Crunchy PostgreSQL Operator

Contents

Crunchy PostgreSQL Operator

Run your own production-grade PostgreSQL-as-a-Service on Kubernetes!

Latest Release: 4.3.0

The Crunchy PostgreSQL Operator automates and simplifies deploying and managing open source PostgreSQL clusters on Kubernetes and other Kubernetes-enabled Platforms by providing the essential features you need to keep your PostgreSQL clusters up and running, including:

PostgreSQL Cluster Provisioning Create, Scale, & Delete PostgreSQL clusters with ease, while fully customizing your Pods and PostgreSQL configuration!

High-Availability Safe, automated failover backed by a distributed consensus based high-availability solution. Uses Pod Anti-Affinity to help resiliency; you can configure how aggressive this can be! Failed primaries automatically heal, allowing for faster recovery time.

Disaster Recovery Backups and restores leverage the open source pgBackRest utility and includes support for full, incremental, and differential backups as well as efficient delta restores. Set how long you want your backups retained for. Works great with very large databases!

Monitoring Track the health of your PostgreSQL clusters using the open source pgMonitor library.

PostgreSQL User Management Quickly add and remove users from your PostgreSQL clusters with powerful commands. Manage password expiration policies or use your preferred PostgreSQL authentication scheme.

Upgrade Management Safely apply PostgreSQL updates with minimal availability impact to your PostgreSQL clusters.

Advanced Replication Support Choose between asynchronous replication and synchronous replication for workloads that are sensitive to losing transactions.

Clone Create new clusters from your existing clusters with a simple `pgo clone` command.

Connection Pooling Use pgBouncer for connection pooling

Node Affinity Have your PostgreSQL clusters deployed to Kubernetes Nodes of your preference

Scheduled Backups Choose the type of backup (full, incremental, differential) and how frequently you want it to occur on each PostgreSQL cluster.

Backup to S3 Store your backups in Amazon S3 or any object storage system that supports the S3 protocol. The PostgreSQL Operator can backup, restore, and create new clusters from these backups.

Multi-Namespace Support You can control how the PostgreSQL Operator leverages Kubernetes Namespaces with several different deployment models:

- Deploy the PostgreSQL Operator and all PostgreSQL clusters to the same namespace
- Deploy the PostgreSQL Operator to one namespaces, and all PostgreSQL clusters to a different namespace
- Deploy the PostgreSQL Operator to one namespace, and have your PostgreSQL clusters managed acrossed multiple namespaces
- Dynamically add and remove namespaces managed by the PostgreSQL Operator using the `pgo create namespace` and `pgo delete namespace` commands

Full Customizability The Crunchy PostgreSQL Operator makes it easy to get your own PostgreSQL-as-a-Service up and running on Kubernetes-enabled platforms, but we know that there are further customizations that you can make. As such, the Crunchy PostgreSQL Operator allows you to further customize your deployments, including:

- Selecting different storage classes for your primary, replica, and backup storage
- Select your own container resources class for each PostgreSQL cluster deployment; differentiate between resources applied for primary and replica clusters!
- Use your own container image repository, including support `imagePullSecrets` and private repositories
- Bring your own trusted certificate authority (CA) for use with the Operator API server
- Override your PostgreSQL configuration for each cluster

How it Works

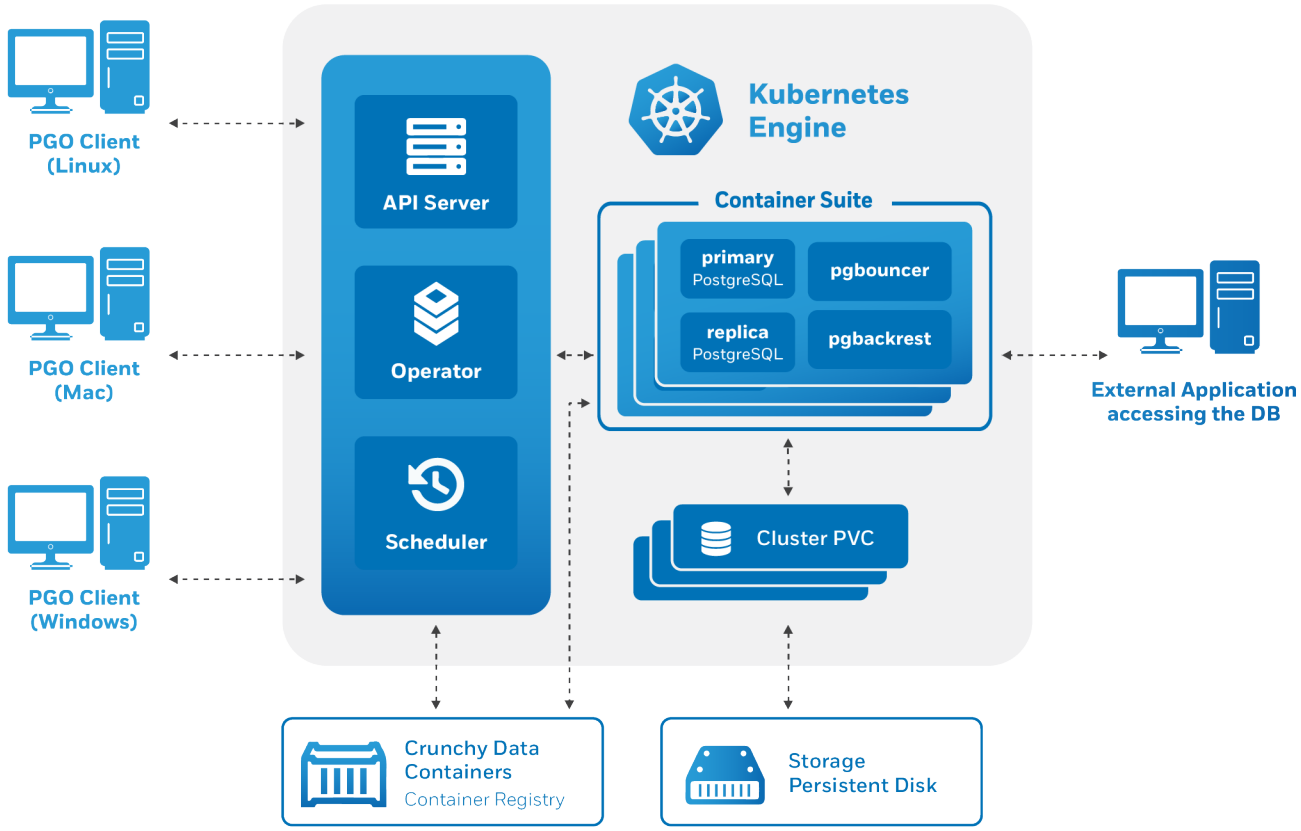

Figure 1: Architecture

The Crunchy PostgreSQL Operator extends Kubernetes to provide a higher-level abstraction for rapid creation and management of PostgreSQL clusters. The Crunchy PostgreSQL Operator leverages a Kubernetes concept referred to as "Custom Resources" to create several custom resource definitions (CRDs) that allow for the management of PostgreSQL clusters.

Supported Platforms

The Crunchy PostgreSQL Operator is tested on the following Platforms:

- Kubernetes 1.13 - 1.15 (See note about 1.16 and beyond)
- OpenShift 3.11+
- Google Kubernetes Engine (GKE), including Anthos
- VMware Enterprise PKS 1.3+

NOTE: At present, while the Crunchy PostgreSQL Operator has compatibility for Kubernetes 1.16 and beyond, it has not been verified for the v4.3.0 release.

Storage

The Crunchy PostgreSQL Operator is tested with a variety of different types of Kubernetes storage and Storage Classes, including:

- Google Compute Engine persistent volumes
- HostPath
- NFS
- Rook
- StorageOS

and more.

We know there are a variety of different types of Storage Classes available for Kubernetes and we do our best to test each one, but due to the breadth of this area we are unable to verify PostgreSQL Operator functionality in each one. With that said, the PostgreSQL Operator is designed to be storage class agnostic and has been demonstrated to work with additional Storage Classes.

PostgreSQL Operator Quickstart

Can't wait to try out the PostgreSQL Operator? Let us show you the quickest possible path to getting up and running.

There are two paths to quickly get you up and running with the PostgreSQL Operator:

- Installation via Ansible
- Installation via a Marketplace
- Installation via Google Cloud Platform Marketplace

Marketplaces can help you get more quickly started in your environment as they provide a mostly automated process, but there are a few steps you will need to take to ensure you can fully utilize your PostgreSQL Operator environment.

Ansible

Below will guide you through the steps for installing and using the PostgreSQL Operator using an installer that works with Ansible.

Step 1: Prerequisites

Kubernetes / OpenShift

- A Kubernetes or OpenShift environment where you have enough privileges to install an application, i.e. you can add a ClusterRole. If you're a Cluster Admin, you're all set.
- Your Kubernetes version should be 1.13+. **NOTE**: For v4.3.0, while we have updated the PostgreSQL Operator for compatibility with 1.16+, we have not fully tested it.
- For OpenShift, the PostgreSQL Operator will work in 3.11+
- PersistentVolumes that are available

Your Environment

- `kubectl` or `oc`. Ensure you can access your Kubernetes or OpenShift cluster (this is outside the scope of this document)
- `ansible` 2.7.0+. Learn how to download ansible
- `git`
- If you are installing to Google Kubernetes Engine, you will need the `gcloud` utility

Step 2: Configuration

Get the PostgreSQL Operator Ansible Installation Playbook

You can download the playbook by cloning the PostgreSQL Operator git repository and running the following commands:

```
git clone https://github.com/CrunchyData/postgres-operator.git
cd postgres-operator
git checkout v4.3.0 # you can substitute this for the version that you want to install
cd ansible
```

Configure your Installation

Within the `ansible` folder, there exists a file called `inventory`. When you open up this file, you can see several options that are used to install the PostgreSQL Operator. Most of these contain some sensible defaults for getting up and running quickly, but some you will need to fill out yourself.

Lines that start with a `#` are commented out. To activate that configuration setting, you will have to delete the `#`.

Set up your `inventory` file based on one of the environments that you are deploying to:

Kubernetes You will have to uncomment and set the `kubernetes_context` variable. This can be determined based on the output of the `kubectl config current-context` e.g.:

```
kubectl config current-context
kubernetes-admin@kubernetes
```

Note that the output will vary based on the Kubernetes cluster you are using.

Using the above example, set the value of `kubernetes_context` to the output of the `kubectl config current-context` command, e.g.

```
kubernetes_context="kubernetes-admin@kubernetes"
```

Find the location of the `pgo_admin_password` configuration variable. Set this to a password of your choosing, e.g.

```
pgo_admin_password="hippo-elephant"
```

Finally, you will need to set the storage default storage classes that you would like the Operator to use. For example, if your Kubernetes environment is using NFS storage, you would set this variables to the following:

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

For a full list of available storage types that can be used with this installation method, see: $URL

OpenShift For an OpenShfit deployment, you will at a minimum have to to uncomment and set the `openshift_host` variable. This is the location of where your OpenShift environment is, and can be obtained from your administrator. For example:

```
openshift_host="https://openshift.example.com:6443"
```

Based on how your OpenShift environment is configured, you may need to set the following variables:

- `openshift_user`
- `openshift_password`
- `openshift_token`

An optional `openshift_skip_tls_verify=true` variable is available if your OpenShift environment allows you to skip TLS verification.

Next, find the location of the `pgo_admin_password` configuration variable. Set this to a password of your choosing, e.g.

```
pgo_admin_password="hippo-elephant"
```

Finally, you will need to set the storage default storage classes that you would like the Operator to use. For example, if your OpenShift environment is using Rook storage, you would set this variables to the following:

```
backrest_storage='rook'
backup_storage='rook'
primary_storage='rook'
replica_storage='rook'
```

For a full list of available storage types that can be used with this installation method, see: $URL

Google Kubernetes Engine (GKE) For deploying the PostgreSQL Operator to GKE, you will need to set up your cluster similar to the Kubernetes set up. First, you will need to get the value for the `kubernetes_context` variable. Using the `gcloud` utility, ensure you are logged into the GCP Project that you are installing the PostgreSQL Operator into:

```
gcloud config set project [PROJECT_ID]
```

You can read about how you can get the value of `[PROJECT_ID]`

From here, you can get the value that needs to be set into the `kubernetes_context`.

You will have to uncomment and set the `kubernetes_context` variable. This can be determined based on the output of the `kubectl config current-context` e.g.:

```
kubectl config current-context
gke_some-name_some-zone-some_project
```

Note that the output will vary based on your GKE project.

Using the above example, set the value of `kubernetes_context` to the output of the `kubectl config current-context` command, e.g.

```
kubernetes_context="gke_some-name_some-zone-some_project"
```

Next, find the location of the `pgo_admin_password` configuration variable. Set this to a password of your choosing, e.g.

```
pgo_admin_password="hippo-elephant"
```

Finally, you will need to set the storage default storage classes that you would like the Operator to use. For deploying to GKE it is recommended to use the `gce` storag class:

```
backrest_storage='gce'
backup_storage='gce'
primary_storage='gce'
replica_storage='gce'
```

Step 3: Installation

Ensure you are still in the `ansible` directory and run the following command to install the PostgreSQL Operator:

```
ansible-playbook -i inventory --tags=install main.yml
```

This can take a few minutes to complete depending on your Kubernetes cluster.

While the PostgreSQL Operator is installing, for ease of using the `pgo` command line interface, you will need to set up some environmental variables. You can do so with the following command:

```
export PGOUSER="${HOME?}/.pgo/pgo/pgouser"
export PGO_CA_CERT="${HOME?}/.pgo/pgo/client.crt"
export PGO_CLIENT_CERT="${HOME?}/.pgo/pgo/client.crt"
export PGO_CLIENT_KEY="${HOME?}/.pgo/pgo/client.pem"
export PGO_APISERVER_URL='https://127.0.0.1:8443'
export PGO_NAMESPACE=pgouser1
```

If you wish to permanently add these variables to your environment, you can run the following:

```
cat <<EOF >> ~/.bashrc
export PGOUSER="${HOME?}/.pgo/pgo/pgouser"
export PGO_CA_CERT="${HOME?}/.pgo/pgo/client.crt"
export PGO_CLIENT_CERT="${HOME?}/.pgo/pgo/client.crt"
export PGO_CLIENT_KEY="${HOME?}/.pgo/pgo/client.pem"
export PGO_APISERVER_URL='https://127.0.0.1:8443'
export PGO_NAMESPACE=pgouser1
EOF

source ~/.bashrc
```

NOTE: For macOS users, you must use `~/.bash_profile` instead of `~/.bashrc`

Step 4: Verification

Below are a few steps to check if the PostgreSQL Operator is up and running.

By default, the PostgreSQL Operator installs into a namespace called `pgo`. First, see that the the Kubernetes Deployment of the Operator exists and is healthy:

```
kubectl -n pgo get deployments
```

If successful, you should see output similar to this:

```
NAME                READY   UP-TO-DATE   AVAILABLE   AGE
postgres-operator   1/1     1            1           16h
```

Next, see if the Pods that run the PostgreSQL Operator are up and running:

```
kubectl -n pgo get pods
```

If successful, you should see output similar to this:

```
NAME                                READY   STATUS    RESTARTS   AGE
postgres-operator-56d6ccb97-tmz7m   4/4     Running   0          2m
```

Finally, let's see if we can connect to the PostgreSQL Operator from the `pgo` command-line client. The Ansible installer installs the `pgo` command line client into your environment, along with the username/password file that allows you to access the PostgreSQL Operator. In order to communicate with the PostgreSQL Operator API server, you will first need to set up a port forward to your local environment.

In a new console window, run the following command to set up a port forward:

```
kubectl -n pgo port-forward svc/postgres-operator 8443:8443
```

Back to your original console window, you can verify that you can connect to the PostgreSQL Operator using the following command:

```
pgo version
```

If successful, you should see output similar to this:

```
pgo client version 4.3.0
pgo-apiserver version 4.3.0
```

Step 5: Have Some Fun - Create a PostgreSQL Cluster

The quickstart installation method creates two namespaces that you can deploy your PostgreSQL clusters into called `pgouser1` and `pgouser2`. Let's create a new PostgreSQL cluster in `pgouser1`:

```
pgo create cluster -n pgouser1 hippo
```

Alternatively, because we set the `PGO_NAMESPACE` environmental variable in our `.bashrc` file, we could omit the `-n` flag from the `pgo create cluster` command and just run this:

```
pgo create cluster hippo
```

Even with `PGO_NAMESPACE` set, you can always overwrite which namespace to use by setting the `-n` flag for the specific command. For explicitness, we will continue to use the `-n` flag in the remaining examples of this quickstart.

If your cluster creation command executed successfully, you should see output similar to this:

```
created Pgcluster hippo
workflow id 1cd0d225-7cd4-4044-b269-aa7bedae219b
```

This will create a PostgreSQL cluster named `hippo`. It may take a few moments for the cluster to be provisioned. You can see the status of this cluster using the `pgo test` command:

```
pgo test -n pgouser1 hippo
```

When everything is up and running, you should see output similar to this:

```
cluster : hippo
    Services
        primary (10.97.140.113:5432): UP
    Instances
        primary (hippo-7b64747476-6dr4h): UP
```

The `pgo test` command provides you the basic information you need to connect to your PostgreSQL cluster from within your Kubernetes environment. For more detailed information, you can use `pgo show cluster -n pgouser1 hippo`.

Marketplaces

Below is the list of the marketplaces where you can find the Crunchy PostgreSQL Operator:

- Google Cloud Platform Marketplace: Crunchy PostgreSQL for GKE

Follow the instructions below for the marketplace that you want to use to deploy the Crunchy PostgreSQL Operator.

Google Cloud Platform Marketplace

The PostgreSQL Operator is installed as part of the Crunchy PostgreSQL for GKE project that is available in the Google Cloud Platform Marketplace (GCP Marketplace). Please follow the steps deploy to get the PostgreSQL Operator deployed!

Step 1: Prerequisites

Install `Kubectl` and `gcloud` SDK

- kubectl is required to execute kube commands with in GKE.
- gcloudsdk essential command line tools for google cloud

Verification Below are a few steps to check if the PostgreSQL Operator is up and running.

For this example we are deploying the operator into a namespace called `pgo`. First, see that the the Kubernetes Deployment of the Operator exists and is healthy:

```
kubectl -n pgo get deployments
```

If successful, you should see output similar to this:

```
NAME                READY   UP-TO-DATE   AVAILABLE   AGE
postgres-operator   1/1     1            1           16h
```

Next, see if the Pods that run the PostgreSQL Operator are up and running:

```
kubectl -n pgo get pods
```

If successful, you should see output similar to this:

```
NAME                                READY   STATUS    RESTARTS   AGE
postgres-operator-56d6ccb97-tmz7m   4/4     Running   0          2m
```

Step 2: Install the PostgreSQL Operator User Keys

After your operator is deployed via GCP Marketplace you will need to get keys used to secure the Operator REST API. For these instructions we will assume the operator is deployed in a namespace named "pgo" if this in not the case for your operator change the namespace to coencide with where your operator is deployed. Using the `gcloud` utility, ensure you are logged into the GKE cluster that you installed the PostgreSQL Operator into, run the following commands to retrieve the cert and key:

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

Step 3: Setup PostgreSQL Operator User

The PostgreSQL Operator implements its own role-based access control (RBAC) system for authenticating and authorization PostgreSQL Operator users access to its REST API. A default PostgreSQL Operator user (aka a "pgouser") is created as part of the marketplace installation (these credentials are set during the marketplace deployment workflow).

Create the pgouser file in `${HOME?}/.pgo/<operatornamespace>/pgouser` and insert the user and password you created on deployment of the PostgreSQL Operator via GCP Marketplace. For example, if you set up a user with the username of `username` and a password of `hippo`:

```
username:hippo
```

Step 4: Setup Environment variables

The PostgreSQL Operator Client uses several environmental variables to make it easier for interfacing with the PostgreSQL Operator.

Set the environmental variables to use the key / certificate pair that you pulled in Step 2 was deployed via the marketplace. Using the previous examples, You can set up environment variables with the following command:

```
export PGOUSER="${HOME?}/.pgo/pgo/pgouser"
export PGO_CA_CERT="/tmp/client.crt"
export PGO_CLIENT_CERT="/tmp/client.crt"
export PGO_CLIENT_KEY="/tmp/client.key"
export PGO_APISERVER_URL='https://127.0.0.1:8443'
export PGO_NAMESPACE=pgouser1
```

If you wish to permanently add these variables to your environment, you can run the following command:

```
cat <<EOF >> ~/.bashrc
export PGOUSER="${HOME?}/.pgo/pgo/pgouser"
export PGO_CA_CERT="/tmp/client.crt"
export PGO_CLIENT_CERT="/tmp/client.crt"
export PGO_CLIENT_KEY="/tmp/client.key"
export PGO_APISERVER_URL='https://127.0.0.1:8443'
export PGO_NAMESPACE=pgouser1
EOF

source ~/.bashrc
```

NOTE: For macOS users, you must use `~/.bash_profile` instead of `~/.bashrc`

Step 5: Install the PostgreSQL Operator Client `pgo`

The pgo client provides a helpful command-line interface to perform key operations on a PostgreSQL Operator, such as creating a PostgreSQL cluster.

The `pgo` client can be downloaded from GitHub Releases (subscribers can download it from the Crunchy Data Customer Portal).

Note that the `pgo` client's version must match the version of the PostgreSQL Operator that you have deployed. For example, if you have deployed version 4.3.0 of the PostgreSQL Operator, you must use the `pgo` for 4.3.0.

Once you have download the `pgo` client, change the permissions on the file to be executable if need be as shown below:

```
chmod +x pgo
```

Step 6: Connect to the PostgreSQL Operator

Finally, let's see if we can connect to the PostgreSQL Operator from the `pgo` client. In order to communicate with the PostgreSQL Operator API server, you will first need to set up a port forward to your local environment.

In a new console window, run the following command to set up a port forward:

```
kubectl -n pgo port-forward svc/postgres-operator 8443:8443
```

Back to your original console window, you can verify that you can connect to the PostgreSQL Operator using the following command:

```
pgo version
```

If successful, you should see output similar to this:

```
pgo client version 4.3.0
pgo-apiserver version 4.3.0
```

Step 7: Create a Namespace

We are almost there! You can optionally add a namespace that can be managed by the PostgreSQL Operator to watch and to deploy a PostgreSQL cluster into.

```
pgo create namespace wateringhole
```

verify the operator has access to the newly added namespace

```
pgo show namespace --all
```

you should see out put similar to this:

```
pgo username: admin
namespace                useraccess          installaccess
application-system       accessible          no access
default                  accessible          no access
kube-public              accessible          no access
kube-system              accessible          no access
pgo                      accessible          no access
wateringhole             accessible          accessible
```

Step 8: Have Some Fun - Create a PostgreSQL Cluster

You are now ready to create a new cluster in the `wateringhole` namespace, try the command below:

```
pgo create cluster -n wateringhole hippo
```

If successful, you should see output similar to this:

```
created Pgcluster hippo
workflow id 1cd0d225-7cd4-4044-b269-aa7bedae219b
```

This will create a PostgreSQL cluster named `hippo`. It may take a few moments for the cluster to be provisioned. You can see the status of this cluster using the `pgo test` command:

```
pgo test -n wateringhole hippo
```

When everything is up and running, you should see output similar to this:

```
cluster : hippo
    Services
        primary (10.97.140.113:5432): UP
    Instances
        primary (hippo-7b64747476-6dr4h): UP
```

The `pgo test` command provides you the basic information you need to connect to your PostgreSQL cluster from within your Kubernetes environment. For more detailed information, you can use `pgo show cluster -n wateringhole hippo`.

The goal of the Crunchy PostgreSQL Operator is to provide a means to quickly get your applications up and running on PostgreSQL for both development and production environments. To understand how the PostgreSQL Operator does this, we want to give you a tour of its architecture, with explains both the architecture of the PostgreSQL Operator itself as well as recommended deployment models for PostgreSQL in production!

Crunchy PostgreSQL Operator Architecture

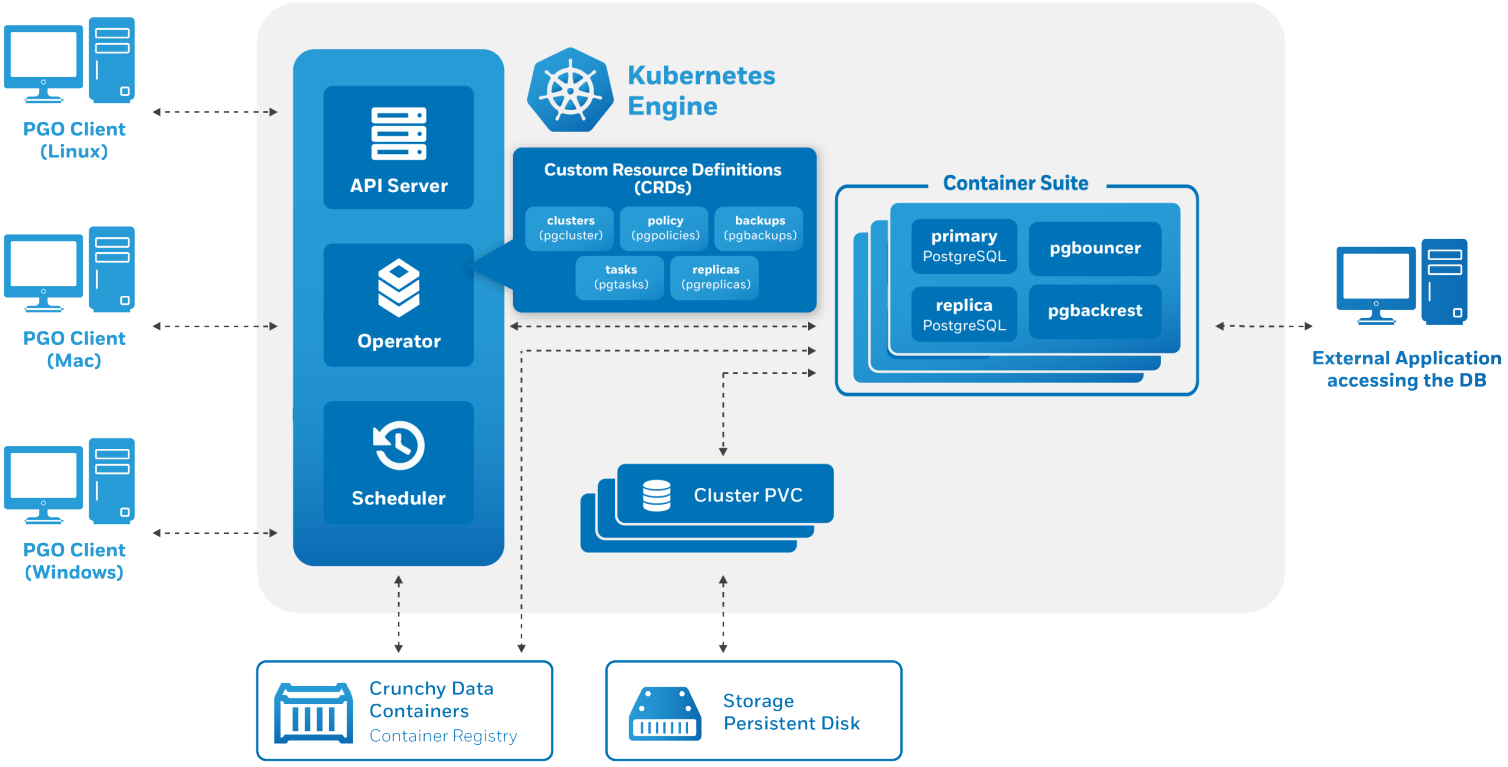

Figure 2: Operator Architecture with CRDs

The Crunchy PostgreSQL Operator extends Kubernetes to provide a higher-level abstraction for rapid creation and management of PostgreSQL clusters. The Crunchy PostgreSQL Operator leverages a Kubernetes concept referred to as "Custom Resources" to create several custom resource definitions (CRDs) that allow for the management of PostgreSQL clusters.

The Custom Resource Definitions include:

- `pgclusters.crunchydata.com`: Stores information required to manage a PostgreSQL cluster. This includes things like the cluster name, what storage and resource classes to use, which version of PostgreSQL to run, information about how to maintain a high-availability cluster, etc.
- `pgreplicas.crunchydata.com`: Stores information required to manage the replicas within a PostgreSQL cluster. This includes things like the number of replicas, what storage and resource classes to use, special affinity rules, etc.
- `pgtasks.crunchydata.com`: A general purpose CRD that accepts a type of task that is needed to run against a cluster (e.g. create a cluster, take a backup, perform a clone) and tracks the state of said task throw its workflow.
- `pgpolicies.crunchydata.com`: Stores a reference to a SQL file that can be executed against a PostgreSQL cluster. In the past, this was used to manage RLS policies on PostgreSQL clusters.

There are also a few legacy Custom Resource Definitions that the PostgreSQL Operator comes with that will be removed in a future release.

The PostgreSQL Operator runs as a deployment in a namespace and is composed of up to four Pods, including:

- `operator` (image: postgres-operator) - This is the heart of the PostgreSQL Operator. It contains a series of Kubernetes controllers that place watch events on a series of native Kubernetes resources (Jobs, Pods) as well as the Custom Resources that come with the PostgreSQL Operator (Pgcluster, Pgtask)
- `apiserver` (image: pgo-apiserver) - This provides an API that a PostgreSQL Operator User (`pgouser`) can interface with via the `pgo` command-line interface (CLI) or directly via HTTP requests. The API server can also control what resources a user can access via a series of RBAC rules that can be defined as part of a `pgorole`.
- `scheduler` (image: pgo-scheduler) - A container that runs `cron` and allows a user to schedule repeatable tasks, such as backups (because it is important to schedule backups in a production environment!)
- `event` (image: pgo-event, optional) - A container that provides an interface to the `nsq` message queue and transmits information about lifecycle events that occur within the PostgreSQL Operator (e.g. a cluster is created, a backup is taken, a clone fails to create)

The main purpose of the PostgreSQL Operator is to create and update information around the structure of a PostgreSQL Cluster, and to relay information about the overall status and health of a PostgreSQL cluster. The goal is to also simplify this process as much as possible for users. For example, let's say we want to create a high-availability PostgreSQL cluster that has a single replica, supports having backups in both a local storage area and Amazon S3 and has built-in metrics and connection pooling, similar to:

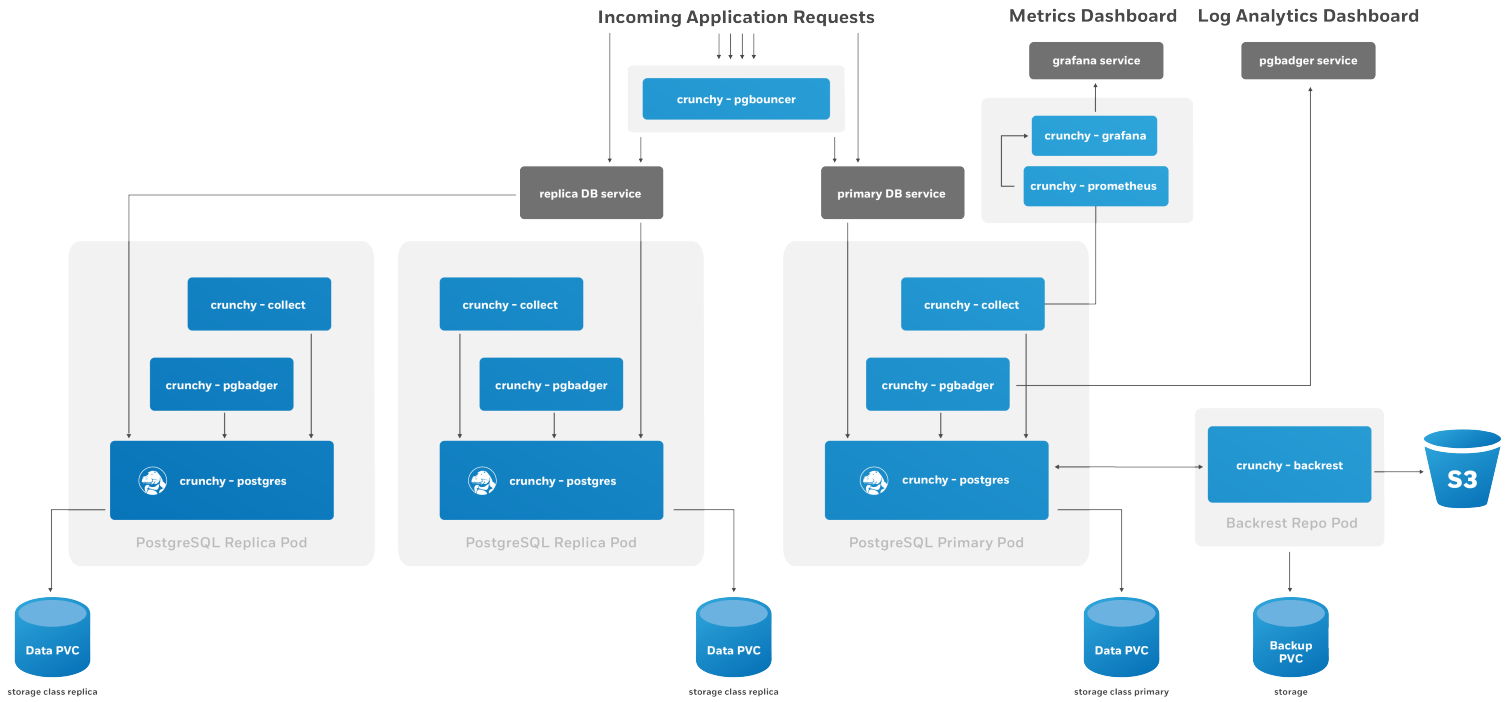

Figure 3: PostgreSQL HA Cluster

We can accomplish that with a single command:

```
pgo create cluster hacluster --replica-count=1 --metrics --pgbackrest-storage-type="local,s3"
    --pgbouncer --pgbadger
```

The PostgreSQL Operator handles setting up all of the various Deployments and sidecars to be able to accomplish this task, and puts in the various constructs to maximize resiliency of the PostgreSQL cluster.

You will also notice that **high-availability is enabled by default**. The Crunchy PostgreSQL Operator uses a distributed-consensus method for PostgreSQL cluster high-availability, and as such delegates the management of each cluster's availability to the clusters themselves. This removes the PostgreSQL Operator from being a single-point-of-failure, and has benefits such as faster recovery times for each PostgreSQL cluster. For a detailed discussion on high-availability, please see the High-Availability section.

Every single Kubernetes object (Deployment, Service, Pod, Secret, Namespace, etc.) that is deployed or managed by the PostgreSQL Operator has a Label associated with the name of `vendor` and a value of `crunchydata`. You can use Kubernetes selectors to easily find out which objects are being watched by the PostgreSQL Operator. For example, to get all of the managed Secrets in the default namespace the PostgreSQL Operator is deployed into (`pgo`):

```
kubectl get secrets -n pgo --selector=vendor=crunchydata
```

Kubernetes Deployments: The Crunchy PostgreSQL Operator Deployment Model

The Crunchy PostgreSQL Operator uses Kubernetes Deployments for running PostgreSQL clusters instead of StatefulSets or other objects. This is by design: Kubernetes Deployments allow for more flexibility in how you deploy your PostgreSQL clusters.

For example, let's look at a specific PostgreSQL cluster where we want to have one primary instance and one replica instance. We want to ensure that our primary instance is using our fastest disks and has more compute resources available to it. We are fine with our replica having slower disks and less compute resources. We can create this envirionment with a command similar to below:

```
pgo create cluster mixed --replica-count=1 \
  --storage-config=fast --resources-config=large \
  --replica-storage-config=standard --resources-config=medium
```

Now let's say we want to have one replica available to run read-only queries against, but we want its hardware profile to mirror that of the primary instance. We can run the following command:

```
pgo scale mixed --replica-count=1 \
  --storage-config=fast --resources-config=large
```

Kubernetes Deployments allow us to create heterogeneous clusters with ease and let us scale them up and down as we please. Additional components in our PostgreSQL cluster, such as the pgBackRest repository or an optional pgBouncer, are deployed as Kubernetes Deployments as well.

We can also leverage Kubernees Deployments to apply Node Affinity rules to individual PostgreSQL instances. For instance, we may want to force one or more of our PostgreSQL replicas to run on Nodes in a different region than our primary PostgreSQL instances.

Using Kubernetes Deployments does create additional management complexity, but the good news is: the PostgreSQL Operator manages it for you! Being aware of this model can help you understand how the PostgreSQL Operator gives you maximum flexibility in for your PostgreSQL clusters while giving you the tools to troubleshoot issues in production.

The last piece of this model is the use of Kubernetes Services for accessing your PostgreSQL clusters and their various components. The PostgreSQL Operator puts services in front of each Deployment to ensure you have a known, consistent means of accessing your PostgreSQL components.

Note that in some production environments, there can be delays in accessing Services during transition events. The PostgreSQL Operator attempts to mitigate delays during critical operations (e.g. failover, restore, etc.) by directly accessing the Kubernetes Pods to perform given actions.

For a detailed analysis, please see Using Kubernetes Deployments for Running PostgreSQL.

Additional Architecture Information

There is certainly a lot to unpack in the overall architecture of the Crunchy PostgreSQL Operator. Understanding the architecture will help you to plan the deployment model that is best for your environment. For more information on the architectures of various components of the PostgreSQL Operator, please read onward!

What happens when the Crunchy PostgreSQL Operator creates a PostgreSQL cluster?

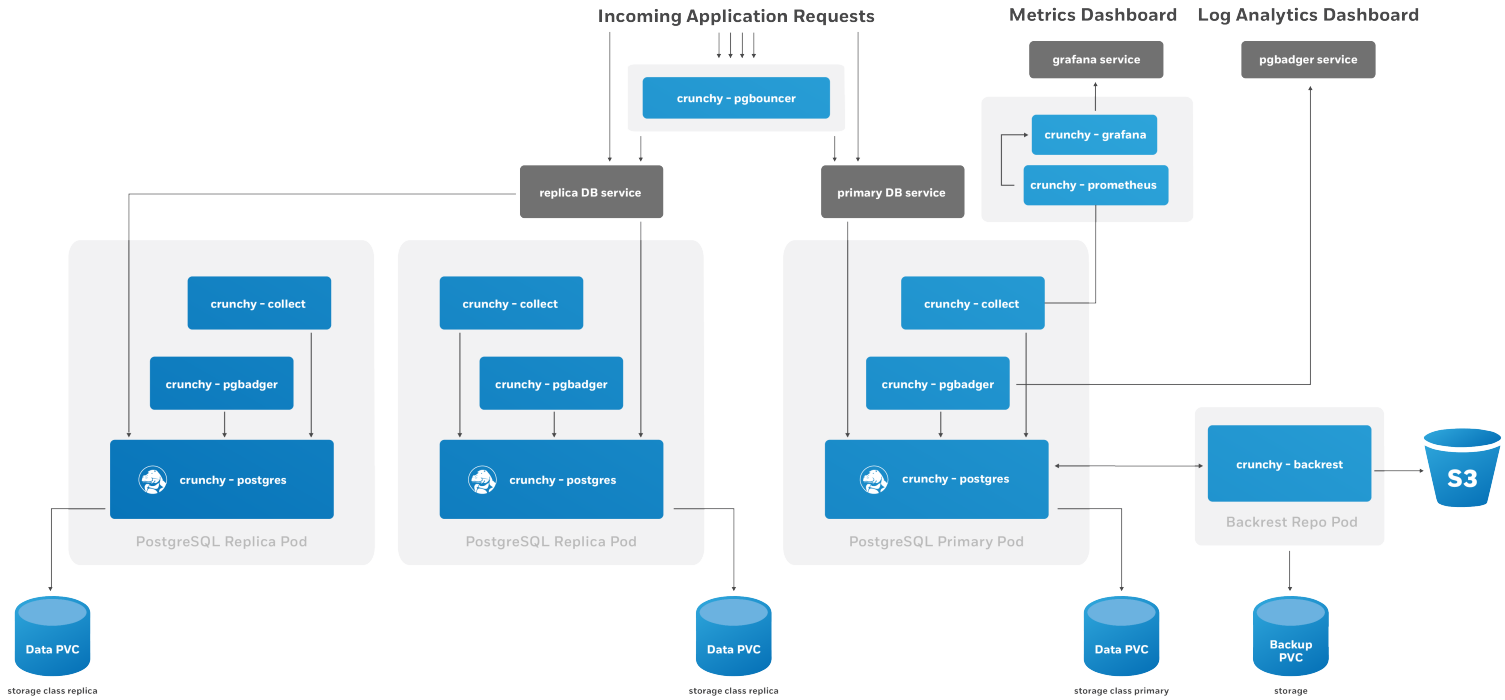

Figure 4: PostgreSQL HA Cluster

First, an entry needs to be added to the `Pgcluster` CRD that provides the essential attributes for maintaining the definition of a PostgreSQL cluster. These attributes include:

- Cluster name

- The storage and resource definitions to use
- References to any secrets required, e.g. ones to the pgBackRest repository
- High-availability rules
- Which sidecars and ancillary services are enabled, e.g. pgBouncer, pgMonitor

After the Pgcluster CRD entry is set up, the PostgreSQL Operator handles various tasks to ensure that a healthy PostgreSQL cluster can be deployed. These include:

- Allocating the PersistentVolumeClaims that are used to store the PostgreSQL data as well as the pgBackRest repository
- Setting up the Secrets specific to this PostgreSQL cluster
- Setting up the ConfigMap entries specific for this PostgreSQL cluster, including entries that may contain custom configurations as well as ones that are used for the PostgreSQL cluster to manage its high-availability
- Creating Deployments for the PostgreSQL primary instance and the pgBackRest repository

You will notice the presence of a pgBackRest repository. As of version 4.2, this is a mandatory feature for clusters that are deployed by the PostgreSQL Operator. In addition to providing an archive for the PostgreSQL write-ahead logs (WAL), the pgBackRest repository serves several critical functions, including:

- Used to efficiently provision new replicas that are added to the PostgreSQL cluster
- Prevent replicas from falling out of sync from the PostgreSQL primary by allowing them to replay old WAL logs
- Allow failed primaries to automatically and efficiently heal using the "delta restore" feature
- Serves as the basis for the cluster cloning feature
- …and of course, allow for one to take full, differential, and incremental backpus and perform full and point-in-time restores

The pgBackRest repository can be configured to use storage that resides within the Kubernetes cluster (the `local` option), Amazon S3 or a storage system that uses the S3 protocol (the `s3` option), or both (`local,s3`).

Once the PostgreSQL primary instance is ready, there are two follow up actions that the PostgreSQL Operator takes to properly leverage the pgBackRest repository:

- A new pgBackRest stanza is created
- An initial backup is taken to facilitate the creation of any new replica

At this point, if new replicas were requested as part of the `pgo create` command, they are provisioned from the pgBackRest repository.

There is a Kubernetes Service created for the Deployment of the primary PostgreSQL instance, one for the pgBackRest repository, and one that encompasses all of the replicas. Additionally, if the connection pooler pgBouncer is deployed with this cluster, it will also have a service as well.

An optional monitoring sidecar can be deployed as well. The sidecar, called `collect`, uses the `crunchy-collect` container that is a part of pgMonitor and scrapes key health metrics into a Prometheus instance. See Monitoring for more information on how this works.

Horizontal Scaling

There are many reasons why you may want to horizontally scale your PostgreSQL cluster:

- Add more redundancy by having additional replicas
- Leveraging load balancing for your read only queries
- Add in a new replica that has more storage or a different container resource profile, and then failover to that as the new primary

and more.

The PostgreSQL Operator enables the ability to scale up and down via the `pgo scale` and `pgo scaledown` commands respectively. When you run `pgo scale`, the PostgreSQL Operator takes the following steps:

- The PostgreSQL Operator creates a new Kubernetes Deployment with the information specified from the `pgo scale` command combined with the information already stored as part of the managing the existing PostgreSQL cluster
- During the provisioning of the replica, a pgBackRest restore takes place in order to bring it up to the point of the last backup. If data already exists as part of this replica, then a "delta restore" is performed. (**NOTE**: If you have not taken a backup in awhile and your database is large, consider taking a backup before performing scaling up.)
- The new replica boots up in recovery mode and recovers to the latest point in time. This allows it to catch up to the current primary.
- Once the replica has recovered, it joins the primary as a streaming replica!

If pgMonitor is enabled, a `collect` sidecar is also added to the replica Deployment.

Scaling down works in the opposite way:

- The PostgreSQL instance on the scaled down replica is stopped. By default, the data is explicitly wiped out unless the `--keep-data` flag on `pgo scaledown` is specified. Once the data is removed, the PersistentVolumeClaim (PVC) is also deleted
- The Kubernetes Deployment associated with the replica is removed, as well as any other Kubernetes objects that are specifically associated with this replcia

Deprovisioning

There may become a point where you need to completely deprovision, or delete, a PostgreSQL cluster. You can delete a cluster managed by the PostgreSQL Operator using the `pgo delete` command. By default, all data and backups are removed when you delete a PostgreSQL cluster, but there are some options that allow you to retain data, including:

- `--keep-backups` - this retains the pgBackRest repository. This can be used to restore the data to a new PostgreSQL cluster.
- `--keep-data` - this retains the PostgreSQL data directory (aka `PGDATA`) from the primary PostgreSQL instance in the cluster. This can be used to recreate the PostgreSQL cluster of the same name.

When the PostgreSQL cluster is deleted, the following takes place:

- All PostgreSQL instances are stopped. By default, the data is explicitly wiped out unless the `--keep-data` flag on `pgo scaledown` is specified. Once the data is removed, the PersistentVolumeClaim (PVC) is also deleted
- Any Services, ConfigMaps, Secrets, etc. Kubernetes objects are all deleted
- The Kubernetes Deployments associated with the PostgreSQL instances are removed, as well as the Kubernetes Deployments associated with pgBackRest repository and, if deployed, the pgBouncer connection pooler

When using the PostgreSQL Operator, the answer to the question "do you take backups of your database" is automatically "yes!"

The PostgreSQL Operator uses the open source pgBackRest backup and restore utility that is designed for working with databases that are many terabytes in size. As described in the Provisioning section, pgBackRest is enabled by default as it permits the PostgreSQL Operator to automate some advanced as well as convenient behaviors, including:

- Efficient provisioning of new replicas that are added to the PostgreSQL cluster
- Preventing replicas from falling out of sync from the PostgreSQL primary by allowing them to replay old WAL logs
- Allowing failed primaries to automatically and efficiently heal using the "delta restore" feature
- Serving as the basis for the cluster cloning feature
- …and of course, allowing for one to take full, differential, and incremental backpus and perform full and point-in-time restores

The PostgreSQL Operator leverages a pgBackRest repository to facilitate the usage of the pgBackRest features in a PostgreSQL cluster. When a new PostgreSQL cluster is created, it simultaneously creates a pgBackRest repository as described in the Provisioning section.

At PostgreSQL cluster creation time, you can specify a specific Storage Class for the pgBackRest repository. Additionally, you can also specify the type of pgBackRest repository that can be used, including:

- `local`: Uses the storage that is provided by the Kubernetes cluster's Storage Class that you select
- `s3`: Use Amazon S3 or an object storage system that uses the S3 protocol
- `local,s3`: Use both the storage that is provided by the Kubernetes cluster's Storage Class that you select AND Amazon S3 (or equivalent object storage system that uses the S3 protocol)

The pgBackRest repository consists of the following Kubernetes objects:

- A Deployment
- A Secret that contains information that is specific to the PostgreSQL cluster that it is deployed with (e.g. SSH keys, AWS S3 keys, etc.)
- A Service

The PostgreSQL primary is automatically configured to use the `pgbackrest archive-push` and push the write-ahead log (WAL) archives to the correct repository.

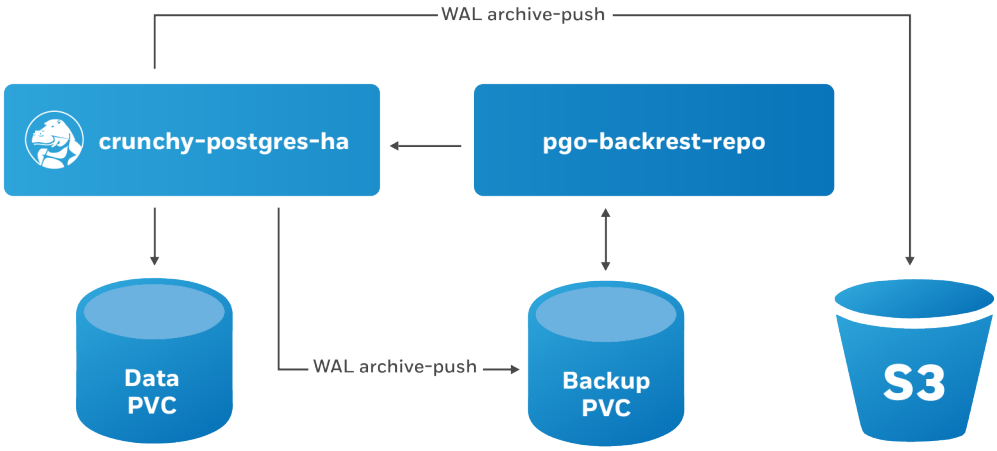

Figure 5: PostgreSQL Operator pgBackRest Integration

Backups

Backups can be taken with the `pgo backup` command

The PostgreSQL Operator supports three types of pgBackRest backups:

- Full (`full`): A full backup of all the contents of the PostgreSQL cluster
- Differential (`diff`): A backup of only the files that have changed since the last full backup
- Incremental (`incr`): A backup of only the files that have changed since the last full or differential backup

By default, `pgo backup` will attempt to take an **incremental (`incr`)** backup unless otherwise specified.

For example, to specify a full backup:

```
pgo backup hacluster --backup-opts="--type=full"
```

The PostgreSQL Operator also supports setting pgBackRest retention policies as well for backups. For example, to take a full backup and to specify to only keep the last 7 backups:

```
pgo backup hacluster --backup-opts="--type=full --repo1-retention-full=7"
```

Restores

The PostgreSQL Operator supports the ability to perform a full restore on a PostgreSQL cluster as well as a point-in-time-recovery using the `pgo restore` command. Note that both of these options are **destructive** to the existing PostgreSQL cluster; to "restore" the PostgreSQL cluster to a new deployment, please see the Clone section.

The `pgo restore` command lets you specify the point at which you want to restore your database using the `--pitr-target` flag with the `pgo restore` command.

NOTE: Ensure you are backing up your PostgreSQL cluster regularly, as this will help expedite your restore times. The next section will cover scheduling regular backups.

When the PostgreSQL Operator issues a restore, the following actions are taken on the cluster:

- The PostgreSQL Operator disables the "autofail" mechanism so that no failovers will occur during the restore.
- Any replicas that may be associated with the PostgreSQL cluster are destroyed
- A new Persistent Volume Claim (PVC) is allocated using the specifications provided for the primary instance. This may have been set with the `--storage-class` flag when the cluster was originally created
- A Kubernetes Job is created that will perform a pgBackRest restore operation to the newly allocated PVC. This is facilitated by the `pgo-backrest-restore` container image.

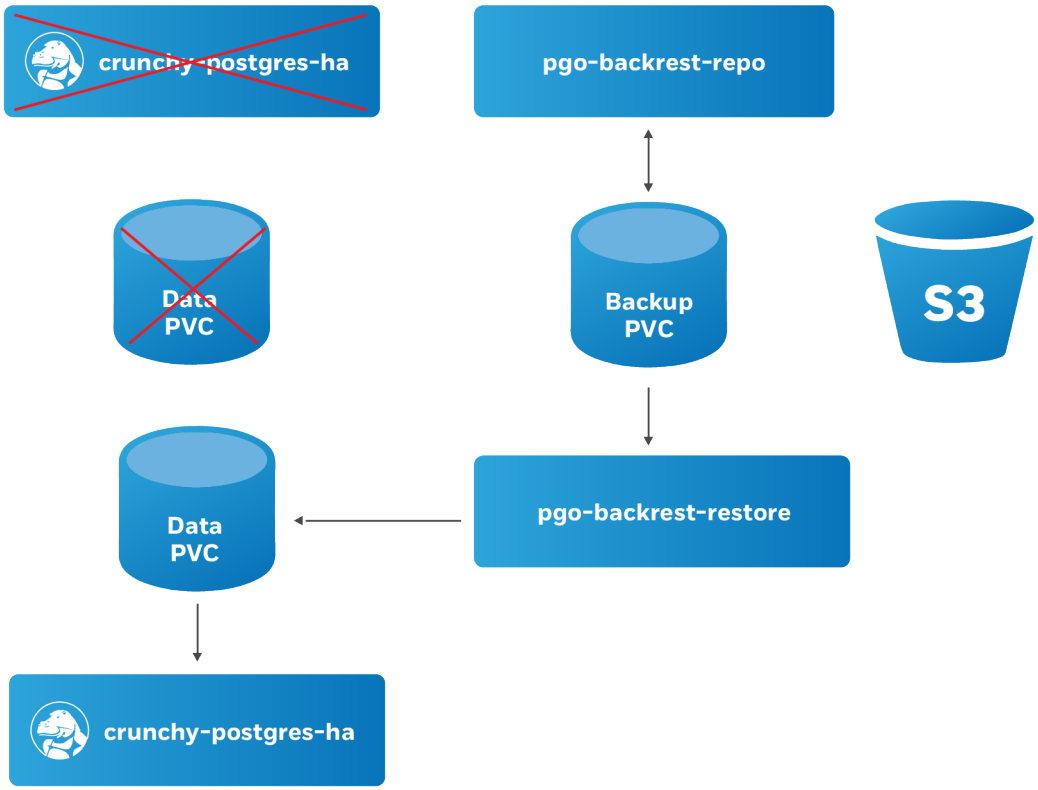

Figure 6: PostgreSQL Operator Restore Step 1

- When restore Job successfully completes, a new Deployment for the PostgreSQL cluster primary instance is created. A recovery is then issued to the specified point-in-time, or if it is a full recovery, up to the point of the latest WAL archive in the repository.
- Once the PostgreSQL primary instance is available, the PostgreSQL Operator will take a new, full backup of the cluster.

At this point, the PostgreSQL cluster has been restored. However, you will need to re-enable autofail if you would like your PostgreSQL cluster to be highly-available. You can re-enable autofail with this command:

```
pgo update cluster hacluster --autofail=true
```

Scheduling Backups

Any effective disaster recovery strategy includes having regularly scheduled backups. The PostgreSQL Operator enables this through its scheduling sidecar that is deployed alongside the Operator.

The PostgreSQL Operator Scheduler is essentially a cron server that will run jobs that it is specified. Schedule commands use the cron syntax to set up scheduled tasks.

For example, to schedule a full backup once a day at 1am, the following command can be used:

```
pgo create schedule hacluster --schedule="0 1 * * *" \
  --schedule-type=pgbackrest  --pgbackrest-backup-type=full
```

To schedule an incremental backup once every 3 hours:

```
pgo create schedule hacluster --schedule="0 */3 * * *" \
  --schedule-type=pgbackrest  --pgbackrest-backup-type=incr
```

Setting Backup Retention Policies

Unless specified, pgBackRest will keep an unlimited number of backups. As part of your regularly scheduled backups, it is encouraged for you to set a retention policy. This can be accomplished using the `--repo1-retention-full` for full backups and `--repo1-retention-diff` for differential backups via the `--schedule-opts` parameter.

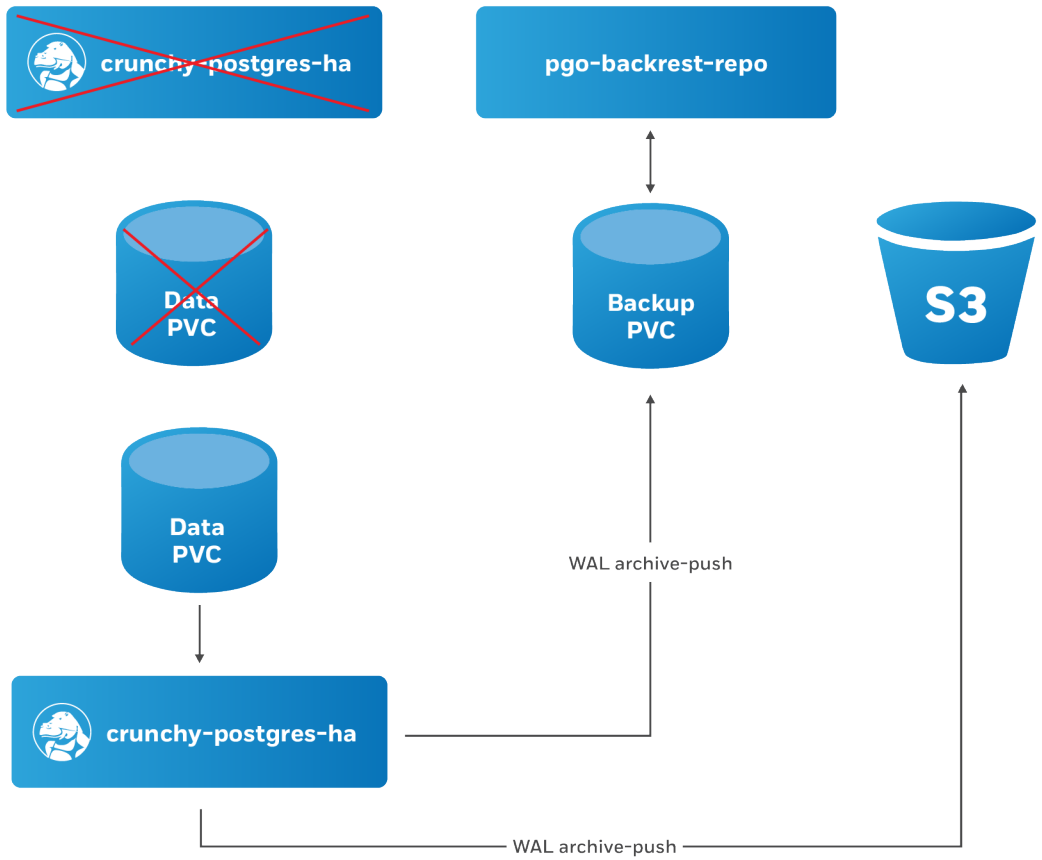

Figure 7: PostgreSQL Operator Restore Step 2

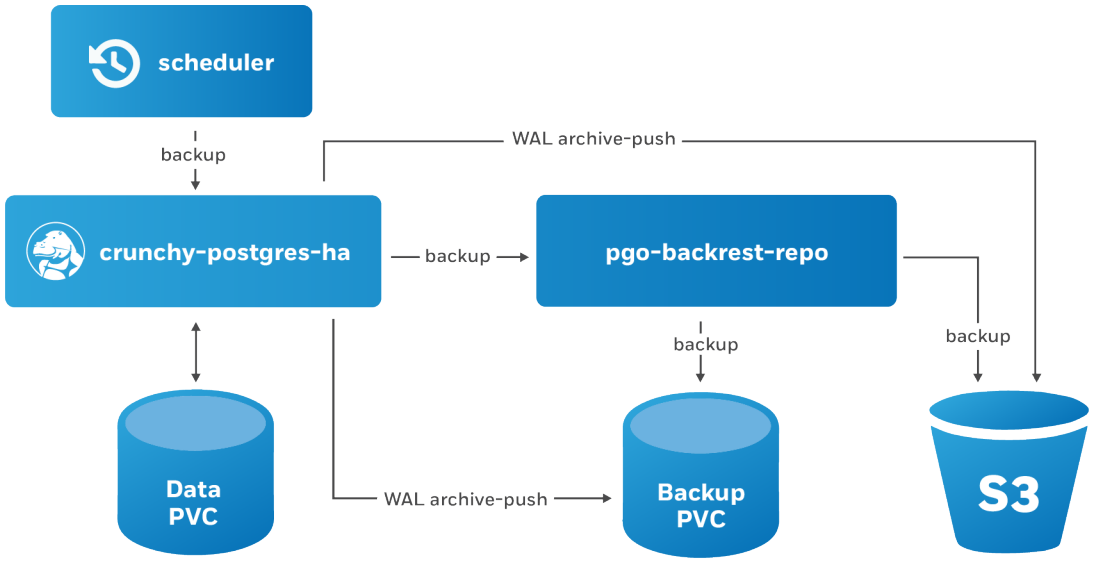

Figure 8: PostgreSQL Operator Schedule Backups

For example, using the above example of taking a nightly full backup, you can specify a policy of retaining 21 backups using the following command:

```
pgo create schedule hacluster --schedule="0 1 * * *" \
  --schedule-type=pgbackrest  --pgbackrest-backup-type=full \
  --schedule-opts="--repo1-retention-full=21"
```

Schedule Expression Format

Schedules are expressed using the following rules, which should be familiar to users of cron:

```
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  | * / , - ?
```

Using S3

The PostgreSQL Operator integration with pgBackRest allows it to use the AWS S3 object storage system, as well as other object storage systems that implement the S3 protocol.

In order to enable S3 storage, it is helpful to provide some of the S3 information prior to deploying the PostgreSQL Operator, or updating the `pgo-config` ConfigMap and restarting the PostgreSQL Operator pod.

First, you will need to add the proper S3 bucket name, AWS S3 endpoint and the AWS S3 region to the `Cluster` section of the `pgo.yaml` configuration file:

```
Cluster:
  BackrestS3Bucket: my-postgresql-backups-example
  BackrestS3Endpoint: s3.amazonaws.com
  BackrestS3Region: us-east-1
```

These values can also be set on a per-cluster basis with the `pgo create cluster` command, i.e.:

- `--pgbackrest-s3-bucket` - specifics the AWS S3 bucket that should be utilized
- `--pgbackrest-s3-endpoint` specifies the S3 endpoint that should be utilized
- `--pgbackrest-s3-key` - specifies the AWS S3 key that should be utilized
- `--pgbackrest-s3-key-secret`- specifies the AWS S3 key secret that should be utilized
- `--pgbackrest-s3-region` - specifies the AWS S3 region that should be utilized

Sensitive information, such as the values of the AWS S3 keys and secrets, are stored in Kubernetes Secrets and are securely mounted to the PostgreSQL clusters.

To enable a PostgreSQL cluster to use S3, the `--pgbackrest-storage-type` on the `pgo create cluster` command needs to be set to `s3` or `local,s3`.

Once configured, the `pgo backup` and `pgo restore` commands will work with S3 similarly to the above!

One of the great things about PostgreSQL is its reliability: it is very stable and typically "just works." However, there are certain things that can happen in the environment that PostgreSQL is deployed in that can affect its uptime, including:

- The database storage disk fails or some other hardware failure occurs
- The network on which the database resides becomes unreachable
- The host operating system becomes unstable and crashes
- A key database file becomes corrupted
- A data center is lost

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.

Fortunately, the Crunchy PostgreSQL Operator is prepared for this.

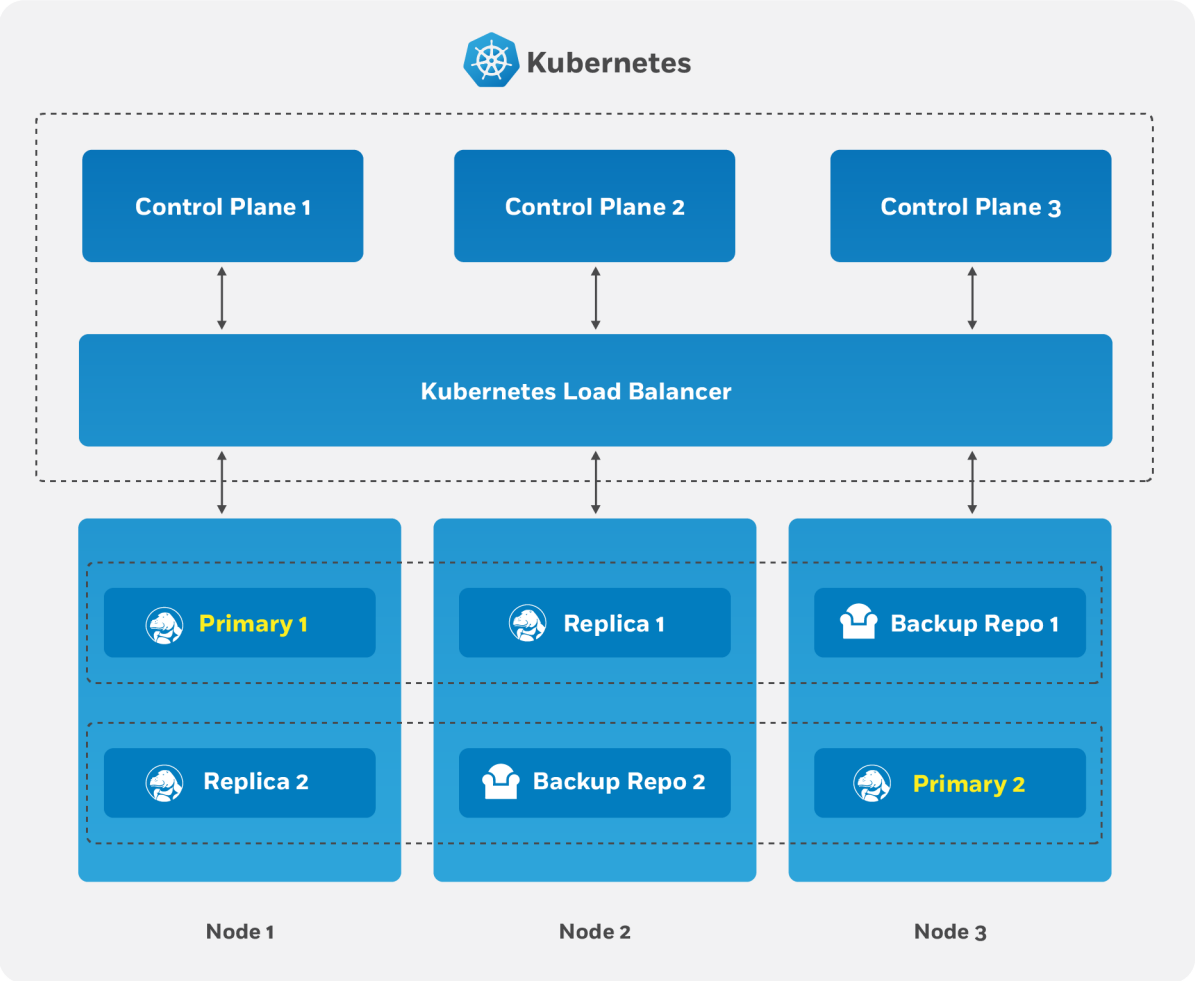

Figure 9: PostgreSQL Operator High-Availability Overview

The Crunchy PostgreSQL Operator supports a distributed-consensus based high-availability (HA) system that keeps its managed PostgreSQL clusters up and running, even if the PostgreSQL Operator disappears. Additionally, it leverages Kubernetes specific features such as Pod Anti-Affinity to limit the surface area that could lead to a PostgreSQL cluster becoming unavailable. The PostgreSQL Operator also supports automatic healing of failed primaries and leverages the efficient pgBackRest "delta restore" method, which eliminates the need to fully reprovision a failed cluster!

The Crunchy PostgreSQL Operator also maintains high-availability during a routine task such as a PostgreSQL minor version upgrade.

For workloads that are sensitive to transaction loss, the Crunchy PostgreSQL Operator supports PostgreSQL synchronous replication, which can be specified with the `--sync-replication` when using the `pgo create cluster` command.

(HA is enabled by default in any newly created PostgreSQL cluster. You can update this setting by either using the `--disable-autofail` flag when using `pgo create cluster`, or modify the `pgo-config` ConfigMap [or the `pgo.yaml` file] to set `DisableAutofail` to `"true"`. These can also be set when a PostgreSQL cluster is running using the `pgo update cluster` command).

One can also choose to manually failover using the `pgo failover` command as well.

The high-availability backing for your PostgreSQL cluster is only as good as your high-availability backing for Kubernetes. To learn more about creating a high-availability Kubernetes cluster, please review the Kubernetes documentation or consult your systems administrator.

The Crunchy PostgreSQL Operator High-Availability Algorithm

A critical aspect of any production-grade PostgreSQL deployment is a reliable and effective high-availability (HA) solution. Organizations want to know that their PostgreSQL deployments can remain available despite various issues that have the potential to disrupt operations, including hardware failures, network outages, software errors, or even human mistakes.

The key portion of high-availability that the PostgreSQL Operator provides is that it delegates the management of HA to the PostgreSQL clusters themselves. This ensures that the PostgreSQL Operator is not a single-point of failure for the availability of any of the PostgreSQL clusters that it manages, as the PostgreSQL Operator is only maintaining the definitions of what should be in the cluster (e.g. how many instances in the cluster, etc.).

Each HA PostgreSQL cluster maintains its availability using concepts that come from the Raft algorithm to achieve distributed consensus. The Raft algorithm ("Reliable, Replicated, Redundant, Fault-Tolerant") was developed for systems that have one "leader" (i.e. a primary) and one-to-many followers (i.e. replicas) to provide the same fault tolerance and safety as the PAXOS algorithm while being easier to implement.

For the PostgreSQL cluster group to achieve distributed consensus on who the primary (or leader) is, each PostgreSQL cluster leverages the distributed etcd key-value store that is bundled with Kubernetes. After it is elected as the leader, a primary will place a lock in the distributed etcd cluster to indicate that it is the leader. The "lock" serves as the method for the primary to provide a heartbeat: the primary will periodically update the lock with the latest time it was able to access the lock. As long as each replica sees that the lock was updated within the allowable automated failover time, the replicas will continue to follow the leader.

The "log replication" portion that is defined in the Raft algorithm is handled by PostgreSQL in two ways. First, the primary instance will replicate changes to each replica based on the rules set up in the provisioning process. For PostgreSQL clusters that leverage "synchronous replication," a transaction is not considered complete until all changes from those transactions have been sent to all replicas that are subscribed to the primary.

In the above section, note the key word that the transaction are sent to each replica: the replicas will acknowledge receipt of the transaction, but they may not be immediately replayed. We will address how we handle this further down in this section.

During this process, each replica keeps track of how far along in the recovery process it is using a "log sequence number" (LSN), a built-in PostgreSQL serial representation of how many logs have been replayed on each replica. For the purposes of HA, there are two LSNs that need to be considered: the LSN for the last log received by the replica, and the LSN for the changes replayed for the replica. The LSN for the latest changes received can be compared amongst the replicas to determine which one has replayed the most changes, and an important part of the automated failover process.

The replicas periodically check in on the lock to see if it has been updated by the primary within the allowable automated failover timeout. Each replica checks in at a randomly set interval, which is a key part of Raft algorithm that helps to ensure consensus during an election process. If a replica believes that the primary is unavailable, it becomes a candidate and initiates an election and votes for itself as the new primary. A candidate must receive a majority of votes in a cluster in order to be elected as the new primary.

There are several cases for how the election can occur. If a replica believes that a primary is down and starts an election, but the primary is actually not down, the replica will not receive enough votes to become a new primary and will go back to following and replaying the changes from the primary.

In the case where the primary is down, the first replica to notice this starts an election. Per the Raft algorithm, each available replica compares which one has the latest changes available, based upon the LSN of the latest logs received. The replica with the latest LSN wins and receives the vote of the other replica. The replica with the majority of the votes wins. In the event that two replicas' logs have the same LSN, the tie goes to the replica that initiated the voting request.

Once an election is decided, the winning replica is immediately promoted to be a primary and takes a new lock in the distributed etcd cluster. If the new primary has not finished replaying all of its transactions logs, it must do so in order to reach the desired state based on the LSN. Once the logs are finished being replayed, the primary is able to accept new queries.

At this point, any existing replicas are updated to follow the new primary.

When the old primary tries to become available again, it realizes that it has been deposed as the leader and must be healed. The old primary determines what kind of replica it should be based upon the CRD, which allows it to set itself up with appropriate attributes. It is then restored from the pgBackRest backup archive using the "delta restore" feature, which heals the instance and makes it ready to follow the new primary, which is known as "auto healing."

How The Crunchy PostgreSQL Operator Uses Pod Anti-Affinity

By default, when a new PostgreSQL cluster is created using the PostgreSQL Operator, pod anti-affinity rules will be applied to any deployments comprising the full PG cluster (please note that default pod anti-affinity does not apply to any Kubernetes jobs created by the PostgreSQL Operator). This includes:

- The primary PG deployment
- The deployments for each PG replica
- The `pgBackrest` dedicated repostiory deployment
- The `pgBouncer` deployment (if enabled for the cluster)

There are three types of Pod Anti-Affinity rules that the Crunchy PostgreSQL Operator supports:

- `preferred`: Kubernetes will try to schedule any pods within a PostgreSQL cluster to different nodes, but in the event it must schedule two pods on the same Node, it will. As described above, this is the default option.
- `required`: Kubernetes will schedule pods within a PostgreSQL cluster to different Nodes, but in the event it cannot schedule a pod to a different Node, it will not schedule the pod until a different node is available. While this guarantees that no pod will share the same node, it can also lead to downtime events as well. This uses the `requiredDuringSchedulingIgnoredDuringExecution` affinity rule.
- `disabled`: Pod Anti-Affinity is not used.

With the default `preferred` Pod Anti-Affinity rule enabled, Kubernetes will attempt to schedule pods created by each of the separate deployments above on a unique node, but will not guarantee that this will occur. This ensures that the pods comprising the PostgreSQL cluster can always be scheduled, though perhaps not always on the desired node. This is specifically done using the following:

- The `preferredDuringSchedulingIgnoredDuringExecution` affinity type, which defines an anti-affinity rule that Kubernetes will attempt to adhere to, but will not guarantee will occur during Pod scheduling
- A combination of labels that uniquely identify the pods created by the various Deployments listed above
- A topology key of `kubernetes.io/hostname`, which instructs Kubernetes to schedule a pod on specific Node only if there is not already another pod in the PostgreSQL cluster scheduled on that same Node

If you want to explicitly create a PostgreSQL cluster with the `preferred` Pod Anti-Affinity rule, you can execute the `pgo create` command using the `--pod-anti-affinity` flag similar to this:

```
pgo create cluster hacluster --replica-count=2 --pod-anti-affinity=preferred
```

or it can also be explicitly enabled globally for all clusters by setting `PodAntiAffinity` to `preferred` in the `pgo.yaml` configuration file.

If you want to create a PostgreSQL cluster with the `required` Pod Anti-Affinity rule, you can execute a command similar to this:

```
pgo create cluster hacluster --replica-count=2 --pod-anti-affinity=required
```

or set the `required` option globally for all clusters by setting `PodAntiAffinity` to `required` in the `pgo.yaml` configuration file.

When `required` is utilized for the default pod anti-affinity, a separate node is required for each deployment listed above comprising the PG cluster. This ensures that the cluster remains highly-available by ensuring that node failures do not impact any other deployments in the cluster. However, this does mean that the PostgreSQL primary, each PostgreSQL replica, the pgBackRest repository and, if deployed, the pgBouncer Pods will each require a unique node, meaning the minimum number of Nodes required for the Kubernetes cluster will increase as more Pods are added to the PostgreSQL cluster. Further, if an insufficient number of nodes are available to support this configuration, certain deployments will fail, since it will not be possible for Kubernetes to successfully schedule the pods for each deployment.

Synchronous Replication: Guarding Against Transactions Loss

Clusters managed by the Crunchy PostgreSQL Operator can be deployed with synchronous replication, which is useful for workloads that are sensitive to losing transactions, as PostgreSQL will not consider a transaction to be committed until it is committed to all synchronous replicas connected to a primary. This provides a higher guarantee of data consistency and, when a healthy synchronous replica is present, a guarantee of the most up-to-date data during a failover event.

This comes at a cost of performance as PostgreSQL: as PostgreSQL has to wait for a transaction to be committed on all synchronous replicas, a connected client will have to wait longer than if the transaction only had to be committed on the primary (which is how asynchronous replication works). Additionally, there is a potential impact to availability: if a synchronous replica crashes, any writes to the primary will be blocked until a replica is promoted to become a new synchronous replica of the primary.

You can enable synchronous replication by using the `--sync-replication` flag with the `pgo create` command, e.g.:

```
pgo create cluster hacluster --replica-count=2 --sync-replication
```

Node Affinity

Kubernetes Node Affinity can be used to scheduled Pods to specific Nodes within a Kubernetes cluster. This can be useful when you want your PostgreSQL instances to take advantage of specific hardware (e.g. for geospatial applications) or if you want to have a replica instance deployed to a specific region within your Kubernetes cluster for high-availability purposes.

The PostgreSQL Operator provides users with the ability to apply Node Affinity rules using the `--node-label` flag on the `pgo create` and the `pgo scale` commands. Node Affinity directs Kubernetes to attempt to schedule these PostgreSQL instances to the specified Node label.

To get a list of available Node labels:

```
kubectl get nodes --show-labels
```

You can then specify one of those Kubernetes node names (e.g. `region=us-east-1`) when creating a PostgreSQL cluster;

```
pgo create cluster thatcluster --node-label=region=us-east-1
```

The Node Affinity only uses the `preferred` scheduling strategy (similar to what is described in the Pod Anti-Affinity section above), so if a Pod cannot be scheduled to a particular Node matching the label, it will be scheduled to a different Node.

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
```

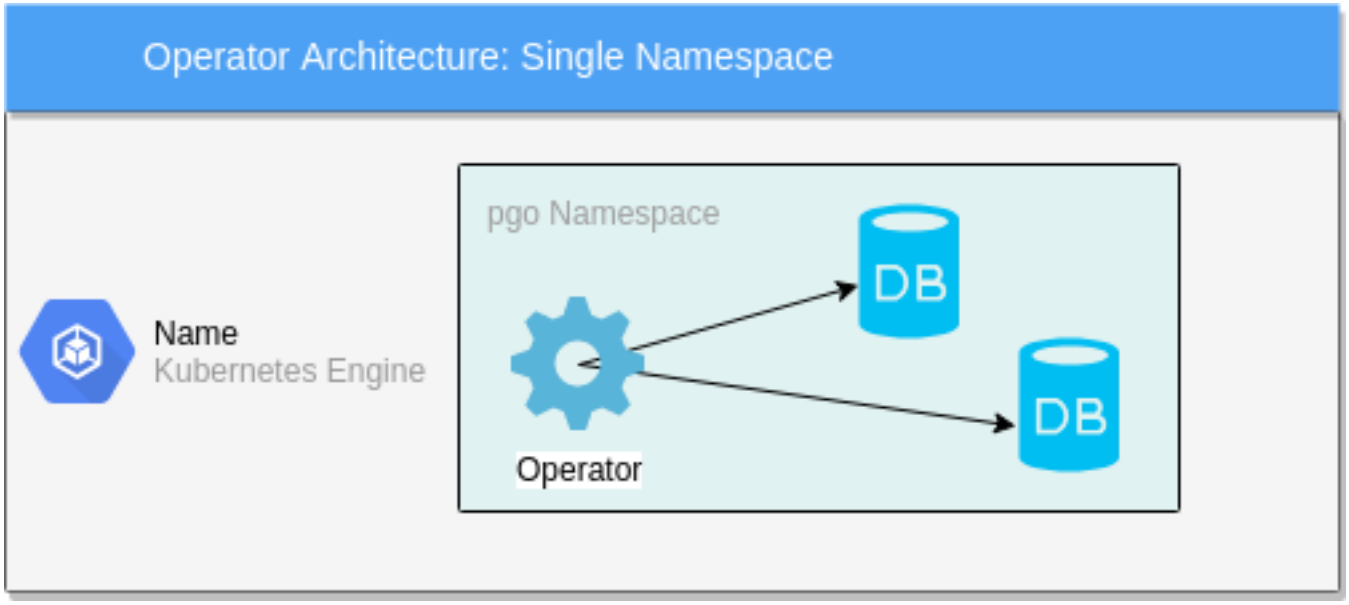

Figure 10: Reference

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
```

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
```

RBAC

To support multiple namespace watching, each namespace that the PostgreSQL Operator watches requires its own copy of the following resources:

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

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 Kubernetes system after running the install scripts:

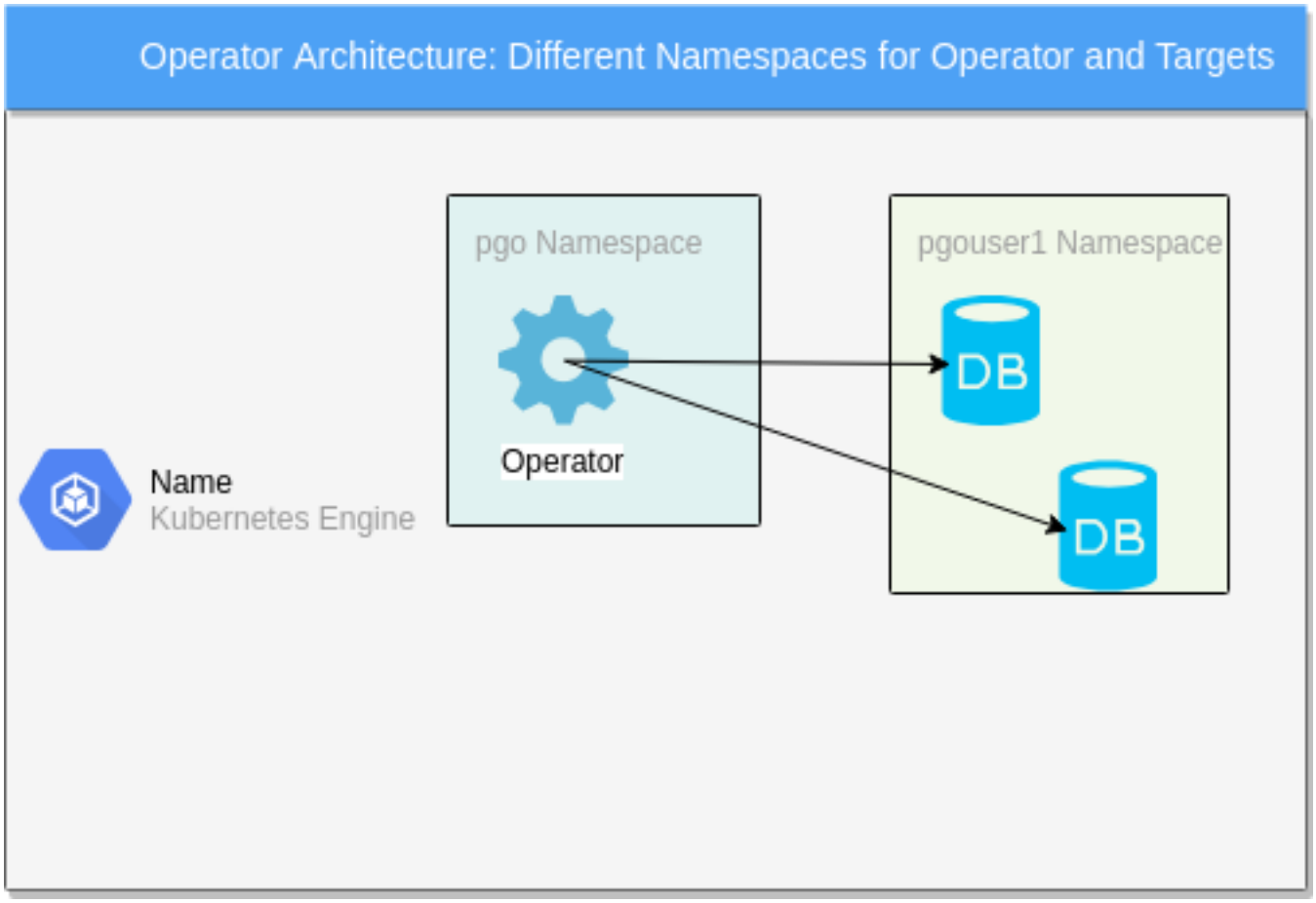

Figure 11: Reference

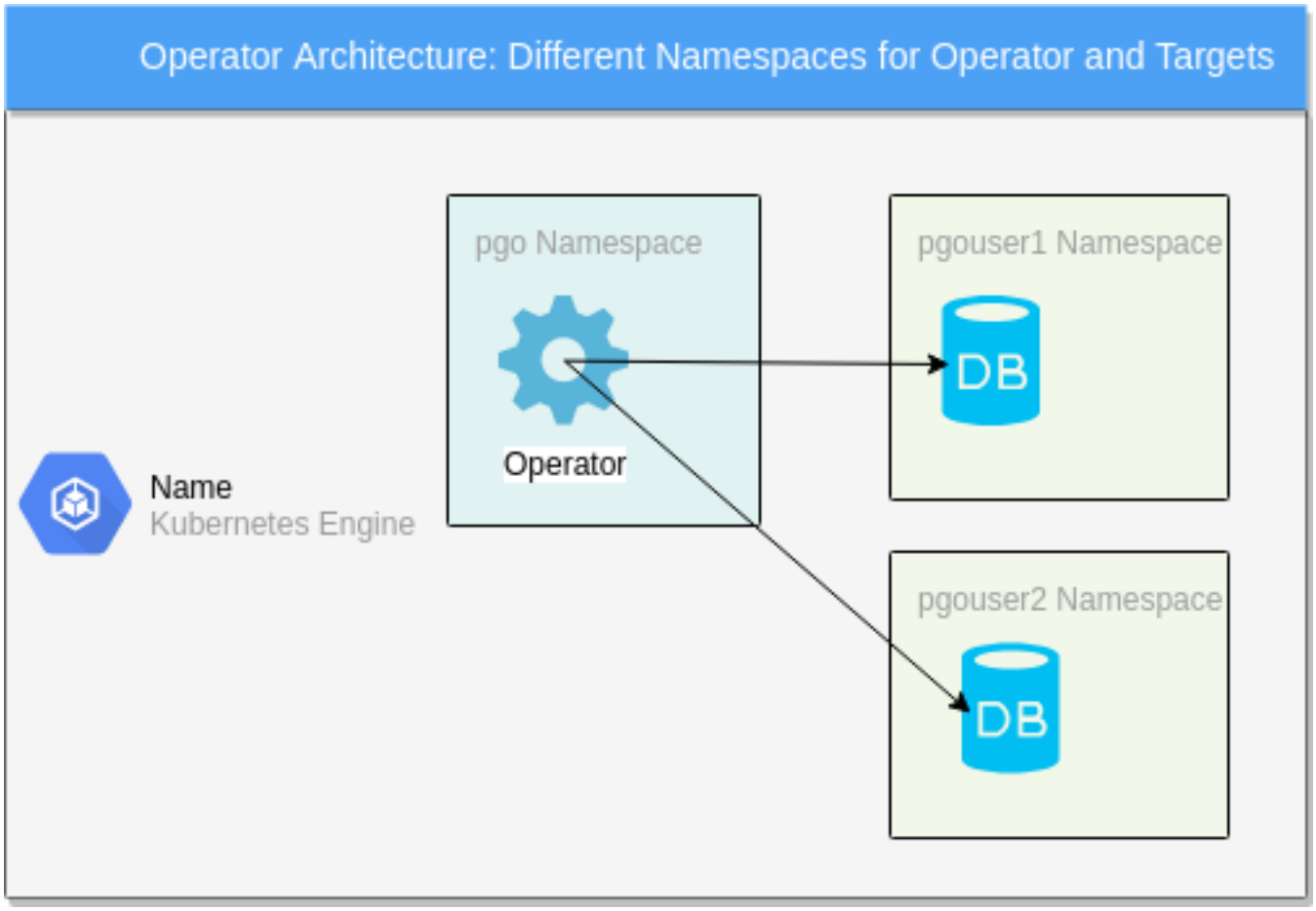

Figure 12: Reference

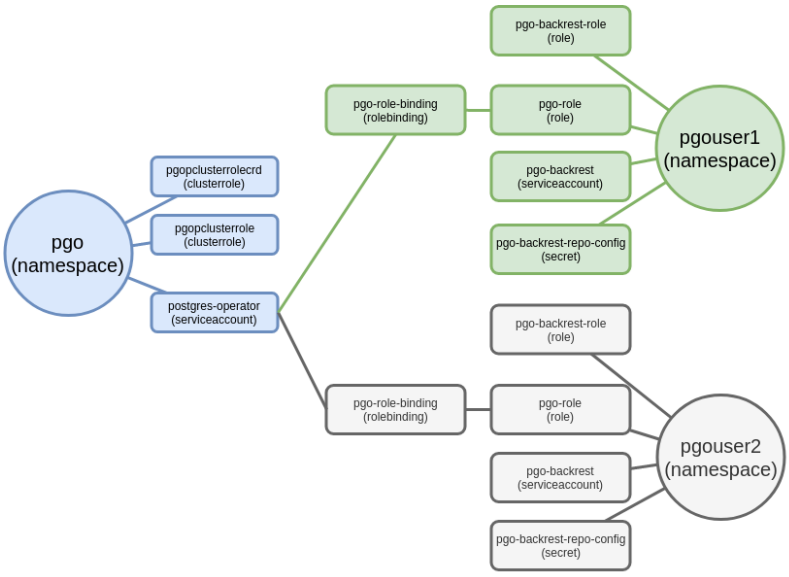

Figure 13: Reference

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.

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
```

```
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’

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-ha). PostgreSQL database server. The crunchy-postgres container image is unmodified, open source PostgreSQL packaged and maintained by Crunchy Data.
- **PostGIS** (crunchy-postgres-ha-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.
- **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

- **pgbouncer** (crunchy-pgbouncer). pgbouncer is a lightweight connection pooler for PostgreSQL. The crunchy-pgbouncer container provides a pgbouncer image.

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.

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.

A Tablespace is a PostgreSQL feature that is used to store data on a volume that is different from the primary data directory. While most workloads do not require them, tablespaces can be particularly helpful for larger data sets or utilizing particular hardware to optimize performance on a particular PostgreSQL object (a table, index, etc.). Some examples of use cases for tablespaces include:

- Partitioning larger data sets across different volumes
- Putting data onto archival systems
- Utilizing hardware (or a storage class) for a particular database
- Storing sensitive data on a volume that supports transparent data-encryption (TDE)

and others.

In order to use PostgreSQL tablespaces properly in a highly-available, distributed system, there are several considerations that need to be accounted for to ensure proper operations:

- Each tablespace must have its own volume; this means that every tablespace for every replica in a system must have its own volume.
- The filesystem map must be consistent across the cluster
- The backup & disaster recovery management system must be able to safely backup and restore data to tablespaces

Additionally, a tablespace is a critical piece of a PostgreSQL instance: if PostgreSQL expects a tablespace to exist and it is unavailable, this could trigger a downtime scenario.

While there are certain challenges with creating a PostgreSQL cluster with high-availability along with tablespaces in a Kubernetes-based environment, the PostgreSQL Operator adds many conveniences to make it easier to use tablespaces in applications.

How Tablespaces Work in the PostgreSQL Operator

As stated above, it is important to ensure that every tablespace created has its own volume (i.e. its own persistent volume claim). This is especially true for any replicas in a cluster: you don't want multiple PostgreSQL instances writing to the same volume, as this is a recipe for disaster!

One of the keys to working with tablespaces in a high-availability cluster is to ensure the filesystem that the tablespaces map to is consistent. Specifically, it is imperative to have the `LOCATION` parameter that is used by PostgreSQL to indicate where a tablespace resides to match in each instance in a cluster.

The PostgreSQL Operator achieves this by mounting all of its tablespaces to a directory called `/tablespaces` in the container. While each tablespace will exist in a unique PVC across all PostgreSQL instances in a cluster, each instance's tablespaces will mount in a predictable way in `/tablespaces`.

The PostgreSQL Operator takes this one step further and abstracts this away from you. When your PostgreSQL cluster initialized, the tablespace definition is automatically created in PostgreSQL; you can start using it immediately! An example of this is demonstrated in the next section.

The PostgreSQL Operator ensures the availability of the tablespaces across the different lifecycle events that occur on a PostgreSQL cluster, including:

- High-Availability: Data in the tablespaces is replicated across the cluster, and is available after a downtime event
- Disaster Recovery: Tablespaces are backed up and are properly restored during a recovery
- Clone: Tablespaces are created in any cloned cluster
- Deprovisioining: Tablespaces are deleted when a PostgreSQL instance or cluster is deleted

Creating Tablespaces

Tablespaces can be used in a cluster with the `pgo create cluster` command. The command follows this general format:

```
pgo create cluster hacluster --tablespaces=tablespace1=storageclass,tablespace2=storageclass
```

For example, to create tablespaces name `faststorage1` and `faststorage2` on PVCs that use the `nfsstorage` storage type, you would execute the following command:

```
pgo create cluster hacluster --tablespaces=faststorage1=nfsstorage,faststorage2=nfsstorage
```

Once the cluster is initialized, you can immediately interface with the tablespaces! For example, if you wanted to create a table called `sensor_data` on the `faststorage1` tablespace, you could execute the following SQL:

```
CREATE TABLE sensor_data (
  sensor_id int,
  sensor_value numeric,
  created_at timestamptz DEFAULT CURRENT_TIMESTAMP
)
TABLESPACE faststorage1;
```

For more information on how tablespaces work in PostgreSQL please refer to the PostgreSQL manual.

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
4.3.0	4.3.0	12.2	2.20
		11.7	2.20

Operator Release	Container Release	Postgres	PgBackrest Version
		10.12	2.20
		9.6.17	2.20
		9.5.21	2.20
4.2.1	4.3.0	12.1	2.20
		11.6	2.20
		10.11	2.20
		9.6.16	2.20
		9.5.20	2.20
4.2.0	4.3.0	12.1	2.20
		11.6	2.20
		10.11	2.20
		9.6.16	2.20
		9.5.20	2.20
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
		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

Operator Release	Container Release	Postgres	PgBackrest Version
		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 PostgreSQL Operator is developed on both CentOS 7 and RHEL 7 operating systems. The underlying containers are designed to use either CentOS 7 or Red Hat UBI 7 as the base container image.

Other Linux variants are possible but are not supported at this time.

Also, please note that as of version 4.2.2 of the PostgreSQL Operator, Red Hat Universal Base Image (UBI) 7 has replaced RHEL 7 as the base container image for the various PostgreSQL Operator containers. You can find out more information about Red Hat UBI from the following article:

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

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 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
```

The `/healthz` route is used by kubernetes probes and has its authentication disabed without requiring NOAUTH_ROUTES.

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
```

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 su
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
CCPImagePrefix	newly created containers will be based on this image prefix (e.g. crunchydata), update this if you require a
CCPImageTag	newly created containers will be based on this image version (e.g. centos7-12.2-4.3.0), unless you override i
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
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 metr
Badger	boolean, if set to true will cause each new cluster to include crunchy-pgbadger as a sidecar container for st
Policies	optional, list of policies to apply to a newly created cluster, comma separated, must be valid policies in th
PasswordAgeDays	optional, if set, will set the VALID UNTIL date on passwords to this many days in the future when creati
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
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
DisableReplicaStartFailReinit	if set to `true` will disable the detection of a “start failed” states in PG replicas, which results in the re-init
PodAntiAffinity	either `preferred`, `required` or `disabled` to either specify the type of affinity that should be utilized for t
SyncReplication	boolean, if set to `true` will automatically enable synchronous replication in new PostgreSQL clusters (defa

Storage

Setting	Definition
PrimaryStorage	required, the value of the storage configuration to use for the primary PostgreSQL deployment

Setting	Definition
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 Dep
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
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
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 Kubernetes 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.

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

```
export DISABLE_TLS=true
export 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 if using Ansible

```
pgo_tls_no_verify='false'
```

TLS Trust

Custom Trust Additions To configure the server to allow connections from any client presenting a certificate issued by CAs within a custom, PEM-encoded certificate list, set the following as a Bash environment variable

```
export TLS_CA_TRUST="/path/to/trust/file"
```

Or the following in the inventory file if using Ansible

```
pgo_tls_ca_store='/path/to/trust/file'
```

System Default Trust To configure the server to allow connections from any client presenting a certificate issued by CAs within the operating system's default trust store, set the following as a Bash environment variable

```
export ADD_OS_TRUSTSTORE=true
```

Or the following in the inventory file if using Ansible

```
pgo_add_os_ca_store='true'
```

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

By default, the pgo client will trust certificates issued by one of the Certificate Authorities listed in the operating system's default CA trust store, if any. To exclude them, either use the environment variable

```
EXCLUDE_OS_TRUST=true
```

or use the –exclude-os-trust flag

```
pgo version --exclude-os-trust
```

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
```

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 | Golang project directory `GOBIN` || GOPATH/bin
NAMESPACE	namespace	pgouser1
PGOROOT		$GOPATH/src/github.com/crunchydata/postgres-operator \|$ Operator repositor
PGO_CMD		kubectl
PGO_CLIENT_CERT		$PGOROOT/conf/postgres-operator/server.crt \|$ TLS Client certificate `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		4.3.0
TLS_NO_VERIFY	pgo_tls_no_verify	false
TLS_CA_TRUST	pgo_tls_ca_store	/var/pki/my_cas.crt
ADD_OS_TRUSTSTORE	pgo_add_os_ca_store	false
NOAUTH_ROUTES	pgo_noauth_routes	"/health"
EXCLUDE_OS_TRUST		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 v4.3.0
```

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 Kubernetes namespaces is typically something that only a privileged Kubernetes user can perform so log into your Kubernetes cluster as a user that has the necessary privileges.

On Openshift if you do not want to install the Operator as the system administrator, you can grant cluster-admin privileges 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 privileges 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 Kubernetes 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 the name of the namespace that the Operator will be installed 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 Kubernetes 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 privileges. 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 Kubernetes RBAC used by the Operator, run the following as a cluster-admin Kubernetes 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 "Configuration/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.

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

This guide is intended for those wanting to build the Operator from source or contribute via pull requests.

Prerequisites

The target development host for these instructions is a CentOS 7 or RHEL 7 host. Others operating systems are possible, however we do not support building or running the Operator on others at this time.

Environment Variables

The following environment variables are expected by the steps in this guide:

Variable	Example
`GOPATH`	$HOME/odev|Golangprojectdirectory`PGOROOT`|GOPATH/src/github.com/crunchydata/postgres-operato
`PGO_BASEOS`	centos7
`PGO_CMD`	kubectl
`PGO_IMAGE_PREFIX`	crunchydata
`PGO_OPERATOR_NAMESPACE`	pgo
`PGO_VERSION`	4.3.0

{{% notice tip %}} `examples/envs.sh` contains the above variable definitions as well as others used by postgres-operator tools {{% /notice %}}

Other requirements

- The development host has been created, has access to `yum` updates, and has a regular user account with `sudo` rights to run `yum`.
- `GOPATH` points to a directory containing `src`,`pkg`, and `bin` directories.
- The development host has `$GOPATH/bin` added to its `PATH` environment variable. Development tools will be installed to this path. Defining a `GOBIN` environment variable other than `$GOPATH/bin` may yield unexpected results.
- The development host has `git` installed and has cloned the postgres-operator repository to `$GOPATH/src/github.com/crunchydata/po` Makefile targets below are run from the repository directory.
- Deploying the Operator will require deployment access to a Kubernetes cluster. Clusters built on OpenShift Container Platform (OCP) or built using `kubeadm` are the validation targets for Pull Requests and thus recommended for devleopment. Instructions for setting up these clusters are outside the scope of this guide.

Building

Dependencies

Configuring build dependencies is automated via the `setup` target in the project Makefile:

```
make setup
```

The setup target ensures the presence of:

- `GOPATH` and `PATH` as described in the prerequisites
- EPEL yum repository
- golang compiler
- `dep` dependency manager
- NSQ messaging binaries
- `docker` container tool
- `buildah` OCI image building tool
- `expenv` config tool

By default, docker is not configured to run its daemon. Refer to the docker post-installation instructions to configure it to run once or at system startup. This is not done automatically.

Compile

{{% notice tip %}} Please be sure to have your GPG Key and `.repo` file in the `conf` directory before proceeding. {{% /notice %}}

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`.

Deployment

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.

Troubleshooting

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.

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
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

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 2.7.0+

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`
- `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
- ‘disable_auto_failover“
- 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	Required	Description
archive_mode	true	**Required**	Set to true enable archive
archive_timeout	60	**Required**	Set to a value in seconds t
backrest	false	**Required**	Set to true enable pgBack
backrest_aws_s3_bucket			Set to configure the bucke
backrest_aws_s3_endpoint			Set to configure the endpo
backrest_aws_s3_key			Set to configure the key us
backrest_aws_s3_region			Set to configure the region
backrest_aws_s3_secret			Set to configure the secret
backrest_storage	storageos	**Required**	Set to configure which sto
backup_storage	storageos	**Required**	Set to configure which sto
badger	false	**Required**	Set to true enable pgBadg
ccp_image_prefix	crunchydata	**Required**	Configures the image prefi
ccp_image_tag		**Required**	Configures the image tag
cleanup	false		Set to configure the playb
create_rbac	true	**Required**	Set to true if the installer
crunchy_debug	false		Set to configure Operator
delete_metrics_namespace	false		Set to configure whether c
delete_operator_namespace	false		Set to configure whether c
delete_watched_namespaces	false		Set to configure whether c
db_name	userdb	**Required**	Set to a value to configure
db_password_age_days	60	**Required**	Set to a value in days to c
db_password_length	24	**Required**	Set to configure the size o
db_port	5432	**Required**	Set to configure the defau

Name	Default	Required	Description
db_replicas	1	**Required**	Set to configure the amou
db_user	testuser	**Required**	Set to configure the usern
disable_failover	false	**Required**	Set to true disable auto fa
exporterport	9187	**Required**	Set to configure the defau
grafana_admin_password			Set to configure the login
grafana_admin_username	admin		Set to configure the login
grafana_install	true		Set to true to install Crun
grafana_storage_access_mode			Set to the access mode us
grafana_storage_class_name			Set to the name of the sto
grafana_volume_size			Set to the size of persisten
kubernetes_context		**Required**, if deploying to Kubernetes	When deploying to Kuber
log_statement	none		Set to `none`, `ddl`, `mod`, or
metrics	false	**Required**	Set to true enable perform
metrics_namespace	metrics		Configures the target nam
namespace			Set to a comma delimited
openshift_host		**Required**, if deploying to OpenShift	When deploying to OpenS
openshift_password		**Required**, if deploying to OpenShift	When deploying to OpenS
openshift_skip_tls_verify		**Required**, if deploying to OpenShift	When deploying to Opens
openshift_token		**Required**, if deploying to OpenShift	When deploying to OpenS
openshift_user		**Required**, if deploying to OpenShift	When deploying to OpenS
pgbadgerport	10000	**Required**	Set to configure the defau
pgo_add_os_ca_store	false		When true, includes syste
pgo_admin_username	admin	**Required**	Configures the pgo admini
pgo_admin_password		**Required**	Configures the pgo admini
pgo_admin_perms	*	**Required**	Sets the access control rul
pgo_admin_role_name	pgoadmin	**Required**	Sets the name of the Post
pgo_apiserver_port	8443		Set to configure the port u
pgo_client_install	true		Configures the playbooks
pgo_client_version		**Required**	Configures which version
pgo_disable_eventing	false		Set to configure whether
pgo_disable_tls	false		Set to configure whether
pgo_image_prefix	crunchydata	**Required**	Configures the image prefi
pgo_image_tag		**Required**	Configures the image tag
pgo_installation_name		**Required**	The name of the PGO ins
pgo_noauth_routes			Configures URL routes wi
pgo_operator_namespace		**Required**	Set to configure the names
pgo_tls_ca_store			Set to add additional Cert
pgo_tls_no_verify	false		Set to configure Operator
pgo_client_container_install	false		Installs the pgo-client dep
pgo_apiserver_url	https://postgres-operator		Sets the `pgo_apiserver_u`
pgo_client_cert_secret	pgo.tls		Sets the secret that the pg
primary_storage	storageos	**Required**	Set to configure which sto
prometheus_install	true		Set to true to install Crun
prometheus_storage_access_mode			Set to the access mode us

Name	Default	Required	Description
prometheus_storage_class_name			Set to the name of the sto
replica_storage	storageos	**Required**	Set to configure which sto
scheduler_timeout	3600	**Required**	Set to a value in seconds t
service_type	ClusterIP		Set to configure the type c
pgo_cluster_admin	false		Determines whether or no

{{% 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.

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_rmdata_resources='small'
default_backup_resources='small'
default_pgbouncer_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 "Configuration/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 C
`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
```

PGO-Client Container

The following will setup the `pgo` client image in a Kubernetes or Openshift environment. The image must be installed using the Ansible installer.

Installing the PGO-Client Container

The pgo-client container can be installed with the Ansible installer by updating the `pgo_client_container_install` variable in the inventory file. Set this variable to true in the inventory file and run the ansible-playbook. As part of the install the `pgo.tls` and `pgouser-<username>` secrets are used to configure the `pgo` client.

Using the PGO-Client Deployment

Once the container has been installed you can access it by exec'ing into the pod. You can run single commands with the kubectl or oc command line tools or multiple commands by exec'ing into the pod with bash.

```
kubectl exec -it -n pgo <pgo-client-deployment-name> -c "pgo version"

# or

kubectl exec -it -n pgo <pgo-client-deployment-name> bash
```

The deployment does not require any configuration to connect to the operator.

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.

The PostgreSQL Operator Client, aka `pgo`, is the most convenient way to interact with the PostgreSQL Operator. `pgo` provides many convenience methods for creating, managing, and deleting PostgreSQL clusters through a series of simple commands. The `pgo` client interfaces with the API that is provided by the PostgreSQL Operator and can leverage the RBAC and TLS systems that are provided by the PostgreSQL Operator

The `pgo` client is available for Linux, macOS, and Windows, as well as a `pgo-client` container that can be deployed alongside the PostgreSQL Operator.

You can download `pgo` from the releases page, or have it installed in your preferred binary format or as a container in your Kubernetes cluster using the Ansible Installer.

General Notes on Using the `pgo` Client

Many of the `pgo` client commands require you to specify a namespace via the `-n` or `--namespace` flag. While this is a very helpful tool when managing PostgreSQL deployments across many Kubernetes namespaces, this can become onerous for the intents of this guide.

If you install the PostgreSQL Operator using the quickstart guide, you will have two namespaces installed: `pgouser1` and `pgouser2`. We can choose to always use one of these namespaces by setting the `PGO_NAMESPACE` environmental variable, which is detailed in the global pgo Client reference,

For convenience, we will use the `pgouser1` namespace in the examples below. For even more convenience, we recommend setting `pgouser1` to be the value of the `PGO_NAMESPACE` variable. In the shell that you will be executing the `pgo` commands in, run the following command:

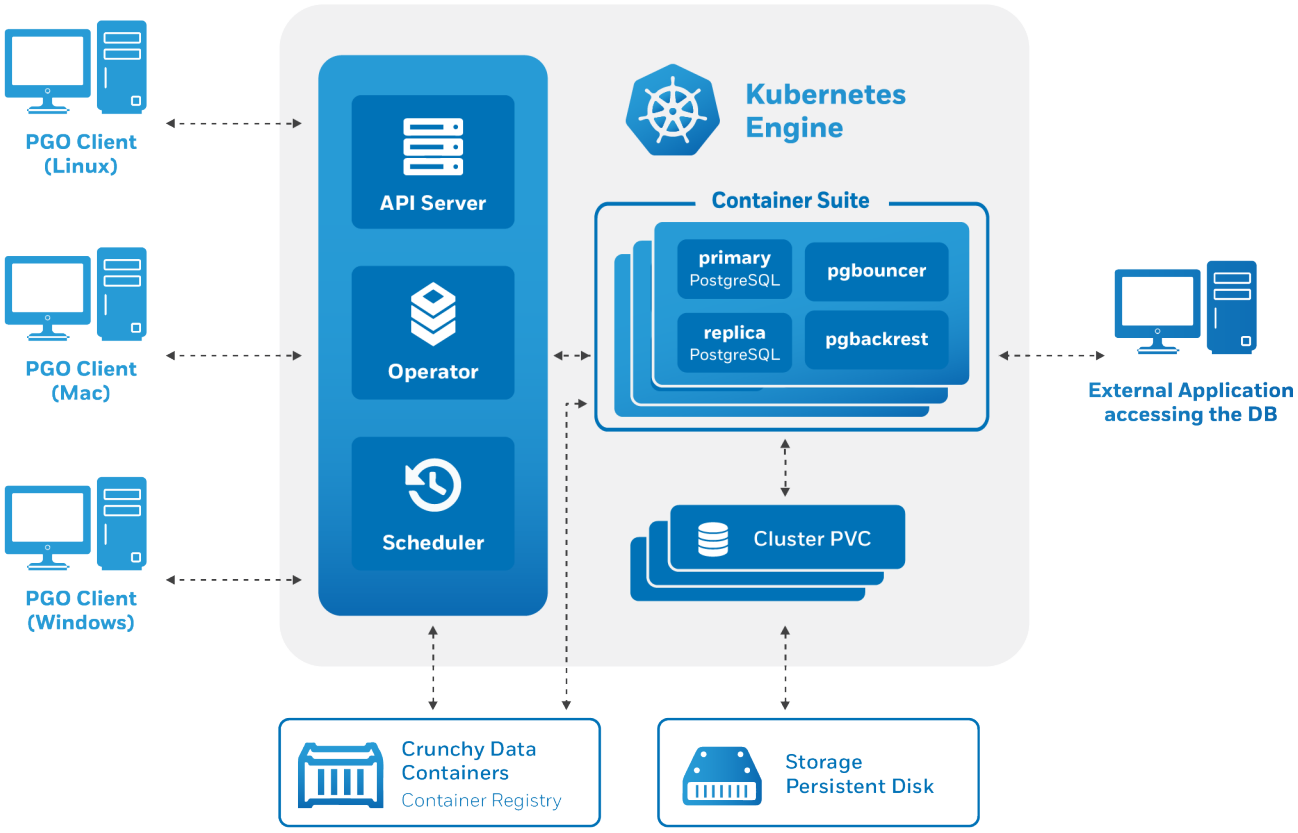

Figure 14: Architecture

```
export PGO_NAMESPACE=pgouser1
```

If you do not wish to set this environmental variable, or are in an environment where you are unable to use environmental variables, you will have to use the `--namespace` (or `-n`) flag for most commands, e.g.

```
pgo version -n pgouser1
```

Syntax

The syntax for `pgo` is similar to what you would expect from using the `kubectl` or `oc` binaries. This is by design: one of the goals of the PostgreSQL Operator project is to allow for seamless management of PostgreSQL clusters in Kubernetes-enabled environments, and by following the command patterns that users are familiar with, the learning curve is that much easier!

To get an overview of everything that is available at the top-level of `pgo`, execute:

```
pgo
```

The syntax for the commands that `pgo` executes typicall follow this format:

```
pgo [command] ([TYPE] [NAME]) [flags]
```

Where *command* is a verb like:

- `create`
- `show`
- `delete`

And *type* is a resource type like:

- `cluster`
- `backup`

- `user`

And *name* is the name of the resource type like:

- hacluster
- gisdba

There are several global flags that are available to every `pgo` command as well as flags that are specific to particular commands. To get a list of all the options and flags available to a command, you can use the `--help` flag. For example, to see all of the options available to the `pgo create cluster` command, you can run the following:

```
pgo create cluster --help
```

Command Overview

The following table provides an overview of the commands that the `pgo` client provides:

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)
cat	`pgo cat mycluster filepath`	Perform a Linux `cat` command on the cluster.
clone	`pgo clone oldcluster newcluster`	Copies the primary database of an existing cluster to
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.
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.

Global Flags

There are several global flags available to the `pgo` client.

NOTE: Flags take precedence over environmental variables.

Flag	Description
`--apiserver-url`	The URL for the PostgreSQL Operator apiserver that will process the request from the pgo client.
`--debug`	Enable additional output for debugging.
`--disable-tls`	Disable TLS authentication to the Postgres Operator.

Flag	Description
`--exclude-os-trust`	Exclude CA certs from OS default trust store.
`-h ,--help`	Print out help for a command command.
`-n ,--namespace`	The namespace to execute the `pgo` command in. This is required for most `pgo` commands.
`--pgo-ca-cert`	The CA certificate file path for authenticating to the PostgreSQL Operator apiserver.
`--pgo-client-cert`	The client certificate file path for authenticating to the PostgreSQL Operator apiserver.
`--pgo-client-key`	The client key file path for authenticating to the PostgreSQL Operator apiserver.

Global Environment Variables

There are several environmental variables that can be used with the `pgo` client.

NOTE Flags take precedence over environmental variables.

Name	Description
`EXCLUDE_OS_TRUST`	Exclude CA certs from OS default trust store.
`GENERATE_BASH_COMPLETION`	If set, will allow `pgo` to leverage "bash completion" to help complete commands as they are typed.
`PGO_APISERVER_URL`	The URL for the PostgreSQL Operator apiserver that will process the request from the pgo client.
`PGO_CA_CERT`	The CA certificate file path for authenticating to the PostgreSQL Operator apiserver.
`PGO_CLIENT_CERT`	The client certificate file path for authenticating to the PostgreSQL Operator apiserver.
`PGO_CLIENT_KEY`	The client key file path for authenticating to the PostgreSQL Operator apiserver.
`PGO_NAMESPACE`	The namespace to execute the `pgo` command in. This is required for most `pgo` commands.
`PGOUSER`	The path to the pgouser file. Will be ignored if either `PGOUSERNAME` or `PGOUSERPASS` are set.
`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.

Additional Information

How can you use the `pgo` client to manage your day-to-day PostgreSQL operations? The next section covers many of the common types of tasks that one needs to perform when managing production PostgreSQL clusters. Beyond that is the full reference for all the available commands and flags for the `pgo` client.

While the full pgo client reference will tell you everything you need to know about how to use `pgo`, it may be helpful to see several examples on how to conduct "day-in-the-life" tasks for administrating PostgreSQL cluster with the PostgreSQL Operator.

The below guide covers many of the common operations that are required when managing PostgreSQL clusters. The guide is broken up by different administrative topics, such as provisioning, high-availability, etc.

Setup Before Running the Examples

Many of the `pgo` client commands require you to specify a namespace via the `-n` or `--namespace` flag. While this is a very helpful tool when managing PostgreSQL deployxments across many Kubernetes namespaces, this can become onerous for the intents of this guide.

If you install the PostgreSQL Operator using the quickstart guide, you will have two namespaces installed: `pgouser1` and `pgouser2`. We can choose to always use one of these namespaces by setting the `PGO_NAMESPACE` environmental variable, which is detailed in the global pgo Client reference,

For convenience, we will use the `pgouser1` namespace in the examples below. For even more convenience, we recommend setting `pgouser1` to be the value of the `PGO_NAMESPACE` variable. In the shell that you will be executing the `pgo` commands in, run the following command:

```
export PGO_NAMESPACE=pgouser1
```

If you do not wish to set this environmental variable, or are in an environment where you are unable to use environmental variables, you will have to use the `--namespace` (or `-n`) flag for most commands, e.g.

```
pgo version -n pgouser1
```

JSON Output

The default for the `pgo` client commands is to output their results in a readable format. However, there are times where it may be helpful to you to have the format output in a machine parseable format like JSON.

Several commands support the `-o`/`--output` flags that delivers the results of the command in the specified output. Presently, the only output that is supported is `json`.

As an example of using this feature, if you wanted to get the results of the `pgo test` command in JSON, you could run the following:

```
pgo test hacluster -o json
```

PostgreSQL Operator System Basics

To get started, it's first important to understand the basics of working with the PostgreSQL Operator itself. You should know how to test if the PostgreSQL Operator is working, check the overall status of the PostgreSQL Operator, view the current configuration that the PostgreSQL Operator us using, and seeing which Kubernetes Namespaces the PostgreSQL Operator has access to.

While this may not be as fun as creating high-availability PostgreSQL clusters, these commands will help you to perform basic troubleshooting tasks in your environment.

Checking Connectivity to the PostgreSQL Operator

A common task when working with the PostgreSQL Operator is to check connectivity to the PostgreSQL Operator. This can be accomplish with the `pgo version` command:

```
pgo version
```

which, if working, will yield results similar to:

```
pgo client version 4.3.0
pgo-apiserver version 4.3.0
```

Inspecting the PostgreSQL Operator Configuration

The `pgo show config` command allows you to view the current configuration that the PostgreSQL Operator is using. This can be helpful for troubleshooting issues such as which PostgreSQL images are being deployed by default, which storage classes are being used, etc.

You can run the `pgo show config` command by running:

```
pgo show config
```

which yields output similar to:

```
BasicAuth: ""
Cluster:
  CCPImagePrefix: crunchydata
  CCPImageTag: centos7-12.2-4.3.0
  PrimaryNodeLabel: ""
  ReplicaNodeLabel: ""
  Policies: ""
  LogStatement: none
  LogMinDurationStatement: "60000"
  Metrics: false
  Badger: false
  Port: "5432"
  PGBadgerPort: "10000"
  ExporterPort: "9187"
  User: testuser
  ArchiveTimeout: "60"
  Database: userdb
```

```
  PasswordAgeDays: "60"
  PasswordLength: "8"
  Strategy: "1"
  Replicas: "0"
  ServiceType: ClusterIP
  BackrestPort: 2022
  Backrest: true
  BackrestS3Bucket: ""
  BackrestS3Endpoint: ""
  BackrestS3Region: ""
  DisableAutofail: false
  PgmonitorPassword: ""
  EnableCrunchyadm: false
  DisableReplicaStartFailReinit: false
  PodAntiAffinity: preferred
  SyncReplication: false
Pgo:
  PreferredFailoverNode: ""
  Audit: false
  PGOImagePrefix: crunchydata
  PGOImageTag: centos7-4.3.0
ContainerResources:
  large:
    RequestsMemory: 2Gi
    RequestsCPU: "2.0"
    LimitsMemory: 2Gi
    LimitsCPU: "4.0"
  small:
    RequestsMemory: 256Mi
    RequestsCPU: "0.1"
    LimitsMemory: 256Mi
    LimitsCPU: "0.1"
PrimaryStorage: nfsstorage
BackupStorage: nfsstorage
ReplicaStorage: nfsstorage
BackrestStorage: nfsstorage
Storage:
  nfsstorage:
    AccessMode: ReadWriteMany
    Size: 1G
    StorageType: create
    StorageClass: ""
    Fsgroup: ""
    SupplementalGroups: "65534"
    MatchLabels: ""
DefaultContainerResources: ""
DefaultLoadResources: ""
DefaultRmdataResources: ""
DefaultBackupResources: ""
DefaultBadgerResources: ""
DefaultPgbouncerResources: ""
```

Viewing PostgreSQL Operator Key Metrics

The `pgo status` command provides a generalized statistical view of the overall resource consumption of the PostgreSQL Operator. These stats include:

- The total number of PostgreSQL instances
- The total number of Persistent Volume Claims (PVC) that are allocated, along with the total amount of disk the claims specify
- The types of container images that are deployed, along with how many are deployed
- The nodes that are used by the PostgreSQL Operator

and more

You can use the `pgo status` command by running:

```
pgo status
```

which yields output similar to:

```
Operator Start:          2019-12-26 17:53:45 +0000 UTC
Databases:               8
Claims:                  8
Total Volume Size:       8Gi

Database Images:
                         4  crunchydata/crunchy-postgres-ha:centos7-12.2-4.3.0
                         4  crunchydata/pgo-backrest-repo:centos7-4.3.0
                         8  crunchydata/pgo-backrest:centos7-4.3.0

Databases Not Ready:

Nodes:
    master
        Status:Ready
        Labels:
            beta.kubernetes.io/arch=amd64
            beta.kubernetes.io/os=linux
            kubernetes.io/arch=amd64
            kubernetes.io/hostname=master
            kubernetes.io/os=linux
            node-role.kubernetes.io/master=
    node01
        Status:Ready
        Labels:
            beta.kubernetes.io/arch=amd64
            beta.kubernetes.io/os=linux
            kubernetes.io/arch=amd64
            kubernetes.io/hostname=node01
            kubernetes.io/os=linux

Labels (count > 1): [count] [label]
    [8] [vendor=crunchydata]
    [4] [pgo-backrest-repo=true]
    [4] [pgouser=pgoadmin]
    [4] [pgo-pg-database=true]
    [4] [crunchy_collect=false]
    [4] [pg-pod-anti-affinity=]
    [4] [pgo-version=4.3.0]
    [4] [archive-timeout=60]
    [2] [pg-cluster=hacluster]
```

Viewing PostgreSQL Operator Managed Namespaces

The PostgreSQL Operator has the ability to manage PostgreSQL clusters across Kubernetes Namespaces. During the course of Operations, it can be helpful to know which namespaces the PostgreSQL Operator can use for deploying PostgreSQL clusters.

You can view which namespaces the PostgreSQL Operator can utilize by using the `pgo show namespace` command. To list out the namespaces that the PostgreSQL Operator has access to, you can run the following command:

```
pgo show namespace --all
```

which yields output similar to:

```
pgo username: pgoadmin
namespace                useraccess          installaccess
default                  accessible          no access
kube-node-lease          accessible          no access
kube-public              accessible          no access
kube-system              accessible          no access
```

```
pgo                     accessible          no access
pgouser1                accessible          accessible
pgouser2                accessible          accessible
somethingelse           no access           no access
```

NOTE: Based on your deployment, your Kubernetes administrator may restrict access to the multi-namespace feature of the PostgreSQL Operator. In this case, you do not need to worry about managing your namespaces and as such do not need to use this command, but we recommend setting the `PGO_NAMESPACE` variable as described in the general notes on this page.

Provisioning: Create, View, Destroy

Creating a PostgreSQL Cluster

You can create a cluster using the `pgo create cluster` command:

```
pgo create cluster hacluster
```

which if successfully, will yield output similar to this:

```
created Pgcluster hacluster
workflow id ae714d12-f5d0-4fa9-910f-21944b41dec8
```

Create a PostgreSQL Cluster with PostGIS To create a PostgreSQL cluster that uses the geospatial extension PostGIS, you can execute the following command:

```
pgo create cluster hagiscluster --ccp-image=crunchy-postgres-gis-ha
```

Create a PostgreSQL Cluster with a Tablespace Tablespaces are a PostgreSQL feature that allows a user to select specific volumes to store data to, which is helpful in several types of scenarios. Often your workload does not require a tablespace, but the PostgreSQL Operator provides support for tablespaces throughout the lifecycle of a PostgreSQL cluster.

To create a PostgreSQL cluster that uses the tablespace feature with NFS storage, you can execute the following command:

```
pgo create cluster hactsluster --tablespaces=ts1=nfsstorage
```

You can use your preferred storage engine instead of `nfsstorage`. For example, to create multiple tablespaces on GKE, you can execute the following command:

```
pgo create cluster hactsluster --tablespaces=ts1=gce,ts2=gce
```

Tablespaces are immediately available once the PostgreSQL cluster is provisioned. For example, to create a table using the tablespace.

Tracking a Newly Provisioned Cluster A new PostgreSQL cluster can take a few moments to provision. You may have noticed that the `pgo create cluster` command returns something called a "workflow id". This workflow ID allows you to track the progress of your new PostgreSQL cluster while it is being provisioned using the `pgo show workflow` command:

```
pgo show workflow ae714d12-f5d0-4fa9-910f-21944b41dec8
```

which can yield output similar to:

```
parameter           value
---------           -----
pg-cluster          hacluster
task completed      2019-12-27T02:10:14Z
task submitted      2019-12-27T02:09:46Z
workflowid          ae714d12-f5d0-4fa9-910f-21944b41dec8
```

View PostgreSQL Cluster Details

To see details about your PostgreSQL cluster, you can use the `pgo show cluster` command. These details include elements such as:

- The version of PostgreSQL that the cluster is using
- The PostgreSQL instances that comprise the cluster
- The Pods assigned to the cluster for all of the associated components, including the nodes that the pods are assigned to

- The Persistent Volume Claims (PVC) that are being consumed by the cluster
- The Kubernetes Deployments associated with the cluster
- The Kubernetes Services associated with the cluster
- The Kubernetes Labels that are assigned to the PostgreSQL instances

and more.

You can view the details of the cluster by executing the following command:

```
pgo show cluster hacluster
```

which will yield output similar to:

```
cluster : hacluster (crunchy-postgres-ha:centos7-12.2-4.3.0)
    pod : hacluster-6dc6cfcfb9-f9knq (Running) on node01 (1/1) (primary)
    pvc : hacluster
    resources : CPU Limit= Memory Limit=, CPU Request= Memory Request=
    storage : Primary=200M Replica=200M
    deployment : hacluster
    deployment : hacluster-backrest-shared-repo
    service : hacluster - ClusterIP (10.102.20.42)
    labels : pg-pod-anti-affinity= archive-timeout=60 crunchy-pgbadger=false crunchy_collect=false
       deployment-name=hacluster pg-cluster=hacluster crunchy-pgha-scope=hacluster autofail=true
       pgo-backrest=true pgo-version=4.3.0 current-primary=hacluster name=hacluster
       pgouser=pgoadmin workflowid=ae714d12-f5d0-4fa9-910f-21944b41dec8
```

Deleting a Cluster

You can delete a PostgreSQL cluster that is managed by the PostgreSQL Operator by executing the following command:

```
pgo delete cluster hacluster
```

This will remove the cluster from being managed by the PostgreSQL Operator, as well as delete the root data Persistent Volume Claim (PVC) and backup PVCs associated with the cluster.

If you wish to keep your PostgreSQL data PVC, you can delete the cluster with the following command:

```
pgo delete cluster hacluster --keep-data
```

You can then recreate the PostgreSQL cluster with the same data by using the `pgo create cluster` command with a cluster of the same name:

```
pgo create cluster hacluster
```

This technique is used when performing tasks such as upgrading the PostgreSQL Operator.

You can also keep the pgBackRest repository associated with the PostgreSQL cluster by using the `--keep-backups` flag with the `pgo delete cluster` command:

```
pgo delete cluster hacluster --keep-backups
```

Testing PostgreSQL Cluster Availability

You can test the availability of your cluster by using the `pgo test` command. The `pgo test` command checks to see if the Kubernetes Services and the Pods that comprise the PostgreSQL cluster are available to receive connections. This includes:

- Testing that the Kubernetes Endpoints are available and able to route requests to healthy Pods
- Testing that each PostgreSQL instance is available and ready to accept client connections by performing a connectivity check similar to the one performed by `pg_isready`

To test the availability of a PostgreSQL cluster, you can run the following command:

```
pgo test hacluster
```

which will yield output similar to:

```
cluster : hacluster
    Services
        primary (10.102.20.42:5432): UP
    Instances
        primary (hacluster-6dc6cfcfb9-f9knq): UP
```

Disaster Recovery: Backups & Restores

The PostgreSQL Operator supports sophisticated functionality for managing your backups and restores. For more information for how this works, please see the disaster recovery guide.

Creating a Backup

The PostgreSQL Operator uses the open source pgBackRest backup and recovery utility for managing backups and PostgreSQL archives. These backups are also used as part of managing the overall health and high-availability of PostgreSQL clusters managed by the PostgreSQL Operator and used as part of the cloning process as well.

When a new PostgreSQL cluster is provisioned by the PostgreSQL Operator, a full pgBackRest backup is taken by default. This is required in order to create new replicas (via `pgo scale`) for the PostgreSQL cluster as well as healing during a failover scenario.

To create a backup, you can run the following command:

```
pgo backup hacluster
```

which by default, will create an incremental pgBackRest backup. The reason for this is that the PostgreSQL Operator initially creates a pgBackRest full backup when the cluster is initial provisioned, and pgBackRest will take incremental backups for each subsequent backup until a different backup type is specified.

Most pgBackRest options are supported and can be passed in by the PostgreSQL Operator via the `--backup-opts` flag. What follows are some examples for how to utilize pgBackRest with the PostgreSQL Operator to help you create your optimal disaster recovery setup.

Creating a Full Backup You can create a full backup using the following command:

```
pgo backup hacluster --backup-opts="--type=full"
```

Creating a Differential Backup You can create a differential backup using the following command:

```
pgo backup hacluster --backup-opts="--type=diff"
```

Creating an Incremental Backup You can create a differential backup using the following command:

```
pgo backup hacluster --backup-opts="--type=incr"
```

An incremental backup is created without specifying any options after a full or differential backup is taken.

Creating Backups in S3

The PostgreSQL Operator supports creating backups in S3 or any object storage system that uses the S3 protocol. For more information, please read the section on PostgreSQL Operator Backups with S3 in the architecture section.

Displaying Backup Information

You can see information about the current state of backups in a PostgreSQL cluster managed by the PostgreSQL Operator by executing the following command:

```
pgo show backup hacluster
```

Setting Backup Retention

By default, pgBackRest will allow you to keep on creating backups until you run out of disk space. As such, it may be helpful to manage how many backups are retained.

pgBackRest comes with several flags for managing how backups can be retained:

- `--repo1-retention-full`: how many full backups to retain
- `--repo1-retention-diff`: how many differential backups to retain
- `--repo1-retention-archive`: how many sets of WAL archives to retain alongside the full and differential backups that are retained

For example, to create a full backup and retain the previous 7 full backups, you would execute the following command:

```
pgo backup hacluster --backup-opts="--type=full --repo1-retention-full=7"
```

Scheduling Backups

Any effective disaster recovery strategy includes having regularly scheduled backups. The PostgreSQL Operator enables this through its scheduling sidecar that is deployed alongside the Operator.

Creating a Scheduled Backup For example, to schedule a full backup once a day at midnight, you can execute the following command:

```
pgo create schedule hacluster --schedule="0 1 * * *" \
  --schedule-type=pgbackrest  --pgbackrest-backup-type=full
```

To schedule an incremental backup once every 3 hours, you can execute the following command:

```
pgo create schedule hacluster --schedule="0 */3 * * *" \
  --schedule-type=pgbackrest  --pgbackrest-backup-type=incr
```

You can also create regularly scheduled backups and combine it with a retention policy. For example, using the above example of taking a nightly full backup, you can specify a policy of retaining 21 backups by executing the following command:

```
pgo create schedule hacluster --schedule="0 0 * * *" \
  --schedule-type=pgbackrest  --pgbackrest-backup-type=full \
  --schedule-opts="--repo1-retention-full=21"
```

Restore a Cluster

The PostgreSQL Operator supports the ability to perform a full restore on a PostgreSQL cluster as well as a point-in-time-recovery using the `pgo restore` command. Note that both of these options are **destructive** to the existing PostgreSQL cluster; to "restore" the PostgreSQL cluster to a new deployment, please see the clone section.

After a restore, there are some cleanup steps you will need to perform. Please review the Post Restore Cleanup section.

Full Restore To perform a full restore of a PostgreSQL cluster, you can execute the following command:

```
pgo restore hacluster
```

If you want your PostgreSQL cluster to be restored to a specific node, you can execute the following command:

```
pgo restore hacluster --node-label=failure-domain.beta.kubernetes.io/zone=us-central1-a
```

There are very few reasons why you will want to execute a full restore. If you want to make a copy of your PostgreSQL cluster, please use `pgo clone`.

Point-in-time-Recovery (PITR) The more likely scenario when performing a PostgreSQL cluster restore is to recover to a particular point-in-time (e.g. before a key table was dropped). For example, to restore a cluster to December 23, 2019 at 8:00am:

```
pgo restore hacluster --pitr-target="2019-12-23 08:00:00.000000+00" \
  --backup-opts="--type=time"
```

The PostgreSQL Operator supports the full set of pgBackRest restore options, which can be passed into the `--backup-opts` parameter. For more information, please review the pgBackRest restore options

Post Restore Cleanup After a restore is complete, you will need to re-enable high-availability on a PostgreSQL cluster manually. You can re-enable high-availability by executing the following command:

```
pgo update cluster hacluster --autofail=true
```

Logical Backups (`pg_dump` / `pg_dumpall`)

The PostgreSQL Operator supports taking logical backups with `pg_dump` and `pg_dumpall`. While they do not provide the same performance and storage optimizations as the physical backups provided by pgBackRest, logical backups are helpful when one wants to upgrade between major PostgreSQL versions, or provide only a subset of a database, such as a table.

Create a Logical Backup To create a logical backup of a full database, you can run the following command:

```
pgo backup hacluster --backup-type=pgdump
```

You can pass in specific options to `--backup-opts`, which can accept most of the options that the pg_dump command accepts. For example, to only dump the data from a specific table called `users`:

```
pgo backup hacluster --backup-type=pgdump --backup-opts="-t users"
```

To use `pg_dumpall` to create a logical backup of all the data in a PostgreSQL cluster, you must pass the `--dump-all` flag in `--backup-opts`, i.e.:

```
pgo backup hacluster --backup-type=pgdump --backup-opts="--dump-all"
```

Viewing Logical Backups To view an available list of logical backups, you can use the `pgo show backup` command:

```
pgo show backup --backup-type=pgdump
```

This provides information about the PVC that the logical backups are stored on as well as the timestamps required to perform a restore from a logical backup.

Restore from a Logical Backup To restore from a logical backup, you need to reference the PVC that the logical backup is stored to, as well as the timestamp that was created by the logical backup.

You can restore a logical backup using the following command:

```
pgo restore hacluster --backup-type=pgdump --backup-pvc=hacluster-pgdump-pvc \
  --pitr-target="2019-01-15-00-03-25" -n pgouser1
```

High-Availability: Scaling Up & Down

The PostgreSQL Operator supports a robust high-availability set up to ensure that your PostgreSQL clusters can stay up and running. For detailed information on how it works, please see the high-availability architecture section.

Creating a New Replica

To create a new replica, also known as "scaling up", you can execute the following command:

```
pgo scale hacluster --replica-count=1
```

If you wanted to add two new replicas at the same time, you could execute the following command:

```
pgo scale hacluster --replica-count=2
```

Viewing Available Replicas

You can view the available replicas in a few ways. First, you can use `pgo show cluster` to see the overall information about the PostgreSQL cluster:

```
pgo show cluster hacluster
```

You can also find specific replica names by using the `--query` flag on the `pgo failover` and `pgo scaledown` commands, e.g.:

```
pgo failover --query hacluster
```

Manual Failover

The PostgreSQL Operator is set up with an automated failover system based on distributed consensus, but there may be times where you wish to have your cluster manually failover. If you wish to have your cluster manually failover, first, query your cluster to determine which failover targets are available. The query command also provides information that may help your decision, such as replication lag:

```
pgo failover --query hacluster
```

Once you have selected the replica that is best for your to failover to, you can perform a failover with the following command:

```
pgo failover hacluster --target=hacluster-abcd
```

where `hacluster-abcd` is the name of the PostgreSQL instance that you want to promote to become the new primary

Destroying a Replica To destroy a replica, first query the available replicas by using the `--query` flag on the `pgo scaledown` command, i.e.:

```
pgo scaledown hacluster --query
```

Once you have picked the replica you want to remove, you can remove it by executing the following command:

```
pgo scaledown hacluster --target=hacluster-abcd
```

where `hacluster-abcd` is the name of the PostgreSQL replica that you want to destroy.

Clone a PostgreSQL Cluster

You can create a copy of an existing PostgreSQL cluster in a new PostgreSQL cluster by using the `pgo clone` command. To create a new copy of a PostgreSQL cluster, you can execute the following command:

```
pgo clone hacluster newhacluster
```

Monitoring

View Disk Utilization

You can see a comparison of Postgres data size versus the Persistent volume claim size by entering the following:

```
pgo df hacluster -n pgouser1
```

Labels

Labels are a helpful way to organize PostgreSQL clusters, such as by application type or environment. The PostgreSQL Operator supports managing Kubernetes Labels as a convenient way to group PostgreSQL clusters together.

You can view which labels are assigned to a PostgreSQL cluster using the `pgo show cluster` command. You are also able to see these labels when using `kubectl` or `oc`.

Add a Label to a PostgreSQL Cluster

Labels can be added to PostgreSQL clusters using the `pgo label` command. For example, to add a label with a key/value pair of `env=production`, you could execute the following command:

```
pgo label hacluster --label=env=production
```

Add a Label to Multiple PostgreSQL Clusters

You can add also add a label to multiple PostgreSQL clusters simultaneously using the `--selector` flag on the `pgo label` command. For example, to add a label with a key/value pair of `env=production` to clusters that have a label key/value pair of `app=payment`, you could execute the following command:

```
pgo label --selector=app=payment --label=env=production
```

Policy Management

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
```

Apply a Policy

```
pgo apply mypolicy --selector=environment=prod
pgo apply mypolicy --selector=name=hacluster
```

Advanced Operations

Connection Pooling via pgBouncer

To add a pgbouncer Deployment to your Postgres cluster, enter:

```
pgo create cluster hacluster --pgbouncer -n pgouser1
```

You can add pgbouncer after a Postgres cluster is created as follows:

```
pgo create pgbouncer hacluster
pgo create pgbouncer --selector=name=hacluster
```

You can also specify a pgbouncer password as follows:

```
pgo create cluster hacluster --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 hacluster -n pgouser1
```

You can create a pgbadger sidecar container in your Postgres cluster pod as follows:

```
pgo create cluster hacluster --pgbadger -n pgouser1
```

Likewise, you can add the Crunchy Collect Metrics sidecar container into your Postgres cluster pod as follows:

```
pgo create cluster hacluster --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.

Create a Cluster using Specific Storage

```
pgo create cluster hacluster --storage-config=somestorageconfig -n pgouser1
```

Likewise, you can specify a storage configuration when creating a replica:

```
pgo scale hacluster --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 hacluster --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 hacluster --node-label=speed=slowerthannormal -n pgouser1
```

Create a Cluster with LoadBalancer ServiceType

```
pgo create cluster hacluster --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
```

PostgreSQL Operator 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"
```

PostgreSQL Cluster User Operations

Managed Postgres users can be viewed using the following command:

```
pgo show user hacluster
```

Postgres users can be created using the following command examples:

```
pgo create user hacluster --username=somepguser --password=somepassword --managed
pgo create user --selector=name=hacluster --username=somepguser --password=somepassword --managed
```

Those commands are identical in function, and create on the hacluster 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 Kubernetes cluster in the Operator namespace.

Postgres users can be deleted using the following command:

```
pgo delete user hacluster --username=somepguser
```

That command deletes the user on the hacluster Postgres cluster.

Postgres users can be updated using the following command:

```
pgo update user hacluster --username=somepguser --password=frodo
```

That command changes the password for the user on the hacluster Postgres cluster.

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]
```

If you wish to add all avaiable permissions to a *pgorole*, you can specify it by using a single `*` in your configuration. Note that if you are editing your YAML file directly, you will need to ensure to write it as `"*"` to ensure it is recognized as a string.

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*
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*
DeletePolicy	allow *pgo delete policy*
DeleteSchedule	allow *pgo delete schedule*
DeleteUpgrade	allow *pgo delete upgrade*
DeleteUser	allow *pgo delete user*
DfCluster	allow *pgo df*

Permission	Description
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: 403
```

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

Setting	Definition
	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-$PGO_OPERATOR_NAMESPACE$and$pgopclusterbindin$

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.

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.

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 (12.2)
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":"4.3.0",
  "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":"4.3.0",
  "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":"4.3.0",
  "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 provisinoning 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.

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.X to 4.3.0 (Bash)] ({{< relref "upgrade/upgrade4xto42_bash.md" >}})

[Upgrade Postgres Operator from 4.X to 4.3.0 (Ansible)] ({{< relref "upgrade/upgrade4xto42_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, and if *autofail* is enabled for the cluster, 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. Please note that the upgrade process respects the *autofail* setting as currently definied for the cluster being upgraded. Therefore, if autofail is enabled when 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 about the PostgreSQL Operator failover process and expected behavior.

When the cluster is not in *autofail* mode (i.e. autofail is disabled), the primary and all replicas are updated at the same time, after which they will remain in an "unready" status. This is because when autofail is disabled, no attempt will be made to start the PostgreSQL databases contained within the primary or replica pods once the containers have been started following the update. It is therefore necessary to re-enable autofail following a minor upgrade during which autofail was disabled in order to fully bring the cluster back online.

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 11.6 to 11.7 using the `crunchy-postgres:centos7-11.7-4.3.0` container:

```
pgo upgrade mycluster --ccp-image-tag=centos7-11.7-4.3.0
```

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

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 4.3.0

This section will outline the procedure to upgrade a given cluster created using Postgres Operator 3.5.x to PGO version 4.3.0. This version of the Postgres Operator has several fundamental changes to the existing PGCluster structure and deployment model. Most notably, all PGClusters use the new Crunchy Postgres HA container in place of the previous Crunchy Postgres containers. The use of this new container is a breaking change from previous versions of the Operator.

Crunchy Postgres High Availability Containers Using the PostgreSQL Operator 4.3.0 requires replacing your `crunchy-postgres` and `crunchy-postgres-gis` containers with the `crunchy-postgres-ha` and `crunchy-postgres-gis-ha` containers respectively. The underlying PostgreSQL installations in the container remain the same but are now optimized for Kubernetes environments to provide the new high-availability functionality.

A major change to this container is that the PostgreSQL process is now managed by Patroni. This allows a PostgreSQL cluster that is deployed by the PostgreSQL Operator to manage its own uptime and availability, to elect a new leader in the event of a downtime scenario, and to automatically heal after a failover event.

When creating your new clusters using version 4.3.0 of the Postgres Operator, the `pgo create cluster` command will automatically use the new `crunchy-postgres-ha` image if the image is unspecified. If you are creating a PostGIS enabled cluster, please be sure to use the updated image name, as with the command:

```
pgo create cluster mygiscluster --ccp-image=crunchy-postgres-gis-ha
```

{{% 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 4.3.0 code for the Postgres Operator available
- The latest PGO 4.3.0 client binary

Step 0 Create a new Linux 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 4.3.0.

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 4.3.0 Postgres Operator.

[Bash Installation] ({{< relref "installation/operator-install.md" >}})

Be sure to add the existing namespace to the Operator's list of watched namespaces (see the [Namespace] ({{< relref "architecture/-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 4.3.0 cluster with the same name and using the same major PostgreSQL version 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 4.3.0:

```
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 <clustername> -n <namespace>
```

Output should be similar to:

```
cluster : mycluster
        Services
                primary (10.106.70.238:5432): UP
        Instances
                primary (mycluster-7d49d98665-7zxzd): UP
```

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

It is also recommended to take full backups of each pgcluster once the upgrade is completed due to version differences between the old and new clusters.

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.

Postgres Operator Ansible Upgrade Procedure from 4.X to 4.3.0

This procedure will give instructions on how to upgrade to version 4.3.0 of the Crunchy Postgres Operator using the Ansible installation method. This version of the Postgres Operator has several fundamental changes to the existing PGCluster structure and deployment model. Most notably, all PGClusters use the new Crunchy Postgres HA container in place of the previous Crunchy Postgres containers. The use of this new container is a breaking change from previous versions of the Operator.

Crunchy Postgres High Availability Containers Using the PostgreSQL Operator 4.3.0 requires replacing your `crunchy-postgres` and `crunchy-postgres-gis` containers with the `crunchy-postgres-ha` and `crunchy-postgres-gis-ha` containers respectively. The underlying PostgreSQL installations in the container remain the same but are now optimized for Kubernetes environments to provide the new high-availability functionality.

A major change to this container is that the PostgreSQL process is now managed by Patroni. This allows a PostgreSQL cluster that is deployed by the PostgreSQL Operator to manage its own uptime and availability, to elect a new leader in the event of a downtime scenario, and to automatically heal after a failover event.

When creating your new clusters using version 4.3.0 of the Postgres Operator, the `pgo create cluster` command will automatically use the new `crunchy-postgres-ha` image if the image is unspecified. If you are creating a PostGIS enabled cluster, please be sure to use the updated image name, as with the command:

```
pgo create cluster mygiscluster --ccp-image=crunchy-postgres-gis-ha
```

{{% 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 4.3.0 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 4.3.0 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 a new 4.3.0 cluster with the same name and using the same major PostgreSQL version 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 <clustername> -n <namespace>
```

Output should be similar to:

```
cluster : mycluster
        Services
                primary (10.106.70.238:5432): UP
        Instances
                primary (mycluster-7d49d98665-7zxzd): UP
```

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

It is also recommended to take full backups of each pgcluster once the upgrade is completed due to version differences between the old and new clusters.

Postgres Operator Bash Upgrade Procedure from 4.X to 4.3.0

This procedure will give instructions on how to upgrade to version 4.3.0 of the Crunchy Postgres Operator using the Bash installation method. This version of the Postgres Operator has several fundamental changes to the existing PGCluster structure and deployment model. Most notably, all PGClusters use the new Crunchy Postgres HA container in place of the previous Crunchy Postgres containers. The use of this new container is a breaking change from previous versions of the Operator.

Crunchy Postgres High Availability Containers Using the PostgreSQL Operator 4.3.0 requires replacing your `crunchy-postgres` and `crunchy-postgres-gis` containers with the `crunchy-postgres-ha` and `crunchy-postgres-gis-ha` containers respectively. The underlying PostgreSQL installations in the container remain the same but are now optimized for Kubernetes environments to provide the new high-availability functionality.

A major change to this container is that the PostgreSQL process is now managed by Patroni. This allows a PostgreSQL cluster that is deployed by the PostgreSQL Operator to manage its own uptime and availability, to elect a new leader in the event of a downtime scenario, and to automatically heal after a failover event.

When creating your new clusters using version 4.3.0 of the Postgres Operator, the `pgo create cluster` command will automatically use the new `crunchy-postgres-ha` image if the image is unspecified. If you are creating a PostGIS enabled cluster, please be sure to use the updated image name, as with the command:

```
pgo create cluster mygiscluster --ccp-image=crunchy-postgres-gis-ha
```

{{% 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 4.3.0 code for the Postgres Operator available
- The latest 4.3.0 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.X 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=4.3.0
```

If you are pulling your images from the same registry as before this should be the only update to the existing 4.X environment variables.

Operator 4.0

If you are upgrading from Postgres Operator 4.0.1, 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 Kubernetes 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
export TLS_CA_TRUST=""
export ADD_OS_TRUSTSTORE=false
export NOAUTH_ROUTES=""

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

There is a new eventing feature in 4.3.0, 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
}
```

Operator 4.1

If you are upgrading from Postgres Operator 4.1.0 or 4.1.1, you will only need the following subset of the environment variables listed above:

```
export TLS_CA_TRUST=""
export ADD_OS_TRUSTSTORE=false
export NOAUTH_ROUTES=""
```

Finally source the updated bash file:

```
source ~/.bashrc
```

Step 4 Ensure you have checked out the latest 4.3.0 version of the source code and update the pgo.yaml file in `$PGOROOT/conf/postgres-op`

You will want to use the 4.3.0 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 4.3.0 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 4.3.0 by replacing the existing 4.X binary with the latest 4.3.0 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 4.3.0 and all users and roles have been recreated. Verify this by running:

```
pgo version
```

Step 10 Once the Operator is installed and functional, create a new 4.3.0 cluster with the same name and using the same major PostgreSQL version as was used previously. This will allow the new cluster to utilize the existing PVCs.

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

Step 11 To verify cluster status, run

```
    pgo test <clustername> -n <namespace>
```

Output should be similar to:

```
cluster : mycluster
        Services
                primary (10.106.70.238:5432): UP
        Instances
                primary (mycluster-7d49d98665-7zxzd): UP
```

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

It is also recommended to take full backups of each pgcluster once the upgrade is completed due to version differences between the old and new clusters.

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.