward to the Crunchy Grafana deployment to ensure connection can be made outside of the cluster:

```
# If deployed to Kubernetes
kubectl port-forward <GRAFANA_POD_NAME> -n <METRICS_NAMESPACE> 3000:3000

# If deployed to OpenShift
oc port-forward <GRAFANA_POD_NAME> -n <METRICS_NAMESPACE> 3000:3000
```

In a browser navigate to `http://127.0.0.1:3000` to access the Grafana dashboard.

{{% notice tip %}} No metrics will be scraped if no exporters are available. To create a PostgreSQL cluster with metric exporters run the following command:

```
pgo create cluster <NAME OF CLUSTER> --metrics --namespace=<NAMESPACE>
```

{{% /notice %}}

Verify Prometheus

In a separate terminal we need to setup a port forward to the Crunchy Prometheus deployment to ensure connection can be made outside of the cluster:

```
# If deployed to Kubernetes
kubectl port-forward <PROMETHEUS_POD_NAME> -n <METRICS_NAMESPACE> 9090:9090

# If deployed to OpenShift
oc port-forward <PROMETHEUS_POD_NAME> -n <METRICS_NAMESPACE> 9090:9090
```

In a browser navigate to `http://127.0.0.1:9090` to access the Prometheus dashboard.

{{% notice tip %}} No metrics will be scraped if no exporters are available. To create a PostgreSQL cluster with metric exporters run the following command:

```
pgo create cluster <NAME OF CLUSTER> --metrics --namespace=<NAMESPACE>
```

{{% /notice %}}

Updating

Updating the Crunchy PostgreSQL Operator is essential to the lifecycle management of the service. Using the `update` flag will:

- Update and redeploy the operator deployment
- Recreate configuration maps used by operator
- Remove any deprecated objects
- Allow administrators to change settings configured in the `inventory`
- Reinstall the `pgo` client if a new version is specified

The following assumes the proper prerequisites are satisfied we can now update the PostgreSQL Operator.

The commands should be run in the directory where the Crunchy PostgreSQL Operator playbooks is stored. See the `ansible` directory in the Crunchy PostgreSQL Operator project for the inventory file, main playbook and ansible roles.

Updating on Linux

On a Linux host with Ansible installed we can run the following command to update
the PostgreSQL Operator:

```
ansible-playbook -i /path/to/inventory --tags=update --ask-become-pass main.yml
```

If the Crunchy PostgreSQL Operator playbooks were installed using `yum`, use the following commands:

```
export ANSIBLE_ROLES_PATH=/usr/share/ansible/roles/crunchydata

ansible-playbook -i /path/to/inventory --tags=update --ask-become-pass \
    /usr/share/ansible/postgres-operator/playbooks/main.yml
```

Updating on MacOS

On a MacOS host with Ansible installed we can run the following command to update
the PostgreSQL Operator.

```
ansible-playbook -i /path/to/inventory --tags=update --ask-become-pass main.yml
```

Updating on Windows Ubuntu Subsystem

On a Windows host with an Ubuntu subsystem we can run the following commands to update
the PostgreSQL Operator.

```
ansible-playbook -i /path/to/inventory --tags=update --ask-become-pass main.yml
```

Verifying the Update

This may take a few minutes to deploy. To check the status of the deployment run the following:

```
# Kubernetes
kubectl get deployments -n <NAMESPACE_NAME>
kubectl get pods -n <NAMESPACE_NAME>

# OpenShift
oc get deployments -n <NAMESPACE_NAME>
oc get pods -n <NAMESPACE_NAME>
```

Configure Environment Variables

After the Crunchy PostgreSQL Operator has successfully been updated we will need to configure local environment variables before using the `pgo` client.

To configure the environment variables used by `pgo` run the following command:

Note: `<PGO_NAMESPACE>` should be replaced with the namespace the Crunchy PostgreSQL Operator was deployed to. Also, if TLS was disabled, or if the port was changed, update PGO_APISERVER_URL accordingly.

```
cat <<EOF >> ~/.bashrc
export PGOUSER="${HOME?}/.pgo/<PGO_NAMESPACE>/pgouser"
export PGO_CA_CERT="${HOME?}/.pgo/<PGO_NAMESPACE>/client.crt"
export PGO_CLIENT_CERT="${HOME?}/.pgo/<PGO_NAMESPACE>/client.crt"
export PGO_CLIENT_KEY="${HOME?}/.pgo/<PGO_NAMESPACE>/client.pem"
export PGO_APISERVER_URL='https://127.0.0.1:8443'
EOF
```

Apply those changes to the current session by running:

```
source ~/.bashrc
```

Verify `pgo` Connection

In a separate terminal we need to setup a port forward to the Crunchy PostgreSQL Operator to ensure connection can be made outside of the cluster:

```
# If deployed to Kubernetes
kubectl port-forward <OPERATOR_POD_NAME> -n <OPERATOR_NAMESPACE> 8443:8443

# If deployed to OpenShift
oc port-forward <OPERATOR_POD_NAME> -n <OPERATOR_NAMESPACE> 8443:8443
```

Note: If a port other than 8443 was configured, update the above command accordingly.

On a separate terminal verify the `pgo` can communicate with the Crunchy PostgreSQL Operator:

```
pgo version
```

If the above command outputs versions of both the client and API server, the Crunchy PostgreSQL Operator has been updated successfully.

Uninstalling PostgreSQL Operator

The following assumes the proper prerequisites are satisfied we can now uninstall the PostgreSQL Operator.

First, it is recommended to use the playbooks tagged with the same version of the PostgreSQL Operator currently deployed.

With the correct playbooks acquired and prerequisites satisfied, simply run the following command:

```
ansible-playbook -i /path/to/inventory --tags=uninstall --ask-become-pass main.yml
```

If the Crunchy PostgreSQL Operator playbooks were installed using `yum`, use the following commands:

```
export ANSIBLE_ROLES_PATH=/usr/share/ansible/roles/crunchydata

ansible-playbook -i /path/to/inventory --tags=uninstall --ask-become-pass \
    /usr/share/ansible/postgres-operator/playbooks/main.yml
```

Deleting `pgo` Client

If variable `pgo_client_install` is set to `true` in the `inventory` file, the `pgo` client will also be removed when uninstalling.

Otherwise, the `pgo` client can be manually uninstalled by running the following command:

```
rm /usr/local/bin/pgo
```

Uninstalling the Metrics Stack

The following assumes the proper prerequisites are satisfied we can now uninstall the PostgreSQL Operator Metrics Infrastructure.

First, it is recommended to use the playbooks tagged with the same version of the Metrics stack currently deployed.

With the correct playbooks acquired and prerequisites satisfied, simply run the following command:

```
ansible-playbook -i /path/to/inventory --tags=uninstall-metrics main.yml
```

If the Crunchy PostgreSQL Operator playbooks were installed using `yum`, use the following commands:

```
export ANSIBLE_ROLES_PATH=/usr/share/ansible/roles/crunchydata

ansible-playbook -i /path/to/inventory --tags=uninstall-metrics \
    /usr/share/ansible/postgres-operator/playbooks/main.yml
```

Install the Postgres Operator (`pgo`) Client

The following will install and configure the `pgo` client on all systems. For the purpose of these instructions it's assumed that the Crunchy PostgreSQL Operator is already deployed.

Prerequisites

- For Kubernetes deployments: kubectl configured to communicate with Kubernetes
- For OpenShift deployments: oc configured to communicate with OpenShift

The Crunchy Postgres Operator als requires the following in order to authenticate with the apiserver:

- Client CA Certificate
- Client TLS Certificate
- Client Key
- `pgouser` file containing `<username>:<password>`

All of the requirements above should be obtained from an administrator who installed the Crunchy PostgreSQL Operator.

Linux and MacOS

The following will setup the `pgo` client to be used on a Linux or MacOS system.

Installing the Client

First, download the `pgo` client from the GitHub official releases. Crunchy Enterprise Customers can download the pgo binaries from https://access.crunchydata.com/ on the downloads page.

Next, install `pgo` in `/usr/local/bin` by running the following:

```
sudo mv /PATH/TO/pgo /usr/local/bin/pgo
sudo chmod +x /usr/local/bin/pgo
```

Verify the `pgo` client is accessible by running the following in the terminal:

```
pgo --help
```

Configuring Client TLS With the client TLS requirements satisfied we can setup `pgo` to use them.

First, create a directory to hold these files by running the following command:

```
mkdir ${HOME?}/.pgo
chmod 700 ${HOME?}/.pgo
```

Next, copy the certificates to this new directory:

```
cp /PATH/TO/client.crt ${HOME?}/.pgo/client.crt && chmod 600 ${HOME?}/.pgo/client.crt
cp /PATH/TO/client.pem ${HOME?}/.pgo/client.pem && chmod 400 ${HOME?}/.pgo/client.pem
```

Finally, set the following environment variables to point to the client TLS files:

```
cat <<EOF >> ${HOME?}/.bashrc
export PGO_CA_CERT="${HOME?}/.pgo/client.crt"
export PGO_CLIENT_CERT="${HOME?}/.pgo/client.crt"
export PGO_CLIENT_KEY="${HOME?}/.pgo/client.pem"
EOF
```

Apply those changes to the current session by running:

```
source ~/.bashrc
```

Configuring `pgouser` The `pgouser` file contains the username and password used for authentication with the Crunchy PostgreSQL Operator.

To setup the `pgouser` file, run the following:

```
echo "<USERNAME_HERE>:<PASSWORD_HERE>" > ${HOME?}/.pgo/pgouser
```

```
cat <<EOF >> ${HOME?}/.bashrc
export PGOUSER="${HOME?}/.pgo/pgouser"
EOF
```

Apply those changes to the current session by running:

```
source ${HOME?}/.bashrc
```

Configuring the API Server URL If the Crunchy PostgreSQL Operator is not accessible outside of the cluster, it's required to setup a port-forward tunnel using the `kubectl` or `oc` binary.

In a separate terminal we need to setup a port forward to the Crunchy PostgreSQL Operator to ensure connection can be made outside of the cluster:

```
# If deployed to Kubernetes
kubectl port-forward <OPERATOR_POD_NAME> -n <OPERATOR_NAMESPACE> 8443:8443

# If deployed to OpenShift
oc port-forward <OPERATOR_POD_NAME> -n <OPERATOR_NAMESPACE> 8443:8443
```

Note: the port-forward will be required for the duration of `pgo` usage.

Next, set the following environment variable to configure the API server address:

```
cat <<EOF >> ${HOME?}/.bashrc
export PGO_APISERVER_URL="https://<IP_OF_OPERATOR_API>:8443"
EOF
```

Note: if port-forward is being used, the IP of the Operator API is `127.0.0.1`

Apply those changes to the current session by running:

```
source ${HOME?}/.bashrc
```

Windows

The following will setup the `pgo` client to be used on a Windows system.

Installing the Client

First, download the `pgo.exe` client from the GitHub official releases.

Next, create a directory for `pgo` using the following:

- Left click the *Start* button in the bottom left corner of the taskbar
- Type `cmd` to search for *Command Prompt*
- Right click the *Command Prompt* application and click "Run as administrator"
- Enter the following command: `mkdir "%ProgramFiles%\postgres-operator"`

Within the same terminal copy the `pgo.exe` binary to the directory created above using the following command:

```
copy %HOMEPATH%\Downloads\pgo.exe "%ProgramFiles%\postgres-operator"
```

Finally, add `pgo.exe` to the system path by running the following command in the terminal:

```
setx path "%path%;C:\Program Files\postgres-operator"
```

Verify the `pgo.exe` client is accessible by running the following in the terminal:

```
pgo --help
```

Configuring Client TLS With the client TLS requirements satisfied we can setup `pgo` to use them.

First, create a directory to hold these files using the following:

- Left click the *Start* button in the bottom left corner of the taskbar
- Type `cmd` to search for *Command Prompt*
- Right click the *Command Prompt* application and click "Run as administrator"
- Enter the following command: `mkdir "%HOMEPATH%\pgo"`

Next, copy the certificates to this new directory:

```
copy \PATH\TO\client.crt "%HOMEPATH%\pgo"
copy \PATH\TO\client.pem "%HOMEPATH%\pgo"
```

Finally, set the following environment variables to point to the client TLS files:

```
setx PGO_CA_CERT "%HOMEPATH%\pgo\client.crt"
setx PGO_CLIENT_CERT "%HOMEPATH%\pgo\client.crt"
setx PGO_CLIENT_KEY "%HOMEPATH%\pgo\client.pem"
```

Configuring `pgouser` The `pgouser` file contains the username and password used for authentication with the Crunchy PostgreSQL Operator.

To setup the `pgouser` file, run the following:

- Left click the *Start* button in the bottom left corner of the taskbar
- Type `cmd` to search for *Command Prompt*
- Right click the *Command Prompt* application and click "Run as administrator"
- Enter the following command: `echo USERNAME_HERE:PASSWORD_HERE > %HOMEPATH%\pgo\pgouser`

Finally, set the following environment variable to point to the `pgouser` file:

```
setx PGOUSER "%HOMEPATH%\pgo\pgouser"
```

Configuring the API Server URL If the Crunchy PostgreSQL Operator is not accessible outside of the cluster, it's required to setup a port-forward tunnel using the `kubectl` or `oc` binary.

In a separate terminal we need to setup a port forward to the Crunchy PostgreSQL Operator to ensure connection can be made outside of the cluster:

```
# If deployed to Kubernetes
kubectl port-forward <OPERATOR_POD_NAME> -n <OPERATOR_NAMESPACE> 8443:8443

# If deployed to OpenShift
oc port-forward <OPERATOR_POD_NAME> -n <OPERATOR_NAMESPACE> 8443:8443
```

Note: the port-forward will be required for the duration of `pgo` usage.

Next, set the following environment variable to configure the API server address:

- Left click the *Start* button in the bottom left corner of the taskbar
- Type `cmd` to search for *Command Prompt*
- Right click the *Command Prompt* application and click "Run as administrator"
- Enter the following command: `setx PGO_APISERVER_URL "https://<IP_OF_OPERATOR_API>:8443"`
- Note: if port-forward is being used, the IP of the Operator API is `127.0.0.1`

Verify the Client Installation

After completing all of the steps above we can verify `pgo` is configured properly by simply running the following:

```
pgo version
```

If the above command outputs versions of both the client and API server, the Crunchy PostgreSQL Operator client has been installed successfully.

Container Dependencies

The Operator depends on the Crunchy Containers and there are version dependencies between the two projects. Below are the operator releases and their dependent container release. For reference, the Postgres and PgBackrest versions for each container release are also listed.

Operator Release	Container Release	Postgres	PgBackrest Version
4.2.0	4.2.0	12.1	2.18
		11.6	2.18
		10.11	2.18
		9.6.16	2.18
		9.5.20	2.18
4.1.1	4.1.1	12.1	2.18
		11.6	2.18
		10.11	2.18
		9.6.16	2.18
		9.5.20	2.18
4.1.0	2.4.2	11.5	2.17
		10.10	2.17
		9.6.15	2.17
		9.5.19	2.17
4.0.1	2.4.1	11.4	2.13

Operator Release	Container Release	Postgres	PgBackrest Version
		10.9	2.13
		9.6.14	2.13
		9.5.18	2.13
4.0.0	2.4.0	11.3	2.13
		10.8	2.13
		9.6.13	2.13
		9.5.17	2.13
3.5.4	2.3.3	11.4	2.13
		10.9	2.13
		9.6.14	2.13
		9.5.18	2.13
3.5.3	2.3.2	11.3	2.13
		10.8	2.13
		9.6.13	2.13
		9.5.17	2.13
3.5.2	2.3.1	11.2	2.10
		10.7	2.10
		9.6.12	2.10
		9.5.16	2.10

Features sometimes are added into the underlying Crunchy Containers to support upstream features in the Operator thus dictating a dependency between the two projects at a specific version level.

Operating Systems

The Operator is developed on both Centos 7 and RHEL 7 operating systems. The underlying containers are designed to use either Centos 7 or RHEL 7 as the base container image.

Other Linux variants are possible but are not supported at this time.

Kubernetes Distributions

The Operator is designed and tested on Kubernetes and Openshift Container Platform.

Storage

The Operator is designed to support HostPath, NFS, and Storage Classes for persistence. The Operator does not currently include code specific to a particular storage vendor.

Releases

The Operator is released on a quarterly basis often to coincide with Postgres releases.

There are pre-release and or minor bug fix releases created on an as-needed basis.

The operator is template-driven; this makes it simple to configure both the client and the operator.

conf Directory

The Operator is configured with a collection of files found in the *conf* directory. These configuration files are deployed to your Kubernetes cluster when the Operator is deployed. Changes made to any of these configuration files currently require a redeployment of the Operator on the Kubernetes cluster.

The server components of the Operator include Role Based Access Control resources which need to be created a single time by a Kubernetes cluster-admin user. See the Installation section for details on installing a Postgres Operator server.

The configuration files used by the Operator are found in 2 places: * the pgo-config ConfigMap in the namespace the Operator is running in * or, a copy of the configuration files are also included by default into the Operator container images themselves to support a very simplistic deployment of the Operator

If the pgo-config ConfigMap is not found by the Operator, it will use the configuration files that are included in the Operator container images.

The container included set of configuration files use the most basic setting values and the image versions of the Operator itself with the latest Crunchy Container image versions. The storage configurations are determined by using the default storage class on the system you are deploying the Operator into, the default storage class is one that is labeled as follows:

```
pgo-default-sc=true
```

If no storage class has that label, then the first storage class found on the system will be used. If no storage class is found on the system, the containers will not run and produce an error in the log.

conf/postgres-operator/pgo.yaml

The *pgo.yaml* file sets many different Operator configuration settings and is described in the [pgo.yaml configuration]({{< ref "pgo-yaml-configuration.md" >}}) documentation section.

The *pgo.yaml* file is deployed along with the other Operator configuration files when you run:

```
make deployoperator
```

conf/postgres-operator Directory

Files within the *conf/postgres-operator* directory contain various templates that are used by the Operator when creating Kubernetes resources. In an advanced Operator deployment, administrators can modify these templates to add their own custom meta-data or make other changes to influence the Resources that get created on your Kubernetes cluster by the Operator.

Files within this directory are used specifically when creating PostgreSQL Cluster resources. Sidecar components such as pgBouncer and pgPool II templates are also located within this directory.

As with the other Operator templates, administrators can make custom changes to this set of templates to add custom features or metadata into the Resources created by the Operator.

Operator API Server

The Operator's API server can be configured to allow access to select URL routes without requiring TLS authentication from the client and without the HTTP Basic authentication used for role-based-access.

This configuration is performed by defining the `NOAUTH_ROUTES` environment variable for the apiserver container within the Operator pod.

Typically, this configuration is made within the `deploy/deployment.json` file for bash-based installations and `ansible/roles/pgo-operator` for ansible installations.

For example:

```
...
    containers: [
        {
            "name": "apiserver"
            "env": [
                {
                    "name": "NOAUTH_ROUTES",
                    "value": "/health"
                }
            ]
            ...
        }
        ...
    ]
...
```

The `NOAUTH_ROUTES` variable must be set to a comma-separated list of URL routes. For example: `/health,/version,/example3` would opt to **disable** authentication for `$APISERVER_URL/health`, `$APISERVER_URL/version`, and `$APISERVER_URL/example3` respectively.

Currently, only the following routes may have authentication disabled using this setting:

```
/health
```

If the health route has its authentication disabled, the existing readiness and liveness probes for the apiserver container could be enhanced by configuring them with HTTP-based checks against the health route.

Security

Setting up pgo users and general security configuration is described in the Security section of this documentation.

Local pgo CLI Configuration

You can specify the default namespace you want to use by setting the PGO_NAMESPACE environment variable locally on the host the pgo CLI command is running.

```
export PGO_NAMESPACE=pgouser1
```

When that variable is set, each command you issue with *pgo* will use that namespace unless you over-ride it using the *–namespace* command line flag.

```
pgo show cluster foo --namespace=pgouser2
```

Namespace Configuration

The Design Design section of this documentation talks further about the use of namespaces within the Operator and configuring different deployment models of the Operator.

pgo.yaml Configuration

The *pgo.yaml* file contains many different configuration settings as described in this section of the documentation.

The *pgo.yaml* file is broken into major sections as described below: ## Cluster

Setting	Definition
BasicAuth	If set to `"true"` will enable Basic Authentication. If set to `"false"`, will allow a valid Operator user to succe
PrimaryNodeLabel	newly created primary deployments will specify this node label if specified, unless you override it using the –
ReplicaNodeLabel	newly created replica deployments will specify this node label if specified, unless you override it using the –n
CCPImagePrefix	newly created containers will be based on this image prefix (e.g. crunchydata), update this if you require a cu
CCPImageTag	newly created containers will be based on this image version (e.g. centos7-12.1-4.2.0), unless you override it u
Port	the PostgreSQL port to use for new containers (e.g. 5432)
PGBadgerPort	the port used to connect to pgbadger (e.g. 10000)
ExporterPort	the port used to connect to postgres exporter (e.g. 9187)
LogStatement	postgresql.conf log_statement value (required field)
LogMinDurationStatement	postgresql.conf log_min_duration_statement value (required field)
User	the PostgreSQL normal user name
Database	the PostgreSQL normal user database
Replicas	the number of cluster replicas to create for newly created clusters, typically users will scale up replicas on the

Setting	Definition
PgmonitorPassword	the password to use for pgmonitor metrics collection if you specify –metrics when creating a PG cluster
Metrics	boolean, if set to true will cause each new cluster to include crunchy-collect as a sidecar container for metrics
Badger	boolean, if set to true will cause each new cluster to include crunchy-pgbadger as a sidecar container for stat
Policies	optional, list of policies to apply to a newly created cluster, comma separated, must be valid policies in the c
PasswordAgeDays	optional, if set, will set the VALID UNTIL date on passwords to this many days in the future when creating
PasswordLength	optional, if set, will determine the password length used when creating passwords, defaults to 8
ServiceType	optional, if set, will determine the service type used when creating primary or replica services, defaults to Cl
Backrest	optional, if set, will cause clusters to have the pgbackrest volume PVC provisioned during cluster creation
BackrestPort	currently required to be port 2022
DisableAutofail	optional, if set, will disable autofail capabilities by default in any newly created cluster

Storage

Setting	Definition
PrimaryStorage	required, the value of the storage configuration to use for the primary PostgreSQL deployment
BackupStorage	required, the value of the storage configuration to use for backups, including the storage for pgbackrest rep
ReplicaStorage	required, the value of the storage configuration to use for the replica PostgreSQL deployments
ReplicaStorage	required, the value of the storage configuration to use for the replica PostgreSQL deployments
BackrestStorage	required, the value of the storage configuration to use for the pgbackrest shared repository deployment cre
StorageClass	for a dynamic storage type, you can specify the storage class used for storage provisioning(e.g. standard, g
AccessMode	the access mode for new PVCs (e.g. ReadWriteMany, ReadWriteOnce, ReadOnlyMany). See below for des
Size	the size to use when creating new PVCs (e.g. 100M, 1Gi)
Storage.storage1.StorageType	supported values are either *dynamic*, *create*, if not supplied, *create* is used
Fsgroup	optional, if set, will cause a *SecurityContext* and *fsGroup* attributes to be added to generated Pod and De
SupplementalGroups	optional, if set, will cause a SecurityContext to be added to generated Pod and Deployment definitions
MatchLabels	optional, if set, will cause the PVC to add a *matchlabels* selector in order to match a PV, only useful when

Storage Configuration Examples

In *pgo.yaml*, you will need to configure your storage configurations depending on which storage you are wanting to use for Operator provisioning of Persistent Volume Claims. The examples below are provided as a sample. In all the examples you are free to change the *Size* to meet your requirements of Persistent Volume Claim size.

HostPath Example

HostPath is provided for simple testing and use cases where you only intend to run on a single Linux host for your Kubernetes cluster.

```
hostpathstorage:
  AccessMode:  ReadWriteMany
  Size:  1G
  StorageType:  create
```

NFS Example

In the following NFS example, notice that the *SupplementalGroups* setting is set, this can be whatever GID you have your NFS mount set to, typically we set this *nfsnobody* as below. NFS file systems offer a *ReadWriteMany* access mode.

```
nfsstorage:
  AccessMode:  ReadWriteMany
```

```
    Size:  1G
    StorageType:  create
    SupplementalGroups:  65534
```

Storage Class Example

In the following example, the important attribute to set for a typical Storage Class is the *Fsgroup* setting. This value is almost always set to *26* which represents the Postgres user ID that the Crunchy Postgres container runs as. Most Storage Class providers offer *ReadWriteOnce* access modes, but refer to your provider documentation for other access modes it might support.

```
  storageos:
    AccessMode:  ReadWriteOnce
    Size:  1G
    StorageType:  dynamic
    StorageClass:  fast
    Fsgroup:  26
```

Container Resources

Setting	Definition
DefaultContainerResource	optional, the value of the container resources configuration to use for all database containers, if not set, no re
DefaultLoadResource	optional, the value of the container resources configuration to use for pgo-load containers, if not set, no resou
DefaultLspvcResource	optional, the value of the container resources configuration to use for pgo-lspvc containers, if not set, no resou
DefaultRmdataResource	optional, the value of the container resources configuration to use for pgo-rmdata containers, if not set, no re
DefaultBackupResource	optional, the value of the container resources configuration to use for crunchy-backup containers, if not set, n
DefaultPgbouncerResource	optional, the value of the container resources configuration to use for crunchy-pgbouncer containers, if not se
DefaultPgpoolResource	optional, the value of the container resources configuration to use for crunchy-pgpool containers, if not set, n
RequestsMemory	request size of memory in bytes
RequestsCPU	request size of CPU cores
LimitsMemory	request size of memory in bytes
LimitsCPU	request size of CPU cores

Miscellaneous (Pgo)

Setting	Definition
PreferredFailoverNode	optional, a label selector (e.g. hosttype=offsite) that if set, will be used to pick the failover target which is runnin
COImagePrefix	image tag prefix to use for the Operator containers
COImageTag	image tag to use for the Operator containers
Audit	boolean, if set to true will cause each apiserver call to be logged with an *audit* marking

Storage Configuration Details

You can define n-number of Storage configurations within the *pgo.yaml* file. Those Storage configurations follow these conventions -

- they must have lowercase name (e.g. storage1)
- they must be unique names (e.g. mydrstorage, faststorage, slowstorage)

These Storage configurations are referenced in the BackupStorage, ReplicaStorage, and PrimaryStorage configuration values. However, there are command line options in the *pgo* client that will let a user override these default global values to offer you the user a way to specify very targeted storage configurations when needed (e.g. disaster recovery storage for certain backups).

You can set the storage AccessMode values to the following:

- *ReadWriteMany* - mounts the volume as read-write by many nodes
- *ReadWriteOnce* - mounts the PVC as read-write by a single node
- *ReadOnlyMany* - mounts the PVC as read-only by many nodes

These Storage configurations are validated when the *pgo-apiserver* starts, if a non-valid configuration is found, the apiserver will abort. These Storage values are only read at *apiserver* start time.

The following StorageType values are possible -

- *dynamic* - this will allow for dynamic provisioning of storage using a StorageClass.
- *create* - This setting allows for the creation of a new PVC for each PostgreSQL cluster using a naming convention of *clustername*. When set, the *Size*, *AccessMode* settings are used in constructing the new PVC.

The operator will create new PVCs using this naming convention: *dbname* where *dbname* is the database name you have specified. For example, if you run:

```
pgo create cluster example1 -n pgouser1
```

It will result in a PVC being created named *example1* and in the case of a backup job, the pvc is named *example1-backup*

Note, when Storage Type is *create*, you can specify a storage configuration setting of *MatchLabels*, when set, this will cause a *selector* of *key=value* to be added into the PVC, this will let you target specific PV(s) to be matched for this cluster. Note, if a PV does not match the claim request, then the cluster will not start. Users that want to use this feature have to place labels on their PV resources as part of PG cluster creation before creating the PG cluster. For example, users would add a label like this to their PV before they create the PG cluster:

```
kubectl label pv somepv myzone=somezone -n pgouser1
```

If you do not specify *MatchLabels* in the storage configuration, then no match filter is added and any available PV will be used to satisfy the PVC request. This option does not apply to *dynamic* storage types.

Example PV creation scripts are provided that add labels to a set of PVs and can be used for testing: `$COROOT/pv/create-pv-nfs-labels.sh` in that example, a label of **crunchyzone=red** is set on a set of PVs to test with.

The *pgo.yaml* includes a storage config named **nfsstoragered** that when used will demonstrate the label matching. This feature allows you to support n-number of NFS storage configurations and supports spreading a PG cluster across different NFS storage configurations.

Container Resources Details

In the *pgo.yaml* configuration file you have the option to configure a default container resources configuration that when set will add CPU and memory resource limits and requests values into each database container when the container is created.

You can also override the default value using the `--resources-config` command flag when creating a new cluster:

```
pgo create cluster testcluster --resources-config=large -n pgouser1
```

Note, if you try to allocate more resources than your host or Kube cluster has available then you will see your pods wait in a *Pending* status. The output from a `kubectl describe pod` command will show output like this in this event: Events:

```
  Type     Reason            Age                 From                Message
  ----     ------            ----                ----                -------
  Warning  FailedScheduling  49s (x8 over 1m)    default-scheduler   No nodes are available that
      match all of the predicates: Insufficient memory (1).
```

Overriding Storage Configuration Defaults

```
pgo create cluster testcluster --storage-config=bigdisk -n pgouser1
```

That example will create a cluster and specify a storage configuration of *bigdisk* to be used for the primary database storage. The replica storage will default to the value of ReplicaStorage as specified in *pgo.yaml*.

```
pgo create cluster testcluster2 --storage-config=fastdisk --replica-storage-config=slowdisk -n
   pgouser1
```

That example will create a cluster and specify a storage configuration of *fastdisk* to be used for the primary database storage, while the replica storage will use the storage configuration *slowdisk*.

```
pgo backup testcluster --storage-config=offsitestorage -n pgouser1
```

That example will create a backup and use the *offsitestorage* storage configuration for persisting the backup.

Using Storage Configurations for Disaster Recovery

A simple mechanism for partial disaster recovery can be obtained by leveraging network storage, Kubernetes storage classes, and the storage configuration options within the Operator.

For example, if you define a Kubernetes storage class that refers to a storage backend that is running within your disaster recovery site, and then use that storage class as a storage configuration for your backups, you essentially have moved your backup files automatically to your disaster recovery site thanks to network storage.

Node Affinity in PostgreSQL Operator

Kubernetes node affinity allows you to constrain which nodes your pod is eligible to be scheduled on, based on labels on the node.

The PostgreSQL Operator provides users with the ability to add a node affinity section to a new Cluster Deployment. By adding a node affinity section to the Cluster Deployment, users can direct Kubernetes to attempt to schedule a primary PostgreSQL instance within a cluster on a specific Kubernetes node.

As an example, you can see the nodes on your Kubernetes cluster by running the following:

```
kubectl get nodes
```

You can then specify one of those Kubernetes node names (e.g. kubeadm-node2) when creating a PostgreSQL cluster;

```
pgo create cluster thatcluster --node-label=kubeadm-node2
```

The node affinity rule inserted in the Deployment uses a *preferred* strategy so that if the node were down or not available, Kubernetes will go ahead and schedule the Pod on another node.

When you scale up a PostgreSQL cluster by adding a PostgreSQL replica instance, the scaling will take into account the use of `--node-label`. If it sees that a PostgreSQL cluster was created with a specific node name, then the PostgreSQL replica Deployment will add an affinity rule to attempt to schedule the Pod.

Storage and the PostgreSQL Operator

The PostgreSQL Operator allows for a variety of different configurations of persistent storage that can be leveraged by the PostgreSQL instances or clusters it deploys.

The PostgreSQL Operator works with several different storage types, HostPath, Network File System(NFS), and Dynamic storage.

- Hostpath is the simplest storage and useful for single node testing.
- NFS provides the ability to do single and multi-node testing.

Hostpath and NFS both require you to configure persistent volumes so that you can make claims towards those volumes. You will need to monitor the persistent volumes so that you do not run out of available volumes to make claims against.

Dynamic storage classes provide a means for users to request persistent volume claims and have the persistent volume dynamically created for you. You will need to monitor disk space with dynamic storage to make sure there is enough space for users to request a volume. There are multiple providers of dynamic storage classes to choose from. You will need to configure what works for your environment and size the Physical Volumes, Persistent Volumes (PVs), appropriately.

Once you have determined the type of storage you will plan on using and setup PV's you need to configure the Operator to know about it. You will do this in the pgo.yaml file.

If you are deploying to a cloud environment with multiple zones, for instance Google Kubernetes Engine (GKE), you will want to review topology aware storage class configurations.

PostgreSQL Operator Backup and Restore Capability

The PostgreSQL Operator provides users with the ability to manage PostgreSQL cluster backups through both native PostgreSQL backup functionality, as well as using pgbackrest, an open source backup and restore solution designed to scale up to the largest databases. By default, beginning with verison 4.0, the PostgreSQL Operator backup command performs a PostgreSQL pgbackrest backup.

The three backup types that can be configured through the PostgreSQL Operator CLI are:

- pgbackrest a simple, reliable backup and restore solution that can seamlessly scale up to the largest databases and workloads. It provides full, incremental, differential backups, and point-in-time recovery.

- pg_basebackup is used to take base backups of a running PostgreSQL database cluster. These are taken without affecting other clients to the database, and can be used both for point-in-time recovery and as the starting point for a log shipping or streaming replication standby servers.
- pg_dump is a utility for backing up a single PostgreSQL database. It makes consistent backups even if the database is being used concurrently. pg_dump does not block other users accessing the database (readers or writers).pg_dump

pgBackRest Integration

The PostgreSQL Operator integrates various features of the pgbackrest backup and restore project to support backup and restore capability.

The *pgo-backrest-repo* container acts as a pgBackRest remote repository for the PostgreSQL cluster to use for storing archive files and backups.

The following diagrams depicts some of the integration features:

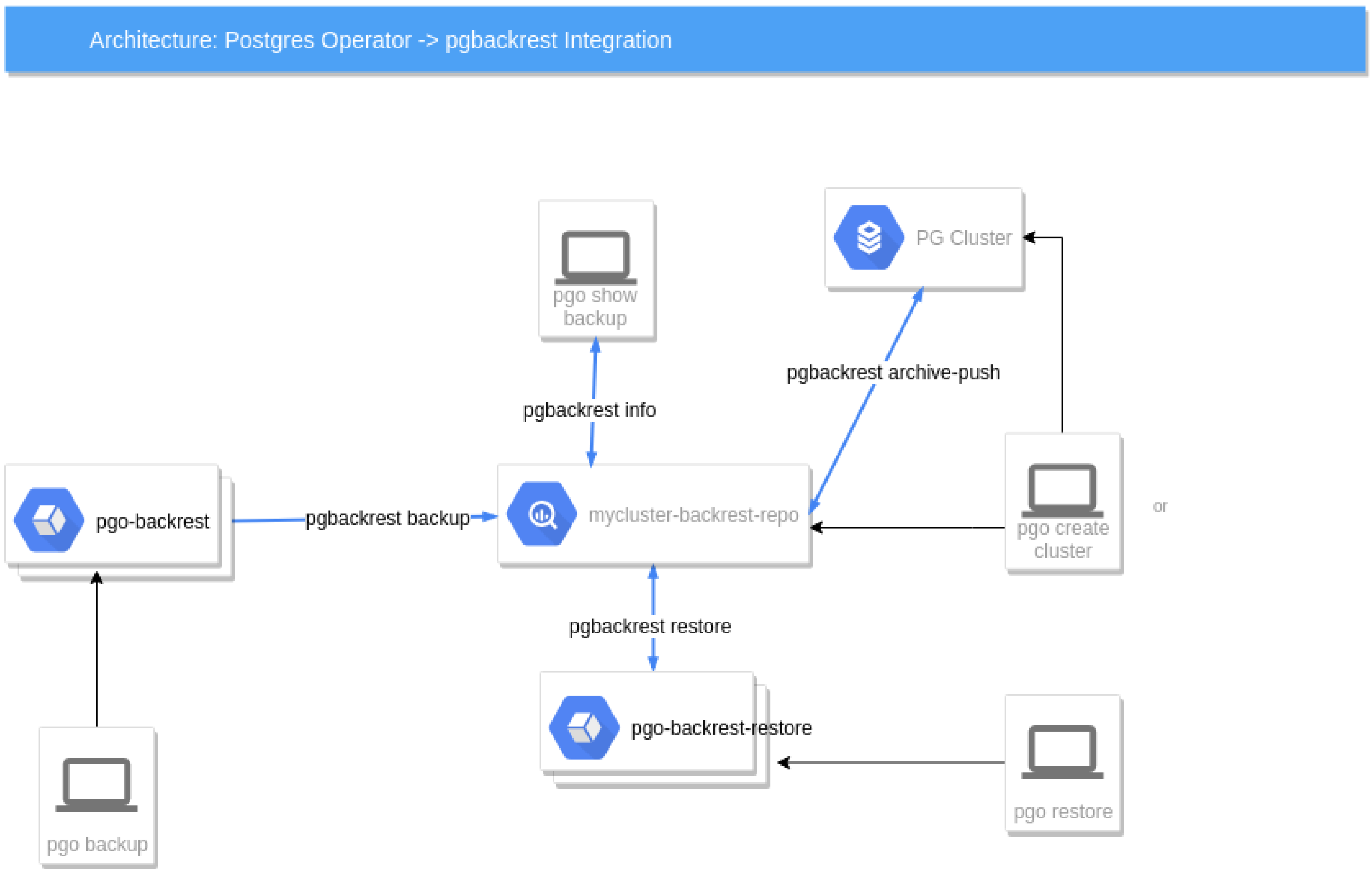

Figure 11: alt text

In this diagram, starting from left to right we see the following:

- a user when they enter *pgo backup mycluster –backup-type=pgbackrest* will cause a pgo-backrest container to be run as a Job, that container will execute a *pgbackrest backup* command in the pgBackRest repository container to perform the backup function.
- a user when they enter *pgo show backup mycluster –backup-type=pgbackrest* will cause a *pgbackrest info* command to be executed on the pgBackRest repository container, the *info* output is sent directly back to the user to view
- the PostgreSQL container itself will use an archive command, *pgbackrest archive-push* to send archives to the pgBackRest repository container
- the user entering *pgo create cluster mycluster –pgbackrest* will cause a pgBackRest repository container deployment to be created, that repository is exclusively used for this Postgres cluster
- lastly, a user entering *pgo restore mycluster* will cause a *pgo-backrest-restore* container to be created as a Job, that container executes the *pgbackrest restore* command

Support for pgBackRest Use of S3 Buckets

The PostgreSQL Operator supports the use AWS S3 storage buckets for the pgbackrest repository in any pgbackrest-enabled cluster. When S3 support is enabled for a cluster, all archives will automatically be pushed to a pre-configured S3 storage bucket, and that same bucket can then be utilized for the creation of any backups as well as when performing restores. Please note that once a storage type has been selected for a cluster during cluster creation (specifically `local`, `s3`, or *both*, as described in detail below), it cannot be changed.

The PostgreSQL Operator allows for the configuration of a single storage bucket, which can then be utilized across multiple clusters. Once S3 support has been enabled for a cluster, pgbackrest will create a `backrestrepo` directory in the root of the configured S3 storage bucket (if it does not already exist), and subdirectories will then be created under the `backrestrepo` directory for each cluster created with S3 storage enabled.

PostgreSQL Operator Custom Resource Definitions

The PostgreSQL Operator defines the following series of Kubernetes Custom Resource Definitions (CRDs):

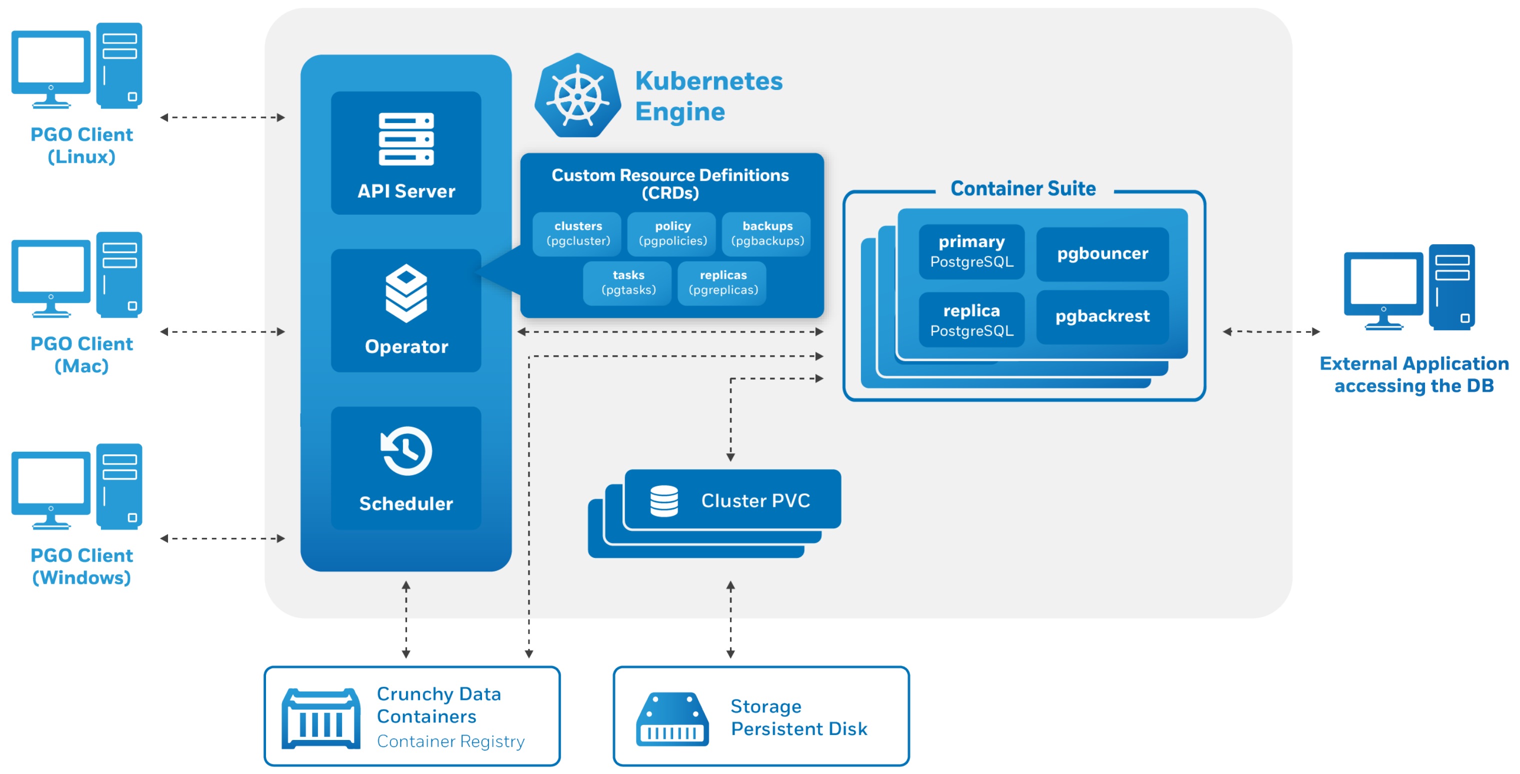

Figure 12: Operator Architecture with CRDs

Each of these CRDs are used in the design of the PostgreSQL Operator to perform PostgreSQL related operations in order to enable with on-demand, PostgreSQL-as-a-Service workflows.

Cluster (pgclusters)

The Cluster or pgcluster CRD is used by the PostgreSQL Operator to define the PostgreSQL cluster definition and make new PostgreSQL cluster requests.

Backup (pgbackups)

The Backup or pgbackup CRD is used by the PostgreSQL Operator to perform a pgbasebackup and to hold the workflow and status of the last backup job. Crunchy Data plans to deprecate this CRD in a future release in favor of a more general pgtask resource

Tasks (pgtask)

The Tasks or pgtask CRD is used by the PostgreSQL Operator to perform workflow and other related administration tasks. The pgtasks CRD captures workflows and administrative tasks for a given pgcluster.

Replica (pgreplica)

The Replica or pgreplica CRD is used by teh PostgreSQL Operator to create a PostgreSQL replica. When a user creates a PostgreSQL replica, a pgreplica CRD is created to define that replica.

Metadata about each PostgreSQL cluster deployed by the PostgreSQL Operator are stored within these CRD resources which act as the source of truth for the Operator. The PostgreSQL Operator makes use of CRDs to maintain state and resource definitions as offered by the PostgreSQL Operator.

PostgreSQL Operator Overview

The PostgreSQL Operator extends Kubernetes to provide a higher- level abstraction enabling the rapid creation and management of PostgreSQL databases and clusters.

The PostgreSQL Operator include the following components:

- PostgreSQL Operator
- PostgreSQL Operator Containers
- PostgreSQL Operator PGO Client
- PostgreSQL Operator REST API Server
- PostgreSQL PGO Scheduler

The following diagram provides an overview of the components of the PostgreSQL Operator:

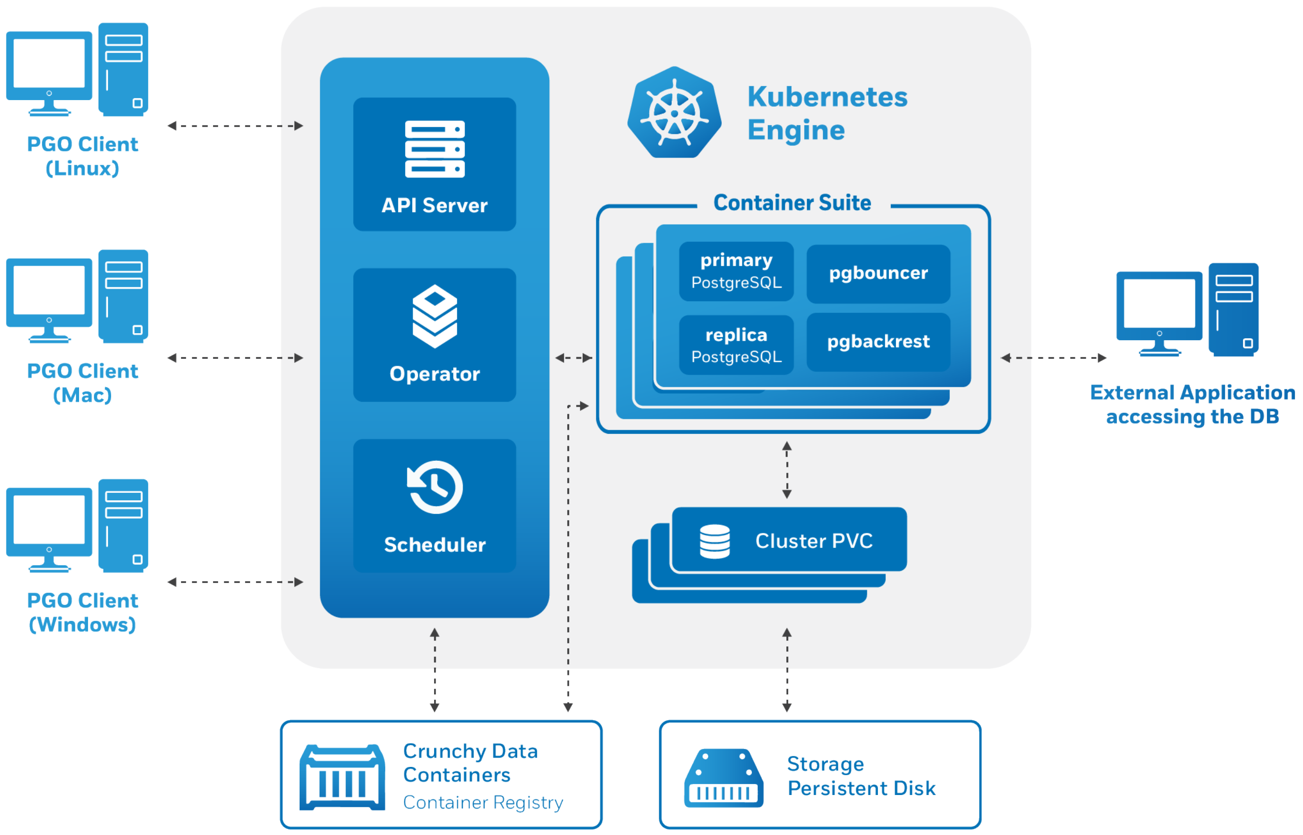

Figure 13: Operator Architecture

PostgreSQL Operator The PostgreSQL Operator makes use of Kubernetes "Custom Resource Definitions" or "CRDs" to extend Kubernetes with custom, PostgreSQL specific, Kubernetes objects such as "Database" and "Cluster". The PostgreSQL Operator users these CRDs to enable users to deploy, configure and administer PostgreSQL databases and clusters as Kubernetes-natvie, open source PostgreSQL-as-a-Service infrastructure.

PostgreSQL Operator Containers The PostgreSQL Operator orchestrates a series of PostgreSQL and PostgreSQL related containers containers that enable rapid deployment of PostgreSQL, including administration and monitoring tools in a Kubernetes environment.

PostgreSQL Operator PGO Client The PostgreSQL Operator provides a command line interface (CLI), pgo. This CLI tool may be used an end-user to create databases or clusters, or make changes to existing databases. The CLI interacts with the REST API deployed within the postgres-operator deployment.

PostgreSQL Operator REST API Server A feature of the PostgreSQL Operator is to provide a REST API upon which users or custom applications can inspect and cause actions within the Operator such as provisioning resources or viewing status of resources. This API is secured by a RBAC (role based access control) security model whereby each API call has a permission assigned to it. API roles are defined to provide granular authorization to Operator services.

PostgreSQL Operator PGO Scheduler The PostgreSQL Operator includes a cron like scheduler application called pgo-scheduler. The purpose of pgo-scheduler is to run automated tasks such as PostgreSQL backups or SQL policies against PostgreSQL clusters created by the PostgreSQL Operator. PGO Scheduler watches Kubernetes for configmaps with the label crunchy-scheduler=true in the same namespace where the PostgreSQL Operator is deployed.

PGO Scheduler

The PostgreSQL Operator includes a cronlike scheduler application called `pgo-scheduler`. Its purpose is to run automated tasks such as PostgreSQL backups or SQL policies against PostgreSQL instances and clusters created by the PostgreSQL Operator.

PGO Scheduler watches Kubernetes for configmaps with the label `crunchy-scheduler=true` in the same namespace the Operator is deployed. The configmaps are json objects that describe the schedule such as:

- Cron like schedule such as: * * * * *
- Type of task: `pgbackrest`, `pgbasebackup` or `policy`

Schedules are removed automatically when the configmaps are deleted.

PGO Scheduler uses the `UTC` timezone for all schedules.

Schedule Expression Format

Schedules are expressed using the following rules:

```
Field name   | Mandatory? | Allowed values  | Allowed special characters
----------   | ---------- | --------------  | --------------------------
Seconds      | Yes        | 0-59            | * / , -
Minutes      | Yes        | 0-59            | * / , -
Hours        | Yes        | 0-23            | * / , -
Day of month | Yes        | 1-31            | * / , - ?
Month        | Yes        | 1-12 or JAN-DEC | * / , -
Day of week  | Yes        | 0-6 or SUN-SAT  | * / , - ?
```

pgBackRest Schedules

pgBackRest schedules require pgBackRest enabled on the cluster to backup. The scheduler will not do this on its own.

pgBaseBackup Schedules

pgBaseBackup schedules require a backup PVC to already be created. The operator will make this PVC using the backup commands:

```
pgo backup mycluster
```

Policy Schedules

Policy schedules require a SQL policy already created using the Operator. Additionally users can supply both the database in which the policy should run and a secret that contains the username and password of the PostgreSQL role that will run the SQL. If no user is specified the scheduler will default to the replication user provided during cluster creation.

User Roles in the PostgreSQL Operator

The PostgreSQL Operator, when used in conjunction with the associated PostgreSQL Containers and Kubernetes, provides you with the ability to host your own open source, Kubernetes native PostgreSQL-as-a-Service infrastructure.

In installing, configuring and operating the PostgreSQL Operator as a PostgreSQL-as-a-Service capability, the following user roles will be required:

Role	Applicable Component	Authorized Privileges and Functions Performed
Platform Admininistrator (Privileged User)	PostgreSQL Operator	The Platform Admininistrator is able to control all aspects of
Platform User	PostgreSQL Operator	The Platform User has access to a limited subset of PostgreSQ
PostgreSQL Administrator(Privileged Account)	PostgreSQL Containers	The PostgreSQL Administrator is the equivalent of a PostgreS
PostgreSQL User	PostgreSQL Containers	The PostgreSQL User has access to a PostgreSQL Instance or

As indicated in the above table, both the Operator Administrator and the PostgreSQL Administrators represent privilege users with components within the PostgreSQL Operator.

Platform Administrator

For purposes of this User Guide, the "Platform Administrator" is a Kubernetes system user with PostgreSQL Administrator privileges and has PostgreSQL Operator admin rights. While PostgreSQL Operator admin rights are not required, it is helpful to have admin rights to be able to verify that the installation completed successfully. The Platform Administrator will be responsible for managing the installation of the Crunchy PostgreSQL Operator service in Kubernetes. That installation can be on RedHat OpenShift 3.11+, Kubeadm, or even Google's Kubernetes Engine.

Platform User

For purposes of this User Guide, a "Platform User" is a Kubernetes system user and has PostgreSQL Operator admin rights. While admin rights are not required for a typical user, testing out functiontionality will be easier, if you want to limit functionality to specific actions section 2.4.5 covers roles. The Platform User is anyone that is interacting with the Crunchy PostgreSQL Operator service in Kubernetes via the PGO CLI tool. Their rights to carry out operations using the PGO CLI tool is governed by PGO Roles(discussed in more detail later) configured by the Platform Administrator. If this is you, please skip to section 2.3.1 where we cover configuring and installing PGO.

PostgreSQL User

In the context of the PostgreSQL Operator, the "PostgreSQL User" is any person interacting with the PostgreSQL database using database specific connections, such as a language driver or a database management GUI.

The default PostgreSQL instance installation via the PostgreSQL Operator comes with the following users:

Role name	Attributes
postgres	Superuser, Create role, Create DB, Replication, Bypass RLS
primaryuser	Replication
testuser	

The postgres user will be the admin user for the database instance. The primary user is used for replication between primary and replicas. The testuser is a normal user that has access to the database "userdb" that is created for testing purposes.

Failover in the PostgreSQL Operator

There are a number of potential events that could cause a primary database instance or cluster to become unavailable during the course of normal operations, including:

- A database storage (disk) failure or any other hardware failure
- The network on which the database resides becomes unreachable
- The host operating system becomes unstable and crashes

- A key database file becomes corrupted
- Total loss of data center

There may also be downtime events that are due to the normal case of operations, such as performing a minor upgrade, security patching of operating system, hardware upgrade, or other maintenance.

To enable rapid recovery from the unavailability of the primary PostgreSQL instance within a PostgreSQL cluster, the PostgreSQL Operator supports both Manual and Automated failover within a single Kubernetes cluster.

PostgreSQL Cluster Architecture

The failover from a primary PostgreSQL instances to a replica PostgreSQL instance within a PostgreSQL cluster.

Manual Failover

Manual failover is performed by PostgreSQL Operator API actions involving a *query* and then a *target* being specified to pick the fail-over replica target.

Automatic Failover

Automatic failover is managed and performed by Patroni, which is running within each primary and replica database pod within the cluster to ensure the PG database remains highly-available. By monitoring the cluster, Patroni is able to detect failures in the primary database, and then automatically failover to (i.e. "promote") a healthy replica. Automatic failover capabilities are enabled by default for any newly created clusters, but can also be disabled for a newly created cluster by setting `DisableFailover` to true in the `pgo.yaml` configuration, or by setting the `--disable-failover` flag via the PGO CLI when creating the cluster. If disabled, failover capabiltiies can then be enabled (as well as disabled once again) at any time by utilizing the `pgo update cluster` command.

When a failover does occur, the system automatically attempts to turn the old primary into a replica (using `pg_rewind` if needed), ensuring the cluster maintains the same amount of database pods and replicas following a failure. Additionally, the `role` label on each pod is updated as needed to properly identify the `master` pod and any `replica` pods following a failover event, therefore ensuring the primary and replica services point to the proper database pods. And finally, the `pgBackRest` dedicated repository host is also automatically reconfigured to point to the `PGDATA` directory of the new `primary` pod following a failover.

Kubernetes Namespaces and the PostgreSQL Operator

In Kubernetes, namespaces provide the user the ability to divide cluster resources between multiple users (via resource quota).

The PostgreSQL Operator makes use of the Kubernetes Namespace support in order to define the Namespace to which the PostgreSQL Operator will deploy PostgreSQL clusters, enabling users to more easily allocate Kubernetes resources to specific areas within their business (users, projects, departments).

Namespaces Applied to Organizational Requirements Prior to version PostgreSQL Operator 4.0, the PostgreSQL Operator could only be deployed with a Namespace deployment pattern where both the PostgreSQL Operator and the PostgreSQL Clusters it deployed existed within a single Kubernetes namespace.

With the PostgreSQL Operator 4.0 release, the operator now supports a variety of Namespace deployment patterns, including:

- **OwnNamespace** Operator and PostgreSQL clusters deployed to the same Kubernetes Namespace
- **SingleNamespace and MultiNamespace** Operator and PostgreSQL clusters deployed to a predefined set of Kubernetes Namespaces
- **AllNamespaces** Operator deployed into a single Kubernetes Namespace but watching all Namespaces on a Kubernetes cluster

Configuration of the Namespace to which PostgreSQL Operator is Deployed In order to configure the Kubernetes Namespace within which the PostgreSQL Operator will run, it is necessary to configure the PGO_OPERATOR_NAMESPACE environment variable. Both the Ansible and Bash installation method enable you to modify this PGO_OPERATOR_NAMESPACE environment variable in connection with the PostgreSQL Operator installation.

Configuration of the Namespaces to which PostgreSQL Operator will Deploy PostgreSQL Clusters At startup time, the PostgreSQL Operator determines the Kubernetes Namespaces to which it will be able to deploy and administer PostgreSQL databases and clusters. The Kubernetes Namespace that the PostgreSQL Operator will be able to service is determined at startup time by the NAMESPACE environment variable. The NAMESPACE variable is set as part of the PostgreSQL Operator installation process. The format of the NAMESPACE value in the PostgreSQL Operator is modeled after the Operator Lifecycle Manager project.

PostgreSQL Operator Containers Overview

The PostgreSQL Operator orchestrates a series of PostgreSQL and PostgreSQL related containers containers that enable rapid deployment of PostgreSQL, including administration and monitoring tools in a Kubernetes environment. The PostgreSQL Operator supports PostgreSQL 9.5+ with multiple PostgreSQL cluster deployment strategies and a variety of PostgreSQL related extensions and tools enabling enterprise grade PostgreSQL-as-a-Service. A full list of the containers supported by the PostgreSQL Operator is provided below.

PostgreSQL Server and Extensions

- **PostgreSQL** (crunchy-postgres). PostgreSQL database server. The crunchy-postgres container image is unmodified, open source PostgreSQL packaged and maintained by Crunchy Data.
- **PostGIS** (crunchy-postgres-gis). PostgreSQL database server including the PostGIS extension. The crunchy-postgres-gis container image is unmodified, open source PostgreSQL packaged and maintained by Crunchy Data. This image is identical to the crunchy-postgres image except it includes the open source geospatial extension PostGIS for PostgreSQL in addition to the language extension PL/R which allows for writing functions in the R statistical computing language.

Backup and Restore

- **pgBackRest** (crunchy-backrest-restore). pgBackRest is a high performance backup and restore utility for PostgreSQL. The crunchy-backrest-restore container executes the pgBackRest utility, allowing FULL and DELTA restore capability.
- **pg_basebackup** (crunchy-backup). pg_basebackup is used to take base backups of a running PostgreSQL database cluster. The crunchy-backup container executes a full backup against another database container using the standard pg_basebackup utility that is included with PostgreSQL.
- **pgdump** (crunchy-pgdump). The crunchy-pgdump container executes either a pg_dump or pg_dumpall database backup against another PostgreSQL database.
- **crunchy-pgrestore** (restore). The restore image provides a means of performing a restore of a dump from pg_dump or pg_dumpall via psql or pg_restore to a PostgreSQL container database.

Administration Tools

- **pgAdmin4** (crunchy-pgadmin4). PGAdmin4 is a graphical user interface administration tool for PostgreSQL. The crunchy-pgadmin4 container executes the pgAdmin4 web application.
- **pgbadger** (crunchy-pgbadger). pgbadger is a PostgreSQL log analyzer with fully detailed reports and graphs. The crunchy-pgbadger container executes the pgBadger utility, which generates a PostgreSQL log analysis report using a small HTTP server running on the container.
- **pg_upgrade** (crunchy-upgrade). The crunchy-upgrade container contains 9.5, 9.6, 10, 11 and 12 PostgreSQL packages in order to perform a pg_upgrade from 9.5 to 9.6, 9.6 to 10, 10 to 11, and 11 to 12 versions.
- **scheduler** (crunchy-scheduler). The crunchy-scheduler container provides a cron like microservice for automating pgBaseBackup and pgBackRest backups within a single namespace.

Metrics and Monitoring

- **Metrics Collection** (crunchy-collect). The crunchy-collect container provides real time metrics about the PostgreSQL database via an API. These metrics are scraped and stored by a Prometheus time-series database and are then graphed and visualized through the open source data visualizer Grafana.
- **Grafana** (crunchy-grafana). Grafana is an open source Visual dashboards are created from the collected and stored data that crunchy-collect and crunchy-prometheus provide for the crunchy-grafana container, which hosts an open source web-based graphing dashboard called Grafana.
- **Prometheus** (crunchy-prometheus). Prometheus is a multi-dimensional time series data model with an elastic query language. It is used in collaboration with Crunchy Collect and Grafana to provide metrics.

Connection Pooling and Load Balancing

- **pgbouncer** (crunchy-pgbouncer). pgbouncer is a lightweight connection pooler for PostgreSQL. The crunchy-pgbouncer container provides a pgbouncer image.
- **pgpool** (crunchy-pgpool). pgPool II is a middleware that works between PostgreSQL servers and a PostgreSQL database client. The crunchy-pgpool container executes the 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.

PGO Command Line Interface (PGO CLI)

One of the suppport methods of interacting with the PostgreSQL Operator is through the command line tool, *pgo* CLI.

The PGO CLI is downloaded from the GitHub Releases page for the PostgreSQL Operator (https://github.com/crunchydata/postgres-operator/releases).

The *pgo* client is provided in Mac, Windows, and Linux binary formats, download the appropriate client to your local laptop or workstation to work with a remote Operator.

PGO CLI Syntax

Use the following syntax to run `pgo` commands from your terminal window:

```
pgo [command] ([TYPE] [NAME]) [flags]
```

Where *command* is a verb like:

- show
- create
- delete

And *type* is a resource type like:

- cluster
- policy
- user

And *name* is the name of the resource type like:

- mycluster
- somesqlpolicy
- john

To get detailed help information and command flag descriptions on each *pgo* command, enter:

```
pgo [command] -h
```

PGO CLI Operations

The following table shows the *pgo* operations currently implemented:

Operation	Syntax	Description
apply	pgo apply mypolicy –selector=name=mycluster	Apply a SQL policy on a Postgres cluster(s) that have a label m
backup	pgo backup mycluster	Perform a backup on a Postgres cluster(s)
create	pgo create cluster mycluster	Create an Operator resource type (e.g. cluster, policy, schedule,
delete	pgo delete cluster mycluster	Delete an Operator resource type (e.g. cluster, policy, user, sch
ls	pgo ls mycluster	Perform a Linux *ls* command on the cluster.
cat	pgo cat mycluster	Perform a Linux *ls* command on the cluster.
df	pgo df mycluster	Display the disk status/capacity of a Postgres cluster.

Operation	Syntax	Description
failover	pgo failover mycluster	Perform a manual failover of a Postgres cluster.
help	pgo help	Display general *pgo* help information.
label	pgo label mycluster –label=environment=prod	Create a metadata label for a Postgres cluster(s).
load	pgo load –load-config=load.json –selector=name=mycluster	Perform a data load into a Postgres cluster(s).
reload	pgo reload mycluster	Perform a pg_ctl reload command on a Postgres cluster(s).
restore	pgo restore mycluster	Perform a pgbackrest or pgdump restore on a Postgres cluster.
scale	pgo scale mycluster	Create a Postgres replica(s) for a given Postgres cluster.
scaledown	pgo scaledown mycluster –query	Delete a replica from a Postgres cluster.
show	pgo show cluster mycluster	Display Operator resource information (e.g. cluster, user, policy
status	pgo status	Display Operator status.
test	pgo test mycluster	Perform a SQL test on a Postgres cluster(s).
update	pgo update cluster –label=autofail=false	Update a Postgres cluster(s).
upgrade	pgo upgrade mycluster	Perform a minor upgrade to a Postgres cluster(s).
user	pgo user –selector=name=mycluster –update-passwords	Perform Postgres user maintenance on a Postgres cluster(s).
version	pgo version	Display Operator version information.

pgo CLI Global Environment Variables

pgo will pick up these settings if set in your environment:

Name	Description	NOTES
PGOUSERNAME	The username (role) used for auth on the operator apiserver.	Requires that PGOUSERPASS be set.
PGOUSERPASS	The password for used for auth on the operator apiserver.	Requires that PGOUSERNAME be set.
PGOUSER	The path the the pgouser file.	Will be ignored if either PGOUSERNAME or PGOU

pgo CLI Global Flags

pgo global command flags include:

Flag	Description
n	namespace targeted for the command
apiserver-url	URL of the Operator REST API service, override with CO_APISERVER_URL environment variable
debug	enable debug messages
pgo-ca-cert	The CA Certificate file path for authenticating to the PostgreSQL Operator apiserver. Override with PGO_CA_CERT e
pgo-client-cert	The Client Certificate file path for authenticating to the PostgreSQL Operator apiserver. Override with PGO_CLIENT_
pgo-client-key	The Client Key file path for authenticating to the PostgreSQL Operator apiserver. Override with PGO_CLIENT_KEY e

Common PGO CLI Operations

In all the examples below, the user is specifying the *pgouser1* namespace as the target of the operator. Replace this value with your own namespace value. You can specify a default namespace to be used by setting the PGO_NAMESPACE environment variable on the *pgo* client environment.

PostgreSQL Cluster Operations

Creating a Cluster A user will typically start using the PostgreSQL Operator by creating PostgreSQL cluster including a single PostgreSQL instance as follows:

```
pgo create cluster mycluster -n pgouser1
```

This command creates a PostgreSQL cluster with a single PostgreSQL instance in the *pgouser1* namespace.

You can see the PostgreSQL cluster using the following:

```
pgo show cluster mycluster -n pgouser1
```

You can test the PostgreSQL cluster by entering:

```
pgo test mycluster -n pgouser1
```

You can optionally add a PostgreSQL replica instance to your PostgreSQL cluster as follows:

```
pgo scale mycluster -n pgouser1
```

You can create a PostgreSQL cluster initially with a PostgreSQL replica as follows:

```
pgo create cluster mycluster --replica-count=1 -n pgouser1
```

You can cluster using the PostgreSQL + PostGIS container image:

```
pgo create cluster mygiscluster --ccp-image=crunchy-postgres-gis -n pgouser1
```

You can cluster using with a Custom ConfigMap:

```
pgo create cluster mycustomcluster --custom-config myconfigmap -n pgouser1
```

To view the PostgreSQL logs, you can enter commands such as:

```
pgo ls mycluster -n pgouser1 /pgdata/mycluster/pg_log
pgo cat mycluster -n pgouser1 /pgdata/mycluster/pg_log/postgresql-Mon.log | tail -3
```

Scaledown a Cluster You can remove a PostgreSQL replica using the following:

```
pgo scaledown mycluster --query -n pgouser1
pgo scaledown mycluster --target=sometarget -n pgouser1
```

Delete a Cluster You can remove a PostgreSQL cluster by entering:

```
pgo delete cluster mycluster -n pgouser1
```

This removes any PostgreSQL instances from being accessed as well as deletes all of its data and backups.

Retain Backups It can often be useful to keep the backups of a cluster even after its deleted, such as for archival purposes or for creating the cluster at a future date. You can delete the cluster but keep its backups using the `--keep-backups` flag:

```
pgo delete cluster mycluster --keep-backups -n pgouser1
```

Retain Cluster Data There are rare circumstances in which you may want to keep a copy of the original cluster data, such as when upgrading manually to a newer version of the Operator. In these cases, you can use the `--keep-data` flag:

```
pgo delete cluster mycluster --keep-data -n pgouser1
```

NOTE: The `--keep-data` flag is deprecated.

View Disk Utilization You can see a comparison of PostgreSQL data size versus the Persistent volume claim size by entering the following:

```
pgo df mycluster -n pgouser1
```

Backups

pgbackrest Operations By default the PostgreSQL Operator deploys *pgbackrest* backup for a PostgreSQL cluster to hold database backup data.

You can create a *pgbackrest* backup job as follows:

```
pgo backup mycluster -n pgouser1
```

You can optionally pass *pgbackrest* command options into the backup command as follows:

```
pgo backup mycluster --backup-type=pgbackrest --backup-opts="--type=diff" -n pgouser1
```

See pgbackrest documentation for *pgbackrest* command flag descriptions.

You can create a PostgreSQL cluster that does not include pgbackrest if you specify the following:

```
pgo create cluster mycluster --pgbackrest=false -n pgouser1
```

Perform a pgbasebackup backup Alternatively, you can perform a *pgbasebackup* job as follows:

```
pgo backup mycluster --backup-type=pgbasebackup -n pgouser1
```

```
pgo backup mycluster --backup-type=pgdump -n pgouser1
pgo backup mycluster --backup-type=pgdump --backup-opts="--dump-all --verbose" -n pgouser1
pgo backup mycluster --backup-type=pgdump --backup-opts="--schema=myschema" -n pgouser1
```

Note: To run pgdump_all instead of pgdump, pass '–dump-all' flag in –backup-opts as shown above. All –backup-opts should be space delimited.

```
pgo restore mycluster -n pgouser1
```

Or perform a restore based on a point in time:

```
pgo restore mycluster --pitr-target="2019-01-14 00:02:14.921404+00" --backup-opts="--type=time" -n
    pgouser1
```

You can also set the any of the pgbackrest restore options :

```
pgo restore mycluster --pitr-target="2019-01-14 00:02:14.921404+00" --backup-opts=" see pgbackrest
    options " -n pgouser1
```

You can also target specific nodes when performing a restore:

```
pgo restore mycluster --node-label=failure-domain.beta.kubernetes.io/zone=us-central1-a -n pgouser1
```

Here are some steps to test PITR:

- pgo create cluster mycluster
- create a table on the new cluster called *beforebackup*
- pgo backup mycluster -n pgouser1
- create a table on the cluster called *afterbackup*
- execute *select now()* on the database to get the time, use this timestamp minus a couple of minutes when you perform the restore
- pgo restore mycluster –pitr-target="2019-01-14 00:02:14.921404+00" –backup-opts="–type=time –log-level-console=info" -n pgouser1
- wait for the database to be restored
- execute *.* in the database and you should see the database state prior to where the* afterbackup* table was created

See the Design section of the Operator documentation for things to consider before you do a restore.

Restore from pgbasebackup You can find available pgbasebackup backups to use for a pgbasebackup restore using the `pgo show backup` command:

```
$ pgo show backup mycluster --backup-type=pgbasebackup -n pgouser1 | grep "Backup Path"
        Backup Path:    mycluster-backups/2019-05-21-09-53-20
        Backup Path:    mycluster-backups/2019-05-21-06-58-50
        Backup Path:    mycluster-backups/2019-05-21-09-52-52
```

You can then perform a restore using any available backup path:

```
pgo restore mycluster --backup-type=pgbasebackup --backup-path=mycluster/2019-05-21-06-58-50
   --backup-pvc=mycluster-backup -n pgouser1
```

When performing the restore, both the backup path and backup PVC can be omitted, and the Operator will use the last pgbasebackup backup created, along with the PVC utilized for that backup:

```
pgo restore mycluster --backup-type=pgbasebackup -n pgouser1
```

Once the pgbasebackup restore is complete, a new PVC will be available with a randomly generated ID that contains the restored database, e.g. PVC **mycluster-ieqe** in the output below:

```
$ pgo show pvc --all
All Operator Labeled PVCs
        mycluster
        mycluster-backup
        mycluster-ieqe
```

A new cluster can then be created with the same name as the new PVC, as well with the secrets from the original cluster, in order to deploy a new cluster using the restored database:

```
pgo create cluster mycluster-ieqe --secret-from=mycluster
```

If you would like to control the name of the PVC created when performing a pgbasebackup restore, use the `--restore-to-pvc` flag:

```
pgo restore mycluster --backup-type=pgbasebackup --restore-to-pvc=mycluster-restored -n pgouser1
```

```
pgo restore mycluster --backup-type=pgdump --backup-pvc=mycluster-pgdump-pvc
   --pitr-target="2019-01-15-00-03-25" -n pgouser1
```

To restore the most recent pgdump at the default path, leave off a timestamp:

```
pgo restore mycluster --backup-type=pgdump --backup-pvc=mycluster-pgdump-pvc -n pgouser1
```

Label Operations

Apply a Label to a PostgreSQL Cluster You can apply a Kubernetes label to a PostgreSQL cluster as follows:

```
pgo label mycluster --label=environment=prod -n pgouser1
```

In this example, the label key is *environment* and the label value is *prod*.

You can apply labels across a category of PostgreSQL clusters by using the *–selector* command flag as follows:

```
pgo label --selector=clustertypes=research --label=environment=prod -n pgouser1
```

In this example, any PostgreSQL cluster with the label of *clustertypes=research* will have the label *environment=prod* set.

In the following command, you can also view PostgreSQL clusters by using the *–selector* command flag which specifies a label key value to search with:

```
pgo show cluster --selector=environment=prod -n pgouser1
```

Policy Operations

Create a Policy To create a SQL policy, enter the following:

```
pgo create policy mypolicy --in-file=mypolicy.sql -n pgouser1
```

This examples creates a policy named *mypolicy* using the contents of the file *mypolicy.sql* which is assumed to be in the current directory.

You can view policies as following:

```
pgo show policy --all -n pgouser1
```

```
pgo apply mypolicy --selector=environment=prod
pgo apply mypolicy --selector=name=mycluster
```

Operator Status

Show Operator Version To see what version of the PostgreSQL Operator client and server you are using, enter:

```
pgo version
```

To see the PostgreSQL Operator server status, enter:

```
pgo status -n pgouser1
```

To see the PostgreSQL Operator server configuration, enter:

```
pgo show config -n pgouser1
```

To see what namespaces exist and if you have access to them, enter:

```
pgo show namespace -n pgouser1
```

Fail-over Operations

To perform a manual failover, enter the following:

```
pgo failover mycluster --query -n pgouser1
```

That example queries to find the available Postgres replicas that could be promoted to the primary.

```
pgo failover mycluster --target=sometarget -n pgouser1
```

That command chooses a specific target, and starts the failover workflow.

Create a Cluster with Auto-fail Enabled To support an automated failover, you can specify the *–autofail* flag on a Postgres cluster when you create it as follows:

```
pgo create cluster mycluster --autofail --replica-count=1 -n pgouser1
```

You can set the auto-fail flag on a Postgres cluster after it is created by the following command:

```
pgo update cluster --autofail=false -n pgouser1
pgo update cluster --autofail=true -n pgouser1
```

Note that if you do a pgbackrest restore, you will need to reset the autofail flag to true after the restore is completed.

Configuring pgbouncer, pgpool or pgbadger to Clusters

pgbouncer Deployment and Configuration To add a pgbouncer Deployment to your PostgreSQL cluster, enter:

```
pgo create cluster mycluster --pgbouncer -n pgouser1
```

You can add pgbouncer after a PostgreSQL cluster is created as follows:

```
pgo create pgbouncer mycluster
pgo create pgbouncer --selector=name=mycluster
```

You can also specify a pgbouncer password as follows:

```
pgo create cluster mycluster --pgbouncer --pgbouncer-pass=somepass -n pgouser1
```

Note, the pgbouncer configuration defaults to specifying only a single entry for the primary database. If you want it to have an entry for the replica service, add the following configuration to pgbouncer.ini:

```
{{.PG_REPLICA_SERVICE_NAME}} = host={{.PG_REPLICA_SERVICE_NAME}} port={{.PG_PORT}}
   auth_user={{.PG_USERNAME}} dbname={{.PG_DATABASE}}
```

You can remove a pgbouncer from a cluster as follows:

```
pgo delete pgbouncer mycluster -n pgouser1
```

pgpool Deployment and Configuration To add a pgpool Deployment to your PostgreSQL cluster, enter:

```
pgo create cluster mycluster --pgpool -n pgouser1
```

You can also add a pgpool to a PostgreSQL cluster after initial creation as follows:

```
pgo create pgpool mycluster -n pgouser1
```

You can remove a pgpool from a PostgreSQL cluster as follows:

```
pgo delete pgpool mycluster -n pgouser1
```

pgbadger Deployment You can create a pgbadger sidecar container in your PostgreSQL cluster pod as follows:

```
pgo create cluster mycluster --pgbadger -n pgouser1
```

Metrics Collection Deployment and Configuration Likewise, you can add the Crunchy Collect Metrics sidecar container into your PostgresQL cluster pod as follows:

```
pgo create cluster mycluster --metrics -n pgouser1
```

Note: backend metric storage such as Prometheus and front end visualization software such as Grafana are not created automatically by the PostgreSQL Operator. For instructions on installing Grafana and Prometheus in your environment, see the Crunchy Container Suite documentation.

Scheduled Tasks

There is a cron based scheduler included into the PostgreSQL Operator Deployment by default.

You can create automated full pgBackRest backups every Sunday at 1 am as follows:

```
pgo create schedule mycluster --schedule="0 1 * * SUN" \
    --schedule-type=pgbackrest --pgbackrest-backup-type=full -n pgouser1
```

You can create automated diff pgBackRest backups every Monday-Saturday at 1 am as follows:

```
pgo create schedule mycluster --schedule="0 1 * * MON-SAT" \
    --schedule-type=pgbackrest --pgbackrest-backup-type=diff -n pgouser1
```

You can create automated pgBaseBackup backups every day at 1 am as follows:

In order to have a backup PVC created, users should run the `pgo backup` command against the target cluster prior to creating this schedule.

```
pgo create schedule mycluster --schedule="0 1 * * *" \
    --schedule-type=pgbasebackup --pvc-name=mycluster-backup -n pgouser1
```

You can create automated Policy every day at 1 am as follows:

```
pgo create schedule --selector=pg-cluster=mycluster --schedule="0 1 * * *" \
     --schedule-type=policy --policy=mypolicy --database=userdb \
     --secret=mycluster-testuser-secret -n pgouser1
```

Benchmark Clusters with pgbench

The pgbench utility containerized and made available to PostgreSQL Operator users.

To create a Benchmark via Cluster Name you enter:

```
pgo benchmark mycluster -n pgouser1
```

To create a Benchmark via Selector, enter:

```
pgo benchmark --selector=pg-cluster=mycluster -n pgouser1
```

To create a Benchmark with a custom transactions, enter:

```
pgo create policy --in-file=/tmp/transactions.sql mytransactions -n pgouser1
pgo benchmark mycluster --policy=mytransactions -n pgouser1
```

To create a Benchmark with custom parameters, enter:

```
pgo benchmark mycluster --clients=10 --jobs=2 --scale=10 --transactions=100 -n pgouser1
```

You can view benchmarks by entering:

```
pgo show benchmark -n pgouser1
```

Complex Deployments

```
pgo create cluster mycluster --storage-config=somestorageconfig -n pgouser1
```

Likewise, you can specify a storage configuration when creating a replica:

```
pgo scale mycluster --storage-config=someslowerstorage -n pgouser1
```

This example specifies the *somestorageconfig* storage configuration to be used by the PostgreSQL cluster. This lets you specify a storage configuration that is defined in the *pgo.yaml* file specifically for a given PostgreSQL cluster.

You can create a PostgreSQL Cluster using a Preferred Node as follows:

```
pgo create cluster mycluster --node-label=speed=superfast -n pgouser1
```

That command will cause a node affinity rule to be added to the PostgreSQL pod which will influence the node upon which Kubernetes will schedule the Pod.

Likewise, you can create a Replica using a Preferred Node as follows:

```
pgo scale mycluster --node-label=speed=slowerthannormal -n pgouser1
```

```
pgo create cluster mycluster --service-type=LoadBalancer -n pgouser1
```

This command will cause the PostgreSQL Service to be of a specific type instead of the default ClusterIP service type.

User Management

```
pgo create user mycluster --username=someuser --password=somepassword --valid-days=10
```

This command will create a Postgres user on `mycluster` using the given username and password. You can add the `--managed` flag and the user will be managed by the operator. This means that a kubernetes secret will be created along with the Postgres user. Any users created with the `create user` command will automatically have access to all databases that were created when the cluster was created. You will need to manually update their privliges either by using an SQL policy or by using psql if you want to restrict access.

```
pgo update user mycluster --username=someuser --password=updatedpass
```

This command allows you to update the password for the given user on a cluster. The update user command also allows you to manage when users will expire.

```
pgo update user mycluster --username=someuser --valid-days=40
```

```
pgo delete user mycluster --username=someuser
```

This command will delete the give user from `mycluster`. You can delete the user from all clusters by using the `--all` flag instead of the cluster name.

The command line tool, `pgo`, is used to interact with the Postgres Operator.

Most users will work with the Operator using the *pgo* CLI tool. That tool is downloaded from the GitHub Releases page for the Operator.

The *pgo* client is provided in Mac, Windows, and Linux binary formats, download the appropriate client to your local laptop or workstation to work with a remote Operator.

Syntax

Use the following syntax to run `pgo` commands from your terminal window:

```
pgo [command] ([TYPE] [NAME]) [flags]
```

Where *command* is a verb like:

- show
- create
- delete

And *type* is a resource type like:

- cluster
- policy
- user

And *name* is the name of the resource type like:

- mycluster
- somesqlpolicy
- john

To get detailed help information and command flag descriptions on each *pgo* command, enter:

```
pgo [command] -h
```

Operations

The following table shows the *pgo* operations currently implemented:

Operation	Syntax	Description
apply	`pgo apply mypolicy --selector=name=mycluster`	Apply a SQL policy on a Postgres cluster(s) that hav
backup	`pgo backup mycluster`	Perform a backup on a Postgres cluster(s)
create	`pgo create cluster mycluster`	Create an Operator resource type (e.g. cluster, policy
delete	`pgo delete cluster mycluster`	Delete an Operator resource type (e.g. cluster, policy,
ls	`pgo ls mycluster filepath`	Perform a Linux `ls` command on the cluster.
cat	`pgo cat mycluster filepath`	Perform a Linux `cat` command on the cluster.
df	`pgo df mycluster`	Display the disk status/capacity of a Postgres cluster
failover	`pgo failover mycluster`	Perform a manual failover of a Postgres cluster.
help	`pgo help`	Display general `pgo` help information.
label	`pgo label mycluster --label=environment=prod`	Create a metadata label for a Postgres cluster(s).
load	`pgo load --load-config=load.json --selector=name=mycluster`	Perform a data load into a Postgres cluster(s).
reload	`pgo reload mycluster`	Perform a `pg_ctl` reload command on a Postgres clus
restore	`pgo restore mycluster`	Perform a `pgbackrest`, `pgbasebackup` or `pgdump` rest
scale	`pgo scale mycluster`	Create a Postgres replica(s) for a given Postgres clust
scaledown	`pgo scaledown mycluster --query`	Delete a replica from a Postgres cluster.
show	`pgo show cluster mycluster`	Display Operator resource information (e.g. cluster, u
status	`pgo status`	Display Operator status.
test	`pgo test mycluster`	Perform a SQL test on a Postgres cluster(s).
update	`pgo update cluster mycluster --disable-autofail`	Update a Postgres cluster(s), pgouser, pgorole, user, c
upgrade	`pgo upgrade mycluster`	Perform a minor upgrade to a Postgres cluster(s).
version	`pgo version`	Display Operator version information.

Common Operations

In all the examples below, the user is specifying the *pgouser1* namespace as the target of the operator. Replace this value with your own namespace value. You can specify a default namespace to be used by setting the PGO_NAMESPACE environment variable on the *pgo* client environment.

Cluster Operations

A user will typically start using the Operator by creating a Postgres cluster as follows:

```
pgo create cluster mycluster -n pgouser1
```

This command creates a Postgres cluster in the *pgouser1* namespace that has a single Postgres primary container.

You can see the Postgres cluster using the following:

```
pgo show cluster mycluster -n pgouser1
```

You can test the Postgres cluster by entering:

```
pgo test mycluster -n pgouser1
```

You can optionally add a Postgres replica to your Postgres cluster as follows:

```
pgo scale mycluster -n pgouser1
```

You can create a Postgres cluster initially with a Postgres replica as follows:

```
pgo create cluster mycluster --replica-count=1 -n pgouser1
```

To view the Postgres logs, you can enter commands such as:

```
pgo ls mycluster -n pgouser1 /pgdata/mycluster/pg_log
pgo cat mycluster -n pgouser1 /pgdata/mycluster/pg_log/postgresql-Mon.log | tail -3
```

Backups By default the Operator deploys pgbackrest for a Postgres cluster to hold database backup data.

You can create a pgbackrest backup job as follows:

```
pgo backup mycluster -n pgouser1
```

You can perform a pgbasebackup job as follows:

```
pgo backup mycluster --backup-type=pgbasebackup -n pgouser1
```

You can optionally pass pgbackrest command options into the backup command as follows:

```
pgo backup mycluster --backup-type=pgbackrest --backup-opts="--type=diff" -n pgouser1
```

See pgbackrest.org for command flag descriptions.

You can create a Postgres cluster that does not include pgbackrest if you specify the following:

```
pgo create cluster mycluster --pgbackrest=false -n pgouser1
```

You can show the current backups on a cluster with the following:

```
pgo show backup mycluster -n pgouser1
```

Scaledown a Cluster You can remove a Postgres replica using the following:

```
pgo scaledown mycluster --query -n pgouser1
pgo scaledown mycluster --target=sometarget -n pgouser1
```

Delete a Cluster You can remove a PostgreSQL cluster by entering:

```
pgo delete cluster mycluster -n pgouser1
```

This removes any PostgreSQL instances from being accessed as well as deletes all of its data and backups.

Retain Backups It can often be useful to keep the backups of a cluster even after its deleted, such as for archival purposes or for creating the cluster at a future date. You can delete the cluster but keep its backups using the `--keep-backups` flag:

```
pgo delete cluster mycluster --keep-backups -n pgouser1
```

Retain Cluster Data There are rare circumstances in which you may want to keep a copy of the original cluster data, such as when upgrading manually to a newer version of the Operator. In these cases, you can use the `--keep-data` flag:

```
pgo delete cluster mycluster --keep-data -n pgouser1
```

NOTE: The `--keep-data` flag is deprecated.

View Disk Utilization You can see a comparison of Postgres data size versus the Persistent volume claim size by entering the following:

```
pgo df mycluster -n pgouser1
```

Label Operations

Apply a Label to a Cluster You can apply a Kubernetes label to a Postgres cluster as follows:

```
pgo label mycluster --label=environment=prod -n pgouser1
```

In this example, the label key is *environment* and the label value is *prod*.

You can apply labels across a category of Postgres clusters by using the *–selector* command flag as follows:

```
pgo label --selector=clustertypes=research --label=environment=prod -n pgouser1
```

In this example, any Postgres cluster with the label of *clustertypes=research* will have the label *environment=prod* set.

In the following command, you can also view Postgres clusters by using the *–selector* command flag which specifies a label key value to search with:

```
pgo show cluster --selector=environment=prod -n pgouser1
```

Policy Operations

Create a Policy To create a SQL policy, enter the following:

```
pgo create policy mypolicy --in-file=mypolicy.sql -n pgouser1
```

This examples creates a policy named *mypolicy* using the contents of the file *mypolicy.sql* which is assumed to be in the current directory.

You can view policies as following:

```
pgo show policy --all -n pgouser1
```

```
pgo apply mypolicy --selector=environment=prod
pgo apply mypolicy --selector=name=mycluster
```

Operator Status

Show Operator Version To see what version of the Operator client and server you are using, enter:

```
pgo version
```

To see the Operator server status, enter:

```
pgo status -n pgouser1
```

To see the Operator server configuration, enter:

```
pgo show config -n pgouser1
```

To see what namespaces exist and if you have access to them, enter:

```
pgo show namespace pgouser1
```

```
pgo backup mycluster --backup-type=pgdump -n pgouser1
pgo backup mycluster --backup-type=pgdump --backup-opts="--dump-all --verbose" -n pgouser1
pgo backup mycluster --backup-type=pgdump --backup-opts="--schema=myschema" -n pgouser1
```

Note: To run `pgdump_all` instead of `pgdump`, pass `--dump-all` flag in `--backup-opts` as shown above. All `--backup-opts` should be space delimited.

```
pgo restore mycluster -n pgouser1
```

Or perform a restore based on a point in time:

```
pgo restore mycluster --pitr-target="2019-01-14 00:02:14.921404+00" --backup-opts="--type=time" -n
   pgouser1
```

You can also set the any of the pgbackrest restore options :

```
pgo restore mycluster --pitr-target="2019-01-14 00:02:14.921404+00" --backup-opts=" see pgbackrest
   options " -n pgouser1
```

You can also target specific nodes when performing a restore:

```
pgo restore mycluster --node-label=failure-domain.beta.kubernetes.io/zone=us-central1-a -n pgouser1
```

Here are some steps to test PITR:

- `pgo create cluster mycluster`
- Create a table on the new cluster called *beforebackup*
- pgo backup mycluster -n pgouser1
- create a table on the cluster called *afterbackup*
- Execute *select now()* on the database to get the time, use this timestamp minus a couple of minutes when you perform the restore
- `pgo restore mycluster --pitr-target="2019-01-14 00:02:14.921404+00" --backup-opts="--type=time --log-level-cons -n pgouser1`
- Wait for the database to be restored
- Execute ** in the database and you should see the database state prior to where the* afterbackup* table was created

See the Design section of the Operator documentation for things to consider before you do a restore.

Restore from pgbasebackup You can find available pgbasebackup backups to use for a pgbasebackup restore using the `pgo show backup` command:

```
$ pgo show backup mycluster --backup-type=pgbasebackup -n pgouser1 | grep "Backup Path"
        Backup Path:    mycluster-backups/2019-05-21-09-53-20
        Backup Path:    mycluster-backups/2019-05-21-06-58-50
        Backup Path:    mycluster-backups/2019-05-21-09-52-52
```

You can then perform a restore using any available backup path:

```
pgo restore mycluster --backup-type=pgbasebackup --backup-path=mycluster/2019-05-21-06-58-50
   --backup-pvc=mycluster-backup -n pgouser1
```

When performing the restore, both the backup path and backup PVC can be omitted, and the Operator will use the last pgbasebackup backup created, along with the PVC utilized for that backup:

```
pgo restore mycluster --backup-type=pgbasebackup -n pgouser1
```

Once the pgbasebackup restore is complete, a new PVC will be available with a randomly generated ID that contains the restored database, e.g. PVC **mycluster-ieqe** in the output below:

```
$ pgo show pvc --all
All Operator Labeled PVCs
        mycluster
        mycluster-backup
        mycluster-ieqe
```

A new cluster can then be created with the same name as the new PVC, as well with the secrets from the original cluster, in order to deploy a new cluster using the restored database:

```
pgo create cluster mycluster-ieqe --secret-from=mycluster
```

If you would like to control the name of the PVC created when performing a pgbasebackup restore, use the `--restore-to-pvc` flag:

```
pgo restore mycluster --backup-type=pgbasebackup --restore-to-pvc=mycluster-restored -n pgouser1
pgo restore mycluster --backup-type=pgdump --backup-pvc=mycluster-pgdump-pvc
   --pitr-target="2019-01-15-00-03-25" -n pgouser1
```

To restore the most recent pgdump at the default path, leave off a timestamp:

```
pgo restore mycluster --backup-type=pgdump --backup-pvc=mycluster-pgdump-pvc -n pgouser1
```

Fail-over Operations

To perform a manual failover, enter the following:

```
pgo failover mycluster --query -n pgouser1
```

That example queries to find the available Postgres replicas that could be promoted to the primary.

```
pgo failover mycluster --target=sometarget -n pgouser1
```

That command chooses a specific target, and starts the failover workflow.

Create a Cluster with Auto-fail Enabled To support an automated failover, you can specify the *–autofail* flag on a Postgres cluster when you create it as follows:

```
pgo create cluster mycluster --autofail=true --replica-count=1 -n pgouser1
```

You can set the auto-fail flag on a Postgres cluster after it is created by the following command:

```
pgo update cluster --autofail=false -n pgouser1
pgo update cluster --autofail=true -n pgouser1
```

Note that if you do a pgbackrest restore, you will need to reset the autofail flag to true after the restore is completed.

Add-On Operations

To add a pgbouncer Deployment to your Postgres cluster, enter:

```
pgo create cluster mycluster --pgbouncer -n pgouser1
```

You can add pgbouncer after a Postgres cluster is created as follows:

```
pgo create pgbouncer mycluster
pgo create pgbouncer --selector=name=mycluster
```

You can also specify a pgbouncer password as follows:

```
pgo create cluster mycluster --pgbouncer --pgbouncer-pass=somepass -n pgouser1
```

Note, the pgbouncer configuration defaults to specifying only a single entry for the primary database. If you want it to have an entry for the replica service, add the following configuration to pgbouncer.ini:

```
{{.PG_REPLICA_SERVICE_NAME}} = host={{.PG_REPLICA_SERVICE_NAME}} port={{.PG_PORT}}
   auth_user={{.PG_USERNAME}} dbname={{.PG_DATABASE}}
```

To add a pgpool Deployment to your Postgres cluster, enter:

```
pgo create cluster mycluster --pgpool -n pgouser1
```

You can also add a pgpool to a cluster after initial creation as follows:

```
pgo create pgpool mycluster -n pgouser1
```

You can remove a pgbouncer or pgpool from a cluster as follows:

```
pgo delete pgbouncer mycluster -n pgouser1
pgo delete pgpool mycluster -n pgouser1
```

You can create a pgbadger sidecar container in your Postgres cluster pod as follows:

```
pgo create cluster mycluster --pgbadger -n pgouser1
```

Likewise, you can add the Crunchy Collect Metrics sidecar container into your Postgres cluster pod as follows:

```
pgo create cluster mycluster --metrics -n pgouser1
```

Note: backend metric storage such as Prometheus and front end visualization software such as Grafana are not created automatically by the PostgreSQL Operator. For instructions on installing Grafana and Prometheus in your environment, see the Crunchy Container Suite documentation.

Scheduled Tasks

There is a cron based scheduler included into the Operator Deployment by default.

You can create automated full pgBackRest backups every Sunday at 1 am as follows:

```
pgo create schedule mycluster --schedule="0 1 * * SUN" \
    --schedule-type=pgbackrest --pgbackrest-backup-type=full -n pgouser1
```

You can create automated diff pgBackRest backups every Monday-Saturday at 1 am as follows:

```
pgo create schedule mycluster --schedule="0 1 * * MON-SAT" \
    --schedule-type=pgbackrest --pgbackrest-backup-type=diff -n pgouser1
```

You can create automated pgBaseBackup backups every day at 1 am as follows:

In order to have a backup PVC created, users should run the `pgo backup` command against the target cluster prior to creating this schedule.

```
pgo create schedule mycluster --schedule="0 1 * * *" \
    --schedule-type=pgbasebackup --pvc-name=mycluster-backup -n pgouser1
```

You can create automated Policy every day at 1 am as follows:

```
pgo create schedule --selector=pg-cluster=mycluster --schedule="0 1 * * *" \
     --schedule-type=policy --policy=mypolicy --database=userdb \
     --secret=mycluster-testuser-secret -n pgouser1
```

Benchmark Clusters

The pgbench utility containerized and made available to Operator users.

To create a Benchmark via Cluster Name you enter:

```
pgo benchmark mycluster -n pgouser1
```

To create a Benchmark via Selector, enter:

```
pgo benchmark --selector=pg-cluster=mycluster -n pgouser1
```

To create a Benchmark with a custom transactions, enter:

```
pgo create policy --in-file=/tmp/transactions.sql mytransactions -n pgouser1
pgo benchmark mycluster --policy=mytransactions -n pgouser1
```

To create a Benchmark with custom parameters, enter:

```
pgo benchmark mycluster --clients=10 --jobs=2 --scale=10 --transactions=100 -n pgouser1
```

You can view benchmarks by entering:

```
pgo show benchmark -n pgouser1 mycluster
```

Complex Deployments

```
pgo create cluster mycluster --storage-config=somestorageconfig -n pgouser1
```

Likewise, you can specify a storage configuration when creating a replica:

```
pgo scale mycluster --storage-config=someslowerstorage -n pgouser1
```

This example specifies the *somestorageconfig* storage configuration to be used by the Postgres cluster. This lets you specify a storage configuration that is defined in the *pgo.yaml* file specifically for a given Postgres cluster.

You can create a Cluster using a Preferred Node as follows:

```
pgo create cluster mycluster --node-label=speed=superfast -n pgouser1
```

That command will cause a node affinity rule to be added to the Postgres pod which will influence the node upon which Kubernetes will schedule the Pod.

Likewise, you can create a Replica using a Preferred Node as follows:

```
pgo scale mycluster --node-label=speed=slowerthannormal -n pgouser1
```

```
pgo create cluster mycluster --service-type=LoadBalancer -n pgouser1
```

This command will cause the Postgres Service to be of a specific type instead of the default ClusterIP service type.

Namespace Operations Create an Operator namespace where Postgres clusters can be created and managed by the Operator:

```
pgo create namespace mynamespace
```

Update a Namespace to be able to be used by the Operator:

```
pgo update namespace somenamespace
```

Delete a Namespace:

```
pgo delete namespace mynamespace
```

PGO User Operations PGO users are users defined for authenticating to the PGO REST API. You can manage those users with the following commands:

```
pgo create pgouser someuser --pgouser-namespaces="pgouser1,pgouser2"
   --pgouser-password="somepassword" --pgouser-roles="pgoadmin"
pgo create pgouser otheruser --all-namespaces --pgouser-password="somepassword"
   --pgouser-roles="pgoadmin"
```

Update a user:

```
pgo update pgouser someuser --pgouser-namespaces="pgouser1,pgouser2"
   --pgouser-password="somepassword" --pgouser-roles="pgoadmin"
pgo update pgouser otheruser --all-namespaces --pgouser-password="somepassword"
   --pgouser-roles="pgoadmin"
```

Delete a PGO user:

```
pgo delete pgouser someuser
```

PGO roles are also managed as follows:

```
pgo create pgorole somerole --permissions="Cat,Ls"
```

Delete a PGO role with:

```
pgo delete pgorole somerole
```

Update a PGO role with:

```
pgo update pgorole somerole --permissions="Cat,Ls"
```

Postgres User Operations Managed Postgres users can be viewed using the following command:

```
pgo show user mycluster
```

Postgres users can be created using the following command examples:

```
pgo create user mycluster --username=somepguser --password=somepassword --managed
pgo create user --selector=name=mycluster --username=somepguser --password=somepassword --managed
```

Those commands are identical in function, and create on the mycluster Postgres cluster, a user named *somepguser*, with a password of *somepassword*, the account is *managed* meaning that these credentials are stored as a Secret on the Kube cluster in the Operator namespace.

Postgres users can be deleted using the following command:

```
pgo delete user mycluster --username=somepguser
```

That command deletes the user on the mycluster Postgres cluster.

Postgres users can be updated using the following command:

```
pgo update user mycluster --username=somepguser --password=frodo
```

That command changes the password for the user on the mycluster Postgres cluster.

Miscellaneous Create a cluster using the Crunchy Postgres + PostGIS container image:

```
pgo create cluster mygiscluster --ccp-image=crunchy-postgres-gis -n pgouser1
```

Create a cluster with a Custom ConfigMap:

```
pgo create cluster mycustomcluster --custom-config myconfigmap -n pgouser1
```

pgo Global Flags

pgo global command flags include:

Flag	Description
`-n`	namespace targeted for the command
`--apiserver-url`	URL of the Operator REST API service, override with `CO_APISERVER_URL` environment variable
`--debug`	Enable debug messages
`--pgo-ca-cert`	The CA Certificate file path for authenticating to the PostgreSQL Operator apiserver. Override with `PGO_CA_CERT`
`--pgo-client-cert`	The Client Certificate file path for authenticating to the PostgreSQL Operator apiserver. Override with `PGO_CLIEN`
`--pgo-client-key`	The Client Key file path for authenticating to the PostgreSQL Operator apiserver. Override with `PGO_CLIENT_KEY`

pgo Global Environment Variables

pgo will pick up these settings if set in your environment:

Name	Description	NOTES
`PGOUSERNAME`	The username (role) used for auth on the operator apiserver.	Requires that `PGOUSERPASS` be set.
`PGOUSERPASS`	The password for used for auth on the operator apiserver.	Requires that `PGOUSERNAME` be set.
`PGOUSER`	The path to the pgouser file.	Will be ignored if either `PGOUSERNAME` or `PGOUSERPASS` are

pgo

The pgo command line interface.

Synopsis

The pgo command line interface lets you create and manage PostgreSQL clusters.

Options

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -h, --help                     help for pgo
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo apply - Apply a policy
- pgo backup - Perform a Backup
- pgo benchmark - Perform a pgBench benchmark against clusters
- pgo cat - Perform a cat command on a cluster
- pgo create - Create a Postgres Operator resource
- pgo delete - Delete an Operator resource
- pgo df - Display disk space for clusters
- pgo failover - Performs a manual failover
- pgo label - Label a set of clusters
- pgo load - Perform a data load
- pgo ls - Perform a ls command on a cluster
- pgo reload - Perform a cluster reload
- pgo restore - Perform a restore from previous backup
- pgo scale - Scale a PostgreSQL cluster
- pgo scaledown - Scale down a PostgreSQL cluster
- pgo show - Show the description of a cluster
- pgo status - Display PostgreSQL cluster status
- pgo test - Test cluster connectivity
- pgo update - Update a pgouser, pgorole, or cluster
- pgo upgrade - Perform an upgrade
- pgo version - Print version information for the PostgreSQL Operator
- pgo watch - Print watch information for the PostgreSQL Operator

Auto generated by spf13/cobra on 4-Oct-2019

pgo apply

Apply a policy

Synopsis

APPLY allows you to apply a Policy to a set of clusters. For example:

```
pgo apply mypolicy1 --selector=name=mycluster
pgo apply mypolicy1 --selector=someotherpolicy
pgo apply mypolicy1 --selector=someotherpolicy --dry-run
```

```
pgo apply [flags]
```

Options

```
      --dry-run            Shows the clusters that the label would be applied to, without labelling
         them.
  -h, --help               help for apply
  -s, --selector string    The selector to use for cluster filtering.
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo - The pgo command line interface.

Auto generated by spf13/cobra on 4-Oct-2019

pgo backup

Perform a Backup

Synopsis

BACKUP performs a Backup, for example:

pgo backup mycluster

```
pgo backup [flags]
```

Options

```
      --backup-opts string               The pgbackup options to pass into pgbasebackup or
         pgbackrest.
      --backup-type string               The backup type to perform. Default is pgbasebackup.
         Valid backup types are pgbasebackup, pgbackrest and pgdump. (default "pgbackrest")
  -h, --help                             help for backup
      --pgbackrest-storage-type string   The type of storage to use when scheduling pgBackRest
         backups. Either "local", "s3" or both, comma separated. (default "local")
      --pvc-name string                  The PVC name to use for the backup instead of the default.
  -s, --selector string                  The selector to use for cluster filtering.
      --storage-config string            The name of a Storage config in pgo.yaml to use for the
         cluster storage.  Only applies to pgbasebackup backups.
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo - The pgo command line interface.

Auto generated by spf13/cobra on 4-Oct-2019

pgo benchmark

Perform a pgBench benchmark against clusters

Synopsis

Benchmark run pgBench against PostgreSQL clusters, for example:

pgo benchmark mycluster

```
pgo benchmark [flags]
```

Options

```
  -b, --benchmark-opts string   The extra flags passed to pgBench during the benchmark.
  -c, --clients int             The number of clients to be used in the benchmark. (default 1)
  -d, --database string         The database where the benchmark should be run. (default
     "postgres")
  -h, --help                    help for benchmark
  -i, --init-opts string        The extra flags passed to pgBench during the initialization of the
     benchmark.
  -j, --jobs int                The number of worker threads to use for the benchmark. (default 1)
  -p, --policy string           The name of the policy specifying custom transaction SQL for
     advanced benchmarking.
      --scale int               The number to scale the amount of rows generated for the
         benchmark. (default 1)
  -s, --selector string         The selector to use for cluster filtering.
  -t, --transactions int        The number of transaction each client should run in the benchmark.
     (default 1)
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo - The pgo command line interface.

Auto generated by spf13/cobra on 4-Oct-2019

pgo cat

Perform a cat command on a cluster

Synopsis

CAT performs a Linux cat command on a cluster file. For example:

```
pgo cat mycluster /pgdata/mycluster/postgresql.conf
```

```
pgo cat [flags]
```

Options

```
  -h, --help   help for cat
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo - The pgo command line interface.

Auto generated by spf13/cobra on 4-Oct-2019

pgo clidoc

Generate Markdown of CLI commandes

Synopsis

The clidoc command allows you to generate markdown files for all CLI commands:

```
pgo clidoc
```

```
pgo clidoc [flags]
```

Options

```
  -h, --help   help for clidoc
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
      --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo - The pgo command line interface.

Auto generated by spf13/cobra on 21-Feb-2019

pgo create

Create a Postgres Operator resource

Synopsis

CREATE allows you to create a new Operator resource. For example: pgo create cluster pgo create pgbouncer pgo create pgpool pgo create pgouser pgo create pgorole pgo create policy pgo create namespace pgo create user

```
pgo create [flags]
```

Options

```
  -h, --help   help for create
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo - The pgo command line interface.
- pgo create cluster - Create a PostgreSQL cluster
- pgo create namespace - Create a namespace
- pgo create pgbouncer - Create a pgbouncer
- pgo create pgorole - Create a pgorole
- pgo create pgouser - Create a pgouser
- pgo create pgpool - Create a pgpool
- pgo create policy - Create a SQL policy
- pgo create schedule - Create a cron-like scheduled task
- pgo create user - Create a PostgreSQL user

Auto generated by spf13/cobra on 4-Oct-2019

pgo create cluster

Create a PostgreSQL cluster

Synopsis

Create a PostgreSQL cluster consisting of a primary and a number of replica backends. For example:

```
pgo create cluster mycluster
```

```
pgo create cluster [flags]
```

Options

```
      --autofail                          If set, will cause autofailover to be enabled on this
          cluster.
      --ccp-image string                  The CCPImage name to use for cluster creation. If
          specified, overrides the value crunchy-postgres.
  -c, --ccp-image-tag string              The CCPImageTag to use for cluster creation. If
      specified, overrides the pgo.yaml setting.
      --custom-config string              The name of a configMap that holds custom PostgreSQL
          configuration files used to override defaults.
  -h, --help                              help for cluster
  -l, --labels string                     The labels to apply to this cluster.
      --metrics                           Adds the crunchy-collect container to the database pod.
      --node-label string                 The node label (key=value) to use in placing the primary
          database. If not set, any node is used.
  -w, --password string                   The password to use for initial database users.
      --pgbackrest string                 Enables a pgBackRest volume for the database pod, "true"
          or "false". Default from pgo.yaml, command line overrides default.
      --pgbackrest-storage-type string   The type of storage to use with pgBackRest. Either
          "local", "s3" or both, comma separated. (default "local")
      --pgbadger                          Adds the crunchy-pgbadger container to the database pod.
      --pgbouncer                         Adds a crunchy-pgbouncer deployment to the cluster.
      --pgbouncer-pass string             Password for the pgbouncer user of the crunchy-pgboucer
          deployment.
      --pgpool                            Adds the crunchy-pgpool container to the database pod.
      --pgpool-secret string              The name of a pgpool secret to use for the pgpool
          configuration.
  -z, --policies string                   The policies to apply when creating a cluster, comma
      separated.
      --replica-count int                 The number of replicas to create as part of the cluster.
      --replica-storage-config string    The name of a Storage config in pgo.yaml to use for the
          cluster replica storage.
  -r, --resources-config string           The name of a container resource configuration in
      pgo.yaml that holds CPU and memory requests and limits.
  -s, --secret-from string                The cluster name to use when restoring secrets.
  -e, --series int                        The number of clusters to create in a series. (default 1)
      --service-type string               The Service type to use for the PostgreSQL cluster. If
          not set, the pgo.yaml default will be used.
      --storage-config string             The name of a Storage config in pgo.yaml to use for the
          cluster storage.
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
          Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
          PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
          Operator apiserver.
```

SEE ALSO

- pgo create - Create a Postgres Operator resource

Auto generated by spf13/cobra on 4-Oct-2019

pgo create namespace

Create a namespace

Synopsis

Create a namespace. For example:

```
pgo create namespace somenamespace

Note: For Kubernetes versions prior to 1.12, this command will not function properly
- use $PGOROOT/deploy/add_targted_namespace.sh scriptor or give the user cluster-admin privileges.
For more details, see the Namespace Creation section under Installing Operator Using Bash in the
   documentation.
```

```
pgo create namespace [flags]
```

Options

```
  -h, --help   help for namespace
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo create - Create a Postgres Operator resource

Auto generated by spf13/cobra on 4-Oct-2019

pgo create pgbouncer

Create a pgbouncer

Synopsis

Create a pgbouncer. For example:

```
pgo create pgbouncer mycluster
```

```
pgo create pgbouncer [flags]
```

Options

```
  -h, --help                     help for pgbouncer
      --pgbouncer-pass string    Password for the pgbouncer user of the crunchy-pgboucer deployment.
  -s, --selector string          The selector to use for cluster filtering.
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo create - Create a Postgres Operator resource

Auto generated by spf13/cobra on 4-Oct-2019

pgo create pgorole

Create a pgorole

Synopsis

Create a pgorole. For example:

```
pgo create pgorole somerole --permissions="Cat,Ls"
```

```
pgo create pgorole [flags]
```

Options

```
  -h, --help                   help for pgorole
      --permissions string     specify a comma separated list of permissions for a pgorole
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo create - Create a Postgres Operator resource

Auto generated by spf13/cobra on 4-Oct-2019

pgo create pgouser

Create a pgouser

Synopsis

Create a pgouser. For example:

```
pgo create pgouser someuser
```

```
pgo create pgouser [flags]
```

Options

```
      --all-namespaces               specifies this user will have access to all namespaces.
  -h, --help                         help for pgouser
      --pgouser-namespaces string    specify a comma separated list of Namespaces for a pgouser
      --pgouser-password string      specify a password for a pgouser
      --pgouser-roles string         specify a comma separated list of Roles for a pgouser
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo create - Create a Postgres Operator resource

Auto generated by spf13/cobra on 4-Oct-2019

pgo create pgpool

Create a pgpool

Synopsis

Create a pgpool. For example:

```
pgo create pgpool mycluster
```

```
pgo create pgpool [flags]
```

Options

```
  -h, --help                    help for pgpool
      --pgpool-secret string    The name of a pgpool secret to use for the pgpool configuration.
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo create - Create a Postgres Operator resource

Auto generated by spf13/cobra on 4-Oct-2019

pgo create policy

Create a SQL policy

Synopsis

Create a policy. For example:

```
pgo create policy mypolicy --in-file=/tmp/mypolicy.sql
```

```
pgo create policy [flags]
```

Options

```
  -h, --help              help for policy
  -i, --in-file string    The policy file path to use for adding a policy.
  -u, --url string        The url to use for adding a policy.
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo create - Create a Postgres Operator resource

Auto generated by spf13/cobra on 4-Oct-2019

pgo create schedule

Create a cron-like scheduled task

Synopsis

Schedule creates a cron-like scheduled task. For example:

```
pgo create schedule --schedule="* * * * *" --schedule-type=pgbackrest
   --pgbackrest-backup-type=full mycluster
```

```
pgo create schedule [flags]
```

Options

```
  -c, --ccp-image-tag string             The CCPImageTag to use for cluster creation. If
     specified, overrides the pgo.yaml setting.
      --database string                  The database to run the SQL policy against.
  -h, --help                             help for schedule
      --pgbackrest-backup-type string    The type of pgBackRest backup to schedule (full or diff).
      --pgbackrest-storage-type string   The type of storage to use when scheduling pgBackRest
         backups. Either "local", "s3" or both, comma separated. (default "local")
      --policy string                    The policy to use for SQL schedules.
      --pvc-name string                  The name of the backup PVC to use (only used in
         pgbasebackup schedules).
      --schedule string                  The schedule assigned to the cron task.
      --schedule-opts string             The custom options passed to the create schedule API.
      --schedule-type string             The type of schedule to be created (pgbackrest,
         pgbasebackup or policy).
      --secret string                    The secret name for the username and password of the
         PostgreSQL role for SQL schedules.
  -s, --selector string                  The selector to use for cluster filtering.
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo create - Create a Postgres Operator resource

Auto generated by spf13/cobra on 4-Oct-2019

pgo create user

Create a PostgreSQL user

Synopsis

Create a postgres user. For example:

```
pgo create user --username=someuser --all --managed
pgo create user --username=someuser  mycluster --managed
pgo create user --username=someuser -selector=name=mycluster --managed
pgo create user --username=user1 --selector=name=mycluster
```

```
pgo create user [flags]
```

Options

```
  -h, --help                  help for user
      --managed               Creates a user with secrets that can be managed by the Operator.
      --password string       The password to use for creating a new user which overrides a
         generated password.
      --password-length int   If no password is supplied, this is the length of the auto generated
         password (default 22)
  -s, --selector string       The selector to use for cluster filtering.
      --username string       The username to use for creating a new user
      --valid-days int        Sets passwords for new users to X days. (default 30)
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo create - Create a Postgres Operator resource

Auto generated by spf13/cobra on 4-Oct-2019

pgo delete

Delete an Operator resource

Synopsis

The delete command allows you to delete an Operator resource. For example:

```
pgo delete backup mycluster
pgo delete benchmark mycluster
pgo delete cluster mycluster
pgo delete cluster mycluster --delete-data
pgo delete cluster mycluster --delete-data --delete-backups
pgo delete label mycluster --label=env=research
pgo delete pgbouncer mycluster
pgo delete pgpool mycluster
pgo delete pgouser someuser
pgo delete pgorole somerole
pgo delete policy mypolicy
pgo delete namespace mynamespace
pgo delete schedule --schedule-name=mycluster-pgbackrest-full
pgo delete schedule --selector=name=mycluster
pgo delete schedule mycluster
pgo delete user --username=testuser --selector=name=mycluster
```

```
pgo delete [flags]
```

Options

```
  -h, --help   help for delete
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo - The pgo command line interface.
- pgo delete backup - Delete a backup
- pgo delete benchmark - Delete benchmarks for a cluster
- pgo delete cluster - Delete a PostgreSQL cluster
- pgo delete label - Delete a label from clusters
- pgo delete namespace - Delete namespaces
- pgo delete pgbouncer - Delete a pgbouncer from a cluster
- pgo delete pgorole - Delete a pgorole
- pgo delete pgouser - Delete a pgouser
- pgo delete pgpool - Delete a pgpool from a cluster
- pgo delete policy - Delete a SQL policy

- pgo delete schedule - Delete a schedule
- pgo delete user - Delete a user

Auto generated by spf13/cobra on 4-Oct-2019

pgo delete backup

Delete a backup

Synopsis

Delete a backup. For example:

```
pgo delete backup mydatabase
```

```
pgo delete backup [flags]
```

Options

```
  -h, --help   help for backup
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo delete - Delete an Operator resource

Auto generated by spf13/cobra on 4-Oct-2019

pgo delete benchmark

Delete benchmarks for a cluster

Synopsis

Delete benchmarks for a cluster. For example:

```
pgo delete benchmark mycluster
pgo delete benchmark --selector=env=test
```

```
pgo delete benchmark [flags]
```

Options

```
  -h, --help              help for benchmark
  -s, --selector string   The selector to use for cluster filtering.
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo delete - Delete an Operator resource

Auto generated by spf13/cobra on 4-Oct-2019

pgo delete cluster

Delete a PostgreSQL cluster

Synopsis

Delete a PostgreSQL cluster. For example:

```
pgo delete cluster --all
pgo delete cluster mycluster
```

```
pgo delete cluster [flags]
```

Options

```
      --all               all clusters. Backups and data subject to --delete-backups and
         --delete-data flags, respectively.
  -b, --delete-backups    Causes the backups for specified cluster to be removed permanently.
  -d, --delete-data       Causes the data for specified cluster to be removed permanently.
      --keep-backups      Keeps the backups available for use at a later time (e.g. recreating the
         cluster)
      --keep-data         Keeps the data for the specified cluster. Can be reassigned to exact
         same cluster in the future
  -h, --help              help for cluster
      --no-prompt         No command line confirmation.
  -s, --selector string   The selector to use for cluster filtering.
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo delete - Delete an Operator resource

Auto generated by spf13/cobra on 4-Oct-2019

pgo delete label

Delete a label from clusters

Synopsis

Delete a label from clusters. For example:

```
pgo delete label mycluster --label=env=research
pgo delete label all --label=env=research
pgo delete label --selector=group=southwest --label=env=research
```

```
pgo delete label [flags]
```

Options

```
  -h, --help              help for label
      --label string      The label to delete for any selected or specified clusters.
  -s, --selector string   The selector to use for cluster filtering.
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo delete - Delete an Operator resource

Auto generated by spf13/cobra on 4-Oct-2019

pgo delete namespace

Delete namespaces

Synopsis

Delete namespaces. For example:

```
pgo delete namespace mynamespace
pgo delete namespace --selector=env=test
```

```
pgo delete namespace [flags]
```

Options

```
  -h, --help   help for namespace
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo delete - Delete an Operator resource

Auto generated by spf13/cobra on 4-Oct-2019

pgo delete pgbouncer

Delete a pgbouncer from a cluster

Synopsis

Delete a pgbouncer from a cluster. For example:

```
pgo delete pgbouncer mycluster
```

```
pgo delete pgbouncer [flags]
```

Options

```
  -h, --help              help for pgbouncer
      --no-prompt         No command line confirmation.
  -s, --selector string   The selector to use for cluster filtering.
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo delete - Delete an Operator resource

Auto generated by spf13/cobra on 4-Oct-2019

pgo delete pgorole

Delete a pgorole

Synopsis

Delete a pgorole. For example:

```
pgo delete pgorole somerole
```

```
pgo delete pgorole [flags]
```

Options

```
      --all          all resources.
  -h, --help         help for pgorole
      --no-prompt    No command line confirmation.
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo delete - Delete an Operator resource

Auto generated by spf13/cobra on 4-Oct-2019

pgo delete pgouser

Delete a pgouser

Synopsis

Delete a pgouser. For example:

```
pgo delete pgouser someuser
```

```
pgo delete pgouser [flags]
```

Options

```
      --all          all resources.
  -h, --help         help for pgouser
      --no-prompt    No command line confirmation.
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo delete - Delete an Operator resource

Auto generated by spf13/cobra on 4-Oct-2019

pgo delete pgpool

Delete a pgpool from a cluster

Synopsis

Delete a pgpool from a cluster. For example:

```
pgo delete pgpool mycluster
```

```
pgo delete pgpool [flags]
```

Options

```
  -h, --help              help for pgpool
      --no-prompt         No command line confirmation.
  -s, --selector string   The selector to use for cluster filtering.
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo delete - Delete an Operator resource

Auto generated by spf13/cobra on 4-Oct-2019

pgo delete policy

Delete a SQL policy

Synopsis

Delete a policy. For example:

```
pgo delete policy mypolicy
```

```
pgo delete policy [flags]
```

Options

```
      --all         all resources.
  -h, --help        help for policy
      --no-prompt   No command line confirmation.
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo delete - Delete an Operator resource

Auto generated by spf13/cobra on 4-Oct-2019

pgo delete schedule

Delete a schedule

Synopsis

Delete a cron-like schedule. For example:

```
pgo delete schedule mycluster
pgo delete schedule --selector=env=test
pgo delete schedule --schedule-name=mycluster-pgbackrest-full
```

```
pgo delete schedule [flags]
```

Options

```
  -h, --help                   help for schedule
      --no-prompt              No command line confirmation.
      --schedule-name string   The name of the schedule to delete.
  -s, --selector string        The selector to use for cluster filtering.
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo delete - Delete an Operator resource

Auto generated by spf13/cobra on 4-Oct-2019

pgo delete upgrade

Delete an upgrade

Synopsis

Delete an upgrade. For example:

```
pgo delete upgrade mydatabase
```

```
pgo delete upgrade [flags]
```

Options

```
  -h, --help   help for upgrade
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo delete - Delete a backup, benchmark, cluster, pgbouncer, pgpool, label, policy, upgrade, or user

Auto generated by spf13/cobra on 27-Mar-2019

pgo delete user

Delete a user

Synopsis

Delete a user. For example:

```
pgo delete user --username=someuser --selector=name=mycluster
```

```
pgo delete user [flags]
```

Options

```
      --all               all clusters.
  -h, --help              help for user
      --no-prompt         No command line confirmation.
  -s, --selector string   The selector to use for cluster filtering.
      --username string   The username to delete.
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo delete - Delete an Operator resource

Auto generated by spf13/cobra on 4-Oct-2019

pgo df

Display disk space for clusters

Synopsis

Displays the disk status for PostgreSQL clusters. For example:

```
pgo df mycluster
pgo df all
pgo df --selector=env=research
```

```
pgo df [flags]
```

Options

```
  -h, --help              help for df
  -s, --selector string   The selector to use for cluster filtering.
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo - The pgo command line interface.

Auto generated by spf13/cobra on 4-Oct-2019

pgo failover

Performs a manual failover

Synopsis

Performs a manual failover. For example:

```
pgo failover mycluster
```

```
pgo failover [flags]
```

Options

```
      --autofail-replace-replica string   If 'true', causes a replica to be created to replace the
          promoted replica.  If 'false', causes a replica to not be created, if not set, the
          pgo.yaml AutofailReplaceReplica setting is used.
  -h, --help                              help for failover
      --no-prompt                         No command line confirmation.
      --query                             Prints the list of failover candidates.
      --target string                     The replica target which the failover will occur on.
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
          Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
          PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
          Operator apiserver.
```

SEE ALSO

- pgo - The pgo command line interface.

Auto generated by spf13/cobra on 4-Oct-2019

pgo label

Label a set of clusters

Synopsis

LABEL allows you to add or remove a label on a set of clusters. For example:

```
pgo label mycluster yourcluster --label=environment=prod
pgo label all --label=environment=prod
pgo label --label=environment=prod --selector=name=mycluster
pgo label --label=environment=prod --selector=status=final --dry-run
```

```
pgo label [flags]
```

Options

```
      --dry-run           Shows the clusters that the label would be applied to, without labelling
          them.
  -h, --help              help for label
      --label string      The new label to apply for any selected or specified clusters.
  -s, --selector string   The selector to use for cluster filtering.
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
          Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
          PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
          Operator apiserver.
```

SEE ALSO

- pgo - The pgo command line interface.

Auto generated by spf13/cobra on 4-Oct-2019

pgo load

Perform a data load

Synopsis

LOAD performs a load. For example:

```
pgo load --load-config=./load.json --selector=project=xray
```

```
pgo load [flags]
```

Options

```
  -h, --help                   help for load
      --load-config string   The load configuration to use that defines the load job.
      --policies string      The policies to apply before loading a file, comma separated.
  -s, --selector string      The selector to use for cluster filtering.
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo - The pgo command line interface.

Auto generated by spf13/cobra on 4-Oct-2019

pgo ls

Perform a ls command on a cluster

Synopsis

LS performs a Linux ls command on a cluster directory. For example:

```
pgo ls mycluster /pgdata/mycluster/pg_log
```

```
pgo ls [flags]
```

Options

```
  -h, --help   help for ls
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo - The pgo command line interface.

Auto generated by spf13/cobra on 4-Oct-2019

pgo reload

Perform a cluster reload

Synopsis

RELOAD performs a PostgreSQL reload on a cluster or set of clusters. For example:

```
pgo reload mycluster
```

```
pgo reload [flags]
```

Options

```
  -h, --help              help for reload
      --no-prompt         No command line confirmation.
  -s, --selector string   The selector to use for cluster filtering.
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo - The pgo command line interface.

Auto generated by spf13/cobra on 4-Oct-2019

pgo restore

Perform a restore from previous backup

Synopsis

RESTORE performs a restore to a new PostgreSQL cluster. This includes stopping the database and recreating a new primary with the restored data. Valid backup types to restore from are pgbackrest, pgbasebackup and pgdump. For example:

```
pgo restore mycluster
```

```
pgo restore [flags]
```

Options

```
      --backup-opts string               The restore options for pgbackrest or pgdump.
      --backup-path string               The path for the directory containing the pg_basebackup
         backup to be utilized for the restore. If omitted, defaults to the latest backup.
      --backup-pvc string                The PVC containing the pgdump or pgbasebackup backup
         directory to restore from.
      --backup-type string               The type of backup to restore from, default is
         pgbackrest. Valid types are pgbackrest, pgdump or pgbasebackup.
  -h, --help                             help for restore
      --no-prompt                        No command line confirmation.
      --node-label string                The node label (key=value) to use when scheduling the
         restore job, and in the case of a pgBackRest restore, also the new (i.e. restored)
         primary deployment. If not set, any node is used.
      --pgbackrest-storage-type string   The type of storage to use for a pgBackRest restore.
         Either "local", "s3". (default "local")
      --pitr-target string               The PITR target, being a PostgreSQL timestamp such as
         '2018-08-13 11:25:42.582117-04'.
      --restore-to-pvc string            The name of the PVC to restore into when restoring from a
         pgbasebackup backup.
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo - The pgo command line interface.

Auto generated by spf13/cobra on 4-Oct-2019

pgo scale

Scale a PostgreSQL cluster

Synopsis

The scale command allows you to adjust a Cluster's replica configuration. For example:

```
pgo scale mycluster --replica-count=1
```

```
pgo scale [flags]
```

Options

```
      --ccp-image-tag string      The CCPImageTag to use for cluster creation. If specified,
         overrides the .pgo.yaml setting.
  -h, --help                      help for scale
      --no-prompt                 No command line confirmation.
      --node-label string         The node label (key) to use in placing the replica database. If
         not set, any node is used.
      --replica-count int         The replica count to apply to the clusters. (default 1)
      --resources-config string   The name of a container resource configuration in pgo.yaml that
         holds CPU and memory requests and limits.
      --service-type string       The service type to use in the replica Service. If not set, the
         default in pgo.yaml will be used.
      --storage-config string     The name of a Storage config in pgo.yaml to use for the replica
         storage.
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo - The pgo command line interface.

Auto generated by spf13/cobra on 4-Oct-2019

pgo scaledown

Scale down a PostgreSQL cluster

Synopsis

The scale command allows you to scale down a Cluster's replica configuration. For example:

```
To list targetable replicas:
pgo scaledown mycluster --query

To scale down a specific replica:
pgo scaledown mycluster --target=mycluster-replica-xxxx
```

```
pgo scaledown [flags]
```

Options

```
  -d, --delete-data     Causes the data for the scaled down replica to be removed permanently.
      --keep-data       Keeps the data for the specified instance. Can be used to troubleshoot
         issues with this specific instance.
  -h, --help            help for scaledown
      --no-prompt       No command line confirmation.
      --query           Prints the list of targetable replica candidates.
      --target string   The replica to target for scaling down
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo - The pgo command line interface.

Auto generated by spf13/cobra on 4-Oct-2019

pgo show

Show the description of a cluster

Synopsis

Show allows you to show the details of a policy, backup, pvc, or cluster. For example:

```
pgo show backup mycluster
pgo show backup mycluster --backup-type=pgbackrest
pgo show benchmark mycluster
pgo show cluster mycluster
pgo show config
pgo show pgouser someuser
pgo show policy policy1
pgo show pvc mycluster
pgo show namespace
pgo show workflow 25927091-b343-4017-be4b-71575f0b3eb5
pgo show user --selector=name=mycluster
```

```
pgo show [flags]
```

Options

```
  -h, --help   help for show
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo - The pgo command line interface.
- pgo show backup - Show backup information
- pgo show benchmark - Show benchmark information
- pgo show cluster - Show cluster information
- pgo show config - Show configuration information
- pgo show namespace - Show namespace information

- pgo show pgorole - Show pgorole information
- pgo show pgouser - Show pgouser information
- pgo show policy - Show policy information
- pgo show pvc - Show PVC information
- pgo show schedule - Show schedule information
- pgo show user - Show user information
- pgo show workflow - Show workflow information

Auto generated by spf13/cobra on 4-Oct-2019

pgo show backup

Show backup information

Synopsis

Show backup information. For example:

```
pgo show backup mycluser
```

```
pgo show backup [flags]
```

Options

```
      --backup-type string   The backup type output to list. Valid choices are pgbasebackup or
         pgbackrest. (default "pgbackrest")
  -h, --help                 help for backup
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo show - Show the description of a cluster

Auto generated by spf13/cobra on 4-Oct-2019

pgo show benchmark

Show benchmark information

Synopsis

Show benchmark results for clusters. For example:

```
pgo show benchmark mycluster
pgo show benchmark --selector=pg-cluster=mycluster
```

```
pgo show benchmark [flags]
```

Options

```
  -h, --help              help for benchmark
  -s, --selector string   The selector to use for cluster filtering.
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo show - Show the description of a cluster

Auto generated by spf13/cobra on 4-Oct-2019

pgo show cluster

Show cluster information

Synopsis

Show a PostgreSQL cluster. For example:

```
pgo show cluster --all
pgo show cluster mycluster
```

```
pgo show cluster [flags]
```

Options

```
      --all                    show all resources.
      --ccp-image-tag string   Filter the results based on the image tag of the cluster.
  -h, --help                   help for cluster
  -o, --output string          The output format. Currently, json is the only supported value.
  -s, --selector string        The selector to use for cluster filtering.
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo show - Show the description of a cluster

Auto generated by spf13/cobra on 4-Oct-2019

pgo show config

Show configuration information

Synopsis

Show configuration information for the Operator. For example:

```
pgo show config
```

```
pgo show config [flags]
```

Options

```
  -h, --help   help for config
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo show - Show the description of a cluster

Auto generated by spf13/cobra on 4-Oct-2019

pgo show namespace

Show namespace information

Synopsis

Show namespace information for the Operator. For example:

```
pgo show namespace
```

```
pgo show namespace [flags]
```

Options

```
      --all    show all resources.
  -h, --help   help for namespace
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo show - Show the description of a cluster

Auto generated by spf13/cobra on 4-Oct-2019

pgo show pgorole

Show pgorole information

Synopsis

Show pgorole information . For example:

```
pgo show pgorole somerole
```

```
pgo show pgorole [flags]
```

Options

```
      --all    show all resources.
  -h, --help   help for pgorole
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo show - Show the description of a cluster

Auto generated by spf13/cobra on 4-Oct-2019

pgo show pgouser

Show pgouser information

Synopsis

Show pgouser information for an Operator user. For example:

```
pgo show pgouser someuser
```

```
pgo show pgouser [flags]
```

Options

```
      --all    show all resources.
  -h, --help   help for pgouser
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo show - Show the description of a cluster

Auto generated by spf13/cobra on 4-Oct-2019

pgo show policy

Show policy information

Synopsis

Show policy information. For example:

```
pgo show policy --all
pgo show policy policy1
```

```
pgo show policy [flags]
```

Options

```
      --all    show all resources.
  -h, --help   help for policy
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo show - Show the description of a cluster

Auto generated by spf13/cobra on 4-Oct-2019

pgo show pvc

Show PVC information

Synopsis

Show PVC information. For example:

```
pgo show pvc mycluster
pgo show pvc --all
pgo show pvc mycluster-backup
pgo show pvc mycluster-xlog
pgo show pvc a2-backup --pvc-root=a2-backups/2019-01-12-17-09-42
```

```
pgo show pvc [flags]
```

Options

```
      --all                  show all resources.
  -h, --help                 help for pvc
      --node-label string   The node label (key=value) to use
      --pvc-root string     The PVC directory to list.
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo show - Show the description of a cluster

Auto generated by spf13/cobra on 4-Oct-2019

pgo show schedule

Show schedule information

Synopsis

Show cron-like schedules. For example:

```
pgo show schedule mycluster
pgo show schedule --selector=pg-cluster=mycluster
pgo show schedule --schedule-name=mycluster-pgbackrest-full
```

```
pgo show schedule [flags]
```

Options

```
  -h, --help                    help for schedule
      --no-prompt               No command line confirmation.
      --schedule-name string   The name of the schedule to show.
  -s, --selector string        The selector to use for cluster filtering.
```

Options inherited from parent commands

```
      --apiserver-url string     The URL for the PostgreSQL Operator apiserver.
      --debug                    Enable debugging when true.
  -n, --namespace string         The namespace to use for pgo requests.
      --pgo-ca-cert string       The CA Certificate file path for authenticating to the PostgreSQL
         Operator apiserver.
      --pgo-client-cert string   The Client Certificate file path for authenticating to the
         PostgreSQL Operator apiserver.
      --pgo-client-key string    The Client Key file path for authenticating to the PostgreSQL
         Operator apiserver.
```

SEE ALSO

- pgo show - Show the description of a cluster

Auto generated by spf13/cobra on 4-Oct-2019